Package: xplainfi (via r-universe)

May 22, 2026

Title Feature Importance Methods for Global Explanations

Version 1.1.0

Description Provides a consistent interface for common feature importance methods as described in Ewald et al. (2024) <doi:10.1007/978-3-031-63797-1_22>, including permutation feature importance (PFI), conditional and relative feature importance (CFI, RFI), leave one covariate out (LOCO), and Shapley additive global importance (SAGE), as well as feature sampling mechanisms to support conditional importance methods.

License LGPL (>= 3)

URL https://mlr-org.github.io/xplainfi/, https://github.com/mlr-org/xplainfi

BugReports https://github.com/mlr-org/xplainfi/issues

Depends R (>= 4.1.0)

Imports checkmate, cli, data.table (>= 1.15.0), mirai, mlr3 (>= 1.1.0), mlr3fselect, mvtnorm, paradox (>= 1.0.0), R6, stats, utils

Suggests arf, DiagrammeR, foreach, future, future.apply, ggplot2, glue, gower, knitr, knockoff, lgr, mlr3data, mlr3learners, partykit, patchwork, ranger, rmarkdown, rpart, testthat (>= 3.0.0), withr

VignetteBuilder knitr

Config/Needs/website rmarkdown, here, jsonlite, cpi

Config/testthat/edition 3

Config/testthat/parallel true

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

Config/pak/sysreqs cmake

Repository https://mlr-org.r-universe.dev

Date/Publication 2026-02-26 19:38:14 UTC

RemoteUrl https://github.com/mlr-org/xplainfi

RemoteRef v1.1.0

RemoteSha 5ae1b5fb029a3ff66f5b27c322b417a622436a2c

Contents

`CFI` *Conditional Feature Importance*

Description

Implementation of CFI using modular sampling approach

Details

CFI replaces feature values with conditional samples from the distribution of the feature given the other features. Any ConditionalSampler or KnockoffSampler can be used.

Statistical Inference:

Two approaches for statistical inference are primarily supported via `$importance(ci_method = "cpi")`:

- **CPI** (Watson & Wright, 2021): The original Conditional Predictive Impact method, designed for use with knockoff samplers (KnockoffGaussianSampler).
- **cARFi** (Blesch et al., 2025): CFI with ARF-based conditional sampling (ConditionalARFSampler), using the same CPI inference framework.

Both require a decomposable measure (e.g., MSE) and out-of-sample evaluation. CPI inference is guaranteed to be valid with holdout (a single train/test split). With cross-validation, test observations are i.i.d. but models are fit on overlapping training data, which may affect inference coverage. With bootstrap or subsampling, both non-i.i.d. test observations and overlapping training data can be an issue. See `vignette("inference", package = "xplainfi")` for details.

Available tests: `"t"` (t-test), `"wilcoxon"` (signed-rank), `"fisher"` (permutation), `"binomial"` (sign test). The Fisher test is recommended.

Method-agnostic inference methods (`"raw"`, `"nadeau_bengio"`, `"quantile"`) are also available; see FeatureImportanceMethod for details.

For a comprehensive overview of inference methods including usage examples, see `vignette("inference", package = "xplainfi")`.

Super classes

`xplainfi::FeatureImportanceMethod` -> `xplainfi::PerturbationImportance` -> `CFI`

Methods

Public methods:

- `CFI$new()`
- `CFI$compute()`
- `CFI$clone()`

Method `new()`**:** Creates a new instance of the CFI class

Usage:

```
CFI$new(
  task,
  learner,
  measure = NULL,
  resampling = NULL,
  features = NULL,
  groups = NULL,
  relation = "difference",
  n_repeats = 30L,
  batch_size = NULL,
  sampler = NULL
)
```

Arguments:

`task`, `learner`, `measure`, `resampling`, `features`, `groups`, `relation`, `n_repeats`, `batch_size` Passed to PerturbationImportance.

`sampler` (ConditionalSampler) Optional custom sampler. Defaults to instantiating `ConditionalARFSampler` internally with default parameters.

Method `compute()`**:** Compute CFI scores

Usage:

```
CFI$compute(
  n_repeats = NULL,
  batch_size = NULL,
  store_models = TRUE,
  store_backends = TRUE
)
```

Arguments:

`n_repeats` (`integer(1)`) Number of permutation iterations. If `NULL`, uses stored value.

`batch_size` (`integer(1)` | `NULL`: `NULL`) Maximum number of rows to predict at once. If `NULL`, uses stored value.

`store_models`, `store_backends` (`logical(1)`: `TRUE`) Whether to store fitted models / data backends, passed to mlr3::resample internally for the initial fit of the learner. This may be required for certain measures and is recommended to leave enabled unless really necessary.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Watson D, Wright M (2021). "Testing Conditional Independence in Supervised Learning Algorithms." *Machine Learning*, **110**(8), 2107–2129. doi:10.1007/s10994021060306.

Blesch K, Koenen N, Kapar J, Golchian P, Burk L, Loecher M, Wright M (2025). "Conditional Feature Importance with Generative Modeling Using Adversarial Random Forests." *Proceedings of the AAAI Conference on Artificial Intelligence*, **39**(15), 15596–15604. doi:10.1609/aaai.v39i15.33712.

Examples

```
library(mlr3)

task <- sim_dgp_correlated(n = 200)

# Using default ConditionalARFSampler
cfi <- CFI$new(
  task = task,
  learner = lrn("regr.rpart"),
  measure = msr("regr.mse"),
```

```
  sampler = ConditionalGaussianSampler$new(task),
  n_repeats = 5
)
cfi$compute()
cfi$importance()
```

---

check_groups *Check group specification*

Description

Check group specification

Usage

```
check_groups(groups, all_features)
```

Arguments

groups	(list) A (named) list of groups
all_features	(character()) All available feature names from the task.

Value

The input list group, with each element now named.

Examples

```
task <- sim_dgp_interactions(n = 100)
task$feature_names

# Intended use
groups1 = list(effects = c("x1", "x2", "x3"), noise = c("noise1", "noise2"))
check_groups(groups1, task$feature_names)

# Names are auto-generated where needed
check_groups(list(a = "x1",  c("x2", "x1")), task$feature_names)

# Examples for cases that throw errors:

# Unexpected features
groups2 = list(effects = c("x1", "foo", "bar", "x1"))
try(check_groups(groups2, task$feature_names))
# Too deeply nested
groups3 = list(effects = c("x1", "x2", "x3"), noise = c("noise1", list(c("noise2"))))
try(check_groups(groups2, task$feature_names))
```

`ConditionalARFSampler` *ARF-based Conditional Sampler*

Description

Implements conditional sampling using Adversarial Random Forests (ARF). ARF can handle mixed data types (continuous and categorical) and provides flexible conditional sampling by modeling the joint distribution.

Details

The ConditionalARFSampler fits an Adversarial Random Forest model on the task data, then uses it to generate samples from $P(X_j|X_{-j})$ where X_j is the feature of interest and X_{-j} are the conditioning features.

Super classes

`xplainfi::FeatureSampler` -> `xplainfi::ConditionalSampler` -> `ConditionalARFSampler`

Public fields

`feature_types` (`character()`) Feature types supported by the sampler. Will be checked against the provided mlr3::Task to ensure compatibility.

`arf_model` Adversarial Random Forest model created by arf::adversarial_rf.

`psi` Distribution parameters estimated from by arf::forde.

Methods

Public methods:

- `ConditionalARFSampler$new()`
- `ConditionalARFSampler$sample()`
- `ConditionalARFSampler$sample_newdata()`
- `ConditionalARFSampler$clone()`

Method `new()`**:** Creates a new instance of the ConditionalARFSampler class. To fit the ARF in parallel, register a parallel backend first (see arf::arf) and set `parallel = TRUE`.

Usage:

```
ConditionalARFSampler$new(
  task,
  conditioning_set = NULL,
  num_trees = 10L,
  min_node_size = 20L,
  finite_bounds = "no",
  epsilon = 1e-15,
  round = TRUE,
  stepsize = 0,
```

```
  verbose = FALSE,
  parallel = FALSE,
  ...
)
```

Arguments:

`task` (mlr3::Task) Task to sample from.

`conditioning_set` (`character` | `NULL`) Default conditioning set to use in `$sample()`. This parameter only affects the sampling behavior, not the ARF model fitting.

`num_trees` (`integer(1)`: `10L`) Number of trees for ARF. Passed to arf::adversarial_rf.

`min_node_size` (`integer(1)`: `20L`) Minimum node size for ARF. Passed to arf::adversarial_rf and in turn to ranger::ranger. This is increased to 20 to mitigate overfitting.

`finite_bounds` (`character(1)`: `"no"`) How to handle variable bounds. Passed to arf::forde. Default is `"no"` for compatibility. `"local"` may improve extrapolation but can cause issues with some data.

`epsilon` (`numeric(1)`: `0`) Slack parameter for when `finite_bounds != "no"`. Passed to arf::forde.

`round` (`logical(1)`: `TRUE`) Whether to round continuous variables back to their original precision in sampling. Can be overridden in `$sample()` calls.

`stepsize` (`numeric(1)`: `0`) Number of rows of evidence to process at a time when `parallel` is `TRUE`. Default (`0`) spreads evidence evenly over registered workers. Can be overridden in `$sample()` calls.

`verbose` (`logical(1)`: `FALSE`) Whether to print progress messages. Default is `FALSE` (arf's default is `TRUE`). Can be overridden in `$sample()` calls.

`parallel` (`logical(1)`: `FALSE`) Whether to use parallel processing via `foreach`. See examples in `arf::forge()`. Can be overridden in `$sample()` calls.

`...` Additional arguments passed to arf::adversarial_rf.

Method `sample()`**:** Sample from stored task. Parameters use hierarchical resolution: function argument > stored `param_set` value > hard-coded default.

Usage:

```
ConditionalARFSampler$sample(
  feature,
  row_ids = NULL,
  conditioning_set = NULL,
  round = NULL,
  stepsize = NULL,
  verbose = NULL,
  parallel = NULL
)
```

Arguments:

`feature` (`character`) Feature(s) to sample.

`row_ids` (`integer()` | `NULL`) Row IDs to use. If `NULL`, uses all rows.

`conditioning_set` (`character` | `NULL`) Features to condition on.

`round` (`logical(1)` | `NULL`) Round continuous variables.

`stepsize` (`numeric(1)` | `NULL`) Batch size for parallel processing.

`verbose` (`logical(1)` | `NULL`) Print progress messages.

`parallel` (`logical(1)` | `NULL`) Use parallel processing.

Returns: Modified copy with sampled feature(s).

Method `sample_newdata()`**:** Sample from external data. See `$sample()` for parameter details.

Usage:

```
ConditionalARFSampler$sample_newdata(
  feature,
  newdata,
  conditioning_set = NULL,
  round = NULL,
  stepsize = NULL,
  verbose = NULL,
  parallel = NULL
)
```

Arguments:

`feature` (`character`) Feature(s) to sample.

`newdata` (`data.table`) External data to use.

`conditioning_set` (`character` | `NULL`) Features to condition on.

`round` (`logical(1)` | `NULL`) Round continuous variables.

`stepsize` (`numeric(1)` | `NULL`) Batch size for parallel processing.

`verbose` (`logical(1)` | `NULL`) Print progress messages.

`parallel` (`logical(1)` | `NULL`) Use parallel processing.

Returns: Modified copy with sampled feature(s).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Watson D, Blesch K, Kapar J, Wright M (2023). "Adversarial Random Forests for Density Estimation and Generative Modeling." In *Proceedings of The 26th International Conference on Artificial Intelligence and Statistics*, 5357–5375. https://proceedings.mlr.press/v206/watson23a.html.

Blesch K, Koenen N, Kapar J, Golchian P, Burk L, Loecher M, Wright M (2025). "Conditional Feature Importance with Generative Modeling Using Adversarial Random Forests." *Proceedings of the AAAI Conference on Artificial Intelligence*, **39**(15), 15596–15604. doi:10.1609/aaai.v39i15.33712.

Examples

```
library(mlr3)
task = tgen("2dnormals")$generate(n = 100)
# Create sampler with default parameters
```

```
sampler = ConditionalARFSampler$new(task, conditioning_set = "x2", verbose = FALSE)
# Sample using row_ids from stored task
sampled_data = sampler$sample("x1", row_ids = 1:10)
# Or use external data
data = task$data()
sampled_data_ext = sampler$sample_newdata("x1", newdata = data, conditioning_set = "x2")

# Example with custom ARF parameters
sampler_custom = ConditionalARFSampler$new(
  task,
  min_node_size = 10L,
  finite_bounds = "local",
  verbose = FALSE
)
sampled_custom = sampler_custom$sample("x1", conditioning_set = "x2")
```

`ConditionalCtreeSampler` *(experimental) Conditional Inference Tree Conditional Sampler*

Description

Implements conditional sampling using conditional inference trees (ctree). Builds a tree predicting target features from conditioning features, then samples from the terminal node corresponding to each test observation.

Details

This sampler approximates the conditional distribution $P(X_B|X_A = x_A)$ by:

1. Building a conditional inference tree with X_B as response and X_A as predictors
2. For each test observation, finding its terminal (leaf) node in the tree
3. Sampling uniformly from training observations in that same terminal node

Conditional inference trees (ctree) use permutation tests to determine splits, which helps avoid overfitting and handles mixed feature types naturally. The tree partitions the feature space based on the conditioning variables, creating local neighborhoods that respect the conditional distribution structure.

Key advantages over other samplers:

- Handles mixed feature types (continuous and categorical)
- Non-parametric (no distributional assumptions)
- Automatic feature selection (splits only on informative features)
- Can capture non-linear conditional relationships
- Statistically principled splitting criteria

Hyperparameters control tree complexity:

- `mincriterion`: Significance level for splits (higher = fewer splits)
- `minsplit`: Minimum observations required for a split
- `minbucket`: Minimum observations in terminal nodes

This implementation is inspired by shapr's ctree approach but simplified for our use case (we build trees on-demand rather than pre-computing all subsets).

Advantages:

- Works with any feature types
- Robust to outliers
- Interpretable tree structure
- Handles high-dimensional conditioning

Limitations:

- Requires model fitting (slower than kNN)
- Can produce duplicates if terminal nodes are small
- Tree building time increases with data size

Super classes

`xplainfi::FeatureSampler` -> `xplainfi::ConditionalSampler` -> `ConditionalCtreeSampler`

Public fields

`feature_types` (`character()`) Feature types supported by the sampler. Will be checked against the provided mlr3::Task to ensure compatibility.

`tree_cache` (`environment`) Cache for fitted ctree models.

Methods

Public methods:

- `ConditionalCtreeSampler$new()`
- `ConditionalCtreeSampler$clone()`

Method `new()`**:** Creates a new ConditionalCtreeSampler.

Usage:

```
ConditionalCtreeSampler$new(
  task,
  conditioning_set = NULL,
  mincriterion = 0.95,
  minsplit = 20L,
  minbucket = 7L,
  use_cache = TRUE
)
```

Arguments:

`task` (mlr3::Task) Task to sample from.

`conditioning_set` (`character` | `NULL`) Default conditioning set to use in `$sample()`.

`mincriterion` (`numeric(1)`: `0.95`) Significance level threshold for splitting (1 - p-value). Higher values result in fewer splits (simpler trees).

`minsplit` (`integer(1)`: `20L`) Minimum number of observations required for a split.

`minbucket` (`integer(1)`: `7L`) Minimum number of observations in terminal nodes.

`use_cache` (`logical(1)`: `TRUE`) Whether to cache fitted trees.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Hothorn T, Hornik K, Zeileis A (2006). "Unbiased Recursive Partitioning: A Conditional Inference Framework." *Journal of Computational and Graphical Statistics*, **15**(3), 651–674. doi:10.1198/106186006X133933.

Aas K, Jullum M, Løland A (2021). "Explaining Individual Predictions When Features Are Dependent: More Accurate Approximations to Shapley Values." *Artificial Intelligence*, **298**, 103502. doi:10.1016/j.artint.2021.103502.

Examples

```
library(mlr3)
task = tgen("friedman1")$generate(n = 100)

# Create sampler with default parameters
sampler = ConditionalCtreeSampler$new(task)

# Sample features conditioned on others
test_data = task$data(rows = 1:5)
sampled = sampler$sample_newdata(
  feature = c("important2", "important3"),
  newdata = test_data,
  conditioning_set = "important1"
)
```

`ConditionalGaussianSampler` *Gaussian Conditional Sampler*

Description

Implements conditional sampling assuming features follow a multivariate Gaussian distribution. Computes conditional distributions analytically using standard formulas for multivariate normal distributions.

Details

For a joint Gaussian distribution $X \sim N(\mu, \Sigma)$, partitioned as $X = (X_A, X_B)$, the conditional distribution is:

$$X_B | X_A = x_A \sim N(\mu_{B|A}, \Sigma_{B|A})$$

where:

$$\mu_{B|A} = \mu_B + \Sigma_{BA}\Sigma_{AA}^{-1}(x_A - \mu_A)$$
$$\Sigma_{B|A} = \Sigma_{BB} - \Sigma_{BA}\Sigma_{AA}^{-1}\Sigma_{AB}$$

This is equivalent to the regression formulation used by fippy:

$$\beta = \Sigma_{BA}\Sigma_{AA}^{-1}$$

$$\mu_{B|A} = \mu_B + \beta(x_A - \mu_A)$$
$$\Sigma_{B|A} = \Sigma_{BB} - \beta\Sigma_{AB}$$

Assumptions:

- Features are approximately multivariate normal
- Only continuous features are supported

Advantages:

- Very fast (closed-form solution)
- Deterministic (given seed)
- No hyperparameters
- Memory efficient

Limitations:

- Strong distributional assumption
- May produce out-of-range values for bounded features
- Cannot handle categorical features
- Integer features are treated as continuous and rounded back to integers

Super classes

xplainfi::FeatureSampler -> xplainfi::ConditionalSampler -> ConditionalGaussianSampler

Public fields

feature_types (character()) Feature types supported by the sampler.

mu (numeric()) Mean vector estimated from training data.

sigma (matrix()) Covariance matrix estimated from training data.

Methods

Public methods:

- ConditionalGaussianSampler$new()
- ConditionalGaussianSampler$clone()

Method new(): Creates a new ConditionalGaussianSampler.

Usage:

```
ConditionalGaussianSampler$new(task, conditioning_set = NULL)
```

Arguments:

task (mlr3::Task) Task to sample from. Must have only numeric/integer features.

conditioning_set (character | NULL) Default conditioning set to use in $sample().

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

References

Anderson T (2003). *An Introduction to Multivariate Statistical Analysis*, 3rd edition. Wiley-Interscience, Hoboken, NJ. ISBN 9780471360919.

Examples

```
library(mlr3)
task = tgen("friedman1")$generate(n = 100)
sampler = ConditionalGaussianSampler$new(task)

# Sample x2, x3 conditioned on x1
test_data = task$data(rows = 1:5)
sampled = sampler$sample_newdata(
  feature = c("important2", "important3"),
  newdata = test_data,
  conditioning_set = "important1"
)
```

`ConditionalKNNSampler` *k-Nearest Neighbors Conditional Sampler*

Description

Implements conditional sampling using k-nearest neighbors (kNN). For each observation, finds the `k` most similar observations based on conditioning features, then samples the target features from these neighbors.

Details

This sampler approximates the conditional distribution $P(X_B|X_A = x_A)$ by:

1. Finding the k nearest neighbors of x_A in the training data
2. Sampling uniformly from the target feature values X_B of these k neighbors

This is a simple, non-parametric approach that:

- Requires no distributional assumptions
- Handles mixed feature types (numeric, integer, factor, ordered, logical)
- Is computationally efficient (no model fitting required)
- Adapts locally to the data structure

The method is related to hot-deck imputation and kNN imputation techniques used in missing data problems. As $k \to \infty$ and $k/n \to 0$, the kNN conditional distribution converges to the true conditional distribution under mild regularity conditions (Lipschitz continuity).

Distance Metrics:

The sampler supports two distance metrics:

- **Euclidean**: For numeric/integer features only. Standardizes features before computing distances.
- **Gower**: For mixed feature types. Handles numeric, factor, ordered, and logical features. Numeric features are range-normalized, categorical features use exact matching (0/1).

The `distance` parameter controls which metric to use:

- `"auto"` (default): Automatically selects Euclidean for all-numeric features, Gower otherwise
- `"euclidean"`: Forces Euclidean distance (errors if non-numeric features present)
- `"gower"`: Forces Gower distance (works with any feature types)

Advantages:

- Very fast (no model training)
- Works with any feature types
- Automatic distance metric selection
- Naturally respects local data structure

Limitations:

- Sensitive to choice of k
- The full task data is required for prediction
- Can produce duplicates if k is small
- May not extrapolate well to new regions

Super classes

`xplainfi::FeatureSampler` -> `xplainfi::ConditionalSampler` -> `ConditionalKNNSampler`

Public fields

`feature_types` (`character()`) Feature types supported by the sampler.

Methods

Public methods:

- `ConditionalKNNSampler$new()`
- `ConditionalKNNSampler$sample()`
- `ConditionalKNNSampler$sample_newdata()`
- `ConditionalKNNSampler$clone()`

Method `new()`**:** Creates a new ConditionalKNNSampler.

Usage:

```
ConditionalKNNSampler$new(task, conditioning_set = NULL, k = 5L)
```

Arguments:

`task` (mlr3::Task) Task to sample from.

`conditioning_set` (`character` | `NULL`) Default conditioning set to use in `$sample()`.

`k` (`integer(1)`: `5L`) Number of nearest neighbors to sample from.

Method `sample()`**:** Sample features from their kNN-based conditional distribution.

Usage:

```
ConditionalKNNSampler$sample(
  feature,
  row_ids = NULL,
  conditioning_set = NULL,
  k = NULL
)
```

Arguments:

`feature` (`character()`) Feature name(s) to sample.

`row_ids` (`integer()` | `NULL`) Row IDs from task to use as conditioning values.

`conditioning_set` (`character()` | `NULL`) Features to condition on. If `NULL`, samples from marginal distribution (random sampling from training data).

`k` (`integer(1)` | `NULL`) Number of neighbors. If `NULL`, uses stored parameter.

Returns: Modified copy with sampled feature(s).

Method `sample_newdata()`**:** Sample from external data conditionally.

Usage:

```
ConditionalKNNSampler$sample_newdata(
  feature,
  newdata,
  conditioning_set = NULL,
  k = NULL
)
```

Arguments:

`feature` (`character()`) Feature(s) to sample.

`newdata` (`data.table`) External data to use.

`conditioning_set` (`character()` | `NULL`) Features to condition on.

`k` (`integer(1)` | `NULL`) Number of neighbors. If `NULL`, uses stored parameter.

Returns: Modified copy with sampled feature(s).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Little R, Rubin D (2019). *Statistical Analysis with Missing Data*, 3rd edition. John Wiley & Sons, Hoboken, NJ. ISBN 9780470526798.

Troyanskaya O, Cantor M, Sherlock G, Brown P, Hastie T, Tibshirani R, Botstein D, Altman R (2001). "Missing Value Estimation Methods for DNA Microarrays." *Bioinformatics*, **17**(6), 520–525. doi:10.1093/bioinformatics/17.6.520.

Examples

```
library(mlr3)
task = tgen("friedman1")$generate(n = 100)
sampler = ConditionalKNNSampler$new(task, k = 5)

# Sample features conditioned on others
test_data = task$data(rows = 1:5)
sampled = sampler$sample_newdata(
  feature = c("important2", "important3"),
  newdata = test_data,
  conditioning_set = "important1"
)
```

ConditionalSAGE *Conditional SAGE*

Description

SAGE with conditional sampling (features are "marginalized" conditionally). Uses ConditionalARFSampler as default ConditionalSampler.

Super classes

xplainfi::FeatureImportanceMethod -> xplainfi::SAGE -> ConditionalSAGE

Public fields

sampler (ConditionalSampler) Sampler for conditional marginalization.

Methods

Public methods:

- ConditionalSAGE$new()
- ConditionalSAGE$clone()

Method new(): Creates a new instance of the ConditionalSAGE class.

Usage:

```
ConditionalSAGE$new(
  task,
  learner,
  measure = NULL,
  resampling = NULL,
  features = NULL,
  n_permutations = 10L,
  sampler = NULL,
  batch_size = 5000L,
  n_samples = 100L,
  early_stopping = FALSE,
  se_threshold = 0.01,
  min_permutations = 10L,
  check_interval = 1L
)
```

Arguments:

task, learner, measure, resampling, features, n_permutations, batch_size, n_samples, early_stopping, Passed to SAGE.

sampler (ConditionalSampler) Optional custom sampler. Defaults to ConditionalARFSampler.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

MarginalSAGE

Examples

```
library(mlr3)
task = tgen("friedman1")$generate(n = 100)


# Using default ConditionalARFSampler (also handles all mixed data)
sage = ConditionalSAGE$new(
  task = task,
  learner = lrn("regr.ranger", num.trees = 50),
  measure = msr("regr.mse"),
  n_permutations = 3L,
  n_samples = 20
)
sage$compute()


# For alternative conditional samplers:
custom_sampler = ConditionalGaussianSampler$new(
  task = task
)
sage_custom = ConditionalSAGE$new(
  task = task,
  learner = lrn("regr.ranger", num.trees = 50),
  measure = msr("regr.mse"),
  n_permutations = 5L,
  n_samples = 20,
  sampler = custom_sampler
)
sage_custom$compute()
```

ConditionalSampler *Conditional Feature Sampler*

Description

Base class for conditional sampling methods where features are sampled conditionally on other features. This is an abstract class that should be extended by concrete implementations.

Super class

`xplainfi::FeatureSampler` -> `ConditionalSampler`

Methods

Public methods:

- `ConditionalSampler$new()`
- `ConditionalSampler$sample()`
- `ConditionalSampler$sample_newdata()`
- `ConditionalSampler$clone()`

Method `new()`**:** Creates a new instance of the ConditionalSampler class

Usage:

```
ConditionalSampler$new(task, conditioning_set = NULL)
```

Arguments:

`task` (mlr3::Task) Task to sample from

`conditioning_set` (`character` | `NULL`) Default conditioning set to use in `$sample()`.

Method `sample()`**:** Sample from stored task conditionally on other features.

Usage:

```
ConditionalSampler$sample(
  feature,
  row_ids = NULL,
  conditioning_set = NULL,
  ...
)
```

Arguments:

`feature` (`character`) Feature(s) to sample.

`row_ids` (`integer()` | `NULL`) Row IDs to use. If `NULL`, uses all rows.

`conditioning_set` (`character` | `NULL`) Features to condition on.

`...` Additional arguments passed to the sampler implementation.

Returns: Modified copy with sampled feature(s).

Method `sample_newdata()`**:** Sample from external data conditionally.

Usage:

```
ConditionalSampler$sample_newdata(
  feature,
  newdata,
  conditioning_set = NULL,
  ...
)
```

Arguments:

`feature` (`character`) Feature(s) to sample.

`newdata` (`data.table`) External data to use.

`conditioning_set` (`character` | `NULL`) Features to condition on.

`...` Additional arguments passed to the sampler implementation.

Returns: Modified copy with sampled feature(s).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`FeatureImportanceMethod`

Feature Importance Method Class

Description

Feature Importance Method Class

Feature Importance Method Class

Public fields

`label` (`character(1)`) Method label.

`task` (mlr3::Task)

`learner` (mlr3::Learner)

`measure` (mlr3::Measure)

`resampling` (mlr3::Resampling), instantiated upon construction.

`resample_result` (mlr3::ResampleResult) of the original `learner` and `task`, used for baseline scores.

`features` (`character`: `NULL`) Features of interest. By default, importances will be computed for each feature in `task`, but optionally this can be restricted to at least one feature. Ignored if `groups` is specified.

`groups` (`list`: `NULL`) A (named) list of features (names or indices as in `task`). If `groups` is specified, `features` is ignored. Importances will be calculated for group of features at a time, e.g., in PFI not one but the group of features will be permuted at each step. Analogously in WVIM, each group of features will be left out (or in) for each model refit. Not all methods support groups (e.g., SAGE).

`param_set` (`paradox::ps()`)

`predictions` (data.table) Feature-specific prediction objects provided for some methods (PFI, WVIM). Contains columns for feature of interest, resampling iteration, refit or perturbation iteration, and mlr3::Prediction objects.

Methods

Public methods:

- FeatureImportanceMethod$new()
- FeatureImportanceMethod$compute()
- FeatureImportanceMethod$importance()
- FeatureImportanceMethod$obs_loss()
- FeatureImportanceMethod$reset()
- FeatureImportanceMethod$print()
- FeatureImportanceMethod$scores()
- FeatureImportanceMethod$clone()

Method `new()`**:** Creates a new instance of this R6 class. This is typically intended for use by derived classes.

Usage:

```
FeatureImportanceMethod$new(
  task,
  learner,
  measure = NULL,
  resampling = NULL,
  features = NULL,
  groups = NULL,
  param_set = paradox::ps(),
  label
)
```

Arguments:

`task`, `learner`, `measure`, `resampling`, `features`, `groups`, `param_set`, `label` Used to set fields

Method `compute()`**:** Compute feature importance scores

Usage:

```
FeatureImportanceMethod$compute(store_backends = TRUE)
```

Arguments:

`store_backends` (`logical(1)`: `TRUE`) Whether to store backends.

Method `importance()`**:** Get aggregated importance scores. The stored `measure` object's `aggregator` (default: `mean`) will be used to aggregated importance scores across resampling iterations and, depending on the method use, permutations (PerturbationImportance or refits LOCO).

Usage:

```
FeatureImportanceMethod$importance(
  relation = NULL,
  standardize = FALSE,
  ci_method = c("none", "raw", "nadeau_bengio", "quantile"),
  conf_level = 0.95,
  alternative = c("two.sided", "greater"),
```

```
  p_adjust = "none",
  ...
)
```

Arguments:

`relation` (character(1)) How to relate perturbed scores to originals ("difference" or "ratio"). If `NULL`, uses stored parameter value. This is only applicable for methods where importance is based on some relation between baseline and post-modification loss, i.e. PerturbationImportance methods such as PFI or WVIM / LOCO. Not available for SAGE methods.

`standardize` (`logical(1)`: `FALSE`) If `TRUE`, importances are standardized by the highest score so all scores fall in `[-1, 1]`.

`ci_method` (`character(1)`: `"none"`) Which confidence interval estimation method to use, defaulting to omitting variance estimation (`"none"`). If `"raw"`, uncorrected (too narrow) CIs are provided purely for informative purposes. If `"nadeau_bengio"`, variance correction is performed according to Nadeau & Bengio (2003) as suggested by Molnar et al. (2023). If `"quantile"`, empirical quantiles are used to construct confidence-like intervals. These methods are model-agnostic and rely on suitable `resamplings`, e.g. subsampling with 15 repeats for `"nadeau_bengio"`. See details.

`conf_level` (`numeric(1)`: `0.95`) Confidence level to use for confidence interval construction when `ci_method != "none"`.

`alternative` (`character(1)`: `"two.sided"`) Type of alternative hypothesis for statistical tests. `"greater"` tests H0: importance <= 0 vs H1: importance > 0 (one-sided). `"two.sided"` tests H0: importance = 0 vs H1: importance != 0. Only used when `ci_method != "none"`.

`p_adjust` (`character(1)`: `"none"`) Method for p-value adjustment for multiple comparisons. Accepts any method supported by stats::p.adjust.methods, e.g. `"holm"`, `"bonferroni"`, `"BH"`, `"none"`. Applied to p-values from `"raw"` and `"nadeau_bengio"` methods. When `"bonferroni"`, confidence intervals are also adjusted (alpha/k). For other correction methods (e.g. `"holm"`, `"BH"`), only p-values are adjusted; confidence intervals remain at the nominal `conf_level` because these sequential/adaptive procedures do not have a clean per-comparison alpha for CI construction.

`...` Additional arguments passed to specialized methods, if any.

Details:

Confidence Interval Methods:

The parametric methods (`"raw"`, `"nadeau_bengio"`) return standard error (`se`), test statistic (`statistic`), p-value (`p.value`), and confidence bounds (`conf_lower`, `conf_upper`). The `"quantile"` method returns only lower and upper bounds.

`"raw"`**: Uncorrected (!) t-test** Uses a standard t-test assuming independence of resampling iterations.

- SE = sd(resampling scores) / sqrt(n_iters)
- Test statistic: t = importance / SE with df = n_iters - 1
- P-value: From t-distribution (one-sided or two-sided depending on `alternative`)
- CIs: importance +/- qt(1 - alpha, df) * SE

Warning: These CIs are too narrow because resampling iterations share training data and are not independent. This method is included only for demonstration purposes.

`"nadeau_bengio"`**: Corrected t-test** Applies the Nadeau & Bengio (2003) correction to account for correlation between resampling iterations due to overlapping training sets.

- Correction factor: (1/n_iters + n_test/n_train)

- SE = sqrt(correction_factor * var(resampling scores))
- Test statistic and p-value: As in `"raw"`, but with corrected SE

Recommended with bootstrap or subsampling (>= 10 iterations).
`"quantile"`**: Non-parametric empirical method** Uses the resampling distribution directly without parametric assumptions.

- CIs: Empirical quantiles of the resampling distribution

This method does not provide `se`, `statistic`, or `p.value`.

Method-Specific CI Methods:
Some importance methods provide additional CI methods tailored to their approach:

- **CFI**: Adds `"cpi"` (Conditional Predictive Impact), which uses observation-wise loss differences with holdout resampling. Supports t-test, Wilcoxon, Fisher permutation, and binomial tests. See Watson & Wright (2021).

Practical Recommendations:
Variance estimates for importance scores are biased due to the resampling procedure. Molnar et al. (2023) suggest using the Nadeau & Bengio correction with approximately 15 iterations of subsampling.
Bootstrapping can cause information leakage with learners that bootstrap internally (e.g., Random Forests), as observations may appear in both train and test sets. Prefer subsampling in such cases:

```
PFI$new(
  task = sim_dgp_interactions(n = 1000),
  learner = lrn("regr.ranger", num.trees = 100),
  measure = msr("regr.mse"),
  resampling = rsmp("subsampling", repeats = 15),
  n_repeats = 20
)
```

The `"nadeau_bengio"` correction was validated for PFI; its use with other methods like LOCO or SAGE is experimental.

Returns: (data.table) Aggregated importance scores with columns `"feature"`, `"importance"`, and depending on `ci_method` also `"se"`, `"statistic"`, `"p.value"`, `"conf_lower"`, `"conf_upper"`.

Method `obs_loss()`**:** Calculate observation-wise importance scores.

Requires that `$compute()` was run and that `measure` is decomposable and has an observation-wise loss (`Measure$obs_loss()`) associated with it. This is not the case for measure like `classif.auc`, which is not decomposable.

Usage:

```
FeatureImportanceMethod$obs_loss(relation = NULL)
```

Arguments:

`relation` (character(1)) How to relate perturbed scores to originals ("difference" or "ratio"). If `NULL`, uses stored parameter value. This is only applicable for methods where importance is based on some relation between baseline and post-modification loss, i.e. PerturbationImportance methods such as PFI or WVIM / LOCO. Not available for SAGE methods.

Returns: (data.table) Observation-wise losses and importance scores with columns `"feature"`, `"iter_rsmp"`, `"iter_repeat"` (if applicable), `"row_ids"`, `"loss_baseline"`, `"loss_post"`, and `"obs_importance"`.

Method `reset()`**:** Resets all stored fields populated by `$compute`: `$resample_result`, `$scores`, `$obs_losses`, and `$predictions`.

Usage:

```
FeatureImportanceMethod$reset()
```

Method `print()`**:** Print importance scores

Usage:

```
FeatureImportanceMethod$print(...)
```

Arguments:

`...` Passed to `print()`

Method `scores()`**:** Calculate importance scores for each resampling iteration and sub-iterations (`iter_rsmp` in PFI for example).

Iteration-wise importance are computed on the fly depending on the chosen relation (`difference` or `ratio`) to avoid re-computation if only a different relation is needed.

Usage:

```
FeatureImportanceMethod$scores(relation = NULL)
```

Arguments:

`relation` (character(1)) How to relate perturbed scores to originals ("difference" or "ratio"). If `NULL`, uses stored parameter value. This is only applicable for methods where importance is based on some relation between baseline and post-modification loss, i.e. PerturbationImportance methods such as PFI or WVIM / LOCO. Not available for SAGE methods.

Returns: (data.table) Iteration-wise importance scores with columns for `"feature"`, iteration indices, baseline and post-modification scores, and `"importance"`.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Nadeau C, Bengio Y (2003). "Inference for the Generalization Error." *Machine Learning*, **52**(3), 239–281. doi:10.1023/A:1024068626366. Molnar C, Freiesleben T, König G, Herbinger J, Reisinger T, Casalicchio G, Wright M, Bischl B (2023). "Relating the Partial Dependence Plot and Permutation Feature Importance to the Data Generating Process." In Longo L (ed.), *Explainable Artificial Intelligence*, 456–479. ISBN 978-3-031-44064-9, doi:10.1007/9783031440649_24.

`FeatureSampler` *Feature Sampler Class*

Description

Base class for implementing different sampling strategies for feature importance methods like PFI and CFI

Public fields

`task` (mlr3::Task) Original task.

`label` (`character(1)`) Name of the sampler.

`feature_types` (`character()`) Feature types supported by the sampler. Will be checked against the provided mlr3::Task to ensure compatibility.

`param_set` (paradox::ParamSet) Parameter set for the sampler.

Methods

Public methods:

- `FeatureSampler$new()`
- `FeatureSampler$sample()`
- `FeatureSampler$sample_newdata()`
- `FeatureSampler$print()`
- `FeatureSampler$clone()`

Method `new()`**:** Creates a new instance of the FeatureSampler class

Usage:

```
FeatureSampler$new(task)
```

Arguments:

`task` (mlr3::Task) Task to sample from

Method `sample()`**:** Sample values for feature(s) from stored task

Usage:

```
FeatureSampler$sample(feature, row_ids = NULL)
```

Arguments:

`feature` (`character`) Feature name(s) to sample (can be single or multiple). Must match those in the stored Task.

`row_ids` (`integer()`: `NULL`) Row IDs of the stored Task to use as basis for sampling.

Returns: Modified copy of the input features with the feature(s) sampled: A data.table with same number of columns and one row matching the supplied `row_ids`

Method `sample_newdata()`**:** Sample values for feature(s) using external data

Usage:

```
FeatureSampler$sample_newdata(feature, newdata)
```

Arguments:

`feature` (`character`) Feature name(s) to sample (can be single or multiple)

`newdata` (`data.table`) External data to use for sampling.

Method `print()`**:** Print sampler

Usage:

```
FeatureSampler$print(...)
```

Arguments:

`...` Ignored.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`KnockoffGaussianSampler`

Gaussian Knockoff Conditional Sampler

Description

A KnockoffSampler defaulting to second-order Gaussian knockoffs as created by knockoff::create.second_order.

Details

This is equivalent to KnockoffSampler using the default `knockoff_fun`.

Super classes

`xplainfi::FeatureSampler` -> `xplainfi::KnockoffSampler` -> `KnockoffGaussianSampler`

Public fields

`feature_types` (`character()`) Feature types supported by the sampler. Will be checked against the provided mlr3::Task to ensure compatibility.

`x_tilde` Knockoff matrix

Methods

Public methods:

- KnockoffGaussianSampler$new()
- KnockoffGaussianSampler$clone()

Method `new()`**:** Creates a new instance using Gaussian knockoffs via knockoff::create.second_order.

Usage:

```
KnockoffGaussianSampler$new(task, iters = 1)
```

Arguments:

`task` (mlr3::Task) Task to sample from.

`iters` (`integer(1)`: 1) Number of repetitions the `knockoff_fun` is applied to create multiple `x_tilde` instances per observation.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Watson D, Wright M (2021). "Testing Conditional Independence in Supervised Learning Algorithms." *Machine Learning*, **110**(8), 2107–2129. doi:10.1007/s10994021060306.

Blesch K, Watson D, Wright M (2023). "Conditional Feature Importance for Mixed Data." *AStA Advances in Statistical Analysis*, **108**(2), 259–278. doi:10.1007/s10182023004779.

Examples

```
library(mlr3)
task = tgen("2dnormals")$generate(n = 100)
# Create sampler
sampler = KnockoffGaussianSampler$new(task)
# Sample using row_ids from stored task
sampled_data = sampler$sample("x1")
```

`KnockoffSampler` *Knockoff Sampler*

Description

Implements conditional sampling using Knockoffs.

Details

The KnockoffSampler samples Knockoffs based on the task data. This class allows arbitrary knockoff_fun, which also means that no input checking against supported feature types can be done. Use KnockoffGaussianSampler for the Gaussian knockoff sampler for numeric features. Alternative knockoff samplers include knockoff_seq() from the seqknockoff package available on GitHub: https://github.com/kormama1/seqknockoff.

Knockoffs are related to the ConditionalSampler familty, with key differences: They do not allow specifying a conditioning_set

Super class

xplainfi::FeatureSampler -> KnockoffSampler

Public fields

x_tilde Knockoff matrix with one (or iters) row(s) per original observation in task.

Methods

Public methods:

- KnockoffSampler$new()
- KnockoffSampler$sample()
- KnockoffSampler$clone()

Method new(): Creates a new instance of the KnockoffSampler class.

Usage:

```
KnockoffSampler$new(
  task,
  knockoff_fun = function(x) knockoff::create.second_order(as.matrix(x)),
  iters = 1
)
```

Arguments:

task (mlr3::Task) Task to sample from

knockoff_fun (function) Function used to create knockoff matrix. Default are second-order Gaussian knockoffs (knockoff::create.second_order())

iters (integer(1): 1) Number of repetitions the knockoff_fun is applied to create multiple x_tilde instances per observation.

Method sample(): Sample from stored task using knockoff values. Replaces specified feature(s) with their knockoff counterparts from the pre-generated knockoff matrix.

Usage:

```
KnockoffSampler$sample(feature, row_ids = NULL)
```

Arguments:

feature (character) Feature(s) to sample.

row_ids (integer() | NULL) Row IDs to use. If NULL, uses all rows.

Returns: Modified copy with knockoff feature(s).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Watson D, Wright M (2021). "Testing Conditional Independence in Supervised Learning Algorithms." *Machine Learning*, **110**(8), 2107–2129. doi:10.1007/s10994021060306.

Blesch K, Watson D, Wright M (2023). "Conditional Feature Importance for Mixed Data." *AStA Advances in Statistical Analysis*, **108**(2), 259–278. doi:10.1007/s10182023004779.

Examples

```
library(mlr3)
task = tgen("2dnormals")$generate(n = 100)
# Create sampler with default parameters
sampler = KnockoffSampler$new(task)
# Sample using row_ids from stored task
sampled_data = sampler$sample("x1")
```

`LOCO` *Leave-One-Covariate-Out (LOCO)*

Description

Calculates Leave-One-Covariate-Out (LOCO) scores.

Details

LOCO measures feature importance by comparing model performance with and without each feature. For each feature, the model is retrained without that feature and the performance difference (reduced_model_loss - full_model_loss) indicates the feature's importance. Higher values indicate more important features.

Super classes

`xplainfi::FeatureImportanceMethod` -> `xplainfi::WVIM` -> `LOCO`

Methods

Public methods:

- `LOCO$new()`
- `LOCO$compute()`
- `LOCO$clone()`

Method `new()`**:** Creates a new instance of this R6 class.

Usage:

```
LOCO$new(
  task,
  learner,
  measure = NULL,
  resampling = NULL,
  features = NULL,
  n_repeats = 30L
)
```

Arguments:

`task` (mlr3::Task) Task to compute importance for.

`learner` (mlr3::Learner) Learner to use for prediction.

`measure` (mlr3::Measure: `NULL`) Measure to use for scoring. Defaults to `classif.ce` for classification and `regr.mse` for regression.

`resampling` (mlr3::Resampling) Resampling strategy. Defaults to holdout.

`features` (`character()`) Features to compute importance for. Defaults to all features.

`n_repeats` (`integer(1)`: `30L`) Number of refit iterations per resampling iteration.

Method `compute()`**:** Compute LOCO importances.

Usage:

```
LOCO$compute(store_models = TRUE, store_backends = TRUE)
```

Arguments:

`store_models`, `store_backends` (`logical(1)`: `TRUE`) Whether to store fitted models / data backends, passed to mlr3::resample internally

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Lei J, G'Sell M, Rinaldo A, Tibshirani R, Wasserman L (2018). "Distribution-Free Predictive Inference for Regression." *Journal of the American Statistical Association*, **113**(523), 1094–1111. doi:10.1080/01621459.2017.1307116.

Examples

```
library(mlr3)
library(mlr3learners)

task <- sim_dgp_correlated(n = 500)

loco <- LOCO$new(
  task = task,
  learner = lrn("regr.rpart"),
  measure = msr("regr.mse"),
  n_repeats = 5
)
loco$compute()
loco$importance()
```

MarginalPermutationSampler

Marginal Permutation Sampler

Description

Implements marginal permutation-based sampling for Permutation Feature Importance (PFI). Each specified feature is randomly shuffled (permuted) independently, breaking the relationship between the feature and the target as well as between rows.

Details

The permutation sampler randomly shuffles feature values across observations:

- Each feature is permuted **independently** within its column
- The association between feature values and target values is broken
- The association between feature values **across rows** is broken
- The marginal distribution of each feature is preserved

Important distinction from SAGE's "marginal" approach:

- `MarginalPermutationSampler`: Shuffles features independently, breaking row structure
- `MarginalSAGE`: Uses reference data but keeps rows intact (features in coalition stay together)

This is the classic approach used in Permutation Feature Importance (PFI) and assumes features are independent.

Super classes

`xplainfi::FeatureSampler` -> `xplainfi::MarginalSampler` -> `MarginalPermutationSampler`

Methods

Public methods:

- `MarginalPermutationSampler$new()`
- `MarginalPermutationSampler$clone()`

Method `new()`**:** Creates a new instance of the MarginalPermutationSampler class.

Usage:

```
MarginalPermutationSampler$new(task)
```

Arguments:

`task` (mlr3::Task) Task to sample from.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
library(mlr3)
task = tgen("2dnormals")$generate(n = 10)
task$data()
sampler = MarginalPermutationSampler$new(task)

# Sample using row_ids from stored task
sampler$sample("x1")

# Or use external data
data = task$data()
sampler$sample_newdata("x1", newdata = data)
```

`MarginalReferenceSampler`

Marginal Reference Sampler

Description

Samples complete observations from reference data to replace feature values. This approach samples from the marginal distribution while preserving within-row feature dependencies.

Details

This sampler implements what is called "marginal imputation" in the SAGE literature (Covert et al. 2020). For each observation, it samples a complete row from reference data and takes the specified feature values from that row. This approach:

- Samples from the marginal distribution $P(X_S)$ where S is the set of features
- Preserves dependencies **within** the sampled reference row
- Breaks dependencies **between** test and reference data

Terminology note: In SAGE literature, this is called "marginal imputation" because features outside the coalition are "imputed" by sampling from their marginal distribution. We use `MarginalReferenceSampler` to avoid confusion with missing data imputation and to clarify that it samples from reference data.

Comparison with other samplers:

- `MarginalPermutationSampler`: Shuffles each feature independently, breaking all row structure
- `MarginalReferenceSampler`: Samples complete rows, preserving within-row dependencies
- `ConditionalSampler`: Samples from $P(X_S|X_{-S})$, conditioning on other features

Use in SAGE:

This is the default approach for `MarginalSAGE`. For a test observation x and features to marginalize S, it samples a reference row x_ref and creates a "hybrid" observation combining x's coalition features with x_ref's marginalized features.

Super classes

`xplainfi::FeatureSampler` -> `xplainfi::MarginalSampler` -> `MarginalReferenceSampler`

Public fields

`reference_data` (`data.table`) Reference data to sample from for marginalization.

Methods

Public methods:

- `MarginalReferenceSampler$new()`
- `MarginalReferenceSampler$clone()`

Method `new()`**:** Creates a new instance of the MarginalReferenceSampler class.

Usage:

```
MarginalReferenceSampler$new(task, n_samples = NULL)
```

Arguments:

`task` (mlr3::Task) Task to sample from.

`n_samples` (`integer(1)` | `NULL`) Number of reference samples to use. If `NULL`, uses all task data as reference.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

References

Covert I, Lundberg S, Lee S (2020). "Understanding Global Feature Contributions With Additive Importance Measures." In *Advances in Neural Information Processing Systems*, volume 33, 17212–17223. https://proceedings.neurips.cc/paper/2020/hash/c7bf0b7c1a86d5eb3be2c722cf2cf746-Abstract.html.

Examples

```
library(mlr3)
task = tgen("friedman1")$generate(n = 100)

# Default: uses all task data as reference
sampler = MarginalReferenceSampler$new(task)
sampled = sampler$sample("important1", row_ids = 1:10)

# Subsample reference data to 50 rows
sampler_subsampled = MarginalReferenceSampler$new(task, n_samples = 50L)
sampled2 = sampler_subsampled$sample("important1", row_ids = 1:10)
```

MarginalSAGE *Marginal SAGE*

Description

SAGE with marginal sampling (features are marginalized independently). This is the standard SAGE implementation.

Super classes

xplainfi::FeatureImportanceMethod -> xplainfi::SAGE -> MarginalSAGE

Methods

Public methods:

- MarginalSAGE$new()
- MarginalSAGE$clone()

Method new(): Creates a new instance of the MarginalSAGE class.

Usage:

```
MarginalSAGE$new(
  task,
  learner,
  measure = NULL,
  resampling = NULL,
  features = NULL,
  n_permutations = 10L,
  batch_size = 5000L,
  n_samples = 100L,
  early_stopping = FALSE,
  se_threshold = 0.01,
  min_permutations = 10L,
  check_interval = 1L
)
```

Arguments:

`task`, `learner`, `measure`, `resampling`, `features`, `n_permutations`, `batch_size`, `n_samples`, `early_stopping`, Passed to SAGE.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

ConditionalSAGE

Examples

```
library(mlr3)
task = tgen("friedman1")$generate(n = 100)
sage = MarginalSAGE$new(
  task = task,
  learner = lrn("regr.ranger", num.trees = 50),
  measure = msr("regr.mse"),
  n_permutations = 3L,
  n_samples = 20
)
sage$compute()
```

`MarginalSampler` *Marginal Sampler Base Class*

Description

Abstract base class for marginal sampling strategies that do not condition on other features. Marginal samplers sample from P(X_S), the marginal distribution of features S, ignoring any dependencies with other features.

Details

This class provides a common interface for different marginal sampling approaches:

- **MarginalPermutationSampler**: Shuffles features independently within the dataset
- **MarginalReferenceSampler**: Samples complete rows from reference data

Both approaches sample from the marginal distribution P(X_S), but differ in how they preserve or break within-row dependencies:

- Permutation breaks ALL dependencies (both with target and between features)
- Reference sampling preserves WITHIN-row dependencies but breaks dependencies with test data

Comparison with ConditionalSampler:

- `MarginalSampler`: Samples from $P(X_S)$ - no conditioning
- `ConditionalSampler`: Samples from $P(X_S|X_{-S})$- conditions on other features

This base class implements the public `$sample()` and `$sample_newdata()` methods, delegating to private `.sample_marginal()` which subclasses must implement.

Super class

`xplainfi::FeatureSampler` -> `MarginalSampler`

Methods

Public methods:

- `MarginalSampler$sample()`
- `MarginalSampler$sample_newdata()`
- `MarginalSampler$clone()`

Method `sample()`**:** Sample features from their marginal distribution.

Usage:

```
MarginalSampler$sample(feature, row_ids = NULL)
```

Arguments:

`feature` (`character()`) Feature name(s) to sample.

`row_ids` (`integer()` | `NULL`) Row IDs from task to use.

Returns: Modified copy with sampled feature(s).

Method `sample_newdata()`**:** Sample from external data.

Usage:

```
MarginalSampler$sample_newdata(feature, newdata)
```

Arguments:

`feature` (`character()`) Feature(s) to sample.

`newdata` (`data.table`) External data to use.

Returns: Modified copy with sampled feature(s).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`op-null-default` *Default value for* `NULL`

Description

A backport of `%||%` available in R versions from 4.4.0.

Usage

```
x %||% y
```

Arguments

`x`, `y`	If `x` is NULL or length 0, will return `y`; otherwise returns `x`.

Examples

```
1 %||% 2
NULL %||% 2
```

PerturbationImportance

Perturbation Feature Importance Base Class

Description

Abstract base class for perturbation-based importance methods PFI, CFI, and RFI

Super class

xplainfi::FeatureImportanceMethod -> PerturbationImportance

Public fields

sampler (FeatureSampler) Sampler object for feature perturbation

Methods

Public methods:

- PerturbationImportance$new()
- PerturbationImportance$importance()
- PerturbationImportance$clone()

Method new(): Creates a new instance of the PerturbationImportance class

Usage:

```
PerturbationImportance$new(
  task,
  learner,
  measure = NULL,
  resampling = NULL,
  features = NULL,
  groups = NULL,
  sampler = NULL,
  relation = "difference",
  n_repeats = 30L,
  batch_size = NULL
)
```

Arguments:

task, learner, measure, resampling, features, groups Passed to FeatureImportanceMethod.

sampler (FeatureSampler) Sampler to use for feature perturbation.

relation (character(1): "difference") How to relate perturbed and baseline scores. Can also be "ratio".

n_repeats (integer(1): 30L) Number of permutation/conditional sampling iterations. Can also be overridden in $compute().

batch_size (integer(1) | NULL: NULL) Maximum number of rows to predict at once. When NULL, predicts all test_size * n_repeats rows in one call. Use smaller values to reduce memory usage at the cost of more prediction calls. Can be overridden in $compute().

Method importance(): Get aggregated importance scores. Extends the base $importance() method to support ci_method = "cpi". For details, see CFI, which is the only sub-method for which it is known to be valid.

Usage:

```
PerturbationImportance$importance(
  relation = NULL,
  standardize = FALSE,
  ci_method = c("none", "raw", "nadeau_bengio", "quantile", "cpi"),
  conf_level = 0.95,
  alternative = c("two.sided", "greater"),
  test = c("t", "wilcoxon", "fisher", "binomial"),
  B = 1999,
  p_adjust = "none",
  ...
)
```

Arguments:

relation (character(1)) How to relate perturbed scores to originals ("difference" or "ratio"). If NULL, uses stored parameter value.

standardize (logical(1): FALSE) If TRUE, importances are standardized by the highest score so all scores fall in [-1, 1].

ci_method (character(1): "none") Variance estimation method. In addition to base methods ("none", "raw", "nadeau_bengio", "quantile"), perturbation methods support "cpi" (Conditional Predictive Impact). CPI is specifically designed for CFI with knockoff samplers and uses one-sided hypothesis tests.

conf_level (numeric(1): 0.95) Confidence level for confidence intervals when ci_method != "none".

alternative (character(1): "two.sided") Type of alternative hypothesis for statistical tests. "greater" tests H0: importance <= 0 vs H1: importance > 0 (one-sided). "two.sided" tests H0: importance = 0 vs H1: importance != 0.

test (character(1): "t") Test to use for CPI. One of "t", "wilcoxon", "fisher", or "binomial". Only used when ci_method = "cpi".

B (integer(1): 1999) Number of replications for Fisher test. Only used when ci_method = "cpi" and test = "fisher".

p_adjust (character(1): "none") Method for p-value adjustment for multiple comparisons. Accepts any method supported by stats::p.adjust.methods, e.g. "holm", "bonferroni", "BH", "none". When "bonferroni", confidence intervals are also adjusted (alpha/k). For other correction methods (e.g. "holm", "BH"), only p-values are adjusted; confidence intervals remain at the nominal conf_level because these sequential/adaptive procedures do not have a clean per-comparison alpha for CI construction.

... Additional arguments passed to the base method.

Returns: (data.table) Aggregated importance scores.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`PFI` *Permutation Feature Importance*

Description

Implementation of Permutation Feature Importance (PFI) using modular sampling approach. PFI measures the importance of a feature by calculating the increase in model error when the feature's values are randomly permuted, breaking the relationship between the feature and the target variable.

Details

Permutation Feature Importance was originally introduced by Breiman (2001) as part of the Random Forest algorithm. The method works by:

1. Computing baseline model performance on the original dataset
2. For each feature, randomly permuting its values while keeping other features unchanged
3. Computing model performance on the permuted dataset
4. Calculating importance as the difference (or ratio) between permuted and original performance

Super classes

`xplainfi::FeatureImportanceMethod` -> `xplainfi::PerturbationImportance` -> `PFI`

Methods

Public methods:

- `PFI$new()`
- `PFI$compute()`
- `PFI$clone()`

Method `new()`**:** Creates a new instance of the PFI class

Usage:

```
PFI$new(
  task,
  learner,
  measure = NULL,
  resampling = NULL,
  features = NULL,
  groups = NULL,
```

```
  relation = "difference",
  n_repeats = 30L,
  batch_size = NULL
)
```

Arguments:

`task`, `learner`, `measure`, `resampling`, `features`, `groups`, `relation`, `n_repeats`, `batch_size`
Passed to PerturbationImportance

Method `compute()`**:** Compute PFI scores

Usage:

```
PFI$compute(
  n_repeats = NULL,
  batch_size = NULL,
  store_models = TRUE,
  store_backends = TRUE
)
```

Arguments:

`n_repeats` (`integer(1)`; `NULL`) Number of permutation iterations. If `NULL`, uses stored value.

`batch_size` (`integer(1)` | `NULL`: `NULL`) Maximum number of rows to predict at once. If `NULL`, uses stored value.

`store_models`, `store_backends` (`logical(1)`: `TRUE`) Whether to store fitted models / data backends, passed to mlr3::resample internally for the initial fit of the learner. This may be required for certain measures and is recommended to leave enabled unless really necessary.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

Breiman L (2001). "Random Forests." *Machine Learning*, **45**(1), 5–32. doi:10.1023/A:1010933404324. Fisher A, Rudin C, Dominici F (2019). "All Models Are Wrong, but Many Are Useful: Learning a Variable's Importance by Studying an Entire Class of Prediction Models Simultaneously." *Journal of Machine Learning Research*, **20**, 177. https://pmc.ncbi.nlm.nih.gov/articles/PMC8323609/. Strobl C, Boulesteix A, Kneib T, Augustin T, Zeileis A (2008). "Conditional Variable Importance for Random Forests." *BMC Bioinformatics*, **9**(1), 307. doi:10.1186/147121059307.

Examples

```
library(mlr3)

task <- sim_dgp_correlated(n = 500)

pfi <- PFI$new(
  task = task,
```

```
  learner = lrn("regr.rpart"),
  measure = msr("regr.mse"),
n_repeats = 5
)
pfi$compute()
pfi$importance()
```

RFI *Relative Feature Importance*

Description

RFI generalizes CFI and PFI with arbitrary conditioning sets and samplers.

Super classes

`xplainfi::FeatureImportanceMethod` -> `xplainfi::PerturbationImportance` -> `RFI`

Methods

Public methods:

- `RFI$new()`
- `RFI$compute()`
- `RFI$clone()`

Method `new()`**:** Creates a new instance of the RFI class

Usage:

```
RFI$new(
  task,
  learner,
  measure = NULL,
  resampling = NULL,
  features = NULL,
  groups = NULL,
  conditioning_set = NULL,
  relation = "difference",
  n_repeats = 30L,
  batch_size = NULL,
  sampler = NULL
)
```

Arguments:

`task`, `learner`, `measure`, `resampling`, `features`, `groups`, `relation`, `n_repeats`, `batch_size` Passed to PerturbationImportance.

`conditioning_set` (`character()`) Set of features to condition on. Can be overridden in `$compute()`. Default (`character(0)`) is equivalent to `PFI`. In `CFI`, this would be set to all features except that of interest.

`sampler` (ConditionalSampler) Optional custom sampler. Defaults to `ConditionalARFSampler`.

Method `compute()`**:** Compute RFI scores

Usage:

```
RFI$compute(
  conditioning_set = NULL,
  n_repeats = NULL,
  batch_size = NULL,
  store_models = TRUE,
  store_backends = TRUE
)
```

Arguments:

`conditioning_set` (`character()`) Set of features to condition on. If `NULL`, uses the stored parameter value.

`n_repeats` (`integer(1)`) Number of permutation iterations. If `NULL`, uses stored value.

`batch_size` (`integer(1)` | `NULL`: `NULL`) Maximum number of rows to predict at once. If `NULL`, uses stored value.

`store_models`, `store_backends` (`logical(1)`: `TRUE`) Whether to store fitted models / data backends, passed to mlr3::resample internally for the initial fit of the learner. This may be required for certain measures and is recommended to leave enabled unless really necessary.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

König G, Molnar C, Bischl B, Grosse-Wentrup M (2021). "Relative Feature Importance." In *2020 25th International Conference on Pattern Recognition (ICPR)*, 9318–9325. doi:10.1109/ICPR48806.2021.9413090.

Examples

```
library(mlr3)
task = tgen("friedman1")$generate(n = 200)
rfi = RFI$new(
  task = task,
  learner = lrn("regr.rpart"),
  measure = msr("regr.mse"),
  conditioning_set = c("important1"),
  sampler = ConditionalGaussianSampler$new(task),
  n_repeats = 5
)
rfi$compute()
rfi$importance()
```

rsmp_all_test *Create a resampling with all data being test data*

Description

Utility for use with a pretrained learner in importance methods which support it

Usage

```
rsmp_all_test(task)
```

Arguments

task	(mlr3::Task)

Details

Note that the resulting Resampling will have an empty train set, making it useless for any other purpose than the use with a pretrained learner.

Value

mlr3::Resampling with an empty `train_set` and a single `test_set` identical to all of the given `Task`.

Examples

```
library(mlr3)
# Create custom task from some data.frame
custom_task <- as_task_regr(mtcars, target = "mpg")
# Create matching Resampling with all-test data
resampling_custom <- rsmp_all_test(custom_task)
```

SAGE *Shapley Additive Global Importance (SAGE) Base Class*

Description

Base class for SAGE (Shapley Additive Global Importance) feature importance based on Shapley values with marginalization. This is an abstract class - use MarginalSAGE or ConditionalSAGE.

Details

SAGE uses Shapley values to fairly distribute the total prediction performance among all features. Unlike perturbation-based methods, SAGE marginalizes features by integrating over their distribution. This is approximated by averaging predictions over a reference dataset.

Standard Error Calculation: The standard errors (SE) reported in `$convergence_history` reflect the uncertainty in Shapley value estimation across different random permutations within a single resampling iteration. These SEs quantify the Monte Carlo sampling error for a fixed trained model and are only valid for inference about the importance of features for that specific model. They do not capture broader uncertainty from model variability across different train/test splits or resampling iterations.

Super class

`xplainfi::FeatureImportanceMethod` -> `SAGE`

Public fields

`n_permutations` (`integer(1)`) Number of permutations to sample.

`convergence_history` (`data.table`) History of SAGE values during computation.

`converged` (`logical(1)`) Whether convergence was detected.

`n_permutations_used` (`integer(1)`) Actual number of permutations used.

Methods

Public methods:

- `SAGE$new()`
- `SAGE$compute()`
- `SAGE$plot_convergence()`
- `SAGE$clone()`

Method `new()`**:** Creates a new instance of the SAGE class.

Usage:

```
SAGE$new(
  task,
  learner,
  measure = NULL,
  resampling = NULL,
  features = NULL,
  n_permutations = 10L,
  batch_size = 5000L,
  n_samples = 100L,
  early_stopping = TRUE,
  se_threshold = 0.01,
  min_permutations = 10L,
  check_interval = 1L
)
```

Arguments:

`task`, `learner`, `measure`, `resampling`, `features` Passed to FeatureImportanceMethod.

`n_permutations` (`integer(1)`: `10L`) Number of permutations to sample for SAGE value estimation. The total number of evaluated coalitions is `1` (`empty`) + `n_permutations` * `n_features`.

`batch_size` (`integer(1)`: `5000L`) Maximum number of observations to process in a single prediction call.

`n_samples` (`integer(1)`: `100L`) Number of samples to use for marginalizing out-of-coalition features. For MarginalSAGE, this is the number of marginal data samples ("background data" in other implementations). For ConditionalSAGE, this is the number of conditional samples per test instance retrieved from `sampler`.

`early_stopping` (`logical(1)`: `TRUE`) Whether to enable early stopping based on convergence detection.

`se_threshold` (`numeric(1)`: `0.01`) Convergence threshold for relative standard error. Convergence is detected when the maximum relative SE across all features falls below this threshold. Relative SE is calculated as SE divided by the range of importance values (max - min), making it scale-invariant across different loss metrics. Default of `0.01` means convergence when relative SE is below 1% of the importance range.

`min_permutations` (`integer(1)`: `10L`) Minimum permutations before checking for convergence. Convergence is judged based on the standard errors of the estimated SAGE values, which requires a sufficiently large number of samples (i.e., evaluated coalitions).

`check_interval` (`integer(1)`: `1L`) Check convergence every N permutations.

Method `compute()`**:** Compute SAGE values.

Usage:

```
SAGE$compute(
  store_backends = TRUE,
  batch_size = NULL,
  early_stopping = NULL,
  se_threshold = NULL,
  min_permutations = NULL,
  check_interval = NULL
)
```

Arguments:

`store_backends` (`logical(1)`) Whether to store data backends.

`batch_size` (`integer(1)`: `5000L`) Maximum number of observations to process in a single prediction call.

`early_stopping` (`logical(1)`: `TRUE`) Whether to check for convergence and stop early.

`se_threshold` (`numeric(1)`: `0.01`) Convergence threshold for relative standard error. SE is normalized by the range of importance values (max - min) to make convergence detection scale-invariant. Default `0.01` means convergence when relative SE < 1%.

`min_permutations` (`integer(1)`: `10L`) Minimum permutations before checking convergence.

`check_interval` (`integer(1)`: `1L`) Check convergence every N permutations.

Method `plot_convergence()`**:** Plot convergence history of SAGE values.

Usage:

`SAGE$plot_convergence(features = NULL)`

Arguments:

`features` (`character` | `NULL`) Features to plot. If NULL, plots all features.

Returns: A ggplot2 object

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

Usage:

`SAGE$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

References

Covert I, Lundberg S, Lee S (2020). "Understanding Global Feature Contributions With Additive Importance Measures." In *Advances in Neural Information Processing Systems*, volume 33, 17212–17223. https://proceedings.neurips.cc/paper/2020/hash/c7bf0b7c1a86d5eb3be2c722cf2cf746-Abstract.html.

See Also

MarginalSAGE ConditionalSAGE

`sim_dgp_ewald` *Simulate data as in Ewald et al. (2024)*

Description

Reproduces the data generating process from Ewald et al. (2024) for benchmarking feature importance methods. Includes correlated features and interaction effects.

Usage

```
sim_dgp_ewald(n = 500)
```

Arguments

`n`	(`integer(1)`) Number of samples to create.

Details

Mathematical Model:

$$X_1, X_3, X_5 \sim \text{Uniform}(0, 1)$$
$$X_2 = X_1 + \varepsilon_2, \quad \varepsilon_2 \sim N(0, \sqrt{0.001})$$
$$X_4 = X_3 + \varepsilon_4, \quad \varepsilon_4 \sim N(0, \sqrt{0.1})$$
$$Y = X_4 + X_5 + X_4 \cdot X_5 + \varepsilon, \quad \varepsilon \sim N(0, \sqrt{0.1})$$

Feature Properties:

- X1, X3, X5: Independent uniform(0,1) distributions
- X2: Nearly perfect copy of X1 (correlation approximately 0.99)
- X4: Noisy copy of X3 (correlation approximately 0.94)
- Y depends on X4, X5, and their interaction

Value

A regression task (mlr3::TaskRegr) with data.table backend.

References

Ewald F, Bothmann L, Wright M, Bischl B, Casalicchio G, König G (2024). "A Guide to Feature Importance Methods for Scientific Inference." In Longo L, Lapuschkin S, Seifert C (eds.), *Explainable Artificial Intelligence*, 440–464. ISBN 978-3-031-63797-1, doi:10.1007/9783031637971_22.

See Also

Other simulation: `sim_dgp_scenarios`

Examples

```
sim_dgp_ewald(100)
```

---

sim_dgp_scenarios *Simulation DGPs for Feature Importance Method Comparison*

Description

These data generating processes (DGPs) are designed to illustrate specific strengths and weaknesses of different feature importance methods like PFI, CFI, and RFI. Each DGP focuses on one primary challenge to make the differences between methods clear.

Usage

```
sim_dgp_correlated(n = 500L, r = 0.9)

sim_dgp_mediated(n = 500L)

sim_dgp_confounded(n = 500L, hidden = TRUE)

sim_dgp_interactions(n = 500L)

sim_dgp_independent(n = 500L)
```

Arguments

n	(integer(1): 500L) Number of observations to generate.
r	(numeric(1): 0.9) Correlation between x1 and x2. Must be between -1 and 1.
hidden	(logical(1): TRUE) Whether to hide the confounder from the returned task. If FALSE, the confounder is included as a feature, allowing direct adjustment. If TRUE (default), only the proxy is available, simulating unmeasured confounding.

Details

Correlated Features DGP: This DGP creates highly correlated predictors where PFI will show artificially low importance due to redundancy, while CFI will correctly identify each feature's conditional contribution.

Mathematical Model:

$$(X_1, X_2)^T \sim \text{MVN}(0, \Sigma)$$

where Σ is a 2×2 covariance matrix with 1 on the diagonal and correlation r on the off-diagonal.

$$X_3 \sim N(0,1), \quad X_4 \sim N(0,1)$$

$$Y = 2 \cdot X_1 + X_3 + \varepsilon$$

where $\varepsilon \sim N(0, 0.2^2)$.

Feature Properties:

- x1: Standard normal from MVN, direct causal effect on y ($\beta = 2.0$)
- x2: Correlated with x1 (correlation = r), NO causal effect on y ($\beta = 0$)
- x3: Independent standard normal, direct causal effect on y ($\beta = 1.0$)
- x4: Independent standard normal, no effect on y ($\beta = 0$)

Expected Behavior:

- Will depend on the used learner and the strength of correlation (r)
- **Marginal methods** (PFI, Marginal SAGE): Should falsely assign importance to x2 due to correlation with x1
- **CFI** Should correctly assign near-zero importance to x2
- x2 is a "spurious predictor" - correlated with causal feature but not causal itself

Mediated Effects DGP: This DGP demonstrates the difference between total and direct causal effects. Some features affect the outcome only through mediators.

Mathematical Model:

$$\text{exposure} \sim N(0,1), \quad \text{direct} \sim N(0,1)$$

$$\text{mediator} = 0.8 \cdot \text{exposure} + 0.6 \cdot \text{direct} + \varepsilon_m$$

$$Y = 1.5 \cdot \text{mediator} + 0.5 \cdot \text{direct} + \varepsilon$$

where $\varepsilon_m \sim N(0, 0.3^2)$ and $\varepsilon \sim N(0, 0.2^2)$.

Feature Properties:

- `exposure`: Has no direct effect on y, only through mediator (total effect = 1.2)
- `mediator`: Mediates the effect of exposure on y
- `direct`: Has both direct effect on y and effect on mediator
- `noise`: No causal relationship to y

Causal Structure: exposure -> mediator -> y <- direct -> mediator

Confounding DGP: This DGP includes a confounder that affects both a feature and the outcome. Uses simple coefficients for easy interpretation.

Mathematical Model:

$$H \sim N(0, 1)$$

$$X_1 = H + \varepsilon_1$$

$$\text{proxy} = H + \varepsilon_p, \quad \text{independent} \sim N(0, 1)$$

$$Y = H + X_1 + \text{independent} + \varepsilon$$

where all $\varepsilon \sim N(0, 0.5^2)$ independently.

Model Structure:

- Confounder H ~ N(0,1) (potentially unobserved)
- x1 = H + noise (affected by confounder)
- proxy = H + noise (noisy measurement of confounder)
- independent ~ N(0,1) (truly independent)
- y = H + x1 + independent + noise

Expected Behavior:

- **PFI**: Will show inflated importance for x1 due to confounding
- **CFI**: Should partially account for confounding through conditional sampling and reduce its importance
- **RFI conditioning on proxy**: Should reduce confounding bias by conditioning on proxy

Interaction Effects DGP: This DGP demonstrates a pure interaction effect where features have no main effects.

Mathematical Model:

$$Y = 2 \cdot X_1 \cdot X_2 + X_3 + \varepsilon$$

where $X_j \sim N(0, 1)$ independently and $\varepsilon \sim N(0, 0.5^2)$.

Feature Properties:

- `x1`, `x2`: Independent features with ONLY interaction effect (no main effects)
- `x3`: Independent feature with main effect only
- `noise1`, `noise2`: No causal effects

Expected Behavior:

- Will depend on the used learner and its ability to model interactions

Independent Features DGP: This is a baseline scenario where all features are independent and their effects are additive. All importance methods should give similar results.

Mathematical Model:

$$Y = 2.0 \cdot X_1 + 1.0 \cdot X_2 + 0.5 \cdot X_3 + \varepsilon$$

where $X_j \sim N(0, 1)$ independently and $\varepsilon \sim N(0, 0.2^2)$.

Feature Properties:

- `important1-3`: Independent features with different effect sizes
- `unimportant1-2`: Independent noise features with no effect

Expected Behavior:

- **All methods**: Should rank features consistently by their true effect sizes
- **Ground truth**: important1 > important2 > important3 > unimportant1,2 (approximately 0)

Value

A regression task (mlr3::TaskRegr) with data.table backend.

Functions

- `sim_dgp_correlated()`: Correlated features demonstrating PFI's limitations
- `sim_dgp_mediated()`: Mediated effects showing direct vs total importance
- `sim_dgp_confounded()`: Confounding scenario for conditional sampling
- `sim_dgp_interactions()`: Interaction effects between features
- `sim_dgp_independent()`: Independent features baseline scenario

References

Ewald F, Bothmann L, Wright M, Bischl B, Casalicchio G, König G (2024). "A Guide to Feature Importance Methods for Scientific Inference." In Longo L, Lapuschkin S, Seifert C (eds.), *Explainable Artificial Intelligence*, 440–464. ISBN 978-3-031-63797-1, doi:10.1007/9783031637971_22.

See Also

Other simulation: `sim_dgp_ewald()`

Other simulation: `sim_dgp_ewald()`

Other simulation: `sim_dgp_ewald()`

Other simulation: `sim_dgp_ewald()`

Other simulation: `sim_dgp_ewald()`

Examples

```
task = sim_dgp_correlated(200)
task$data()

# With different correlation
task_high_cor = sim_dgp_correlated(200, r = 0.95)
cor(task_high_cor$data()$x1, task_high_cor$data()$x2)
task = sim_dgp_mediated(200)
task$data()
# Hidden confounder scenario (traditional)
task_hidden = sim_dgp_confounded(200, hidden = TRUE)
task_hidden$feature_names  # proxy available but not confounder

# Observable confounder scenario
task_observed = sim_dgp_confounded(200, hidden = FALSE)
task_observed$feature_names  # both confounder and proxy available
task = sim_dgp_interactions(200)
task$data()
task = sim_dgp_independent(200)
task$data()
```

---

wvim_design_matrix *Create Feature Selection Design Matrix*

Description

Creates a logical design matrix for leave-in or leave-out feature evaluation. Used internally with mlr3fselect to evaluate feature subsets.

Usage

```
wvim_design_matrix(
  all_features,
  feature_names = all_features,
  direction = c("leave-out", "leave-in")
)
```

Arguments

`all_features`	(character()) All available feature names from the task.
`feature_names`	(character() | list of character()) Features or feature groups to evaluate. Can be a vector for individual features or a named list for grouped features. Defaults to `all_features` if unspecified.
`direction`	(character(1)) Either `"leave-in"` or `"leave-out"` (default). Controls which features are selected in the design matrix. `"leave-out"` sets features of interest to `FALSE`, and `"leave-in"` analogously sets them to `TRUE`.

Value

data.table with logical columns for each feature in `all_features` and `length(feature_names)` rows, one for each entry in `feature_names`

Examples

```
task = mlr3::tsk("mtcars")

# Individual features
feature_names = task$feature_names[1:3]
wvim_design_matrix(task$feature_names, feature_names, "leave-in")
wvim_design_matrix(task$feature_names, feature_names, "leave-out")

# Feature groups
feature_groups = list(
  A = task$feature_names[1:2],
  B = task$feature_names[3:5]
)
wvim_design_matrix(task$feature_names, feature_groups, "leave-out")
```

---

xplain_opt *xplainfi Package Options*

Description

Get or set package-level options for xplainfi.

Usage

```
xplain_opt(...)
```

Arguments

`...`	Option names to retrieve (as character strings) or options to set (as named arguments). • To **get** an option: `xplain_opt("verbose")` returns the current value • To **set** an option: `xplain_opt(verbose = FALSE)` sets the value • To **get all** options: `xplain_opt()` returns a named list of all options

Details

Options can be set in three ways (in order of precedence):

1. Using `xplain_opt(option_name = value)` (recommended)
2. Using `options("xplain.option_name" = value)`
3. Using environment variables `XPLAIN_OPTION_NAME=value`

Available Options:

Option	Default	Description
`verbose`	`TRUE`	Show informational messages (e.g., when using default measure or resampling)
`progress`	`FALSE`	Show progress bars during computation
`sequential`	`FALSE`	Force sequential execution (disable parallelization)
`debug`	`FALSE`	Enable debug output for development and troubleshooting

Value

- When **getting** a single option: the option value (logical)
- When **getting** multiple options: a named list of option values
- When **setting** options: the previous values (invisibly)

Examples

```
# Get current value of an option
xplain_opt("verbose")

# Get all options
xplain_opt()

# Set an option (returns previous value invisibly)
old <- xplain_opt(verbose = FALSE)
xplain_opt("verbose")  # Now FALSE

# Restore previous value
xplain_opt(verbose = old$verbose)

# Temporary option change with withr
if (requireNamespace("withr", quietly = TRUE)) {
  withr::with_options(
    list("xplain.verbose" = FALSE),
    {
      # Code here runs with verbose = FALSE
      xplain_opt("verbose")
    }
  )
}
```

Index