NEURALGAM: AN R PACKAGE FOR FITTING GENERALIZED ADDITIVE NEURAL NETWORKS

A PREPRINT

Ines Ortega-Fernandez[*]
Galician Research and Development Center in Advanced Telecommunications (Gradiant), Vigo (Spain)
`iortega@gradiant.org`

Marta Sestelo
Galician Centre for Mathematical Research and Technology (CITMAga), Santiago de Compostela (Spain)
Universidade de Vigo, Department of Statistics and O.R. & SiDOR Group, 36310 Vigo (Spain)
`sestelo@uvigo.gal`

ABSTRACT

Nowadays, Neural Networks are considered one of the most effective methods for various tasks such as anomaly detection, computer-aided disease detection, or natural language processing. However, these networks suffer from the "black-box" problem which makes it difficult to understand how they make decisions. In order to solve this issue, an R package called **neuralGAM** is introduced. This package implements a Neural Network topology based on Generalized Additive Models, allowing to fit an independent Neural Network to estimate the contribution of each feature to the output variable, yielding a highly accurate and interpretable Deep Learning model. The **neuralGAM** package provides a flexible framework for training Generalized Additive Neural Networks, which does not impose any restrictions on the Neural Network architecture. We illustrate the use of the **neuralGAM** package in both synthetic and real data examples.

Keywords Explanaible AI · Generalized additive models · Interpetable deep learning

1 Introduction

Neural Networks (NN) are currently among the most widely used predictive modeling techniques, demonstrating superior performance across a broad range of tasks such as image recognition, anomaly detection, and natural language processing [Goodfellow et al., 2016]. Despite their success, a well-known shortcoming of Neural Networks is that they often function as "black-box" models. In most practical applications, it can be challenging to understand precisely *how* the network processes information and reaches a particular prediction [Szegedy et al., 2013].

In recent years, there has been a growing research focus on enhancing trust in AI systems by improving their interpretability and explainability. Broadly, interpretability methods can be categorized into *post-hoc* and *ante-hoc* approaches [Došilović et al., 2018]. *Post-hoc* methods aim to explain a trained black-box model (like a typical Neural Network) using an external, intrinsically interpretable model ("white-box"), whose role is to approximate or interpret the black-box decisions. Examples of these methods include LIME [Ribeiro et al., 2016a], Anchor-LIME [Ribeiro et al., 2016b], and SHAP [Lundberg and Lee, 2017]. While such methods can give valuable insights, they may fail to deliver fully global explanations and often depend on the set of hyperparameters chosen during training, sometimes yielding inconsistent explanations across similar inputs.

In contrast, *ante-hoc* methods aim to build and train inherently interpretable ("white-box") models that balance high predictive accuracy with transparency. The **neuralGAM** package follows an *ante-hoc* approach by implementing a

---

Neural Network-based Generalized Additive Model (GAM) yielding a Generalized Additive Neural Network (GANN). Specifically, **neuralGAM** implements the framework proposed in Ortega-Fernandez et al. [2023], using an ensemble of independent Neural Networks, each responsible for learning a single feature's contribution to the response variable, thereby achieving both high accuracy and interpretability. We use backpropagation to estimate each feature network, and the GAM model is fitted using local scoring and backfitting algorithms to ensure that it converges and is additive.

neuralGAM is, therefore, a "white-box" Deep Learning model that provides an exact description of how each covariate affects the response variable. The partial effects of each learned function can be visualized independently, providing information on whether the relationship between the response variable and each covariate is linear, monotonic, or complex. This is not a typical feature of "black-box" Neural Networks, which are hard to interpret. These characteristics make **neuralGAM** suitable for high-risk AI applications such as medical decision-making. Furthermore, by observing the partial effects, **neuralGAM** can be used to identify and mitigate bias introduced in the training data set. For example, partial functions with a positive slope can increase the probability of observing a particular class in binary classification problems. To the best of our knowledge, we are the first to implement a GAM using independent deep Neural Networks to estimate the smooth functions. The development of the package has been motivated by recent contributions in the development of Generalized Additive Neural Networks (GANN), in particular, the method proposed by Ortega-Fernandez et al. [2023] to train a GANN using independent Neural Networks and the local scoring and backfitting algorithms.

In recent years, several implementations of Generalized Additive Neural Networks (GANNs) have emerged, aiming to combine the interpretability of GAMs with the flexibility of Neural Networks. Early GANNs, such as the model by Potts [1999], used univariate multi-layer perceptrons without backpropagation and required manual tuning. Subsequent approaches, like that of Brás-Geraldes et al. [2019], improved flexibility with parametric link functions and bootstrap-based confidence intervals but remained constrained to shallow architectures. More recent methods take advantage of Deep Learning techniques: Neural Additive Models (NAMs) Agarwal et al. [2021] train feature-specific subnetworks jointly with backpropagation but can suffer from overfitting due to sharp activation functions. GAMI-Net Yang et al. [2021] extends this by allowing pairwise interactions and incorporating sparsity and marginal clarity constraints, but with an increased computational cost. At last, IGANN Kraus et al. [2023] introduces a GANN based on gradient boosting techniques with sparsified shallow networks, assuming initially linear effects and avoiding deep architectures.

Among these GANN models, **neuralGAM** is the only available implementation in R that leverages fully independent deep Neural Networks per feature and fits a true additive model via local scoring and backfitting, offering both interpretability and deep learning flexibility within the R ecosystem.

In order to streamline this task, it is imperative to design software that can effectively implement the proposed methods in a researcher-friendly and comprehensible environment. Our package fulfills this objective by offering a range of user-friendly functions. The package **neuralGAM** is freely available from the Comprehensive R Archive Network (CRAN) at `https://cran.r-project.org/package=neuralGAM`. The recent surge in the adoption of Deep Learning has led to a remarkable expansion of R Deep Learning packages in the CRAN. As per the Machine Learning and Statistical Learning CRAN task view, a comprehensive inventory of such packages is presented below.

Single hidden layer (shallow) Neural Networks are implemented in R in the **nnet** [Venables and Ripley, 2002], which is shipped with base R. Deep learning flavors of Neural Networks are implemented on the **deepnet** [Rong and Rong, 2014], **RcppDL** [Kou and Sugomori, 2014], **neuralnet**, and **h2o** [Aiello et al., 2016] packages, among others. Regarding interfaces to high-level APIs, which provide a user-friendly interface to build and train Neural Networks, the most used R packages are **keras** [Chollet et al., 2015] and **tensorflow** [Abadi et al., 2015]. TensorFlow is a powerful and widely used open-source Deep Learning library that provides a comprehensive set of tools and resources for building Neural Networks. In R, TensorFlow offers a flexible and intuitive API, enabling users to construct custom deep-learning models effortlessly. It supports dynamic computation graphs, allowing for efficient training and deployment of models. In 2020 the **torch** [Falbel and Luraschi, 2023] package for R was announced, which uses a C++ backend instead of Python, and R6 to provide object-oriented capabilities, which might be unfamiliar to R users. Keras is a user-friendly high-level Deep Learning package that can be utilized to build and train Neural Networks. It provides a wide range of pre-built models and supports various network architectures. Its integration with TensorFlow (and Python) as the backend ensures efficient computation and seamless interoperability.

Regarding R software which provides explainability capabilities to AI models, **NeuralNetTools** [Beck, 2018] can be used to interpret supervised Neural Network models that are created in R. The functions in the package can be used to visualize the model with the help of a Neural Network interpretation diagram, analyze the importance of variables by breaking down the model weights, and conduct a sensitivity analysis of the response variables to changes in the input variables. It provides functions to plot Neural Network architectures, weight histograms, and activation functions, allowing visualization of the connectivity and structure of Neural Networks, which can aid in understanding and debugging complex models. **iml** [Molnar et al., 2018] provides a set of tools for interpreting machine learning

models, but focusing on local interpretation of individual predictions rather than global model explanations, including features such as feature importance measures with permutation-based variable importance, partial dependence plots, and accumulated local effect plots. It also includes functions to explain model predictions using Shapley values. **DALEX** [Biecek, 2018] (Descriptive mAchine Learning EXplanations) provides model-agnostic explanation and visualization for machine learning models using explainers such as LIME, Break Down, or Shapley values. The package offers both local and global interpretability, enabling users to explore model behavior at different levels of granularity. While **iml** and **DALEX** are model-agnostic, **NeuralNetTools** is specifically designed for Neural Networks.

With respect to other major statistical software for fitting GAMs, **mgcv** is the main GAM package in R which implements a penalized likelihood approach to smoothing (e.g. using splines). It supports a wide range of distributions with automatic smoothness selection using generalized cross-validation (GCV) or restricted maximum likelihood (REML). While both **mgcv** and **neuralGAM** implement additive models, their core implementations and use-cases differ substantially. In particular, while **mgcv** constructs smooth terms using predefined basis functions (e.g., splines) and optimizes smoothness via GCV or REML, **neuralGAM** focuses on the use of stochastic gradient descent using backpropagation to learn smooth functions directly from data using independent subnetworks trained with backpropagation. The neural network approach of **neuralGAM** allows for automatic feature learning, scalability to high-dimensional or noisy data, and potential integration with complex deep learning architectures such as CNNs or embeddings.

In this paper, we explain and illustrate how Neural Networks can be used to fit Generalized Additive Models using the **neuralGAM** package via a simulated scenario with Gaussian response, and a real-life application related to flight delay prediction based on weather and flight condition's data.

The remainder of the paper is structured as follows: in Section 2 we briefly review the estimation procedures and explain the use of the main functions and methods of **neuralGAM**; Section 3 gives an illustration of the practical application of the package using simulated and real data; and finally, Section 4 concludes with a discussion and possible future extensions of the package.

2 Models and software

2.1 An overview of the methodology

The most common way to model the relationship between the response variable and the covariates is by using the multiple linear regression model, where the response variable (Y) is assumed to be normally distributed, and the covariates (X_j, $j = 1, \ldots, p$) are assumed to have a linear effect on the response. However, the response variable may not be normally distributed and, in such cases, Generalized Linear Models [Nelder and Wedderburn, 1972] allow the use of other distribution families, e.g. Binomial, Poisson, etc. Additionally, in some cases, the assumption of linearity in the effects of covariates can be too restrictive, and not supported by the available data. In this setting, nonparametric regression techniques emerge, allowing us to model this dependence between the response and the covariates without specifying in advance the function that links them. This leads to the Generalized Additive Models [Hastie and Tibshirani, 1990] defined by

$$E[Y \mid \mathbf{X}] = m(\mathbf{X}) = h^{-1}(\alpha + \sum_{j=1}^{p} f_j(X_j)), \tag{1}$$

where $h(\cdot)$ is a monotonic known function (the link function) and $f_1, ..., f_p$ are smooth and unknown functions. To ensure the identifiability of the model (the ability to uniquely determine the parameters of the model based on the observed data), a constant denoted by α is introduced in the model and the partial functions must satisfy the condition $E[f_j(X_j)] = 0$ where $j = 1, \ldots, p$. This implies that $E[Y] = \alpha$, a crucial condition to ensure that the model's predictions remain unchanged even if we add a constant value to f_1 while subtracting the same constant value from f_2 [Hastie and Tibshirani, 1990].

As we mentioned, GAMs are widely used for their ability to capture complex relationships between covariates and response variables without requiring the shape of these relationships to be specified in advance. This flexibility makes GAMs a powerful and effective class of regression models. They also offer the advantage of being fully nonparametric while allowing certain covariates—such as categorical features—to enter the model linearly, resulting in a semi-parametric framework. Thanks to this versatility, GAMs are particularly well-suited for tackling real-world problems.

To fit the previous model in (1), from an independent random sample $\{\mathbf{X}_i, Y_i\}_{i=1}^{n}$, with $\mathbf{X} = X_1, \ldots, X_p$, we use a combination of the local scoring algorithm (see Algorithm 1) and the backfitting algorithm (see Algorithm 2). Particularly, the estimation of the additive predictor $\eta = \alpha + \sum_{j=1}^{p} f_j(\cdot)$ is obtained by fitting a weighted additive

model using Neural Networks as function estimators, yielding a Generalized Additive Neural Network. For each f_j, the backfitting algorithm iteratively estimates the effect of each covariate $X_1, \ldots, X_p$, with weights W_i, on the adjusted dependent variable Z_i, updated at each step of the local scoring algorithm.

Distribution	Link	Z_i	W_i	$DEV_i(Y_i, \hat{\mu}_i)$
Gaussian	identity	Y_i	1	$(Y_i - \hat{\mu}_i)^2$
Binomial (s, μ)	logit	$\eta_i + (Y_i - \mu_i)s\mu_i(1 - \mu_i)$	$s\mu_i(1 - \mu_i)$	$-2(Y_i \log \hat{\mu}_i + (1 - Y_i) \log(1 - \hat{\mu}_i)$

Table 1: Adjusted dependent variable Z_i, weights W_i, and deviance $DEV_i(Y_i, \hat{\mu}_i)$, at the local scoring algorithm for Gaussian and binomial distribution [Hastie and Tibshirani, 1990].

Algorithm 1: Local scoring algorithm

Initialise $\hat{\alpha} = h(\overline{Y_i})$, $\hat{f}_j^0(\cdot) = 0 \, \forall \, j$, $l = 0$

for $l \leftarrow l + 1$ **do**

> Construct an adjusted dependent variable Z_i according to Table 1 with
> $\hat{\eta}_i^{l-1} = \hat{\alpha} + \sum_{j=1}^p \hat{f}_j^{l-1}(X_{ij})$ and
> $\hat{\mu}_i^{l-1} = h^{-1}(\eta_i^{l-1})$
>
> Form the weights W_i according to Table 1.
>
> Fit a weighted additive model to Z_i using the backfitting algorithm (Algorithm 2) with weights W_i to obtain the estimated functions $\hat{f}_j^l$, additive predictor η_i^l and fitted values $\hat{\mu}_i^l$

end

Compute the convergence criterion DEV $< \delta$, with $DEV = \frac{\sum_{i=1}^n DEV_i(Y_i, \hat{\mu}_i^{l-1}) - DEV_i(Y_i, \hat{\mu}_i^l)}{\sum_{j=1}^n DEV_i(Y_i, \hat{\mu}_i^{l-1})}$ according to Table

1, where δ is a small threshold

Until the convergence criterion is satisfied

Note that if the link function is the identity and the error distribution is Gaussian, then $Z_i = Y_i$ and the weights do not change, thus the procedure is simply an additive fit. In any other case, the dependent variable Z_i and the weights W_i are updated at each iteration l of the local scoring algorithm. This process is repeated until the convergence criterion is satisfied (see last step of Algorithm 1). Note that we use the deviance because it is an appropriate measure of discrepancy between observed and fitted values.

Algorithm 2: Backfitting Algorithm with Neural Networks

Initialise $\hat{f}_j^0(\cdot) = 0 \, \forall \, j$, $m = 0$
Iterate $m \leftarrow m + 1$

for $j \leftarrow 1, p$ **do**

> Compute partial residuals $R_{ij} = Z_i - \hat{\alpha} - \sum_{k=1}^{j-1} \hat{f}_k^{m-1}(X_{ik}) - \sum_{k=j+1}^p \hat{f}_k^{m-1}(X_{ik})$
>
> Fit a Neural Network with R_{ij} and X_{ij} to obtain $\hat{f}_j^m$
>
> Replace $\hat{f}_j^m$ with its centered version $\hat{f}_j^m(\cdot) - \frac{\sum_{i=1}^n \hat{f}_j^m(X_{ij})}{n}$

end
Until $\frac{\sum_{j=1}^p \sum_{i=1}^n (\hat{f}_j^m(X_{ij}) - \hat{f}_j^{m-1}(X_{ij}))^2}{\sum_{j=1}^p \sum_{i=1}^n (\hat{f}_j^{m-1}(X_{ij}))^2} < \epsilon$

Particularly, given the fitted mean response $\hat{\mu}_i = \hat{E}[Y_i \mid \mathbf{X}_i]$, the deviance is defined as $DEV = \frac{\sum_{i=1}^n DEV_i(Y_i, \hat{\mu}_i^{l-1}) - DEV_i(Y_i, \hat{\mu}_i^l)}{\sum_{j=1}^n DEV_i(Y_i, \hat{\mu}_i^{l-1})}$, with DEV$_i$ depending on the link (see Table 1). Several approaches have been described in the literature to estimate the regression model in (1), such as Bayesian approaches [Lang and Brezger,

2004], local polynomial kernel smoothers [Wand and Jones, 1994, Copeland, 1997] or regression splines [De Boor et al., 1978]. Unlike these quite common techniques, we propose to use independent Neural Networks —which are universal function estimators [Hornik et al., 1989]— to learn the contribution of each covariate to the dependent variable, i.e., at each iteration of the backfitting algorithm, we train a feed-forward Neural Network for each covariate X_j with the entire set of training data for one epoch. The fits are improved at each epoch as the learned adjusted dependent variable Z_i approaches Y_i at each iteration of the local scoring algorithm. Algorithms 1 and 2 summarize the core idea.

2.2 Package structure and functionality

The **neuralGAM** package introduces a new methodology for fitting Generalized Additive Models, with Gaussian and binary responses, based on Neural Network techniques, and it is composed of several functions that enable users to fit the models with the methods described above. **neuralGAM** is implemented using Keras [Chollet et al., 2015], a high-level API for the implementation of Neural Networks well known for its simplicity, flexibility, and ease of use.

Function	Description
`neuralGAM`	Main function to fit a `neuralGAM` model. The function builds one Neural Network to attend to each feature in `data`, using the backfitting and local scoring algorithms to fit a weighted additive model using Neural Networks as function approximators.
`summary.neuralGAM`	Method of the generic summary function for `neuralGAM` objects.
`print.neuralGAM`	Method of the generic print function for `neuralGAM` object, which prints out some key components.
`plot.neuralGAM`	Visualization of `neuralGAM` objects with the generic function for plotting of R objects. Provides default plot showing the smooth and linear components of a fitted `neuralGAM`.
`autoplot.neuralGAM`	Visualization of `neuralGAM` objects with **ggplot2** [Wickham, 2016] graphics. Provides default plot showing the smooth and linear components of a fitted `neuralGAM`.
`predict.neuralGAM`	Takes a `neuralGAM` object produced by `neuralGAM()` and, given a new set of values for the model covariates, produces predictions.
`install_neuralGAM`	Creates a `conda` environment (installing `miniconda` if required) and sets up the Python requirements to run neuralGAM (Tensorflow and Keras).

Table 2: Summary of functions in the **neuralGAM** package.

The package is designed along lines similar to those of other R regression packages. The functions within **neuralGAM** are briefly described in Table 2. The main function of the package is `neuralGAM`, which fits a Generalized Additive Neural Network to estimate the contribution of each smooth function to the response. The arguments of this function are shown in Table 3. Note that through the argument `formula` users can decide to fit a model with smooth (`s(x)`), linear, or factor terms (`x`), and using the argument `family` it is possible to select the conditional distribution of the response variable. So far, the user can select between Gaussian and binomial. Numerical and graphical summaries of the fitted object can be obtained by using the `print`, `summary`, `plot`, and `autoplot` methods implemented for `neuralGAM` objects. Another of these methods is available for the `predict` function which takes a fitted model of the `neuralGAM` class and, given a new data set of values of the covariates by means of the argument `newdata`, produces predictions. At last, we provide a helper function `install_neuralGAM()` to assist the user in installing the required Python dependencies in a custom `conda` environment. Once the dependencies are installed using `install_neuralGAM()`, the user must reload the library again using `library(neuralGAM)` for the changes to take effect. Note that the first time the package is used after installing and setting up the dependencies, the process of loading the required Python packages might take some time.

An example of the use of **neuralGAM**, illustrating all the features provided, is given in the following example of code. **neuralGAM** is configured to fit a Deep Neural Network with 64 units on each of its three deep layers (`num_units`). In this example, the linear predictor is composed of a linear term in x and smooth terms of z and w, and the response is assumed to follow a Gaussian distribution (`family`).

```
neuralGAM(y ~ x + s(z) + s(w), data = data, num_units = c(64, 64, 64),
+     family = "gaussian", learning_rate = 0.001, activation = "relu",
+     kernel_initializer = "glorot_normal", loss = "mse",
+     kernel_regularizer = NULL, bias_regularizer = NULL,
+     bias_initializer = 'zeros', activity_regularizer = NULL,
```

neuralGAM() arguments	
formula	An object of class "formula": a description of the model to be fitted. Smooth terms can be included using `s()`.
data	A data frame containing the model response variable and covariates required by the formula. Additional terms not present in the formula will be ignored.
num_units	Defines the architecture of each Neural Network. If a scalar value is provided, a single hidden layer Neural Network with that number of units is used. If a list of values is provided, a multi-layer Neural Network with each element of the list defining the number of hidden units on each hidden layer is used.
family	This is a family object specifying the distribution and link to use for fitting. By default, it is `"gaussian"` but also works to `"binomial"` for logistic regression.
learning_rate	Learning rate for the Neural Network optimizer.
activation	Activation function to use on every layer of the Neural Network. Defaults to `relu`. See `layer_dense` documentation from the Keras library.
loss	Loss function to use during Neural Network training. Defaults to the Mean Squared Error (`mse`).
kernel_initializer	Kernel initializer for the Dense layers. Defaults to Xavier initializer `glorot_normal` [Glorot and Bengio, 2010].
kernel_regularizer	Optional regularizer function applied to the kernel weights matrix.
bias_regularizer	Optional regularizer function applied to the bias vector.
bias_initializer	Optional initializer for the bias vector.
activity_regularizer	Optional regularizer function applied to the output of the layer.
w_train	Optional sample weights.
bf_threshold	Convergence criterion of the backfitting algorithm. By default it is `0.001`.
ls_threshold	Convergence criterion of the local scoring algorithm. Defaults to `0.1`.
max_iter_backfitting	An integer with the maximum number of iterations of the backfitting algorithm. Defaults to 10.
max_iter_ls	An integer with the maximum number of iterations of the local scoring Algorithm. Defaults to 10.
seed	Specifies the random number generator seed for algorithms dependent on randomization.
verbose	Verbosity mode (0 = silent, 1 = print messages). Defaults to 1.
...	Other arguments to pass on to the Adam optimizer [Kingma and Ba, 2014].

Table 3: Arguments of `neuralGAM()` function.

```
+    bf_threshold = 0.001, ls_threshold = 0.1, max_iter_backfitting = 10,
+    max_iter_ls = 10, seed = NULL, verbose = 0, ...)
```

Other arguments of `neuralGAM` will be familiar to `keras` users: `activation` defines the activation function for the Dense layers of each fitted Neural Network (defaults to `relu`). **neuralGAM** uses Adam [Kingma and Ba, 2014] as an optimizer for stochastic gradient descent, whose behaviour can be customised using the `loss`, `learning_rate`, `kernel_initializer`, `kernel_regularizer`, `bias_regularizer`, `bias_initializer`, and `activity_regularizer` parameters.

The backfitting and local scoring algorithms used to fit the GAM model can be configured using the `bf_threshold`, `ls_threshold` arguments which adjust the convergence criterion of both algorithms, while `max_iter_backfitting` and `max_iter_ls` adjust the maximum number of iterations of each algorithm. Finally, `seed` specifies an optional random number generator seed for algorithms dependent on randomization, `verbose` sets the verbosity of the printed outputs, while the `...` argument allows setting other arguments available for the Adam implementation in `keras` such as the exponential decay rate for the 1st and 2nd-moment estimates.

Creating efficient and appropriate data visualizations becomes increasingly important as the complexity of the data increases. **neuralGAM** provides two different methods for plotting data: one based on R's standard plotting function (`plot.default`) in which the `plot` function inherits all parameters from `graphics` package and can be set as usual, and one based on `ggplot2` [Wickham, 2016]. With this latter plot method, the `autoplot` function creates a `ggplot` object that can be modified later by the user just using the functionality provided in the `ggplot2` package. The following excerpt of code shows analogous plots using the two methods implemented:

```
plot(ngam, main = "My plot")
library(ggplot2)
autoplot(ngam, select = "x1") + ggtitle("My plot")
```

3   Illustrative examples

In this section, the use of the **neuralGAM** package is illustrated using both simulated and real data. We consider examples for each of the conditional distributions of the response variable, i.e., both Gaussian and binary.

3.1   Application to simulated data

This subsection reports the capabilities of the **neuralGAM** package in a simulated scenario. For this scenario, we used a sample size of $n = 30625$ which was split into 80% for training the model and 20% for testing. We consider the following predictor

$$\eta = \alpha + \sum_{j=1}^{3} f_j(X_j),$$

with

$$f_j(X_j) = \begin{cases} X_j^2 & \text{if } j = 1, \\ 2X_j & \text{if } j = 2, \\ \sin X_j & \text{if } j = 3, \end{cases}$$

$\alpha = 2$, and covariates X_j drawn from an uniform distribution $U[-2.5, 2.5]$. The response follows a Gaussian distribution with $Y = \eta + \epsilon$, where ϵ is the error distributed in accordance to a $N(0, \sigma(x))$ in a homoscedastic situation with $\sigma(x) = 0.25$. Additional simulation scenarios including different response types and conditions on the variance of the error term ϵ are available at Ortega-Fernandez et al. [2023].

The code for the generation of this data (along with a summary) can be found below:

```
seed <- 42
set.seed(seed)
n <- 30625
x1 <- runif(n, -2.5, 2.5)
x2 <- runif(n, -2.5, 2.5)
x3 <- runif(n, -2.5, 2.5)
f1 <- x1 ** 2
f2 <- 2 * x2
f3 <- sin(x3)
f1 <- f1 - mean(f1)
f2 <- f2 - mean(f2)
f3 <- f3 - mean(f3)
fs <- data.frame(f1, f2, f3)
eta0 <- 2 + f1 + f2 + f3
epsilon <- rnorm(n, 0.25)
y <- eta0 + epsilon
dat <- data.frame(x1, x2, x3, y)
sample <- sample(c(TRUE, FALSE), n, replace = TRUE, prob = c(0.8, 0.2))
train <- dat[sample,]
test <- dat[!sample,]

names(fs) <- names(dat[,1:3])
fs_train <- fs[sample,]
fs_test <- fs[!sample,]

summary(dat)
```

```
        x1                      x2                      x3                        y
 Min.    :-2.499959    Min.    :-2.49949    Min.    :-2.4994955    Min.    :-8.2672
 1st Qu.:-1.272666    1st Qu.:-1.24730    1st Qu.:-1.2516295    1st Qu.:-0.5044
 Median :-0.004220    Median : 0.01913    Median : 0.0013850    Median : 2.2692
 Mean    :-0.006529    Mean    : 0.01788    Mean    :-0.0004924    Mean    : 2.2574
 3rd Qu.: 1.247740    3rd Qu.: 1.28176    3rd Qu.: 1.2468695    3rd Qu.: 4.8916
 Max.    : 2.499962    Max.    : 2.49988    Max.    : 2.4997349    Max.    :13.5417
```

Once the data have been generated, we can fit the model using the `train` data set. Since the effect of each covariate in the response is *a priori* unknown, we will use smooth effects for all the covariates. We begin by installing the **neuralGAM** package and its Python dependencies with Reticulate using the `install_neuralGAM()` function (we omit the installation logs for simplicity):

```
if (!require("neuralGAM")) install.packages("neuralGAM", quiet = TRUE)
if (!require("ggplot2")) install.packages("ggplot2")
suppressMessages(neuralGAM::install_neuralGAM())
library(neuralGAM)
library(ggplot2)
```

```
Loading required package: neuralGAM
NOTE: conda environment not found... run 'install_neuralGAM()' and load library again...
```

Once the package is installed, we can start the model fitting procedure:

```
ngam <- neuralGAM(y ~ s(x1) + s(x2) + s(x3), data = train,
                  num_units = 1024,
                  learning_rate = 0.001,
                  bf_threshold = 0.001,
                  seed = seed, verbose = 0)
ngam
```

```
Class: neuralGAM

Distribution Family:  gaussian
Formula:  y ~ s(x1) + s(x2) + s(x3)
Intercept: 2.2422
MSE: 1.0311
Sample size: 24522
```

We can observe that the obtained Mean Squared Error (MSE) during training was 1.0311, which reflects a good performance. **neuralGAM** model can be also visualized using the `print`, `summary`, `plot` and `autoplot` functions, while the model components can be extracted using `predict`. In the following example, we use the `autoplot` function to obtain the estimated function for each fitted term. We also include in the plot (in blue) the true shape of each function to verify the ability of **neuralGAM** to learn the true shape of each function:

```
plots <- lapply(c("x1", "x2", "x3"), function(x) {
  autoplot(ngam, select = x) +
  ggplot2::geom_line(aes(x = train[[x]], y = fs_train[[x]]),linetype = 2, colour = "blue")
})
gridExtra::grid.arrange(grobs = plots, ncol = 3, nrow = 1)
```

Figure 1 shows how **neuralGAM** was able to successfully capture both smooth effects in x1 and x3, and a linear effect in x2 using Neural Networks as function estimators. We can observe how the model can properly estimate the true shape of each partial function, showcasing the good performance of the model in this scenario. Note that, given the observed linear effect in x2, one can choose to force a linear fit for this term by modifying the `formula` parameter and fitting the model in a semi-parametric manner.

The `summary` method returns a summary of the fit where it is possible to observe the model architecture for both the Neural Networks and the linear terms (if working in a semi-parametric setting). In this case, for each Neural Network,
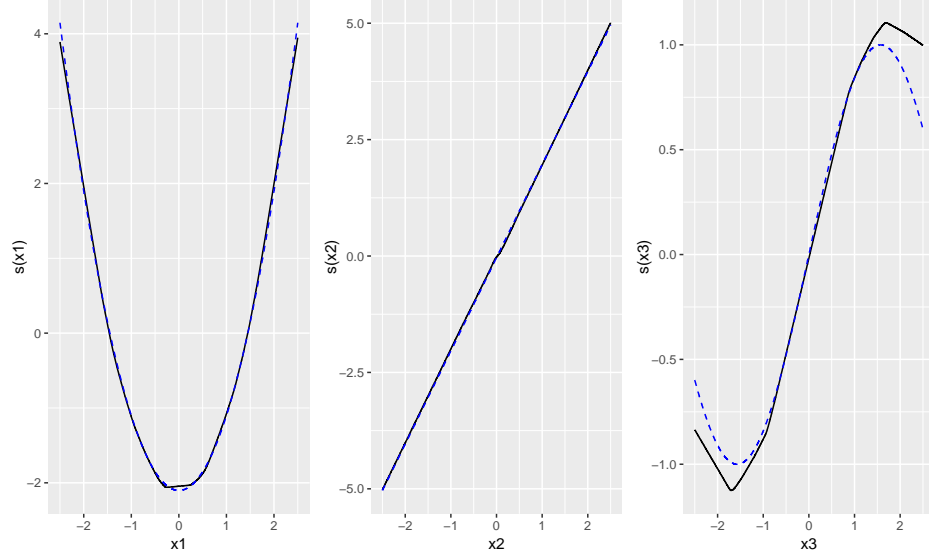
Figure 1: Estimated function (black line) for each fitted term obtained using the `autoplot` function, and true function (blue)

one can observe the type of each layer, its shape, and the number of parameters. Moreover, the function provides a summary of the model training history, including information about the evolution of the model fit for each covariate, allowing the monitoring of both the training loss (in this case, the Mean Squared Error `mse`) and the elapsed time on each training epoch.

```
summary(ngam)

Class: neuralGAM

Distribution Family:  gaussian
Formula:  y ~ s(x1) + s(x2) + s(x3)
Intercept: 2.2422
MSE: 1.0311
Sample size: 24522

Training History:

            Timestamp Model Epoch TrainLoss
1 2025-03-07 11:19:01    x1     1   10.5817
2 2025-03-07 11:19:03    x2     1    1.8857
3 2025-03-07 11:19:04    x3     1    1.1577
4 2025-03-07 11:19:06    x1     2    1.0791
5 2025-03-07 11:19:08    x2     2    1.0576
6 2025-03-07 11:19:10    x3     2    1.0387
7 2025-03-07 11:19:12    x1     3    1.0507
8 2025-03-07 11:19:13    x2     3    1.0458
9 2025-03-07 11:19:14    x3     3    1.0285


Model architecture:

$x1
Model: "x1"

_____
 Layer (type)                    Output Shape                 Param #
======================================================================
 dense (Dense)                   (None, 1)                    2
```

```
 dense_1 (Dense)                        (None, 1024)                    2048
 dense_2 (Dense)                        (None, 1)                       1025
================================================================================
Total params: 3075 (12.01 KB)
Trainable params: 3075 (12.01 KB)
Non-trainable params: 0 (0.00 Byte)

--------------------------------------------------------------------------------


$x2
Model: "x2"

--------------------------------------------------------------------------------
 Layer (type)                           Output Shape                    Param #
================================================================================
 dense_3 (Dense)                        (None, 1)                       2
 dense_4 (Dense)                        (None, 1024)                    2048
 dense_5 (Dense)                        (None, 1)                       1025
================================================================================
Total params: 3075 (12.01 KB)
Trainable params: 3075 (12.01 KB)
Non-trainable params: 0 (0.00 Byte)

--------------------------------------------------------------------------------


$x3
Model: "x3"

--------------------------------------------------------------------------------
 Layer (type)                           Output Shape                    Param #
================================================================================
 dense_6 (Dense)                        (None, 1)                       2
 dense_7 (Dense)                        (None, 1024)                    2048
 dense_8 (Dense)                        (None, 1)                       1025
================================================================================
Total params: 3075 (12.01 KB)
Trainable params: 3075 (12.01 KB)
Non-trainable params: 0 (0.00 Byte)

--------------------------------------------------------------------------------
```

The `predict` function can be used to obtain the predictions given a new set of values for the model covariates from the `test` set. We can choose to get the predicted response, the terms, or the additive predictor by adjusting the `type` argument. When using `type = "terms"`, the specific set of terms to be obtained can be selected using the `terms` argument. If no `newdata` parameter is provided, the function returns the predictions for the original training data.

```
eta <- predict(ngam, newdata = test, type = "link", verbose = 0)
head(eta)

[1] -1.793064 -2.181689  1.671155  5.670309  2.807211  5.099676

yhat <- predict(ngam, newdata = test, type = "response", verbose = 0)
head(yhat)

[1] -1.793064 -2.181689  1.671155  5.670309  2.807211  5.099676

terms <- predict(ngam, newdata = test, type = "terms", verbose = 0)
head(terms)

          x1          x2          x3
1 -1.952659 -1.9873928 -0.09517136
2  1.375532 -4.8206234 -0.97875702
3  3.818488 -4.9609156  0.57142270
4 -1.719761  4.3348103  0.81309992
5 -1.971988  1.4300301  1.10700893
6  3.836700 -0.1894903 -0.78969347
```

```
terms <- predict(ngam, newdata = test, type = "terms", terms = c("x1", "x3"), verbose = 0)
head(terms)

          x1          x3
1 -1.952659 -0.09517136
2  1.375532 -0.97875702
3  3.818488  0.57142270
4 -1.719761  0.81309992
5 -1.971988  1.10700893
6  3.836700 -0.78969347
```

At last, we can study the performance of the model in a set of test data using different metrics, such as the Mean Squared Error (`mse`):

```
mean((yhat - test\$y)^2)

[1] 1.063829
```

We can observe that the Mean Squared Error in the test set has similar values to the obtained values during training (`MSE_training` = 1.0311), reflecting that the model has a good capacity to generalize to a set of unseen data.

3.2 Application to real data

We decided to also show the capabilities of the **neuralGAM** package with another data set. Particularly, this section details an example of its application to real data taken from the NYC Flights 13 data set from the `nycflights13` package [Wickham, 2022]. The data set includes airline on-time data for flights departing from all the airports in New York City during 2013. It also includes useful metadata on airlines, airports, weather conditions at different airports, and plane information. We aim to predict whether a flight will be delayed (upon arrival) given departure flight information and certain weather conditions. With this in mind, firstly, we load the required data from the `nycflights13` package. We can join the flights and weather data to obtain the weather conditions at a given airport (`origin`) and time (`time_hour`). We will focus on flights departing from Newark Liberty International Airport (`origin == "EWR"`) in October, November, and December (`month %in% c(10,11,12)`).

```
if (!require("magrittr")) install.packages("magrittr", quiet = TRUE)
if (!require("dplyr")) install.packages("dplyr", quiet = TRUE)
if (!require("ggplot2")) install.packages("ggplot2", quiet = TRUE)
if (!require("gridExtra")) install.packages("gridExtra", quiet = TRUE)
if (!require("nycflights13")) install.packages("nycflights13", quiet = TRUE)
if (!require("pROC")) install.packages("pROC", quiet = TRUE)

suppressMessages(library(magrittr))
suppressMessages(library(dplyr))
suppressMessages(library(ggplot2))
suppressMessages(library(gridExtra))
suppressMessages(library(nycflights13))

seed <- 1234
set.seed(seed)
data(flights, package="nycflights13")
data(weather, package="nycflights13")
data(airlines, package="nycflights13")
dat <- filter(flights, origin == "EWR" & month %in% c(12, 11, 10)) %>%
  left_join(weather, by = c("origin", "time_hour"))  %>%
  select(arr_delay, dep_delay, dep_time, air_time, arr_time,
         origin, carrier, visib, distance, air_time, temp, humid) %>%
  data.frame
```

We construct a binary response variable `delay` which describes if the flight was delayed at arrival:

```
dat$delay = ifelse(dat$arr_delay > 0, 1, 0)
dat <- dat[!rowSums(is.na(dat)),]
```

```
print(dat %>% count(delay))
head(dat)
```

```
  delay     n
1     0 16215
2     1 12358
```

```
  arr_delay dep_delay dep_time air_time arr_time origin carrier visib distance  temp humid delay
1       -34       -13      447       69      614    EWR      US    10      529 53.06 89.31     0
2       -22         5      522      174      735    EWR      UA    10     1400 53.06 89.31     0
3        -3        -9      551      117      727    EWR      UA    10      719 55.04 89.40     0
4       -13        -9      551       40      655    EWR      B6    10      200 55.04 89.40     0
5       -46        -6      554      169      757    EWR      UA    10     1400 55.04 89.40     0
6       -30        -5      555      117      810    EWR      B6    10      937 55.04 89.40     0
```

We split the data set into training (80%) and test (20%) splits, and fit a **neuralGAM** model using the `binomial` family with 2 layers with 256 and 128 units on each layer. The model will try to predict whether a flight is going to be delayed considering the flight air time, the departure delay, and weather conditions (temperature and humidity at origin) as covariates:

```
sample <- sample(nrow(dat), 0.8 * nrow(dat))
train <- dat[sample, ]
test <- dat[-sample, ]
ngam <- neuralGAM(delay ~ s(air_time) + s(dep_delay) + s(temp) + s(humid),
                  data = train,
                  num_units = c(256, 128),
                  family = "binomial",
                  seed = seed,
                  bf_threshold = 1e-2,
                  ls_threshold = 0.01,
                  loss = "mse",
                  verbose = 0)
ngam
```

```
Class: neuralGAM

Distribution Family:  binomial
Formula:  delay ~ s(air_time) + s(dep_delay) + s(temp) + s(humid)
Intercept: 0.1024
MSE: 0.1529
Sample size: 22858
```

The model is able to achieve a MSE in training of 0.1529 which showcases a good performance on real data. The graphical representation of the model can be obtained using the `autoplot` function, specifying with the `select` argument the covariate to be plotted.

```
p1 <- autoplot(ngam, select = "air_time", xlab = "Air Time (min)")
p2 <- autoplot(ngam, select = "dep_delay", xlab = "Departure Delay (min)")
p3 <- autoplot(ngam, select = "temp", xlab = "Temperature (ºC)")
p4 <- autoplot(ngam, select = "air_time", xlab = "Relative Humidity")
gridExtra::grid.arrange(grobs = list(p1,p2,p3,p4), ncol = 2, nrow = 2)
```

Figure 2 shows the learned partial effects for each covariate. Observing the plots we can study how the departure (`air_time`, `dep_delay`) and weather conditions (`temp`, `humid`) influence the delay at arrival. As expected, higher values of departure delay increase the probability of a flight being delayed at arrival. Regarding the influence of the `air_time` (the number of minutes the flight is on air), we can see how flights of up to 60 minutes have a higher probability of being delayed, and the probability decreases for larger flights. This could be because larger flights have more time to recover from departure delays on the route. At last, as expected the arrival delay is also influenced by the weather conditions: lower temperatures increase the probability of delay, especially below 32ºF (0ºC). Similarly, higher
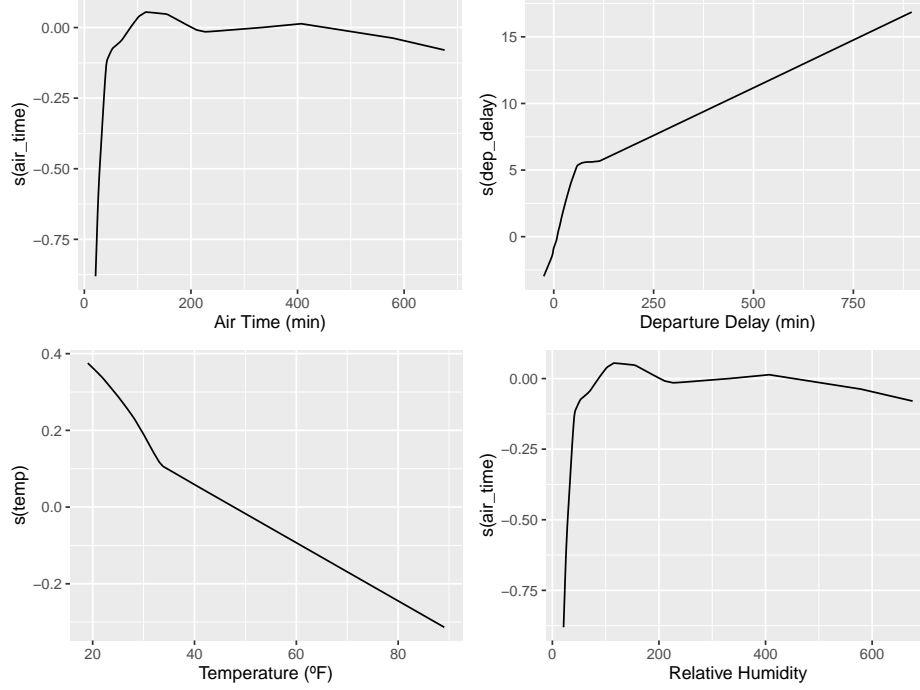
Figure 2: Learned partial effect plots by the neural network model for each covariate

relative humidity values increase the delay probability, since they represent less favorable weather conditions for flight departure.

Moreover, we can study the performance of the model in a set of test data using the `predict` function and analyse its performance using the area under the ROC curve:

```
predictions <- predict(ngam, newdata = test, type = "response", verbose = 0)
suppressMessages(library(pROC))
roc(test$delay, predictions)$auc
```

```
Area under the curve: 0.8301
```

An AUC of 0.83 indicates robust discrimination between on-time and delayed flights. Thus, **neuralGAM** delivers both interpretability and strong performance in this real data example.

4 Summary and discussion

In this paper, we introduced **neuralGAM**, an R package for fitting Generalized Additive Neural Networks (GANNs) which combines the interpretability of additive models with the flexibility of Neural Networks. The proposed methodology extends the classical Generalized Additive Model (GAM) framework by using independent Neural Networks to estimate the contribution of each covariate to the response, ensuring additivity through the local scoring and backfitting algorithms. This white-box modeling approach enables the visualization and interpretation of each feature's partial effect while maintaining the ability to learn complex, non-linear patterns from data.

Through comprehensive examples on both simulated and real data, we demonstrated the ability of **neuralGAM** to recover true functional relationships and generalize well in predictive tasks, while offering clear interpretability of learned effects. Importantly, **neuralGAM** is, to the best of our knowledge, the only R-based implementation of a GANN using deep learning, positioning it as a valuable tool for researchers seeking interpretable neural network implementations.

The current version of the package supports both Gaussian and binomial response distributions, making it suitable for multiple applications. Future work includes the extension to other response types, such as the Poisson and multinomial, and the integration of additional deep learning backends and APIs beyond Keras and TensorFlow. We also foresee

the incorporation of support for feature interactions and feature selection. These additions will further enhance the interpretability, flexibility, and applicability of the model across a broader range of statistical and machine learning problems.

Computational details

The results in this paper were obtained using R 4.4.1 with the **neuralGAM** 1.1.1 package. R itself and all packages used are available from the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/`.

Acknowledgments

References

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL `http://www.deeplearningbook.org`.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013. URL `https://arxiv.org/abs/1312.6199`.

Filip Karlo Došilović, Mario Brčić, and Nikica Hlupić. Explainable artificial intelligence: A survey. In *2018 41st International convention on information and communication technology, electronics and microelectronics (MIPRO)*, pages 0210–0215. IEEE, 2018.

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Model-Agnostic Interpretability of Machine Learning. *arXiv preprint arXiv:1606.05386*, 2016a. URL `http://arxiv.org/abs/1606.05386`.

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Nothing else matters: Model-agnostic explanations by identifying prediction invariance. *arXiv preprint arXiv:1611.05817*, 2016b.

Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *Advances in neural information processing systems*, 30, 2017.

Ines Ortega-Fernandez, Marta Sestelo, and Nora M. Villanueva. Explainable generalized additive neural networks with independent neural network training. *Statistics and Computing*, 34(1):6, Oct 2023. ISSN 1573-1375. doi:10.1007/s11222-023-10320-5. URL `https://doi.org/10.1007/s11222-023-10320-5`.

William J. E. Potts. Generalized additive neural networks. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '99, page 194–200, New York, NY, USA, 1999. Association for Computing Machinery. ISBN 1581131437. doi:10.1145/312129.312228.

Carlos Brás-Geraldes, Ana Papoila, and Patricia Xufre. Generalized additive neural network with flexible parametric link function: model estimation using simulated and real clinical data. *Neural Computing and Applications*, 31(3): 719–736, 2019. ISSN 1433-3058. doi:10.1007/s00521-017-3105-6.

Rishabh Agarwal, Levi Melnick, Nicholas Frosst, Xuezhou Zhang, Ben Lengerich, Rich Caruana, and Geoffrey E Hinton. Neural additive models: Interpretable machine learning with neural nets. *Advances in Neural Information Processing Systems*, 34, 2021.

Zebin Yang, Aijun Zhang, and Agus Sudjianto. Gami-net: An explainable neural network based on generalized additive models with structured interactions. *Pattern Recognition*, 120:108192, 2021.

Mathias Kraus, Daniel Tschernutter, Sven Weinzierl, and Patrick Zschech. Interpretable generalized additive neural networks. *European Journal of Operational Research*, 2023.

W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S*. Springer, New York, fourth edition, 2002. URL `https://www.stats.ox.ac.uk/pub/MASS4/`. ISBN 0-387-95457-0.

Xiao Rong and Maintainer Xiao Rong. Package 'deepnet', 2014.

Q Kou and Y Sugomori. Rcppdl: Deep learning methods via rcpp, 2014.

Spencer Aiello, Eric Eckstrand, Anqi Fu, Mark Landry, and Patrick Aboyoun. Machine learning with r and h2o. *H2O booklet*, 550, 2016.

François Chollet et al. Keras. `https://keras.io`, 2015.

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

Daniel Falbel and Javier Luraschi. *torch: Tensors and Neural Networks with 'GPU' Acceleration*, 2023. https://torch.mlverse.org/docs, https://github.com/mlverse/torch.

Marcus W Beck. Neuralnettools: Visualization and analysis tools for neural networks. *Journal of statistical software*, 85(11):1, 2018.

Christoph Molnar, Giuseppe Casalicchio, and Bernd Bischl. iml: An r package for interpretable machine learning. *Journal of Open Source Software*, 3(26):786, 2018.

Przemysław Biecek. Dalex: Explainers for complex predictive models in r. *The Journal of Machine Learning Research*, 19(1):3245–3249, 2018.

John Ashworth Nelder and Robert WM Wedderburn. Generalized linear models. *Journal of the Royal Statistical Society Series A: Statistics in Society*, 135(3):370–384, 1972.

Trevor Hastie and Robert Tibshirani. Generalized additive models. *London: Chapman and Hall*, 1931(11):683–741, 1990.

Stefan Lang and Andreas Brezger. Bayesian p-splines. *Journal of computational and graphical statistics*, 13(1): 183–212, 2004.

Matt P Wand and M Chris Jones. *Kernel smoothing*. CRC press, 1994.

Karen A.F. Copeland. Local polynomial modelling and its applications, 1997.

Carl De Boor, Carl De Boor, Etats-Unis Mathématicien, Carl De Boor, and Carl De Boor. *A practical guide to splines*, volume 27. Springer-Verlag New York, 1978.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989. ISSN 0893-6080. doi:https://doi.org/10.1016/0893-6080(89)90020-8. URL `https://www.sciencedirect.com/science/article/pii/0893608089900208`.

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

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR. URL `https://proceedings.mlr.press/v9/glorot10a.html`.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Hadley Wickham. *nycflights13: Flights that Departed NYC in 2013*, 2022. URL `https://github.com/hadley/nycflights13`. R package version 1.0.2.