

Package ‘iml’

October 3, 2018

Type Package

Title Interpretable Machine Learning

Version 0.7.1

Date 2018-09-09

Maintainer Christoph Molnar <christoph.molnar@gmail.com>

Description Interpretability methods to analyze the behavior and predictions of any machine learning model.

Implemented methods are:

Feature importance described by Fisher et al. (2018) <arXiv:1801.01489>, accumulated local effects plots described by Apley (2018) <arXiv:1612.08468>, partial dependence plots described by Friedman (2001) <http://www.jstor.org/stable/2699986>, individual conditional expectation ('ice') plots described by Goldstein et al. (2013) <doi:10.1080/10618600.2014.907095>, local models (variant of 'lime') described by Ribeiro et. al (2016) <arXiv:1602.04938>, the Shapley Value described by Strumbelj et. al (2014) <doi:10.1007/s10115-013-0679-x>, feature interactions described by Friedman et. al <doi:10.1214/07-AOAS148> and tree surrogate models.

URL <https://github.com/christophM/iml>

BugReports <https://github.com/christophM/iml/issues>

Imports R6, checkmate, ggplot2, partykit, glmnet, Metrics, data.table, foreach, yaImpute

Suggests randomForest, gower, testthat, rpart, MASS, caret, e1071, knitr, mlr, covr, rmarkdown, devtools, doParallel, ALEPlot, ranger

License MIT + file LICENSE

RoxygenNote 6.1.0

VignetteBuilder knitr

NeedsCompilation no

Author Christoph Molnar [aut, cre]

Repository CRAN

Date/Publication 2018-10-03 09:50:02 UTC

R topics documented:

iml-package	2
calculate.ale.cat	3
calculate.ale.num	3
calculate.ale.num.cat	4
calculate.ale.num.num	4
FeatureEffect	5
FeatureImp	8
impute_cells	11
Interaction	12
LocalModel	14
order_levels	17
Partial	18
plot.FeatureEffect	18
plot.FeatureImp	19
plot.Interaction	20
plot.LocalModel	21
plot.Shapley	22
plot.TreeSurrogate	23
predict.LocalModel	24
predict.TreeSurrogate	25
Predictor	26
Shapley	28
TreeSurrogate	30
Index	33

iml-package

Make machine learning models and predictions interpretable

Description

The iml package provides tools to analyse machine learning models and predictions.

Author(s)

Maintainer: Christoph Molnar <christoph.molnar@gmail.com>

See Also

[Book on Interpretable Machine Learning](#)

calculate.ale.cat *Compute ALE for 1 categorical feature*

Description

Compute ALE for 1 categorical feature

Usage

```
calculate.ale.cat(dat, run.prediction, feature.name)
```

Arguments

dat the data.frame with same columns as training data
run.prediction Predict function of type: f(newdata)
feature.name The column name of the feature for which to compute ALE

calculate.ale.num *Compute ALE for 1 numerical feature*

Description

Compute ALE for 1 numerical feature

Usage

```
calculate.ale.num(dat, run.prediction, feature.name, grid.size)
```

Arguments

dat the data.frame with same columns as training data
run.prediction Predict function of type: f(newdata)
feature.name The column name of the feature for which to compute ALE
grid.size The number of intervals

calculate.ale.num.cat *Compute ALE for 2 features, one numerical, one categorical*

Description

Compute ALE for 2 features, one numerical, one categorical

Usage

```
calculate.ale.num.cat(dat, run.prediction, feature.name, grid.size)
```

Arguments

dat	the data.frame with same columns as training data
run.prediction	Predict function of type: f(newdata)
feature.name	The column name of the features for which to compute ALE
grid.size	The number of intervals for the numerical feature

calculate.ale.num.num *Compute ALE for 2 numerical features*

Description

Compute ALE for 2 numerical features

Usage

```
calculate.ale.num.num(dat, run.prediction, feature.name, grid.size)
```

Arguments

dat	the data.frame with same columns as training data
run.prediction	Predict function of type: f(newdata)
feature.name	The column names of the feature for which to compute ALE
grid.size	The number of cells

FeatureEffect

Effect of features on the model predictions

Description

FeatureEffect computes and plots (individual) feature effects of prediction models.

Format

R6Class object.

Usage

```
effect = FeatureEffect$new(predictor, feature, method = "ale",  
  grid.size = 20, center.at = NULL, run = TRUE)
```

```
plot(effect)  
effect$results  
print(effect)  
effectd$set.feature("x2")
```

Arguments

For FeatureEffect\$new():

predictor: (Predictor)

The object (created with Predictor\$new()) holding the machine learning model and the data.

feature: ('character(1)' | 'character(2)' | 'numeric(1)' | 'numeric(2)')

The feature name or index for which to compute the partial dependencies.

method: ('character(1)')

'ale' for accumulated local effects (the default), 'pdp' for partial dependence plot, 'ice' for individual conditional expectation curves, 'pdp+ice' for partial dependence plot and ice curves within the same plot.

center.at: ('numeric(1)')

Value at which the plot should be centered. Ignored in the case of two features.

grid.size: ('numeric(1)' | 'numeric(2)')

The size of the grid for evaluating the predictions

run: ('logical(1)')

Should the Interpretation method be run?

Details

The FeatureEffect class compute the effect a feature has on the prediction. Different methods are implemented:

- Accumulated Local Effect (ALE) plots

- Partial Dependence Plots (PDPs)
- Individual Conditional Expectation (ICE) curves

Accumulated local effects and partial dependence plots both show the average model prediction over the feature. The difference is that ALE are computed as accumulated differences over the conditional distribution and partial dependence plots over the marginal distribution. ALE plots preferable to PDPs, because they are faster and unbiased when features are correlated.

ALE plots for categorical features are automatically ordered by the similarity of the categories based on the distribution of the other features for instances in a category. When the feature is an ordered factor, the ALE plot leaves the order as is.

Individual conditional expectation curves describe how, for a single observation, the prediction changes when the feature changes and can be combined with partial dependence plots.

To learn more about accumulated local effects, read the Interpretable Machine Learning book: <https://christophm.github.io/interpretable-ml-book/ale.html>

And for the partial dependence plot: <https://christophm.github.io/interpretable-ml-book/pdp.html>

And for individual conditional expectation: <https://christophm.github.io/interpretable-ml-book/ice.html>

Fields

- method:** ('character(1)')
'ale' for accumulated local effects, 'pdp' for partial dependence plot, 'ice' for individual conditional expectation curves, 'pdp+ice' for partial dependence plot and ice curves within the same plot.
- feature.name:** ('character(1)' | 'character(2)')
The names of the features for which the partial dependence was computed.
- feature.type:** ('character(1)' | 'character(2)')
The detected types of the features, either "categorical" or "numerical".
- grid.size:** ('numeric(1)' | 'numeric(2)')
The size of the grid.
- center.at:** ('numeric(1)' | 'character(1)')
The value for the centering of the plot. Numeric for numeric features, and the level name for factors.
- n.features:** ('numeric(1)')
The number of features (either 1 or 2)
- predictor:** (Predictor)
The prediction model that was analysed.
- results:** (data.frame)
data.frame with the grid of feature of interest and the predicted \hat{y} . Can be used for creating custom partial dependence plots.

Methods

- center()** method to set the value at which the ice computations are centered. See examples.
- set.feature()** method to get/set feature(s) (by index) for which to compute pdp. See examples for usage.
- plot()** method to plot the partial dependence function. See [plot.FeatureEffect](#)
- run()** [internal] method to run the interpretability method. Use `obj$run(force = TRUE)` to force a rerun.
- clone()** [internal] method to clone the R6 object.
- initialize()** [internal] method to initialize the R6 object.

References

Apley, D. W. 2016. "Visualizing the Effects of Predictor Variables in Black Box Supervised Learning Models." ArXiv Preprint.

Friedman, J.H. 2001. "Greedy Function Approximation: A Gradient Boosting Machine." *Annals of Statistics* 29: 1189-1232.

Goldstein, A., Kapelner, A., Bleich, J., and Pitkin, E. (2013). Peeking Inside the Black Box: Visualizing Statistical Learning with Plots of Individual Conditional Expectation, 1-22. <https://doi.org/10.1080/10618600.2014.907>

See Also

[plot.FeatureEffect](#)

Examples

```
# We train a random forest on the Boston dataset:
if (require("randomForest")) {
  data("Boston", package = "MASS")
  rf = randomForest(medv ~ ., data = Boston, ntree = 50)
  mod = Predictor$new(rf, data = Boston)

  # Compute the accumulated local effects for the first feature
  eff = FeatureEffect$new(mod, feature = "rm", grid.size = 30)
  eff$plot()

  # Again, but this time with a partial dependence plot and ice curves
  eff = FeatureEffect$new(mod, feature = "rm", method = "pdp+ice", grid.size = 30)
  plot(eff)

  # Since the result is a ggplot object, you can extend it:
  if (require("ggplot2")) {
    plot(eff) + ggtitle("Partial dependence")
  }

  # If you want to do your own thing, just extract the data:
  eff.dat = eff$results
  head(eff.dat)
```

```

# You can reuse the pdp object for other features:
eff$set.feature("lstat")
plot(eff)

# Only plotting the aggregated partial dependence:
eff = FeatureEffect$new(mod, feature = "crim", method = "pdp")
eff$plot()

# Only plotting the individual conditional expectation:
eff = FeatureEffect$new(mod, feature = "crim", method = "ice")
eff$plot()

# Accumulated local effects and partial dependence plots support up to two features:
eff = FeatureEffect$new(mod, feature = c("crim", "lstat"))
plot(eff)

# Partial dependence plots also works with multiclass classification
rf = randomForest(Species ~ ., data = iris, ntree=50)
mod = Predictor$new(rf, data = iris, type = "prob")

# For some models we have to specify additional arguments for the predict function
plot(FeatureEffect$new(mod, feature = "Petal.Width"))

# Partial dependence plots support up to two features:
eff = FeatureEffect$new(mod, feature = c("Sepal.Length", "Petal.Length"))
eff$plot()

# show where the actual data lies
eff$plot(show.data = TRUE)

# For multiclass classification models, you can choose to only show one class:
mod = Predictor$new(rf, data = iris, type = "prob", class = 1)
plot(FeatureEffect$new(mod, feature = "Sepal.Length"))
}

```

FeatureImp

Feature importance

Description

FeatureImp computes feature importances for prediction models. The importance is measured as the factor by which the model's prediction error increases when the feature is shuffled.

Format

[R6Class](#) object.

Usage

```
imp = FeatureImp$new(predictor, loss, method = "shuffle", n.repetitions = 3, run = TRUE)

plot(imp)
imp$results
print(imp)
```

Arguments

For FeatureImp\$new():

predictor: (Predictor)

The object (created with Predictor\$new()) holding the machine learning model and the data.

loss: ('character(1)' | function)

The loss function. Either the name of a loss (e.g. "ce" for classification or "mse") or a loss function. See Details for allowed losses.

method: ('character(1)')

Either "shuffle" or "cartesian". See Details.

n.repetitions: 'numeric(1)'

How often should the shuffling of the feature be repeated? Ignored if method is set to "cartesian". The higher the number of repetitions the more stable the results will become.

parallel: 'logical(1)'

Should the method be executed in parallel? If TRUE, requires a cluster to be registered, see ?foreach::foreach.

run: ('logical(1)')

Should the Interpretation method be run?

Details

Read the Interpretable Machine Learning book to learn in detail about feature importance: <https://christophm.github.io/interpretable-ml-book/feature-importance.html>

Two permutation schemes are implemented:

- shuffle: A simple shuffling of the feature values, yielding n perturbed instances per feature (fast)
- cartesian: Matching every instance with the feature value of all other instances, yielding n x (n-1) perturbed instances per feature (very slow)

The loss function can be either specified via a string, or by handing a function to FeatureImp(). If you want to use your own loss function it should have this signature: function(actual, predicted). Using the string is a shortcut to using loss functions from the Metrics package. Only use functions that return a single performance value, not a vector. Allowed losses are: "ce", "f1", "logLoss", "mae", "mse", "rmse", "mape", "mdae", "msle", "percent_bias", "rae", "rmse", "rmsle", "rse", "rrse", "smape" See library(help = "Metrics") to get a list of functions.

Fields

- original.error:** ('numeric(1)')
The loss of the model before perturbing features.
- predictor:** (Predictor)
The prediction model that was analysed.
- results:** (data.frame)
data.frame with the results of the feature importance computation.

Methods

- loss(actual,predicted)** The loss function. Can also be applied to data: `object$loss(actual, predicted)`
- plot()** method to plot the feature importances. See [plot.FeatureImp](#)
- `run()` [internal] method to run the interpretability method. Use `obj$run(force = TRUE)` to force a rerun.
- `clone()` [internal] method to clone the R6 object.
- `initialize()` [internal] method to initialize the R6 object.

References

Fisher, A., Rudin, C., and Dominici, F. (2018). Model Class Reliance: Variable Importance Measures for any Machine Learning Model Class, from the "Rashomon" Perspective. Retrieved from <http://arxiv.org/abs/1801.01489>

Examples

```
if (require("rpart")) {
# We train a tree on the Boston dataset:
data("Boston", package = "MASS")
tree = rpart(medv ~ ., data = Boston)
y = Boston$medv
X = Boston[-which(names(Boston) == "medv")]
mod = Predictor$new(tree, data = X, y = y)

# Compute feature importances as the performance drop in mean absolute error
imp = FeatureImp$new(mod, loss = "mae")

# Plot the results directly
plot(imp)

# Since the result is a ggplot object, you can extend it:
if (require("ggplot2")) {
plot(imp) + theme_bw()
# If you want to do your own thing, just extract the data:
imp.dat = imp$results
head(imp.dat)
ggplot(imp.dat, aes(x = feature, y = importance)) + geom_point() +
theme_bw()
}
```

```

}

# FeatureImp also works with multiclass classification.
# In this case, the importance measurement regards all classes
tree = rpart(Species ~ ., data= iris)
X = iris[-which(names(iris) == "Species")]
y = iris$Species
mod = Predictor$new(tree, data = X, y = y, type = "prob")

# For some models we have to specify additional arguments for the predict function
imp = FeatureImp$new(mod, loss = "ce")
plot(imp)

# For multiclass classification models, you can choose to only compute performance for one class.
# Make sure to adapt y
mod = Predictor$new(tree, data = X, y = y == "virginica",
  type = "prob", class = "virginica")
imp = FeatureImp$new(mod, loss = "ce")
plot(imp)
}

```

impute_cells

Impute missing cells of grid

Description

by default assumes first column of cell.dat is x1 and second is x2 leave grid1 NULL if feature x1 is a factor the difference variable has to be named .yhat.diff

Usage

```
impute_cells(cell.dat, grid1 = NULL, grid2, x1.ind = 1, x2.ind = 2)
```

Arguments

cell.dat	data.table with at least 4 columns: .yhat.diff and the two interval indices. Make sure that empty cells are also included and cell.dat is not the sparse representation.
grid1	data.frame where each row is the actual value for a given interval index for feature 1. If empty impute_cells assumes that the feature is categorical (factor).
grid2	data.frame where each row is the actual value for a given interval index for feature 2
x1.ind	column number or name of cell.dat for feature 1. If one feature is categorical, has to be x1
x2.ind	column number or name of cell.dat for feature 2

 Interaction

Feature interactions

Description

Interaction estimates the feature interactions in a prediction model.

Format

R6Class object.

Usage

```
ia = Interaction$new(predictor, feature = NULL, grid.size = 20, run = TRUE)

plot(ia)
ia$results
print(ia)
```

Arguments

For Interaction\$new():

predictor: (Predictor)

The object (created with Predictor\$new()) holding the machine learning model and the data.

feature: ('numeric(1)'|NULL)

If NULL, for each feature the interactions with all other features are estimated. If one feature name is selected, the 2-way interactions of this feature with all other features are estimated

grid.size: ('logical(1)')

The number of values per feature that should be used to estimate the interaction strength. A larger grid.size means more accurate the results but longer the computation time. For each of the grid points, the partial dependence functions have to be computed, which involves marginalizing over all data points.

parallel: 'logical(1)'

Should the method be executed in parallel? If TRUE, requires a cluster to be registered, see ?foreach::foreach.

run: ('logical(1)')

Should the Interpretation method be run?

Details

Interactions between features are measured via the decomposition of the prediction function: If a feature j has no interaction with any other feature, the prediction function can be expressed as the sum of the partial function that depends only on j and the partial function that only depends on features other than j . If the variance of the full function is completely explained by the sum of the partial functions, there is no interaction between feature j and the other features. Any variance that is not explained can be attributed to the interaction and is used as a measure of interaction strength.

The interaction strength between two features is the proportion of the variance of the 2-dimensional partial dependence function that is not explained by the sum of the two 1-dimensional partial dependence functions.

The interaction is measured by Friedman's H-statistic (square root of the H-squared test statistic) and takes on values between 0 (no interaction) to 1 (100)

To learn more about interaction effects, read the Interpretable Machine Learning book: <https://christophm.github.io/interpretable-ml-book/interaction.html>

Fields

grid.size: ('logical(1)')

The number of values per feature that should be used to estimate the interaction strength.

predictor: (Predictor)

The prediction model that was analysed.

results: (data.frame)

Data.frame with the interaction strength (column '.interaction') per feature calculated as Friedman's H-statistic and - in the case of a multi-dimensional outcome - per class.

Methods

plot() method to plot the feature interactions. See [plot.Interaction](#).

run() [internal] method to run the interpretability method. Use `obj$run(force = TRUE)` to force a rerun.

clone() [internal] method to clone the R6 object.

initialize() [internal] method to initialize the R6 object.

References

Friedman, Jerome H., and Bogdan E. Popescu. "Predictive learning via rule ensembles." *The Annals of Applied Statistics* 2.3 (2008): 916-954.

Examples

```
if (require("rpart")) {
  set.seed(42)
  # Fit a CART on the Boston housing data set
  data("Boston", package = "MASS")
  rf = rpart(medv ~ ., data = Boston)
  # Create a model object
  mod = Predictor$new(rf, data = Boston[-which(names(Boston) == "medv")])

  # Measure the interaction strength
  ia = Interaction$new(mod)

  # Plot the resulting leaf nodes
  plot(ia)

  # Extract the results
```

```

dat = ia$results
head(dat)
## Not run:
# Interaction also works with multiclass classification
rf = rpart(Species ~ ., data = iris)
mod = Predictor$new(rf, data = iris, type = "prob")

# For some models we have to specify additional arguments for the predict function
ia = Interaction$new(mod)

ia$plot()

# For multiclass classification models, you can choose to only show one class:
mod = Predictor$new(rf, data = iris, type = "prob", class = "virginica")
plot(Interaction$new(mod))

## End(Not run)
}

```

LocalModel

LocalModel

Description

LocalModel fits locally weighted linear regression models (logistic regression for classification) to explain single predictions of a prediction model.

Format

[R6Class](#) object.

Usage

```
lime = LocalModel$new(predictor, x.interest = NULL, dist.fun = "gower",
                      kernel.width = NULL, k = 3, run = TRUE)
```

```

plot(lime)
predict(lime, newdata)
lime$results
lime$explain(x.interest)
print(lime)

```

Arguments

For LocalModel\$new():

predictor: (Predictor)

The object (created with Predictor\$new()) holding the machine learning model and the data.

- x.interest:** (data.frame)
Single row with the instance to be explained.
- dist.fun:** ('character(1)')
The name of the distance function for computing proximities (weights in the linear model). Defaults to "gower". Otherwise will be forwarded to [stats::dist].
- kernel.width:** ('numeric(1)')
The width of the kernel for the proximity computation. Only used if dist.fun is not 'gower'.
- k:** ('numeric(1)')
The (maximum) number of features to be used for the surrogate model.
- run:** ('logical(1)')
Should the Interpretation method be run?

Details

A weighted glm is fitted with the machine learning model prediction as target. Data points are weighted by their proximity to the instance to be explained, using the gower proximity measure. L1-regularisation is used to make the results sparse. The resulting model can be seen as a surrogate for the machine learning model, which is only valid for that one point. Categorical features are binarized, depending on the category of the instance to be explained: 1 if the category is the same, 0 otherwise. To learn more about local models, read the Interpretable Machine Learning book: <https://christophm.github.io/interpretable-ml-book/lime.html>

The approach is similar to LIME, but has the following differences:

- Distance measure: Uses as default the gower proximity (= 1 - gower distance) instead of a kernel based on the Euclidean distance. Has the advantage to have a meaningful neighbourhood and no kernel width to tune. When the distance is not "gower", then the stats::dist() function with the chosen method will be used, and turned into a similarity measure: $\sqrt{\exp(-(\text{distance}^2) / (\text{kernel.width}^2))}$.
- Sampling: Uses the original data instead of sampling from normal distributions. Has the advantage to follow the original data distribution.
- Visualisation: Plots effects instead of betas. Both are the same for binary features, but are different for numerical features. For numerical features, plotting the betas makes no sense, because a negative beta might still increase the prediction when the feature value is also negative.

To learn more about local surrogate models, read the Interpretable Machine Learning book: <https://christophm.github.io/interpretable-ml-book/lime.html>

Fields

- best.fit.index:** ('numeric(1)')
The index of the best glmnet fit.
- k:** ('numeric(1)')
The number of features as set by the user.
- model:** (glmnet)
The fitted local model.

predictor: (Predictor)

The prediction model that was analysed.

results: (data.frame)

Results with the feature names (feature) and contributions to the prediction

x.interest: (data.frame)

The instance to be explained. See Examples for usage.

Methods

explain(x.interest) method to set a new data point which to explain.

plot() method to plot the LocalModel feature effects. See [plot.LocalModel](#)

predict() method to predict new data with the local model See also [predict.LocalModel](#)

run() [internal] method to run the interpretability method. Use `obj$run(force = TRUE)` to force a rerun.

clone() [internal] method to clone the R6 object.

initialize() [internal] method to initialize the R6 object.

References

Ribeiro, M. T., Singh, S., & Guestrin, C. (2016). "Why Should I Trust You?": Explaining the Predictions of Any Classifier. Retrieved from <http://arxiv.org/abs/1602.04938>

Gower, J. C. (1971), "A general coefficient of similarity and some of its properties". *Biometrics*, 27, 623–637.

See Also

[plot.LocalModel](#) and [predict.LocalModel](#)

[Shapley](#) can also be used to explain single predictions

The package ‘lime’ with the original implementation

Examples

```
if (require("randomForest")) {
# First we fit a machine learning model on the Boston housing data
data("Boston", package = "MASS")
X = Boston[-which(names(Boston) == "medv")]
rf = randomForest(medv ~ ., data = Boston, ntree = 50)
mod = Predictor$new(rf, data = X)

# Explain the first instance of the dataset with the LocalModel method:
x.interest = X[1,]
lemon = LocalModel$new(mod, x.interest = x.interest, k = 2)
lemon

# Look at the results in a table
lemon$results
# Or as a plot
plot(lemon)
```

```

# Reuse the object with a new instance to explain
lemon$x.interest
lemon$explain(X[2,])
lemon$x.interest
plot(lemon)

# LocalModel also works with multiclass classification
rf = randomForest(Species ~ ., data= iris, ntree=50)
X = iris[-which(names(iris) == 'Species')]
mod = Predictor$new(rf, data = X, type = "prob", class = "setosa")

# Then we explain the first instance of the dataset with the LocalModel method:
lemon = LocalModel$new(mod, x.interest = X[1,], k = 2)
lemon$results
plot(lemon)
}

```

order_levels

Order levels of a categorical features

Description

Goal: Compute the distances between two categories. Input: Instances from category 1 and 2

Usage

```
order_levels(dat, feature.name)
```

Arguments

dat	data.frame with the training data
feature.name	the name of the categorical feature

Details

- For all features, do (excluding the categorical feature for which we are computing the order):
 - If the feature is numerical: Take instances from category 1, calculate the empirical cumulative probability distribution function (ecdf) of the feature. The ecdf is a function that tells us for a given feature value, how many values are smaller. Do the same for category 2. The distance is the absolute maximum point-wise distance of the two ecdf. Practically, this value is high when the distribution from one category is strongly shifted far away from the other. This measure is also known as the [Kolmogorov-Smirnov distance](https://en.wikipedia.org/wiki/Kolmogorov-Smirnov_test)
 - If the feature is categorical: Take instances from category 1 and calculate a table with the relative frequency of each category of the other feature. Do the same for instances from category 2. The distance is the sum of the absolute difference of both relative frequency tables.
- Sum up the distances over all features

This algorithm we run for all pairs of categories. Then we have a k times k matrix, when k is the number of categories, where each entry is the distance between two categories. Still not enough to

have a single order, because, a (dis)similarity tells you the pair-wise distances, but does not give you a one-dimensional ordering of the classes. To kind of force this thing into a single dimension, we have to use a dimension reduction trick called multi-dimensional scaling. This can be solved using multi-dimensional scaling, which takes in a distance matrix and returns a distance matrix with reduced dimension. In our case, we only want 1 dimension left, so that we have a single ordering of the categories and can compute the accumulated local effects. After reducing it to a single ordering, we are done and can use this ordering to compute ALE. This is not the Holy Grail how to order the factors, but one possibility.

Orders the levels by their similarity in other features. Computes per feature the distance, sums up all distances and does multi-dimensional scaling

Value

the order of the levels (not levels itself)

Partial	<i>Effect of features on the model predictions (deprecated)</i>
---------	---

Description

Deprecated, please use 'FeatureEffect', see ?FeatureEffect

Format

[R6Class](#) object.

See Also

[FeatureEffect](#)

plot.FeatureEffect	<i>Plot FeatureEffect</i>
--------------------	---------------------------

Description

plot.FeatureEffect() plots the results of a FeatureEffect object.

Usage

```
## S3 method for class 'FeatureEffect'
plot(x, rug = TRUE, show.data = FALSE)
```

Arguments

x	A FeatureEffect R6 object
rug	[logical] Should a rug be plotted to indicate the feature distribution? The rug will be jittered a bit, so the location may not be exact, but it avoids overplotting.
show.data	Should the data points be shown? Only affects 2D plots, and ignored for 1D plots, because rug has the same information.

Value

ggplot2 plot object

See Also

[FeatureEffect](#)

Examples

```
# We train a random forest on the Boston dataset:
if (require("randomForest")) {
  data("Boston", package = "MASS")
  rf = randomForest(medv ~ ., data = Boston, ntree = 50)
  mod = Predictor$new(rf, data = Boston)

  # Compute the partial dependence for the first feature
  eff = FeatureEffect$new(mod, feature = "crim")

  # Plot the results directly
  plot(eff)
}
```

plot.FeatureImp *Plot Feature Importance*

Description

plot.FeatureImp() plots the feature importance results of a FeatureImp object.

Usage

```
## S3 method for class 'FeatureImp'
plot(x, sort = TRUE, ...)
```

Arguments

x	A FeatureImp R6 object
sort	logical. Should the features be sorted in descending order? Defaults to TRUE.
...	Further arguments for the objects plot function

Value

ggplot2 plot object

See Also

[FeatureImp](#)

Examples

```
if (require("rpart")) {
  # We train a tree on the Boston dataset:
  data("Boston", package = "MASS")
  tree = rpart(medv ~ ., data = Boston)
  y = Boston$medv
  X = Boston[-which(names(Boston) == "medv")]
  mod = Predictor$new(tree, data = X, y = y)

  # Compute feature importances as the performance drop in mean absolute error
  imp = FeatureImp$new(mod, loss = "mae")

  # Plot the results directly
  plot(imp)
}
```

plot.Interaction *Plot Interaction*

Description

plot.Interaction() plots the results of an Interaction object.

Usage

```
## S3 method for class 'Interaction'
plot(x, sort = TRUE)
```

Arguments

x An Interaction R6 object
 sort logical. Should the features be sorted in descending order? Defaults to TRUE.

Value

ggplot2 plot object

See Also

[Interaction](#)

Examples

```
# We train a tree on the Boston dataset:
if (require("rpart")) {
  data("Boston", package = "MASS")
  rf = rpart(medv ~ ., data = Boston)
  mod = Predictor$new(rf, data = Boston)

  # Compute the interactions
  ia = Interaction$new(mod)

  # Plot the results directly
  plot(ia)
}
```

plot.LocalModel	<i>Plot Local Model</i>
-----------------	-------------------------

Description

plot.LocalModel() plots the feature effects of a LocalModel object.

Usage

```
## S3 method for class 'LocalModel'
plot(object)
```

Arguments

object A LocalModel R6 object

Value

ggplot2 plot object

See Also

[LocalModel](#)

Examples

```
if (require("randomForest")) {
  # First we fit a machine learning model on the Boston housing data
  data("Boston", package = "MASS")
  X = Boston[-which(names(Boston) == "medv")]
  rf = randomForest(medv ~ ., data = Boston, ntree = 50)
  mod = Predictor$new(rf, data = X)

  # Explain the first instance of the dataset with the LocalModel method:
  x.interest = X[1,]
```

```
lemon = LocalModel$new(mod, x.interest = x.interest, k = 2)
plot(lemon)
}
```

plot.Shapley

Plot Shapley

Description

plot.Shapley() plots the Shapley values - the contributions of feature values to the prediction.

Usage

```
## S3 method for class 'Shapley'
plot(object, sort = TRUE)
```

Arguments

object	A Shapley R6 object
sort	logical. Should the feature values be sorted by Shapley value? Ignored for multi.class output.

Value

ggplot2 plot object

See Also

[Shapley](#)

Examples

```
## Not run:
if (require("rpart")) {
# First we fit a machine learning model on the Boston housing data
data("Boston", package = "MASS")
rf = rpart(medv ~ ., data = Boston)
X = Boston[-which(names(Boston) == "medv")]
mod = Predictor$new(rf, data = X)

# Then we explain the first instance of the dataset with the Shapley method:
x.interest = X[1,]
shapley = Shapley$new(mod, x.interest = x.interest)
plot(shapley)
}

## End(Not run)
```

plot.TreeSurrogate *Plot Tree Surrogate*

Description

Plot the response for newdata of a TreeSurrogate object. Each plot facet is one leaf node and visualises the distribution of the \hat{y} from the machine learning model.

Usage

```
## S3 method for class 'TreeSurrogate'  
plot(object)
```

Arguments

object A TreeSurrogate R6 object

Value

ggplot2 plot object

See Also

[TreeSurrogate](#)

Examples

```
if (require("randomForest")) {  
  # Fit a Random Forest on the Boston housing data set  
  data("Boston", package = "MASS")  
  rf = randomForest(medv ~ ., data = Boston, ntree = 50)  
  # Create a model object  
  mod = Predictor$new(rf, data = Boston[-which(names(Boston) == "medv")])  
  
  # Fit a decision tree as a surrogate for the whole random forest  
  dt = TreeSurrogate$new(mod)  
  
  # Plot the resulting leaf nodes  
  plot(dt)  
}
```

predict.LocalModel *Predict LocalModel*

Description

Predict the response for newdata with the LocalModel model.

Usage

```
## S3 method for class 'LocalModel'  
predict(object, newdata = NULL, ...)
```

Arguments

object	A LocalModel R6 object
newdata	A data.frame for which to predict
...	Further arguments for the objects predict function

Value

A data.frame with the predicted outcome.

See Also

[LocalModel](#)

Examples

```
if (require("randomForest")) {  
  # First we fit a machine learning model on the Boston housing data  
  data("Boston", package = "MASS")  
  X = Boston[-which(names(Boston) == "medv")]  
  rf = randomForest(medv ~ ., data = Boston, ntree = 50)  
  mod = Predictor$new(rf, data = X)  
  
  # Explain the first instance of the dataset with the LocalModel method:  
  x.interest = X[1,]  
  lemon = LocalModel$new(mod, x.interest = x.interest, k = 2)  
  predict(lemon, newdata = x.interest)  
}
```

predict.TreeSurrogate *Predict Tree Surrogate*

Description

Predict the response for newdata of a TreeSurrogate object.

Usage

```
## S3 method for class 'TreeSurrogate'
predict(object, newdata, type = "prob", ...)
```

Arguments

object	The surrogate tree. A TreeSurrogate R6 object
newdata	A data.frame for which to predict
type	Either "prob" or "class". Ignored if the surrogate tree does regression.
...	Further arguments for predict_party

Details

This function makes the TreeSurrogate object call its internal object\$predict() method.

Value

A data.frame with the predicted outcome. In case of regression it is the predicted \hat{y} . In case of classification it is either the class probabilities (for type "prob") or the class label (type "class")

See Also

[TreeSurrogate](#)

Examples

```
if (require("randomForest")) {
  # Fit a Random Forest on the Boston housing data set
  data("Boston", package = "MASS")
  rf = randomForest(medv ~ ., data = Boston, ntree = 50)
  # Create a model object
  mod = Predictor$new(rf, data = Boston[-which(names(Boston) == "medv")])

  # Fit a decision tree as a surrogate for the whole random forest
  dt = TreeSurrogate$new(mod)

  # Plot the resulting leaf nodes
  predict(dt, newdata = Boston)
}
```

Predictor

Predictor object

Description

A Predictor object holds any machine learning model (mlr, caret, randomForest, ...) and the data to be used of analysing the model. The interpretation methods in the iml package need the machine learning model to be wrapped in a Predictor object.

Format

[R6Class](#) object.

Usage

```
model = Predictor$new(model = NULL, data, y = NULL, class=NULL,  
  predict.fun = NULL, type = NULL)
```

```
model$predict(newdata)
```

Arguments

model: (any)

The machine learning model. Recommended are models from mlr and caret. Other machine learning with a S3 predict functions work as well, but less robust (e.g. randomForest).

data: (data.frame)

The data to be used for analysing the prediction model.

y: (('character(1)' | numeric | factor)

The target vector or (preferably) the name of the target column in the data argument.

class: ('character(1)')

The class column to be returned in case of multiclass output.

predict.fun: (function)

The function to predict newdata. Only needed if model is not a model from mlr or caret package.

type: ('character(1)')

This argument is passed to the prediction function of the model. The classic use case is to say type="prob" for classification models. For example for caret models or the most S3 predict methods. If both predict.fun and type are used, then type is passed as an argument to predict.fun.

batch.size: ('numeric(1)')

The maximum number of rows to be input the model for prediction at once. Currently only respected for FeatureImp, Partial and Interaction.

Details

A Predictor object is a container for the prediction model and the data. This ensures that the machine learning model can be analysed robustly.

Note: In case of classification, the model should return one column per class with the class probability.

Fields

- class:** ('character(1)')
The class column to be returned.
- data:** (data.frame)
data object with the data for the model interpretation.
- prediction.colnames:** (character)
The column names of the predictions.
- task:** ('character(1)')
The inferred prediction task: "classification" or "regression".

Methods

- predict(newdata)** method to predict new data with the machine learning model.
- clone()** [internal] method to clone the R6 object.
- initialize()** [internal] method to initialize the R6 object.

Examples

```
if (require("mlr")) {
  task = makeClassifTask(data = iris, target = "Species")
  learner = makeLearner("classif.rpart", minsplit = 7, predict.type = "prob")
  mod.mlr = train(learner, task)
  mod = Predictor$new(mod.mlr, data = iris)
  mod$predict(iris[1:5,])

  mod = Predictor$new(mod.mlr, data = iris, class = "setosa")
  mod$predict(iris[1:5,])
}

if (require("randomForest")) {
  rf = randomForest(Species ~ ., data = iris, ntree = 20)

  mod = Predictor$new(rf, data = iris, type = "prob")
  mod$predict(iris[50:55,])

  # Feature importance needs the target vector, which needs to be supplied:
  mod = Predictor$new(rf, data = iris, y = "Species", type = "prob")
}
```

Description

Shapley computes feature contributions for single predictions with the Shapley value, an approach from cooperative game theory. The features values of an instance cooperate to achieve the prediction. The Shapley value fairly distributes the difference of the instance's prediction and the datasets average prediction among the features.

Format

`R6Class` object.

Usage

```
shapley = Shapley$new(predictor, x.interest = NULL, sample.size = 100, run = TRUE)

plot(shapley)
shapley$results
print(shapley)
shapley$explain(x.interest)
```

Arguments

For `Shapley$new()`:

predictor: (Predictor)

The object (created with `Predictor$new()`) holding the machine learning model and the data.

x.interest: (data.frame)

Single row with the instance to be explained.

sample.size: ('numeric(1)')

The number of Monte Carlo samples for estimating the Shapley value.

run: ('logical(1)')

Should the Interpretation method be run?

Details

For more details on the algorithm see <https://christophm.github.io/interpretable-ml-book/shapley.html>

Fields

predictor: (Predictor)

The object (created with `Predictor$new()`) holding the machine learning model and the data.

results: (data.frame)

data.frame with the Shapley values (ϕ) per feature.

sample.size: ('numeric(1)')

The number of times coalitions/marginals are sampled from data X. The higher the more accurate the explanations become.

x.interest: (data.frame)

Single row with the instance to be explained.

y.hat.interest: (numeric)

Predicted value for instance of interest

y.hat.average: ('numeric(1)')

Average predicted value for data X

Methods

explain(x.interest) method to set a new data point which to explain.

plot() method to plot the Shapley value. See [plot.Shapley](#)

run() [internal] method to run the interpretability method. Use `obj$run(force = TRUE)` to force a rerun.

clone() [internal] method to clone the R6 object.

initialize() [internal] method to initialize the R6 object.

References

Strumbelj, E., Kononenko, I. (2014). Explaining prediction models and individual predictions with feature contributions. *Knowledge and Information Systems*, 41(3), 647-665. <https://doi.org/10.1007/s10115-013-0679-x>

See Also

[Shapley](#)

A different way to explain predictions: [LocalModel](#)

Examples

```
if (require("rpart")) {
  # First we fit a machine learning model on the Boston housing data
  data("Boston", package = "MASS")
  rf = rpart(medv ~ ., data = Boston)
  X = Boston[-which(names(Boston) == "medv")]
  mod = Predictor$new(rf, data = X)

  # Then we explain the first instance of the dataset with the Shapley method:
  x.interest = X[1,]
  shapley = Shapley$new(mod, x.interest = x.interest)
  shapley

  # Look at the results in a table
  shapley$results
  # Or as a plot
  plot(shapley)
```

```

# Explain another instance
shapley$explain(X[2,])
plot(shapley)
## Not run:
# Shapley() also works with multiclass classification
rf = rpart(Species ~ ., data = iris)
X = iris[-which(names(iris) == "Species")]
mod = Predictor$new(rf, data = X, type = "prob")

# Then we explain the first instance of the dataset with the Shapley() method:
shapley = Shapley$new(mod, x.interest = X[1,])
shapley$results
plot(shapley)

# You can also focus on one class
mod = Predictor$new(rf, data = X, type = "prob", class = "setosa")
shapley = Shapley$new(mod, x.interest = X[1,])
shapley$results
plot(shapley)

## End(Not run)
}

```

TreeSurrogate

Decision tree surrogate model

Description

TreeSurrogate fits a decision tree on the predictions of a prediction model.

Format

[R6Class](#) object.

Usage

```

tree = TreeSurrogate$new(predictor, maxdepth = 2, tree.args = NULL, run = TRUE)

plot(tree)
predict(tree, newdata)
tree$results
print(tree)

```

Arguments

For TreeSurrogate\$new():

predictor: (Predictor)

The object (created with Predictor\$new()) holding the machine learning model and the data.

- maxdepth:** ('numeric(1)')
The maximum depth of the tree. Default is 2.
- run:** ('logical(1)')
Should the Interpretation method be run?
- tree.args:** (named list)
Further arguments for ctree.

Details

A conditional inference tree is fitted on the predicted \hat{y} from the machine learning model and the data. The partykit package and function are used to fit the tree. By default a tree of maximum depth of 2 is fitted to improve interpretability.

To learn more about global surrogate models, read the Interpretable Machine Learning book: <https://christophm.github.io/interpretable-ml-book/global.html>

Fields

- maxdepth:** ('numeric(1)')
The maximum tree depth.
- predictor:** (Predictor)
The prediction model that was analysed.
- r.squared:** ('numeric(1|n.classes)')
R squared measures how well the decision tree approximates the underlying model. It is calculated as $1 - (\text{variance of prediction differences} / \text{variance of black box model predictions})$. For the multi-class case, r.squared contains one measure per class.
- results:** (data.frame)
Data.frame with sampled feature X together with the leaf node information (columns .node and .path) and the predicted \hat{y} for tree and machine learning model (columns starting with .y.hat).
- tree:** (party)
The fitted tree. See also [ctree](#).

Methods

- plot()** method to plot the leaf nodes of the surrogate decision tree. See [plot.TreeSurrogate](#).
- predict()** method to predict new data with the tree. See also [predict.TreeSurrogate](#)
- run()** [internal] method to run the interpretability method. Use `obj$run(force = TRUE)` to force a rerun.
- clone()** [internal] method to clone the R6 object.
- initialize()** [internal] method to initialize the R6 object.

References

Craven, M., & Shavlik, J. W. (1996). Extracting tree-structured representations of trained networks. In Advances in neural information processing systems (pp. 24-30).

See Also

[predict.TreeSurrogate](#) [plot.TreeSurrogate](#)

For the tree implementation [ctree](#)

Examples

```

if (require("randomForest")) {
# Fit a Random Forest on the Boston housing data set
data("Boston", package = "MASS")
rf = randomForest(medv ~ ., data = Boston, ntree = 50)
# Create a model object
mod = Predictor$new(rf, data = Boston[-which(names(Boston) == "medv")])

# Fit a decision tree as a surrogate for the whole random forest
dt = TreeSurrogate$new(mod)

# Plot the resulting leaf nodes
plot(dt)

# Use the tree to predict new data
predict(dt, Boston[1:10,])

# Extract the results
dat = dt$results
head(dat)

# It also works for classification
rf = randomForest(Species ~ ., data = iris, ntree = 50)
X = iris[-which(names(iris) == "Species")]
mod = Predictor$new(rf, data = X, type = "prob")

# Fit a decision tree as a surrogate for the whole random forest
dt = TreeSurrogate$new(mod, maxdepth=2)

# Plot the resulting leaf nodes
plot(dt)

# If you want to visualise the tree directly:
plot(dt$tree)

# Use the tree to predict new data
set.seed(42)
iris.sample = X[sample(1:nrow(X), 10),]
predict(dt, iris.sample)
predict(dt, iris.sample, type = "class")

# Extract the dataset
dat = dt$results
head(dat)
}

```

Index

calculate.ale.cat, [3](#)
calculate.ale.num, [3](#)
calculate.ale.num.cat, [4](#)
calculate.ale.num.num, [4](#)
ctree, [31](#), [32](#)

FeatureEffect, [5](#), [18](#), [19](#)
FeatureImp, [8](#), [20](#)

iml (iml-package), [2](#)
iml-package, [2](#)
impute_cells, [11](#)
Interaction, [12](#), [20](#)

LocalModel, [14](#), [21](#), [24](#), [29](#)

order_levels, [17](#)

Partial, [18](#)
plot.FeatureEffect, [7](#), [18](#)
plot.FeatureImp, [10](#), [19](#)
plot.Interaction, [13](#), [20](#)
plot.LocalModel, [16](#), [21](#)
plot.Shapley, [22](#), [29](#)
plot.TreeSurrogate, [23](#), [31](#), [32](#)
predict.LocalModel, [16](#), [24](#)
predict.TreeSurrogate, [25](#), [31](#), [32](#)
Predictor, [26](#)

R6Class, [5](#), [8](#), [12](#), [14](#), [18](#), [26](#), [28](#), [30](#)

Shapley, [16](#), [22](#), [28](#), [29](#)

TreeSurrogate, [23](#), [25](#), [30](#)