

# Package ‘vtreat’

May 8, 2026

**Type** Package

**Title** A Statistically Sound 'data.frame' Processor/Conditioner

**Version** 1.6.5

**Date** 2024-06-12

**URL** <https://github.com/WinVector/vtreat/>,  
<https://winvector.github.io/vtreat/>

**BugReports** <https://github.com/WinVector/vtreat/issues>

**Maintainer** John Mount <jmount@win-vector.com>

**Description** A 'data.frame' processor/conditioner that prepares real-world data for predictive modeling in a statistically sound manner. 'vtreat' prepares variables so that data has fewer exceptional cases, making it easier to safely use models in production. Common problems 'vtreat' defends against: 'Inf', 'NA', too many categorical levels, rare categorical levels, and new categorical levels (levels seen during application, but not during training). Reference: ``'vtreat': a data.frame Processor for Predictive Modeling'', Zumel, Mount, 2016, <[DOI:10.5281/zenodo.1173313](https://doi.org/10.5281/zenodo.1173313)>.

**License** GPL-2 | GPL-3

**Depends** R (>= 3.4.0), wrapr (>= 2.1.0)

**Imports** stats, digest

**Suggests** rquery (>= 1.4.99), rqdatatable (>= 1.3.3), data.table (>= 1.12.2), knitr, rmarkdown, parallel, DBI, RSQLite, datasets, R.rsp, tinytest

**VignetteBuilder** knitr, R.rsp

**RoxygenNote** 7.3.1

**ByteCompile** true

**NeedsCompilation** no

**Author** John Mount [aut, cre],  
Nina Zumel [aut],  
Win-Vector LLC [cph]

**Repository** CRAN

**Date/Publication** 2024-06-12 16:40:02 UTC

## Contents

vtreat-package . . . . .	3
apply_transform . . . . .	4
as_rquery_plan . . . . .	4
BinomialOutcomeTreatment . . . . .	6
buildEvalSets . . . . .	7
center_scale . . . . .	9
classification_parameters . . . . .	9
designTreatmentsC . . . . .	10
designTreatmentsN . . . . .	12
designTreatmentsZ . . . . .	15
design_missingness_treatment . . . . .	16
fit . . . . .	18
fit_prepare . . . . .	18
fit_transform . . . . .	19
format.vtreatment . . . . .	20
getSplitPlanAppLabels . . . . .	20
get_feature_names . . . . .	21
get_score_frame . . . . .	21
get_transform . . . . .	22
kWayCrossValidation . . . . .	22
kWayStratifiedY . . . . .	23
kWayStratifiedYReplace . . . . .	24
makeCustomCoderCat . . . . .	24
makeCustomCoderNum . . . . .	25
makekWayCrossValidationGroupedByColumn . . . . .	26
mkCrossFrameCExperiment . . . . .	27
mkCrossFrameMExperiment . . . . .	30
mkCrossFrameNExperiment . . . . .	33
MultinomialOutcomeTreatment . . . . .	36
multinomial_parameters . . . . .	37
novel_value_summary . . . . .	37
NumericOutcomeTreatment . . . . .	38
oneWayHoldout . . . . .	39
patch_columns_into_frame . . . . .	40
prepare . . . . .	40
prepare.multinomial_plan . . . . .	41
prepare.simple_plan . . . . .	42
prepare.treatmentplan . . . . .	43
pre_comp_xval . . . . .	45
print.multinomial_plan . . . . .	46
print.simple_plan . . . . .	47
print.treatmentplan . . . . .	47
print.vtreatment . . . . .	48
problemAppPlan . . . . .	48
regression_parameters . . . . .	49
rquery_prepare . . . . .	49

solve_pieewise . . . . .	51
solve_pieewisec . . . . .	52
spline_variable . . . . .	52
spline_variablec . . . . .	53
square_window . . . . .	53
square_windowc . . . . .	54
track_values . . . . .	55
UnsupervisedTreatment . . . . .	56
unsupervised_parameters . . . . .	56
value_variables_C . . . . .	57
value_variables_N . . . . .	59
variable_values . . . . .	60
vnames . . . . .	61
vorig . . . . .	61
<b>Index</b>	<b>62</b>

---

vtreat-package	<i>vtreat: A Statistically Sound 'data.frame' Processor/Conditioner</i>
----------------	---

---

## Description

A 'data.frame' processor/conditioner that prepares real-world data for predictive modeling in a statistically sound manner. 'vtreat' prepares variables so that data has fewer exceptional cases, making it easier to safely use models in production. Common problems 'vtreat' defends against: 'Inf', 'NA', too many categorical levels, rare categorical levels, and new categorical levels (levels seen during application, but not during training). 'vtreat::prepare' should be used as you would use 'model.matrix'.

## Details

For more information:

- `vignette('vtreat', package='vtreat')`
- `vignette(package='vtreat')`
- Website: <https://github.com/WinVector/vtreat>

## Author(s)

**Maintainer:** John Mount <jmount@win-vector.com>

Authors:

- Nina Zumel <nzumel@win-vector.com>

Other contributors:

- Win-Vector LLC [copyright holder]

**See Also**

Useful links:

- <https://github.com/WinVector/vtreat/>
- <https://winvector.github.io/vtreat/>
- Report bugs at <https://github.com/WinVector/vtreat/issues>

---

apply_transform	<i>Transform second argument by first.</i>
-----------------	--

---

**Description**

Apply first argument to second as a transform.

**Usage**

```
apply_transform(vps, dframe, ..., parallelCluster = NULL)
```

**Arguments**

vps	vtreat pipe step, object defining transform.
dframe	data.frame, data to transform
...	not used, forces later arguments to bind by name.
parallelCluster	optional, parallel cluster to run on.

**Value**

transformed dframe

---

as_rquery_plan	<i>Convert vtreatment plans into a sequence of rquery operations.</i>
----------------	---

---

**Description**

Convert vtreatment plans into a sequence of rquery operations.

**Usage**

```
as_rquery_plan(treatmentplans, ..., var_restriction = NULL)
```

**Arguments**

treatmentplans vtreat treatment plan or list of vtreat treatment plan sharing same outcome and outcome type.

... not used, force any later arguments to bind to names.

var\_restriction character, if not null restrict to producing these variables.

**Value**

list(optree\_generator (ordered list of functions), temp\_tables (named list of tables))

**See Also**

[rquery\\_prepare](#)

**Examples**

```
if(requireNamespace("rquery", quietly = TRUE)) {
  dTrainC <- data.frame(x= c('a', 'a', 'a', 'b', NA, 'b'),
                      z= c(1, 2, NA, 4, 5, 6),
                      y= c(FALSE, FALSE, TRUE, FALSE, TRUE, TRUE),
                      stringsAsFactors = FALSE)
  dTrainC$id <- seq_len(nrow(dTrainC))
  treatmentsC <- designTreatmentsC(dTrainC, c("x", "z"), 'y', TRUE)
  print(prepare(treatmentsC, dTrainC))
  rqplan <- as_rquery_plan(list(treatmentsC))
  ops <- flatten_fn_list(rquery::local_td(dTrainC), rqplan$optree_generators)
  cat(format(ops))
  if(requireNamespace("rqdatatable", quietly = TRUE)) {
    treated <- rqdatatable::ex_data_table(ops, tables = rqplan$tables)
    print(treated[])
  }
  if(requireNamespace("DBI", quietly = TRUE) &&
      requireNamespace("RSQLite", quietly = TRUE)) {
    db <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
    source_data <- rquery::rq_copy_to(db, "dTrainC", dTrainC,
                                     overwrite = TRUE, temporary = TRUE)

    rest <- rquery_prepare(db, rqplan, source_data, "dTreatedC",
                          extracols = "id")
    resd <- DBI::dbReadTable(db, rest$table_name)
    print(resd)

    rquery::rq_remove_table(db, source_data$table_name)
    rquery::rq_remove_table(db, rest$table_name)
    DBI::dbDisconnect(db)
  }
}
```

---

BinomialOutcomeTreatment

*Stateful object for designing and applying binomial outcome treatments.*

---

## Description

Hold settings and results for binomial classification data preparation.

## Usage

```
BinomialOutcomeTreatment(
  ...,
  var_list,
  outcome_name,
  outcome_target = TRUE,
  cols_to_copy = NULL,
  params = NULL,
  imputation_map = NULL
)
```

## Arguments

...	not used, force arguments to be specified by name.
var_list	Names of columns to treat (effective variables).
outcome_name	Name of column holding outcome variable. <code>dframe[[outcomename]]</code> must be only finite and non-missing values.
outcome_target	Value/level of outcome to be considered "success", and there must be a cut such that <code>dframe[[outcomename]]==outcome_target</code> at least twice and <code>dframe[[outcomename]]!=outcome_target</code> at least twice.
cols_to_copy	list of extra columns to copy.
params	parameters list from <code>classification_parameters</code>
imputation_map	map from column names to functions of signature <code>f(values: numeric, weights: numeric)</code> , simple missing value imputers.

## Details

Please see [https://github.com/WinVector/vtreat/blob/main/Examples/fit\\_transform/fit\\_transform\\_api.md](https://github.com/WinVector/vtreat/blob/main/Examples/fit_transform/fit_transform_api.md), [mkCrossFrameCExperiment](#), [designTreatmentsC](#), and [prepare.treatmentplan](#) for details.

---

buildEvalSets	<i>Build set carve-up for out-of sample evaluation.</i>
---------------	---

---

### Description

Return a carve-up of `seq_len(nRows)`. Very useful for any sort of nested model situation (such as data prep, stacking, or super-learning).

### Usage

```
buildEvalSets(
  nRows,
  ...,
  dframe = NULL,
  y = NULL,
  splitFunction = NULL,
  nSplits = 3
)
```

### Arguments

<code>nRows</code>	scalar, $\geq 1$ number of rows to sample from.
<code>...</code>	no additional arguments, declared to forced named binding of later arguments.
<code>dframe</code>	(optional) original data.frame, passed to user <code>splitFunction</code> .
<code>y</code>	(optional) numeric vector, outcome variable (possibly to stratify on), passed to user <code>splitFunction</code> .
<code>splitFunction</code>	(optional) function taking arguments <code>nSplits</code> , <code>nRows</code> , <code>dframe</code> , and <code>y</code> ; returning a user desired split.
<code>nSplits</code>	integer, target number of splits.

### Details

Also sets attribute "splitmethod" on return value that describes how the split was performed. `attr(returnValue,'splitmethod')` is one of: 'notsplit' (data was not split; corner cases like single row data sets), 'oneway' (leave one out holdout), 'kwaycross' (a simple partition), 'userfunction' (user supplied function was actually used), or a user specified attribute. Any user desired properties (such as stratification on `y`, or preservation of groups designated by original data row numbers) may not apply unless you see that 'userfunction' has been used.

The intent is the user `splitFunction` only needs to handle "easy cases" and maintain user invariants. If the user `splitFunction` returns `NULL`, throws, or returns an unacceptable carve-up then `vtrout::buildEvalSets` returns its own eval set plan. The signature of `splitFunction` should be `splitFunction(nRows,nSplits,dframe,y)` where `nSplits` is the number of pieces we want in the carve-up, `nRows` is the number of rows to split, `dframe` is the original dataframe (useful for any group control variables), and `y` is a numeric vector representing outcome (useful for outcome stratification).

Note that `buildEvalSets` may not always return a partition (such as one row dataframes), or if the user split function chooses to make rows eligible for application a different number of times.

**Value**

list of lists where the app portion of the sub-lists is a disjoint carve-up of seq\_len(nRows) and each list as a train portion disjoint from app.

**See Also**

[kWayCrossValidation](#), [kWayStratifiedY](#), and [makekWayCrossValidationGroupedByColumn](#)

**Examples**

```
# use
buildEvalSets(200)

# longer example
# helper fns
# fit models using experiment plan to estimate out of sample behavior
fitModelAndApply <- function(trainData, applicaitonData) {
  model <- lm(y~x,data=trainData)
  predict(model,newdata=applicaitonData)
}
simulateOutOfSampleTrainEval <- function(d,fitApplyFn) {
  eSets <- buildEvalSets(nrow(d))
  evals <- lapply(eSets,
    function(ei) { fitApplyFn(d[ei$train,],d[ei$app,]) })
  pred <- numeric(nrow(d))
  for(eii in seq_len(length(eSets))) {
    pred[eSets[[eii]]$app] <- evals[[eii]]
  }
  pred
}

# run the experiment
set.seed(2352356)
# example data
d <- data.frame(x=rnorm(5),y=rnorm(5),
  outOfSampleEst=NA,inSampleEst=NA)

# fit model on all data
d$inSampleEst <- fitModelAndApply(d,d)
# compute in-sample R^2 (above zero, falsely shows a
# relation until we adjust for degrees of freedom)
1-sum((d$y-d$inSampleEst)^2)/sum((d$y-mean(d$y))^2)

d$outOfSampleEst <- simulateOutOfSampleTrainEval(d,fitModelAndApply)
# compute out-sample R^2 (not positive,
# evidence of no relation)
1-sum((d$y-d$outOfSampleEst)^2)/sum((d$y-mean(d$y))^2)
```

---

center_scale	<i>Center and scale a set of variables.</i>
--------------	---

---

**Description**

Center and scale a set of variables. Other columns are passed through.

**Usage**

```
center_scale(d, center, scale)
```

**Arguments**

d	data.frame to work with
center	named vector of variables to center
scale	named vector of variables to scale

**Value**

d with centered and scaled columns altered

**Examples**

```
d <- data.frame(x = 1:5,
               y = c('a', 'a', 'b', 'b', 'b'))
vars_to_transform = "x"
t <- base::scale(as.matrix(d[, vars_to_transform, drop = FALSE]),
                 center = TRUE, scale = TRUE)

t

centering <- attr(t, "scaled:center")
scaling <- attr(t, "scaled:scale")
center_scale(d, center = centering, scale = scaling)
```

---

classification_parameters	<i>vtreat classification parameters.</i>
---------------------------	--

---

**Description**

A list of settings and values for vtreat binomial classification fitting. Please see [https://github.com/WinVector/vtreat/blob/main/Examples/fit\\_transform/fit\\_transform\\_api.md](https://github.com/WinVector/vtreat/blob/main/Examples/fit_transform/fit_transform_api.md), [mkCrossFrameCEExperiment](#), [designTreatmentsC](#), and [prepare.treatmentplan](#) for details.

**Usage**

```
classification_parameters(user_params = NULL)
```

**Arguments**

user\_params     list of user overrides.

**Value**

filled out parameter list

---

designTreatmentsC     *Build all treatments for a data frame to predict a categorical outcome.*

---

**Description**

Function to design variable treatments for binary prediction of a categorical outcome. Data frame is assumed to have only atomic columns except for dates (which are converted to numeric). Note: re-encoding high cardinality categorical variables can introduce undesirable nested model bias, for such data consider using [mkCrossFrameCExperiment](#).

**Usage**

```
designTreatmentsC(  
  dframe,  
  varlist,  
  outcomename,  
  outcometarget = TRUE,  
  ...,  
  weights = c(),  
  minFraction = 0.02,  
  smFactor = 0,  
  rareCount = 0,  
  rareSig = NULL,  
  collarProb = 0,  
  codeRestriction = NULL,  
  customCoders = NULL,  
  splitFunction = NULL,  
  ncross = 3,  
  forceSplit = FALSE,  
  catScaling = TRUE,  
  verbose = TRUE,  
  parallelCluster = NULL,  
  use_parallel = TRUE,  
  missingness_imputation = NULL,  
  imputation_map = NULL  
)
```

**Arguments**

dframe	Data frame to learn treatments from (training data), must have at least 1 row.
varlist	Names of columns to treat (effective variables).
outcomename	Name of column holding outcome variable. dframe[[outcomename]] must be only finite non-missing values.
outcometarget	Value/level of outcome to be considered "success", and there must be a cut such that dframe[[outcomename]]==outcometarget at least twice and dframe[[outcomename]]!=outcometarget at least twice.
...	no additional arguments, declared to forced named binding of later arguments
weights	optional training weights for each row
minFraction	optional minimum frequency a categorical level must have to be converted to an indicator column.
smFactor	optional smoothing factor for impact coding models.
rareCount	optional integer, allow levels with this count or below to be pooled into a shared rare-level. Defaults to 0 or off.
rareSig	optional numeric, suppress levels from pooling at this significance value greater. Defaults to NULL or off.
collarProb	what fraction of the data (pseudo-probability) to collar data at if doCollar is set during <a href="#">prepare.treatmentplan</a> .
codeRestriction	what types of variables to produce (character array of level codes, NULL means no restriction).
customCoders	map from code names to custom categorical variable encoding functions (please see <a href="https://github.com/WinVector/vtreat/blob/main/extras/CustomLevelCoders.md">https://github.com/WinVector/vtreat/blob/main/extras/CustomLevelCoders.md</a> ).
splitFunction	(optional) see vtreat::buildEvalSets .
ncross	optional scalar >=2 number of cross validation splits use in rescoring complex variables.
forceSplit	logical, if TRUE force cross-validated significance calculations on all variables.
catScaling	optional, if TRUE use glm() linkspace, if FALSE use lm() for scaling.
verbose	if TRUE print progress.
parallelCluster	(optional) a cluster object created by package parallel or package snow.
use_parallel	logical, if TRUE use parallel methods (when parallel cluster is set).
missingness_imputation	function of signature f(values: numeric, weights: numeric), simple missing value imputer.
imputation_map	map from column names to functions of signature f(values: numeric, weights: numeric), simple missing value imputers.

**Details**

The main fields are mostly vectors with names (all with the same names in the same order):

- vars : (character array without names) names of variables (in same order as names on the other diagnostic vectors) - varMoves : logical TRUE if the variable varied during hold out scoring, only variables that move will be in the treated frame - #' - sig : an estimate significance of effect

See the vtreat vignette for a bit more detail and a worked example.

Columns that do not vary are not passed through.

Note: re-encoding high cardinality on training data can introduce nested model bias, consider using `mkCrossFrameCExperiment` instead.

**Value**

treatment plan (for use with `prepare`)

**See Also**

[prepare.treatmentplan](#), [designTreatmentsN](#), [designTreatmentsZ](#), [mkCrossFrameCExperiment](#)

**Examples**

```
dTrainC <- data.frame(x=c('a','a','a','b','b','b'),
  z=c(1,2,3,4,5,6),
  y=c(FALSE,FALSE,TRUE,FALSE,TRUE,TRUE))
dTestC <- data.frame(x=c('a','b','c',NA),
  z=c(10,20,30,NA))
treatmentsC <- designTreatmentsC(dTrainC,colnames(dTrainC),'y',TRUE)
dTestCTreated <- prepare(treatmentsC,dTestC,pruneSig=0.99)
```

---

designTreatmentsN

*build all treatments for a data frame to predict a numeric outcome*

---

**Description**

Function to design variable treatments for binary prediction of a numeric outcome. Data frame is assumed to have only atomic columns except for dates (which are converted to numeric). Note: each column is processed independently of all others. Note: re-encoding high cardinality on training data categorical variables can introduce undesirable nested model bias, for such data consider using [mkCrossFrameNExperiment](#).

**Usage**

```
designTreatmentsN(
  dframe,
  varlist,
  outcomename,
```

```

...,
weights = c(),
minFraction = 0.02,
smFactor = 0,
rareCount = 0,
rareSig = NULL,
collarProb = 0,
codeRestriction = NULL,
customCoders = NULL,
splitFunction = NULL,
ncross = 3,
forceSplit = FALSE,
verbose = TRUE,
parallelCluster = NULL,
use_parallel = TRUE,
missingness_imputation = NULL,
imputation_map = NULL
)

```

### Arguments

dframe	Data frame to learn treatments from (training data), must have at least 1 row.
varlist	Names of columns to treat (effective variables).
outcomename	Name of column holding outcome variable. <code>dframe[[outcomename]]</code> must be only finite non-missing values and there must be a cut such that <code>dframe[[outcomename]]</code> is both above the cut at least twice and below the cut at least twice.
...	no additional arguments, declared to forced named binding of later arguments
weights	optional training weights for each row
minFraction	optional minimum frequency a categorical level must have to be converted to an indicator column.
smFactor	optional smoothing factor for impact coding models.
rareCount	optional integer, allow levels with this count or below to be pooled into a shared rare-level. Defaults to 0 or off.
rareSig	optional numeric, suppress levels from pooling at this significance value greater. Defaults to NULL or off.
collarProb	what fraction of the data (pseudo-probability) to collar data at if <code>doCollar</code> is set during <code>prepare.treatmentplan</code> .
codeRestriction	what types of variables to produce (character array of level codes, NULL means no restriction).
customCoders	map from code names to custom categorical variable encoding functions (please see <a href="https://github.com/WinVector/vtreat/blob/main/extras/CustomLevelCoders.md">https://github.com/WinVector/vtreat/blob/main/extras/CustomLevelCoders.md</a> ).
splitFunction	(optional) see <code>vtreat::buildEvalSets</code> .

ncross	optional scalar $\geq 2$ number of cross validation splits use in rescoring complex variables.
forceSplit	logical, if TRUE force cross-validated significance calculations on all variables.
verbose	if TRUE print progress.
parallelCluster	(optional) a cluster object created by package parallel or package snow.
use_parallel	logical, if TRUE use parallel methods (when parallel cluster is set).
missingness_imputation	function of signature $f(\text{values: numeric, weights: numeric})$ , simple missing value imputer.
imputation_map	map from column names to functions of signature $f(\text{values: numeric, weights: numeric})$ , simple missing value imputers.

### Details

The main fields are mostly vectors with names (all with the same names in the same order):

- vars : (character array without names) names of variables (in same order as names on the other diagnostic vectors) - varMoves : logical TRUE if the variable varied during hold out scoring, only variables that move will be in the treated frame - sig : an estimate significance of effect

See the vtreat vignette for a bit more detail and a worked example.

Columns that do not vary are not passed through.

### Value

treatment plan (for use with prepare)

### See Also

[prepare.treatmentplan](#), [designTreatmentsC](#), [designTreatmentsZ](#), [mkCrossFrameNExperiment](#)

### Examples

```
dTrainN <- data.frame(x=c('a', 'a', 'a', 'a', 'b', 'b', 'b'),
  z=c(1,2,3,4,5,6,7),y=c(0,0,0,1,0,1,1))
dTestN <- data.frame(x=c('a', 'b', 'c', NA),
  z=c(10,20,30,NA))
treatmentsN = designTreatmentsN(dTrainN,colnames(dTrainN),'y')
dTestNTreated <- prepare(treatmentsN,dTestN,pruneSig=0.99)
```

---

designTreatmentsZ      *Design variable treatments with no outcome variable.*

---

### Description

Data frame is assumed to have only atomic columns except for dates (which are converted to numeric). Note: each column is processed independently of all others.

### Usage

```
designTreatmentsZ(
  dframe,
  varlist,
  ...,
  minFraction = 0,
  weights = c(),
  rareCount = 0,
  collarProb = 0,
  codeRestriction = NULL,
  customCoders = NULL,
  verbose = TRUE,
  parallelCluster = NULL,
  use_parallel = TRUE,
  missingness_imputation = NULL,
  imputation_map = NULL
)
```

### Arguments

dframe	Data frame to learn treatments from (training data), must have at least 1 row.
varlist	Names of columns to treat (effective variables).
...	no additional arguments, declared to forced named binding of later arguments
minFraction	optional minimum frequency a categorical level must have to be converted to an indicator column.
weights	optional training weights for each row
rareCount	optional integer, allow levels with this count or below to be pooled into a shared rare-level. Defaults to 0 or off.
collarProb	what fraction of the data (pseudo-probability) to collar data at if doCollar is set during <a href="#">prepare.treatmentplan</a> .
codeRestriction	what types of variables to produce (character array of level codes, NULL means no restriction).
customCoders	map from code names to custom categorical variable encoding functions (please see <a href="https://github.com/WinVector/vtreat/blob/main/extras/CustomLevelCoders.md">https://github.com/WinVector/vtreat/blob/main/extras/CustomLevelCoders.md</a> ).

**verbose** if TRUE print progress.  
**parallelCluster** (optional) a cluster object created by package parallel or package snow.  
**use\_parallel** logical, if TRUE use parallel methods (if parallel cluster is set).  
**missingness\_imputation** function of signature f(values: numeric, weights: numeric), simple missing value imputer.  
**imputation\_map** map from column names to functions of signature f(values: numeric, weights: numeric), simple missing value imputers.

### Details

The main fields are mostly vectors with names (all with the same names in the same order):

- vars : (character array without names) names of variables (in same order as names on the other diagnostic vectors) - varMoves : logical TRUE if the variable varied during hold out scoring, only variables that move will be in the treated frame

See the vtreat vignette for a bit more detail and a worked example.

Columns that do not vary are not passed through.

### Value

treatment plan (for use with prepare)

### See Also

[prepare.treatmentplan](#), [designTreatmentsC](#), [designTreatmentsN](#)

### Examples

```

dTrainZ <- data.frame(x=c('a','a','a','a','b','b',NA,'e','e'),
  z=c(1,2,3,4,5,6,7,NA,9))
dTestZ <- data.frame(x=c('a','x','c',NA),
  z=c(10,20,30,NA))
treatmentsZ = designTreatmentsZ(dTrainZ, colnames(dTrainZ),
  rareCount=0)
dTrainZTreated <- prepare(treatmentsZ, dTrainZ)
dTestZTreated <- prepare(treatmentsZ, dTestZ)
  
```

---

design\_missingness\_treatment

*Design a simple treatment plan to indicate missingness and perform simple imputation.*

---

### Description

Design a simple treatment plan to indicate missingness and perform simple imputation.

**Usage**

```
design_missingness_treatment(
  dframe,
  ...,
  varlist = colnames(dframe),
  invalid_mark = "_invalid_",
  drop_constant_columns = FALSE,
  missingness_imputation = NULL,
  imputation_map = NULL
)
```

**Arguments**

`dframe` data.frame to drive design.

`...` not used, forces later arguments to bind by name.

`varlist` character, names of columns to process.

`invalid_mark` character, name to use for NA levels and novel levels.

`drop_constant_columns` logical, if TRUE drop columns that do not vary from the treatment plan.

`missingness_imputation` function of signature `f(values: numeric)`, simple missing value imputer.

`imputation_map` map from column names to functions of signature `f(values: numeric)`, simple missing value imputers.

**Value**

simple treatment plan.

**See Also**

[prepare.simple\\_plan](#)

**Examples**

```
d <- wrapr::build_frame(
  "x1", "x2", "x3" |
  1 , 4 , "A" |
  NA , 5 , "B" |
  3 , 6 , NA )

plan <- design_missingness_treatment(d)
prepare(plan, d)

prepare(plan, data.frame(x1=NA, x2=NA, x3="E"))
```

---

fit	<i>Fit first arguemnt to data in second argument.</i>
-----	---

---

### Description

Update the state of first argument to have learned or fit from second argument.

### Usage

```
fit(vps, dframe, ..., weights = NULL, parallelCluster = NULL)
```

### Arguments

vps	vtreat pipe step, object specifying fit
dframe	data.frame, data to fit from.
...	not used, forces later arguments to bind by name.
weights	optional, per-dframe data weights.
parallelCluster	optional, parallel cluster to run on.

### Details

Note: input vps is not altered, fit is in returned value.

### Value

new fit object

---

fit_prepare	<i>Fit and prepare in a cross-validated manner.</i>
-------------	---

---

### Description

Update the state of first argument to have learned or fit from second argument, and compute a cross validated example of such a transform.

### Usage

```
fit_prepare(vps, dframe, ..., weights = NULL, parallelCluster = NULL)
```

**Arguments**

vps	vtreat pipe step, object specifying fit.
dframe	data.frame, data to fit from.
...	not used, forces later arguments to bind by name.
weights	optional, per-dframe data weights.
parallelCluster	optional, parallel cluster to run on.

**Details**

Note: input vps is not altered, fit is in returned list.

**Value**

@return named list containing: treatments and cross\_frame

---

fit_transform	<i>Fit and transform in a cross-validated manner.</i>
---------------	---

---

**Description**

Update the state of first argument to have learned or fit from second argument, and compute a cross validated example of such a transform.

**Usage**

```
fit_transform(vps, dframe, ..., weights = NULL, parallelCluster = NULL)
```

**Arguments**

vps	vtreat pipe step, object specifying fit.
dframe	data.frame, data to fit from.
...	not used, forces later arguments to bind by name.
weights	optional, per-dframe data weights.
parallelCluster	optional, parallel cluster to run on.

**Details**

Note: input vps is not altered, fit is in returned list.

**Value**

@return named list containing: treatments and cross\_frame

---

<code>format.vtreatment</code>	<i>Display treatment plan.</i>
--------------------------------	--------------------------------

---

**Description**

Display treatment plan.

**Usage**

```
## S3 method for class 'vtreatment'  
format(x, ...)
```

**Arguments**

<code>x</code>	treatment plan
<code>...</code>	additional args (to match general signature).

---

<code>getSplitPlanAppLabels</code>	<i>read application labels off a split plan.</i>
------------------------------------	--

---

**Description**

read application labels off a split plan.

**Usage**

```
getSplitPlanAppLabels(nRow, plan)
```

**Arguments**

<code>nRow</code>	number of rows in original data.frame.
<code>plan</code>	split plan

**Value**

vector of labels

**See Also**

[kWayCrossValidation](#), [kWayStratifiedY](#), and [makekWayCrossValidationGroupedByColumn](#)

**Examples**

```
plan <- kWayStratifiedY(3,2,NULL,NULL)  
getSplitPlanAppLabels(3,plan)
```

---

*get\_feature\_names*      *Return feasible feature names.*

---

**Description**

Return previously fit feature names.

**Usage**

`get_feature_names(vps)`

**Arguments**

`vps`                  vtreat pipe step, mutable object to read from.

**Value**

feature names

---

*get\_score\_frame*      *Return score frame from vps.*

---

**Description**

Return previously fit score frame.

**Usage**

`get_score_frame(vps)`

**Arguments**

`vps`                  vtreat pipe step, mutable object to read from.

**Value**

score frame

---

get_transform	<i>Return underlying transform from vps.</i>
---------------	--

---

**Description**

Return previously fit transform.

**Usage**

```
get_transform(vps)
```

**Arguments**

vps	vtreat pipe step, mutable object to read from.
-----	--

**Value**

transform

---

kWayCrossValidation	<i>k-fold cross validation, a splitFunction in the sense of vtreat::buildEvalSets</i>
---------------------	---

---

**Description**

k-fold cross validation, a splitFunction in the sense of vtreat::buildEvalSets

**Usage**

```
kWayCrossValidation(nRows, nSplits, dframe, y)
```

**Arguments**

nRows	number of rows to split (>1).
nSplits	number of groups to split into (>1,<=nRows).
dframe	original data frame (ignored).
y	numeric outcome variable (ignored).

**Value**

split plan

**Examples**

```
kWayCrossValidation(7,2,NULL,NULL)
```

---

kWayStratifiedY	<i>k-fold cross validation stratified on y, a splitFunction in the sense of vtreat::buildEvalSets</i>
-----------------	---

---

**Description**

k-fold cross validation stratified on y, a splitFunction in the sense of vtreat::buildEvalSets

**Usage**

```
kWayStratifiedY(nRows, nSplits, dframe, y)
```

**Arguments**

nRows	number of rows to split (>1)
nSplits	number of groups to split into (<nRows,>1).
dframe	original data frame (ignored).
y	numeric outcome variable try to have equidistributed in each split.

**Value**

split plan

**Examples**

```
set.seed(23255)
d <- data.frame(y=sin(1:100))
pStrat <- kWayStratifiedY(nrow(d),5,d,d$y)
problemAppPlan(nrow(d),5,pStrat,TRUE)
d$stratGroup <- vtreat::getSplitPlanAppLabels(nrow(d),pStrat)
pSimple <- kWayCrossValidation(nrow(d),5,d,d$y)
problemAppPlan(nrow(d),5,pSimple,TRUE)
d$simpleGroup <- vtreat::getSplitPlanAppLabels(nrow(d),pSimple)
summary(tapply(d$y,d$simpleGroup,mean))
summary(tapply(d$y,d$stratGroup,mean))
```

---

kWayStratifiedYReplace

*k-fold cross validation stratified with replacement on y, a splitFunction in the sense of vtreat::buildEvalSets .*

---

### Description

Build a k-fold cross validation sample where training sets are the same size as the original data, and built by sampling disjoint from test/application sets (sampled with replacement).

### Usage

```
kWayStratifiedYReplace(nRows, nSplits, dframe, y)
```

### Arguments

nRows	number of rows to split (>1)
nSplits	number of groups to split into (<nRows,>1).
dframe	original data frame (ignored).
y	numeric outcome variable try to have equidistributed in each split.

### Value

split plan

### Examples

```
set.seed(23255)
d <- data.frame(y=sin(1:100))
pStrat <- kWayStratifiedYReplace(nrow(d),5,d,d$y)
```

---

makeCustomCoderCat     *Make a categorical input custom coder.*

---

### Description

Make a categorical input custom coder.

**Usage**

```

makeCustomCoderCat(
  ...,
  customCode,
  coder,
  codeSeq,
  v,
  vcolin,
  zoY,
  zC,
  zTarget,
  weights = NULL,
  catScaling = FALSE
)

```

**Arguments**

...	not used, force arguments to be set by name
customCode	code name
coder	user supplied variable re-coder (see vignette for type signature)
codeSeq	arguments to custom coder
v	variable name
vcolin	data column, character
zoY	outcome column as numeric
zC	if classification outcome column as character
zTarget	if classification target class
weights	per-row weights
catScaling	optional, if TRUE use glm() linkspace, if FALSE use lm() for scaling.

**Value**

wrapped custom coder

---

makeCustomCoderNum     *Make a numeric input custom coder.*

---

**Description**

Make a numeric input custom coder.

**Usage**

```
makeCustomCoderNum(
  ...,
  customCode,
  coder,
  codeSeq,
  v,
  vcolin,
  zoY,
  zC,
  zTarget,
  weights = NULL,
  catScaling = FALSE
)
```

**Arguments**

...	not used, force arguments to be set by name
customCode	code name
coder	user supplied variable re-coder (see vignette for type signature)
codeSeq	arguments to custom coder
v	variable name
vcolin	data column, numeric
zoY	outcome column as numeric
zC	if classification outcome column as character
zTarget	if classification target class
weights	per-row weights
catScaling	optional, if TRUE use glm() linkspace, if FALSE use lm() for scaling.

**Value**

wrapped custom coder

---

makekWayCrossValidationGroupedByColumn

*Build a k-fold cross validation splitter, respecting (never splitting) groupingColumn.*

---

**Description**

Build a k-fold cross validation splitter, respecting (never splitting) groupingColumn.

**Usage**

```
makekWayCrossValidationGroupedByColumn(groupingColumnName)
```

**Arguments**

groupingColumnName  
name of column to group by.

**Value**

splitting function in the sense of vtreat::buildEvalSets.

**Examples**

```
d <- data.frame(y=sin(1:100))
d$group <- floor(seq_len(nrow(d))/5)
splitter <- makekWayCrossValidationGroupedByColumn('group')
split <- splitter(nrow(d),5,d,d$y)
d$splitLabel <- vtreat::getSplitPlanAppLabels(nrow(d),split)
rowSums(table(d$group,d$splitLabel)>0)
```

---

mkCrossFrameCExperiment

*Run categorical cross-frame experiment.*

---

**Description**

Builds a [designTreatmentsC](#) treatment plan and a data frame prepared from dframe that is "cross" in the sense each row is treated using a treatment plan built from a subset of dframe disjoint from the given row. The goal is to try to and supply a method of breaking nested model bias other than splitting into calibration, training, test sets.

**Usage**

```
mkCrossFrameCExperiment(  
  dframe,  
  varlist,  
  outcomename,  
  outcometarget,  
  ...,  
  weights = c(),  
  minFraction = 0.02,  
  smFactor = 0,  
  rareCount = 0,  
  rareSig = 1,  
  collarProb = 0,  
  codeRestriction = NULL,  
  customCoders = NULL,  
  scale = FALSE,  
  doCollar = FALSE,
```

```

splitFunction = NULL,
ncross = 3,
forceSplit = FALSE,
catScaling = TRUE,
verbose = TRUE,
parallelCluster = NULL,
use_parallel = TRUE,
missingness_imputation = NULL,
imputation_map = NULL
)

```

### Arguments

dframe	Data frame to learn treatments from (training data), must have at least 1 row.
varlist	Names of columns to treat (effective variables).
outcomename	Name of column holding outcome variable. dframe[[outcomename]] must be only finite non-missing values.
outcometarget	Value/level of outcome to be considered "success", and there must be a cut such that dframe[[outcomename]]==outcometarget at least twice and dframe[[outcomename]]!=outcometarget at least twice.
...	no additional arguments, declared to forced named binding of later arguments
weights	optional training weights for each row
minFraction	optional minimum frequency a categorical level must have to be converted to an indicator column.
smFactor	optional smoothing factor for impact coding models.
rareCount	optional integer, allow levels with this count or below to be pooled into a shared rare-level. Defaults to 0 or off.
rareSig	optional numeric, suppress levels from pooling at this significance value greater. Defaults to NULL or off.
collarProb	what fraction of the data (pseudo-probability) to collar data at if doCollar is set during <code>prepare.treatmentplan</code> .
codeRestriction	what types of variables to produce (character array of level codes, NULL means no restriction).
customCoders	map from code names to custom categorical variable encoding functions (please see <a href="https://github.com/WinVector/vtreat/blob/main/extras/CustomLevelCoders.md">https://github.com/WinVector/vtreat/blob/main/extras/CustomLevelCoders.md</a> ).
scale	optional if TRUE replace numeric variables with regression ("move to outcome-scale").
doCollar	optional if TRUE collar numeric variables by cutting off after a tail-probability specified by collarProb during treatment design.
splitFunction	(optional) see vtreat::buildEvalSets .
ncross	optional scalar >=2 number of cross-validation rounds to design.
forceSplit	logical, if TRUE force cross-validated significance calculations on all variables.

catScaling      optional, if TRUE use glm() linkspace, if FALSE use lm() for scaling.  
 verbose        if TRUE print progress.  
 parallelCluster  
                  (optional) a cluster object created by package parallel or package snow.  
 use\_parallel   logical, if TRUE use parallel methods.  
 missingness\_imputation  
                  function of signature f(values: numeric, weights: numeric), simple missing  
                  value imputer.  
 imputation\_map map from column names to functions of signature f(values: numeric, weights:  
                  numeric), simple missing value imputers.

### Value

named list containing: treatments, crossFrame, crossWeights, method, and evalSets

### See Also

[designTreatmentsC](#), [designTreatmentsN](#), [prepare.treatmentplan](#)

### Examples

```

# categorical example
set.seed(23525)

# we set up our raw training and application data
dTrainC <- data.frame(
  x = c('a', 'a', 'a', 'b', 'b', NA, NA),
  z = c(1, 2, 3, 4, NA, 6, NA),
  y = c(FALSE, FALSE, TRUE, FALSE, TRUE, TRUE, TRUE))

dTestC <- data.frame(
  x = c('a', 'b', 'c', NA),
  z = c(10, 20, 30, NA))

# we perform a vtreat cross frame experiment
# and unpack the results into treatmentsC
# and dTrainCTreated
unpack[
  treatmentsC = treatments,
  dTrainCTreated = crossFrame
] <- mkCrossFrameCExperiment(
  dframe = dTrainC,
  varlist = setdiff(colnames(dTrainC), 'y'),
  outcomename = 'y',
  outcometarget = TRUE,
  verbose = FALSE)

# the treatments include a score frame relating new
# derived variables to original columns
treatmentsC$scoreFrame[, c('origName', 'varName', 'code', 'rsq', 'sig', 'extraModelDegrees')] %>%

```

```

print(.)

# the treated frame is a "cross frame" which
# is a transform of the training data built
# as if the treatment were learned on a different
# disjoint training set to avoid nested model
# bias and over-fit.
dTrainCTreated %.>%
  head(.) %.>%
  print(.)

# Any future application data is prepared with
# the prepare method.
dTestCTreated <- prepare(treatmentsC, dTestC, pruneSig=NULL)

dTestCTreated %.>%
  head(.) %.>%
  print(.)

```

---

mkCrossFrameMExperiment

*Function to build multi-outcome vtreat cross frame and treatment plan.*

---

## Description

Please see vignette("MultiClassVtreat", package = "vtreat") <https://winvector.github.io/vtreat/articles/MultiClassVtreat.html>.

## Usage

```

mkCrossFrameMExperiment(
  dframe,
  varlist,
  outcomename,
  ...,
  weights = c(),
  minFraction = 0.02,
  smFactor = 0,
  rareCount = 0,
  rareSig = 1,
  collarProb = 0,
  codeRestriction = NULL,
  customCoders = NULL,
  scale = FALSE,
  doCollar = FALSE,
  splitFunction = vtreat::kWayCrossValidation,
  ncross = 3,

```

```

    forceSplit = FALSE,
    catScaling = FALSE,
    y_dependent_treatments = c("catB"),
    verbose = FALSE,
    parallelCluster = NULL,
    use_parallel = TRUE,
    missingness_imputation = NULL,
    imputation_map = NULL
  )

```

### Arguments

dframe	data to learn from
varlist	character, vector of independent variable column names.
outcomename	character, name of outcome column.
...	not used, declared to forced named binding of later arguments
weights	optional training weights for each row
minFraction	optional minimum frequency a categorical level must have to be converted to an indicator column.
smFactor	optional smoothing factor for impact coding models.
rareCount	optional integer, allow levels with this count or below to be pooled into a shared rare-level. Defaults to 0 or off.
rareSig	optional numeric, suppress levels from pooling at this significance value greater. Defaults to NULL or off.
collarProb	what fraction of the data (pseudo-probability) to collar data at if doCollar is set during <code>prepare.multinomial_plan</code> .
codeRestriction	what types of variables to produce (character array of level codes, NULL means no restriction).
customCoders	map from code names to custom categorical variable encoding functions (please see <a href="https://github.com/WinVector/vtreat/blob/main/extras/CustomLevelCoders.md">https://github.com/WinVector/vtreat/blob/main/extras/CustomLevelCoders.md</a> ).
scale	optional if TRUE replace numeric variables with regression ("move to outcome-scale").
doCollar	optional if TRUE collar numeric variables by cutting off after a tail-probability specified by collarProb during treatment design.
splitFunction	(optional) see <code>vtreat::buildEvalSets</code> .
ncross	optional scalar $\geq 2$ number of cross-validation rounds to design.
forceSplit	logical, if TRUE force cross-validated significance calculations on all variables.
catScaling	optional, if TRUE use <code>glm()</code> linkspace, if FALSE use <code>lm()</code> for scaling.
y_dependent_treatments	character what treatment types to build per-outcome level.
verbose	if TRUE print progress.

parallelCluster (optional) a cluster object created by package parallel or package snow.

use\_parallel logical, if TRUE use parallel methods.

missingness\_imputation function of signature f(values: numeric, weights: numeric), simple missing value imputer.

imputation\_map map from column names to functions of signature f(values: numeric, weights: numeric), simple missing value imputers.

**Value**

a names list containing cross\_frame, treat\_m, score\_frame, and fit\_obj\_id

**See Also**

[prepare.multinomial\\_plan](#)

**Examples**

```
# numeric example
set.seed(23525)

# we set up our raw training and application data
dTrainM <- data.frame(
  x = c('a', 'a', 'a', 'a', 'b', 'b', NA, NA),
  z = c(1, 2, 3, 4, 5, NA, 7, NA),
  y = c(0, 0, 0, 1, 0, 1, 2, 1))

dTestM <- data.frame(
  x = c('a', 'b', 'c', NA),
  z = c(10, 20, 30, NA))

# we perform a vtreat cross frame experiment
# and unpack the results into treatmentsM,
# dTrainMTreated, and score_frame
unpack[
  treatmentsM = treat_m,
  dTrainMTreated = cross_frame,
  score_frame = score_frame
] <- mkCrossFrameMExperiment(
  dframe = dTrainM,
  varlist = setdiff(colnames(dTrainM), 'y'),
  outcomename = 'y',
  verbose = FALSE)

# the score_frame relates new
# derived variables to original columns
score_frame[, c('origName', 'varName', 'code', 'rsq', 'sig', 'outcome_level')] %>%
  print(.)

# the treated frame is a "cross frame" which
```

```

# is a transform of the training data built
# as if the treatment were learned on a different
# disjoint training set to avoid nested model
# bias and over-fit.
dTrainMTreated %.>%
  head(.) %.>%
  print(.)

# Any future application data is prepared with
# the prepare method.
dTestMTreated <- prepare(treatmentsM, dTestM, pruneSig=NULL)

dTestMTreated %.>%
  head(.) %.>%
  print(.)

```

---

mkCrossFrameNExperiment

*Run a numeric cross frame experiment.*

---

## Description

Builds a [designTreatmentsN](#) treatment plan and a data frame prepared from `dframe` that is "cross" in the sense each row is treated using a treatment plan built from a subset of `dframe` disjoint from the given row. The goal is to try to and supply a method of breaking nested model bias other than splitting into calibration, training, test sets.

## Usage

```

mkCrossFrameNExperiment(
  dframe,
  varlist,
  outcomename,
  ...,
  weights = c(),
  minFraction = 0.02,
  smFactor = 0,
  rareCount = 0,
  rareSig = 1,
  collarProb = 0,
  codeRestriction = NULL,
  customCoders = NULL,
  scale = FALSE,
  doCollar = FALSE,
  splitFunction = NULL,
  ncross = 3,
  forceSplit = FALSE,

```

```

    verbose = TRUE,
    parallelCluster = NULL,
    use_parallel = TRUE,
    missingness_imputation = NULL,
    imputation_map = NULL
  )

```

## Arguments

<code>dframe</code>	Data frame to learn treatments from (training data), must have at least 1 row.
<code>varlist</code>	Names of columns to treat (effective variables).
<code>outcomename</code>	Name of column holding outcome variable. <code>dframe[[outcomename]]</code> must be only finite non-missing values and there must be a cut such that <code>dframe[[outcomename]]</code> is both above the cut at least twice and below the cut at least twice.
<code>...</code>	no additional arguments, declared to forced named binding of later arguments
<code>weights</code>	optional training weights for each row
<code>minFraction</code>	optional minimum frequency a categorical level must have to be converted to an indicator column.
<code>smFactor</code>	optional smoothing factor for impact coding models.
<code>rareCount</code>	optional integer, allow levels with this count or below to be pooled into a shared rare-level. Defaults to 0 or off.
<code>rareSig</code>	optional numeric, suppress levels from pooling at this significance value greater. Defaults to NULL or off.
<code>collarProb</code>	what fraction of the data (pseudo-probability) to collar data at if <code>doCollar</code> is set during <code>prepare.treatmentplan</code> .
<code>codeRestriction</code>	what types of variables to produce (character array of level codes, NULL means no restriction).
<code>customCoders</code>	map from code names to custom categorical variable encoding functions (please see <a href="https://github.com/WinVector/vtreat/blob/main/extras/CustomLevelCoders.md">https://github.com/WinVector/vtreat/blob/main/extras/CustomLevelCoders.md</a> ).
<code>scale</code>	optional if TRUE replace numeric variables with regression ("move to outcome-scale").
<code>doCollar</code>	optional if TRUE collar numeric variables by cutting off after a tail-probability specified by <code>collarProb</code> during treatment design.
<code>splitFunction</code>	(optional) see <code>vtreat::buildEvalSets</code> .
<code>ncross</code>	optional scalar $\geq 2$ number of cross-validation rounds to design.
<code>forceSplit</code>	logical, if TRUE force cross-validated significance calculations on all variables.
<code>verbose</code>	if TRUE print progress.
<code>parallelCluster</code>	(optional) a cluster object created by package <code>parallel</code> or package <code>snow</code> .
<code>use_parallel</code>	logical, if TRUE use parallel methods.

missingness\_imputation  
 function of signature f(values: numeric, weights: numeric), simple missing value imputer.

imputation\_map map from column names to functions of signature f(values: numeric, weights: numeric), simple missing value imputers.

**Value**

named list containing: treatments, crossFrame, crossWeights, method, and evalSets

**See Also**

[designTreatmentsC](#), [designTreatmentsN](#), [prepare.treatmentplan](#)

**Examples**

```
# numeric example
set.seed(23525)

# we set up our raw training and application data
dTrainN <- data.frame(
  x = c('a', 'a', 'a', 'a', 'b', 'b', NA, NA),
  z = c(1, 2, 3, 4, 5, NA, 7, NA),
  y = c(0, 0, 0, 1, 0, 1, 1, 1))

dTestN <- data.frame(
  x = c('a', 'b', 'c', NA),
  z = c(10, 20, 30, NA))

# we perform a vtreat cross frame experiment
# and unpack the results into treatmentsN
# and dTrainNTreated
unpack[
  treatmentsN = treatments,
  dTrainNTreated = crossFrame
] <- mkCrossFrameNExperiment(
  dframe = dTrainN,
  varlist = setdiff(colnames(dTrainN), 'y'),
  outcomename = 'y',
  verbose = FALSE)

# the treatments include a score frame relating new
# derived variables to original columns
treatmentsN$scoreFrame[, c('origName', 'varName', 'code', 'rsq', 'sig', 'extraModelDegrees')] %>%
  print(.)

# the treated frame is a "cross frame" which
# is a transform of the training data built
# as if the treatment were learned on a different
# disjoint training set to avoid nested model
# bias and over-fit.
dTrainNTreated %>%
```

```

head(.) %.>%
print(.)

# Any future application data is prepared with
# the prepare method.
dTestNTreated <- prepare(treatmentsN, dTestN, pruneSig=NULL)

dTestNTreated %.>%
  head(.) %.>%
  print(.)

```

---

MultinomialOutcomeTreatment

*Stateful object for designing and applying multinomial outcome treatments.*

---

## Description

Hold settings and results for multinomial classification data preparation.

## Usage

```

MultinomialOutcomeTreatment(
  ...,
  var_list,
  outcome_name,
  cols_to_copy = NULL,
  params = NULL,
  imputation_map = NULL
)

```

## Arguments

...	not used, force arguments to be specified by name.
var_list	Names of columns to treat (effective variables).
outcome_name	Name of column holding outcome variable. <code>dframe[[outcome_name]]</code> must be only finite non-missing values.
cols_to_copy	list of extra columns to copy.
params	parameters list from <code>multinomial_parameters</code>
imputation_map	map from column names to functions of signature <code>f(values: numeric, weights: numeric)</code> , simple missing value imputers.

**Details**

Please see [https://github.com/WinVector/vtreat/blob/main/Examples/fit\\_transform/fit\\_transform\\_api.md](https://github.com/WinVector/vtreat/blob/main/Examples/fit_transform/fit_transform_api.md), [mkCrossFrameMExperiment](#) and [prepare.multinomial\\_plan](#) for details.

Note: there currently is no `designTreatmentsM`, so `MultinomialOutcomeTreatment$fit()` is implemented in terms of `MultinomialOutcomeTreatment$fit_transform()`

---

multinomial\_parameters

*vtreat multinomial parameters.*

---

**Description**

A list of settings and values for vtreat multinomial classification fitting. Please see [https://github.com/WinVector/vtreat/blob/main/Examples/fit\\_transform/fit\\_transform\\_api.md](https://github.com/WinVector/vtreat/blob/main/Examples/fit_transform/fit_transform_api.md), [mkCrossFrameMExperiment](#) and [prepare.multinomial\\_plan](#) for details.

**Usage**

```
multinomial_parameters(user_params = NULL)
```

**Arguments**

`user_params` list of user overrides.

**Value**

filled out parameter list

---

novel\_value\_summary

*Report new/novel appearances of character values.*

---

**Description**

Report new/novel appearances of character values.

**Usage**

```
novel_value_summary(dframe, trackedValues)
```

**Arguments**

`dframe` Data frame to inspect.

`trackedValues` optional named list mapping variables to know values, allows warnings upon novel level appearances (see [track\\_values](#))

**Value**

frame of novel occurrences

**See Also**

[prepare.treatmentplan](#), [track\\_values](#)

**Examples**

```
set.seed(23525)
zip <- c(NA, paste('z', 1:10, sep = "_"))
N <- 10
d <- data.frame(zip = sample(zip, N, replace=TRUE),
                zip2 = sample(zip, N, replace=TRUE),
                y = runif(N))
dSample <- d[1:5, , drop = FALSE]
trackedValues <- track_values(dSample, c("zip", "zip2"))
novel_value_summary(d, trackedValues)
```

---

NumericOutcomeTreatment

*Stateful object for designing and applying numeric outcome treatments.*

---

**Description**

Hold settings and results for regression data preparation.

**Usage**

```
NumericOutcomeTreatment(
  ...,
  var_list,
  outcome_name,
  cols_to_copy = NULL,
  params = NULL,
  imputation_map = NULL
)
```

**Arguments**

...	not used, force arguments to be specified by name.
var_list	Names of columns to treat (effective variables).
outcome_name	Name of column holding outcome variable. <code>dframe[[outcomename]]</code> must be only finite non-missing values.
cols_to_copy	list of extra columns to copy.

params            parameters list from regression\_parameters

imputation\_map   map from column names to functions of signature f(values: numeric, weights: numeric), simple missing value imputers.

### Details

Please see [https://github.com/WinVector/vtreat/blob/main/Examples/fit\\_transform/fit\\_transform\\_api.md](https://github.com/WinVector/vtreat/blob/main/Examples/fit_transform/fit_transform_api.md), [mkCrossFrameNExperiment](#), [designTreatmentsN](#), and [prepare.treatmentplan](#) for details.

---

oneWayHoldout            *One way holdout, a splitFunction in the sense of vtreat::buildEvalSets.*

---

### Description

Note one way holdout can leak target expected values, so it should not be preferred in nested modeling situations. Also, doesn't respect nSplits.

### Usage

```
oneWayHoldout(nRows, nSplits, dframe, y)
```

### Arguments

nRows            number of rows to split (integer >1).

nSplits           number of groups to split into (ignored).

dframe            original data frame (ignored).

y                  numeric outcome variable (ignored).

### Value

split plan

### Examples

```
oneWayHoldout(3, NULL, NULL, NULL)
```

---

```
patch_columns_into_frame
```

*Patch columns into data.frame.*

---

### Description

Add columns from `new_frame` into `old_frame`, replacing any columns with matching names in `orig_frame` with values from `new_frame`.

### Usage

```
patch_columns_into_frame(orig_frame, new_frame)
```

### Arguments

<code>orig_frame</code>	data.frame to patch into.
<code>new_frame</code>	data.frame to take replacement columns from.

### Value

patched data.frame

### Examples

```
orig_frame <- data.frame(x = 1, y = 2)
new_frame <- data.frame(y = 3, z = 4)
patch_columns_into_frame(orig_frame, new_frame)
```

---

```
prepare
```

*Apply treatments and restrict to useful variables.*

---

### Description

Apply treatments and restrict to useful variables.

### Usage

```
prepare(treatmentplan, dframe, ...)
```

### Arguments

<code>treatmentplan</code>	Plan built by <code>designTreatmentsC()</code> or <code>designTreatmentsN()</code>
<code>dframe</code>	Data frame to be treated
<code>...</code>	no additional arguments, declared to forced named binding of later arguments

**See Also**

[prepare.treatmentplan](#), [prepare.simple\\_plan](#), [prepare.multinomial\\_plan](#)

---

```
prepare.multinomial_plan
```

*Function to apply mkCrossFrameMExperiment treatemnts.*

---

**Description**

Please see vignette("MultiClassVtreat", package = "vtreat") <https://winvector.github.io/vtreat/articles/MultiClassVtreat.html>.

**Usage**

```
## S3 method for class 'multinomial_plan'
prepare(
  treatmentplan,
  dframe,
  ...,
  pruneSig = NULL,
  scale = FALSE,
  doCollar = FALSE,
  varRestriction = NULL,
  codeRestriction = NULL,
  trackedValues = NULL,
  extracols = NULL,
  parallelCluster = NULL,
  use_parallel = TRUE,
  check_for_duplicate_frames = TRUE
)
```

**Arguments**

treatmentplan	multinomial_plan from mkCrossFrameMExperiment.
dframe	new data to process.
...	not used, declared to forced named binding of later arguments
pruneSig	suppress variables with significance above this level
scale	optional if TRUE replace numeric variables with single variable model regressions ("move to outcome-scale"). These have mean zero and (for variables with significant less than 1) slope 1 when regressed (lm for regression problems/glm for classification problems) against outcome.
doCollar	optional if TRUE collar numeric variables by cutting off after a tail-probability specified by collarProb during treatment design.
varRestriction	optional list of treated variable names to restrict to

codeRestriction	optional list of treated variable codes to restrict to
trackedValues	optional named list mapping variables to know values, allows warnings upon novel level appearances (see <a href="#">track_values</a> )
extracols	extra columns to copy.
parallelCluster	(optional) a cluster object created by package parallel or package snow.
use_parallel	logical, if TRUE use parallel methods.
check_for_duplicate_frames	logical, if TRUE check if we called prepare on same data.frame as design step.

**Value**

prepared data frame.

**See Also**

[mkCrossFrameMExperiment](#), [prepare](#)

---

prepare.simple\_plan    *Prepare a simple treatment.*

---

**Description**

Prepare a simple treatment.

**Usage**

```
## S3 method for class 'simple_plan'
prepare(treatmentplan, dframe, ...)
```

**Arguments**

treatmentplan	A simple treatment plan.
dframe	data.frame to be treated.
...	not used, present for S3 signature consistency.

**See Also**

[design\\_missingness\\_treatment](#), [prepare](#)

**Examples**

```
d <- wrapr::build_frame(
  "x1", "x2", "x3" |
  1 , 4 , "A" |
  NA , 5 , "B" |
  3 , 6 , NA )

plan <- design_missingness_treatment(d)
prepare(plan, d)

prepare(plan, data.frame(x1=NA, x2=NA, x3="E"))
```

---

prepare.treatmentplan *Apply treatments and restrict to useful variables.*

---

**Description**

Use a treatment plan to prepare a data frame for analysis. The resulting frame will have new effective variables that are numeric and free of NaN/NA. If the outcome column is present it will be copied over. The intent is that these frames are compatible with more machine learning techniques, and avoid a lot of corner cases (NA,NaN, novel levels, too many levels). Note: each column is processed independently of all others. Also copies over outcome if present. Note: treatmentplan's are not meant for long-term storage, a warning is issued if the version of vtreat that produced the plan differs from the version running prepare().

**Usage**

```
## S3 method for class 'treatmentplan'
prepare(
  treatmentplan,
  dframe,
  ...,
  pruneSig = NULL,
  scale = FALSE,
  doCollar = FALSE,
  varRestriction = NULL,
  codeRestriction = NULL,
  trackedValues = NULL,
  extracols = NULL,
  parallelCluster = NULL,
  use_parallel = TRUE,
  check_for_duplicate_frames = TRUE
)
```

**Arguments**

treatmentplan	Plan built by designTreatmentsC() or designTreatmentsN()
dframe	Data frame to be treated
...	no additional arguments, declared to forced named binding of later arguments
pruneSig	suppress variables with significance above this level
scale	optional if TRUE replace numeric variables with single variable model regressions ("move to outcome-scale"). These have mean zero and (for variables with significant less than 1) slope 1 when regressed (lm for regression problems/glm for classification problems) against outcome.
doCollar	optional if TRUE collar numeric variables by cutting off after a tail-probability specified by collarProb during treatment design.
varRestriction	optional list of treated variable names to restrict to
codeRestriction	optional list of treated variable codes to restrict to
trackedValues	optional named list mapping variables to know values, allows warnings upon novel level appearances (see <a href="#">track_values</a> )
extracols	extra columns to copy.
parallelCluster	(optional) a cluster object created by package parallel or package snow.
use_parallel	logical, if TRUE use parallel methods.
check_for_duplicate_frames	logical, if TRUE check if we called prepare on same data.frame as design step.

**Value**

treated data frame (all columns numeric- without NA, NaN)

**See Also**

[mkCrossFrameCExperiment](#), [mkCrossFrameNExperiment](#), [designTreatmentsC](#) [designTreatmentsN](#) [designTreatmentsZ](#), [prepare](#)

**Examples**

```
# categorical example
set.seed(23525)

# we set up our raw training and application data
dTrainC <- data.frame(
  x = c('a', 'a', 'a', 'b', 'b', NA, NA),
  z = c(1, 2, 3, 4, NA, 6, NA),
  y = c(FALSE, FALSE, TRUE, FALSE, TRUE, TRUE, TRUE))

dTestC <- data.frame(
  x = c('a', 'b', 'c', NA),
  z = c(10, 20, 30, NA))
```

```

# we perform a vtreat cross frame experiment
# and unpack the results into treatmentsC
# and dTrainCTreated
unpack[
  treatmentsC = treatments,
  dTrainCTreated = crossFrame
] <- mkCrossFrameCExperiment(
  dframe = dTrainC,
  varlist = setdiff(colnames(dTrainC), 'y'),
  outcomename = 'y',
  outcometarget = TRUE,
  verbose = FALSE)

# the treatments include a score frame relating new
# derived variables to original columns
treatmentsC$scoreFrame[, c('origName', 'varName', 'code', 'rsq', 'sig', 'extraModelDegrees')] %.>%
  print(.)

# the treated frame is a "cross frame" which
# is a transform of the training data built
# as if the treatment were learned on a different
# disjoint training set to avoid nested model
# bias and over-fit.
dTrainCTreated %.>%
  head(.) %.>%
  print(.)

# Any future application data is prepared with
# the prepare method.
dTestCTreated <- prepare(treatmentsC, dTestC, pruneSig=NULL)

dTestCTreated %.>%
  head(.) %.>%
  print(.)

```

---

pre\_comp\_xval

*Pre-computed cross-plan (so same split happens each time).*


---

## Description

Pre-computed cross-plan (so same split happens each time).

## Usage

```
pre_comp_xval(nRows, nSplits, splitplan)
```

**Arguments**

nRows	number of rows to split (integer >1).
nSplits	number of groups to split into (ignored).
splitplan	split plan to actually use

**Value**

splitplan

**Examples**

```
p1 <- oneWayHoldout(3,NULL,NULL,NULL)
p2 <- pre_comp_xval(3, 3, p1)
p2(3, 3)
```

---

```
print.multinomial_plan
```

*Print treatmentplan.*

---

**Description**

Print treatmentplan.

**Usage**

```
## S3 method for class 'multinomial_plan'
print(x, ...)
```

**Arguments**

x	treatmentplan
...	additional args (to match general signature).

---

`print.simple_plan`      *Print treatmentplan.*

---

### **Description**

Print treatmentplan.

### **Usage**

```
## S3 method for class 'simple_plan'  
print(x, ...)
```

### **Arguments**

x                      treatmentplan  
...                    additional args (to match general signature).

---

`print.treatmentplan`      *Print treatmentplan.*

---

### **Description**

Print treatmentplan.

### **Usage**

```
## S3 method for class 'treatmentplan'  
print(x, ...)
```

### **Arguments**

x                      treatmentplan  
...                    additional args (to match general signature).

### **See Also**

[designTreatmentsC](#), [designTreatmentsN](#), [designTreatmentsZ](#), [prepare.treatmentplan](#)

---

```
print.vtreatment      Print treatmentplan.
```

---

**Description**

Print treatmentplan.

**Usage**

```
## S3 method for class 'vtreatment'
print(x, ...)
```

**Arguments**

```
x          treatmentplan
...        additional args (to match general signature).
```

**See Also**

[designTreatmentsC](#), [designTreatmentsN](#), [designTreatmentsZ](#), [prepare.treatmentplan](#)

---

```
problemAppPlan      check if appPlan is a good carve-up of 1:nRows into nSplits groups
```

---

**Description**

check if appPlan is a good carve-up of 1:nRows into nSplits groups

**Usage**

```
problemAppPlan(nRows, nSplits, appPlan, strictCheck)
```

**Arguments**

```
nRows      number of rows to carve-up
nSplits    number of sets to carve-up into
appPlan    carve-up to critique
strictCheck logical, if true expect application data to be a carve-up and training data to be a maximal partition and to match nSplits.
```

**Value**

problem with carve-up (null if good)

**See Also**

[kWayCrossValidation](#), [kWayStratifiedY](#), and [makekWayCrossValidationGroupedByColumn](#)

**Examples**

```
plan <- kWayStratifiedY(3,2,NULL,NULL)
problemAppPlan(3,3,plan,TRUE)
```

---

regression\_parameters *vtreat regression parameters.*

---

**Description**

A list of settings and values for vtreat regression fitting. Please see [https://github.com/WinVector/vtreat/blob/main/Examples/fit\\_transform/fit\\_transform\\_api.md](https://github.com/WinVector/vtreat/blob/main/Examples/fit_transform/fit_transform_api.md), [mkCrossFrameCExperiment](#), [designTreatmentsC](#), and [mkCrossFrameNExperiment](#), [designTreatmentsN](#), [prepare.treatmentplan](#) for details.

**Usage**

```
regression_parameters(user_params = NULL)
```

**Arguments**

user\_params      list of user overrides.

**Value**

filled out parameter list

---

rquery\_prepare      *Materialize a treated data frame remotely.*

---

**Description**

Materialize a treated data frame remotely.

**Usage**

```
rquery_prepare(
  db,
  rqplan,
  data_source,
  result_table_name,
  ...,
  extracols = NULL,
  temporary = FALSE,
  overwrite = TRUE,
  attempt_nan_inf_mapping = FALSE,
  col_sample = NULL,
  return_ops = FALSE
)
```

```
materialize_treated(
  db,
  rqplan,
  data_source,
  result_table_name,
  ...,
  extracols = NULL,
  temporary = FALSE,
  overwrite = TRUE,
  attempt_nan_inf_mapping = FALSE,
  col_sample = NULL,
  return_ops = FALSE
)
```

**Arguments**

db	a db handle.
rqplan	an query plan produced by <code>as_rquery_plan()</code> .
data_source	relop, data source (usually a <code>relop_table_source</code> ).
result_table_name	character, table name to land result in
...	force later arguments to bind by name.
extracols	extra columns to copy.
temporary	logical, if TRUE try to make result temporary.
overwrite	logical, if TRUE try to overwrite result.
attempt_nan_inf_mapping	logical, if TRUE attempt to map NaN and Infinity to NA/NULL (goot on PostgreSQL, not on Spark).
col_sample	sample of data to determine column types.
return_ops	logical, if TRUE return operator tree instead of materializing.

**Value**

description of treated table.

**Functions**

- `materialize_treated()`: old name for `rquery_prepare` function

**See Also**

[as\\_rquery\\_plan](#), [rqdatatable\\_prepare](#)

---

solve_piecewise	<i>Solve as piecewise linear problem, numeric target.</i>
-----------------	---

---

**Description**

Return a vector of length `y` that is a piecewise function of `x`. This vector is picked as close to `y` (by square-distance) as possible for a set of `x`-only determined cut-points. Cross-validates for a good number of segments.

**Usage**

```
solve_piecewise(varName, x, y, w = NULL)
```

**Arguments**

<code>varName</code>	character, name of variable
<code>x</code>	numeric input (not empty, no NAs).
<code>y</code>	numeric or castable to such (same length as <code>x</code> no NAs), output to match
<code>w</code>	numeric positive, same length as <code>x</code> (weights, can be NULL)

**Value**

segmented `y` prediction

---

solve_piecewisec	<i>Solve as piecewise logit problem, categorical target.</i>
------------------	--

---

**Description**

Return a vector of length y that is a piecewise function of x. This vector is picked as close to y (by square-distance) as possible for a set of x-only determined cut-points. Cross-validates for a good number of segments.

**Usage**

```
solve_piecewisec(varName, x, y, w = NULL)
```

**Arguments**

varName	character, name of variable
x	numeric input (not empty, no NAs).
y	numeric or castable to such (same length as x no NAs), output to match
w	numeric positive, same length as x (weights, can be NULL)

**Value**

segmented y prediction

---

spline_variable	<i>Spline variable numeric target.</i>
-----------------	--

---

**Description**

Return a spline approximation of data.

**Usage**

```
spline_variable(varName, x, y, w = NULL)
```

**Arguments**

varName	character, name of variable
x	numeric input (not empty, no NAs).
y	numeric or castable to such (same length as x no NAs), output to match
w	numeric positive, same length as x (weights, can be NULL)

**Value**

spline y prediction

---

spline_variablec	<i>Spline variable categorical target.</i>
------------------	--

---

**Description**

Return a spline approximation of the change in log odds.

**Usage**

```
spline_variablec(varName, x, y, w = NULL)
```

**Arguments**

varName	character, name of variable
x	numeric input (not empty, no NAs).
y	numeric or castable to such (same length as x no NAs), output to match
w	numeric positive, same length as x (weights, can be NULL)

**Value**

spline y prediction

---

square_window	<i>Build a square windows variable, numeric target.</i>
---------------	---

---

**Description**

Build a square moving average window (KNN in 1d). This is a high-frequency feature.

**Usage**

```
square_window(varName, x, y, w = NULL)
```

**Arguments**

varName	character, name of variable
x	numeric input (not empty, no NAs).
y	numeric or castable to such (same length as x no NAs), output to match
w	numeric positive, same length as x (weights, can be NULL) IGNORED

**Value**

segmented y prediction

### Examples

```
d <- data.frame(x = c(NA, 1:6), y = c(0, 0, 0, 1, 1, 0, 0))
square_window("v", d$x, d$y)
```

---

square_windowc	<i>Build a square windows variable, categorical target.</i>
----------------	---

---

### Description

Build a square moving average window (KNN in 1d). This is a high-frequency feature. Approximation of the change in log odds.

### Usage

```
square_windowc(varName, x, y, w = NULL)
```

### Arguments

varName	character, name of variable
x	numeric input (not empty, no NAs).
y	numeric or castable to such (same length as x no NAs), output to match
w	numeric positive, same length as x (weights, can be NULL) IGNORED

### Value

segmented y prediction

### Examples

```
d <- data.frame(x = c(NA, 1:6), y = c(0, 0, 0, 1, 1, 0, 0))
square_window("v", d$x, d$y)
```

---

track_values	<i>Track unique character values for variables.</i>
--------------	---

---

### Description

Builds lists of observed unique character values of varlist variables from the data frame.

### Usage

```
track_values(dframe, varlist)
```

### Arguments

dframe	Data frame to learn treatments from (training data), must have at least 1 row.
varlist	Names of columns to treat (effective variables).

### Value

named list of values seen.

### See Also

[prepare.treatmentplan](#), [novel\\_value\\_summary](#)

### Examples

```
set.seed(23525)
zip <- c(NA, paste('z', 1:100, sep = "_"))
N <- 500
d <- data.frame(zip = sample(zip, N, replace=TRUE),
                zip2 = sample(zip, N, replace=TRUE),
                y = runif(N))
dSample <- d[1:300, , drop = FALSE]
tplan <- designTreatmentsN(dSample,
                           c("zip", "zip2"), "y",
                           verbose = FALSE)
trackedValues <- track_values(dSample, c("zip", "zip2"))
# don't normally want to catch warnings,
# doing it here as this is an example
# and must not have unhandled warnings.
tryCatch(
  prepare(tplan, d, trackedValues = trackedValues),
  warning = function(w) { cat(paste(w, collapse = "\n")) })
```

---

UnsupervisedTreatment *Stateful object for designing and applying unsupervised treatments.*

---

### Description

Hold settings and results for unsupervised data preparation.

### Usage

```
UnsupervisedTreatment(
  ...,
  var_list,
  cols_to_copy = NULL,
  params = NULL,
  imputation_map = NULL
)
```

### Arguments

...	not used, force arguments to be specified by name.
var_list	Names of columns to treat (effective variables).
cols_to_copy	list of extra columns to copy.
params	parameters list from unsupervised_parameters
imputation_map	map from column names to functions of signature f(values: numeric, weights: numeric), simple missing value imputers.

### Details

Please see [https://github.com/WinVector/vtreat/blob/main/Examples/fit\\_transform/fit\\_transform\\_api.md](https://github.com/WinVector/vtreat/blob/main/Examples/fit_transform/fit_transform_api.md), [designTreatmentsZ](#) and [prepare.treatmentplan](#) for details.

Note: for UnsupervisedTreatment `fit_transform(d)` is implemented as `fit(d)$transform(d)`.

---

unsupervised\_parameters  
*vtreat unsupervised parameters.*

---

### Description

A list of settings and values for vtreat unsupervised fitting. Please see [https://github.com/WinVector/vtreat/blob/main/Examples/fit\\_transform/fit\\_transform\\_api.md](https://github.com/WinVector/vtreat/blob/main/Examples/fit_transform/fit_transform_api.md), [designTreatmentsZ](#), and [prepare.treatmentplan](#) for details.

### Usage

```
unsupervised_parameters(user_params = NULL)
```

**Arguments**

user\_params      list of user overrides.

**Value**

filled out parameter list

---

value\_variables\_C      *Value variables for prediction a categorical outcome.*

---

**Description**

Value variables for prediction a categorical outcome.

**Usage**

```
value_variables_C(
  dframe,
  varlist,
  outcomename,
  outcometarget,
  ...,
  weights = c(),
  minFraction = 0.02,
  smFactor = 0,
  rareCount = 0,
  rareSig = 1,
  collarProb = 0,
  scale = FALSE,
  doCollar = FALSE,
  splitFunction = NULL,
  ncross = 3,
  forceSplit = FALSE,
  catScaling = TRUE,
  verbose = FALSE,
  parallelCluster = NULL,
  use_parallel = TRUE,
  customCoders = list(c.PiecewiseV.num = vtreat::solve_pieewisec, n.PiecewiseV.num =
    vtreat::solve_pieewise, c.knearest.num = vtreat::square_windowc, n.knearest.num =
    vtreat::square_window),
  codeRestriction = c("PiecewiseV", "knearest", "clean", "isBAD", "catB", "catP"),
  missingness_imputation = NULL,
  imputation_map = NULL
)
```

**Arguments**

dframe	Data frame to learn treatments from (training data), must have at least 1 row.
varlist	Names of columns to treat (effective variables).
outcomename	Name of column holding outcome variable. dframe[[outcomename]] must be only finite non-missing values.
outcometarget	Value/level of outcome to be considered "success", and there must be a cut such that dframe[[outcomename]]==outcometarget at least twice and dframe[[outcomename]]!=outcometarget at least twice.
...	no additional arguments, declared to forced named binding of later arguments
weights	optional training weights for each row
minFraction	optional minimum frequency a categorical level must have to be converted to an indicator column.
smFactor	optional smoothing factor for impact coding models.
rareCount	optional integer, allow levels with this count or below to be pooled into a shared rare-level. Defaults to 0 or off.
rareSig	optional numeric, suppress levels from pooling at this significance value greater. Defaults to NULL or off.
collarProb	what fraction of the data (pseudo-probability) to collar data at if doCollar is set during <code>prepare.treatmentplan</code> .
scale	optional if TRUE replace numeric variables with regression ("move to outcome-scale").
doCollar	optional if TRUE collar numeric variables by cutting off after a tail-probability specified by collarProb during treatment design.
splitFunction	(optional) see vtreat::buildEvalSets .
ncross	optional scalar >=2 number of cross-validation rounds to design.
forceSplit	logical, if TRUE force cross-validated significance calculations on all variables.
catScaling	optional, if TRUE use glm() linkspace, if FALSE use lm() for scaling.
verbose	if TRUE print progress.
parallelCluster	(optional) a cluster object created by package parallel or package snow.
use_parallel	logical, if TRUE use parallel methods.
customCoders	additional coders to use for variable importance estimate.
codeRestriction	codes to restrict to for variable importance estimate.
missingness_imputation	function of signature f(values: numeric, weights: numeric), simple missing value imputer.
imputation_map	map from column names to functions of signature f(values: numeric, weights: numeric), simple missing value imputers.

**Value**

table of variable valuations

---

value_variables_N	<i>Value variables for prediction a numeric outcome.</i>
-------------------	--

---

### Description

Value variables for prediction a numeric outcome.

### Usage

```
value_variables_N(
  dframe,
  varlist,
  outcomename,
  ...,
  weights = c(),
  minFraction = 0.02,
  smFactor = 0,
  rareCount = 0,
  rareSig = 1,
  collarProb = 0,
  scale = FALSE,
  doCollar = FALSE,
  splitFunction = NULL,
  ncross = 3,
  forceSplit = FALSE,
  verbose = FALSE,
  parallelCluster = NULL,
  use_parallel = TRUE,
  customCoders = list(c.PiecewiseV.num = vtreat::solve_pieciwsec, n.PiecewiseV.num =
    vtreat::solve_pieciwise, c.knearest.num = vtreat::square_windowc, n.knearest.num =
    vtreat::square_window),
  codeRestriction = c("PiecewiseV", "knearest", "clean", "isBAD", "catB", "catP"),
  missingness_imputation = NULL,
  imputation_map = NULL
)
```

### Arguments

dframe	Data frame to learn treatments from (training data), must have at least 1 row.
varlist	Names of columns to treat (effective variables).
outcomename	Name of column holding outcome variable. dframe[[outcomename]] must be only finite non-missing values and there must be a cut such that dframe[[outcomename]] is both above the cut at least twice and below the cut at least twice.
...	no additional arguments, declared to forced named binding of later arguments
weights	optional training weights for each row

minFraction	optional minimum frequency a categorical level must have to be converted to an indicator column.
smFactor	optional smoothing factor for impact coding models.
rareCount	optional integer, allow levels with this count or below to be pooled into a shared rare-level. Defaults to 0 or off.
rareSig	optional numeric, suppress levels from pooling at this significance value greater. Defaults to NULL or off.
collarProb	what fraction of the data (pseudo-probability) to collar data at if doCollar is set during <code>prepare.treatmentplan</code> .
scale	optional if TRUE replace numeric variables with regression ("move to outcome-scale").
doCollar	optional if TRUE collar numeric variables by cutting off after a tail-probability specified by collarProb during treatment design.
splitFunction	(optional) see <code>vtreat::buildEvalSets</code> .
ncross	optional scalar $\geq 2$ number of cross-validation rounds to design.
forceSplit	logical, if TRUE force cross-validated significance calculations on all variables.
verbose	if TRUE print progress.
parallelCluster	(optional) a cluster object created by package <code>parallel</code> or package <code>snow</code> .
use_parallel	logical, if TRUE use parallel methods.
customCoders	additional coders to use for variable importance estimate.
codeRestriction	codes to restrict to for variable importance estimate.
missingness_imputation	function of signature <code>f(values: numeric, weights: numeric)</code> , simple missing value imputer.
imputation_map	map from column names to functions of signature <code>f(values: numeric, weights: numeric)</code> , simple missing value imputers.

**Value**

table of variable valuations

---

variable_values	<i>Return variable evaluations.</i>
-----------------	-------------------------------------

---

**Description**

Return variable evaluations.

**Usage**

`variable_values(sf)`

**Arguments**

sf                    scoreFrame from from vtreat treatments

**Value**

per-original variable evaluations

---

vnames                    *New treated variable names from a treatmentplan\$treatment item.*

---

**Description**

New treated variable names from a treatmentplan\$treatment item.

**Usage**

vnames(x)

**Arguments**

x                    vtreatment item

**See Also**

[designTreatmentsC](#) [designTreatmentsN](#) [designTreatmentsZ](#)

---

vorig                    *Original variable name from a treatmentplan\$treatment item.*

---

**Description**

Original variable name from a treatmentplan\$treatment item.

**Usage**

vorig(x)

**Arguments**

x                    vtreatment item.

**See Also**

[designTreatmentsC](#) [designTreatmentsN](#) [designTreatmentsZ](#)

# Index

apply\_transform, 4  
as\_rquery\_plan, 4, 51

BinomialOutcomeTreatment, 6  
buildEvalSets, 7

center\_scale, 9  
classification\_parameters, 9

design\_missingness\_treatment, 16, 42  
designTreatmentsC, 6, 9, 10, 14, 16, 27, 29, 35, 44, 47–49, 61  
designTreatmentsN, 12, 12, 16, 29, 33, 35, 39, 44, 47–49, 61  
designTreatmentsZ, 12, 14, 15, 44, 47, 48, 56, 61

fit, 18  
fit\_prepare, 18  
fit\_transform, 19  
format.vtreatment, 20

get\_feature\_names, 21  
get\_score\_frame, 21  
get\_transform, 22  
getSplitPlanAppLabels, 20

kWayCrossValidation, 8, 20, 22, 49  
kWayStratifiedY, 8, 20, 23, 49  
kWayStratifiedYReplace, 24

makeCustomCoderCat, 24  
makeCustomCoderNum, 25  
makekWayCrossValidationGroupedByColumn, 8, 20, 26, 49  
materialize\_treated (rquery\_prepare), 49  
mkCrossFrameCExperiment, 6, 9, 10, 12, 27, 44, 49  
mkCrossFrameMExperiment, 30, 37, 42  
mkCrossFrameNExperiment, 12, 14, 33, 39, 44, 49

multinomial\_parameters, 37  
MultinomialOutcomeTreatment, 36

novel\_value\_summary, 37, 55  
NumericOutcomeTreatment, 38

oneWayHoldout, 39

patch\_columns\_into\_frame, 40  
pre\_comp\_xval, 45  
prepare, 40, 42, 44  
prepare.multinomial\_plan, 31, 32, 37, 41, 41  
prepare.simple\_plan, 17, 41, 42  
prepare.treatmentplan, 6, 9, 11–16, 28, 29, 34, 35, 38, 39, 41, 43, 47–49, 55, 56, 58, 60  
print.multinomial\_plan, 46  
print.simple\_plan, 47  
print.treatmentplan, 47  
print.vtreatment, 48  
problemAppPlan, 48

regression\_parameters, 49  
rqdatatable\_prepare, 51  
rquery\_prepare, 5, 49

solve\_piecewise, 51  
solve\_piecewisec, 52  
spline\_variable, 52  
spline\_variablec, 53  
square\_window, 53  
square\_windowc, 54

track\_values, 37, 38, 42, 44, 55

unsupervised\_parameters, 56  
UnsupervisedTreatment, 56

value\_variables\_C, 57  
value\_variables\_N, 59

variable\_values, [60](#)  
vnames, [61](#)  
vorig, [61](#)  
vtreat (vtreat-package), [3](#)  
vtreat-package, [3](#)