Package 'iml'

September 24, 2020

Type Package

Title Interpretable Machine Learning

Version 0.10.1

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) <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.

License MIT + file LICENSE

URL https://christophm.github.io/iml/, https://github.com/christophM/iml/

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

Imports checkmate, data.table, Formula, future, future.apply, ggplot2, keras (>= 2.2.5.0), Metrics, prediction, R6

Suggests ALEPlot, bench, caret, covr, e1071, future.callr, glmnet, gower, h2o, knitr, MASS, mlr, mlr3, party, partykit, patchwork, randomForest, ranger, rmarkdown, rpart, testthat, yaImpute

VignetteBuilder knitr

Encoding UTF-8

RoxygenNote 7.1.1

NeedsCompilation no

Author Christoph Molnar [aut, cre], Patrick Schratz [aut] (<https://orcid.org/0000-0003-0748-6624>)

Repository CRAN

Date/Publication 2020-09-24 12:30:14 UTC

R topics documented:

`iml-package` *Make machine learning models and predictions interpretable*

Description

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

Author(s)

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

Authors:

- Patrick Schratz <patrick.schratz@gmail.com> (ORCID)

See Also

Book on Interpretable Machine Learning

`extract.glmnet.effects` *Extract glmnet effects*

Description

Extract glmnet effects

Usage

```
extract.glmnet.effects(betas, best.index, x.recoded, x.original)
```

Arguments

`betas`	glmnet$beta
`best.index`	index k
`x.recoded`	the recoded version of x
`x.original`	the original x Assuming that the first row is the x.interest

---

`FeatureEffect` *Effect of a feature on predictions*

Description

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

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.

For the partial dependence plots: https://christophm.github.io/interpretable-ml-book/pdp.html

For individual conditional expectation: https://christophm.github.io/interpretable-ml-book/ice.html

Super class

`iml::InterpretationMethod` -> `FeatureEffect`

Public fields

`grid.size` (`numeric(1)` | `numeric(2)`)
: The size of the grid.

`feature.name` (`character(1)` | `character(2)`)
: The names of the features for which the partial dependence was computed.

`n.features` (`numeric(1)`)
: The number of features (either 1 or 2).

`feature.type` (`character(1)` | `character(2)`)
: The detected types of the features, either "categorical" or "numerical".

`method` (`character(1)`)

Active bindings

`center.at` numeric
: Value at which the plot was centered. Ignored in the case of two features.

Methods

Public methods:

- `FeatureEffect$new()`
- `FeatureEffect$set.feature()`
- `FeatureEffect$center()`
- `FeatureEffect$predict()`
- `FeatureEffect$clone()`

Method `new()`**:** Create a FeatureEffect object

Usage:

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

Arguments:

`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 effects.

`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.

`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.

Method `set.feature()`**:** Get/set feature(s) (by index) for which to compute PDP.

Usage:

```
FeatureEffect$set.feature(feature)
```

Arguments:

`feature (character(1))`
: Feature name.

Method `center()`**:** Set the value at which the ice computations are centered.

Usage:

```
FeatureEffect$center(center.at)
```

Arguments:

`center.at (numeric(1))`
: Value at which the plot should be centered. Ignored in the case of two features.

Method `predict()`**:** Predict the marginal outcome given a feature.

Usage:

```
FeatureEffect$predict(data, extrapolate = FALSE)
```

Arguments:

`data` data.frame
: Data.frame with the feature or a vector.

`extrapolate (character(1))`
: If TRUE, predict returns NA for x values outside of observed range. If FALSE, predcit returns the closest PDP value for x values outside the range. Ignored for categorical features

Returns: Values of the effect curves at the given values.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FeatureEffect$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

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:
data("Boston", package = "MASS")
library("randomForest")
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:
library("ggplot2")
plot(eff) +
  # Adds a title
  ggtitle("Partial dependence") +
  # Adds original predictions
```

```
  geom_point(
    data = Boston, aes(y = mod$predict(Boston)[[1]], x = rm),
    color = "pink", size = 0.5
  )

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

# You can also use the object to "predict" the marginal values.
eff$predict(Boston[1:3, ])
# Instead of the entire data.frame, you can also use feature values:
eff$predict(c(5, 6, 7))

# 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)

# FeatureEffect 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"))

# FeatureEffect 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"))
```

`FeatureEffects` *Effect of a feature on predictions*

Description

`FeatureEffects` computes and plots feature effects of multiple features at once.

Details

FeatureEffects computes the effects for all given features on the model prediction. FeatureEffects is a convenience class that calls FeatureEffect multiple times. See `?FeatureEffect` for details what's actually computed.

Only first-order effects can be computed with the FeatureEffects interface. If you are interested in the visualization of interactions between two features, directly use FeatureEffect.

Parallelization

Parallelization is supported via package **future**. To initialize future-based parallelization, select an appropriate backend and specify the amount of workers. For example, to use a PSOCK based cluster backend do:

```
future::plan(multisession, workers = 2)
<iml function here>
```

Consult the resources of the **future** package for more parallel backend options.

Super class

`iml::InterpretationMethod` -> `FeatureEffects`

Public fields

`grid.size` (`numeric(1)` | `numeric(2)`)
The size of the grid.

`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.

`effects` (list)
Named list of FeatureEffects.

`features` (`character()`)
The names of the features for which the effects were computed.

`center.at` numeric
Value at which the plot was centered. Ignored in the case of two features.

Methods

Public methods:

- `FeatureEffects$new()`
- `FeatureEffects$clone()`

Method `new()`**:** Create a FeatureEffects object

Usage:

```
FeatureEffects$new(
  predictor,
  features = NULL,
  method = "ale",
  center.at = NULL,
  grid.size = 20
)
```

Arguments:

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

`features (character())`
: The names of the features for which to compute the feature effects.

`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.

`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.

`feature (character(1) | character(2) | numeric(1) | numeric(2))`
: The feature name or index for which to compute the effects.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FeatureEffects$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

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.FeatureEffects

Examples

```
# We train a random forest on the Boston dataset:
library("rpart")
data("Boston", package = "MASS")
rf <- rpart(medv ~ ., data = Boston)
mod <- Predictor$new(rf, data = Boston)

# Compute the accumulated local effects for all features
eff <- FeatureEffects$new(mod)
eff$plot()
## Not run:
# Again, but this time with a partial dependence plot
eff <- FeatureEffects$new(mod, method = "pdp")
eff$plot()

# Only a subset of features
eff <- FeatureEffects$new(mod, features = c("nox", "crim"))
eff$plot()

# You can access each FeatureEffect individually

eff.nox <- eff$effects[["nox"]]
eff.nox$plot()


# FeatureEffects also works with multiclass classification
rf <- rpart(Species ~ ., data = iris)
mod <- Predictor$new(rf, data = iris, type = "prob")

FeatureEffects$new(mod)$plot(ncol = 2)

## End(Not run)
```

FeatureImp *Feature importance*

Description

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

Details

To compute the feature importance for a single feature, the model prediction loss (error) is measured before and after shuffling the values of the feature. By shuffling the feature values, the association between the outcome and the feature is destroyed. The larger the increase in prediction error, the more important the feature was. The shuffling is repeated to get more accurate results, since the permutation feature importance tends to be quite unstable. Read the Interpretable Machine Learning book to learn about feature importance in detail: https://christophm.github.io/interpretable-ml-book/feature-importance.html

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"` and `"smape"`.

See `library(help = "Metrics")` to get a list of functions.

Parallelization

Parallelization is supported via package **future**. To initialize future-based parallelization, select an appropriate backend and specify the amount of workers. For example, to use a PSOCK based cluster backend do:

```
future::plan(multisession, workers = 2)
<iml function here>
```

Consult the resources of the **future** package for more parallel backend options.

Super class

`iml::InterpretationMethod` -> `FeatureImp`

Public fields

`loss` (`character(1)` | function)
: The loss function. Either the name of a loss (e.g. `"ce"` for classification or `"mse"`) or a function.

`original.error` (`numeric(1)`)
: The loss of the model before perturbing features.

`n.repetitions` integer
: Number of repetitions.

`compare` (`character(1)`)
: Either `"ratio"` or `"difference"`, depending on whether the importance was calculated as difference between original model error and model error after permutation or as ratio.

Methods

Public methods:

- `FeatureImp$new()`
- `FeatureImp$clone()`

Method `new()`**:** Create a FeatureImp object

Usage:

```
FeatureImp$new(predictor, loss, compare = "ratio", n.repetitions = 5)
```

Arguments:

`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 function. See Details for allowed losses.

`compare` (`character(1)`)
: Either `"ratio"` or `"difference"`. Should importance be measured as the difference or as the ratio of original model error and model error after permutation?
  - Ratio: error.permutation/error.orig
  - Difference: error.permutation - error.orig

`n.repetitions` (`numeric(1)`)
: How often should the shuffling of the feature be repeated? The higher the number of repetitions the more stable and accurate the results become.

Returns: (data.frame)
data.frame with the results of the feature importance computation. One row per feature with the following columns:

- importance.05 (5% quantile of importance values from the repetitions)
- importance (median importance)
- importance.95 (95% quantile) and the permutation.error (median error over all repetitions).

The distribution of the importance is also visualized as a bar in the plots, the median importance over the repetitions as a point.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
FeatureImp$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

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

```
library("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:
library("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()

# We can also look at the difference in model error instead of the ratio
imp <- FeatureImp$new(mod, loss = "mae", compare = "difference")

# Plot the results directly
plot(imp)


# 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)
```

`has.predict` *returns TRUE if object has predict function*

Description

returns TRUE if object has predict function

Usage

```
has.predict(object)
```

Arguments

`object`	The object to check.

`Interaction` *Feature interactions*

Description

Feature interactions

Feature interactions

Details

`Interaction` estimates the feature interactions in a prediction model.

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% of standard deviation of f(x) du to interaction).

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

Parallelization

Parallelization is supported via package **future**. To initialize future-based parallelization, select an appropriate backend and specify the amount of workers. For example, to use a PSOCK based cluster backend do:

```
future::plan(multisession, workers = 2)
<iml function here>
```

Consult the resources of the **future** package for more parallel backend options.

Super class

`iml::InterpretationMethod` -> `Interaction`

Public fields

`grid.size` (`logical(1)`)
The number of values per feature that should be used to estimate the interaction strength.

Methods

Public methods:

- `Interaction$new()`
- `Interaction$clone()`

Method `new()`**:** Create an Interaction object

Usage:

```
Interaction$new(predictor, feature = NULL, grid.size = 30)
```

Arguments:

`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 effects.

`grid.size` (`numeric(1)` | `numeric(2)`)
The size of the grid for evaluating the predictions.

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

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Interaction$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

References

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

Examples

```
## Not run:
library("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)

# 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)
```

InterpretationMethod *Interpretation Method*

Description

Superclass container for Interpretation Method objects

Public fields

`results` data.frame
The aggregated results of the experiment

`predictor` Predictor object.

Methods

Public methods:

- `InterpretationMethod$new()`
- `InterpretationMethod$plot()`
- `InterpretationMethod$print()`
- `InterpretationMethod$clone()`

Method `new()`**:** Create an InterpretationMethod object

Usage:

```
InterpretationMethod$new(predictor)
```

Arguments:

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

Method `plot()`**:** Plot function. Calls `private$generatePlot()` of the respective subclass.

Usage:

```
InterpretationMethod$plot(...)
```

Arguments:

`...` Passed to `private$generatePlot()`.

Method `print()`**:** Printer for InterpretationMethod objects

Usage:

```
InterpretationMethod$print()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
InterpretationMethod$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

LocalModel *LocalModel*

Description

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

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-regularization 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 neighborhood 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: $sqrt(exp(-(distance^2)/(kernel.width$
- **Sampling**: Uses the original data instead of sampling from normal distributions. Has the advantage to follow the original data distribution.
- **Visualization**: Plots effects instead of betas. Both are the same for binary features, but ared 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

Super class

iml::InterpretationMethod -> LocalModel

Public fields

x.interest data.frame
: Single row with the instance to be explained.

k numeric(1)
: The number of features as set by the user.

model glmnet
: The fitted local model.

`best.fit.index numeric(1)`
The index of the best glmnet fit.

Methods

Public methods:

- `LocalModel$new()`
- `LocalModel$predict()`
- `LocalModel$explain()`
- `LocalModel$clone()`

Method `new()`**:** Create a Local Model object.

Usage:

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

Arguments:

`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 number of features.

Returns: data.frame
Results with the feature names (`feature`) and contributions to the prediction.

Method `predict()`**:** Method to predict new data with the local model See also predict.LocalModel.

Usage:

```
LocalModel$predict(newdata = NULL, ...)
```

Arguments:

`newdata` data.frame
Data to predict on.

`...` Not used

Method `explain()`**:** Set a new data point to explain.

Usage:

```
LocalModel$explain(x.interest)
```

Arguments:

`x.interest` data.frame
Single row with the instance to be explained.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
LocalModel$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

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

```
library("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

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

Usage

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

Arguments

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

Details

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

1. 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%E2%80%93Smirnov_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.

1. 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.

Value

the order of the levels (not levels itself)

Partial *Effect of one or two feature(s) on the model predictions (deprecated)*

Description

Effect of one or two feature(s) on the model predictions (deprecated)

Effect of one or two feature(s) on the model predictions (deprecated)

Details

Deprecated, please use FeatureEffect.

Super classes

`iml::InterpretationMethod` -> `iml::FeatureEffect` -> `Partial`

Methods

Public methods:

- `Partial$new()`
- `Partial$clone()`

Method `new()`**:** Effect of one or two feature(s) on the model predictions

Usage:

```
Partial$new(
  predictor,
  feature,
  aggregation = "pdp",
  ice = TRUE,
  center.at = NULL,
  grid.size = 20
)
```

Arguments:

`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 effects.

`aggregation (character(1))`
: The aggregation approach to use. Possible values are `"pdp"`, `"ale"` or `"none"`.

`ice` logical
: Whether to compute ice plots.

`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.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Partial$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

FeatureEffect

`plot.FeatureEffect` *Plot FeatureEffect*

Description

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

Usage

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

Arguments

`x`	A FeatureEffect 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`	(`logical(1)`) Should the data points be shown? Only affects 2D plots, and ignored for 1D plots, because rug has the same information.

ylim	(numeric(2)) Vector with two coordinates for the y-axis. Only works when one feature is used in FeatureEffect, ignored when two are used.

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 ALE for the first feature
  eff <- FeatureEffect$new(mod, feature = "crim")

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

---

plot.FeatureEffects *Plot FeatureEffect*

Description

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

Usage

```
## S3 method for class 'FeatureEffects'
plot(x, features = NULL, nrows = NULL, ncols = NULL, fixed_y = TRUE, ...)
```

Arguments

x	A FeatureEffect object.
features	character For which features should the effects be plotted? Default is all features. You can also sort the order of the plots with this argument.
nrows	The number of rows in the table of graphics
ncols	The number of columns in the table of graphics
fixed_y	Should the y-axis range be the same for all effects? Defaults to TRUE.
...	Further arguments for FeatureEffect$plot()

Details

In contrast to other plot methods in iml, for FeatureEffects the returned plot is not a ggplot2 object, but a grid object, a collection of multiple ggplot2 plots.

Value

grid object

See Also

FeatureEffects plot.FeatureEffect

Examples

```
# We train a random forest on the Boston dataset:
library("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 <- FeatureEffects$new(mod)

# Plot the results directly
eff$plot()

# For a subset of features
eff$plot(features = c("lstat", "crim"))

# With a different layout
eff$plot(nrows = 2)
```

---

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

Details

The plot shows the importance per feature.

When `n.repetitions` in `FeatureImp$new` was larger than 1, then we get multiple importance estimates per feature. The importance are aggregated and the plot shows the median importance per feature (as dots) and also the 90%-quantile, which helps to understand how much variance the computation has per feature.

Value

ggplot2 plot object

See Also

FeatureImp

Examples

```
library("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:
## Not run:
library("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)

## End(Not run)
```

---

plot.LocalModel *Plot Local Model*

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

```
library("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:
library("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 visualizes the distribution of the $\hat{y}$ from the machine learning model.

Usage

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

Arguments

`object`	A TreeSurrogate object.

Value

ggplot2 plot object

See Also

TreeSurrogate

Examples

```
library("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

```
library("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.

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

Usage

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

Arguments

`object`	The surrogate tree. A TreeSurrogate 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`.

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

```
library("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 for analyzing the model. The interpretation methods in the `iml` package need the machine learning model to be wrapped in a `Predictor` object.

Details

A Predictor object is a container for the prediction model and the data. This ensures that the machine learning model can be analyzed in a robust way.

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

Public fields

`data` data.frame
: Data object with the data for the model interpretation.

`model` (any)
: The machine learning model.

`batch.size numeric(1)`
: The number of rows to be input the model for prediction at once.

`class character(1)`
: The class column to be returned.

`prediction.colnames` character
: The column names of the predictions.

`prediction.function` function
: The function to predict newdata.

`task character(1)`
: The inferred prediction task: `"classification"` or `"regression"`.

Methods

Public methods:

- `Predictor$new()`
- `Predictor$predict()`
- `Predictor$print()`
- `Predictor$clone()`

Method `new()`**:** Create a Predictor object

Usage:

```
Predictor$new(
  model = NULL,
  data = NULL,
  predict.function = NULL,
  y = NULL,
  class = NULL,
  type = NULL,
  batch.size = 1000
)
```

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 analyzing the prediction model. Allowed column classes are: numeric, factor, integer, ordered and character For some models the data can be extracted automatically. `Predictor$new()` throws an error when it can't extract the data automatically.

`predict.function` function
: The function to predict newdata. Only needed if `model` is not a model from `mlr` or `caret` package. The first argument of `predict.fun` has to be the model, the second the `newdata`: `function(model, newdata)`

`y` `character(1)` | numeric | factor
: The target vector or (preferably) the name of the target column in the `data` argument. Predictor tries to infer the target automatically from the model.

`class` `character(1))`
: The class column to be returned in case of multiclass output. You can either use numbers, e.g. `class=2` would take the 2nd column from the predictions, or the column name of the predicted class, e.g. `class="dog"`.

`type` `character(1))`
: This argument is passed to the prediction function of the model. For regression models you usually don't have to provide the type argument. The classic use case is to say `type="prob"` for classification models. Consult the documentation of the machine learning package you use to find which type options you have. 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.

Method `predict()`**:** Predict new data with the machine learning model.

Usage:

```
Predictor$predict(newdata)
```

Arguments:

`newdata` data.frame
: Data to predict on.

Method `print()`**:** Print the Predictor object.

Usage:

```
Predictor$print()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Predictor$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
library("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, ])

library("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")
```

---

probs.to.labels *Turn class probabilities into class labels*

Description

Turn class probabilities into class labels

Usage

```
probs.to.labels(prediction)
```

Arguments

prediction	Prediction object.

Shapley *Prediction explanations with game theory*

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.

Details

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

Super class

iml::InterpretationMethod -> Shapley

Public fields

`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.

`sample.size numeric(1)`
: The number of times coalitions/marginals are sampled from data X. The higher the more accurate the explanations become.

Methods

Public methods:

- `Shapley$new()`
- `Shapley$explain()`
- `Shapley$clone()`

Method `new()`**:** Create a Shapley object

Usage:

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

Arguments:

`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.

Returns: data.frame
data.frame with the Shapley values (phi) per feature.

Method `explain()`**:** Set a new data point which to explain.

Usage:

```
Shapley$explain(x.interest)
```

Arguments:

`x.interest` data.frame
Single row with the instance to be explained.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Shapley$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

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

```
library("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.

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

Super class

`iml::InterpretationMethod` -> `TreeSurrogate`

Public fields

`tree party`
: The fitted tree. See also partykit::ctree.

`maxdepth numeric(1)`
: The maximum tree depth.

`r.squared numeric(1|n.classes)`
R squared measures how well the decision tree approximates the underlying model. It is calculated as 1 - (variance of prediction differences / variance of black box model predictions). For the multi-class case, r.squared contains one measure per class.

Methods

Public methods:

- `TreeSurrogate$new()`
- `TreeSurrogate$predict()`
- `TreeSurrogate$clone()`

Method `new()`**:** Create a TreeSurrogate object

Usage:

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

Arguments:

`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.

`tree.args` (named list)
Further arguments for `party::ctree()`.

Method `predict()`**:** Predict new data with the tree. See also predict.TreeSurrogate

Usage:

```
TreeSurrogate$predict(newdata, type = "prob", ...)
```

Arguments:

`newdata` data.frame
Data to predict on.

`type` Prediction type.

`...` Further arguments passed to `predict()`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
TreeSurrogate$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

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 `partykit::ctree()`

Examples

```
library("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 visualize 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