Journal of Statistical Software

November 2024, Volume 111, Issue 8. doi: 10.18637/jss.v111.i08

Interpreting Deep Neural Networks with the Package innsight

Niklas Koenen
Leibniz Institute for Prevention Research and Epidemiology – BIPS,
University of Bremen

Marvin N. Wright
Leibniz Institute for Prevention Research and Epidemiology – BIPS,
University of Bremen
University of Copenhagen

Abstract

The R package **innsight** offers a general toolbox for revealing variable-wise interpretations of deep neural networks' predictions with so-called feature attribution methods. Aside from the unified and user-friendly framework, the package stands out in three ways: It is generally the first R package implementing feature attribution methods for neural networks. Secondly, it operates independently of the deep learning library, allowing the interpretation of neural networks from any R package, including **keras**, **torch**, **neuralnet**, and even custom models. Despite its flexibility, **innsight** benefits internally from the **torch** package's fast and efficient array calculations, which builds on **LibTorch** – **PyTorch**'s C++ backend – without a Python dependency. Finally, it offers a variety of visualization tools for tabular, signal, image data, or a combination of these. Additionally, the plots can be rendered interactively using the **plotly** package.

Keywords: neural networks, feature attribution, interpretable machine learning, explainable artificial intelligence, XAI, IML, **torch**, **keras**, R.

1. Introduction

Throughout the past decade, neural networks have unleashed a tremendous surge of attention and infiltrated almost all conceivable domains of science, industry, and public life. Mainly, their increasing popularity is due to their natural ability to extract patterns and knowledge from vast amounts of structured raw data thanks to modern computing capacities and deliver outstanding performance (Krizhevsky, Sutskever, and Hinton 2017; LeCun, Bengio, and Hinton 2015; Silver *et al.* 2016; Bengio, Lecun, and Hinton 2021). However, the intelligently

learned decision-making process of a neural network remains inscrutable and hidden from the user due to its enormous complexity. Interpretations cannot be inferred as straightforward from this so-called *black box* as, for example, the coefficients of a linear model. As a consequence, the gain in predictive accuracy and model flexibility generally comes at the price of an increasingly opaque and intricate machine learning model, as was already noted by Gunning and Aha (2019). Nevertheless, it is precisely this question of interpretability – or, informally speaking: Why did a network make a certain prediction? – that is becoming more and more relevant for applications with high-stake decisions and possibly becoming a legal requirement, e.g., in autonomous systems (O'Sullivan *et al.* 2019), healthcare (Schneeberger, Stöger, and Holzinger 2020) or data processing in general (European Union 2016; Goodman and Flaxman 2017).

Arising from this question and need, several methods have been proposed to explain predictions of machine learning models in a supervised learning setting. These methods are mainly classified according to the criteria for which model class they are applicable and at which level they provide explanations: Regarding the first criterion, *model-agnostic* approaches analyze the association of input data and model predictions of arbitrary models. Contrary, *model-specific* methods additionally exploit internal structures to reveal insights, but their application is restricted to a specific group of models. Secondly, interpretability methods for machine learning models can be categorized into *local* and *global* in terms of the explanation level. Established local model-agnostic methods, such as Shapley additive explanations (SHAP, Lundberg and Lee 2017), individual conditional expectations (ICE, Goldstein, Kapelner, Bleich, and Pitkin 2015), and local interpretable model-agnostic explanations (LIME, Ribeiro, Singh, and Guestrin 2016), explain only individual or groups of instances from the dataset, e.g., a single picture for image classification or one patient in the context of medical disease prediction. In contrast, global approaches describe the entire model behavior independent of individual effects. Commonly applied methods from this category are permutation feature importance (Fisher, Rudin, and Dominici 2019), accumulated local effect (ALE) plots (Apley and Zhu 2020) and partial dependence plots (Friedman 2001; Greenwell, Boehmke, and McCarthy 2018). One way to describe this distinction is to look at the classical linear model with input variables $\boldsymbol{x} = (x_1, \ldots, x_p)^\top$ and prediction $\hat{y}$:

$$\hat{y} = \beta_0 + x_1\,\beta_1 + \ldots + x_p\,\beta_p.$$

In this case, the coefficients $\beta_1, \ldots, \beta_p$ describe the global effect of the input variables $x_1, \ldots, x_p$ taking their values independent of $\boldsymbol{x}$, i.e., they indicate how much the variables affect the prediction $\hat{y}$ in general. On the other hand, the local explanations $x_1\,\beta_1, \ldots, x_p\,\beta_p$ demonstrate how much each input variable contributes to or impacts the prediction for a chosen input instance $\boldsymbol{x}$, commonly leading to a variable-wise decomposition of $\hat{y}$ in additive effects.

Despite the quantity and universality of model-agnostic methods, they are barely applicable to modern deep neural networks mainly because of two reasons: Firstly, many of these model-agnostic approaches are based on repeated evaluation of perturbed or permuted input instances, and secondly, they scale poorly for a higher number of input variables. Since the forward pass of deep neural networks is computationally intensive and the inputs are often high-dimensional RGB images, applying model-agnostic approaches is time-consuming and challenging. Moreover, global variable-wise methods are generally not appropriate for image data since the importance of pixels is rather independent of the exact localization and depends more on the neighboring pixels varying in each individual image. Alternatively, global model-

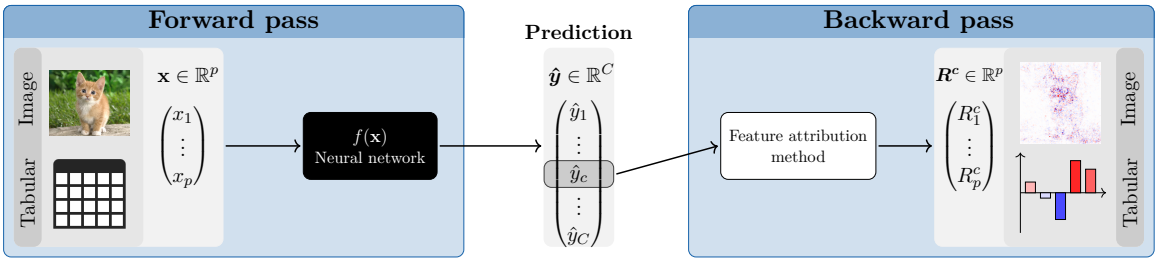


Figure 1: General procedure of feature attribution methods: First, an input instance $\boldsymbol{x}$ flows through the model f to obtain a prediction $\hat{\boldsymbol{y}}$. Then, the desired output node or class $\hat{y}_c$ to be explained is selected. Finally, the relevance R_i^c of the individual input variables i at the selected output c is calculated in a backward pass.

specific approaches such as feature visualizations (Olah, Mordvintsev, and Schubert 2017) or concept-based methods (Kim *et al.* 2018) can only be used with further optimization procedures or concept-labeled datasets. This gap of interpretability methods for neural networks is being filled by *feature attribution* methods, which leverage all model-internal components in addition to the input and output relation, requiring only the model and the input instance to be explained. Particularly, this group of local model-specific methods, which assign relevance scores to each input variable for one of the output nodes or classes, prevailed and has successfully been applied in many domains (Zuallaert, Godin, Kim, Soete, Saeys, and De Neve 2018; Anders, Montavon, Samek, and Müller 2019; Lauritsen *et al.* 2020). Furthermore, only a single regular forward pass and then a modified backward pass need to be performed for an explanation, i.e., they are generated without an optimization or estimation procedure. This two-step technique is illustrated in Figure 1: First, a prediction is produced in the standard forward pass, and the class to be explained is selected. In the subsequent method-specific backward pass, each input variable is assigned a relevance score to the chosen output class and can be visualized in a heat map or bar chart depending on the input type. An overview of the most popular feature attribution methods can be found in Section 2.

Several software packages have been developed in the last few years to make feature attribution methods widely accessible to users and to provide them with a unified and easy-to-use interface. The most popular packages are **innvestigate** (Alber *et al.* 2019) for the deep learning library **Keras** (Chollet *et al.* 2015), and **captum** (Kokhlikyan *et al.* 2020) and **zennit** (Anders, Neumann, Samek, Müller, and Lapuschkin 2021) for **PyTorch** (Paszke *et al.* 2019). In addition, the **shap** package (Lundberg and Lee 2017) implements many Shapley-value-based methods and can handle both **Keras** and **PyTorch** models. However, all these packages are exclusive for Python (Van Rossum *et al.* 2011) and mostly only support networks of specific deep learning libraries. Despite the existing R packages for model-agnostic interpretability methods, such as **iml** and **dalex** (Molnar, Casalicchio, and Bischl 2018; Biecek 2018), we want to make feature attribution methods easily accessible to the R community and therefore provide the software package **innsight**, which pursues the following goals:

- *First feature attribution R package*: **innsight** is the first R package that implements the most popular feature attribution methods for neural networks unified in a single user-friendly package.
- *Computationally efficient*: The powerful **torch** (Falbel and Luraschi 2024) package is

utilized internally for all calculations, which builds on **LibTorch**, the C++ (Stroustrup 2013) variant of **PyTorch** (Paszke *et al.* 2019), and does not rely on a Python dependency.

- *Deep-learning-library-agnostic*: The passed trained models are not limited to a specific deep learning library. The package supports models from the R packages **keras** (Allaire and Chollet 2024), **torch** and **neuralnet** (Günther and Fritsch 2010). However, under some constraints, an arbitrary model can be passed as a list to be fully flexible.

- *Visualization tools*: **innsight** offers several visualization methods for individual or summarized results regardless of whether it is tabular, signal, image data, or a combination of these. Additionally, interactive plots can be created based on the **plotly** package (Sievert 2020).

The **innsight** package is available from the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/package=innsight` or from our GitHub repository at `https://github.com/bips-hb/innsight/`.

The rest of the paper is structured as follows: First, we overview the most popular feature attribution methods for neural networks in Section 2. Then, in Section 3, we elaborate on the package's design, functionality, and capabilities. Next, the package is applied to a basic example on a penguin dataset and an advanced example for melanoma detection based on image and tabular data as input types. In the concluding Section 5, the obtained package's outputs are compared and validated with the already mentioned Python equivalents.

2. Methodology of feature attribution

Feature attribution methods for neural networks describe a group of local interpretation methods that assign to each input variable the contribution or impact to a chosen model output. For example, suppose an input instance $\boldsymbol{x} \in \mathbb{R}^p$ with $p \in \mathbb{N}$ variables is fed forward through a neural network $f : \mathbb{R}^p \to \mathbb{R}^C$ resulting in an output $f(\boldsymbol{x}) = \hat{\boldsymbol{y}} \in \mathbb{R}^C$ with $C \in \mathbb{N}$ classes or regression outputs. In this case, a feature attribution method assigns relevance scores $R_1^c, \dots, R_p^c$ to each of the input features $x_1, \dots, x_p$ of $\boldsymbol{x}$ on a chosen output class or node $\hat{y}_c$ of the prediction $\hat{\boldsymbol{y}}$ to be explained, as already described in Figure 1.

2.1. Gradient-based methods

Gradient-based methods are the fastest and most straightforward interpretation methods because they operate on the default techniques of the high-level deep learning libraries for computing gradients during the training loop. However, these methods – in a sense – calculate the derivatives of chosen output to the input variables instead of the derivatives of the loss value to the model parameters during gradient descent. Although the terminology of gradient-based methods can often be interpreted more broadly, we only consider techniques that use the default gradient methods. For example, Ancona, Ceolini, Öztireli, and Gross (2018) showed that variants of the *layer-wise relevance propagation (LRP)* and *deep learning important features (DeepLift)*, discussed later in Section 2.2 and 2.3, can approximately be considered gradient-based. Regardless, they are not mentioned in this section, since not all *LRP* and *DeepLift* variants can be considered gradient-based. Additionally, these variants require an overwriting of the standard gradients, i.e., they do not use the mathematical definition of

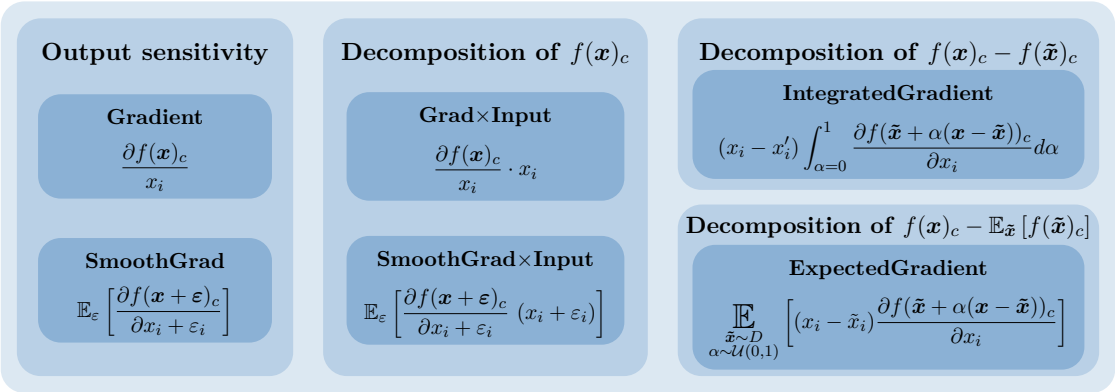


Figure 2: A summary of gradient-based feature attribution methods, including their mathematical representation. They are divided into blocks based on their underlying objectives. For example, in the case of feature-wise relevances R_i^c obtained from *Gradient* × *Input*, the goal is to achieve a sum that equals $f(\boldsymbol{x})_c$, i.e., $\sum_{i=1}^{p} R_i^c = f(\boldsymbol{x})_c$.

the gradient anymore. In the following, the most common gradient-based feature attribution methods are briefly explained, including their underlying objectives. For a more mathematical overview, see Figure 2.

One of the first and most intuitive methods for interpreting neural networks is the *Gradient* method introduced by Simonyan, Vedaldi, and Zisserman (2014), also known as *vanilla gradients* or *saliency maps*. This method computes the gradients of the selected output with respect to the input variables. Therefore, the resulting relevance values indicate prediction-sensitive variables that can be locally perturbed the least to change the outcome the most. Assuming that the model f behaves linearly overall, increasing x_i by one raises the output by the calculated gradient. In general, neural networks are highly nonlinear, which forces the gradients to have large fluctuations or abrupt changes. This phenomenon can introduce noise and potential misinterpretations in the *Gradient* method. A simple extension of this basic *Gradient* method to tackle this issue is the *smoothed gradients (SmoothGrad)* approach introduced by Smilkov, Thorat, Kim, Viégas, and Wattenberg (2017). They proposed computing the gradients of randomly Gaussian perturbed copies of x_i and determining the average gradient from that, instead of calculating only the gradient in x_i. As a result, locally very noisy gradients are smoothed out and the method provides the average behavior in a larger neighborhood of x_i. The estimation accuracy and size of the neighborhood can be adjusted with the hyperparameters n for the number of perturbations and σ^2 for the variance of the Gaussian noise. With the value of n, the estimation accuracy for the average gradient can be increased, but this goes hand in hand with a higher computational effort. The second parameter σ^2 is mostly specified indirectly via a noise level $\lambda \geq 0$ determining the proportion of the total range of the input domain that is covered by the standard deviation σ, i.e., $\lambda = \frac{\sigma}{x_{\max}-x_{\min}}$. Especially for images, this argument can be used to control the visual smoothness of the explanation.

A simple modification can change both previously discussed methods to the methods *Gradient* × *Input* and *SmoothGrad* × *Input*. The gradients are calculated as for the respective methods and then multiplied by the corresponding feature values. The *Gradient* × *Input* method was introduced by Shrikumar, Greenside, Shcherbina, and Kundaje (2017b) and relies on a well-

grounded mathematical background despite its simple idea: The basic concept is decomposing the output prediction $\hat{y}_c$ according to its relevance to each input variable x_i, i.e., into variable-wise additive effects

$$\hat{y}_c = f(\boldsymbol{x})_c = \sum_{i=1}^{p} R_i^c. \tag{1}$$

Mathematically, this method is based on the first-order Taylor decomposition. Assuming that a function $g : \mathbb{R}^p \to \mathbb{R}$ is continuously differentiable in $\boldsymbol{x} \in \mathbb{R}^p$, a remainder term $\varepsilon(g, \boldsymbol{z}, \boldsymbol{x}) : \mathbb{R}^p \to \mathbb{R}$ with $\lim_{\boldsymbol{z}\to\boldsymbol{x}} \varepsilon(g, \boldsymbol{z}, \boldsymbol{x}) = 0$ exists such that

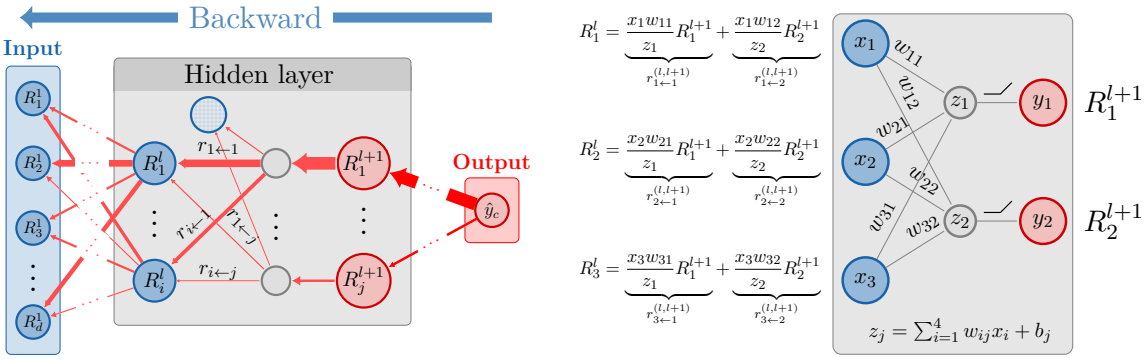
$$\begin{aligned} g(\boldsymbol{z}) &= g(\boldsymbol{x}) + \nabla_{\boldsymbol{x}} g(\boldsymbol{x}) \cdot (\boldsymbol{z} - \boldsymbol{x})^\top + \varepsilon(g, \boldsymbol{z}, \boldsymbol{x}) \\ &= g(\boldsymbol{x}) + \sum_{i=1}^{p} \frac{\partial\, g(\boldsymbol{x})}{\partial\, x_i}(z_i - x_i) + \varepsilon(g, \boldsymbol{z}, \boldsymbol{x}), \quad \boldsymbol{z} \in \mathbb{R}^p. \end{aligned}$$

The first-order Taylor formula thus describes a linear approximation of the function g at the point $\boldsymbol{x}$ since only the first derivatives are considered. Consequently, a highly nonlinear and continuous function g is well approximated only in a small neighborhood around $\boldsymbol{x}$. For larger distances from $\boldsymbol{x}$, sufficient small values of the residual term are not guaranteed anymore. The *Gradient* $\times$ *Input* method considers the data point $\boldsymbol{x}$ and sets $\boldsymbol{z} = \boldsymbol{0}$. In addition, the residual term $\varepsilon(f_c, \boldsymbol{0}, \boldsymbol{x})$ and the summand $f(\boldsymbol{0})_c$ are ignored. Analogously, this multiplication can be applied to all gradients in the summation of the *SmoothGrad* method in order to compensate for local fluctuations.

Even though the multiplication of gradients by the inputs provides an approximate decomposition of $f(\boldsymbol{x})_c - f(\boldsymbol{0})_c$, this approach only captures the feature-wise effects of $\boldsymbol{x}$ concerning a baseline of $\boldsymbol{0}$. However, this value does not necessarily reflect a prediction-neutral reference value and can be challenging to interpret or even lie outside the data distribution. Sundararajan, Taly, and Yan (2017) proposed a method called *IntegratedGradient* as a way to find a decomposition of $f(\boldsymbol{x})_c - f(\tilde{\boldsymbol{x}})_c$ into feature-wise effects for an arbitrary reference value $\tilde{\boldsymbol{x}}$, using integration over the *Gradient* $\times$ *Input* values along an integration path. In practice, the integral is approximated by the sum of gradients multiplied by the inputs along an interpolated path from $\boldsymbol{x}$ to $\tilde{\boldsymbol{x}}$. Nevertheless, choosing the reference value $\tilde{\boldsymbol{x}}$ remains a challenging task and ideally requires domain-specific knowledge. The *ExpectedGradient* method (Lundberg and Lee 2017; Erion, Janizek, Sturmfels, Lundberg, and Lee 2021) addresses this issue by estimating the mean value of the *IntegratedGradient* method through Monte Carlo integration, considering the whole distribution of reference values instead of a single baseline. In this sense, the method finds a feature-wise decomposition of $f(\boldsymbol{x})_c - \mathsf{E}_{\tilde{\boldsymbol{x}}}[f(\tilde{\boldsymbol{x}})_c]$ and, thus, calculates approximately Shapley values.

2.2. Layer-wise relevance propagation (LRP)

The *layer-wise relevance propagation (LRP)* method was introduced by Bach, Binder, Montavon, Klauschen, Müller, and Samek (2015) and has a similar goal as the *Gradient* $\times$ *Input* approach explained in the previous section: decomposing the output into variable-wise relevances conforming to Equation 1. The distinguishing aspect is that the prediction $\hat{y}_c$ is redistributed layer by layer from the output node back to the inputs according to the layer's weights and intermediate values. The entire procedure is accomplished by rule-based relevance messages defining how to redistribute the upper-layer relevance to the lower layer. A



(a) Backward pass of the *LRP* method.

(b) Calculation of the relevance of layer l based on the relevances of the upper layer $l+1$.

Figure 3: (a) illustrates the layer-by-layer backpropagation of relevances R_i^l from the prediction score to the input variables through the use of relevance messages $r_{i\leftarrow j}$. For a hidden layer, (b) demonstrates how the relevance of the lower layer l results from summing all incoming relevance messages.

high-level overview of this method applied to a neural network with L layers can be seen in Figure 3a. The method mainly consists of the following iterative steps: As the starting point, the relevance for the considered output node R_1^L is set to the respective prediction score $\hat{y}_c$. Subsequently, the relevance of the lower layer's node R_i^{L-1} is calculated using the sum of all incoming relevance messages. A relevance message describes the proportion of the upper-layer relevance R_1^L that is sent to a node i in the lower layer. This process is repeated layer by layer backwards, as shown in Figure 3a, until the input layer is reached and relevances are obtained for each input feature. More precisely, for a hidden layer l, the relevance message $r_{i\leftarrow j}^{(l,l+1)}$ from node j in the upper layer $l+1$ to node i in the preceding layer defines the proportion of the relevance from R_j^{l+1} attributed to the node i in the lower layer. Since the relevance messages are based solely on the contribution from a single upper-layer node, the overall relevance of the node i is obtained by summing up all incoming relevance messages (see Figure 3b), i.e.,

$$R_i^l = \sum_j r_{i\leftarrow j}^{(l,l+1)}.$$

Since the publication of the *LRP* method, various variations of relevance messages flowing from the upper layer to the lower-layer node have emerged. However, the fundamental rule on which all other variations of relevance messages are more or less based is the *simple rule* (also known as *LRP-0*). The relevances are redistributed to the lower layers according to the ratio between local and global pre-activation. Despite being a rule-based approach, if the neural network only includes ReLU activations, this rule makes the method *LRP* equivalent to *Gradient* $\times$ *Input* presented in Section 2.1 (Ancona *et al.* 2018). Let $\boldsymbol{x}$ be the input, $\boldsymbol{w}$ the weight matrix and $\boldsymbol{b}$ the bias vector of layer l, and R_j^{l+1} the upper-layer relevance; then $x_i\, w_{ij}$ is the local and $z_j = b_j + \sum_k x_k\, w_{kj}$ the global pre-activation defining the simple rule as (also used in Figure 3b)

$$r_{i\leftarrow j}^{(l,\,l+1)} = \frac{x_i\, w_{ij}}{z_j}\, R_j^{l+1}.$$

Many other rules for relevance messages are built upon this principle. The well-known variations *ε-rule* and *α-β-rule* of the simple rule and their advantages and disadvantages are explained in the Appendix A.1. Additionally, a brief summary of other rules discussed in the literature is provided there.

2.3. Deep learning important features (DeepLift)

One method that, to some extent, echoes the idea of *LRP* is the so-called *deep learning important features (DeepLift)* method introduced by Shrikumar, Greenside, and Kundaje (2017a). It behaves similarly to *LRP* in a layer-by-layer backpropagation fashion from a selected output node back to the input variables, considering the simple rule. However, it incorporates a reference value $\tilde{\boldsymbol{x}}$ to compare the relevances with each other, analogously to the *IntegratedGradient* method discussed in Section 2.1. Hence, the relevances of *DeepLift* represent the relative effect of the outputs of the instance to be explained $f(\boldsymbol{x})_c$ and the output of the reference value $f(\tilde{\boldsymbol{x}})_c$. By taking the difference, the bias term is eliminated in the relevance messages, preventing the relevance absorption and leading to an exact variable-wise decomposition of the difference-from-reference output $\Delta\hat{y}_c = f(\boldsymbol{x})_c - f(\tilde{\boldsymbol{x}})_c$, i.e.,

$$\Delta\hat{y}_c = f(\boldsymbol{x})_c - f(\tilde{\boldsymbol{x}})_c = \sum_{i=1}^{p} R_i^c.$$

Similar to the relevance messages for *LRP*, *DeepLift* defines so-called *multipliers* for each layer or part of a layer. Based on these multipliers, the contribution of an arbitrary input (or intermediate) variable to the difference-from-reference output can be obtained by multiplying it by the corresponding difference-from-reference value. For an arbitrary layer with the layer's input $\boldsymbol{x}$, reference input $\tilde{\boldsymbol{x}}$, and multiplier $m_{\Delta\boldsymbol{x}\Delta\hat{y}_c}$, this means:

$$\sum_i m_{\Delta x_i \Delta\hat{y}_c} \left(x_i - \tilde{x}_i\right) = m_{\Delta\boldsymbol{x}\Delta\hat{y}_c} \cdot (\Delta\boldsymbol{x})^\top = \Delta\hat{y}_c. \tag{2}$$

The multipliers fulfill a chain rule allowing the computation of the multiplier for the preceding layer given the already calculated one $m_{\Delta\boldsymbol{t},\Delta\hat{y}_c}$, i.e.,

$$m_{\Delta x_i \Delta\hat{y}_c} = \sum_j m_{\Delta x_i \Delta t_j}\, m_{\Delta t_j \Delta\hat{y}_c}. \tag{3}$$

In other words, the chain rule justifies defining the multipliers for each layer or part of a layer separately before combining them with the upper-layer multipliers. For linear components, such as the matrix multiplication in dense or convolutional layers, the weights are used as the multipliers, i.e., $m_{\Delta x_i \Delta z_j} = w_{ij}$. For nonlinear components, like, e.g., all point-wise activations such as ReLU, tanh, or sigmoid, Shrikumar *et al.* (2017a) propose the *Rescale* and *RevealCancel* rule. While the *Rescale* rule uses the ratio of the layer's difference-from-reference output and difference-from-reference pre-activation as the multiplier, the *RevealCancel* rule is designed to propagate meaningful relevances even for saturated activations and discontinuous gradients through the layers' activation part. For a more detailed explanation, we refer to the Appendix A.2. These rules, along with the chain rule (Equations 2 and 3), enable the successive computation of the input variables' contributions R_i^c to the difference-from-reference output $\Delta\hat{y}_c$ in a single backward pass.

The reference value is the only crucial hyperparameter for the *DeepLift* method, apart from the rule for non-linearities. This choice depends significantly on the application and usually requires proficient domain-specific knowledge. Nevertheless, the authors suggest asking oneself the question of what one wants to measure an effect against. For example, taking the background color or blurred versions of the original picture as the reference values for images are reasonable choices. In many cases, zeros as a baseline are also used. Ancona *et al.* (2018) showed that using the *Rescale* rule with activations crossing the origin (i.e., $\sigma(0) = 0$) and a zero baseline as reference value $\tilde{\boldsymbol{x}}$ coincides with the *Gradient* $\times$ *Input* method discussed in Section 2.1 and with *LRP* with the simple rule. Similar to how the *ExpectedGradient* method generalizes *IntegratedGradient* considering the distribution of baseline values instead of a single reference value, the *DeepSHAP* (Lundberg and Lee 2017) method extends the *DeepLift* technique. It calculates the average *DeepLift* value across various baseline values, thereby achieving a decomposition of $f(\boldsymbol{x}) - \mathbb{E}_{\tilde{\boldsymbol{x}}}[f(\tilde{\boldsymbol{x}})]$ into feature-wise effects and, thus, gives approximately Shapley values.

2.4. Connection weights

One of the earliest methods specifically designed for neural networks is the *connection weights (CW)* method invented by Olden, Joy, and Death (2004), resulting in a global relevance score for each input variable. The basic idea of this approach is to multiply all path weights for each possible connection between an input variable x_i and the output node or class $\hat{y}_c$ and then calculate the sum of all of them. However, this method ignores all bias vectors and all activation functions during calculation. Analogously to the previous methods, CW can also be defined layer by layer, deriving the relevance for layer l from the upper layer as follows:

$$R_i^l = \sum_j w_{ij} R_j^{l+1}.$$

Since only the model weights are used, this method is independent of input data and, thus, a global interpretation method. Inspired by the method *Gradient* $\times$ *Input*, it can also be extended into a local method by taking the point-wise product of the global CW method and the input data.

2.5. Choice of the method

Overall, the choice of methods for a user remains an open research question, but there are several recommendations that can be derived from the complexity or stated goals of the methods. Firstly, consider the computational efficiency of the method. Standard gradient methods such as *Gradient* and *Gradient* $\times$ *Input*, as well as *LRP*, are very fast but can be very noisy as they only examine a very local behavior. On the other hand, methods like *SmoothGrad*, *IntegratedGradient*, and *ExpectedGradient* often require a high number of forward passes to deliver accurate results, but also consider the local neighborhood or baseline values. Furthermore, it has been demonstrated that many methods (such as *Gradient* $\times$ *Input*, *LRP*, *IntegratedGradient*, *DeepLift*) are not invariant to constant shifts of the inputs (Kindermans *et al.* 2019; Haug, Zürn, El-Jiz, and Kasneci 2022; Nielsen, Dera, Rasool, Ramachandran, and Bouaynaya 2022). This is because the Taylor approximation is only accurate around zero (or the reference value), and the choice of the reference value has a crucial impact. One solution to this is provided by Shapley-based methods like *ExpectedGradient* and *DeepSHAP*. However, these methods require a highly informative reference dataset in addition to the instance

to be explained, and they are noticeably slower. These considerations highlight the trade-offs involved in selecting a feature attribution method for a given task, and it is crucial for users to weigh the speed, accuracy, and invariance characteristics based on the specific requirements of their application.

3. Functionality and usage

The R package **innsight** combines all the methods discussed in the previous section in a user-friendly structure and a unified step-based workflow from the trained model to the visualization of the relevances of a feature attribution method. For efficient high-dimensional array calculations, the package utilizes the R package **torch** (Falbel and Luraschi 2024), which builds on **LibTorch** (the C++ variant of **PyTorch** (Paszke *et al.* 2019)), and consequently runs without a Python dependency (see Figure 4). The following three steps yield the requested results, regardless of the class of the passed model or the chosen feature attribution method:

- Step 1: Convert the model.
- Step 2: Apply selected method.
- Step 3: Get or visualize results.

Internally, a class structure is being built using **R6** classes based on the equally named package **R6** (Chang 2021). This type of object-oriented programming class is used because **torch** also relies on it, and it simplifies inheritance and argument passing compared to conventional S3 and S4 classes. Apart from the utilized packages for the internal workflows, calculations, and visualizations discussed in the following sections, the packages **checkmate** (Lang 2017) and **cli** (Csárdi 2024) are generally used for all argument verifications, internal checks, and terminal outputs of messages, warnings, and errors.

For illustration and better comprehension, the three steps will be exemplified from a user's perspective using the *bike sharing dataset* (Fanaee-T 2013). The internal mechanisms and more detailed descriptions are provided in the subsequent sections. This regression dataset contains information about the number of bicycles rented on a given day, along with various features such as weather conditions, holidays, and temperature. When using the **innsight** package, object-oriented **R6** objects are internally created at each step, but their initialization is simplified through function calls providing a conventional R usage without prior knowledge about **R6** classes. This facilitates easy application for R users of the package.

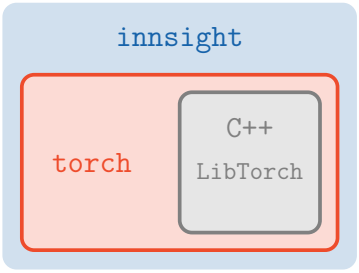


Figure 4: **innsight** utilizes package **torch**, which builds directly on the C++ library **LibTorch** without a Python dependency.

In the following code snippet, the dataset is loaded and restricted to a few variables. The outcome in this regression dataset is `"cnt"`, indicating the total number of bicycles rented on the given day. The model is trained on this dataset using the **neuralnet** package (Günther and Fritsch 2010) containing one hidden layer with 64 neurons. In addition, the outcome variable `"cnt"` is scaled to bikes per 10 000.

```
R> library("neuralnet")
R> set.seed(42)
R> bike <- read.csv("additional_files/bike_sharing/day.csv")
R> bike <- bike[, c("cnt", "holiday", "workingday", "temp", "hum",
+    "windspeed")]
R> bike$cnt <- bike$cnt / 10000
R> bike <- as.matrix(bike)
R> model <- neuralnet(cnt ~ ., data = bike, hidden = c(64),
+    linear.output = TRUE)
```

To enable a deep-learning-library-agnostic implementation, the given model `model` is analyzed in the first step, and internally, a replica based on the **torch** package is reconstructed. However, for the user, this process is abstracted through the `convert()` function, allowing the adjustment of the used variable and outcome names, e.g., with the argument `output_names`:

```
R> library("innsight")
R> conv <- convert(model, output_names = c("Number of rented bikes/10,000"))
```

In the second step, the user's method of choice can be applied to the provided data (argument `data`). Internally, an **R6** class for the respective method is initialized but hidden from the user again. In this case, the *DeepSHAP* method (`run_deepshap()`) is run on the first 20 instances with the entire dataset as reference values. For computational reasons, the internal calculation uses 100 samples from this dataset, as this is the default value of `limit_ref`. In addition, the data must always be passed as input data only, which is why the outcome variable `"cnt"` is removed in the following code:

```
R> res_deepshap <- run_deepshap(conv, bike[1:20, -1], data_ref = bike[, -1])
```

In the final step, results can be extracted, for instance, using the `get_result()` function, or visualized using `plot()` or `plot_global()`/`boxplot()`. The `boxplot()` method is an alias for `plot_global()` in case of tabular and signal data, as boxplots are created for these data types. It is noted that the variable `"hum"` for humidity is scaled between 0 and 1, and `"tmp"` and `"windspeed"` are divided by the respective maximal value, i.e., 100 and 67. The visualization relies on the **ggplot2** package and can be customized accordingly (the plots are shown in Figure 5a):

```
R> library("ggplot2")
R> head(get_result(res_deepshap))
R> plot(res_deepshap)
R> boxplot(res_deepshap, ref_data_idx = 1) +
+    theme(text = element_text(face = "bold"))
```

```
, , Number of rented bikes/10,000

          holiday   workingday       temp         hum     windspeed
[1,]  0.004659358  0.050143935 -0.1218946 -0.08540524  0.0109126559
[2,]  0.007893519  0.064216673 -0.1305959 -0.01469872 -0.0320372544
[3,] -0.009826845  0.007002956 -0.2871930  0.09675989 -0.0359177366
[4,] -0.011643781  0.028260766 -0.2517902  0.04398110  0.0213681515
[5,] -0.009875727  0.015752951 -0.2339799  0.09297864 -0.0008396581
[6,]  0.002854239 -0.004088928 -0.2964459  0.05857125  0.0088869939
```

For example, in the box plots (see Figure 5a bottom), bold font is used for the labels, which is achieved through the `theme(text = element_text(face = "bold"))` feature of **ggplot2**. By default, the orientation of relevances is concealed by absolute values, as global attention is usually focused on the strength of the effects. Consequently, it can be observed that the model considers temperature as the most crucial feature for its predictions. Furthermore, the argument `ref_data_idx` is employed to highlight the relevances of the first instance of the dataset as a reference value with red lines. This instance and its local effects are more precisely visualized in the upper illustration of Figure 5a. It also shows that temperature is an influential factor for the model since it predicts a relatively low count of 2 994 bicycles (compared to the average of 4 504 bicycles). This aligns with the data, as on that particular day, the temperature is 8.18°C (normalized value 0.344167), while the dataset's average temperature is 15.28°C (normalized value 0.49538). Additionally, the plot includes a small box displaying the model's prediction of the instance (0.2994), the sum of calculated relevances (−0.1416), and the decomposition target of the method (−0.1416). For example, the DeepSHAP method is based on the decomposition of the difference between the prediction and the average prediction in the baseline dataset (i.e., $f(\boldsymbol{x})_c - \mathbb{E}_{\tilde{\boldsymbol{x}}}[f(\tilde{\boldsymbol{x}})_c]$), which is for the first instance −0.1416, i.e., 1 416 bikes below the average.

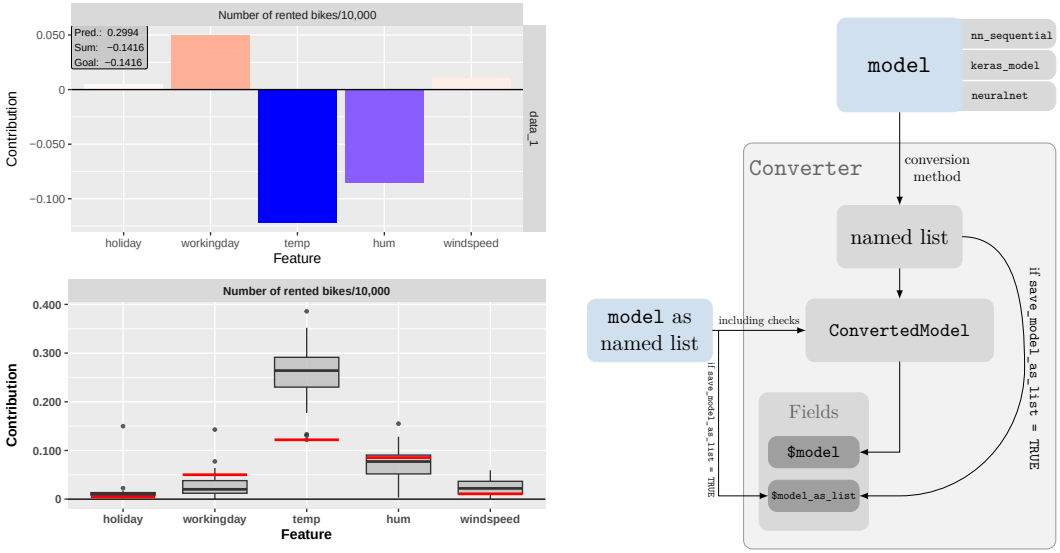
3.1. Step 1 – Convert the model

The key step that turns the **innsight** package into a deep-learning-library-agnostic approach and unlocks the provided **torch** toolbox to all methods is this first step, which essentially analyzes a passed model and creates a **torch**-based replication. For the user, however, the internal processes remain hidden, and the entire conversion step is accomplished by creating a new instance of the class '`Converter`':

```
Converter$new(model,
  input_dim = NULL, input_names = NULL, output_names = NULL,
  dtype = "float", save_model_as_list = FALSE)
```

As previously mentioned, this object is implemented using the object-oriented **R6** class from the **R6** package. To overcome prior knowledge of **R6** classes, the shortcut function `convert()` is implemented, which simply forwards all arguments to the '`Converter`' object's initialization function from above.

The only necessary argument is the passed `model`, which can be either a '`nn_sequential`' object from **torch**, a '`keras_model`' or '`keras_model_sequential`' object from **keras** (Chollet *et al.* 2015), a '`neuralnet`' object from **neuralnet** (Günther and Fritsch 2010), or a named



(a) Visual results of `plot()` and `boxplot()` from the bike sharing example.

(b) Internal conversion process of a '`Converter`' object.

Figure 5: (a) displays the visualizations of the `plot()` and `boxplot()` functions applied to the *DeepSHAP* method on the bike sharing dataset. In (b), the internal conversion process of creating a new '`Converter`' object is shown, which is identical to calling the shortcut function `convert()`.

list in a specific style. The other arguments `input_dim`, `input_names` and `output_names` are optional – except `input_dim` in combination with **torch** models – and are used for internal validation of the copied model or to assign labels to the input and output nodes used for the visualizations in Step 3 (see Section 3.3). This is already demonstrated in the previous example with the bike sharing dataset, where the output name `"Numbers of rented bikes/10,000"` is passed. In addition, the arguments `dtype` and `save_model_as_list` specify the calculations' numerical precision and save the entire model as a named list in the instance's field `model_as_list`, which is created as an intermediate step during the conversion process and are explained in more detail in the next paragraph and Figure 5b.

To be as flexible as possible and to interpret neural networks from almost any R package, a conversion method is implemented for each of the model classes mentioned above of the packages **torch**, **keras** and **neuralnet**, summarizing all decisive components and layers of the passed model in an ordered and unified way into a list. Then, a **torch**-based model '`ConvertedModel`' (i.e., a subclass of '`nn_module`') is created internally from this list. In addition, the rule-based interpretation methods described in Section 2 are pre-implemented for each valid layer type, which can be called layer by layer in the following Step 3. Since the creation of the converted model is consequently independent of the class of the given model, the conversion call can be bypassed by directly passing the desired model as a list. Hence, custom wrappers for other packages' models can be written, allowing an interpretation of models not being created by the packages **torch**, **keras**, or **neuralnet**. An overview of the individual steps that are performed internally when initializing a new instance of the '`Converter`' class is summarized in Figure 5b. In addition to the fields shown in Figure 5b, there are also fields containing the labels (`$input_names`, `$output_names`) and shapes (`$input_dim`, `$output_dim`) of the input

and output layers in a unified list structure. What kind of list structure is required for a model passed as a list, which layers are generally accepted and even more is explained in detail in Appendix B or in the vignette "In-depth explanation" (see `vignette("detailed_overview", package = "innsight")` and is only referred to at this point.

3.2. Step 2 – Apply selected method

As previously mentioned, the **innsight** package provides the most popular feature attribution techniques in a unified framework. For package users, simple functions are provided to apply the respective method to the data, as demonstrated at the beginning of this section with the bike sharing dataset using the `run_deepshap()` function. However, before delving into the individual user-facing functions, their internal class-related origin is explained, since the fundamental structure of the methods remains consistent. Internally, the unification is achieved by the **R6** super class '`InterpretingMethod`', from which all methods intended for users inherit and only add method-specific arguments to those of the super class. The rudimentary call of initializing a new method object looks like this:

```
InterpretingMethod$new(converter, data,
  channels_first = TRUE, output_idx = NULL, output_label = NULL,
  ignore_last_act = TRUE, verbose = interactive(), dtype = "float")
```

The key arguments for every method are the '`Converter`' object from the first step (see Section 3.1), containing the **torch**-converted model, and the `data` to be interpreted. The data can be passed in any format as long as the R base method `as.array()` can convert it into an array, and it matches the expected input dimension of the model. In addition, it is common for image or signal data to place either the channel axis directly after the batch axis or at the last position. However, this placement can generally not be extracted unambiguously from the data, which is why the `channels_first` argument specifies where the channel axis is located, allowing the use of both formats. The remaining arguments `output_idx`/`output_label`, `ignore_last_act`, `verbose` and `dtype` set which output nodes or labels are to be explained, whether the last activation function is ignored, whether a progress bar is displayed, or change the numerical precision for the calculations.

The feature attribution techniques designed for the package user's regular use cases and applications are inheritors of the super class '`InterpretingMethod`' and extend it by method-specific arguments. Since each method is implemented as an **R6** class, its application involves initializing a new class object through the `$new()` call. Therefore, helper functions are implemented, such as `run_deepshap(...)` in the example from the section's beginning, serving as a more user-friendly alternative to `DeepSHAP$new(...)`. For clarity, the subsequent presentation of the methods from Section 2 focuses solely on these shortcut functions, although it is noted which **R6** class they initialize:

- The methods *Gradient* and *Gradient* × *Input* are implemented as the **R6** class '`Gradient`', which has `times_input` as the only additional argument apart from the inherited ones. This argument switches between the usual gradients (`times_input = FALSE`) and the gradients multiplied by the corresponding inputs (`times_input = TRUE`):

  ```
  run_grad(converter, data, times_input = FALSE, ...)
  ```

- Similarly, the methods *SmoothGrad* and *SmoothGrad* × *Input* are realized in the **R6** class '`SmoothGrad`' containing the arguments `n` for the number of perturbations and `noise_level` for the noise scale in addition to the `times_input` argument:

```
run_smoothgrad(converter, data, times_input = FALSE, n = 50,
  noise_level = 0.1, ...)
```

- The method *IntegratedGradient* is implemented in the **R6** class '`IntegratedGradient`'. The method's baseline value can be specified using the argument `x_ref` defaulting to a zero baseline, and the number of discretization points for the integral approximation can be set with `n`:

```
run_intgrad(converter, data, x_ref = NULL, n = 50, ...)
```

- Similarly, the *ExpectedGradient* method can be applied by initializing an object of the class '`ExpectedGradient`'. The reference dataset can be passed with the `data_ref` argument defaulting to a zero baseline, and the number of samples with `n`:

```
run_expgrad(convert, data, data_ref = NULL, n = 50, ...)
```

- The *LRP* method, including the simple rule (`"simple"`), ε-rule (`"epsilon"`), α-β-rule (`"alpha_beta"`), and a composition of these rules, is implemented in the **R6** class '`LRP`'. The rule and its corresponding parameter (if available) are set with the arguments `rule_name` and `rule_param`. The default value and meaning of `rule_param` depends on the selected rule, more precisely, for `"epsilon"` the rule's ε value is set to 0.01 and for `"alpha_beta"` a value of $\alpha = 0.5$ (i.e., $\beta = 1 - \alpha$) is used. For both arguments, named lists can also be passed to assign a rule or parameter to each layer type separately. Since many zeros are produced in a maximum pooling layer during the backward pass due to the selection of the maximum value in the pooling kernel, the argument `winner_takes_all` can be used to treat a maximum as an average pooling layer in the backward pass instead.

```
run_lrp(converter, data, rule_name = "simple", rule_param = NULL,
  winner_takes_all = TRUE, ...)
```

- Analogously, the method *DeepLift* is realized in the **R6** class '`DeepLift`' including the argument `rule_name` for selecting the *Rescale* (`"rescale"`) or *RevealCancel* (`"reveal_cancel"`) rule for non-linearities. The reference value is set with `x_ref` defaulting to a baseline of zeros. *DeepLift* can also run into problems in maximum pooling layers since the maximum values in the pooling kernel from the normal and reference input generally do not coincide. Hence, with the `winner_takes_all` argument, this layer type can be treated as an average pooling layer in a backward pass.

```
run_deeplift(converter, data, rule_name = "rescale", x_ref = NULL,
  winner_takes_all = TRUE, ...)
```

- In the same way, the *DeepSHAP* method is implemented as an **R6** class named '`DeepSHAP`'. Instead of a single reference value, the entire reference dataset is passed with the `data_ref` argument. For computational reasons, by default, a maximum of 100 baseline values is considered for calculation, which can be adjusted using the `limit_ref` argument:

```
run_deepshap(converter, data, data_ref = NULL, limit_ref = 100, ...)
```

- The last method provided by **innsight** is the *connection weights (CW)* method realized in the **R6** class '`ConnectionWeights`'. The argument `times_input` specifies whether the global result of the CW method is calculated or whether it is additionally multiplied by the inputs to obtain local instance-wise explanations. A notable aspect, in this case, is that the `data` argument is not needed for the global variant, but it is required for the local one.

  ```
  run_cw(converter, times_input = FALSE, ...) # global
  run_cw(converter, data, times_input = TRUE, ...) # local
  ```

Although **innsight** primarily focuses on feature attribution methods specifically designed for neural networks, it also includes two well-known model-agnostic approaches, *local interpretable model-agnostic explanation (LIME)* (Ribeiro *et al.* 2016) and *Shapley values* (Lundberg and Lee 2017). LIME locally fits an intrinsic surrogate model (e.g., a generalized linear model) on the original model prediction to explain individual instances, while the game-theoretical Shapley values attribute the contributions of each feature by considering all possible feature combinations and their impact on the model output. Internally, they utilize the suggested packages **lime** (Hvitfeldt, Pedersen, and Benesty 2022) and **fastshap** (Greenwell 2024), and are incorporated into the class structure based on the **R6** class '`InterpretingMethod`'. They are realized in the **R6** classes '`LIME`' and '`SHAP`' and can be applied as follows:

```
run_lime(converter, data, data_ref, pred_fun = NULL, ...)
run_shap(converter, data, data_ref, pred_fun = NULL, ...)
```

Since these methods are model-agnostic, any other predictive model can be passed instead of a '`Converter`' object in the argument `converter`. However, for this, the prediction function `pred_fun` must be specified so that **innsight** knows how to make predictions. This function is already pre-implemented for '`Converter`' objects and for models from the packages **neuralnet**, **keras** and **torch**. Additionally, both methods require a reference dataset, which is passed with `data_ref`. Inheriting from '`InterpretingMethod`' are only the arguments `channels_first`, `output_idx`, and `output_label`. Similarly to the '`Converter`' class, `input_dim`, `input_names`, and `output_names` can be passed. All other arguments are forwarded to the corresponding methods `lime::explain()` or `fastshap::explain()`, which are called internally.

3.3. Step 3 – Get and visualize the results

After creating an object of a selected method, in the third step the results can be extracted or, if required, presented in a descriptive and visual way. For this purpose, the **innsight** package provides three generic methods `get_result()`, `plot()` and `plot_global()` that either return the results as an R object (such as `array`, `torch_tensor` or `data.frame`) or create visualizations for individual instances or aggregated results over the whole passed dataset. All three generic functions call the respective class methods in the '`InterpretingMethod`' super class, which are inherited by all the interpreting methods from the second step by design. For instance, in the bike sharing example from the section's beginning, the plot is generated using `plot(res_deepshap)` instead of `res_deepshap$plot()`.

Generic function `get_result()`

The function `get_result()` can be used to obtain the results in various forms, whatever is favored according to the user's subsequent workflow or application. This method has only the argument `type` (besides the method object), which determines the representation of the returned results. By default (`type = "array"`), the result is returned as an R base `array`, including the input and output names in the corresponding dimensions specified in the first step in the '`Converter`' object (see Section 3.1). The shape of the array is composed of the input shape including the batch size and the number of computed output nodes, i.e., for a tabular input with ten instances and four input variables, the shape is $10 \times 4 \times 3$ if the method was applied to three output nodes in Step 2. In the example with the bike sharing dataset, an array of size $20 \times 5 \times 1$ was returned using `get_result(res_deepshap)` which also includes the class label `"Number of rented bikes/10,000"`. This is because 20 instances were explained, and the model has 5 features and one output node. In the same way, `type = "torch_tensor"` returns a '`torch_tensor`' object having the same shape as the array, but without dimension labels. However, both variants can also return a list or list of lists with the related results as an `array` or '`torch_tensor`' for models with multiple input or output layers. The third and last format of the results is an R base `data.frame` obtained with `type = "data.frame"`. Included are columns for the input instance (`"data"`), the input and output layer of the model (`"model_input"` and `"model_output"`), the input variable (`"feature"`) – possibly also a second one for images (`"feature_2"`) and the channel for signal and image data (`"channel"`) – the output node or class (`"output_node"`), and the relevance (`"value"`) for the corresponding values. Moreover, the generated `data.frame` contains variables that show the prediction of the instance or already aggregated relevances for the respective output node and instance. The column `decomp_goal` indicates the decomposition goal of the method aimed by the aggregated relevances explained in Section 2, e.g., $f(\boldsymbol{x})_c - \mathbb{E}_{\tilde{\boldsymbol{x}}}[f(\tilde{\boldsymbol{x}})_c]$ for the *DeepSHAP* method.

Generic function `plot()`

The generic function `plot()` visualizes individual instances of the result of the method applied before based on the graphic package **ggplot2** (Wickham 2016) or the package **plotly** (Sievert 2020) for interactive graphics if the corresponding argument `as_plotly` is set. The call for a method's object `method` is executed as follows:

```
plot(method, data_idx = 1, output_idx = NULL, output_label = NULL,
  aggr_channels = "sum", as_plotly = FALSE, same_scale = FALSE,
  show_preds = TRUE)
```

The key arguments are `data_idx` and `output_idx`/`output_label`, which specify the indices for the dataset instances and the indices/labels for the desired output nodes or classes whose result is to be visualized. By default, the first data instance and the first computed output node are used. In the argument `output_idx`, no arbitrary indices can be passed, but only those for which the results were calculated previously in the second step. The same applies to `output_label`, which must additionally be a subset of the output names `output_names` in the '`Converter`' object. The further argument `aggr_channels` can be used to define how the channels are aggregated for image and signal data. There are various options for choosing this aggregation function, which may significantly influence the quality of the explanation (Arras,

Osman, and Samek 2022). Nevertheless, for **innsight**, the default is to compute the sum over the channels, which aligns with the additivity axiom for Shapley values to accurately reflect the collective group effect (Strumbelj and Kononenko 2010). Since visualization depends on the data type, it is internally distinguished between tabular/signal data and image data; accordingly, a bar chart or a raster chart is created, as shown in Figure 6 on the left. The relevances in the bars or the pixels are also scaled by color, facilitating a visual comparison; red means positive, blue negative, and white the absence of relevance. Since, in general, the scales vary significantly for the selected output class or data instance, the plots are scaled separately for each value in `output_idx`/`output_label` and `data_idx`. When several input layers are to be visualized, the remaining argument `same_scale` can be used to select whether the individual input layers are also scaled separately in terms of color. This decision depends on the use case, as illustrated in the melanoma example in Section 4.2. Additionally, in each plot, a small box displaying information for the respective instance and output node is presented. This includes the prediction, the sum of relevances, and, if available, the method's decomposition target of the sum of relevance. The appearance of the box can be toggled with the `show_preds` argument. For the bike rental example, Figure 5a top shows the visualized relevances of the first instance and the box with the additional predictive and decomposition information. Furthermore, instead of returning objects from the **ggplot2** or **plotly** packages, instances of the S4 classes '`innsight_ggplot2`' and '`innsight_plotly`' are produced, which are explained in the Appendix C for advanced visualizations.

Generic function `plot_global()`

Global behavioral patterns and insights into the model's decision-making process can be derived from the results of multiple instances by appropriately summarizing and aggregating them. The generic function `plot_global()` visualizes these global interpretations over the whole or parts of the given dataset based on the graphics package **ggplot2** or the package **plotly** for interactive charts analogous to the previously discussed function `plot()`. Box plots are created for the features of tabular and signal data, and the median pixels' relevance is shown for image data due to the high dimensionality. For the former, the alias function `boxplot()` can also be used analogously. The call for a method's object `method` is the following:

```
plot_global(method, output_idx = NULL, output_label = NULL, data_idx = "all",
  ref_data_idx = NULL, aggr_channels = "sum", preprocess_FUN = abs,
  as_plotly = FALSE, same_scale = FALSE, ...)
```

and for tabular and signal data:

```
boxplot(method, ...)
```

In addition to the identical arguments `output_idx`/`output_label`, `aggr_channels`, `as_plotly`, and `same_scale` for the `plot()` function, options for selecting the data points to be aggregated (`data_idx`), for drawing a reference data point (`ref_data_idx`) and a preprocess function of the results (`preprocess_FUN`) are added. By default, the absolute values of the relevances are calculated, as in global contexts, the focus is usually more on the intensity of effects rather than their orientation. However, this can be adjusted with the `preprocess_FUN` argument, e.g., showing the orientation with `identity`. Analogous to `plot()`, the visualization style depends on the type of input data; tabular and signal data are displayed as box

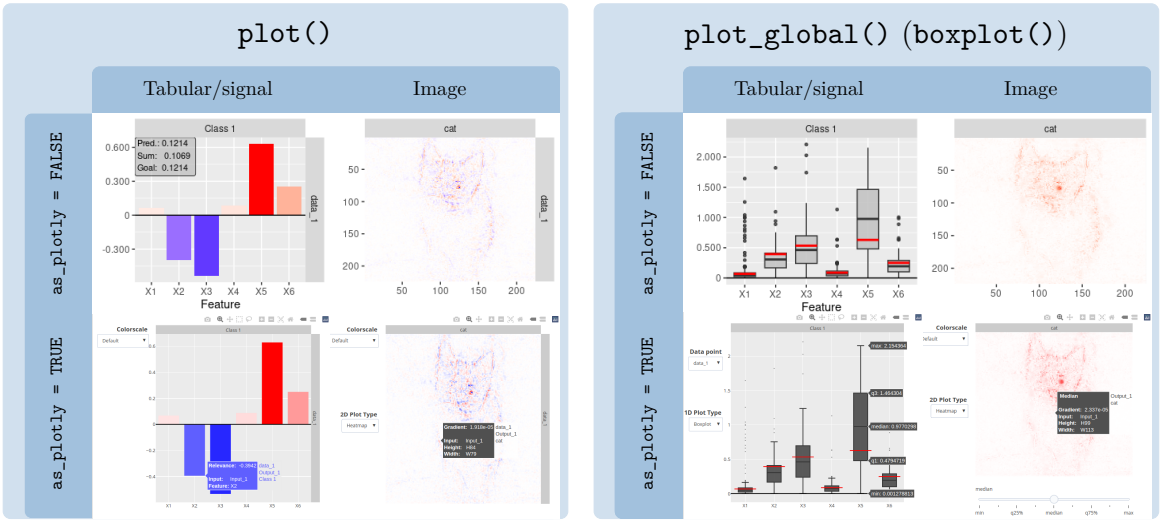


Figure 6: Overview of the visualization tools `plot()` and `plot_global()` provided by the **innsight** package depending on the type of input and the argument `as_plotly`. The function `boxplot()` is an alias for `plot_global()` in case of tabular or signal data.

plots, whereas only a raster plot with the pixel-wise median is rendered for image data due to the high dimensionality. For example, a box plot is created in the bike sharing example since it contains tabular data (see Figure 5a bottom). In this visualization, the red reference lines are drawn for the first dataset instance using the argument `ref_data_ix`. However, if the chart is **plotly**-based, there is a slider to select which quantile to display. Basic examples and an overview of the `plot_global()` function are given in Figure 6 on the right. Despite the creation of **ggplot2** or **plotly** graphs, instances of the S4 class '`innsight_ggplot2`' or '`innsight_plotly`' are returned, which are explained in the Appendix C.

4. Illustrations

To exemplify the methods and step-by-step execution of the **innsight** package, a standard dataset with only numerical tabular inputs on a simple model and a more complex dataset with image and tabular data on an extensive non-sequential network are analyzed in the following. The penguin dataset from the **palmerpenguins** package (Horst, Hill, and Gorman 2022) is used as the simple dataset, taking only the numerical variables of bill length and depth, flipper length, and body weight as inputs. The melanoma dataset (Rotemberg *et al.* 2020) of the Kaggle competition[1] is taken as the second dataset, which classifies the malignancy or benignity of the skin cell based on images of skin lesions and moles, and patient-level contextual information. Both datasets are classification tasks. Even though the bike sharing dataset has already exemplified a regression problem, feature attribution methods typically treat classification problems similarly to regression problems: The activation of the last layer – usually sigmoid or softmax functions – is ignored, and the pre-activation score is explained instead of the actual probability $\hat{y}_c$ (Shrikumar *et al.* 2017a; Montavon, Binder, Lapuschkin, Samek, and Müller 2019).

[1]See the following link for the official dataset description https://www.kaggle.com/competitions/siim-isic-melanoma-classification/overview/description.

4.1. Example 1: Penguin dataset

In the first example, the penguin dataset provided by the **palmerpenguins** package (Horst *et al.* 2022) is used, and a neural network consisting of a dense layer is trained using the **neuralnet** package (Günther and Fritsch 2010). Before the **innsight** package can be applied, the dataset must be processed, and the neural network must be trained on the modified dataset. As a first pre-processing step, only the variables with the species, bill length and depth, flipper length, and body weight are selected, cleaned of missing values, and numerical variables are normalized:

```
R> library("palmerpenguins")
R> set.seed(42)
R> data <- na.omit(penguins[, c(1, 3, 4, 5, 6)])
R> data[, 2:5] <- scale(data[, 2:5])
```

Next, the dataset is divided into training data and test data at a ratio of 75% to 25%:

```
R> train_idx <- sample.int(nrow(data), as.integer(nrow(data) * 0.75))
R> train_data <- data[train_idx, ]
R> test_data <- data[-train_idx, -1]
```

In the second pre-processing step, a network with 128 units in a single hidden layer and the logistic function as activation is fitted on the training data `train_data`:

```
R> library("neuralnet")
R> model <- neuralnet(species ~ ., data = train_data, hidden = 128,
+    act.fct = "logistic", err.fct = "ce", linear.output = FALSE)
```

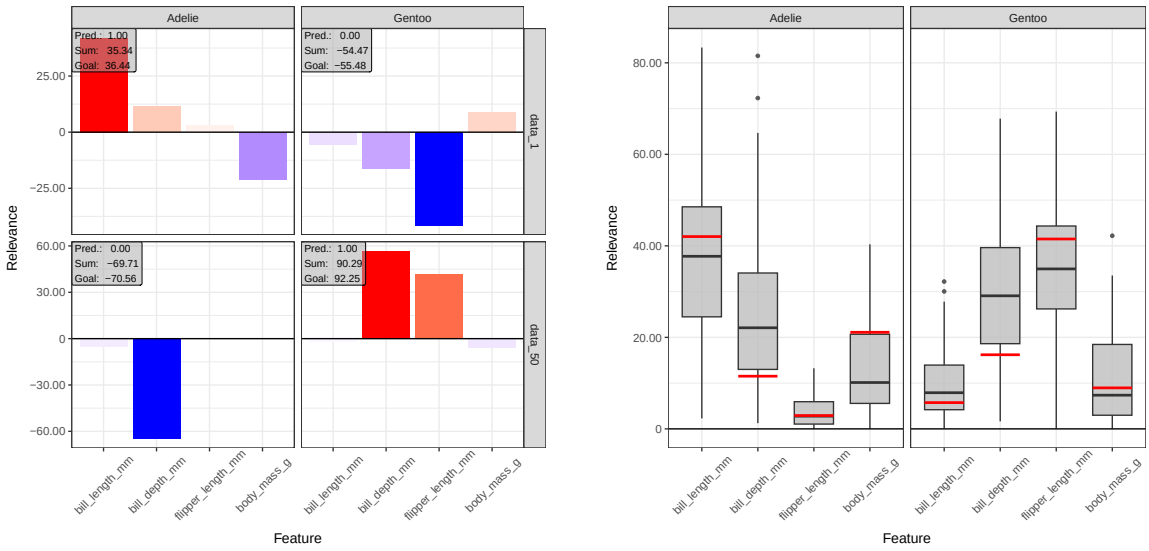
Now, we follow the three steps that provide and visualize an explanation of the model `model` on the test data `test_data`, which were described in detail in Section 3. As a reminder, the first step uses the `convert()` – a shortcut function for initializing an object of the **R6** class '`Converter`' – to convert the given `model` to a **torch**-based model:

```
R> library("innsight")
R> conv <- convert(model)
```

Then, in the second step, the desired method is selected and applied to the test data `test_data` via the corresponding function `run_*` which is identical to initializing the respective **R6** class. In this example, the *IntegratedGradient* method is applied with the average feature value as a baseline:

```
R> intgrad <- run_intgrad(conv, test_data,
+    x_ref = matrix(colMeans(test_data), 1))
```

In the last step, the results are visualized in two ways: Using the `plot()` function, the relevances of one instance of the species Adelie (data index 1) and one of the species Gentoo (data index 50) are displayed for both corresponding classes. Secondly, the results for the two classes, Adelie and Gentoo, are aggregated over the entire test data, and box plots are generated using the `boxplot()` function, showing the first penguin as a reference. As



(a) Visualization created with `plot()`

(b) Visualization created with `boxplot()`

Figure 7: Generated visualizations of *IntegratedGradient* results with average feature value as a baseline on the penguin dataset. Sub-figure (a) shows the individual results from data points 1 and 50 from the test data `test_data` for the Adelie and Gentoo classes. In contrast, the summarized results as box plots across the whole test data for the same two classes can be found in (b), including the individual result of the first data point with the red line.

mentioned in Section 3, these two variants can be treated and modified like ordinary **ggplot2** objects, e.g., adding themes or rotating the x axis labels. Both visualizations are executed by the following code and can be viewed in Figure 7:

```
R> library("ggplot2")
R> plot(intgrad, data_idx = c(1, 50), output_label = c("Adelie", "Gentoo")) +
+    theme_bw() + theme(axis.text.x = element_text(angle = 45, vjust = 0.6))
R> boxplot(intgrad, output_label = c("Adelie", "Gentoo"), ref_data_idx = 1) +
+    theme_bw() + theme(axis.text.x = element_text(angle = 45, vjust = 0.6))
```

In Figure 7a, it can be seen that the bill length for the chosen penguin of the Adelie class (index 1 in the dataset `test_data`) is highly relevant – based on the trained model – for this particular class aligning to the prediction of 100% which is shown in the info box. At the same time, the penguin's flipper length argues against the Gentoo class due to its strong negative relevance. For the Gentoo penguin, the bottom row in Figure 7a reveals that the bill depth is decisively in favor of the Gentoo class and concurrently against the Adelie species. The respective info boxes show that the model is generally very confident in its prediction and that the method has achieved its decomposition goal very well. As mentioned at the beginning of the section, the pre-activations are decomposed by default for classification problems and not the probability scores. Besides these instance-wise explanations, the `boxplot()` function provides aggregate insights across the entire test data `test_data`, summarized in Figure 7b. The box plots show that, indeed, the bill length and depth are the most crucial variables for the Adelie class and consequently strongly influence it. Simultaneously, however, the length

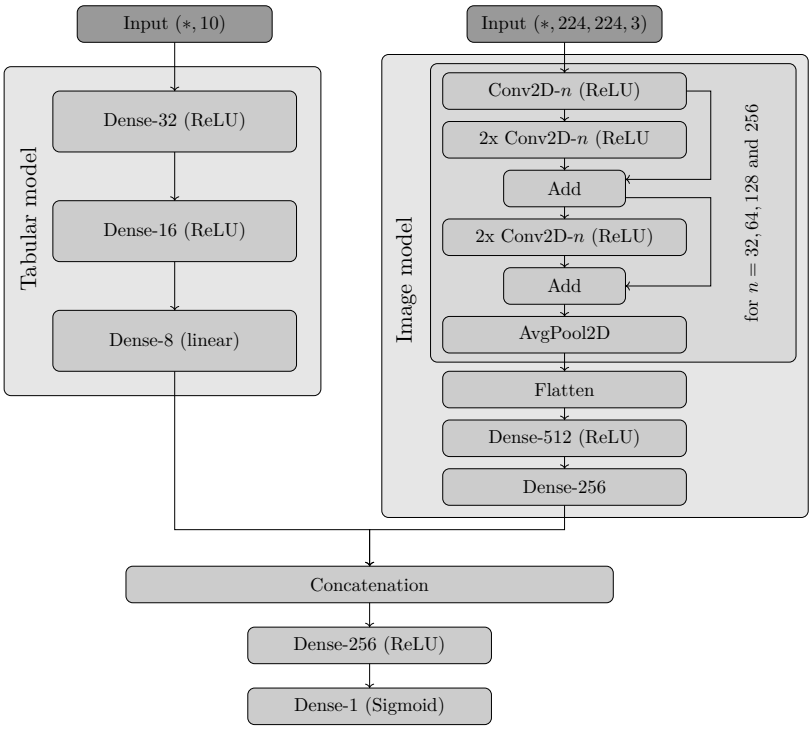


Figure 8: Model architecture for the melanoma dataset.

of the bill is not as decisive for the Gentoo class, but the bill depth is. It further emerges that the flipper length is another crucial feature for the Gentoo class. The red lines for the first penguin also show that the bill depth is not as relevant as it is on a global scale.

4.2. Example 2: Melanoma dataset

The second example examines the melanoma dataset (Rotemberg *et al.* 2020) from the Kaggle challenge[2] in 2020, issued by the society of imaging informatics in medicine (SIIM) and based on the international skin imaging collaboration (ISIC) archive, the most extensive publicly available collection of quality-controlled dermoscopic images of skin lesions. This dataset consists of 33 126 labeled images with associated patient-level contextual information, such as the age, gender, and image location of the skin lesion or mole.

Due to the complexity and high dimensionality of the data, training a neural network is not straightforward and overall not the main focus of this paper thus, reference is made to the GitHub repository for reproduction (`https://github.com/bips-hb/JSS_innsight/`), and only the most notable points are summarized in the following: The tabular input part's numerical and one-hot encoded categorical variables are fed into a sequential model of dense layers. On the other hand, an architecture based on the established residual layers (He, Zhang, Ren, and Sun 2016) considering skip connections between convolutional layers is used for the image data. Afterward, the two outputs of the respective input parts are merged by concatenation and finally flow in a sequential model with only dense layers to obtain a prediction

[2]See the following link for the official dataset description `https://www.kaggle.com/competitions/siim-isic-melanoma-classification/overview/description`.

probability for the skin lesion status. The coarse structure is summarized in Figure 8, where additional dropout layers are used between dense layers. Furthermore, the numerical variable age and the one-hot encoded variables gender and location yield ten features as inputs for the tabular model, and the images are resized to $224 \times 224 \times 3$ for the image model. This model architecture is trained on the melanoma dataset with a validation split of 20% and a batch size of 256 instances using the **Keras** library (Chollet *et al.* 2015) with stochastic gradient descent as the optimizer and class-weighted binary cross-entropy as the loss function. The best model is selected based on the highest value of the area under the ROC curve (AUC) on the validation data. This metric is chosen because the dataset is highly imbalanced with only 584 of the 33 126 images containing a malignant skin lesion. Since the model is trained from scratch and the image model has significantly more parameters than the tabular one, training starts with 300 warm-up epochs on the image model using the image data only. Then, the image model is joined with the tabular and the dense output model. Afterward, training continues on the image and tabular data, saving the model with the highest value of the AUC metric on the validation data. In addition, the initial learning rate of 0.01 is reduced by a factor of 0.1 after 20 epochs without a validation AUC improvement, and training is terminated after 40 unimproved epochs. With this approach, an AUC value of 87.71% and an accuracy of 84.19% on the validation data are achieved, and the model to be interpreted is selected.

Based on this model, the obtained predictions can now be explained using the 3-step approach of **innsight**: In the first step, the trained model is loaded and converted to a **torch**-based model using the `convert()` function for initializing a '`Converter`' object. However, since **keras** models do not include names of the input variables and output nodes, these can be passed along when initializing the converter to preserve meaningful labels of the input and output variables in the visualizations. Thus, the first step is executed by the following R code:

```
R> library("keras")
R> library("innsight")
R> model <- load_model_tf("additional_files/melanoma_model.h5")
R> input_names <- list(
+    list(paste0("C", 1:3), paste0("H", 1:224), paste0("W", 1:224)),
+    list(c("Sex: Male", "Sex: Female", "Age",
+      "Loc: Head/neck", "Loc: Torso", "Loc: Upper extrm.",
+      "Loc: Lower extrem.", "Loc: Palms/soles", "Loc: Oral/genital",
+      "Loc: Missing")))
R> output_name <- c("Probability of malignant lesion")
R> converter <- convert(model, input_names = input_names,
+    output_names = output_name)
```

Next, the *LRP* method with composite rules is applied, which selects the propagation rule depending on the layer type. For convolutional layers, the α-β-rule with $\alpha = 1.5$ is used to favor the positive over the negative relevances. In addition, the ε-rule with $\varepsilon = 0.01$ is performed on all dense layers and the simple rule – the rule used by default – on average pooling layers. This second step is performed with **innsight** as follows:

```
R> rule_name <- list(Conv2D_Layer = "alpha_beta", Dense_Layer = "epsilon")
R> rule_param <- list(Conv2D_Layer = 1.5, Dense_Layer = 0.01)
```

```
R> res <- run_lrp(converter, inputs, channels_first = FALSE,
+    rule_name = rule_name, rule_param = rule_param)
```

For the sake of simplicity, the loading of the input data `inputs` is omitted in the above code snippet and can be found in the reproduction material together with the whole example. In addition, the channel axis of the images is located at the last position, which is why the argument `channels_first` must be set to `FALSE`. The results can be visualized using the implemented `plot()` function. By default, the results are scaled using colors (red for positive and blue for negative relevances) for each instance, each considered output node and each input layer individually. This behavior is especially appropriate for models with multiple input layers consisting of images mixed with tabular data. Because even if the relevances are the same at the end of the tabular and image model before merging, they are further propagated to only ten input variables for the tabular and $224 \times 224 \times 3$ variables for the image model, leading to potentially different relevance scales. The following code produces the plot object based on the S4 class '`innsight_ggplot2`' for the first three dataset instances, which can be treated as a **ggplot2** object (see Appendix C for details):

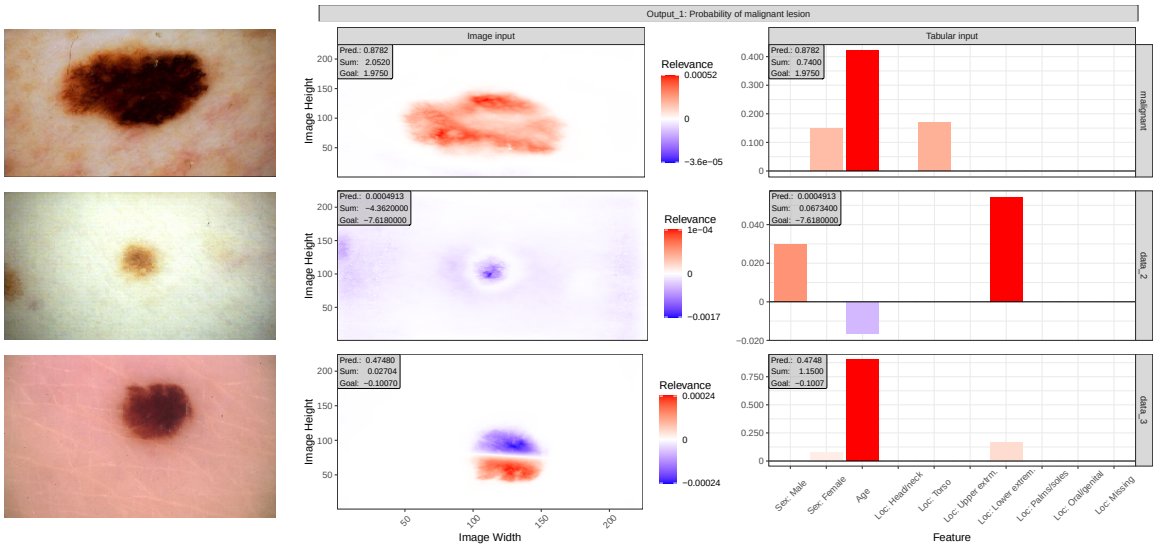
```
R> library("ggplot2")
R> p <- plot(res, data_idx = 1:3) + theme_bw()
```

Since this model has no standard architecture and the visualization is more extensive, the suggested packages **gridExtra** (Auguie 2017) and **gtable** (Wickham and Pedersen 2024) are required. However, each individual plot in the object `p` can now be modified individually based on the **ggplot2** syntax. The indexing works as the objects are plotted in a matrix-wise fashion, provided by the facet rows and columns. It is pointed out that each plot object is based on the same dataset, which is also created by the method `get_result(type = "data.frame")`, i.e., the same column names can be used within the **ggplot2** syntax. In the following code snippet, the facet and the x axis labels are changed manually, and the plot is visualized, which can be found in Figure 9:

```
R> p[1, 1] <- p[1, 1, restyle = FALSE] + facet_grid(cols = vars(model_input),
+    labeller = as_labeller(c(Input_1 = "Image input")))
R> p[1, 2] <- p[1, 2, restyle = FALSE] + facet_grid(cols = vars(model_input),
+    rows = vars(data), labeller = as_labeller(c(Input_2 = "Tabular input",
+      data_1 = "malignant")))
R> p <- p + theme(axis.text.x = element_text(angle = 45, vjust = 0.6))
R> plot(p, heights = c(0.31, 0.31, 0.38))
```

The argument `restyle` is set when indexing the '`innsight_ggplot2`' object, ensuring that the subplots are extracted in the same way as they are displayed in the whole plot. Otherwise, the entire plot's corresponding facet stripes and axis labels are transferred to the selection. In addition, the arguments in the generic function `plot()` for '`innsight_ggplot2`' objects are forwarded to the function `gridExtra::arrangeGrob()` when the plot is finally rendered. This feature allows adjusting the relative heights and widths, demonstrated in the last line of code, to slightly compensate for the increased vertical space of the rotated axis labels.

The three instances in Figure 9 describe different explanatory approaches to the trained model's predictions: The top image in Figure 9a of a malignant lesion was recorded on the



(a) Input images (b) Plot generated by **innsight**

Figure 9: The image part of the instance of the melanoma dataset to be explained and the associated visualization generated by **innsight**. Figure (a) shows a (top) malignant lesion image of a 65-year-old female, (middle) benign lesion of a 40-year-old male, and (bottom) malignant lesion of a 90-year-old female patient. Figure (b) displays the *LRP* explanation of the patients from (a) created with the `plot()` function and subsequent minor modifications such as facet and x axis labels.

torso of a 65-year-old female patient. In the associated interpretation generated by **innsight** (top row in Figure 9b), it can be observed that, on the one hand, the model identifies the lesioned skin area. On the other hand, the darker and patchy pigmentation and the ragged borders positively influenced the prediction of 87.82% for melanoma (shown as 0.8782 in the corresponding infobox in Figure 9b). This observation is also consistent with the official ABCD checklist for melanoma (Friedman, Rigel, and Kopf 1985), which states that asymmetry, irregular borders, varying color, and large diameters are indicative of a malignant skin lesion. However, the patient's age also positively affected the prediction, as evident from the tabular patient-level information explanation in Figure 9b. A complimentary picture results from the middle image in Figure 9a, showing a benign mole located on the lower extremities of a 40-year-old man. The model predicted a probability of only 0.05% for a malignant lesion and explains its decision with the symmetrical shape, uniform color pigmentation, and lack of notched borders. In addition, the age of 40 also has a slightly negative influence on the prediction, whereas the location seems to contribute a large positive effect (middle row in Figure 9b). The last instance exemplifies a situation where the model is uncertain whether it is a malignant or benign skin lesion since its prediction is 47.48%. The truly malignant skin area originates from the lower extremities of a 90-year-old woman (bottom image in Figure 9a). Especially the image input explanation in the last row in Figure 9b shows the model's uncertainty because the mole's upper part looks very regular, arguing for a healthy lesion and consistently highlighted with negative relevance (blue) by the model's explanation. In contrast, the lower part contains some notches potentially favoring melanoma, which the model also correctly identified. Furthermore, the high age of the 90-year-old patient has a

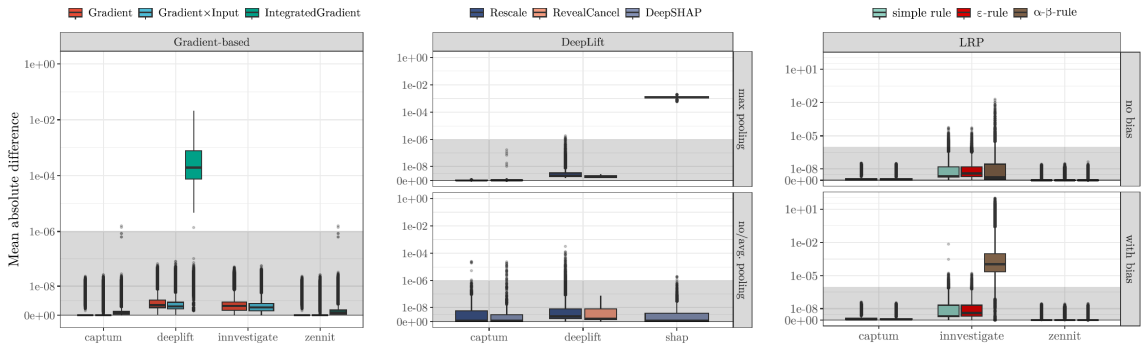
strong positive relevance to the model's prediction, demonstrating the strong effect of the feature age. The corresponding infoboxes in Figure 9b also show that – while the decomposition goals are not achieved, which, however, is expected in *LRP* due to the absorption of relevance into the bias vectors – the majority of relevance for the top two patients comes from the image and less from patient-level data. Only in the case of the last patient do the image relevances seem to cancel each other out so that the tabular features have the strongest contribution to the prediction.

5. Validation and runtime

To evaluate the validity and computational performance of **innsight**, the results of the presented feature attribution methods on simulated models and data are compared with the results of the Python implementations **zennit** (Anders *et al.* 2021), **innvestigate** (Alber *et al.* 2019), **captum** (Kokhlikyan *et al.* 2020), **deeplift** (Shrikumar *et al.* 2017a), and **shap** (Lundberg and Lee 2017). The packages **deeplift** and **innvestigate** are based on the high-level machine learning library **Keras** (Chollet *et al.* 2015) and utilize **TensorFlow** (Abadi *et al.* 2015) as the backend for all calculations. In addition, both packages initially create a replication of the passed model with the interpretation methods pre-implemented in the individual layers, similar to **innsight**. In contrast, the packages **zennit** and **captum** use **PyTorch** (Paszke *et al.* 2019) and run without a conversion step since hooks are used to modify the automated backward pass according to the applied method on the fly (see Appendix B for more details). The package **shap** can handle both **Keras** and **PyTorch** models, and also uses hooks to modify automatic gradients. However, this only enables the application of methods that can be considered independent of the preceding and following layers, which complicates, for example, an implementation of *DeepLift* with the *RevealCancel* rule. Furthermore, not every package supports all methods. For example, **innvestigate** and **zennit** are more geared towards standard gradient methods and *LRP*, whereas **deeplift** is more or less an implementation of

	Package					
	captum	**deeplift**	**innvestigate**	**shap**	**zennit**	**innsight**
Gradient	✓	✓	✓	✖	✓	✓
SmoothGrad	✓	✓	✓	✖	✓	✓
Gradient × *Input*	✓	✓	✓	✖	✓	✓
IntegratedGradient	✓	✓*	✓	✖	✓	✓
ExpectedGradient	✓	✖	✖	✓	✖	✓
LRP (simple rule)	✓	✖	✓	✖	✓	✓
LRP (ε-rule)	✓	✖	✓	✖	✓	✓
LRP (α-β-rule)	✖	✖	✓†	✖	✓	✓
DeepLift (rescale)	✓	✓	✖	✖	✖	✓
DeepLift (reveal-cancel)	✖	✓	✖	✖	✖	✓
DeepSHAP	✓	✖	✖	✓‡	✖	✓

Table 1: Summary of the implemented feature attribution methods in each package. *__deeplift__ applies the middle Riemann sum, whereas **captum**, **zennit** and **innsight** use right Riemann sum. †__innvestigate__ uses another definition of the positive and negative part of the bias vector (see Appendix E). ‡For max pooling layers, **shap** uses the cross maximal value of $\boldsymbol{x}$ and reference value $\tilde{\boldsymbol{x}}$ instead their individual maximal value.



(a) Gradient-based (Section 2.1) (b) *DeepLift* (Section 2.3) (c) *LRP* (Section 2.2)

Figure 10: Comparison of feature attribution methods' results of **innsight** and the reference implementations **captum**, **zennit**, **innvestigate**, **deeplift** and **shap** regarding the mean absolute difference as box plots over different model architectures and repetitions. It shows the results separated into (a) gradient-based methods, (b) *DeepLift*, and (c) *LRP*. The shaded gray area indicates the error tolerance of 10^{-6}.

the methods from its associated paper and focuses on the *DeepLift* method. Package **shap**, originating from a methods paper, primarily considers Shapley-value-based methods. On the other hand, **captum** is a good all-rounder, but still has some gaps in the context of *LRP*. A summary of the packages' implemented feature attribution methods is provided in Table 1.

5.1. Validity comparison

For the validation, shallow untrained dense and convolutional models with the most commonly used layer types – such as 2D convolution, 2D maximum/average pooling, and dense layers – and normally distributed input data are generated. More specifically, 32 different architectures are considered, using ReLU and hyperbolic tangent to include both constrained and unconstrained activation functions, with and without bias vectors, with varying pooling layers, and a different number of output nodes. From each of these architectures, 50 randomly initialized models are created, resulting in 1 600 distinct models, and evaluated on normally distributed datasets with 16 input instances each. The experimental details can be found in Appendix D.1. Moreover, all figures and results are reproducible using the code in the reproduction material or on GitHub (`https://github.com/bips-hb/JSS_innsight`). As a measure of quality, the mean absolute difference between the results of **innsight** and the corresponding reference implementation over all input variables and output nodes is considered. Consequently, for each combination of method, model, input instance, and output node, a value for this quality measure is derived, leading to box plots for visualizing the differences. In addition to the box plots, an acceptable error range of up to 10^{-6} is highlighted in light gray to distinguish numerical tolerated differences caused by calculations of single-precision floating point numbers according to the IEEE 754 standard (IEEE 2019) from abnormal discrepancies. The results are summarized in Figure 10.

For the gradient-based methods *Gradient* and *Gradient* × *Input*, the method's results mostly coincide for all packages (see Figure 10a). The only discrepancy is with the *IntegratedGradient* method for the package **deeplift**, but this is due to the fact that **innsight** approximates the

integral with the right and **deeplift** with the middle Riemann sum.[3] The right Riemann sum is also used in **zennit**[4], whereas various approximation methods can be selected in **captum**.

A similar picture results for the *DeepLift* method with the *Rescale* and *RevealCancel* rules, but with a few outliers with a maximum error of up to 10^{-3} for the *Rescale* rule (see Figure 10b). However, all outliers with an error exceeding 10^{-6} originate from models with the hyperbolic tangent as activation and can thus be explained by numerical inaccuracies due to the saturated activation. In addition, minor discrepancies are probably caused by different treatments of vanishing denominators in the multipliers or numerical uncertainties between the backends **PyTorch**/**LibTorch** and **TensorFlow** in general. With the *DeepSHAP* method – similar to *IntegratedGradient* – the results of **innsight** largely match those of **captum**, but they differ from the results of **shap**. However, this is mainly due to the max pooling layer (see Figure 10b), which is handled differently in **shap** than in **innsight** and **captum**. Since *DeepSHAP* is a repeated application of *DeepLift* with different reference values, the numerical inaccuracies from the *DeepLift* method accumulate and, thus, cause the visible outliers.

For the *LRP* methods, a few adjustments are needed for the **innvestigate** and **captum** packages since they use only the simple rule for average pooling layers, which is modified in **innsight** using composite rules. Apart from that, the results from **innsight** compared to **captum** or **zennit** for the simple, ε-rule and α-β-rule differ negligibly and are far below the maximally tolerated error of 10^{-6} (see Figure 10c). For the simple and ε-rule, **innvestigate** is consistent with **innsight** except for a few deviations. Again, almost all of the cases with errors exceeding 10^{-6} are caused by a saturated hyperbolic tangent activation, lower errors on different stabilizers for the denominators in the relevance messages, and general numerical inaccuracies between their backends. However, significant discrepancies can be observed using the α-β-rule, which only occur in models with a bias vector (see Figure 10c bottom). The reason for this is a different interpretation of the positive or negative part of the bias vector, which is discussed in more detail in the Appendix E.

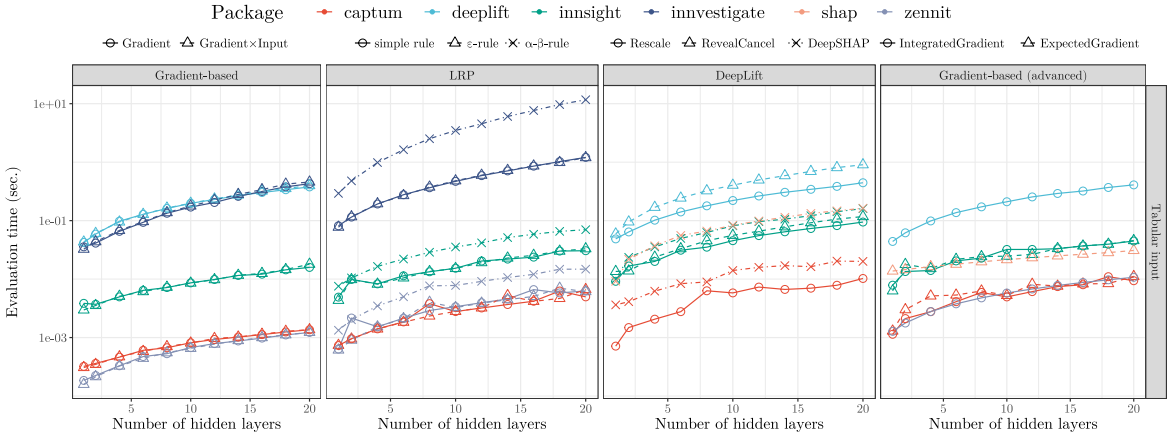
5.2. Runtime comparison

In addition to comparing whether **innsight**'s results are consistent with the reference implementations, a runtime comparison is also conducted concerning the number of output nodes, hidden units or filters, hidden layers, batch size, and, for images, the size of the input images. It must be noted again that the packages based on **Keras** and **innsight** first convert the passed model, and the **PyTorch**-based packages use hooks to overwrite the automated backward pass while executing, making them considerably faster. Therefore, in the results, only the execution time excluding the conversion step – as far as possible – is presented and not the total time. For comparisons of the total time needed to calculate an explanation, see Appendix D.3. However, the **innvestigate** package has a special characteristic in this regard since the entire conversion process and the construction of the underlying graph only happens during the analysis of the first batch of input data.[5] For this reason, conversion times are almost hardly present in the results. Since this simulation assumes that an interpretation method is being applied for the first time to a model and only to a single batch of input

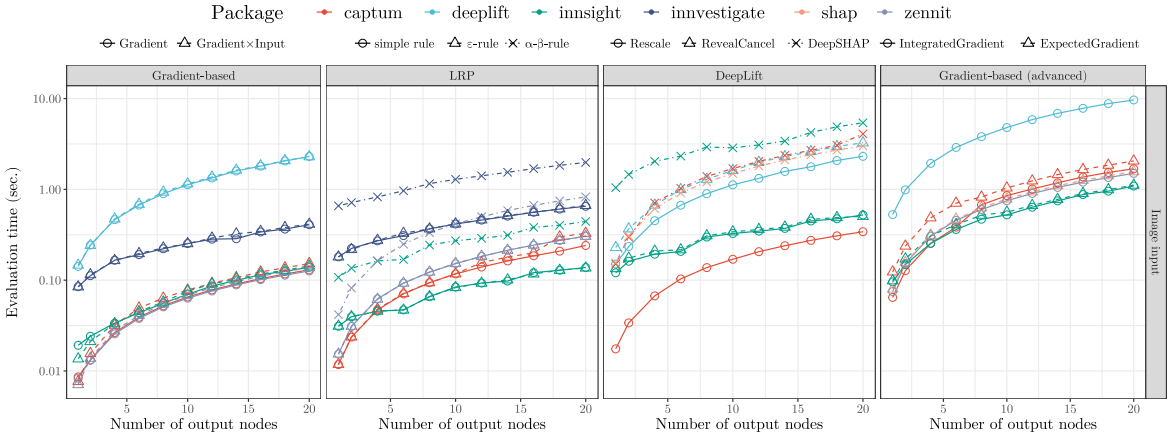
[3] See https://github.com/kundajelab/deeplift/blob/master/deeplift/util.py#L261.

[4] See https://github.com/chr5tphr/zennit/blob/0.5.1/src/zennit/attribution.py#L453.

[5] See the GitHub issues https://github.com/albermax/innvestigate/issues/50 and https://github.com/albermax/innvestigate/issues/129.



(a) Time comparison for a varying number of hidden layers (L)



(b) Time comparison for a varying number of output nodes (C)

Figure 11: Package's average evaluation time in seconds over 20 repetitions for applying different feature attribution methods on models with (a) a varying number of hidden layers and (b) a varying number of output nodes (only image data).

instances, the results of **innvestigate** are slightly biased and would be notably quicker if the same model is employed with more input batches.

Analogously to the comparison from Section 5.1, untrained dense and convolutional neural networks, and normally distributed input data are used for the runtime comparisons. Depending on the type of time comparison, the hyperparameters for the number of output nodes, number of hidden units or filter size, number of hidden layers, batch size, and the size of the input images are varied. The hyperparameters not considered in the respective comparison remain unchanged and take default values, i.e., one output node, 128 hidden units for the tabular model and 5 filters for the image model, two layers in total, a batch size of 16 and an input image size of 64×64. In addition, 20 replicates of each architecture are created to compensate for potential numerical fluctuations. For a more detailed simulation description or analysis of the results, including the total time, please refer to Appendix D.2 and Appendix D.3, and for a reproduction of the results, see the reproduction material or the code in the GitHub repository at `https://github.com/bips-hb/JSS_innsight/`.

In general, comparing the runtimes of the different packages reveals that **innsight** is faster than **innvestigate** and **deeplift** (which are based on **Keras**), but slower than **captum**, **zennit** and **shap** (which are based on **PyTorch**). This overall trend is particularly evident when adding more layers to dense models: **innsight** is $10 - 15$ times slower than **captum** and **zennit**, but still for the same order of magnitude faster or even faster than **innvestigate** and **deeplift** (see Figure 11a). In addition, **innsight** is similarly fast as **shap** for the methods *DeepSHAP* and *ExpectedGradient*. This trend can largely be extended to image inputs as well, with the exception that the evaluation times in **innsight** approach those of **PyTorch**-based implementations very closely. However, the *DeepLift* methods, particularly with the *Rescale* rule and thus even more with *DeepSHAP*, are considerably slower when applied to images compared to **deeplift** (see Figure 15).

The same trend can also be observed for a varying number of output nodes. In the case of image data, **innsight** is similarly fast as the **PyTorch**-based packages and mostly considerably faster than **innvestigate** and **deeplift**. However, the exception concerning the *DeepLift* methods applies here as well: *DeepLift* using the *Rescale* rule runs almost 10 times slower than in **captum**. Consequently, *DeepSHAP* is also considerably slower than the runtime in **shap** and **captum** (see Figure 11b). On the other hand, **innsight** stands out for all rule-based methods applied to dense models and performs similarly or even faster than the **PyTorch**-based packages (see Figure 14). The primary reason for this is that the results for several output nodes can be calculated at once in **innsight**. In contrast, all other implementations only allow the calculation for single nodes, and thus, the method's results are computed by iterative execution.

These tendencies can also be observed in the other simulations' results when changing the number of hidden units/filters, batch size, or image size (see Figure 16, 17 and 18 in the Appendix). In the gradient-based methods and *LRP*, the execution times of **innsight** are increasingly approaching the times of the **PyTorch**-based implementations. With a larger number of filters in the convolutional models, the runtimes are almost identical. At the same time, it can be seen that the **Keras**-based packages run considerably slower than all other packages. An exception to this is with a high number of filters in convolutional models: in these cases, **innvestigate** and **innsight** applied to standard gradient-based methods and LRP are faster than all other implementations. In addition, the previously observed pattern with the *DeepLift* methods repeats in the other simulations as well: the execution time of *DeepLift* with the *Rescale* rule for image data is noticeably slower than in the reference implementations, which also explains the high execution time of *DeepSHAP*. Nevertheless, **innsight** is considerably faster than **deeplift** in dense models running *DeepLift* methods, and, in the case of batch size, also faster than **shap**.

In summary, **innsight** is mostly faster than the **Keras**-based packages and becomes more comparable to the **PyTorch** packages with increasing number of input instances, hidden units/filters, image size, and number of output nodes. However, the evaluation time of *DeepLift* with the *Rescale* rule and *DeepSHAP* applied to images is a weakness of **innsight** and shows the largest discrepancy in the runtime comparison. Even though the packages in this section are compared regarding runtime and some packages' weaknesses are revealed, all considered implementations provide an explanation within a reasonable time of a few seconds, even for deep neural networks with several large images as inputs.

6. Summary and discussion

In summary, we have presented **innsight**, an R package that provides the most well-known feature attribution methods for interpreting neural network predictions. After a detailed introduction of the implemented feature attribution methods, the internal structure utilizing **torch**'s fast array calculations and conversion function `convert()` for initializing an `Converter` object demonstrated how the deep-learning-model-agnostic approach was implemented to enable the analysis of models from any R package in an efficient way. This flexibility is complemented by a unified 3-step approach from model to plotted results, including multiple visualization tools based on **ggplot2** or **plotly** for interactive plots. The step-wise procedure was illustrated using a model on the tabular penguin dataset and a deep neural network on the melanoma dataset consisting of structured patient-level information and images. Furthermore, the results of the simulation study show that **innsight** returns nearly identical feature-wise explanations to the reference implementations **captum**, **zennit**, **innvestigate**, **deeplift**, and **shap** in Python. Any observed differences are either below the accepted error tolerance of 10^{-6}, reflecting negligible numerical inaccuracies, or caused by variations in layer treatments, e.g., max pooling layer in **shap** for *DeepSHAP* or another integral estimation for *IntegratedGradient* in **deeplift**. In terms of runtime, the package shows that it is generally faster than the **Keras**-based packages and slower or comparable fast than **captum**, **zennit**, and **shap**. It only suffers with the *DeepLift* method using the *Rescale* rule for image data, which is also reflected in the *DeepSHAP* method that builds on it. Apart from that, the package also has some limitations that could be improved in the future. For example, only converting sequential models (i.e., `nn_sequential`) from the **torch** package is possible because no structured network graph can be extracted from an arbitrary `nn_module`. For all the gradient-based methods, however, this is not strictly necessary, which is why these could be extended for arbitrary `nn_modules`. Nevertheless, passing a model as a list allows the user to do the conversion step on their own in such cases. The package can also be extended methodically and offer permutation-based methods such as e.g., Occlusion or RISE (Zeiler and Fergus 2014; Petsiuk, Das, and Saenko 2018). Furthermore, an activation function is assigned to a linear or convolutional layer only if it is defined in the layer itself or immediately after the layer. This behavior is especially relevant for the *RevealCancel* rule in the *DeepLift* method, because **innsight** handles separated activations with the *Rescale* rule, which is the case, for example, with the layer sequence of convolution, batch normalization, and activation. Moreover, even if it is possible in the **torch** package, the **innsight** package currently only supports computations on CPUs and not on GPUs.

Computational details

A 64-bit Linux platform running Ubuntu 20.04 with an AMD Ryzen Threadrippper 3960X (24 cores, 48 threads) CPU including 256 gigabyte RAM and two NVIDIA Titan RTX GPUs was used for all computations. All comparisons and calculations with the reference implementations were performed in a separate session – created by **callr** (Csárdi and Chang 2024) – using only a single CPU thread per job. An exception was the neural network training on the melanoma dataset using a single GPU, which was also the only code executed by Python and not from R (R Core Team 2024). Due to the package requirement mismatch, separate environments were created for each of the **Keras**-based and the **PyTorch**-based packages, i.e.,

- **innvestigate** 2.0.2: Using Python 3.8.15 with **Keras** 2.10.0 and **TensorFlow** 2.10
- **deeplift** 0.6.13: Using Python 3.6.15 with **Keras** 2.2.4 and **TensorFlow** 1.15
- **captum** 0.6.0, **zennit** 0.5.0 and **shap** 0.44.0: Using Python 3.8.12 with **PyTorch** 1.13.1 (cpu)

The corresponding environments were loaded in R, and then the code was executed in Python using **reticulate** 1.39 (Ushey, Allaire, and Tang 2024). In addition, the computer was used exclusively for the runtime measurements for the corresponding job and was not distorted by other simultaneous processes.

Acknowledgments

This project was funded by the German Research Foundation (DFG), Emmy Noether Grant 437611051.

References

Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, Ghemawat S, Goodfellow I, Harp A, Irving G, Isard M, Jia Y, Jozefowicz R, Kaiser L, Kudlur M, Levenberg J, Mané D, Monga R, Moore S, Murray D, Olah C, Schuster M, Shlens J, Steiner B, Sutskever I, Talwar K, Tucker P, Vanhoucke V, Vasudevan V, Viégas F, Vinyals O, Warden P, Wattenberg M, Wicke M, Yu Y, Zheng X (2015). "**TensorFlow**: Large-Scale Machine Learning on Heterogeneous Systems." URL https://www.tensorflow.org/.

Alber M, Lapuschkin S, Seegerer P, Hägele M, Schütt KT, Montavon G, Samek W, Müller KR, Dähne S, Kindermans PJ (2019). "**iNNvestigate** Neural Networks!" *Journal of Machine Learning Research*, **20**(93), 1–8.

Allaire JJ, Chollet F (2024). ***keras**: R Interface to **Keras***. doi:10.32614/CRAN.package.keras. R package version 2.15.0.

Ancona M, Ceolini E, Öztireli C, Gross M (2018). "Towards Better Understanding of Gradient-Based Attribution Methods for Deep Neural Networks." In *Proceedings of the 6th International Conference on Learning Representations*. URL https://openreview.net/forum?id=Sy21R9JAW.

Anders CJ, Montavon G, Samek W, Müller KR (2019). "Understanding Patch-Based Learning of Video Data by Explaining Predictions." In *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, Lecture Notes in Computer Science, pp. 297–309. Springer-Verlag. doi:10.1007/978-3-030-28954-6_16.

Anders CJ, Neumann D, Samek W, Müller KR, Lapuschkin S (2021). "Software for Dataset-Wide XAI: From Local Explanations to Global Insights with **Zennit**, **CoRelAy**, and **ViRelAy**." *arXiv 2106.13200*, arXiv.org E-Print Archive. doi:10.48550/arxiv.2106.13200.

Apley DW, Zhu J (2020). "Visualizing the Effects of Predictor Variables in Black Box Supervised Learning Models." *Journal of the Royal Statistical Society B*, **82**(4), 1059–1086. doi:10.1111/rssb.12377.

Arras L, Osman A, Samek W (2022). "CLEVR-XAI: A Benchmark Dataset for the Ground Truth Evaluation of Neural Network Explanations." *Information Fusion*, **81**, 14–40. doi:10.1016/j.inffus.2021.11.008.

Auguie B (2017). ***gridExtra**: Miscellaneous Functions for "Grid" Graphics*. doi:10.32614/CRAN.package.gridExtra. R package version 2.3.

Bach S, Binder A, Montavon G, Klauschen F, Müller KR, Samek W (2015). "On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation." *Public Library of Science One*, **10**(7), 1–46. doi:10.1371/journal.pone.0130140.

Bengio Y, Lecun Y, Hinton G (2021). "Deep Learning for AI." *Communications of the Association for Computing Machinery*, **64**(7), 58–65. doi:10.1145/3448250.

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

Chang W (2021). ***R6**: Encapsulated Classes with Reference Semantics*. doi:10.32614/CRAN.package.R6. R package version 2.5.1.

Chollet F, *et al.* (2015). "**Keras**." https://keras.io/.

Csárdi G (2024). ***cli**: Helpers for Developing Command Line Interfaces*. doi:10.32614/CRAN.package.cli. R package version 3.6.3.

Csárdi G, Chang W (2024). ***callr**: Call R from R*. doi:10.32614/CRAN.package.callr. R package version 3.7.6.

Erion G, Janizek JD, Sturmfels P, Lundberg SM, Lee SI (2021). "Improving Performance of Deep Learning Models with Axiomatic Attribution Priors and Expected Gradients." *Nature Machine Intelligence*, **3**(7), 620–631. doi:10.1038/s42256-021-00343-w.

European Union (2016). "Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the Protection of Natural Persons with Regard to the Processing of Personal Data and on the Free Movement of Such Data, and Repealing Directive 95/46/EC (General Data Protection Regulation)." *Official Journal L119*, **59**, 1–88. doi:10.5593/sgemsocial2019v/1.1/s02.022.

Falbel D, Luraschi J (2024). ***torch**: Tensors and Neural Networks with 'GPU' Acceleration*. doi:10.32614/CRAN.package.torch. R package version 0.13.0.

Fanaee-T H (2013). "Bike Sharing Dataset." UCI Machine Learning Repository. doi:10.24432/c5w894.

Fisher A, Rudin C, Dominici F (2019). "All Models Are Wrong, but Many Are Useful: Learning a Variable's Importance by Studying an Entire Class of Prediction Models Simultaneously." *Journal of Machine Learning Research*, **20**(177), 1–81. doi:10.1101/2023.03.27.534311.

Friedman JH (2001). "Greedy Function Approximation: A Gradient Boosting Machine." *The Annals of Statistics*, **29**(5), 1189–1232. doi:10.1214/aos/1013203451.

Friedman RJ, Rigel DS, Kopf AW (1985). "Early Detection of Malignant Melanoma: The Role of Physician Examination and Self-Examination of the Skin." *CA: A Cancer Journal for Clinicians*, **35**(3), 130–151. doi:10.3322/canjclin.35.3.130.

Goldstein A, Kapelner A, Bleich J, Pitkin E (2015). "Peeking Inside the Black Box: Visualizing Statistical Learning with Plots of Individual Conditional Expectation." *Journal of Computational and Graphical Statistics*, **24**(1), 44–65. doi:10.1080/10618600.2014.907095.

Goodman B, Flaxman S (2017). "European Union Regulations on Algorithmic Decision-Making and a "Right to Explanation"." *AI Magazine*, **38**(3), 50–57. doi:10.1609/aimag.v38i3.2741.

Greenwell B (2024). ***fastshap**: Fast Approximate Shapley Values*. doi:10.32614/CRAN.package.fastshap. R package version 0.1.1.

Greenwell BM, Boehmke BC, McCarthy AJ (2018). "A Simple and Effective Model-Based Variable Importance Measure." *arXiv 1805.04755*, arXiv.org E-Print Archive. doi:10.48550/arxiv.1805.04755.

Gunning D, Aha D (2019). "DARPA's Explainable Artificial Intelligence (XAI) Program." *AI Magazine*, **40**(2), 44–58. doi:10.1609/aimag.v40i2.2850.

Günther F, Fritsch S (2010). "**neuralnet**: Training of Neural Networks." *The R Journal*, **2**(1), 30–38. doi:10.32614/rj-2010-006.

Haug J, Zürn S, El-Jiz P, Kasneci G (2022). "On Baselines for Local Feature Attributions." *arXiv 2101.00905*, arXiv.org E-Print Archive. doi:10.48550/arxiv.2101.00905.

He K, Zhang X, Ren S, Sun J (2016). "Deep Residual Learning for Image Recognition." In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778. IEEE. doi:10.1109/cvpr.2016.90.

Horst AM, Hill AP, Gorman KB (2022). ***palmerpenguins**: Palmer Archipelago (Antarctica) Penguin Data*. doi:10.32614/CRAN.package.palmerpenguins. R package version 0.1.1.

Hvitfeldt E, Pedersen TL, Benesty M (2022). ***lime**: Local Interpretable Model-Agnostic Explanations*. doi:10.32614/CRAN.package.lime. R package version 0.5.3.

IEEE (2019). "IEEE Standard for Floating-Point Arithmetic." *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84. doi:10.1109/ieeestd.2019.8766229.

Kim B, Wattenberg M, Gilmer J, Cai C, Wexler J, Viegas F, sayres R (2018). "Interpretability beyond Feature Attribution: Quantitative Testing with Concept Activation Vectors (TCAV)." In J Dy, A Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80, pp. 2668–2677. PMLR. URL https://proceedings.mlr.press/v80/kim18d.html.

Kindermans PJ, Hooker S, Adebayo J, Alber M, Schütt KT, Dähne S, Erhan D, Kim B (2019). "The (Un)Reliability of Saliency Methods." *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, pp. 267–280. doi:10.1007/978-3-030-28954-6_14.

Kohlbrenner M, Bauer A, Nakajima S, Binder A, Samek W, Lapuschkin S (2020). "Towards Best Practice in Explaining Neural Network Decisions with LRP." In *Proceedings of the 2020 International Joint Conference on Neural Networks*, pp. 1–7. IEEE. doi:10.1109/ijcnn48605.2020.9206975.

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 2009.07896*, arXiv.org E-Print Archive. doi:10.48550/arxiv.2009.07896.

Krizhevsky A, Sutskever I, Hinton GE (2017). "ImageNet Classification with Deep Convolutional Neural Networks." *Communications of the Association for Computing Machinery*, **60**(6), 84–90. doi:10.1145/3065386.

Lang M (2017). "**checkmate**: Fast Argument Checks for Defensive R Programming." *The R Journal*, **9**(1), 437–445. doi:10.32614/rj-2017-028.

Lauritsen SM, Kristensen M, Olsen MV, Larsen MS, Lauritsen KM, Jørgensen MJ, Lange J, Thiesson B (2020). "Explainable Artificial Intelligence Model to Predict Acute Critical Illness from Electronic Health Records." *Nature Communications*, **11**(1), 3852. doi:10.1038/s41467-020-17431-x.

LeCun Y, Bengio Y, Hinton G (2015). "Deep Learning." *Nature*, **521**(7553), 436–444. doi:10.1038/nature14539.

Lundberg SM, Lee SI (2017). "A Unified Approach to Interpreting Model Predictions." In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pp. 4768–4777. Curran Associates Inc. doi:10.5555/3295222.3295230.

Molnar C, Casalicchio G, Bischl B (2018). "**iml**: An R Package for Interpretable Machine Learning." *Journal of Open Source Software*, **3**(26), 786. doi:10.21105/joss.00786.

Montavon G, Binder A, Lapuschkin S, Samek W, Müller KR (2019). "Layer-Wise Relevance Propagation: An Overview." In *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, Lecture Notes in Computer Science, pp. 193–209. Springer-Verlag. doi:10.1007/978-3-030-28954-6_10.

Montavon G, Lapuschkin S, Binder A, Samek W, Müller KR (2017). "Explaining Nonlinear Classification Decisions with Deep Taylor Decomposition." *Pattern Recognition*, **65**, 211–222. doi:10.1016/j.patcog.2016.11.008.

Nielsen IE, Dera D, Rasool G, Ramachandran RP, Bouaynaya NC (2022). "Robust Explainability: A Tutorial on Gradient-Based Attribution Methods for Deep Neural Networks." *IEEE Signal Processing Magazine*, **39**(4), 73–84. doi:10.1109/msp.2022.3142719.

Olah C, Mordvintsev A, Schubert L (2017). "Feature Visualization." *Distill*, **2**(11), e7. doi:10.23915/distill.00007.

Olden JD, Joy MK, Death RG (2004). "An Accurate Comparison of Methods for Quantifying Variable Importance in Artificial Neural Networks Using Simulated Data." *Ecological Modelling*, **178**(3), 389–397. doi:10.1016/j.ecolmodel.2004.03.013.

O'Sullivan S, Nevejans N, Allen C, Blyth A, Leonard S, Pagallo U, Holzinger K, Holzinger A, Sajid MI, Ashrafian H (2019). "Legal, Regulatory, and Ethical Frameworks for Development of Standards in Artificial Intelligence (AI) and Autonomous Robotic Surgery." *The International Journal of Medical Robotics and Computer Assisted Surgery*, **15**(1), e1968. doi:10.1002/rcs.1968.

Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, Desmaison A, Kopf A, Yang E, DeVito Z, Raison M, Tejani A, Chilamkurthy S, Steiner B, Fang L, Bai J, Chintala S (2019). "**PyTorch**: An Imperative Style, High-Performance Deep Learning Library." In *Proceedings of the 2019 Conference on Advances in Neural Information Processing Systems*, volume 32. Curran Associates. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

Petsiuk V, Das A, Saenko K (2018). "Rise: Randomized Input Sampling for Explanation of Black-Box Models." *arXiv 1806.07421*, arXiv.org E-Print Archive. doi:10.48550/arxiv.1806.07421.

R Core Team (2024). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Ribeiro MT, Singh S, Guestrin C (2016). "Why Should I Trust You?": Explaining the Predictions of Any Classifier." In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1135–1144. Association for Computing Machinery. doi:10.1145/2939672.2939778.

Rotemberg VM, Kurtansky NR, Betz-Stablein B, Caffery LJ, Chousakos E, Codella NCF, Combalia M, Dusza SW, Guitera P, Gutman DA, Halpern AC, Kittler H, Köse K, Langer SG, Liopryis K, Malvehy J, Musthaq S, Nanda J, Reiter O, Shih G, Stratigos AJ, Tschandl P, Weber J, Soyer HP (2020). "A Patient-Centric Dataset of Images and Metadata for Identifying Melanomas Using Clinical Context." *Scientific Data*, **8**(1). doi:10.1038/s41597-021-00815-z.

Schneeberger D, Stöger K, Holzinger A (2020). "The European Legal Framework for Medical AI." In *Lecture Notes in Computer Science*, pp. 209–226. Springer-Verlag, Cham. doi:10.1007/978-3-030-57321-8_12.

Shrikumar A, Greenside P, Kundaje A (2017a). "Learning Important Features through Propagating Activation Differences." In *Proceedings of the 34th International Conference on Machine Learning*, volume 70, pp. 3145–3153. PMLR. URL https://proceedings.mlr.press/v70/shrikumar17a.html.

Shrikumar A, Greenside P, Shcherbina A, Kundaje A (2017b). "Not Just a Black Box: Learning Important Features through Propagating Activation Differences." *arXiv 1605.01713*, arXiv.org E-Print Archive. doi:10.48550/arxiv.1605.01713.

Sievert C (2020). *Interactive Web-Based Data Visualization with R, **plotly**, and **shiny***. Chapman & Hall/CRC. ISBN 9781138331457. doi:10.1201/9780429447273.

Silver D, Huang A, Maddison CJ, Guez A, Sifre L, Van den Driessche G, Schrittwieser J, Antonoglou I, Panneershelvam V, Lanctot M, Dieleman S, Grewe D, Nham J, Kalchbrenner N, Sutskever I, Lillicrap T, Leach M, Kavukcuoglu K, Graepel T, Hassabis D (2016). "Mastering the Game of Go with Deep Neural Networks and Tree Search." *Nature*, **529**(7587), 484–489. doi:10.1038/nature16961.

Simonyan K, Vedaldi A, Zisserman A (2014). "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps." *arXiv 1312.6034*, arXiv.org E-Print Archive. doi:10.48550/arxiv.1312.6034.

Smilkov D, Thorat N, Kim B, Viégas F, Wattenberg M (2017). "SmoothGrad: Removing Noise by Adding Noise." *arXiv 1706.03825*, arXiv.org E-Print Archive. doi:10.48550/arxiv.1706.03825.

Stroustrup B (2013). *The C++ Programming Language*. 4th edition. Addison-Wesley.

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

Sundararajan M, Taly A, Yan Q (2017). "Axiomatic Attribution for Deep Networks." *arXiv 1703.01365*, arXiv.org E-Print Archive. doi:10.48550/arxiv.1703.01365.

Ushey K, Allaire JJ, Tang Y (2024). ***reticulate**: Interface to Python*. doi:10.32614/CRAN.package.reticulate. R package version 1.39.0.

Van Rossum G, *et al.* (2011). *Python Programming Language*. URL http://www.python.org/.

Wickham H (2016). ***ggplot2**: Elegant Graphics for Data Analysis*. Springer-Verlag, New York. ISBN 978-3-319-24277-4. doi:10.1007/978-0-387-98141-3.

Wickham H, Pedersen TL (2024). ***gtable**: Arrange 'Grobs' in Tables*. doi:10.32614/CRAN.package.gtable. R package version 0.3.6.

Zeiler MD, Fergus R (2014). "Visualizing and Understanding Convolutional Networks." In *Computer Vision – ECCV 2014: Proceedings, Part I 13*, pp. 818–833. Springer-Verlag. doi:10.1007/978-3-319-10590-1_53.

Zuallaert J, Godin F, Kim M, Soete A, Saeys Y, De Neve W (2018). "SpliceRover: Interpretable Convolutional Neural Networks for Improved Splice Site Prediction." *Bioinformatics*, **34**(24), 4180–4188. doi:10.1093/bioinformatics/bty497.

A. Details on *LRP* and *DeepLift*

A.1. Details on *LRP*

The *layer-wise relevance propagation (LRP)* method was introduced by Bach *et al.* (2015) and has a similar goal as the *Gradient* $\times$ *Input* approach: decomposing the output into variable-wise relevances conforming to Equation 1. The distinguishing aspect is that the prediction $\hat{y}_c$ is redistributed layer by layer from the output node back to the inputs according to the layer's weights and intermediate values. The entire procedure is accomplished by rule-based relevance messages defining how to redistribute the upper-layer relevance to the lower layer. More precisely, the relevance message $r_{i\leftarrow j}^{(l,l+1)}$ describes the amount of the relevance R_j^{l+1} of node j in layer $l+1$ sent to the lower-layer node i. The relevance for the lower-layer node i is calculated as the sum of all incoming relevance messages, i.e.,

$$R_i^l = \sum_j r_{i\leftarrow j}^{(l,l+1)}. \tag{4}$$

In the following, we briefly overview the most popular variations of relevance messages flowing from a node of index j in layer $l+1$ to node i in the preceding layer:

- **The simple rule:** The fundamental rule on which all other variations of relevance messages are more or less based is the *simple rule* (also known as *LRP-0*). The relevances are redistributed to the lower layers according to the ratio between local and global pre-activation. Let $\boldsymbol{x}$ be the inputs of the preceding layer, $\boldsymbol{w}$ the weight matrix and $\boldsymbol{b}$ the bias vector of layer l, and R_j^{l+1} the upper-layer relevance; then $x_i\, w_{ij}$ is the local and $z_j = b_j + \sum_k x_k\, w_{kj}$ the global pre-activation defining the simple rule as

$$r_{i\leftarrow j}^{(l,l+1)} = \frac{x_i\, w_{ij}}{z_j}\, R_j^{l+1}.$$

- **The ε-rule:** One issue with the simple rule is that it is numerically unstable when the global pre-activation z_j vanishes and causes a division by zero. The *ε-rule* (also known as *LRP-ε*) tackles those situations by adding a stabilizer $\varepsilon > 0$ that moves the denominator away from zero, i.e.,

$$r_{i\leftarrow j}^{(l,l+1)} = \frac{x_i\, w_{ij}}{z_j + \text{sign}(z_j)\,\varepsilon}\, R_j^{l+1}.$$

  This inserted value ε absorbs some of the relevance and can, therefore, be utilized to achieve sparser and less noisy results for the explanation. As ε increases, a greater portion of the relevance is intercepted, sustaining only the most salient relevances for this relevance message.

Both variants have in common that they distribute the upper-layer relevance proportionally downward regarding the local and global pre-activations, i.e., $x_i w_{ij}$ and z_j. Even though the ε-rule avoids division by zero, numerical inconsistencies can occur in both variants for very deep models. Since the pre-activations are not necessarily guaranteed to be positive, the local pre-activations may take on substantial positive or negative values that cancel out in the global pre-activation, leading to magnified values in the preceding layer. As a result,

larger relevances in the lower layers potentially accumulate in deep models and increasingly reach the limits of computational representation of floating point numbers. To prevent this blow-up of relevances, the authors introduced the *α-β-rule*, which treats positive and negative pre-activations separately:

- **The α-β-rule:** The *α-β-rule* was introduced to avoid numerical instabilities and enable a weighting between positive and negative relevances depending on the user's focus. This relevance message applies the simple rule to the positive and negative parts of the pre-activations, respectively, and takes the weighted sum of both. The weighting can be regulated by the hyperparameters $\alpha, \beta \in \mathbb{R}$ satisfying $\alpha + \beta = 1$. Mathematically formulated, the rule is defined as follows:

$$r_{i \leftarrow j}^{(l,l+1)} = \left(\alpha \frac{(x_i \, w_{ij})^+}{z_j^+} + \beta \frac{(x_i \, w_{ij})^-}{z_j^-} \right) R_j^{l+1}$$

$$\text{with} \quad z_j^{\pm} = (b_j)^{\pm} + \sum_k (x_k \, w_{kj})^{\pm}, \quad (\cdot)^+ = \max(\cdot, 0), \quad (\cdot)^- = \min(\cdot, 0).$$

Since the bias vector b_j is included in the computation of the global pre-activations in all presented variants, this term absorbs a certain amount of the upper-layer relevance. Consequently, the *LRP* methods approximate the output prediction rather than providing an accurate representation of the targeted decomposition in Equation 1.

There are even more variants of relevance messages discussed in the literature suitable for various situations or layer types: For example, the *deep Taylor decomposition* (also called *z^+-rule*) in ReLU models – also achieved with the *α-β*-rule with $\alpha = 1$ – allows filtering out only positive relevances (Montavon, Lapuschkin, Binder, Samek, and Müller 2017), or the *γ-rule* favoring positive over negative relevances (Montavon *et al.* 2019). Moreover, some rules are specifically designed for the input layer (Montavon *et al.* 2017). Due to the rule independence of how the lower-layer relevances are computed from the relevance messages in Equation 4, the rules can also be set individually for each layer, called *composite-rule* (Montavon *et al.* 2019; Kohlbrenner, Bauer, Nakajima, Binder, Samek, and Lapuschkin 2020).

A.2. Details on DeepLift

The *deep learning important features (DeepLift)* method introduced by Shrikumar *et al.* (2017a) behaves similarly to *LRP* in a layer-by-layer backpropagation fashion from a selected output node back to the input variables considering the simple rule. However, it incorporates a reference value $\tilde{\boldsymbol{x}}$ to compare the relevances with each other. Hence, the relevances of *DeepLift* represent the relative effect of the outputs of the instance to be explained $f(\boldsymbol{x})_c$ and the output of the reference value $f(\tilde{\boldsymbol{x}})_c$. By taking the difference, the bias term is eliminated in the relevance messages, preventing the relevance absorption and leading to an exact variable-wise decomposition of the difference-from-reference output $\Delta\hat{y}_c = f(\boldsymbol{x})_c - f(\tilde{\boldsymbol{x}})_c$, i.e.,

$$\Delta\hat{y}_c = f(\boldsymbol{x})_c - f(\tilde{\boldsymbol{x}})_c = \sum_{i=1}^{d} R_i^c.$$

In contrast to the *LRP* method, *DeepLift* defines a multiplier layer by layer, starting from the output layer and propagating to the input layer instead of directly determining the relevances

in each intermediate stage. Based on these multipliers, the contribution of an arbitrary variable to the difference-from-reference output can be obtained by multiplying it by the corresponding difference-from-reference input. For an arbitrary layer with the layer's input $\boldsymbol{x}$, reference input $\tilde{\boldsymbol{x}}$ and multiplier $m_{\Delta \boldsymbol{x} \Delta \hat{y}_c}$, this means:

$$\sum_i m_{\Delta x_i \Delta \hat{y}_c} \left(x_i - \tilde{x}_i\right) = m_{\Delta \boldsymbol{x} \Delta \hat{y}_c} \cdot (\Delta \boldsymbol{x})^\top = \Delta \hat{y}_c. \tag{5}$$

The multipliers fulfill a chain rule allowing the computation of the multiplier for the preceding layer given the already calculated one $m_{\Delta \boldsymbol{t}, \Delta \hat{y}_c}$, i.e.,

$$m_{\Delta x_i \Delta \hat{y}_c} = \sum_j m_{\Delta x_i \Delta t_j} \, m_{\Delta t_j \Delta \hat{y}_c}. \tag{6}$$

In other words, the chain rule justifies defining the multipliers for each layer or part of a layer separately before combining them with the upper-layer multipliers. The authors distinguish between the linear and nonlinear components of a layer and provide definitions of the multipliers for each of them:

- **Linear rule:** For the linear components of a layer, such as matrix multiplication in dense or convolution layers, the weights of the corresponding layer are used as the multipliers, i.e., $m_{\Delta x_i \Delta z_j} = w_{ij}$.
- **Rescale rule:** This rule can be used for all nonlinear parts of a layer that can be reduced to a one-dimensional function σ, e.g., all activations such as ReLU, tanh, or sigmoid. In this case, the ratio between the difference-from-reference activation $\Delta\sigma(z)_j = \sigma(z_j) - \sigma(\tilde{z}_j)$ and the pre-activation $\Delta z_j = z_j - \tilde{z}_j$ gives the multiplier, i.e., $m_{\Delta z_j \Delta \sigma(z)_j} = \frac{\Delta \sigma(z)_j}{\Delta z_j}$. To avoid numerical instability caused by a vanishing denominator, the gradient of σ at z_j is used instead of the multiplier when z_j is close to its reference value $\tilde{z}_j$.
- **RevealCancel rule:** This rule is designed for non-linearities σ to propagate meaningful relevances for saturated activations and discontinuous gradients through the layers' activation part, even when activations like ReLU eliminate the values. Similar to the normal pre-activations in the α-β-rule for *LRP*, the positive Δz_j^+ and negative Δz_j^- difference-from-reference pre-activations are considered separately, ensuring the propagation of expressive contribution scores. Descriptively, the *RevealCancel* rule can be explained in a way that the multiplier for the positive part $m_{\Delta z_j^+ \Delta y_j^+}$ is the ratio between the average effect of Δz_j^+ after activating, before and after the negative part Δz_j^- has been added, and the positive difference-from-reference pre-activation Δz_j^+. In the same way, the negative multiplier $m_{\Delta z_j^- \Delta y_j^-}$ is given by the ratio of the average impact of Δz_j^- after activating, before and after the positive part Δz_j^+ has been added, to Δz_j^-. Mathematically, the rule is defined as

$$m_{\Delta z_j^\pm \Delta y_j^\pm} = \frac{\frac{1}{2}\left(\sigma(\tilde{z}_j + \Delta z_j^\pm) - \sigma(\tilde{z}_j) + \sigma(\tilde{z}_j + \Delta z_j^\pm + \Delta z_j^\mp) - \sigma(\tilde{z}_j + \Delta z_j^\pm)\right)}{\Delta z_j^\pm}.$$

These rules, along with the chain rule (Equations 5 and 6), enable the successive computation of the input variables' contributions R_i to the difference-from-reference output $\Delta \hat{y}_c$ in a single backward pass.

B. Remarks on accepted models and layers

As described in Section 3.1, conversion functions are only provided by default for the packages **torch**, **keras**, and **neuralnet** to transfer a neural network into a list structure, which is explained in more detail in the next paragraph. However, any model in this list format can be directly passed to the `Converter`. When initializing the `Converter` object, a **torch**-based copy of the model is then generated from this list. This is one of the most crucial differences compared to packages like **captum**, **zennit**, and **shap**, as these packages override the automatic differentiation graph in the backward pass of the method through so-called *backward hooks*.[6] Consequently, there is no need to analyze and copy the computational graph of the neural network; instead, adjusting the gradient computation of the relevant layers in the backward pass is sufficient. This feature is not available in version 0.12.0 of **torch** in R, which is why we opted for copying the layers as in the packages **innvestigate** and **deeplift** and extended it to a deep-learning-model-agnostic approach. However, this advantage comes with the requirement that all calculations in the network need to be registered and transferred to **torch**. For example, the gradient function does not need to be overridden and registered for a flatten layer in **captum**, **zennit** and **shap** – as it only rearranges the values – aligning with the existing automatic differentiation function. In **innsight**, the entire model is copied, so an equivalent in **torch** must be created for this layer as well. In general, every model from the **neuralnet** and **keras** packages can be converted. In the case of **keras**, this includes both sequential models created with `keras_model_sequential()` and non-sequential models created with `keras_model()`, as long as they only include the accepted layers listed in Table 2. Since it is not possible to reconstruct a computational graph for **torch** models, only `nn_sequential()` models are accepted, and only the layers listed in Table 2 are recognized.

Internally, a model is being transferred from the packages **keras**, **torch**, and **neuralnet** into a list, from which a **torch**-based model is subsequently created. This list requires the entries `"input_dim"` representing the input dimension excluding the batch dimension, `"layers"` for the model's layers (again a list), and `"input_nodes"`/`"output_nodes"` for the indices of the model's input and output layers from `"layers"`. Additionally, input and output labels can be specified using the entries `"input_names"` and `"output_names"`. The input and output names are identical to the fields in the converter object, but when passing a model, they are optional and will be automatically filled otherwise. They always represent a list for each input or output layer, and then contain a list of names for each dimension. For example, `list(list("a", "b", "c"), list("1", "2", "3"))` is valid as input names for a model with one input layer that expects a two-dimensional input of shape 3×3. The entry `"layers"` again forms a list of individual layers of the neural network. For each layer, the `"type"` entry specifies the layer type, and, depending on the type, other relevant components of the layer. For example, a dense layer (`type = "Dense"`) has the entries `"weight"` for the weight matrix and `"bias"` for the bias vector. In addition to these layer-specific entries, each layer has entries `"input_layers"` and `"output_layers"`, indicating the indices in `"layers"` of the incoming and outgoing layers. All available layer types are listed in Table 2. For a more detailed description of the entries and the requirements for the other layer types, please refer to the vignette "In-depth explanation" (see `vignette("detailed_overview", package = "innsight")` or the online documentation at https://bips-hb.github.io/innsight/

[6] See, e.g., `register_full_backward_hook()` in **PyTorch** (https://pytorch.org/docs/stable/generated/torch.nn.Module.html#torch.nn.Module.register_full_backward_hook) or `tf.RegisterGradient` in **TensorFlow**/**Keras** (https://www.tensorflow.org/api_docs/python/tf/RegisterGradient).

	Package		
	keras	**torch**	as `list` (`type =`)
Dense	`layer_dense()`	`nn_linear()`	`"Dense"`
Convolution	`layer_conv_1d()` `layer_conv_2d()`	`nn_conv1d()` `nn_conv2d()`	`"Conv1D"` `"Conv2D"`

Pooling	`layer_max_pooling_1d()`
	`layer_max_pooling_2d()`
	`layer_average_pooling_1d()`
	`layer_average_pooling_2d()`
	`layer_max_pooling_1d()`
	`layer_max_pooling_2d()`
	`layer_average_pooling_1d()`
	`layer_average_pooling_2d()`
		`nn_max_pool1d()`
		`nn_max_pool2d()`
		`nn_avg_pool1d()`
		`nn_avg_pool2d()`
			`"MaxPooling1D"`
			`"MaxPooling2D"`
			`"AveragePooling1D"`
			`"AveragePooling2D"`
			`"GlobalPooling"`

Batch Normalization	`layer_batch_normalization()`	`nn_batch_norm1d()` `nn_batch_norm2d()`	`"BatchNorm"`

Activation	`layer_activation_relu()`
	`layer_activation_leaky_relu()`
	`layer_activation_softmax()`
	`layer_activation()` with
	`"relu"`, `"softplus"`,
	`"sigmoid"`, `"softmax"`, `"tanh"`
		`nn_relu()`
		`nn_leaky_relu()`
		`nn_softplus()`
		`nn_sigmoid()`
		`nn_softmax()`
		`nn_tanh()`
			`"Activation"`
			with `"relu"`, `"softplus"`,
			`"sigmoid"`, `"softmax"`,
			`"tanh"`, `"linear"`

Other	`layer_input()`
	`layer_flatten()`
	`layer_add()`
	`layer_concatenate()`
	`layer_zero_padding_1d()`
	`layer_zero_padding_2d()`
	`layer_dropout()`
		`nn_flatten()`
		`nn_dropout()`
			`"Flatten"`
			`"Add"`
			`"Concatenate"`
			`"Padding"`

Table 2: Summary of the accepted layer types for the packages **keras** and **torch**, as well as the types for the layers provided as a list object.

articles/detailed_overview.html). For the model trained on the bike sharing dataset in Section 3, the argument `save_model_as_list` in the `convert()` function can be used to exemplify the list structure for a dense model:

```
R> conv <- convert(model,
+    output_names = c("Number of rented bikes/10,000"),
+    save_model_as_list = TRUE)
R> str(conv$model_as_list, max.level = 3)

List of 7
 $ layers      :List of 2
  ..$ Dense_1:List of 8
  .. ..$ type           : chr "Dense"
  .. ..$ weight         :Float [1:64, 1:5]
  .. ..$ bias           :Float [1:64]
  .. ..$ activation_name: chr "logistic"
  .. ..$ dim_in         : int 5
```

```
 .. ..$ dim_out        : int 64
 .. ..$ input_layers   : num 0
 .. ..$ output_layers  : num 2
 ..$ Dense_2:List of 8
 .. ..$ type           : chr "Dense"
 .. ..$ weight         :Float [1:1, 1:64]
 .. ..$ bias           :Float [1:1]
 .. ..$ activation_name: chr "linear"
 .. ..$ dim_in         : int 64
 .. ..$ dim_out        : int 1
 .. ..$ input_layers   : num 1
 .. ..$ output_layers  : num -1
$ input_dim   :List of 1
 ..$ : int 5
$ output_dim  :List of 1
 ..$ : int 1
$ input_names :List of 1
 ..$ :List of 1
 .. ..$ : Factor w/ 5 levels "holiday","workingday",..: 1 2 3 4 5
$ output_names:List of 1
 ..$ :List of 1
 .. ..$ : Factor w/ 1 level "Number of rented bikes/10,000": 1
$ input_nodes : num 1
$ output_nodes: int 2
```

C. Advanced visualization

In Sections 3.3 the basic `plot()` and `plot_global()` functions have already been explained. As mentioned there, these functions create either an object of the S4 class '`innsight_ggplot2`' (if `as_plotly = FALSE`) or one of the S4 class '`innsight_plotly`' (if `as_plotly = TRUE`). These functions are intended to generalize the usual **ggplot2**, or **plotly** objects since, with these packages, the limits of clear visualization possibilities for models with multiple input layers are quickly reached. For example, two charts with different scales for each output node or class need to be generated in a model with images and tabular data as inputs. In this case, a **ggplot2**-based or **plotly**-based plot is generated for each single input instance and output node and then combined into one large visualization using `arrangeGrob()` from **gridExtra** (Auguie 2017) or `subplot()` from **plotly**, respectively. In contrast, the S4 class '`innsight_ggplot2`' behaves as a wrapper for the **ggplot2** object for ordinary models with only one input or output layer. Nevertheless, instances of the '`innsight_ggplot2`' class can be treated and modified as regular **ggplot2** objects providing a **ggplot2**-typical usage by adding, for example, themes, scales, or geometric objects; hence the intermediate step via this class is generally not noticeable to the user. For example, the following code is valid:

```
plot(method) +
  ggplot2::theme_bw() +
  ggplot2::xlab("My new x label") +
```

```
  ggplot2::scale_y_continuous(trans = "pseudo_log") +
  ggplot2::geom_text(ggplot2::aes(label = signif(value))
```

Conveniently, all **ggplot2** objects are based on the same `data.frame`, which is also obtained via the `get_result()` method (see Section 3.3), i.e., the corresponding column names can be used as variables in the **ggplot2** objects, as can be seen in the last line of the code chunk above. For objects of the '`innsight_plotly`' class, the entire plot is always created using the `plotly::subplot()` function. However, this has the consequence that individually assigned modifications are partially overwritten by the grouping, which is why the usual **plotly**-typical adaptations can only be performed after the '`innsight_plotly`' object has been printed and returned by the generic `print()` function for this class, i.e.,

```
print(plot(method, as_plotly = TRUE)) %>%
  plotly::hide_colorbar() %>%
  plotly::layout(xaxis = list(title = "My new x label"))
```

In addition, generic functions for both S4 classes are implemented, which provide a deeper and more detailed examination of an already created plot through indexing or indexed modification. Section 4.2 demonstrates the application and illustration of some of these generic methods using visualized explanations of a model that takes tabular and image data as inputs. However, for a more detailed description and usage of these classes, please refer to the vignette "In-depth explanation" (see `vignette("detailed_overview", package = "innsight")` or the online documentation at https://bips-hb.github.io/innsight/articles/detailed_overview.html).

D. Simulation details

D.1. Validity comparison

In order to verify the correctness of the **innsight** package, a comparative simulation is performed with the reference implementations **zennit**, **captum**, **innvestigate**, **deeplift** and **shap** using convolutional and dense neural networks, for which the basic structure is shown in Figure 12. The dense architecture starts with an input of 10 input variables, then has 64 units in the middle hidden layer and either one or five output units. In addition, either ReLU or hyperbolic tangent is used to consider both unbounded and bounded activation functions. The convolutional architecture starts with inputs of shape $32 \times 32 \times 3$ and a convolutional layer with five filters and a kernel size of 5×5, followed by activation with ReLU or hyperbolic tangent. Models are created with and without a pooling layer, which comes at this point. Average or maximum pooling layers with a kernel size of 3×3 are used. Nevertheless, if no pooling layer is considered, strides of 2×2 are used in the preceding convolutional layer to get a similar number of units after flattening. A dense layer with one or five output nodes follows the flattening. Based on these models and normally distributed dataset with 16 input instances, the following methods are compared:

- Gradient-based: *Gradient*, *Gradient* $\times$ *Input*, *IntegatedGradient* with `n = 20` and with zeros and normally distributed reference values.

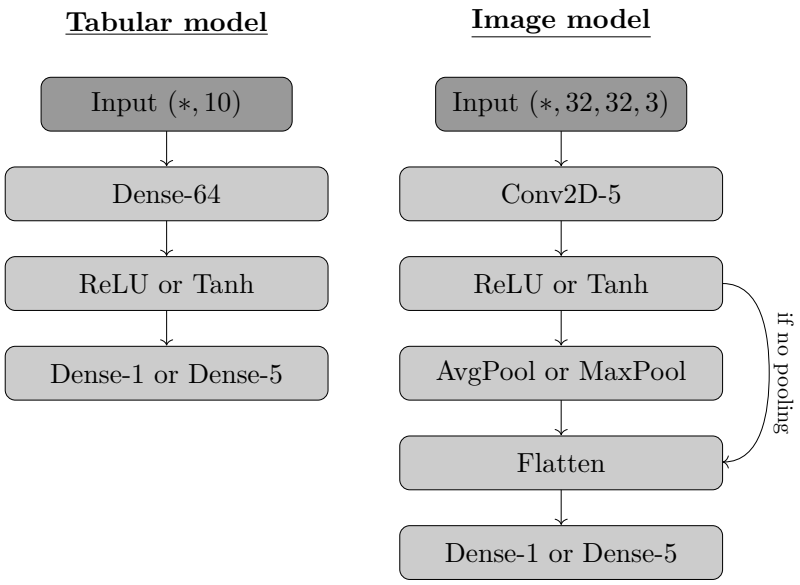


Figure 12: The basic setup of the architectures for the comparison simulation. The expression "-n" indicates the number n of units for dense and the number of filters for convolutional layers.

- *LRP*: simple rule, ε-rule with $\varepsilon = 0.01$, α-β-rule with $\alpha = 1$ and $\alpha = 2$.
- *DeepLift*: *Rescale* and *RevealCancel* rule with zeros and normally distributed reference values each.
- *DeepSHAP* with 32 baseline values.

In this comparison, the methods *SmoothGrad* and *ExpectedGradient* are excluded because they are based on sampling, and consequently, they would only yield the same results with the exact same seed and calculation order. Basically, both methods, however, rely on the gradient calculation of *Gradient* or *IntegratedGradient*. Therefore, an agreement with these methods can also imply the validity of *SmoothGrad* and *ExpectedGradient*.

D.2. Runtime comparison

Besides the consistency of the methods' results, a runtime comparison of basic dense and convolutional models between **innsight** and the reference implementations **zennit**, **captum**, **innvestigate** and **deeplift** is also performed regarding several aspects: The number of output nodes (C), hidden units or filters (U), depth of the model (L), batch size (B), and for images the height/width (W) are varied resulting in the different architectures described in more detail in Figure 13. In the image model, for the 2D convolutional layer, which is repeated $L - 1$ times, a kernel size of 5×5 with default strides of 1×1 and padding is applied so that the output shape corresponds to the input shape. But for the subsequent convolutional layer, a valid padding with strides of $\frac{W-4}{6} \times \frac{W-4}{6}$ is used, producing an equal number of flattened values regardless of the height and width W of the input image. Overall, the time comparison is performed for the following methods, and the average time of 20 replications is calculated:

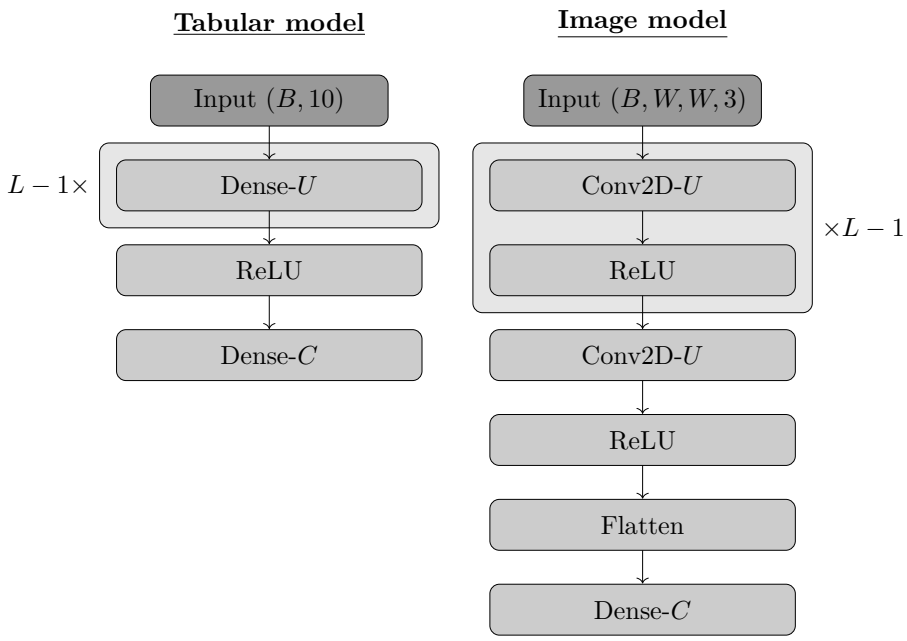


Figure 13: Model architectures for the runtime comparison. The hyperparameters for the number of outputs (C), number of hidden units or filter size (U), number of hidden layers (L), batch size (B), and the height/width of the input images (W) were varied in each case.

- Gradient-based: *Gradient*, *Gradient* $\times$ *Input*, *IntegratedGradient* with `n = 10` and a normally distributed reference value and *ExpectedGradient* with 20 normally distributed reference values and 10 samples.
- *LRP*: simple rule, ε-rule with $\varepsilon = 0.01$, α-β-rule with $\alpha = 2$.
- *DeepLift*: *Rescale* and *RevealCancel* rule with a normally distributed reference value.
- *DeepSHAP* with 20 normally distributed reference values.

In addition to the total time required by a method of one of the considered packages for generating an explanation, the pure execution time is also measured separately. How exactly the time measurement of each method of the packages is accomplished can be found in the reproduction material or in the GitHub repository at `https://github.com/bips-hb/JSS_innsight`.

Number of output nodes (C)

In the time analysis regarding the number of output nodes, 20 dense and image models of each of the architecture shown in Figure 13 are created for $C = 1, 2, 4, 6, \ldots, 20$. The default values are set for the other hyperparameters, i.e., $W = 64$, $U = 128$ for the tabular models, $U = 5$ for image models, $L = 2$, and $B = 16$. Since **innsight** is explicitly designed to analyze multiple output nodes or output classes simultaneously, it is only possible to generate the results in one run with **innsight**. All other implementations are forced to perform a for-loop over the output nodes.

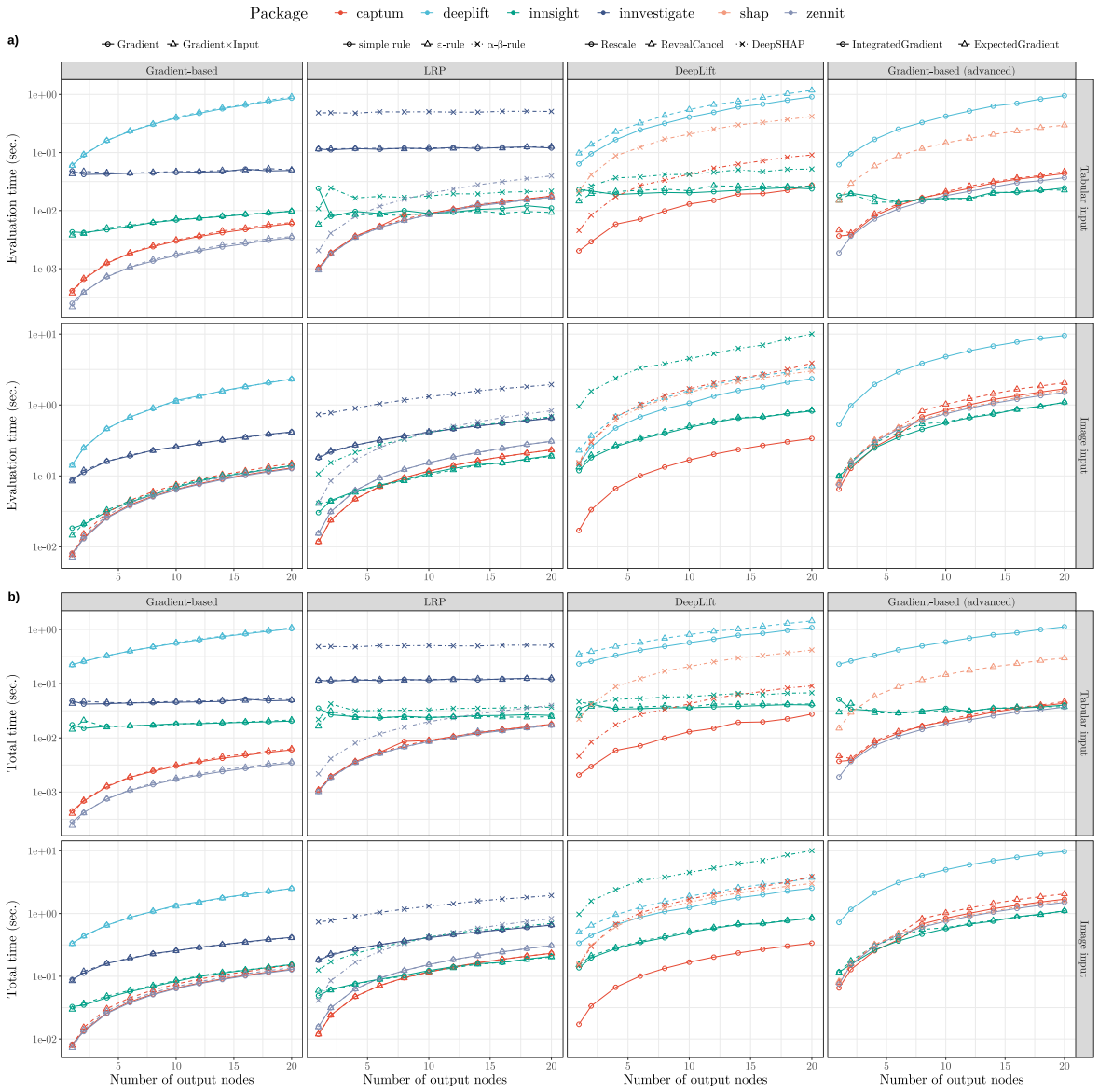


Figure 14: Package's average a) evaluation and b) total runtime in seconds over 20 repetitions for applying different feature attribution methods on models with a varying number of output nodes.

Number of layers (L)

In the time analysis regarding the number of hidden layers, 20 dense and image models of each of the architecture shown in Figure 13 are created for $L = 1, 2, 4, 6, \ldots, 20$. The default values are set for the other hyperparameters, i.e., $C = 1$, $W = 64$, $U = 128$ for the tabular models and $U = 5$ for image models, and $B = 16$.

Number of hidden units/filters (U)

In the time analysis regarding the number of hidden units for dense models and number of

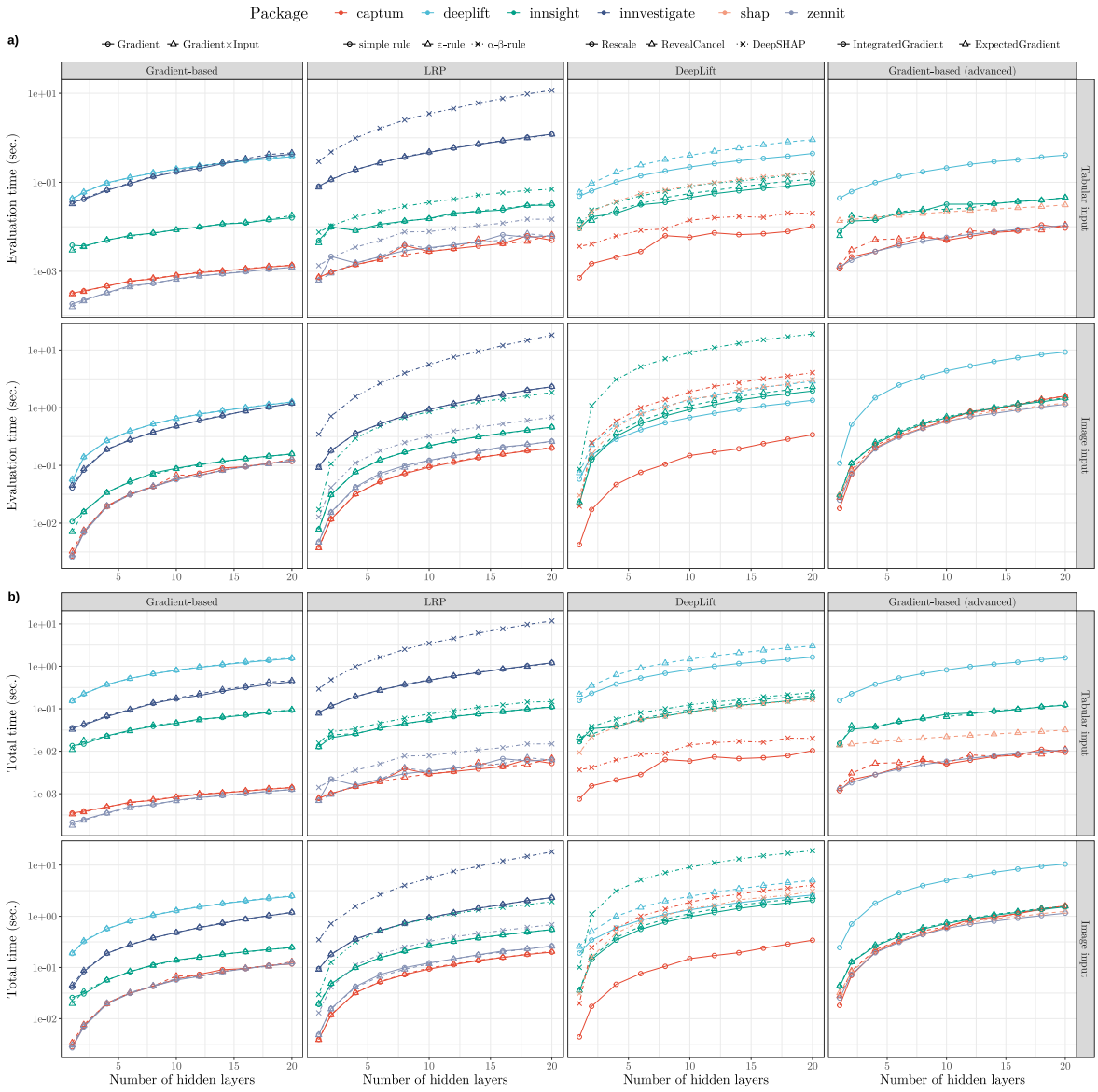


Figure 15: Package's average a) evaluation and b) total runtime in seconds over 20 repetitions for applying different feature attribution methods on models with a varying number of hidden layers.

filters for image models, 20 models each of the architecture shown in Figure 13 are created for $U = 128, 256, 512, 768, \ldots, 2560$ for the tabular and $U = 10, 50, 100, 150, \ldots, 500$ for the image model. The default values are set for the other hyperparameters, i.e., $C = 1$, $W = 64$, $L = 2$, and $B = 16$.

Batch size (B)

In the time analysis regarding the number of input instances, 20 dense and image models of each of the architecture shown in Figure 13 are created for $B = 32, 64, 128, 192, \ldots, 640$ for tabular and $B = 16, 32, 64, 96, \ldots, 320$ for image models. The default values are set for the

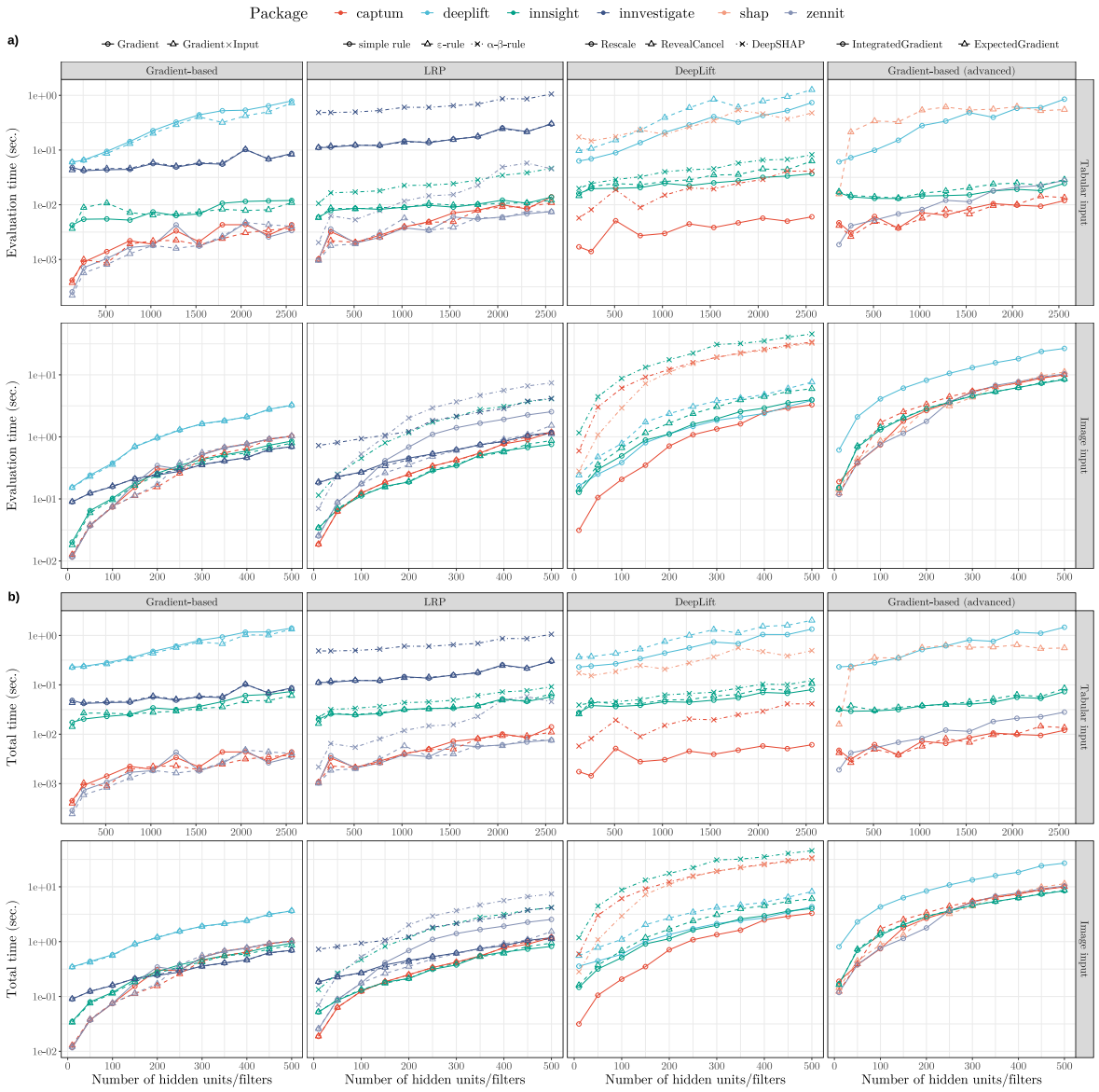


Figure 16: Package's average a) evaluation and b) total runtime in seconds over 20 repetitions for applying different feature attribution methods on models with a varying number of hidden units/filters.

other hyperparameters, i.e., $C = 1$, $W = 64$, $U = 128$ for the tabular models and $U = 5$ for image models, and $L = 2$.

Height and width of the image inputs (W)

In the time analysis regarding the height/width of the image inputs, 20 image models for each of the architecture shown in Figure 13 are created for $W = 16, 32, 64, 96, \ldots, 320$. The default values are set for the other hyperparameters, i.e., $C = 1$, $U = 128$ for the tabular models and $U = 5$ for image models, and $L = 2$.

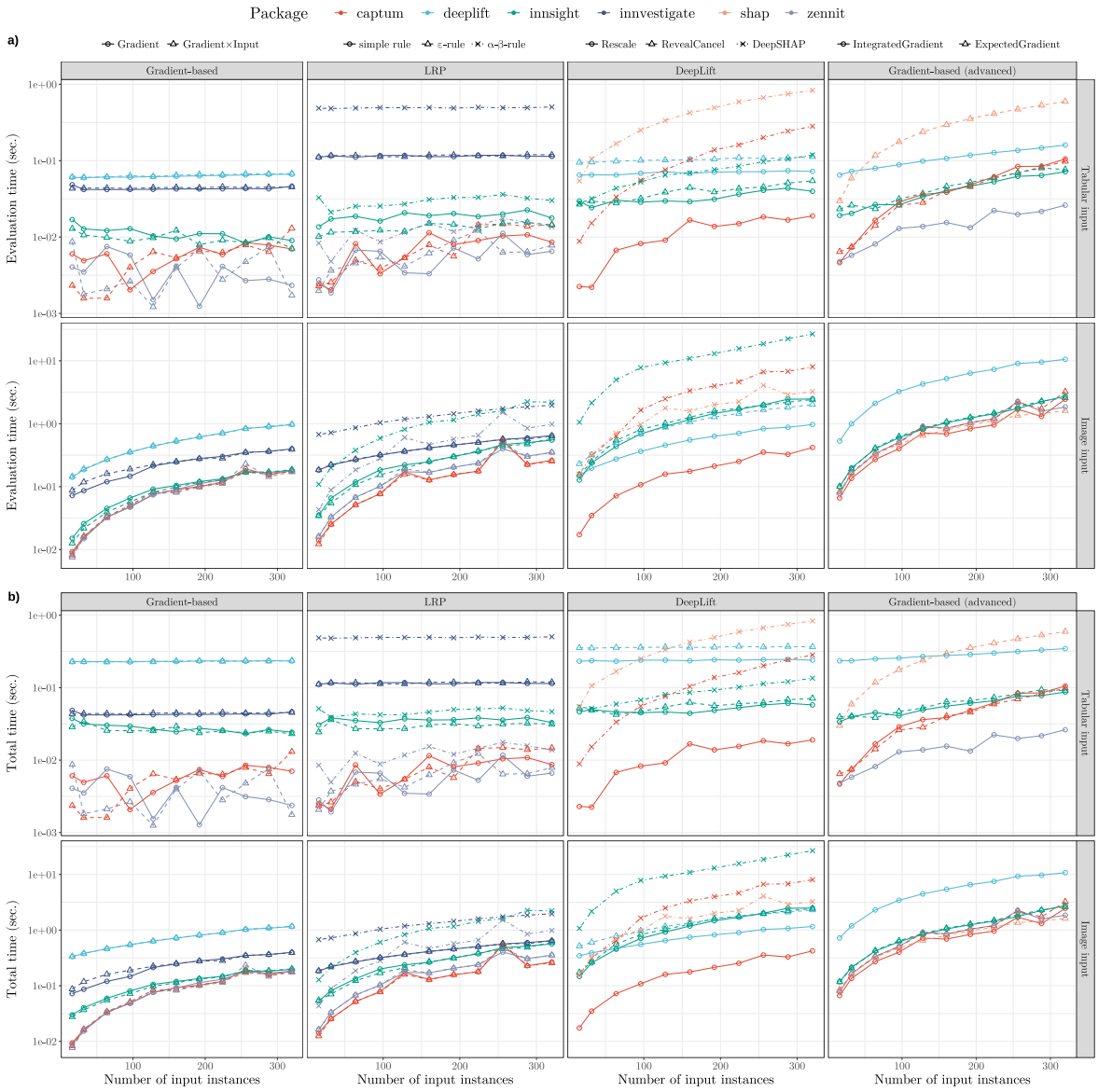


Figure 17: Package's average a) evaluation and b) total runtime in seconds over 20 repetitions for applying different feature attribution methods on models with a varying batch size.

D.3. Additional figures

Results are presented in Figure 14 for a varying number of output nodes layers, in Figure 15 for a varying number of hidden layers, in Figure 16 for a varying number of hidden units/filters, in Figure 17 for varying batch size. Results for varying input image size are presented in Figure 18.

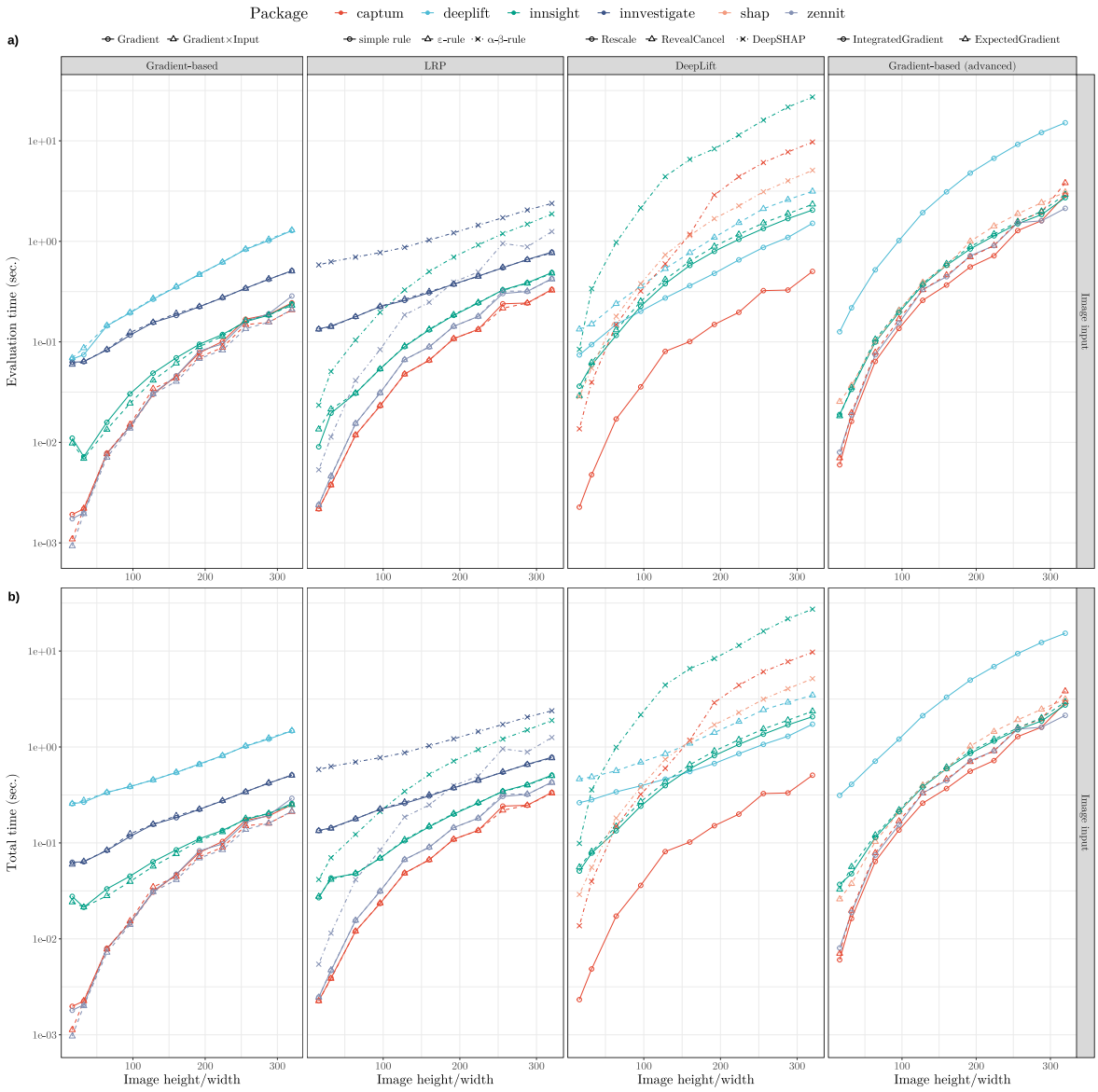


Figure 18: Package's average a) evaluation and b) total runtime in seconds over 20 repetitions for applying different feature attribution methods on models with a varying input image size.

E. *LRP* with bias for innvestigate

As already observed in Section 5, the results of **innsight** and **innvestigate** differ significantly in the *LRP* method with the α-β-rule for models with a bias vector. In the following, this problem is analyzed using a very simple neural network with only one dense layer consisting of one input variable and one output node. This layer has a weight of $w = 1$, a bias vector of $b = -0.25$, and is applied to the input $x = 1$. Mathematically, this results in the following

input relevance for x:

$$\begin{aligned} R_x &= \left(\alpha \frac{(xw)^+}{(xw)^+ + (b)^+} + \beta \frac{(xw)^-}{(xw)^- + (b)^-}\right) \hat{y} \\ &= \left(\alpha \frac{(1)^+}{(1)^+ + (-0.25)^+} + \beta \frac{(1)^-}{(1)^- + (-0.25)^-}\right) 0.75 \\ &= (\alpha \cdot 1 + \beta \cdot 0)\, 0.75 = 0.75\alpha. \end{aligned}$$

This yields, for example, in a relevance of $R_x = 0.75$ for the α-β-rule with $\alpha = 1$ and $R_x = 1.5$ for $\alpha = 2$, which are exactly the values that **innsight** outputs. The package **innvestigate**, on the other hand, outputs relevance 1 and 2, which is probably because the bias vector b is included in the calculation of the positive part despite the negative sign. This short comparison can be reproduced with the reproduction material or in our GitHub repository (`https://github.com/bips-hb/JSS_innsight`) and is based on version 2.0.2 of **innvestigate**.

Affiliation:

Niklas Koenen
Leibniz Institute for Prevention Research and Epidemiology – BIPS
Achterstraße 30
28359 Bremen, Germany
and
Faculty of Mathematics and Computer Science
University of Bremen
E-mail: koenen@leibniz-bips.de

Marvin N. Wright
Leibniz Institute for Prevention Research and Epidemiology – BIPS
Achterstraße 30
28359 Bremen, Germany
and
Faculty of Mathematics and Computer Science
University of Bremen, Germany
and
Section of Biostatistics, Department of Public Health
University of Copenhagen, Denmark
E-mail: wright@leibniz-bips.de

Journal of Statistical Software https://www.jstatsoft.org/
published by the Foundation for Open Access Statistics https://www.foastat.org/
November 2024, Volume 111, Issue 8 *Submitted:* 2023-06-12
doi:10.18637/jss.v111.i08 *Accepted:* 2024-04-30