Bachelor's Thesis

`fme` – An `R` Package for Forward Marginal Effects

DEPARTMENT OF STATISTICS

Ludwig-Maximilians-Universität München

Degree course: B.Sc. Statistics

Author	Holger Löwe
Supervisors	Prof. Dr. Christian Heumann
	Christian A. Scholbeck, M.Sc.
Date	München, August 4, 2022

Abstract

Marginal effects (ME) are a well-known framework to interpret feature effects in traditional statistical models. In part, their appeal is based on their simplicity and conceptual connection to beta coefficients in linear regression models. Forward marginal effects (fME) are a new class of MEs designed as a model-agnostic method for interpreting feature effects in machine learning (ML) models. They are computed as a forward difference in prediction due to a specified change in feature values. This thesis introduces `fme`, an `R` package that computes fMEs for arbitrary supervised regression models. As of now, it is the only `R` package with the ability to estimate MEs for tree-based learners, such as the popular random forests or gradient-boosted trees. The main objective of this implementation is to facilitate a full workflow of computing fMEs for regression models: extrapolation point detection, estimating the non-linearity of the prediction function along the forward difference, aggregating MEs, identifying feature subspaces through recursive partitioning, and visualizing results. The main idea is to separate the functionality into modular components implemented as classes, allowing for maximum flexibility in extending the design. In this way, `fme` can be easily expanded to new models or visualization methods. While `fme` internally works on a `R6` object-oriented class system, it provides convenient wrapper functions for a highly intuitive user experience.

Contents

1 Introduction

While the excellent predictive performance of machine learning (ML) models has been demonstrated in a variety of settings, the biggest challenge to a more widespread adoption of ML systems likely resides in a perceived lack of interpretability (Doshi-Velez and Kim, 2017). Specifically, in supervised learning, there is some interest in interpreting model properties in a way similar to beta coefficients in linear regression models. Marginal effects (ME) are a popular method for interpreting feature effects in traditional statistical models, such as logistic regression or generalized linear models (GLM), and are implemented in a wide variety of `R` packages. However, they have not yet been used as a model-agnostic interpretation method in supervised ML.

Scholbeck et al. (2022) introduce forward marginal effects (fME) to overcome some of the conceptual deficiencies of traditional MEs. They are computed as a forward difference in prediction due to a specified change in feature values. This thesis implements fMEs as a method to interpret feature effects in `R`. The result of this work is `fme`, a fully functional and documented `R` package designed to compute fMEs for arbitrary supervised regression models. To our knowledge, this is the only `R` package that implements the concept of MEs as a model-agnostic interpretation method. By implication, this means MEs can now be computed for popular ML models like random forests or gradient-boosted trees. `Fme` allows users to compute, analyze and visualize feature effects for numerical and categorical features. For numerical features, this comprises both univariate and bivariate feature changes. Furthermore, `fme` performs extrapolation detection and estimates the non-linearity of the shape of the prediction function along the forward difference. As a further tool, it allows for semi-global interpretations of MEs by partitioning the feature space into subspaces with more homogeneous MEs.

The thesis is structured as follows: Chapter 2 explains the theoretical background. We introduce the necessary notation, conceptualize interpretability in ML and provide a short survey of methods for interpreting feature effects. We review the existing methodology of MEs, in particular the framework for computing fMEs as suggested by Scholbeck et al. (2022). Technical aspects of the implementation, including software design and class architecture, are covered in chapter 3. We show how to install and use the `fme` package in chapter 4. Using real-life data of bike sharing usage in Washington, D.C., we demonstrate its main functionalities, user interface and outputs. Lastly, chapter 5 summarizes the main results and discusses the challenges and tasks for `fme`'s further development.

The reader can find the source code of the developer version of `fme` on GitHub[1].

[1]`https://github.com/holgstr/fme`.

2 Background

2.1 Notation and Terminology

In supervised ML, the goal is to infer a model that maps features from the training data to an target, and then use this model to predict on (potentially unseen) data from the same underlying distribution (Bischl et al., 2021). Consider a data set $\mathcal{D}$ with n observations where each observation $(\mathbf{x}^{(i)},\, y^{(i)})$ consists of a p-dimensional vector of feature variables $\mathbf{x}^{(i)} = (x_1^{(i)},\, \dots,\, x_p^{(i)}) \in \mathcal{X}$ and a scalar target variable $y^{(i)} \in \mathcal{Y}$. Thus, the set of all feature values in $\mathcal{D}$, often called feature matrix, can be represented by $\{\mathbf{x}^{(i)}\}_{i=1}^{n}$ and the set of all target values by $\{y^{(i)}\}_{i=1}^{n}$. An arbitrary feature subspace is denoted by $\mathcal{X}_{[]} \subsetneq \mathcal{X}$. A partitioning $\mathcal{P} = \{\mathcal{X}_{[1]},\, \dots,\, \mathcal{X}_{[m]}\}$ divides the feature space $\mathcal{X}$ in m mutually disjoint (specific) partitions, with $\dot{\bigcup}_{i=1}^{m} \mathcal{X}_{[i]} = \mathcal{X}$. It is assumed that all observations in $\mathcal{D}$ are independently and identically distributed and sampled from an usually unknown underlying probability distribution $\mathbb{P}_{XY}$, defined on the sample space $\mathcal{X} \times \mathcal{Y}$. The corresponding random variables are $\mathbf{X} = (X_1,\, \dots,\, X_p)$ and Y. Let $P = \{1,\, \dots,\, p\}$ be the index set denoting all features and $S = \{1,\, \dots,\, s\} \subseteq P$ be a subset of P. The feature values of a single observation in the dimensions represented by S are denoted by $\mathbf{x}_S^{(i)}$. Complementary feature values are indexed by $-S$ or $-j$. In this way, we can express an observation $\mathbf{x}^{(i)}$ as $(\mathbf{x}_S^{(i)},\, \mathbf{x}_{-S}^{(i)})$.

Each of the p dimensions of the vector $\mathbf{x}^{(i)}$ is of numerical or categorical type, meaning entries are either real numbers or elements of a finite set of distinct categories, e.g., city names or blood groups. Within supervised ML, a common task is (univariate) regression, where the target variable $y^{(i)} \in \mathcal{Y} \subseteq \mathbb{R}$ is of numerical type. A learner is a procedure or computer algorithm that maps a data set $\mathcal{D}$ to a prediction function $\widehat{f} : \mathcal{X} \to \mathcal{Y}$, which becomes $\widehat{f} : \mathcal{X} \to \mathbb{R}$ in the case of a regression task. A major objective of supervised ML is to minimize the generalization error of the model $\widehat{f}$, i.e., the expected performance of the prediction function on new observations sampled from $\mathbb{P}_{xy}$ that were not used for learning. Here, performance is often measured by a function that maps a prediction from $\widehat{f}$ and the true value of the target variable $y^{(i)}$ to some metric, such as mean squared error or mean absolute error.

There is a wide range of learners that can be used to generate prediction functions. A learner typically restricts the set of admissible prediction functions and then uses a simple rule or iterative algorithm to identify a prediction function $\widehat{f}$. An example for a more complicated learner are random forests (Breiman, 2001), as implemented in the `ranger` `R` package (Wright and Ziegler, 2017). Here, the learner constructs a large number of decision trees, each of which is trained on a bootstrap sample of $\mathcal{D}$. A decision tree is “grown” by a forward-looking algorithm that recursively partitions the feature space in a way that creates subspaces as homogeneous as possible with respect to the target variable. In the context of decision trees, such subspaces are commonly referred to as nodes. The unpartitioned feature space is called root node and the subspaces of the final partitioning are called terminal nodes.

2.2 Interpretability in Supervised Machine Learning

2.2.1 What is Interpretability?

The advent of ML and an ever-expanding range of learners has been accompanied by a growing desire to understand what a model has learned from the data. Efforts to make ML model outputs explainable and interpretable can be summarized with the terms Explainable Artificial Intelligence (XAI) or Interpretable Machine Learning (IML). However, consensus seems to be weak on what exactly interpretability means in the context of (supervised) ML. Murdoch et al. (2019) note that indeed "[o]n its own, interpretability is a broad, poorly defined concept", and attempt to define it as "the extraction of relevant knowledge from a machine-learning model concerning relationships either contained in data or learned by the model." Here, knowledge can be understood to be what may provide insight into what a particular audience perceives as relevant for the given task. One aspect that makes a precise definition of interpretability so challenging is that it highly depends on the perspective of the user. In some cases, a visualization can provide the most meaningful insights, in others, a mathematical equation may be preferred.

Murdoch et al. (2019) suggest to guide potential trade-offs of choosing one interpretation method over another by three desiderata: predictive accuracy, descriptive accuracy, and relevancy. The first aspect, predictive accuracy, maps to what we have described before as the generalization error of a supervised ML model. If a model cannot be expected to predict well on data it has not been trained on, any interpretation falls short of its objective, as the underlying relationship between feature and target variables has not been approximated well enough. As for the second aspect, descriptive accuracy relates to the post hoc analysis stage. Often, an interpretation method reduces the complex representation learned by a model to a simpler one. While this is a desired property in many cases, it can lead to substantial distortions in the inferred relationship between feature and target variables. This is especially relevant for models where the prediction function is based on interactions of the feature variables with one another, often in a highly non-linear fashion, e.g., neural networks. Lastly, any interpretation method needs to have relevancy, i.e., it needs to be capable of providing the intended insight into the relationship the user wants to understand.

2.2.2 Interpreting Feature Effects

Considering the question of interpretability of supervised ML models, one is often interested in feature effects, i.e., the learned relationship between feature and target variables. In principle, there are two pathways to achieve interpretability in this context. The first one is to select models such that the shape of their prediction function can be leveraged to offer means of interpretation. They allow for model-based interpretations. The other one is to detach model selection from interpretation altogether. This allows for greater flexibility in the choice of a learner, as considerations about model-inherent interpretability can be discarded. After a model has been trained, an interpretation method is chosen in a subsequent, post hoc interpretation stage. Such an interpretation method approximates the modelled relationship between feature and target variables in a way that often enables an audience to gain insight into the true underlying relationship. This allows for model-agnostic interpretations. Notably, post hoc interpretation does not necessarily run contrary to model-based interpretation: There are settings where users may benefit from conducting a model-agnostic interpretation even if they have trained a model which offers (some) inherent interpretability. For example, estimated beta coefficients of a logistic regression model can be interpreted with respect to how an increase of the feature variable by one unit changes the odds ratio (multiplicatively, by a factor of $exp(\beta_j)$). This insight can be further enriched by a model-agnostic interpretation method that allows for an interpretation on the level of the probability scale, e.g., average marginal effects. Subsequently, we provide a short overview of both model-based and model-agnostic interpretations for feature effects in supervised ML.

Model-based interpretation of feature effects

Model-based interpretation solely relies on a model's prediction function to interpret feature effects. Molnar (2022) lists some of the most widely used interpretable models. They are linear regression, logistic regression, generalized linear models (GLM), generalized additive models (GAM), decision trees, and others. Among these, linear regression and its extensions (see, for example, Hastie et al., 2009) can likely be counted as the most well-known modeling framework in statistics and ML. One could argue that its perceived attractiveness stems in part from its intuitive interpretability: Given the absence of feature interactions, an increase of the j-th feature variable by one unit changes the expected target variable by β_j. The simplicity of the prediction function lets users interpret a linear regression model directly through the explicit coefficient estimates. Furthermore, it lends itself to global as well as local interpretations, as the beta coefficients are assumed to be constant over the entire feature space.

Generalizations of the linear model may allow for a more flexible modeling of the feature-target relationship. For example, a GLM as introduced by Nelder and Wedderburn (1972) may be a better model to fit a target variable that does not follow a normal distribution, but rather a Poisson or Gamma distribution. Often, interpretation of a GLM is not as straightforward as that of a linear regression model. However, its prediction function exhibits properties that facilitate insights into the modeled relationship between feature and target variables. More specifically, the choice of assumed distribution and link function determines how the estimated beta coefficients can be interpreted. Molnar (2022) notes that most GLMs are less interpretable than linear regression models, especially because any non-identity link function severely restricts interpretation. Furthermore, it is argued that other learners, e.g., ensemble methods such as random forests or gradient-boosted trees, tend to achieve better performance than linear models. On the upside, such expected gains in model performance are desired on the grounds of the desideratum of predictive accuracy. On the downside, a model may hardly be inherently interpretable if it consists of hundreds of base learners as the aforementioned ensemble learners. In part, this has led to efforts to separate interpretation from model building.

Model-agnostic interpretation of feature effects

Ribeiro et al. (2016a) suggest that confining ML to interpretable models implies a serious limitation. They argue the desire for inherent interpretability impairs unnecessary constraints on the choice of model. This hints at a general trade-off between model complexity and interpretability, as observed by Freitas (2014). However, the lack of interpretability seems to be an impediment to the further adoption of ML models. Therefore, model-agnostic interpretation frameworks can be used to bridge the gap between arbitrarily complex models and the valid desire to explain a model's predictions.

Ribeiro et al. (2016a) explain the rationale for such an approach: Firstly, predictive accuracy does not need to be sacrificed on the grounds of explainability concerns. Secondly, interpretation methods can be tailored to the information need, while keeping the model fixed. This allows for combining explanations of different types to obtain a comprehensive, multi-pronged understanding of the underlying feature-target relationship. Thirdly, as model selection and interpretation do not depend on each other, a model-agnostic approach can easily take advantage of switching to a different model, especially as this is not uncommon in typical ML pipelines. Model-agnostic interpretation therefore builds upon treating the original model as a black box, which means that the model's prediction function can be known to the user, but interpretations are not derived directly from the prediction function itself. Rather, information is extracted from the black box in a second step. There are many ways to do this. In general, a distinction between local and global model-agnostic interpretation methods is made (see, e.g., Molnar, 2022).

Local model-agnostic interpretation methods of feature effects

Local interpretation methods provide explanations for predictions on individual observations. For example, Individual Conditional Expectation (ICE) plots (Goldstein et al., 2015) visualize the modeled relationship between one or more features and the predicted target variable. For a single observation and feature, this means the feature vector is held constant in all other dimensions and changed only in the dimension of the selected feature. The feature value is then changed according to a grid and the respective predictions are plotted, isolating the effect the feature has on the model's prediction for a single observation.

Alternatively, local surrogate models, such as Local interpretable model-agnostic explanations (LIME) (Ribeiro et al., 2016b), try to approximate a model's local behavior with an interpretable surrogate model. LIME trains the surrogate model on a perturbed sample of the observation of interest, where the training instances are weighted by a proximity measure. The surrogate model, e.g., a linear regression model or decision tree, is then used to analyse feature effects. The specific implementation of LIME for tabular data depends on hyperparameters such as the choice of the smoothing kernel, which influences the sample used for training the surrogate model. Slack et al. (2020) suggest that LIME explanations are susceptible to deliberate manipulation. Alvarez-Melis and Jaakkola (2018) find that LIME explanations of neighboring observations have high instability and are often inconsistent. Molnar (2022) concludes that although local surrogate models are very promising, LIME in its current form cannot be safely applied.

Counterfactual explanations are a local interpretation method which specify the smallest change to an individual observation's feature values that results in a pre-defined change of the model's prediction. Compared to LIME, counterfactuals (Wachter et al., 2018) rely on fewer assumptions that might impose limitations on its interpretability. Arguably, the need to decide on how to handle the possibility of multiple competing counterfactual explanations poses the biggest obstacle in their application (Molnar, 2022).

Shapley values (Štrumbelj and Kononenko, 2014) measure how much the concrete value of a feature contributes to the difference between the model's prediction for the individual observation and the mean prediction (over all observations), given the observation's feature values. The underlying concept has its origin in game theory and has inspired Shapley additive explanations (SHAP) (Lundberg and Lee, 2017), a framework which estimates Shapley values with more efficient procedures.

Global model-agnostic interpretation methods of feature effects

Global interpretation methods provide explanations for the average behavior of a model. For example, the partial dependence (PD) plot (Friedman, 2001) visualizes a model's prediction as a function of one or more features. This is done by computing the average prediction of the ML model over all observations in the training data, while changing only the feature variables of interest. Therefore, PD plots are the global equivalent of ICE plots, which means they can be conveniently displayed in the same plot, e.g., as implemented in the `iml` `R` package (Molnar et al., 2018). Although the interpretation of PD plots is straightforward, they can face a limitation in their assumption of independence between individual features, which can become problematic if features are correlated (Molnar, 2022).

Accumulated local effects (ALE) (Apley and Zhu, 2020) constitute an effort to correct this shortcoming. In the presence of correlated features, they seek to avoid averaging predictions of unlikely observations. This is done by partitioning the feature of interest in many small intervals and computing differences in predictions rather than simple averages for each interval. The differences calculated for the observations within each interval are then accumulated and centered, resulting in a straightforward interpretation: For a given value of the feature variable, the estimated ALE measures the effect of the feature value on the prediction, compared to the average prediction. The computation of effects within intervals ensures that ALE works with the conditional rather than the marginal distribution of a feature, which alleviates the main problem of PD plots. Molnar (2022) highlights a potential shortcoming of ALEs: For convenience, ALE plots display a smoothed curve of the effect, which invites for a global interpretation along perceived gradual changes of the feature value. However, ALEs are calculated separately for each interval using different observations. Therefore, it is argued that an interpretation across intervals is not permissible, especially in the presence of correlated features.

2.3 Marginal Effects

2.3.1 Existing Methodology

Marginal effects (ME) are a widely used concept in statistics and other fields, such as economics (see, for example, Greene, 2012). They have a long history in application, which is arguably grounded upon its connection to the interpretation of the simple linear regression model: "[...] [M]arginal effects should measure the change in the expected value of y as one independent variable increases by unity while all other variables are kept constant" (Bartus, 2005). As noted before, for linear regression models, the ME of a feature variable corresponds to the estimated beta coefficient (in the absence of feature interaction). This is simply an implication of the linearity of the prediction function. However, the shape of the prediction function of many ML models cannot be leveraged to allow for a straightforward computation of MEs based on estimated model parameters. Therefore, a variety of techniques have been devised to compute MEs for both categorical and numerical features, a survey of which is provided by Scholbeck et al. (2022). Among those for numerical features, the most commonly used framework for MEs is based on estimating the partial derivative of the prediction function w.r.t. the feature variable of interest. For example, this can be done by approximating with a finite (forward) difference:

$$\text{ME}_{j,\mathbf{x}^{(i)}} = \frac{\partial \widehat{f}(\mathbf{X})}{\partial X_j}|_{\mathbf{X}=\mathbf{x}^{(i)}} \approx \frac{\widehat{f}(x_1^{(i)},\, \dots,\, x_j^{(i)} + h,\, \dots,\, x_p^{(i)}) - \widehat{f}(\mathbf{x}^{(i)})}{h} \ ,\ h > 0 \qquad (1)$$

One advantage of a derivative-based definition is the widespread familiarity with the concepts of derivatives and finite differences. On the contrary, the estimate may depend on the kind of finite difference used (forward, backward or central) and the choice of the step size h. This can lead to deviations between the estimates of MEs of different software packages, depending on the concrete approximation used. Scholbeck et al. (2022) discuss the most prominent weakness of this class of MEs. Their criticism reflects on the default interpretation of derivative-based MEs: if the feature value increases by one unit, the predicted outcome should increase by the estimated ME. This intuition holds true for linear prediction functions but not for non-linear prediction functions. The fact that the interpretation of the ME requires a step size h (with $h = 1$ being the default) much larger than h in Eq. 1 can easily mislead one into confusing the tangent value of the prediction function $\widehat{f}$ at point $\mathbf{x}^{(i)}$ with the true value of $\widehat{f}$ at point $(x_1^{(i)},\, \dots,\, x_j^{(i)} + h,\, \dots,\, x_p^{(i)})$.

For the `R` programming language for statistical computing (R Core Team, 2021), there is a variety of packages that allow users to compute MEs for different trained ML models. The `margins` package is an open-source implementation of Stata's (StataCorp, 2019) `margins` command by Leeper (2021) that uses central differences to estimate the partial derivative numerically. Here, MEs can be calculated through calling `margins()`, which is a generic function of the package. Additionally, the package includes further functions that allow for visualizations of the computed MEs. It can be used for regression models of classes `lm` (linear regression), `glm` (GLM), and `loess` (local polynomial regression).

The `effects` package (Fox, 2003) and the `mfx` package (Fernihough, 2019) mark other efforts to compute and visualize MEs in `R`. However, they share the detriment of being restricted to the model subclass of GLMs. The `marginaleffects` package (Arel-Bundock, 2022) features compatibility for a wider range of models, e.g., GAMs as implemented in the `mgcv` package (Wood, 2011). It estimates MEs as partial derivatives with a forward differences approach. Its visualizations are created with `ggplot2` (Wickham, 2016), enabling a user-friendly, flexible extension of plots.

All of the aforementioned `R` packages are severely restricted in the space of ML models they can be applied to. While they can be used for `R` objects of class `lm`, `glm`, and some others, they cannot be used for tree-based models, e.g., `R` objects of class `ranger`, `randomForest`, or `xgboost`. Scholbeck et al. (2022) note this is a constraint induced by the derivative-based ME: most observations lie on piecewise constant parts of the prediction function and have a finite difference of zero when h is small enough. As a consequence, the lack of compatibility with the popular subclass of tree-based ML models makes derivative-based MEs ill-suited to serving as a general post hoc interpretation tool for supervised regression models.

2.3.2 Forward Marginal Effects

Forward marginal effects (fME) are a new class of MEs introduced by Scholbeck et al. (2022). They are motivated to compensate for the shortcomings of derivative-based MEs and entail a simple, more intuitive definition of MEs. For a single numerical feature variable, the fME of an observation is defined as follows:

$$\text{fME}_{\mathbf{x}^{(i)},\, h_j} = \widehat{f}(x_1^{(i)},\, \ldots,\, x_j^{(i)} + h_j,\, \ldots,\, x_p^{(i)}) - \widehat{f}(\mathbf{x}^{(i)}) \tag{2}$$

Note that this is simply the forward difference of the prediction function at point $\mathbf{x}^{(i)}$. The step size h_j can be specified w.r.t. the required scale of interpretation. Most of the time, this can be a unit change: $h_j = 1$. In practice, it can be any number one finds most useful for the purpose of interpretation, e.g., for a feature variable 'annual household income in euros', it could be $h_j = 1{,}000$. The merit of fMEs is that contrary to derivative-based MEs, the step size used for effect estimation is the same as the (implied) step size used for interpretation. As the fME is computed without dividing by the step size, it is interpreted as a change in prediction (rather than a rate of change): if the j-th feature value increases by h_j, the predicted outcome increases by the estimated fME.

The univariate fME can be extended to incorporate multivariate changes in numerical feature values: for the feature subset $S = \{1, \ldots, s\} \subseteq P$ affected by the multivariate step, the multivariate change $(x_1^{(i)} + h_1, \ldots, x_s^{(i)} + h_s)$ of an observation is denoted by $(\mathbf{x}_S^{(i)} + \mathbf{h}_S)$, an observation's unaffected features are denoted by $\mathbf{x}_{-S}^{(i)}$. Thus, the multivariate fME of an observation becomes:

$$\text{fME}_{\mathbf{x}^{(i)},\, \mathbf{h}_S} = \widehat{f}(\mathbf{x}_S^{(i)} + \mathbf{h}_S,\, \mathbf{x}_{-S}^{(i)}) - \widehat{f}(\mathbf{x}^{(i)}) \tag{3}$$

Scholbeck et al. (2022) further propose an univariate observation-wise ME for categorical features. Equivalent to the step size h_j for a numerical feature, a reference category c_h is chosen for the categorical feature of interest $\mathbf{x}_j^{(i)}$. Correspondingly, the unaffected features are denoted by $\mathbf{x}_{-j}^{(i)}$. For observations with $x_j^{(i)} \neq c_h$, the ME is then computed as follows:

$$\mathrm{ME}_{\mathbf{x}^{(i)},\, c_h} = \widehat{f}(c_h,\, \mathbf{x}_{-j}^{(i)}) - \widehat{f}(\mathbf{x}^{(i)}) \tag{4}$$

This definition corresponds to the fME insofar as the model's prediction for the unaltered observation serves as reference point and is compared against the prediction for the altered observation. Drawing from this similarity, we could refer to the modification of $\mathbf{x}^{(i)}$ to $(c_h,\, \mathbf{x}_{-j}^{(i)})$ in Eq. 4 as categorical step in the same way as we do to the modification of $\mathbf{x}^{(i)}$ to $(\mathbf{x}_S^{(i)} + \mathbf{h}_S,\, \mathbf{x}_{-S}^{(i)})$ in Eq. 3 as (multivariate) numerical step.

Extrapolation point detection

Computing fMEs as in Eq. (3) and categorical MEs as in Eq. (4) requires changing feature values of observations in the training data. This creates artificial data points that can be located outside of the multivariate joint distribution of the data $\mathbb{P}_X$. Furthermore, it can lead to predictions in areas of $\mathcal{X}$ with a low density of training points (Molnar et al., 2020). This is much less a challenge for derivative-based MEs than it is for fMEs. Derivative-based MEs as defined in Eq. (1) are typically estimated with h being close to zero. On the other hand, fMEs are defined for arbitrary step sizes.

There are many ways to identify such extrapolation points (EP). For example, King and Zeng (2006) define an EP as an observation outside of the convex hull of the training data. Scholbeck et al. (2022) discuss a multi-step approach termed Monte-Carlo extrapolation classification (MCEC). In general, they argue to exclude EPs from the data used to calculate fMEs. Their workflow suggests to classify an observation as EP if it is located outside of the multivariate envelope of the training data after the change in feature values. Using this simple criterion, the set of EPs for training data $\mathcal{D}$ and a step $\mathbf{h}_S$ can be represented as the following index vector:

$$\begin{aligned}\mathcal{E}_{\mathcal{D},\, \mathbf{h}_S} = \{\, i \mid (\mathbf{x}_S^{(i)} + \mathbf{h}_S,\, \mathbf{x}_{-S}^{(i)}) \notin [\min(x_1),\, \max(x_1)] \times \ldots \times [\min(x_p),\, \max(x_p)] \,\} \\ \subseteq \{1,\, \ldots,\, n\}\end{aligned} \tag{5}$$

where $\min(x_j)$ and $\max(x_j)$ denote the minimum and maximum values in the j-th dimension across all observations in $\mathcal{D}$, respectively. Note that the multivariate envelope is only sensibly defined for numerical features.

Non-linearity measure (NLM)

As the fME is not designed to capture information about the prediction function's shape along the forward difference, Scholbeck et al. (2022) devise a non-linearity measure (NLM) to close this gap. Defined in arbitrary dimensions, the NLM quantifies how well the multivariate secant along the forward difference as a linear reference function can describe the shape of the prediction function. The multivariate secant of an observation $\mathbf{x}^{(i)}$ and step size $\mathbf{h}_S$ along the forward difference is:

$$g_{\mathbf{x}^{(i)},\,\mathbf{h}_s}(t) = \begin{pmatrix} x_1^{(i)} + t \cdot h_1 \\ \vdots \\ x_s^{(i)} + t \cdot h_s \\ \vdots \\ x_p^{(i)} \\ \widehat{f}(\mathbf{x}^{(i)}) + t \cdot \mathrm{fME}_{\mathbf{x}^{(i)},\,\mathbf{h}_S} \end{pmatrix} \quad , \ t \in [0,1] \tag{6}$$

In order to compare the multivariate secant with the prediction function along the forward difference, the concept of the coefficient of determination R^2 is extended to continuous integrals. In analogy to R^2, this means comparing the squared deviation of the prediction function and the secant with the squared deviation of the prediction function and its mean. Both deviations are computed as line integrals along the same path through the feature space. For a single observation, $\gamma(t)$ yields the parametrization of the path through $\mathcal{X}$:

$$\gamma(t) = \begin{pmatrix} x_1^{(i)} \\ \vdots \\ x_p^{(i)} \end{pmatrix} + t \cdot \begin{pmatrix} h_1 \\ \vdots \\ h_s \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad , \ t \in [0,1] \tag{7}$$

with $\gamma(0) = \mathbf{x}^{(i)}$ and $\gamma(1) = (\mathbf{x}_S^{(i)} + \mathbf{h}_S,\, \mathbf{x}_{-S}^{(i)})$. The line integral of the squared deviation between the prediction function and the multivariate secant along the forward difference is represented by:

$$(\mathrm{I}) = \int_0^1 \left(\widehat{f}(\gamma(t)) - g_{\mathbf{x}^{(i)},\,\mathbf{h}_s}(\gamma(t))\right)^2 \left\| \frac{\partial \gamma(t)}{\partial t} \right\|_2 dt \tag{8}$$

and the line integral of the squared deviation between the prediction function and the mean prediction along the forward difference is represented by:

$$\text{(II)} = \int_0^1 \left(\widehat{f}(\gamma(t)) - \overline{\widehat{f}}\right)^2 \left\|\frac{\partial\gamma(t)}{\partial t}\right\|_2 dt \tag{9}$$

with the mean prediction $\overline{\widehat{f}}$ as the integral of the prediction function along the forward difference divided by the length of the path:

$$\overline{\widehat{f}} = \frac{\int_0^1 \widehat{f}(\gamma(t))\left\|\frac{\partial\gamma(t)}{\partial t}\right\|_2 dt}{\int_0^1 \left\|\frac{\partial\gamma(t)}{\partial t}\right\|_2 dt} = \int_0^1 \widehat{f}(\gamma(t))dt \tag{10}$$

and

$$\left\|\frac{\partial\gamma(t)}{\partial t}\right\|_2 = \sqrt{h_1^2 + \ldots + h_s^2} \tag{11}$$

The line integrals can be approximated with Simpson's rule or other methods. Corresponding to the definition of the R^2, the NLM is computed as follows:

$$\text{NLM}_{\mathbf{x}^{(i)},\mathbf{h}_S} = 1 - \frac{\text{(I)}}{\text{(II)}} \tag{12}$$

An NLM of 1 indicates that the prediction function is equivalent to the secant and, therefore, linear. A lower value implies increasing non-linearity, a negative value suggests the mean prediction to predict better than the multivariate secant. Scholbeck et al. (2022) recommend to use a hard (lower) bound of zero to indicate non-linearity.

2.3.3 Aggregations of marginal effects

Often, one is more interested in the interpretation of MEs for a set of observations rather than for a single one. Consider one wants to aggregate MEs over the entire data set $\mathcal{D}$. The two most commonly used concepts to aggregate MEs are the marginal effect at means (MEM) and the average marginal effect (AME) (see, for example, Williams, 2012). Both concepts can be applied to derivative-based MEs and fMEs, respectively.

Marginal effect at means (MEM)

The idea of the MEM is to to compute the ME for a single observation that is representative for the distribution $\mathbb{P}_X$. Therefore, feature values are replaced by their sample mean. In the case of fMEs, the MEM for a multivariate change in feature values is:

$$\text{MEM}_{\mathcal{D},\,\mathbf{h}_S} = \widehat{f}\left(\left(\frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_S^{(i)}\right) + \mathbf{h}_S,\, \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_{-S}^{(i)}\right) - \widehat{f}\left(\frac{1}{n}\sum_{i=1}^{n}\mathbf{x}^{(i)}\right) \tag{13}$$

Note that this version of the MEM is only defined for observations with exclusively numerical features, as categorical features cannot be averaged.

Average marginal effect (AME)

The AME is the average of all MEs computed for the observations in $\mathcal{D}$. In the case of fMEs, the AME for a multivariate change in feature values is:

$$\text{AME}_{\mathcal{D},\,\mathbf{h}_S} = \frac{1}{n}\sum_{i=1}^{n} \text{fME}_{\mathbf{x}^{(i)},\,\mathbf{h}_S} = \frac{1}{n}\sum_{i=1}^{n}\left(\widehat{f}(\mathbf{x}_S^{(i)} + \mathbf{h}_S,\, \mathbf{x}_{-S}^{(i)}) - \widehat{f}(\mathbf{x}^{(i)})\right) \tag{14}$$

In contrast to the MEM, the AME can be computed for data with both numerical and categorical features, providing greater practical utility.

Beyond that, the AME can be used to aggregate the observation-wise MEs for categorical features as defined in Eq. (4). It is computed as follows:

$$\text{AME}_{\mathcal{D},\,\mathbf{c}_h} = \frac{1}{n}\sum_{i=1}^{n} \text{ME}_{\mathbf{x}^{(i)},\,c_h} = \frac{1}{n}\sum_{i=1}^{n}\left(\widehat{f}(c_h,\, \mathbf{x}_{-j}^{(i)}) - \widehat{f}(\mathbf{x}^{(i)})\right) \quad \forall i : x_j^{(i)} \neq c_h \tag{15}$$

Conditional average marginal effects (cAME) on feature subspaces

Scholbeck et al. (2022) discuss how aggregating MEs across the entire feature space $\mathcal{X}$ can fail to capture the heterogeneity of MEs. In order to compromise between the benefit of aggregation and the desire to conduct a more local evaluation of feature effects, they suggest to compute AMEs conditional on a feature subspace $\mathcal{X}_{[\,]} \subsetneq \mathcal{X}$. The resulting conditional average marginal effect (cAME) may then serve as a device for semi-global interpretations. The cAME for numerical fMEs as defined in Eq. (3) is computed for observations represented as index vector $\mathcal{I} = \{\, i \mid \mathbf{x}^{(i)} \in \mathcal{X}_{[\,]}\,\} \subsetneq \{1,\, \ldots,\, n\}$:

$$\text{cAME}_{\mathcal{D},\,\mathcal{I},\,\mathbf{h}_S} = \frac{1}{|\mathcal{I}|}\sum_{i\in\mathcal{I}} \text{fME}_{\mathbf{x}^{(i)},\,\mathbf{h}_S} = \frac{1}{|\mathcal{I}|}\sum_{i\in\mathcal{I}}\left(\widehat{f}(\mathbf{x}_S^{(i)} + \mathbf{h}_S,\, \mathbf{x}_{-S}^{(i)}) - \widehat{f}(\mathbf{x}^{(i)})\right) \tag{16}$$

The cAME for categorical MEs as defined in Eq. (4) is computed for observations represented as index vector $\mathcal{J} = \{\, i \mid \mathbf{x}^{(i)} \in \mathcal{X}_{[\,]} \wedge x_j^{(i)} \neq c_h \,\} \subsetneq \{1, \ldots, n\}$:

$$\text{cAME}_{\mathcal{D},\,\mathcal{J},\,\mathbf{c}_h} = \frac{1}{|\,\mathcal{J}\,|} \sum_{i \in \mathcal{J}} \text{ME}_{\mathbf{x}^{(i)},\, c_h} = \frac{1}{|\,\mathcal{J}\,|} \sum_{i \in \mathcal{J}} \left(\widehat{f}(c_h,\, \mathbf{x}_{-j}^{(i)}) - \widehat{f}(\mathbf{x}^{(i)}) \right) \tag{17}$$

For the purpose of semi-global interpretations, they suggest to partition the feature space into mutually exclusive subspaces. This should be done in a way that results in subspaces containing observations with more homogeneous MEs. The partitioning can be conducted with recursive partitioning (RP) algorithms like CART (Breiman et al., 1984) or CTREE (Hothorn et al., 2006). Subsequently, one computes cAMEs for each partition of the feature space. In a similar way, a conditional average non-linearity measure (cANLM) can be computed for such a subspace. The interpretation is as follows: For an arbitrary partition $\mathcal{X}_{[\,]}$, the cAME represents the average ME of observations that are located in $\mathcal{X}_{[\,]}$. The cANLM then seeks to describe the average linearity of the shape of the prediction function along the forward difference for observations located in $\mathcal{X}_{[\,]}$. Thus, one gets a comprehensive overview of the behavior of the ML model across subsets of the feature space.

3 Implementation

The idea of the fme package is to separate the functionality into modular components implemented as classes. The package is written entirely in R and in accordance with open-source software design principles. Users can review the source code by inspecting the corresponding GitHub repository and provide dynamic feedback regarding potential malfunctions, feature requests or similar issues.

This chapter describes the main idea of fme. Section 3.1 provides an overview about the design principles used to guide the implementation. The illustrations of design patterns are inspired by the framework used by Gamma et al. (1995). In section 3.2, we introduce the class architecture used and the functionality contained within each class.

3.1 Software Design Principles

3.1.1 Object-Oriented Design

The modular design principle mentioned above can be realized effectively with object-oriented programming (OOP). R is a programming language that is both functional and object-oriented (Morandat et al., 2012). It offers multiple class-based OOP systems, the most notable ones being S3, S4 and R6 (Wickham, 2019). Whereas S3 and S4 are included in base R, R6 is provided by the R6 package (Chang, 2021). The fme package relies on R6 for its class architecture and on S3 methods for enhanced user convenience.

R6

All classes are implemented as R6 classes. Instantiating an object of a R6 class can be done through `class$new()`. R6 utilizes the OOP paradigm of encapsulation, which implies that methods belong to classes. A method of an R6 object can be called by `object$method()`. In the same way, fields can be called by `object$field`. This has the advantage that the user may quickly identify appropriate methods and fields of an R6 object by typing `object$` in the console in RStudio. A further advantage of R6 is method chaining, i.e., calling multiple methods sequentially on the same object can be expressed efficiently as `object$method()$method()`. This works for the same method as it does for different ones. On the contrary, many users are likely not familiar with the R6 syntax, such as the infix operator `$`. In the implementation, this is alleviated with the help of wrapper functions. Here, a wrapper function is a generic R function that simply masks the underlying R6 commands. For example, the `fme()` function is a wrapper for `FME$new()$compute()`. This allows users to access the package's main functionality in the way that is most convenient for them.

S3 methods

As mentioned before, `fme` implements classes exclusively in R6. However, it provides additional functions that are commonly called S3 methods (see, for example, Wickham, 2019). These methods have their origin in the S3 OOP system, but can be defined for R6 objects as well. In principle, a S3 method is a function that can be applied to objects of different classes, with the internal implementation of the function being specific to the class of the object provided as argument. So far, the `fme` package includes the well-established S3 methods `print()`, `summary()`, and `plot()` for its two main classes, `FME` and `Partitioning`.

3.1.2 Defensive Programming

In software engineering, defensive programming is a concept that entails software behaving in a predictable manner in the case of unexpected user actions (Wickham, 2014). For example, if a user provides an unsuitable argument to a function, it is helpful to throw an error in combination with an informative error message. This has the merit of enabling efficient debugging and preventing logical errors, i.e., errors that do not necessarily cause the program to terminate or crash but produce wrongful output. In `fme`, this is facilitated through the use of the `checkmate` package (Lang, 2017). For the three main classes of `fme` that the user is expected to interact with (`Predictor`, `FME` and `Partitioning`), it is ensured that if arguments required at instantiation have the wrong form, an informative error message is displayed. For an example of `fme`'s use of assertions, the reader is referred to chapter 4.

3.1.3 Polymorphism

In OOP, polymorphism relates to class inheritance (Gamma et al., 1995). The benefit of inheritance-based polymorphism is that many classes can share the same interface without the need to define the interface separately for each class. In practice, this means that a superclass contains the relevant minimal functionality that is inherited by each of the subclasses. The subclasses can contain additional functionality, such as supplemental fields or methods. They may also override methods and fields of the superclass. Thereby, polymorphism allows for a more flexible design. For example, Fig. 1 illustrates how `fme` utilizes polymorphism to implement a versatile framework for finding feature space partitions. The abstract superclass `Partitioning` provides the interface shared by its subclasses `PartitioningRpart` and `PartitioningCtree`. Beyond that, a subclass-specific method `growTree()` is defined. This means that regardless of the type of RP algorithm used to grow the tree to compute the feature space partitioning, an instance of the subclass responds in the expected way to the methods (e.g., `plot()`) defined in the superclass.

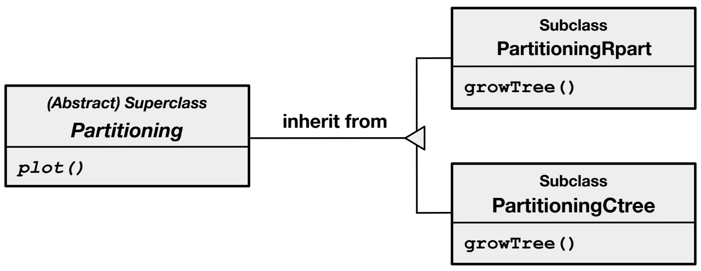

Figure 1: Illustration of polymorphism for a simplified example of the `Partitioning` classes in `fme`. The subclasses `PartitioningRpart` and `PartitioningCtree` inherit the `plot()` method from the superclass `Partitioning`. In addition, each of the subclasses defines a separate method `growTree()` that depends on the specific RP algorithm used.

3.1.4 Adapter Pattern

The adapter pattern is a structural pattern that converts the interface of one class into that of another (Gamma et al., 1995). This can become necessary if one wants to make independently developed class libraries with incompatible interfaces work together. The general idea is that an adapter class provides a uniform abstraction of different interfaces. The functionality contained in the adaptee classes is transformed and expressed in a unified form in the interface of the adapter class. The corresponding structure is exemplified in Fig. 2. In `fme`, ML models produced by different `R` packages are consolidated in objects of the adapter class called `Predictor`. This has the main objective to give rise to a unified `predict()` method. As a result, one can expect different models expressed in the `Predictor` object to generate predictions in exactly the same shape, regardless of how they were implemented in their original class interfaces.

3.1.5 Strategy Pattern

The strategy pattern is a behavioral pattern that encapsulates an algorithm in a class termed strategy class (Gamma et al., 1995). Other classes which use the algorithm maintain a reference to an object of the strategy class and forward the task of running the algorithm encapsulated therein. Fig. 3 illustrates this principle. In `fme`, EPs are computed as part of the `compute()` method of the `FME` class. However, the explicit algorithm for EP detection is detached from the rest of the program. The algorithm is embedded in its own strategy class called `ExtrapolationDetector`. The `FME` object delegates EP detection to `ExtrapolationDetector`, which returns an index vector $\mathcal{E}_{\mathcal{D},\, \mathbf{h}_S}$ representing the set of EPs. The strategy pattern confers multiple advantages. Firstly, the behavior of the algorithm

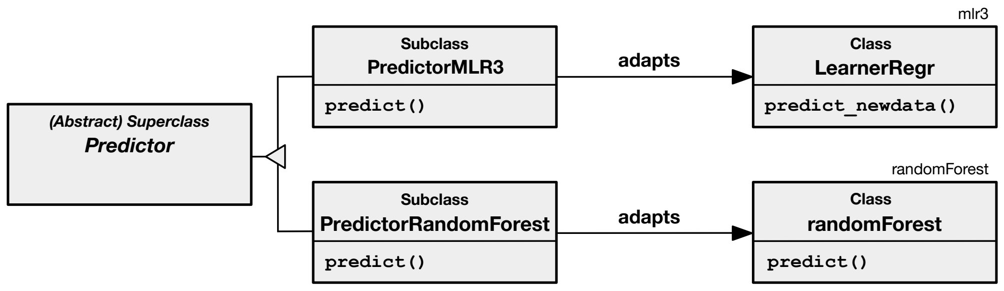

Figure 2: Illustration of the adapter pattern for a simplified example of the Predictor classes in `fme`. The subclass `PredictorMLR3` adapts the `predict_newdata()` method from the class `LearnerRegr` of the `mlr3` package. The subclass `PredictorRandomForest` adapts the `predict()` method from the class `randomForest` of the `randomForest` package. Consequently, all objects of class `Predictor` have a `predict()` method that can be expected to produce predictions of the same type and format.

implemented as strategy becomes easier to understand, as it is contained in its own class. Secondly, encapsulating an algorithm in its own strategy class means the algorithm can be changed independently of the class(es) using it. This design also allows for a flexible and uncomplicated extension of the algorithm: a different implementation of EP detection can be added by defining a subclass of `ExtrapolationDetector` that overwrites the `computeEP()` method. Thirdly, the strategy pattern allows for efficient reuse of algorithms in case they are shared by two or more classes.

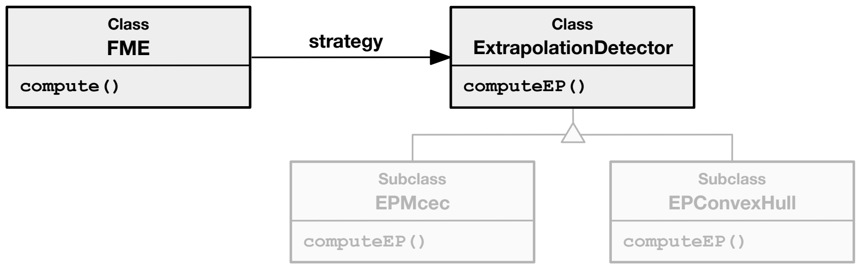

Figure 3: Illustration of the strategy pattern for a simplified example of EP detection in `fme`. The strategy class `ExtrapolationDetector` implements the algorithm for detecting EPs. The class `FME` requires EP detection and forwards this task to `ExtrapolationDetector`. The subclasses `EPMcec` and `EPConvexHull` overwrite the `computeEP()` method in the superclass and implement different methods of the EP detection algorithm, as described in section 2.3.2. Transparency indicates a class is still under development and not part of the current version of `fme`.

3.2 Class Architecture

This section provides an overview of `fme`'s classes and an idea of how they are constructed, how they relate to other classes and how they could be extended. Fields and methods of a class may be referenced for this purpose. For a comprehensive overview of all fields and methods, the reader is referred to the GitHub repository. Fig. 4 illustrates the classes currently implemented or under development and the respective relationships between them.

3.2.1 Predictor Classes

As noted in section 3.1.4, the `Predictor` class is an adapter class which enables a uniform expression of various ML models and the training data. `Predictor` is an abstract superclass and cannot be instantiated. It has the sole purpose of defining fields and methods that each of its subclasses must have. The subclasses of `Predictor` can be instantiated. The current implementation of `fme` has two subclasses: `PredictorMLR3` and `PredictorRandomForest`.

Creating a new instance of a subclass of `Predictor` requires the following arguments:

- `model:` The ML model one wants to use to compute fMEs or categorical MEs. It must possess the ability to generate predictions for an arbitrary observation $\mathbf{x} \in \mathcal{X}$ according to its prediction function $\widehat{f}$.
- `data:` The data one wants to use to compute fMEs or categorical MEs. This can be the data the ML model was trained on. It must be a `data.table` or `data.frame`. In the latter case, it will be transformed to a `data.table` object internally. The column names of `data` must indicate the names of the feature and target variables.
- `target:` A string indicating the name of the target variable.

At instantiation, `Predictor` extracts feature names, feature types and the feature matrix from `data`, and stores the `model`. Throughout `fme`, we rely on `data.table` (Dowle and Srinivasan, 2021) as a dependency to store and handle data frames. This is because it is faster and scales better with larger data sets than base `R`'s `data.frame`.

Furthermore, the class has a `predict()` method that depends on the subclass of `Predictor` that was instantiated. The specific implementation of `predict()` is a consequence of the design of `model`, which varies between different algorithms and packages. At the moment, `Predictor` offers support for a range of ML models through its subclasses:

- `PredictorMLR3:` Provides compatibility for regression models trained with the `mlr3` package (Lang et al., 2019). The package is continuously extended and gives access to a wide range of popular learners.

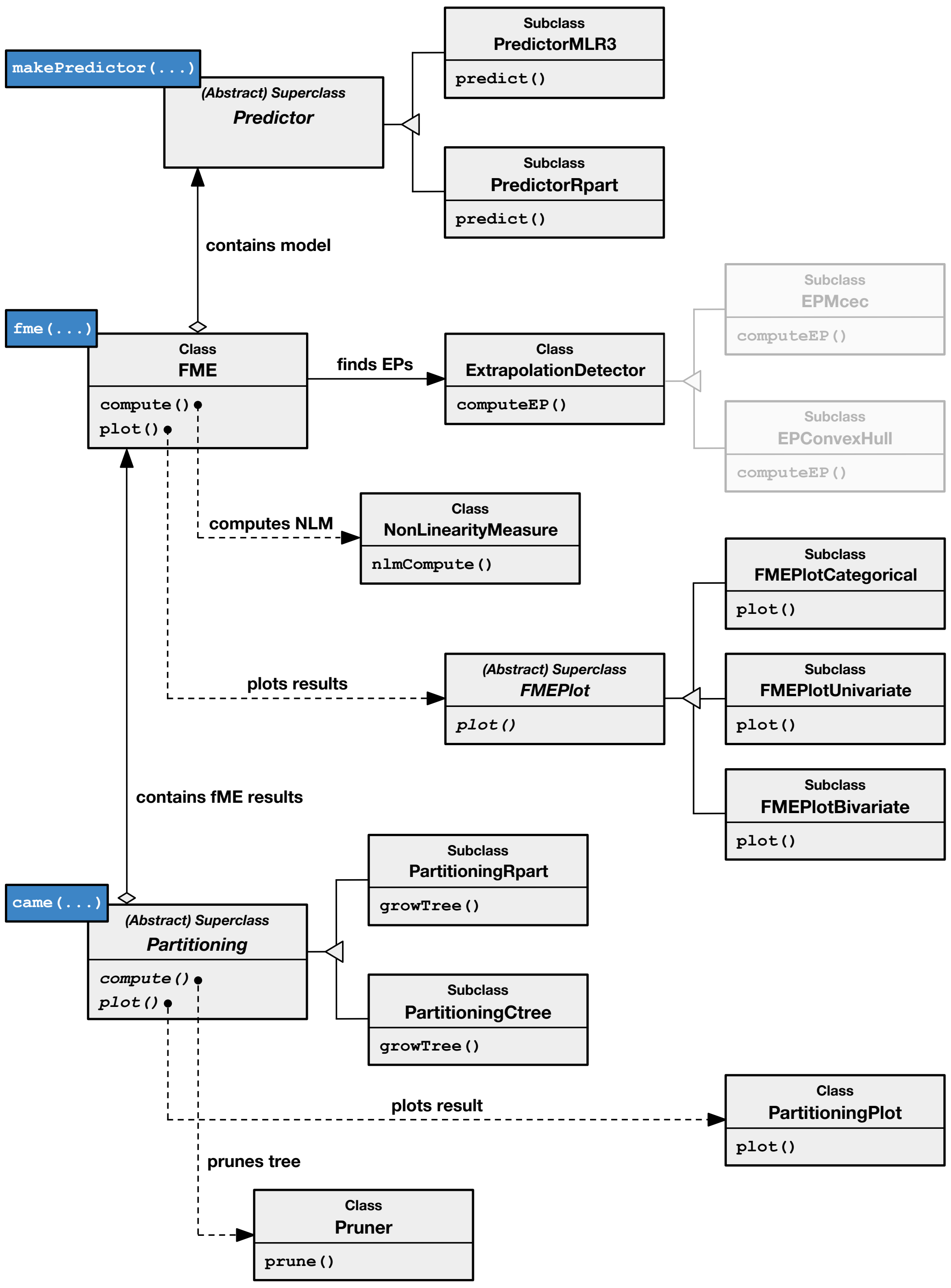

Figure 4: Class diagram of fme's classes and their dependencies. Methods are visualized for the purpose of indicating the main functionality of a class and classes may contain more methods than shown here. Transparency indicates a class is still under development and not part of the current version of fme. Functions in blue boxes are wrapper functions that can be used for instantiation of the respective class.

- `PredictorRandomForest:` Provides compatibility for regression models trained with the `randomForest` package (Liaw and Wiener, 2002).

The design benefits from the polymorphism enabled by `R6`'s class system. As a consequence, `fme` can be extended dynamically to offer compatibility for model objects from other packages by defining a new subclass of `Predictor`. Most of the time, this will require only a few additional lines of code. Furthermore, for enhanced user experience, we provide `makePredictor()` as a wrapper function for initializing the specific subclass required by the class of `model`. The wrapper automatically instantiates the correct subclass, depending on the type of ML model provided as argument. An example of this can be seen in chapter 4.

3.2.2 Forward Marginal Effect Class

After a `Predictor` has been instantiated to adapt the model, one can compute fMEs as in Eq. (3) and categorical MEs as in Eq. (4) with the help of the `FME` class. In theory, the definition of fMEs allows for a multivariate step in p dimensions of the feature space. However, we believe that most of the time, a user would want for a maximum of two features to be affected by the multivariate step, i.e., $|\,S\,| \in \{1, 2\}$. Restricting the fME to univariate and bivariate changes also implies the results can be visualized in a meaningful way.

Creating a new instance of `FME` requires the following arguments:

- `predictor:` The `Predictor` object containing the model and training data.
- `feature:` In the case of the fME, the character vector with the names of the features (one or two) affected by the multivariate step. In the case of the categorical ME, the name of the categorical feature of interest.
- `step.size:` In the case of the fME, the numeric vector of the step lengths $\mathbf{h}_S$. In the case of the categorical ME, the name of the reference category c_h.
- `ep.method:` The method for EP detection. Currently, there are two options: 'none' and 'envelope'. The latter implements the multivariate envelope strategy of Eq. 5 for the numerical features in the training data.
- `compute.nlm:` A logical indicating whether NLMs should be computed for the fMEs.
- `nlm.intervals:` The number of subintervals used for the approximation of the NLM, with a default of one. See section 3.2.5 for details.

After instantiation, the user calls the `compute()` method of the `FME` object to compute MEs. This initiates the following workflow within the object: Firstly, an `Extrapolation-Detector` is instantiated to identify EPs. Subsequently, the set of observations classified as

EP is represented by an index vector stored in the field `ExtrapolationDetector$extra-polation.ids`, which is forwarded to the `FME` object. This is an implementation of the strategy pattern. The next step encompasses computing the MEs for each observation not classified as EP. Note that the algorithm infers automatically from the `feature` argument whether it should compute the fME for a numerical feature or the ME for a categorical feature. In the case of categorical MEs, observations from the reference category in the training data are excluded. In a third optional step, the NLM is computed for each observation's fME if the step type is numerical. This instantiates a `NonLinearityMeasure` object for each observation and forwards the resulting NLM estimate to the `FME` object. This design deploys the strategy pattern, thereby allowing for a structural separation of the `FME` class and the algorithm computing the NLM. After `compute()` is completed, the field `FME$results` contains a `data.table` with MEs and NLMs for all non-EP observations.

In analogy to `Predictor`, `FME` has its own wrapper function. Here, `fme()` is wrapper for `FME$new()$compute()` and returns the `FME` object after computing results. The results can be visualized through the method `plot()`, which creates and plots an `FMEPlot` object.

3.2.3 Forward Marginal Effect Plot Class

`FMEPlot` is another example of how `fme` makes use of both the strategy pattern and the advantages of polymorphism. In principle, we can distinguish between three categories of `FME` objects, each of which demands its own approach to plotting results. Therefore, the algorithm implementing `FME$plot()` is encapsulated in one of the following subclasses of `FMEPlot`:

- `FMEPlotUnivariate:` Creates visualizations for a univariate numerical step, i.e., for the fME as in Eq. 3 with $|S| = 1$. In this case, `$plot()` produces a scatterplot of the non-EP observations, with the feature values $\mathbf{x}_S^{(i)}$ on the x-axis and the effects $\text{fME}_{\mathbf{x}^{(i)}, \mathbf{h}_S}$ on the y-axis. In addition, the AME is displayed as a horizontal line. If one has computed NLM values (by setting the argument `compute.nlm = TRUE`), NLMs can be plotted alongside fMEs in a separate plot through `$plot(with.nlm = TRUE)`. Therefore, it is for the user to decide whether to plot fMEs exclusively or to get a more comprehensive overview by extending the plot with NLMs. Similarly, this includes the ANLM graphed as a horizontal line. As discussed in section 2.3.2, we use a hard lower bound of zero for visualizing NLMs throughout `fme`.

- `FMEPlotBivariate:` Creates visualizations for a bivariate numerical step, i.e., for the fME as in Eq. 3 with $|S| = 2$. In this case, `$plot()` produces a scatterplot of the non-EP observations, with the values of the first feature on the x-axis and the values of the second feature on the y-axis. The colour of a point indicates the direction and magnitude of the effect $\text{fME}_{\mathbf{x}^{(i)},\,\mathbf{h}_S}$, according to a colour scale. Similar to `FMEPlotUnivariate`, NLMs can be plotted next to fMEs with `$plot(with.nlm = TRUE)`. As with the bivariate fMEs, the colour of a point indicates the magnitude of the NLM.

- `FMEPlotCategorical:` Creates visualizations for a categorical step, i.e., for the categorical ME as in Eq. 4 with reference category c_h. In this case, `$plot()` produces a histogram and kernel density estimate plot of the effects $\text{ME}_{\mathbf{x}^{(i)},\,c_h}$ for the non-EP observations. In addition, the AME is displayed as a vertical line.

This design allows to tailor the implementation of `$plot()` to the needs of the underlying `FME` object. The polymorphism that shapes the class facilitates flexible modification. For instance, one can change the visualization of a categorical step by overwriting `FMEPlotCategorical$plot()`, leaving the other subclasses of `FMEPlot` unchanged.

Throughout `FMEPlot`, we rely on `ggplot2` (Wickham, 2016) to create plots and `cowplot` (Wilke, 2020) to arrange multiple plots into a grid. Furthermore, we deliberately utilize the viridis colour scale (Garnier et al., 2021) embedded in `ggplot2` for the visualizations created by `FMEPlotBivariate`. This is because viridis can be perceived by users suffering from the most common types of colour-blindness, a design feature we believe is ignored in many popular `R` packages.

3.2.4 Extrapolation Detector Class

`ExtrapolationDetector` is a strategy class called exclusively by `FME$compute()` to identify EPs by the method specified with `ep.method`. In principle, `ExtrapolationDetector` is implemented to incorporate multiple methods of EP detection through its subclasses (see Fig. 3). The current version of `fme` includes the EP detection method 'envelope', i.e., the multivariate envelope of Eq. 5 for the numerical features in the training data. As a result, `ExtrapolationDetector` computes an index vector `extrapolation.ids` indicating the set of observations that are classified as EPs. As a consequence, `fme` does not in its current form implement any kind of EP detection for categorical MEs.

3.2.5 Non-linearity Measure Class

`NonLinearityMeasure` is a strategy class called exclusively by `FME$compute()` to compute the NLM of an observation as in Eq. 12. As described in section 2.3.2, computing the NLM requires approximating three line integrals. A common way to do so is by Simpson's 3/8 rule (see, e.g., Abramowitz and Stegun, 1972), which is defined for an arbitrary interval $[a, b]$

and a function $f(x)$:

$$\int_a^b f(x) \approx \frac{b-a}{8}\left[f(a) + 3f\left(\frac{2a+b}{3}\right) + 3f\left(\frac{a+2b}{3}\right) + f(b)\right] \tag{18}$$

In `fme`, we use a form of the composite Simpson's 3/8 rule, which divides $[a, b]$ in n subintervals of equal size and approximates each subinterval with Eq. 18. The user can specify n with the argument `nlm.intervals` at the time of instantiation of an `FME` object, with a default value of one. In this way, the user is given the option to reduce the approximation error and get a more precise estimate of the true non-linearity of the prediction function along the forward difference. Of course, this comes at the cost of an increased use of computational resources. This consideration is important as even at `nlm.intervals = 1`, computing NLMs accounted for over 99% of the total computing time of `FME$compute()` in our use case (see chapter 4), a large part of which is spent on repeated evaluations of the prediction function. This is because for each of the three line integrals to be approximated, the prediction function has to be evaluated $4 \times n$ times.

3.2.6 Feature Space Partitioning Classes

The `Partitioning` class implements the idea of facilitating semi-global interpretations through partitioning the feature space $\mathcal{X}$ into a set of mutually exclusive subspaces where MEs are more homogeneous, as discussed in section 2.3.3. Scholbeck et al. (2022) suggest to use the coefficient of variation (CoV) as a scale-invariant measure for the homogeneity of MEs in a subspace, where a lower value indicates higher homogeneity. Here, the CoV is a variable's (sample) standard deviation divided by the absolute value of its mean. A partitioning $\mathcal{P} = \{\mathcal{X}_{[1]}, \ldots, \mathcal{X}_{[m]}\}$ divides the feature space in m partitions, with $\dot{\bigcup}_{i=1}^{m} \mathcal{X}_{[i]} = \mathcal{X}$. The task to find a suitable partitioning is not a trivial one. In principle, we believe there are two criteria that should guide the process by which a partitioning is created:

- The homogeneity of MEs in each partition: Partitioning the feature space only makes sense if one observes heterogeneous MEs across the entire feature space $\mathcal{X}$ to begin with. Furthermore, if the partitioning does not result in more homogeneous subspaces, one will be better off by discarding the partitioning and conducting a global interpretation of MEs.
- The number of partitions m: While a higher number of partitions may enable a more local interpretation of feature effects, it runs the risk of introducing too much complexity, and thereby impeding interpretability. Usually, this is aggravated when the number of dimensions of the feature space that are affected by the partitioning is high.

For example, partitionings that affect one, two or three dimensions of the feature space can be illustrated graphically. However, this is not reasonably possible with more than three dimensions.

In `fme`, we define two alternative approaches to finding a suitable partitioning $\mathcal{P}$:

- `max.cov`: Find a partitioning $\mathcal{P}$ where the MEs in each partition have a CoV of at most `max.cov`. Among multiple partitionings that fulfil this criterion, select the one with the lowest number of partitions m for reduced complexity. The user can specify `max.cov`, thus ensuring a certain degree of homogeneity is attained in every subspace.
- `number.partitions`: Find a partitioning $\mathcal{P}$ with an exact number of partitions m, where `number.partitions` is specified by the user. This can be helpful in situations where a user may not necessarily want to specify a minimum degree of homogeneity but wants to find out whether partitioning may lead to more homogeneous subspaces. Furthermore, this approach ensures that the complexity of $\mathcal{P}$ is limited by holding m fixed. Of course, it must be ensured that the way such partitionings are identified can be expected to produce (relatively) homogeneous subspaces.

As mentioned before, RP algorithms like CART and CTREE represent an efficient way to produce decision trees that can be used to partition the feature space into mutually disjoint subspaces. However, they have two major drawbacks: Firstly, neither CART nor CTREE use the CoV as a split criterion. For instance, CART splits a parent node such that the between-groups sum-of-squares is minimized (Therneau and Atkinson, 2022b). This means they may not necessarily provide the most efficient way to optimize for lower CoV in the child nodes. Secondly, none of the two have a stopping criterion w.r.t. the number of terminal nodes, i.e., the number of partitions m.

In `fme`, we try to address these issues as best as possible by inducing the RP algorithm (either CART or CTREE) to produce a relatively large tree and prune the tree until a CoV-related optimality criterion is met. We find that such an approach can work well to identify partitionings with substantially more homogeneous subspaces. In addition, results can be obtained in a reasonable amount of time owing to the greediness of the RP algorithms used to grow the tree. For the `max.cov` approach, we iteratively prune the tree until only the root node is left. Among all the decision trees which are created as intermediate steps of this procedure, we identify the smallest tree (with the lowest number of terminal nodes) that fulfils the `max.cov` criterion. For the `number.partitions` approach, we simply prune the tree until the tree has exactly as many terminal nodes as specified.

Partitioning class

`Partitioning` is an abstract class with two subclasses, each of which uses a different RP algorithm to grow a large decision tree that can be used for subsequent pruning:

- `PartitioningCtree:` The partitioning is identified with a decision tree created by the CTREE algorithm, which is part of the `R` package `partykit` (Hothorn and Zeileis, 2015).
- `PartitioningRpart:` The partitioning is identified with a decision tree created by the CART algorithm, implemented in the `R` package `rpart` (Therneau and Atkinson, 2022a).

After one has computed MEs and the results are stored in an `FME` object, a partitioning can be constructed by instantiating a `Partitioning` object and computing the result with the wrapper function `came()` (an allusion to cAME), which requires the following arguments:

- `effects:` An `FME` object with MEs computed.
- `number.partitions:` The number of partitions m, implements the eponymous strategy explained above. The minimum possible value is 2, the maximum is 8. This argument can be omitted if the user desires the `max.cov` approach.
- `max.cov:` The maximum CoV of MEs in every partition, implements the eponymous strategy. This argument can be omitted if the user desires the `number.partitions` approach.
- `rp.method:` The RP algorithm used to create the initial (large) tree. Currently, there are two options: 'rpart' and 'ctree'. This determines the respective subclass of `Partitioning` that will be instantiated.
- `tree.control:` A control argument overwriting the defaults for the RP algorithm that is induced to grow a large tree. This must match the structure of `rpart.control` or `ctree.control`. This argument is for the experienced user who wants to finetune the behavior of the RP algorithm. This argument can be omitted.

Note that the user has to select one of two alternative strategies. If the procedure is able to find a partitioning that fulfils the respective criterion, the decision tree which represents the final partitioning is stored in the field `tree`. We use the `party` class from the `partykit` package to represent decision trees independently of the RP algorithm, as `rpart()` and `ctree()` express decision trees in different formats. This uniform representation of tree structures and the polymorphism of the `Partitioning` class allow for a flexible future extension to other RP algorithms.

Sometimes, `came()` may not be able to identify a suitable partitioning. Often, this is due to `max.cov` being chosen too low. Furthermore, in our experience the `max.cov` criterion may lead to higher homogeneity (as compared to the root node) in most but not all terminal nodes . Sometimes, this is the case when the cAME of a terminal node is close to zero, which can lead to a very high CoV as the (absolute of the) cAME represents the denominator in the CoV formula.

After a partitioning has been found, one can extract instructive summary statistics of the partitions from the field `results`, or simply inspect the results in a reader-friendly way by calling `summary()`. In both cases, one is provided with information regarding the number of partitions, the number of observations assigned to each partition, the cAME, the CoV of MEs in each partition and, if applicable, the ANLM and the CoV of NLMs in each partition. Moreover, as the partitions are represented as terminal nodes in the corresponding decision tree, `results` provides the same information for all parent nodes in the tree. In this way, the user may gain insights into how further pruning of the tree would affect the homogeneity of MEs. In addition, a partitioning and the corresponding descriptive statistics can be visualized by calling `plot()`. This instantiates a `PartitioningPlot` object.

Pruner class

`Pruner` is a strategy class called by `Partitioning$compute()` to prune a tree. After a large tree has been constructed by the RP algorithm, `Pruner` is used to iteratively reduce the complexity of the tree until it has the desired size. Firstly, `Pruner` identifies all pairs of terminal nodes in the tree that share the same parent node and are therefore candidates for pruning. Then, the CoV of MEs is computed for each parent node of the candidate nodes. Lastly, pair of the candidate nodes with the parent node with the lowest CoV is pruned. The merit of this procedure is that pruning is conducted in a fashion inclined towards selecting the most homogeneous parent node. `Pruner` is called iteratively by `Partitioning$compute()` until the desired result is obtained.

Partitioning plot class

`PartitioningPlot` is a strategy class called by `Partitioning$plot()` that contains all necessary functionality to visualize a partitioning, returning a `ggplot` object. It consists of three parts:

- A graphic representation of the decision tree that describes the partitioning. This includes the names of the feature variables used for splitting the nodes, the (numeric and categorical) split points and lines connecting the labels to illustrate the tree structure.
- Descriptive statistics for each partition, equivalent to the information contained in the

field `Partitioning$results`. As every partition corresponds to a terminal node in the decision tree, the statistics appear right below the terminal node they belong to.

- Histograms of the MEs in each partition, including the cAME as a vertical line. Furthermore, if applicable, histograms of the NLMs in each partition, including the cANLM as a vertical line. As before, we use zero as a hard lower bound for visualizing NLMs.

We rely on the `ggparty` package (Borkovec and Madin, 2019) to represent the tree-like structures contained in `party` objects in a format that is compatible with `ggplot`.

4 Demo

Fme was designed to provide an easily extensible, modular framework for computing and interpreting fMEs for arbitrary supervised regression models. This chapter gives an overview of the necessary steps required for estimating and computing feature effects with fme. It illustrates its main functionalities and showcases how common user demands are addressed. The example introduced below can be reproduced by downloading and running the script from https://github.com/holgstr/fme/blob/master/inst/Demo.R.

4.1 Installation, Data and Model

Installation

The latest version of fme can be installed and loaded directly from the console with the devtools package (Wickham et al., 2021):

```
library(devtools)
install_github("holgstr/fme")
library(fme)
```

Data: bike sharing demand in Washington, D.C.

For demonstration purposes, we will consider usage data from the Capital Bike Sharing scheme in Washington, D.C. (Fanaee-T and Gama, 2014). It was obtained from the OpenML database (Vanschoren et al., 2013) and contains information about hourly bike sharing usage in Washington, D.C. for the years 2011-2012. The original data set has 17,379 observations and 15 variables. Here, we consider a subset of the data: We are interested in predicting count (the total number of bikes lent out to users) during the period from 7 to 8 a.m. The data used for training the model has 727 observations and 11 variables. Table 1 provides an overview of the data.

Model

Fme was designed for arbitrary regression models. Currently, support for randomForest and mlr3 is provided. However, users can create their own Predictor subclass to adapt other models with a few lines of code. In this example, we train a random forest with ranger with the mlr3 framework, predicting count with the remaining variables as features of the model:

```
library(mlr3verse)
task = as_task_regr(x = bikes, id = "bikes", target = "count")
forest = lrn("regr.ranger")$train(task)
```

Variable Name	Description	Range
season	Season of the year (spring, summer, autumn, winter).	{0, 1, 2 , 3}
year	Year (2011, 2012).	{0, 1}
month	Month of the year.	[1, 12]
weekday	Day of the week.	[1, 7]
holiday	If a day is a holiday (yes, no).	{0, 1}
workingday	If a day is neither weekend nor holiday (yes, no).	{0, 1}
weather	Weather situation (clear, misty, rain).	{0, 1, 2}
temp	Temperature in degrees Celsius (°C).	[0.82, 34.44]
humidity	Humidity (relative).	[0, 1]
windspeed	Windspeed in miles per hour (mph).	[0.00, 40.99]
count (target)	Total number of bikes lent out to users.	[6, 839]

Table 1: Description of the training data from the Washington, D.C. bike sharing scheme.

4.2 Computing and Interpreting Marginal Effects

The regression model trained in section 4.1 can be adapted to `fme` by instantiating the corresponding subclass of `Predictor`:

```
pred = PredictorMLR3$new(model = forest, data = bikes, target = "count")
```

Alternatively, this can be done with a wrapper function, which automatically creates the correct subclass of `Predictor`:

```
pred = makePredictor(model = forest, data = bikes, target = "count")
```

The `Predictor` class is important for the inner mechanics of `fme` and for users who want to adapt their own models through self-made subclasses. However, instantiating a `Predictor` is not required in the standard workflow for computing MEs. `Fme` provides wrapper functions for all necessary functions in the package. As R6 syntax can be confusing to some users, we recommend to use the wrappers and will do so for the remainder of this chapter.

Categorical features

The default workflow for computing MEs for categorical and numerical features requires only one function: `fme()`. To view the documentation, simply type `?fme` in the console. Assume we want to compute the ME for a categorical feature as in Eq. 4. For instance, one could be interested in the effect of rainy weather on the bike sharing demand, i.e., the ME of changing the feature `weather` to `'rain'` for observations where `weather` is either `'clear'` or `'misty'`:

```
effects = fme(model = forest, data = bikes, target = "count",
              feature = "weather", step.size = "rain")
```

By doing so, we have created an object of class `FME`:

```
class(effects)
##[1] "FME" "R6"
```

As is common with `R` objects, we can produce a summary for `effects` to inspect it:

```
summary(effects)
##
## Forward Marginal Effects Object
##
## Step type:
##   categorical
##
## Feature & reference category:
##   weather, rain
##
## Extrapolation point detection:
##   none, EPs:  0 of 657 obs.  (0 %)
##
## Average Marginal Effect (AME):
##   -53.801
```

As `weather` is a factor variable, `fme` inferred that this implies a categorical step. Since the default option for EP detection is `'none'`, no EPs were detected in the training data. We can see the that the estimated AME is -53.801. This means that on average, we can expect rainy weather to contribute negatively to bike sharing usage, as compared to clear and misty weather. Note that this interpretation does not allow for a comparison between rainy and misty or rainy and clear days, respectively. Rather, we compare rainy to non-rainy days. In our training data, non-rainy days are comprised of 214 misty and 443 clear days. For further analysis, we could extract the estimated AME from the `FME` object or inspect MEs computed for individual observations:

```
effects$ame
##[1] -53.80098

head(effects$results)
##   obs.id         fme
##1:      1   0.7416677
##2:      3  -6.3319125
##3:      4 -12.8569499
##4:      5 -13.6807424
##5:      7 -11.5924678
##6:      9   1.4546422
```

For categorical feature steps, we can visualize the empirical distribution of the estimated MEs:

```
plot(effects)
```

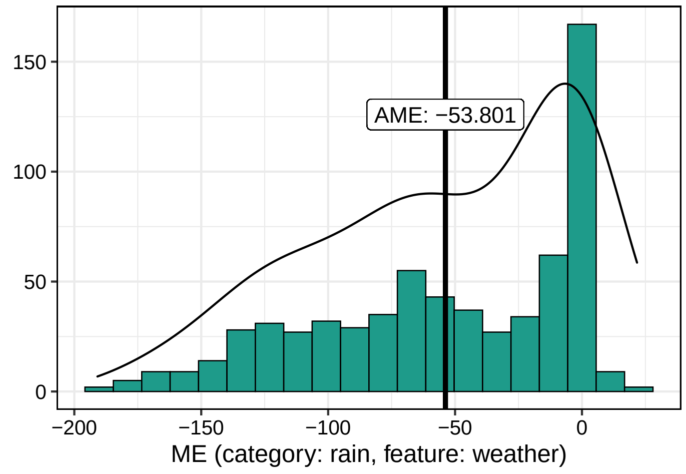

Figure 5: MEs for the feature change in `weather` with `step.size = 'rain'`.

The output of `plot(effects)` is displayed in Figure 5. The histogram and kernel density estimate have negative skewness. For more than 150 observations, the ME is close to zero. While few observations exhibit positive MEs, the majority has negative MEs, with values up to minus 190.

Numerical features

Next, we consider numerical features, beginning with a univariate change. For instance, one might be interested in the fME of an increase in temperature by 3 degrees Celsius (°C) on bike sharing usage. Thus, we compute fMEs for the feature `temp` and `step.size = 3`. Moreover, to examine the shape of the prediction function along the forward difference, we compute NLMs (depending on the computer, this can take some time[2]). We exclude observations classified as EP with `ep.method = 'envelope'`:

```
effects2 = fme(model = forest, data = bikes, target = "count",
               feature = "temp", step.size = 3, ep.method = "envelope",
               compute.nlm = TRUE)

effects2$ame)
##[1] 8.70945

effects2$anlm)
##[1] 0.06
```

[2]approx. three minutes on a 2021 Macbook Air.

We estimate an AME of 8.7 and an ANLM of 0.06. The latter indicates a highly non-linear prediction function along the forward difference. Again, we visualize results with `plot()`. Furthermore, we include NLMs in the plot and add minimal noise to the individual points to reduce overlapping:

```
plot(effects2, with.nlm = TRUE, jitter = c(0.2, 0))
```

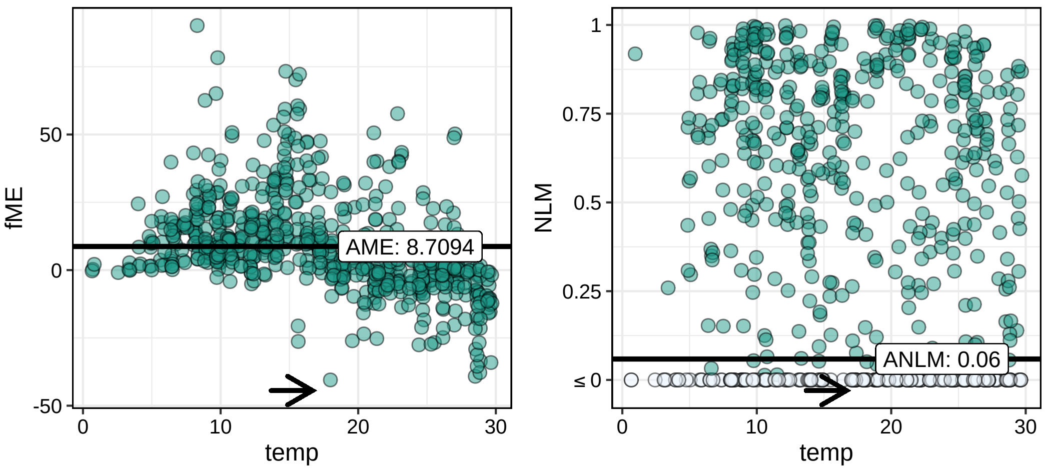

Figure 6: fMEs and NLMs for the feature change in `temp` with `step.size = 3`. Direction and length of the arrows indicate the step length of the feature change. Negative NLM values are white colored.

The output is visualized in Fig. 6. The effect of a temperature increase by 3 °C on bike sharing usage seems to vary for different values of `temp`. While the fMEs tend to be positive for lower temperatures between 0 °C and 17 °C, they tend to be negative for higher temperatures above 17 °C. Next, we consider bivariate changes in feature values. For instance, assume one wants to estimate the effect of a decrease in temperature by 2 °C combined with a decrease in humidity by 10 percentage points, i.e., the fME for `feature = c(temp, humidity)` and `step.size = c(−2, −0.1)`:

```
effects3 = fme(model = forest, data = bikes, target = "count",
               feature = c("temp", "humidity"), step.size = c(-2, 0.1),
               ep.method = "envelope", compute.nlm = TRUE)
```

```
plot(effects3, with.nlm = TRUE, jitter = c(0.1, 0,02))
```

Fig. 7 represents the corresponding output. We could extract summary statistics from the `FME` object in the same way as before, and get AME = 0.64 and ANLM = 0. Whereas the AME is a global estimate of the expected ME, effects could be heterogeneous. Thus, we will consider semi-global interpretations in the next step.

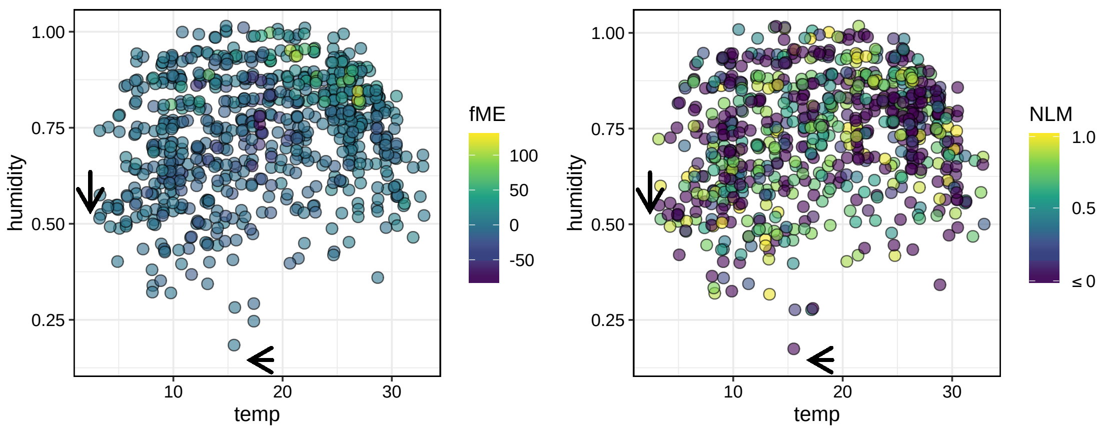

Figure 7: fMEs and NLMs for the bivariate feature change in `(temp, humidity)` with `step.size = (−2, −0.1)`. Direction and length of the arrows indicate the step length of the feature changes.

4.3 Finding Feature Subspaces for Semi-Global Interpretations

For the purpose of enabling semi-global interpretations on feature subspaces, `fme` provides the `came()` function. To view the documentation, simply type `?came` in the console. Assume one wants to investigate whether the fMEs computed in the example of Fig. 7 can be assigned to subspaces of the feature space where effects are more homogeneous. In this example, we are interested in finding a partitioning with three partitions:

```
subspaces = came(effects = effects3, number.partitions = 3)
```

The `came()` function throws an error if the RP algorithm did not succeed in finding a suitable partitioning. As usual, we can inspect the result with `summary()`:

```
summary(subspaces)
##
## PartitioningCtree of an FME object
##
## Method:  partitions = 3
##
##    n        cAME    CoV(fME)          cANLM    CoV(NLM)
## 722   0.6382352   32.079175   -0.003834562  247.513745 *
## 358  -6.7855826    2.337526    0.081459950   10.658651
## 295   5.1501882    3.554864   -0.143634479    7.106654
##  69  19.8657809    1.527629    0.151318200    6.410529
## ---
## * root node (non-partitioned)
## cANLM: ≤ 0 implies non-linearity, 1 implies linearity
##
```

```
## AME (Global):  0.6382
## ANLM (Global):  0
```

We can infer from the summary that the fMEs in the root node have a CoV of 32. Furthermore, we can see that the CoV is substantially smaller in the three terminal nodes (partitions). Thus, one can conclude that the partitioning successfully identified feature subspaces with more homogeneous effects. The best way to investigate the partitions is to plot the `Partitioning` object:

```r
plot(subspaces)
```

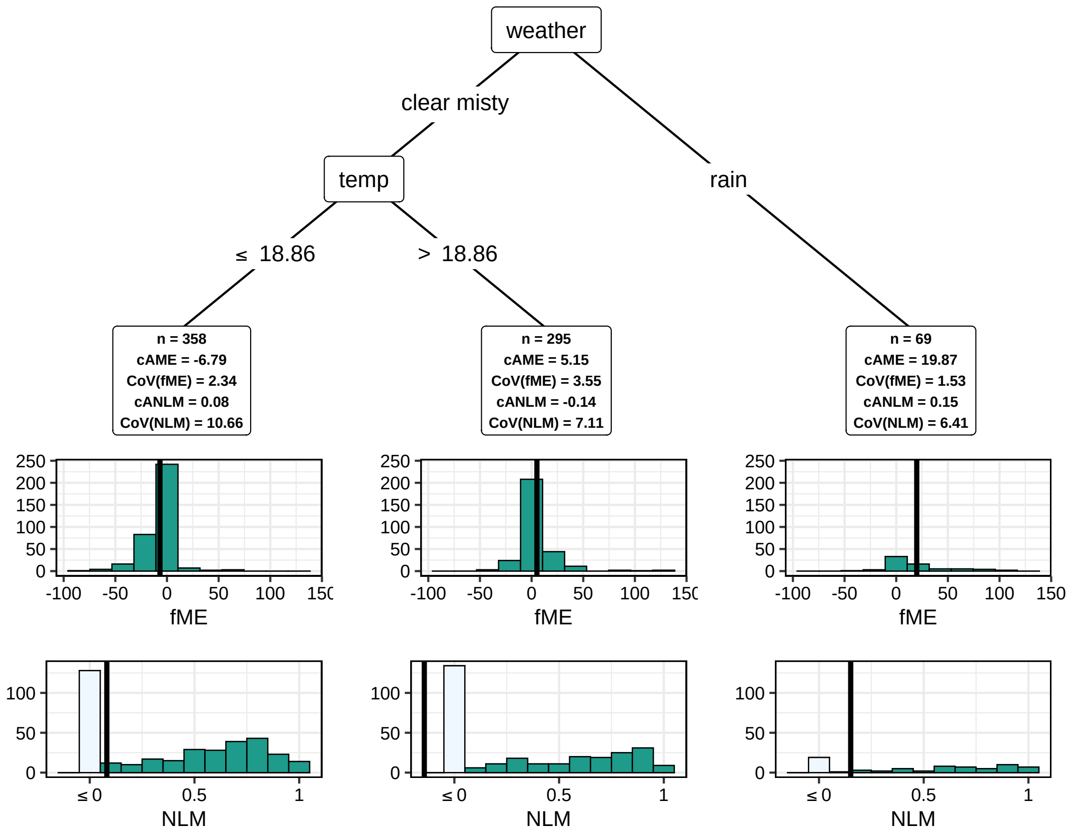

Figure 8: Partitioning of `effects3` with `number.partitions = 3`. The histograms show the distribution of fMEs and NLMs within the partitions.

In this case, we get a decision tree that assigns observations to a feature subspace according to the weather situation (`weather`) and temperature (`temp`). The information contained in the boxes below the terminal nodes are equivalent to the summary output and can be extracted from `subspaces$results`. This redundancy makes it as easy as possible for users to find and access relevant data for interpretation and further analysis of a partitioning. With cAMEs of -6.79, 5.15, and 19.87, respectively, the expected ME is estimated to vary substantially in direction and magnitude across the subspaces. For example, the cAME is negative for non-rainy days with temperatures below 18.9 °C. For non-rainy days with temperatures above 18.9 °C, the cAME turns positive. Lastly, the cAME is the highest on rainy days. In all three partitions, the cANLM indicates a highly non-linear shape of the prediction function with values ranging from -0.14 to 0.15.

The alternative method of finding partitionings is `max.cov`, which tries to find the smallest tree that conforms to a maximum CoV of MEs in all partitions. For instance, recall the example of the effect of a change in the categorical feature `weather` to the reference category `'rain'` (see Fig. 5). We can try to find a partitioning of the corresponding `FME` object with `max.cov = 3` and a custom specification of `rpart` with the following code:

```
subspaces2 = came(effects = effects, max.cov = 3, rp.method = "rpart",
                  tree.control = rpart.control(minsplit = 100, cp= 0.1))
plot(subspaces2)
```

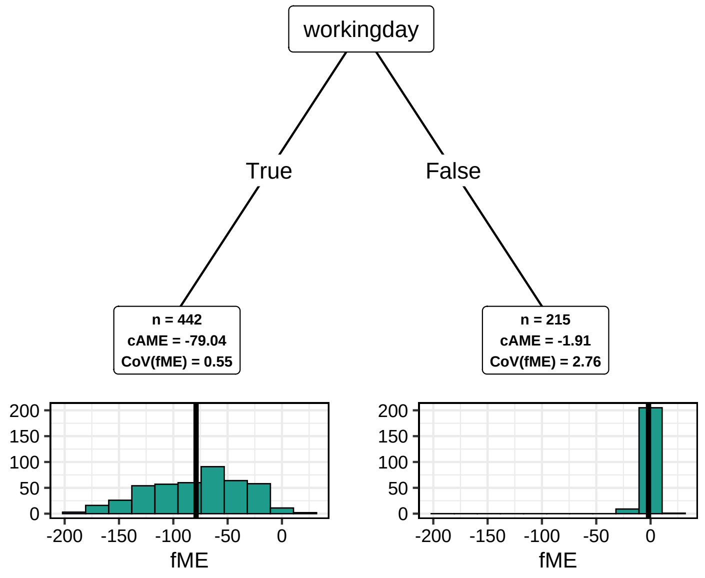

Figure 9: Partitioning of `effects` with `max.cov = 3`. The histograms show the distribution of fMEs within the partitions.

As visualized in Fig. 9, this can be achieved with only two subspaces, represented by a decision tree with a split in the feature `workingday`. Thus, the ME of rainy weather seems to depend on whether a day is a working day or not. On working days, the estimated ME of rain is -79.04. On non-working days, the effect seems to be much smaller, with a cAME of -1.91. In both partitions, the CoV is smaller than `max.cov = 3`. In our experience, the `max.cov` method often results in partitionings of that size ($m = 2$).

In section 3.1.2, we explained the merit of defensive programming as a guiding principle in software design. `Fme` seeks to improve the user experience by ensuring that sensible inputs are provided to its functions, and throws informative error messages if otherwise. For instance, assume a user is confused by the way partitionings can be identified and provides two incompatible arguments to `came()`:

```
came(effects = effects2, max.cov = 1.5, number.partitions = 8)
##Error in came(effects = effects, max.cov = 1.5, number.partitions = 2) :
##  Must supply either 'number.partitions' or 'max.cov'.
```

Note that in its current form, `fme` allows for a maximum of eight partitions in a partitioning. This is due to limitations in `ggparty`'s ability to generate plots for decision trees of this size. An example of a partitioning created with `number.partitions = 8` for a feature change in `temp` with `step.size = 3` is visualized in Fig. 10 and can be reconstructed with the following code:

```
plot(came(effects = effects2, number.partitions = 8))
```

As noted before, a partitioning represented by a large decision tree can impede interpretability. In this example, the decision tree has split points in five features: `season`, `weather`, `temp`, `workingday`, and `year`. Moreover, as with `temp`, features can be selected repeatedly as split points, further complicating the interpretation.

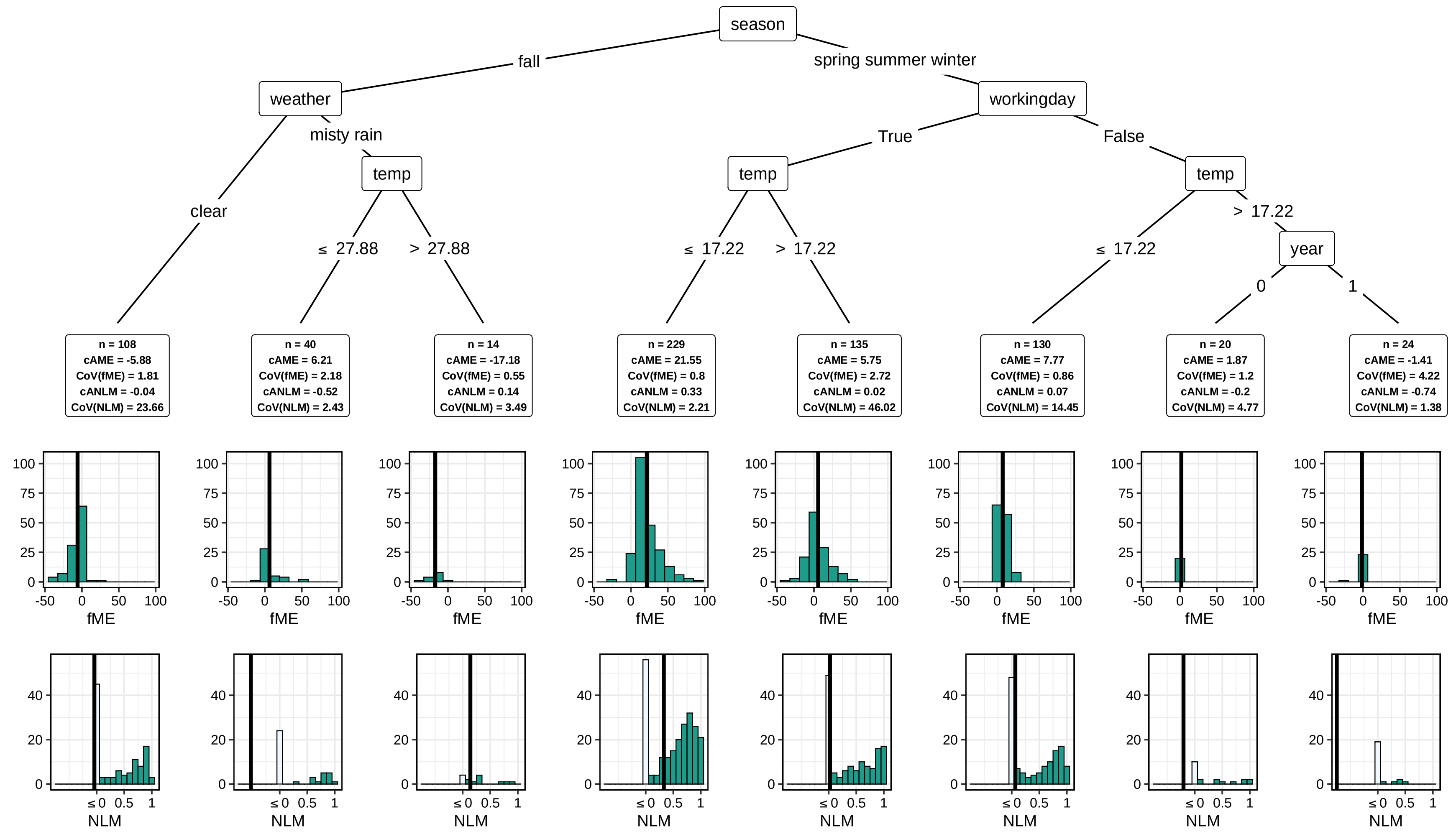

Figure 10: Partitioning of `effects2` with `number.partitions = 8`. The histograms show the distribution of fMEs and NLMS within the partitions.

5 Conclusion and Outlook

Throughout this thesis, we have established the merit of fMEs, a new class of MEs, as a relevant contribution to the family of methods for interpreting feature effects in supervised ML. In this context, we introduced `fme`, a fully functional `R` package for computing, analyzing and visualizing fMEs. This includes fMEs for univariate and bivariate changes in numerical features, and observation-wise categorical MEs for changing feature values to a reference category. Additionally, `fme` implements further elements of the application workflow suggested by Scholbeck et al. (2022): detecting EPs, computing NLMs, and interpretations conditional on feature subspaces.

As described in chapter 3, `fme` benefits from the R6 OOP framework, which allows for a modular design that is open to future extensions in its functionality. While `fme` exploits the advantages of an object-oriented design, it circumvents the downsides of the relatively complicated R6 syntax by providing wrapper functions for a more intuitive user interface. The use of assertions for checking user-provided arguments and a documentation of the main functions contribute to a smooth user experience.

In principle, `fme` can be applied to arbitrary supervised regression models in `R`. Through its native support of `mlr3`, it already extends its coverage to a multitude of different learners. By implication, this means at present `fme` is the only `R` package that can be used to compute MEs for tree-based models, such as random forests or gradient-boosted trees.

Moreover, `fme` facilitates semi-global interpretations of fMEs through finding feature subspaces with more homogeneous effects that are identified with RP algorithms. In this respect, its class design enables a uniform representation and visualization of decision trees created by different RP methods. Nonetheless, we have identified several ways by which `fme` could be improved, outlining a clear path for the further development of the package.

Extension to classification models

The current version of `fme` is restricted to regression tasks. However, as Scholbeck et al. (2022) note, the concept of fMEs can be generalized to multi-dimensional predictions, e.g., multi-class classification. In addition, the current class design of `fme` can be straightforwardly applied to binary classification tasks. In principle, this should merely require an additional subclass of `Predictor` and some minor changes to `FMEPlot`.

Uncertainty quantification for AMEs and cAMEs

Scholbeck et al. (2022) suggest to quantify the uncertainty in effect estimation by constructing confidence intervals (CI) for AMEs and cAMEs. Equivalently, CIs can be computed for the ANLM and cANLM, respectively.

Speeding up NLM estimation

At the moment, computing NLMs accounts for almost the entire run time of `fme()`. Therefore, one priority for future development efforts should be to accelerate NLM estimation. As NLMs are computed observation-wise, the obvious way to do so is through parallelization, which exploits modern processor architectures to perform multiple calculations simultaneously (McCallum and Weston, 2011). Furthermore, efforts should be made to investigate whether the current class design of `NonLinearityMeasure` can be improved, as we require a separate instance of this class for each observation to compute the NLM. Lastly, one could consider using alternative methods for numerical integration, some of which would entail fewer evaluations of the prediction function and, therefore, shorter run times.

Alternatives to `rpart` and `ctree`

In chapter 3, we discussed the shortcomings of `rpart` and `ctree` w.r.t. the objectives of `fme`. In this context, we believe a comparable RP method with the number of terminal nodes as additional optional stopping criterion and with the CoV as loss function would likely result in a more streamlined process of finding suitable feature subspaces. This becomes even more important as the current implementation is not straightforward to understand, a potential deterrent to its adoption by users.

More refined semi-global interpretations

Another avenue of further development may reside in enhancing the interpretability of feature subspaces. In some way, this relates to addressing the demands of different research domains. For instance, consider the example of bike sharing usage in Washington, D.C. introduced in chapter 4. Assume one has computed the MEs for a change in the categorical feature `weather` to the reference category `'rain'`. If we now want to determine whether MEs are heterogeneous w.r.t. to the temperature (`temp`) (as compared to the entire feature space), the current implementation fails to provide meaningful tools to investigate this question. Thus, the design of the `Partitioning` class could be modified to identify subspaces only in a specific subset of the features.

On a further note, some researchers may only be interested in finding subspaces w.r.t. the features that were affected by the multivariate step. In the example of the bivariate feature change in `temp` and `humidity`, this means that partitions would be described entirely by two-dimensional intervals of these variables. This comes with the advantage that such a partitioning can be visualized in the same plot as the fMEs, with intervals graphed as vertical and horizontal lines.

List of Figures

List of Tables

References

Abramowitz, M. and Stegun, I. A. (1972). Handbook of mathematical functions, with formulas, graphs, and mathematical tables, *Dover books on advanced mathematics.* pp. 885–887.

Alvarez-Melis, D. and Jaakkola, T. S. (2018). On the robustness of interpretability methods, *arXiv preprint arXiv:1806.08049* .

Apley, D. W. and Zhu, J. (2020). Visualizing the effects of predictor variables in black box supervised learning models, *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* **82**(4): 1059–1086.

Arel-Bundock, V. (2022). *marginaleffects: Marginal Effects, Marginal Means, Predictions, and Contrasts.* R package version 0.5.0.
URL: *https://CRAN.R-project.org/package=marginaleffects*

Bartus, T. (2005). Estimation of marginal effects using margeff, *The Stata Journal* **5**(3): 309–329.

Bischl, B., Binder, M., Lang, M., Pielok, T., Richter, J., Coors, S., Thomas, J., Ullmann, T., Becker, M., Boulesteix, A. et al. (2021). Hyperparameter optimization: Foundations, algorithms, best practices and open challenges, *arXiv preprint arXiv:2107.05847* .

Borkovec, M. and Madin, N. (2019). *ggparty: 'ggplot' Visualizations for the 'partykit' Package.* R package version 1.0.0.
URL: *https://CRAN.R-project.org/package=ggparty*

Breiman, L. (2001). Random forests, *Machine learning* **45**(1): 5–32.

Breiman, L., Friedman, J., Olshen, R. and Stone, C. (1984). Classification and regression trees.

Chang, W. (2021). *R6: Encapsulated Classes with Reference Semantics.* R package version 2.5.1.
URL: *https://CRAN.R-project.org/package=R6*

Doshi-Velez, F. and Kim, B. (2017). Towards a rigorous science of interpretable machine learning, *arXiv preprint arXiv:1702.08608* .

Dowle, M. and Srinivasan, A. (2021). *data.table: Extension of 'data.frame'.* R package version 1.14.2.
URL: *https://CRAN.R-project.org/package=data.table*

Fanaee-T, H. and Gama, J. (2014). Event labeling combining ensemble detectors and background knowledge, *Progress in Artificial Intelligence* **2**(2): 113–127.

Fernihough, A. (2019). *mfx: Marginal Effects, Odds Ratios and Incidence Rate Ratios for GLMs.* R package version 1.2-2.
URL: *https://CRAN.R-project.org/package=mfx*

Fox, J. (2003). Effect displays in R for generalised linear models, *Journal of Statistical Software* **8**(15): 1–27.

Freitas, A. A. (2014). Comprehensible classification models: a position paper, *ACM SIGKDD explorations newsletter* **15**(1): 1–10.

Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine, *Annals of statistics* pp. 1189–1232.

Gamma, E., Helm, R., Johnson, R., Johnson, R. E., Vlissides, J. et al. (1995). *Design patterns: elements of reusable object-oriented software*, Pearson.

Garnier, Simon, Ross, Noam, Rudis, Robert, Camargo, Pedro, A., Sciaini, Marco, Scherer and Cédric (2021). *viridis - Colorblind-Friendly Color Maps for R.* R package version 0.4.0.
URL: *https://sjmgarnier.github.io/viridis/*

Goldstein, A., Kapelner, A., Bleich, J. and 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.

Greene, W. H. (2012). *Econometric analysis*, Pearson Education.

Hastie, T., Tibshirani, R., Friedman, J. H. and Friedman, J. H. (2009). *The elements of statistical learning: data mining, inference, and prediction*, Vol. 2, Springer.

Hothorn, T., Hornik, K. and Zeileis, A. (2006). Unbiased recursive partitioning: A conditional inference framework, *Journal of Computational and Graphical statistics* **15**(3): 651–674.

Hothorn, T. and Zeileis, A. (2015). partykit: A modular toolkit for recursive partytioning in R, *Journal of Machine Learning Research* **16**: 3905–3909.
URL: *https://jmlr.org/papers/v16/hothorn15a.html*

King, G. and Zeng, L. (2006). The dangers of extreme counterfactuals, *Political analysis* **14**(2): 131–159.

Lang, M. (2017). checkmate: Fast argument checks for defensive R programming, *The R Journal* **9**(1): 437–445.

Lang, M., Binder, M., Richter, J., Schratz, P., Pfisterer, F., Coors, S., Au, Q., Casalicchio, G., Kotthoff, L. and Bischl, B. (2019). mlr3: A modern object-oriented machine learning framework in R, *Journal of Open Source Software* .
URL: *https://joss.theoj.org/papers/10.21105/joss.01903*

Leeper, T. J. (2021). *margins: Marginal Effects for Model Objects.* R package version 0.3.26.

Liaw, A. and Wiener, M. (2002). Classification and regression by randomforest, *R News* **2**(3): 18–22.
URL: *https://CRAN.R-project.org/doc/Rnews/*

Lundberg, S. M. and Lee, S.-I. (2017). A unified approach to interpreting model predictions, *Advances in neural information processing systems* **30**.

McCallum, Q. E. and Weston, S. (2011). Parallel r.

Molnar, C. (2022). *Interpretable Machine Learning.*

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

Molnar, C., König, G., Herbinger, J., Freiesleben, T., Dandl, S., Scholbeck, C. A., Casalicchio, G., Grosse-Wentrup, M. and Bischl, B. (2020). General pitfalls of model-agnostic interpretation methods for machine learning models, *arXiv preprint arXiv:2007.04131* .

Morandat, F., Hill, B., Osvald, L. and Vitek, J. (2012). Evaluating the design of the r language, *ECOOP 2012–Object-Oriented Programming* p. 104.

Murdoch, W. J., Singh, C., Kumbier, K., Abbasi-Asl, R. and Yu, B. (2019). Definitions, methods, and applications in interpretable machine learning, *Proceedings of the National Academy of Sciences* **116**(44): 22071–22080.

Nelder, J. A. and Wedderburn, R. W. (1972). Generalized linear models, *Journal of the Royal Statistical Society: Series A (General)* **135**(3): 370–384.

R Core Team (2021). *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria.
URL: *https://www.R-project.org/*

Ribeiro, M. T., Singh, S. and Guestrin, C. (2016a). Model-agnostic interpretability of machine learning, *arXiv preprint arXiv:1606.05386* .

Ribeiro, M. T., Singh, S. and Guestrin, C. (2016b). Why should i trust you? explaining the predictions of any classifier, *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 1135–1144.

Scholbeck, C. A., Casalicchio, G., Molnar, C., Bischl, B. and Heumann, C. (2022). Marginal effects for non-linear prediction functions, *arXiv preprint arXiv:2201.08837* .

Slack, D., Hilgard, S., Jia, E., Singh, S. and Lakkaraju, H. (2020). Fooling lime and shap: Adversarial attacks on post hoc explanation methods, *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, pp. 180–186.

StataCorp (2019). *Stata: Release 16*, StataCorp LLC., College Station, TX.
URL: *https://www.stata.com/*

Štrumbelj, E. and Kononenko, I. (2014). Explaining prediction models and individual predictions with feature contributions, *Knowledge and information systems* **41**(3): 647–665.

Therneau, T. and Atkinson, B. (2022a). *rpart: Recursive Partitioning and Regression Trees.* R package version 4.1.16.
URL: *https://CRAN.R-project.org/package=rpart*

Therneau, T. M. and Atkinson, E. J. (2022b). An introduction to recursive partitioning using the rpart routines.

Vanschoren, J., van Rijn, J. N., Bischl, B. and Torgo, L. (2013). Openml: networked science in machine learning, *SIGKDD Explorations* **15**(2): 49–60.
URL: *http://doi.acm.org/10.1145/2641190.264119*

Wachter, S., Mittelstadt, B. and Russell, C. (2018). Counterfactual explanations without opening the black box: Automated decisions and the gdpr, *Harvard Journal of Law & Technology* **31**(2).

Wickham, H. (2014). *Advanced R*, first edn, CRC press.

Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*, Springer-Verlag New York.
URL: *https://ggplot2.tidyverse.org*

Wickham, H. (2019). *Advanced R*, second edn, CRC press.

Wickham, H., Hester, J., Chang, W. and Bryan, J. (2021). *devtools: Tools to Make Developing R Packages Easier*. R package version 2.4.3.
URL: *https://CRAN.R-project.org/package=devtools*

Wilke, C. O. (2020). *cowplot: Streamlined Plot Theme and Plot Annotations for 'ggplot2'*. R package version 1.1.1.
URL: *https://CRAN.R-project.org/package=cowplot*

Williams, R. (2012). Using the margins command to estimate and interpret adjusted predictions and marginal effects, *The Stata Journal* **12**(2): 308–331.

Wood, S. N. (2011). Fast stable restricted maximum likelihood and marginal likelihood estimation of semi-parametric generalized linear models, *Journal of the Royal Statistical Society (B)* **73**(1): 3–36.

Wright, M. N. and Ziegler, A. (2017). ranger: A fast implementation of random forests for high dimensional data in c++ and r, *Journal of Statistical Software* **77**: 1–17.

Declaration of Authorship

I hereby declare that the report submitted is my own unaided work. All direct or indirect sources used are acknowledged as references. I am aware that the thesis in digital form can be examined for the use of unauthorized aid and in order to determine whether the report as a whole or parts incorporated in it may be deemed as plagiarism. For the comparison of my work with existing sources I agree that it shall be entered in a database where it shall also remain after examination, to enable comparison with future theses submitted. Further rights of reproduction and usage, however, are not granted here. This paper was not previously presented to another examination board and has not been published.

München, August 4, 2022

Holger Löwe