Package 'iml'

April 27, 2018

Type Package

Title Interpretable Machine Learning

Version 0.4.0

Date 2018-04-27

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>,
partial dependence plots described by Friedman (2001) <http://www.jstor.org/stable/2699986>,
individual conditional expectation ('ice') plots described by Gold-
stein 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> 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

Suggests randomForest, gower, testthat, rpart, MASS, caret, e1071,
lime, mlr, covr, knitr, rmarkdown

License MIT + file LICENSE

RoxygenNote 6.0.1

VignetteBuilder knitr

NeedsCompilation no

Author Christoph Molnar [aut, cre]

Repository CRAN

Date/Publication 2018-04-27 16:07:52 UTC

R topics documented:

`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

`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", 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.

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$Species
predict.fun = function(object, newdata) predict(object, newdata, type = "prob")
mod = Predictor$new(tree, data = X, y = y, predict.fun)

# 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$new(tree, data = X, y = y == "virginica",
  predict.fun = predict.fun, class = "virginica")
imp = FeatureImp$new(mod, loss = "ce")
plot(imp)
}
```

`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.
- 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 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.

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

`lime`, 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')]
predict.fun = function(object, newdata) predict(object, newdata, type = "prob")
mod = Predictor$new(rf, data = X, predict.fun = predict.fun, 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)
}
```

Partial *Partial Dependence and Individual Conditional Expectation*

Description

Partial computes and plots (individual) partial dependence functions of prediction models.

Format

R6Class object.

Usage

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

plot(pd)
pd$results
print(pd)
pd$set.feature(2)
pd$center(1)
```

Arguments

For Partial$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.

ice: ('logical(1)')
Should individual curves be calculated? Ignored in the case of two features.

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 partial dependence plot calculates and plots the dependence of f(X) on a single or two features. It's the aggregate of all individual conditional expectation curves, that describe how, for a single observation, the prediction changes when the feature changes.

To learn more about partial dependence plot, read the Interpretable Machine Learning book: https://christophm.github.io/interpretable-ml-book/pdp.html

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

Fields

feature.index: ('numeric(1)' | 'numeric(2)')
The index of the feature(s) for which the partial dependence was computed.

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) fpr which to compute pdp. See examples for usage.

plot() method to plot the partial dependence function. See plot.Partial

`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, 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.Partial

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
pdp.obj = Partial$new(mod, feature = "crim")

# Plot the results directly
plot(pdp.obj)

# Since the result is a ggplot object, you can extend it:
if (require("ggplot2")) {
 plot(pdp.obj) + theme_bw()
}

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

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

# Only plotting the aggregated partial dependence:
pdp.obj = Partial$new(mod, feature = "crim", ice = FALSE)
pdp.obj$plot()

# Only plotting the individual conditional expectation:
pdp.obj = Partial$new(mod, feature = "crim", aggregation = "none")
pdp.obj$plot()

# Partial dependence plots support up to two features:
pdp.obj = Partial$new(mod, feature = c("crim", "lstat"))
plot(pdp.obj)


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

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

# Partial dependence plots support up to two features:
```

```
pdp.obj = Partial$new(mod, feature = c("Sepal.Length", "Petal.Length"))
pdp.obj$plot()

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

`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.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

```
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.Partial *Plot Partial Dependence*

Description

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

Usage

```
## S3 method for class 'Partial'
plot(x, rug = TRUE)
```

Arguments

x	A Partial R6 object
rug	[logical] Should a rug be plotted to indicate the feature distribution?

Value

ggplot2 plot object

See Also

Partial

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
pdp.obj = Partial$new(mod, feature = "crim")

# Plot the results directly
plot(pdp.obj)
}
```

`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

```
if (require("randomForest")) {
# First we fit a machine learning model on the Boston housing data
data("Boston", package  = "MASS")
rf =  randomForest(medv ~ ., data = Boston, ntree = 50)
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)
}
```

`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 argumets for predict_party

Details

This function makes the TreeSurrogate object call its iternal 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 = function(object, newdata) predict(object, 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.

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)

# We need this for the randomForest
predict.fun = function(obj, newdata) {
  predict(obj, newdata = newdata, type = "prob")
}

mod = Predictor$new(rf, data = iris, predict.fun = predict.fun)
mod$predict(iris[50:55,])

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

`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.

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 (phi) 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("randomForest")) {
# First we fit a machine learning model on the Boston housing data
data("Boston", package  = "MASS")
rf =  randomForest(medv ~ ., data = Boston, ntree = 50)
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)

# Shapley() also works with multiclass classification
rf = randomForest(Species ~ ., data= iris, ntree=50)
X = iris[-which(names(iris) == "Species")]
predict.fun = function(object, newdata) predict(object, newdata, type = "prob")
mod = Predictor$new(rf, data = X, predict.fun = predict.fun)

# 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$new(rf, data = X, predict.fun = predict.fun, class = "setosa")
shapley = Shapley$new(mod, x.interest = X[1,])
shapley$results
plot(shapley)
}
```

`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.

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 - (variance of prediction differences / 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")]
predict.fun = function(object, newdata) predict(object, newdata, type = "prob")
mod = Predictor$new(rf, data = X, predict.fun = predict.fun)

# 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