Package ‘hardhat’

July 14, 2021

**Title**  Construct Modeling Packages

**Version**  0.1.6

**Maintainer**  Davis Vaughan  <davis@rstudio.com>

**Description**  Building modeling packages is hard. A large amount
of effort generally goes into providing an implementation for a new
method that is efficient, fast, and correct, but often less emphasis
is put on the user interface. A good interface requires specialized
knowledge about S3 methods and formulas, which the average package
developer might not have. The goal of ‘hardhat’ is to reduce the
burden around building new modeling packages by providing
functionality for preprocessing, predicting, and validating input.

**License**  MIT + file LICENSE

**URL**  https://github.com/tidymodels/hardhat

**BugReports**  https://github.com/tidymodels/hardhat/issues

**Depends**  R (>= 2.10)

**Imports**  glue,
         rlang (>= 0.4.2),
         tibble,
         vctrs (>= 0.3.0)

**Suggests**  covr,
        crayon,
        devtools,
        knitr,
        Matrix,
        modeldata (>= 0.0.2),
        recipes (>= 0.1.8),
        rmarkdown (>= 2.3),
        roxygen2,
        testthat (>= 2.1.0),
        usethis

**Config/Needs/website**  tidyverse/tidytemplate

**VignetteBuilder**  knitr

**Encoding**  UTF-8

**LazyData**  true

**Roxygen**  list(markdown = TRUE)

**RoxygenNote**  7.1.1
R topics documented:

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>modeling-package</td>
<td>2</td>
</tr>
<tr>
<td>add_intercept_column</td>
<td>4</td>
</tr>
<tr>
<td>default_formula_blueprint</td>
<td>4</td>
</tr>
<tr>
<td>default_recipe_blueprint</td>
<td>10</td>
</tr>
<tr>
<td>default_xy_blueprint</td>
<td>13</td>
</tr>
<tr>
<td>delete_response</td>
<td>15</td>
</tr>
<tr>
<td>forge</td>
<td>16</td>
</tr>
<tr>
<td>get_data_classes</td>
<td>17</td>
</tr>
<tr>
<td>get_levels</td>
<td>18</td>
</tr>
<tr>
<td>hardhat-example-data</td>
<td>19</td>
</tr>
<tr>
<td>hardhat-extract</td>
<td>20</td>
</tr>
<tr>
<td>is_blueprint</td>
<td>21</td>
</tr>
<tr>
<td>model_frame</td>
<td>21</td>
</tr>
<tr>
<td>model_matrix</td>
<td>22</td>
</tr>
<tr>
<td>model_offset</td>
<td>24</td>
</tr>
<tr>
<td>mold</td>
<td>25</td>
</tr>
<tr>
<td>new_default_formula_blueprint</td>
<td>26</td>
</tr>
<tr>
<td>new_formula_blueprint</td>
<td>28</td>
</tr>
<tr>
<td>new_model</td>
<td>31</td>
</tr>
<tr>
<td>refresh_blueprint</td>
<td>32</td>
</tr>
<tr>
<td>run_mold</td>
<td>33</td>
</tr>
<tr>
<td>scream</td>
<td>34</td>
</tr>
<tr>
<td>shrink</td>
<td>37</td>
</tr>
<tr>
<td>spruce</td>
<td>38</td>
</tr>
<tr>
<td>standardize</td>
<td>39</td>
</tr>
<tr>
<td>update_blueprint</td>
<td>40</td>
</tr>
<tr>
<td>validate_column_names</td>
<td>40</td>
</tr>
<tr>
<td>validate_no_formula_duplication</td>
<td>42</td>
</tr>
<tr>
<td>validate_outcomes_are_binary</td>
<td>44</td>
</tr>
<tr>
<td>validate_outcomes_are_factors</td>
<td>45</td>
</tr>
<tr>
<td>validate_outcomes_are_numeric</td>
<td>46</td>
</tr>
<tr>
<td>validate_outcomes_are_univariate</td>
<td>47</td>
</tr>
<tr>
<td>validate_prediction_size</td>
<td>49</td>
</tr>
<tr>
<td>validate_predictors_are_numeric</td>
<td>50</td>
</tr>
</tbody>
</table>

Index 52

*modeling-package*  
*Create a modeling package*

Description

create_modeling_package() will:

- Call usethis::create_package() to set up a new R package.
- Call use_modeling_deps().
- Call use_modeling_files().

use_modeling_deps() will:
• Add hardhat, rlang, and stats to Imports
• Add recipes to Suggests
• If roxygen2 is available, use roxygen markdown

use_modeling_files() will:

• Add a package documentation file
• Generate and populate 3 files in R/:
  – {{model}}-constructor.R
  – {{model}}-fit.R
  – {{model}}-predict.R

Usage

create_modeling_package(path, model, fields = NULL, open = interactive())

use_modeling_deps()

use_modeling_files(model)

Arguments

path          A path. If it exists, it is used. If it does not exist, it is created, provided that the parent path exists.
model         A string. The name of the high level modeling function that users will call. For example, "linear_regression". This will be used to populate the skeleton. Spaces are not allowed.
fields        A named list of fields to add to DESCRIPTION, potentially overriding default values. See usethis::use_description() for how you can set personalized defaults using package options.
open          If TRUE, activates the new project:
  • If RStudio desktop, the package is opened in a new session.
  • If on RStudio server, the current RStudio project is activated.
  • Otherwise, the working directory and active project is changed.

Value

create_modeling_package() returns the project path invisibly.
use_modeling_deps() returns invisibly.
use_modeling_files() return model invisibly.
add_intercept_column  Add an intercept column to data

Description
This function adds an integer column of 1’s to data.

Usage
add_intercept_column(data, name = "(Intercept)")

Arguments
data  A data frame or matrix.
name  The name for the intercept column. Defaults to "(Intercept)", which is the same name that stats::lm() uses.

Details
If a column named name already exists in data, then data is returned unchanged and a warning is issued.

Value
data with an intercept column.

Examples
add_intercept_column(mtcars)
add_intercept_column(mtcars, "intercept")
add_intercept_column(as.matrix(mtcars))

default_formula_blueprint

Description
This pages holds the details for the formula preprocessing blueprint. This is the blueprint used by default from mold() if x is a formula.
Usage

```r
default_formula_blueprint(
    intercept = FALSE,
    allow_novel_levels = FALSE,
    indicators = "traditional",
    composition = "tibble"
)
```

```r
## S3 method for class 'formula'
mold(formula, data, ..., blueprint = NULL)
```

Arguments

- **intercept** A logical. Should an intercept be included in the processed data? This information is used by the `process` function in the `mold` and `forge` function list.

- **allow_novel_levels** A logical. Should novel factor levels be allowed at prediction time? This information is used by the `clean` function in the `forge` function list, and is passed on to `scream()`.

- **indicators** A single character string. Control how factors are expanded into dummy variable indicator columns. One of:
  - "traditional" - The default. Create dummy variables using the traditional `model.matrix()` infrastructure. Generally this creates $K - 1$ indicator columns for each factor, where $K$ is the number of levels in that factor.
  - "none" - Leave factor variables alone. No expansion is done.
  - "one_hot" - Create dummy variables using a one-hot encoding approach that expands unordered factors into all $K$ indicator columns, rather than $K - 1$.

- **composition** Either "tibble", "matrix", or "dgCMatix" for the format of the processed predictors. If "matrix" or "dgCMatix" are chosen, all of the predictors must be numeric after the preprocessing method has been applied; otherwise an error is thrown.

- **formula** A formula specifying the predictors and the outcomes.

- **data** A data frame or matrix containing the outcomes and predictors.

- **...** Not used.

- **blueprint** A preprocessing blueprint. If left as NULL, then a `default_formula_blueprint()` is used.

Details

While not different from base R, the behavior of expanding factors into dummy variables when `indicators = "traditional"` and an intercept is *not* present is not always intuitive and should be documented.

- When an intercept is present, factors are expanded into $K-1$ new columns, where $K$ is the number of levels in the factor.
When an intercept is not present, the first factor is expanded into all $K$ columns (one-hot encoding), and the remaining factors are expanded into $K-1$ columns. This behavior ensures that meaningful predictions can be made for the reference level of the first factor, but is not the exact "no intercept" model that was requested. Without this behavior, predictions for the reference level of the first factor would always be forced to 0 when there is no intercept.

Offsets can be included in the formula method through the use of the inline function `stats::offset()`. These are returned as a tibble with 1 column named `.offset` in the `$extras$offset` slot of the return value.

Value

For `default_formula_blueprint()`, a formula blueprint.

Mold

When `mold()` is used with the default formula blueprint:

- Predictors
  - The RHS of the formula is isolated, and converted to its own 1 sided formula: `~ RHS`.
  - Runs `stats::model.frame()` on the RHS formula and uses `data`.
  - If `indicators = "traditional"`, it then runs `stats::model.matrix()` on the result.
  - If `indicators = "none"`, factors are removed before `model.matrix()` is run, and then added back afterwards. No interactions or inline functions involving factors are allowed.
  - If `indicators = "one_hot"`, it then runs `stats::model.matrix()` on the result using a contrast function that creates indicator columns for all levels of all factors.
  - If any offsets are present from using `offset()`, then they are extracted with `model.offset()`.
  - If `intercept = TRUE`, adds an intercept column.
  - Coerces the result of the above steps to a tibble.

- Outcomes
  - The LHS of the formula is isolated, and converted to its own 1 sided formula: `~ LHS`.
  - Runs `stats::model.frame()` on the LHS formula and uses `data`.
  - Coerces the result of the above steps to a tibble.

Forge

When `forge()` is used with the default formula blueprint:

- It calls `shrink()` to trim `new_data` to only the required columns and coerce `new_data` to a tibble.
- It calls `scream()` to perform validation on the structure of the columns of `new_data`.
- Predictors
  - It runs `stats::model.frame()` on `new_data` using the stored terms object corresponding to the `predictors`. 
– If, in the original `mold()` call, `indicators = "traditional"` was set, it then runs `stats::model.matrix()` on the result.
– If, in the original `mold()` call, `indicators = "none"` was set, it runs `stats::model.matrix()` on the result without the factor columns, and then adds them on afterwards.
– If, in the original `mold()` call, `indicators = "one hot"` was set, it runs `stats::model.matrix()` on the result with a contrast function that includes indicators for all levels of all factor columns.
– If any offsets are present from using `offset()` in the original call to `mold()`, then they are extracted with `model.offset()`.
– If `intercept = TRUE` in the original call to `mold()`, then an intercept column is added.
– It coerces the result of the above steps to a tibble.

**Outcomes**

– It runs `stats::model.frame()` on `new data` using the stored terms object corresponding to the `outcomes`.
– Coerces the result to a tibble.

### Differences From Base R

There are a number of differences from base R regarding how formulas are processed by `mold()` that require some explanation.

Multivariate outcomes can be specified on the LHS using syntax that is similar to the RHS (i.e. `outcome_1 + outcome_2 ~ predictors`). If any complex calculations are done on the LHS and they return matrices (like `stats::poly()`), then those matrices are flattened into multiple columns of the tibble after the call to `model.frame()`. While this is possible, it is not recommended, and if a large amount of preprocessing is required on the outcomes, then you are better off using a `recipes::recipe()`.

Global variables are not allowed in the formula. An error will be thrown if they are included. All terms in the formula should come from `data`.

By default, intercepts are not included in the predictor output from the formula. To include an intercept, set `blueprint = default_formula_blueprint(intercept = TRUE)`. The rationale for this is that many packages either always require or never allow an intercept (for example, the `earth` package), and they do a large amount of extra work to keep the user from supplying one or removing it. This interface standardizes all of that flexibility in one place.

### Examples

```r
# Call mold() with the training data
processed <- mold(
  log(num_1) ~ num_2 + fac_1,
  example_train,
  blueprint = default_formula_blueprint(intercept = TRUE)
)
```
# Then, call forge() with the blueprint and the test data
# to have it preprocess the test data in the same way
forge(example_test, processed$blueprint)

# Use 'outcomes = TRUE' to also extract the preprocessed outcome
forge(example_test, processed$blueprint, outcomes = TRUE)

# Factors without an intercept

# No intercept is added by default
processed <- mold(num_1 ~ fac_1 + fac_2, example_train)

# So, for factor columns, the first factor is completely expanded into all
# 'K' columns (the number of levels), and the subsequent factors are expanded
# into 'K - 1' columns.
processed$predictors

# In the above example, `fac_1` is expanded into all three columns,
# `fac_2` is not. This behavior comes from `model.matrix()`, and is somewhat
# known in the R community, but can lead to a model that is difficult to
# interpret since the corresponding p-values are testing wildly different
# hypotheses.

# To get all indicators for all columns (irrespective of the intercept),
# use the `indicators = "one_hot"` option
processed <- mold(
  num_1 ~ fac_1 + fac_2,
  example_train,
  blueprint = default_formula_blueprint(indicators = "one_hot")
)

processed$predictors

# It is not possible to construct a no-intercept model that expands all
# factors into 'K - 1' columns using the formula method. If required, a
# recipe could be used to construct this model.

# Global variables
y <- rep(1, times = nrow(example_train))

# In base R, global variables are allowed in a model formula
frame <- model.frame(fac_1 ~ y + num_2, example_train)
head(frame)

# mold() does not allow them, and throws an error
try(mold(fac_1 ~ y + num_2, example_train))

# Dummy variables and interactions

# By default, factor columns are expanded
# and interactions are created, both by
# calling `model.matrix()`. Some models (like
# tree based models) can take factors directly
# but still might want to use the formula method. 
# In those cases, set `indicators = "none"` to not 
# run `model.matrix()` on factor columns. Interactions 
# are still allowed and are run on numeric columns.

```r
bp_no_indicators <- default_formula_blueprint(indicators = "none")
```

```r
processed <- mold(
  ~ fac_1 + num_1:num_2,
  example_train,
  blueprint = bp_no_indicators
)
```

```r
processed$predictors
```

# An informative error is thrown when `indicators = "none"` and 
# factors are present in interaction terms or in inline functions

```r
try(mold(num_1 ~ num_2:fac_1, example_train, blueprint = bp_no_indicators))
try(mold(num_1 ~ paste0(fac_1), example_train, blueprint = bp_no_indicators))
```

# -----------------------------------------------------------------------------
# Multivariate outcomes

# Multivariate formulas can be specified easily
```r
processed <- mold(num_1 + log(num_2) ~ fac_1, example_train)
```

```r
processed$outcomes
```

# Inline functions on the LHS are run, but any matrix 
# output is flattened (like what happens in `model.matrix()`) 
# (essentially this means you don't wind up with columns 
# in the tibble that are matrices)
```r
processed <- mold(poly(num_2, degree = 2) ~ fac_1, example_train)
```

```r
processed$outcomes
```

# TRUE
```r
ncol(processed$outcomes) == 2
```

# Multivariate formulas specified in mold() 
# carry over into forge()
```r
forge(example_test, processed$blueprint, outcomes = TRUE)
```

# -----------------------------------------------------------------------------
# Offsets

# Offsets are handled specially in base R, so they deserve special 
# treatment here as well. You can add offsets using the inline function 
# `offset()`
```r
processed <- mold(num_1 ~ offset(num_2) + fac_1, example_train)
```

```r
processed$extras$offset
```

# Multiple offsets can be included, and they get added together 
```r
processed <- mold(
  num_1 ~ offset(num_2) + offset(num_3),
  example_train
)
```
# Forging test data will also require
# and include the offset
forge(example_test, processed$blueprint)

# Interception only
# Because '1' and '0' are intercept modifying terms, they are
# not allowed in the formula and are instead controlled by the
# 'intercept' argument of the blueprint. To use an intercept
# only formula, you should supply 'NULL' on the RHS of the formula.
mold(
  ~ NULL,
  example_train,
  blueprint = default_formula_blueprint(intercept = TRUE)
)

# Matrix output for predictors
# You can change the 'composition' of the predictor data set
bp <- default_formula_blueprint(composition = "dgCMatrix")
processed <- mold(log(num_1) ~ num_2 + fac_1, example_train, blueprint = bp)
class(processed$predictors)

---

**default_recipe_blueprint**

*Default recipe blueprint*

**Description**

This page holds the details for the recipe preprocessing blueprint. This is the blueprint used by default from mold() if x is a recipe.

**Usage**

default_recipe_blueprint(
  intercept = FALSE,
  allow_novel_levels = FALSE,
  fresh = TRUE,
  composition = "tibble"
)

## S3 method for class 'recipe'
mold(x, data, ..., blueprint = NULL)
default_recipe_blueprint

Arguments

intercept A logical. Should an intercept be included in the processed data? This information is used by the process function in the mold and forge function list.

allow_novel_levels A logical. Should novel factor levels be allowed at prediction time? This information is used by the clean function in the forge function list, and is passed on to scream().

fresh Should already trained operations be re-trained when prep() is called?

composition Either "tibble", "matrix", or "dgCMatrix" for the format of the processed predictors. If "matrix" or "dgCMatrix" are chosen, all of the predictors must be numeric after the preprocessing method has been applied; otherwise an error is thrown.

x An unprepped recipe created from recipes::recipe().

data A data frame or matrix containing the outcomes and predictors.

... Not used.

blueprint A preprocessing blueprint. If left as NULL, then a default_recipe_blueprint() is used.

Value

For default_recipe_blueprint(), a recipe blueprint.

Mold

When mold() is used with the default recipe blueprint:

- It calls recipes::prep() to prep the recipe.
- It calls recipes::juice() to extract the outcomes and predictors. These are returned as tibbles.
- If intercept = TRUE, adds an intercept column to the predictors.

Forge

When forge() is used with the default recipe blueprint:

- It calls shrink() to trim new_data to only the required columns and coerce new_data to a tibble.
- It calls scream() to perform validation on the structure of the columns of new_data.
- It calls recipes::bake() on the new_data using the prepped recipe used during training.
- It adds an intercept column onto new_data if intercept = TRUE.

Examples

library(recipes)

# ---------------------------------------------------------------------------
# Setup
#
train <- iris[1:100,]
test <- iris[101:150,]

# Create a recipe that logs a predictor
rec <- recipe(Species ~ Sepal.Length + Sepal.Width, train) %>%
    step_log(Sepal.Length)

processed <- mold(rec, train)

# Sepal.Length has been logged
processed$predictors

# The underlying blueprint is a prepped recipe
processed$blueprint$recipe

# Call forge() with the blueprint and the test data
# to have it preprocess the test data in the same way
forge(test, processed$blueprint)

# Use `outcomes = TRUE` to also extract the preprocessed outcome!
# This logged the Sepal.Length column of `new_data`
forge(test, processed$blueprint, outcomes = TRUE)

# With an intercept

# You can add an intercept with `intercept = TRUE`
processed <- mold(rec, train, blueprint = default_recipe_blueprint(intercept = TRUE))

processed$predictors

# But you also could have used a recipe step
rec2 <- step_intercept(rec)
mold(rec2, iris)$predictors

# Non standard roles

# If you have custom recipe roles, they are processed and returned in
# the `$extras$roles` slot of the return value of `mold()` and `forge()`.
rec_roles <- recipe(train) %>%
    update_role(Sepal.Width, new_role = "predictor") %>%
    update_role(Species, new_role = "outcome") %>%
    update_role(Sepal.Length, new_role = "custom_role") %>%
    update_role(Petal.Length, new_role = "custom_role2")

processed_roles <- mold(rec_roles, train)

processed_roles$extras

forge(test, processed_roles$blueprint)
# Matrix output for predictors

# You can change the 'composition' of the predictor data set
bp <- default_recipe_blueprint(composition = "dgCMatrix")
processed <- mold(rec, train, blueprint = bp)
class(processed$predictors)

---

**default_xy_blueprint**  
*Default XY blueprint*

---

**Description**

This page holds the details for the XY preprocessing blueprint. This is the blueprint used by default from `mold()` if x and y are provided separately (i.e. the XY interface is used).

**Usage**

```r
default_xy_blueprint(
  intercept = FALSE,
  allow_novel_levels = FALSE,
  composition = "tibble"
)
```

```r
## S3 method for class 'data.frame'
mold(x, y, ..., blueprint = NULL)
```

```r
## S3 method for class 'matrix'
mold(x, y, ..., blueprint = NULL)
```

**Arguments**

- **intercept**  
  A logical. Should an intercept be included in the processed data? This information is used by the `process` function in the `mold` and `forge` function list.

- **allow_novel_levels**  
  A logical. Should novel factor levels be allowed at prediction time? This information is used by the `clean` function in the `forge` function list, and is passed on to `scream()`.

- **composition**  
  Either "tibble", "matrix", or "dgCMatrix" for the format of the processed predictors. If "matrix" or "dgCMatrix" are chosen, all of the predictors must be numeric after the preprocessing method has been applied; otherwise an error is thrown.

- **x**  
  A data frame or matrix containing the predictors.

- **y**  
  A data frame, matrix, or vector containing the outcomes.

- **...**  
  Not used.

- **blueprint**  
  A preprocessing blueprint. If left as NULL, then a `default_xy_blueprint()` is used.
Details

As documented in `standardize()`, if `y` is a `vector`, then the returned outcomes tibble has 1 column with a standardized name of ".outcome".

The one special thing about the XY method’s forge function is the behavior of `outcomes = TRUE` when a `vector` `y` value was provided to the original call to `mold()`. In that case, `mold()` converts `y` into a tibble, with a default name of `.outcome`. This is the column that `forge()` will look for in `new_data` to preprocess. See the examples section for a demonstration of this.

Value

For `default_xy_blueprint()`, an XY blueprint.

Mold

When `mold()` is used with the default xy blueprint:

- It converts `x` to a tibble.
- It adds an intercept column to `x` if `intercept = TRUE`.
- It runs `standardize()` on `y`.

Forge

When `forge()` is used with the default xy blueprint:

- It calls `shrink()` to trim `new_data` to only the required columns and coerce `new_data` to a tibble.
- It calls `scream()` to perform validation on the structure of the columns of `new_data`.
- It adds an intercept column onto `new_data` if `intercept = TRUE`.

Examples

```r
# ---------------------------------------------------------------------------
# Setup

train <- iris[1:100,]
test <- iris[101:150,]

train_x <- train[, "Sepal.Length", drop = FALSE]
train_y <- train[, "Species", drop = FALSE]

test_x <- test[, "Sepal.Length", drop = FALSE]
test_y <- test[, "Species", drop = FALSE]

# ---------------------------------------------------------------------------
# XY Example

# First, call mold() with the training data
processed <- mold(train_x, train_y)

# Then, call forge() with the blueprint and the test data
# to have it preprocess the test data in the same way
forge(test_x, processed$blueprint)
```
# Intercept

processed <- mold(train_x, train_y, blueprint = default_xy_blueprint(intercept = TRUE))

forge(test_x, processed$blueprint)

# XY Method and forge(outcomes = TRUE)

# You can request that the new outcome columns are preprocessed as well, but
# they have to be present in `new_data`!

processed <- mold(train_x, train_y)

# Can't do this!
try(forge(test_x, processed$blueprint, outcomes = TRUE))

# Need to use the full test set, including `y`
forge(test, processed$blueprint, outcomes = TRUE)

# With the XY method, if the Y value used in `mold()` is a vector,
# then a column name of `.outcome` is automatically generated.
# This name is what forge() looks for in `new_data`.

# Y is a vector!
y_vec <- train_y$Species

processed_vec <- mold(train_x, y_vec)

# This throws an informative error that tell you
# to include an `".outcome"` column in `new_data`.
try(forge(iris, processed_vec$blueprint, outcomes = TRUE))

test2 <- test
test2$.outcome <- test2$Species
test2$Species <- NULL

# This works, and returns a tibble in the $outcomes slot
forge(test2, processed_vec$blueprint, outcomes = TRUE)

# Matrix output for predictors

bp <- default_xy_blueprint(composition = "dgCMatrix")
processed <- mold(train_x, train_y, blueprint = bp)
class(processed$predictors)
### Description

`delete_response()` is exactly the same as `delete.response()`, except that it fixes a long standing bug by also removing the part of the "dataClasses" attribute corresponding to the response, if it exists.

### Usage

```r
delete_response(terms)
```

### Arguments

- **terms**: A terms object.

### Details

The bug is described here:


### Value

- **terms** with the response sections removed.

### Examples

```r
framed <- model.frame(Species ~ Sepal.Width, iris)
attr(delete.response(framed$terms), "dataClasses")
attr(delete_response(framed$terms), "dataClasses")
```

---

### forge

**Forge prediction-ready data**

### Description

`forge()` applies the transformations requested by the specific blueprint on a set of `new_data`. This `new_data` contains new predictors (and potentially outcomes) that will be used to generate predictions.

All blueprints have consistent return values with the others, but each is unique enough to have its own help page. Click through below to learn how to use each one in conjunction with `forge()`.

- XY Method - `default_xyBlueprint()`
- Formula Method - `default_formulaBlueprint()`
- Recipes Method - `default_recipeBlueprint()`

### Usage

```r
forge(new_data, blueprint, ..., outcomes = FALSE)
```
get_data_classes

Arguments

- **new_data**: A data frame or matrix of predictors to process. If `outcomes = TRUE`, this should also contain the outcomes to process.

- **blueprint**: A preprocessing blueprint.

- **...**: Not used.

- **outcomes**: A logical. Should the outcomes be processed and returned as well?

Details

If the outcomes are present in `new_data`, they can optionally be processed and returned in the `outcomes` slot of the returned list by setting `outcomes = TRUE`. This is very useful when doing cross validation where you need to preprocess the outcomes of a test set before computing performance.

Value

A named list with 3 elements:

- **predictors**: A tibble containing the preprocessed `new_data` predictors.

- **outcomes**: If `outcomes = TRUE`, a tibble containing the preprocessed outcomes found in `new_data`. Otherwise, NULL.

- **extras**: Either NULL if the blueprint returns no extra information, or a named list containing the extra information.

Examples

```r
# See the blueprint specific documentation linked above
# for various ways to call forge with different
# blueprints.

train <- iris[1:100,]
test <- iris[101:150,]

# Formula
processed <- mold(
  log(Sepal.Width) ~ Species,
  train,
  blueprint = default_formula_blueprint(indicators = "none")
)

forge(test, processed$blueprint, outcomes = TRUE)
```

Description

When predicting from a model, it is often important for the `new_data` to have the same classes as the original data used to fit the model. `get_data_classes()` extracts the classes from the original training data.
Usage

get_data_classes(data)

Arguments

data A data frame or matrix.

Value

A named list. The names are the column names of data and the values are character vectors containing the class of that column.

Examples

gt_data_classes(iris)

gt_data_classes(as.matrix(mtcars))

# Unlike .MFclass(), the full class vector is returned
data <- data.frame(col = ordered(c("a", "b")))

.MFclass(data$col)

gt_data_classes(data)

get_levels

Extract factor levels from a data frame

Description

gt_levels() extracts the levels from any factor columns in data. It is mainly useful for extracting the original factor levels from the predictors in the training set. gt_outcome_levels() is a small wrapper around get_levels() for extracting levels from a factor outcome that first calls standardize() on y.

Usage

gt_levels(data)

gt_outcome_levels(y)

Arguments

data A data.frame to extract levels from.

y The outcome. This can be:

- A factor vector
- A numeric vector
- A 1D numeric array
- A numeric matrix with column names
- A 2D numeric array with column names
- A data frame with numeric or factor columns
Value

A named list with as many elements as there are factor columns in `data` or `y`. The names are the names of the factor columns, and the values are character vectors of the levels.

If there are no factor columns, `NULL` is returned.

See Also

`stats::.getXlevels()`

Examples

```r
# Factor columns are returned with their levels
get_levels(iris)

# No factor columns
get_levels(mtcars)

# standardize() is first run on 'y'
# which converts the input to a data frame
# with an automatically named column, "outcome"
get_outcome_levels(y = factor(letters[1:5]))
```

---

**hardhat-example-data**  
*Example data for hardhat*

Description

Example data for hardhat

Details

Data objects for a training and test set with the same variables: three numeric and two factor columns.

Value

`example_train, example_test`  
tibbles

Examples

`data("hardhat-example-data")`
Description

These generics are used to extract elements from various model objects. Methods are defined in other packages, such as tune, workflows, and workflowsets, but the returned object is always the same.

- `extract_fit_engine()` returns the engine specific fit embedded within a parsnip model fit. For example, when using `parsnip::linear_reg()` with the "lm" engine, this returns the underlying lm object.
- `extract_fit_parsnip()` returns a parsnip model fit.
- `extract_mold()` returns the preprocessed "mold" object returned from `mold()`. It contains information about the preprocessing, including either the prepped recipe, the formula terms object, or variable selectors.
- `extract_spec_parsnip()` returns a parsnip model specification.
- `extract_preprocessor()` returns the formula, recipe, or variable expressions used for preprocessing.
- `extract_recipe()` returns a recipe, possibly estimated.
- `extract_workflow()` returns a workflow, possibly fit.

Usage

```r
extract_workflow(x, ...)
extract_recipe(x, ...)
extract_spec_parsnip(x, ...)
extract_fit_parsnip(x, ...)
extract_fit_engine(x, ...)
extract_mold(x, ...)
extract_preprocessor(x, ...)
```

Arguments

- `x` An object.
- `...` Extra arguments passed on to methods.

Examples

```r
# See packages where methods are defined for examples, such as 'parsnip' or
# 'workflows'.
```
is_blueprint

Is x a preprocessing blueprint?

Description

is_blueprint() checks if x inherits from "hardhat_blueprint".

Usage

is_blueprint(x)

Arguments

x An object.

Examples

is_blueprint(default_xy_blueprint())

model_frame

Construct a model frame

Description

model_frame() is a stricter version of stats::model.frame(). There are a number of differences, with the main being that rows are never dropped and the return value is a list with the frame and terms separated into two distinct objects.

Usage

model_frame(formula, data)

Arguments

formula A formula or terms object representing the terms of the model frame.
data A data frame or matrix containing the terms of formula.

Details

The following explains the rationale for some of the difference in arguments compared to stats::model.frame:

- subset: Not allowed because the number of rows before and after model_frame() has been run should always be the same.
- na.action: Not allowed and is forced to "na.pass" because the number of rows before and after model_frame() has been run should always be the same.
- drop.unused.levels: Not allowed because it seems inconsistent for data and the result of model_frame() to ever have the same factor column but with different levels, unless specified through original_levels. If this is required, it should be done through a recipe step explicitly.
- `xlev`: Not allowed because this check should have been done ahead of time. Use `scream()` to check the integrity of `data` against a training set if that is required.
- `...`: Not exposed because offsets are handled separately, and it is not necessary to pass weights here any more because rows are never dropped (so weights don’t have to be subset alongside the rest of the design matrix). If other non-predictor columns are required, use the ”roles” features of recipes.

It is important to always use the results of `model_frame()` with `model_matrix()` rather than `stats::model.matrix()` because the tibble in the result of `model_frame()` does not have a terms object attached. If `model.matrix()` is called directly, then a call to `model.frame()` will be made automatically, which can give faulty results.

**Value**

A named list with two elements:

- "data": A tibble containing the model frame.
- "terms": A terms object containing the terms for the model frame.

**Examples**

```r
# Example usage
framed <- model_frame(Species ~ Sepal.Width, iris)
framed$data
framed$terms

# Missing values never result in dropped rows
iris2 <- iris
iris2$Sepal.Width[1] <- NA
framed2 <- model_frame(Species ~ Sepal.Width, iris2)
head(framed2$data)
nrow(framed2$data) == nrow(iris2)
```

---

**model_matrix**

**Description**

`model_matrix()` is a stricter version of `stats::model.matrix()`. Notably, `model_matrix()` will *never* drop rows, and the result will be a tibble.

**Usage**

`model_matrix(terms, data)`
Arguments

- **terms**: A terms object to construct a model matrix with. This is typically the terms object returned from the corresponding call to `model.frame()`.
- **data**: A tibble to construct the design matrix with. This is typically the tibble returned from the corresponding call to `model.frame()`.

Details

The following explains the rationale for some of the difference in arguments compared to `stats::model.matrix()`:

- **contrasts.arg**: Set the contrasts argument, `options("contrasts")` globally, or assign a contrast to the factor of interest directly using `stats::contrasts()`. See the examples section.
- **xlev**: Not allowed because `model.frame()` is never called, so it is unnecessary.
- **...**: Not allowed because the default method of `model.matrix()` does not use it, and the `lm` method uses it to pass potential offsets and weights through, which are handled differently in hardhat.

Value

A tibble containing the design matrix.

Examples

```r
# Example usage
framed <- model_frame(Sepal.Width ~ Species, iris)
model_matrix(framed$terms, framed$data)

# Missing values never result in dropped rows
iris2 <- iris
iris2$Species[1] <- NA
framed2 <- model_frame(Sepal.Width ~ Species, iris2)
model_matrix(framed2$terms, framed2$data)

# Default contrasts
y <- factor(c("a", "b"))
x <- data.frame(y = y)
framed <- model_frame(~y, x)

# Setting contrasts directly
y_with_contrast <- y
contrasts(y_with_contrast) <- contr.sum(2)
x2 <- data.frame(y = y_with_contrast)
framed2 <- model_frame(~y, x2)
```
# Compare!
model_matrix(framed$terms, framed$data)
model_matrix(framed2$terms, framed2$data)

# Also, can set the contrasts globally
global_override <- c(unordered = "contr.sum", ordered = "contr.poly")

rlang::with_options(.expr = {
    model_matrix(framed$terms, framed$data)
},
    contrasts = global_override
)

---

model_offset  
Extract a model offset

Description

model_offset() extracts a numeric offset from a model frame. It is inspired by stats::model.offset(), but has nicer error messages and is slightly stricter.

Usage

model_offset(terms, data)

Arguments

terms  
A "terms" object corresponding to data, returned from a call to model.frame().
data  
A data frame returned from a call to model.frame().

Details

If a column that has been tagged as an offset is not numeric, a nice error message is thrown telling you exactly which column was problematic.

stats::model.offset() also allows for a column named "(offset)" to be considered an offset along with any others that have been tagged by stats::offset(). However, stats::model.matrix() does not recognize these columns as offsets (so it doesn’t remove them as it should). Because of this inconsistency, columns named "(offset)" are not treated specially by model_offset().

Value

A numeric vector representing the offset.

Examples

x <- model.frame(Species ~ offset(Sepal.Width), iris)

model_offset(terms(x), x)
xx <- model.frame(Species ~ offset(Sepal.Width) + offset(Sepal.Length), iris)
model_offset(terms(xx), xx)

# Problematic columns are caught with intuitive errors
tryCatch(
  expr = {
    x <- model.frame(~ offset(Species), iris)
    model_offset(terms(x), x)
  },
  error = function(e) {
    print(e$message)
  }
)

---

mold  

Mold data for modeling

Description

mold() applies the appropriate processing steps required to get training data ready to be fed into a model. It does this through the use of various blueprints that understand how to preprocess data that come in various forms, such as a formula or a recipe.

All blueprints have consistent return values with the others, but each is unique enough to have its own help page. Click through below to learn how to use each one in conjunction with mold().

- XY Method - default_xy_blueprint()
- Formula Method - default_formula_blueprint()
- Recipes Method - default_recipe_blueprint()

Usage

mold(x, ...)

Arguments

x  
An object. See the method specific implementations linked in the Description for more information.

...  
Not used.

Value

A named list containing 4 elements:

- predictors: A tibble containing the molded predictors to be used in the model.
- outcome: A tibble containing the molded outcomes to be used in the model.
- blueprint: A method specific "hardhat_blueprint" object for use when making predictions.
- extras: Either NULL if the blueprint returns no extra information, or a named list containing the extra information.
Examples

# See the method specific documentation linked in Description
# for the details of each blueprint, and more examples.

# XY
mold(iris[, "Sepal.Width", drop = FALSE], iris$Species)

# Formula
mold(Species ~ Sepal.Width, iris)

# Recipe
library(recipes)
mold(recipe(Species ~ Sepal.Width, iris), iris)

new_default_formula_blueprint

Create a new default blueprint

Description

This page contains the constructors for the default blueprints. They can be extended if you
want to add extra behavior on top of what the default blueprints already do, but generally
you will extend the non-default versions of the constructors found in the documentation for
new_blueprint().

Usage

new_default_formula_blueprint(
mold,  
forge,  
intercept = FALSE,  
allow_novel_levels = FALSE,  
ptypes = NULL,  
formula = NULL,  
indicators = "traditional",  
composition = "tibble",  
terms = list(predictors = NULL, outcomes = NULL),  
...,  
subclass = character()
)

new_default_recipe_blueprint(
mold,  
forge,  
intercept = FALSE,  
allow_novel_levels = FALSE,  
fresh = TRUE,  
composition = "tibble",  
ptypes = NULL,  
recipe = NULL,  
extra_role_ptypes = NULL,
new_default_formula_blueprint

..., subclass = character()
)

new_default_xy_blueprint(
mold, forge, intercept = FALSE,
allow_novel_levels = FALSE,
composition = "tibble",
ptypes = NULL,
...., subclass = character()
)

Arguments

mold A named list with two elements, clean and process, see the new_blueprint() section, Mold Functions, for details.

forge A named list with two elements, clean and process, see the new_blueprint() section, Forge Functions, for details.

intercept A logical. Should an intercept be included in the processed data? This information is used by the process function in the mold and forge function list.

allow_novel_levels A logical. Should novel factor levels be allowed at prediction time? This information is used by the clean function in the forge function list, and is passed on to scream().

ptypes Either NULL, or a named list with 2 elements, predictors and outcomes, both of which are 0-row tibbles. ptypes is generated automatically at mold() time and is used to validate new_data at prediction time. At mold() time, the information found in blueprint$mold$process()$ptype is used to set ptypes for the blueprint.

formula Either NULL, or a formula that specifies how the predictors and outcomes should be preprocessed. This argument is set automatically at mold() time.

indicators A single character string. Control how factors are expanded into dummy variable indicator columns. One of:
  * "traditional" - The default. Create dummy variables using the traditional model.matrix() infrastructure. Generally this creates K -1 indicator columns for each factor, where K is the number of levels in that factor.
  * "none" - Leave factor variables alone. No expansion is done.
  * "one_hot" - Create dummy variables using a one-hot encoding approach that expands unordered factors into all K indicator columns, rather than K -1.

composition Either "tibble", "matrix", or "dgCMatrix" for the format of the processed predictors. If "matrix" or "dgCMatrix" are chosen, all of the predictors must be numeric after the preprocessing method has been applied; otherwise an error is thrown.
new_formula_blueprint

Description

These are the base classes for creating new preprocessing blueprints. All blueprints inherit from the one created by `new.blueprint()`, and the default method specific blueprints inherit from the other three here.

If you want to create your own processing blueprint for a specific method, generally you will subclass one of the method specific blueprints here. If you want to create a completely new preprocessing blueprint for a totally new preprocessing method (i.e. not the formula, xy, or recipe method) then you should subclass `new.blueprint()`.

Usage

```r
new_formula_blueprint(
  mold,
  forge,
  intercept = FALSE,
  allow_novel_levels = FALSE,
  ptypes = NULL,
  formula = NULL,
  indicators = "traditional",
  composition = "tibble",
  ...,
  subclass = character()
)
```

```r
new_recipe_blueprint(
  mold,
  forge,
  intercept = FALSE,
  allow_novel_levels = FALSE,
  fresh = TRUE,
  ...,
  ...,
)```
new_formula_blueprint

composition = "tibble",
ptypes = NULL,
recipe = NULL,
...
subclass = character()
)

new_xy_blueprint(
mold,
forge,
intercept = FALSE,
allow_novel_levels = FALSE,
composition = "tibble",
ptypes = NULL,
...
subclass = character()
)

new_blueprint(
mold,
forge,
intercept = FALSE,
allow_novel_levels = FALSE,
composition = "tibble",
ptypes = NULL,
...
subclass = character()
)

Arguments

mold A named list with two elements, clean and process, see the new_blueprint() section, Mold Functions, for details.
forge A named list with two elements, clean and process, see the new_blueprint() section, Forge Functions, for details.
intercept A logical. Should an intercept be included in the processed data? This information is used by the process function in the mold and forge function list.
allow_novel_levels A logical. Should novel factor levels be allowed at prediction time? This information is used by the clean function in the forge function list, and is passed on to scream().
ptypes Either NULL, or a named list with 2 elements, predictors and outcomes, both of which are 0-row tibbles. ptypes is generated automatically at mold() time and is used to validate new_data at prediction time. At mold() time, the information found in blueprint$mold$process()$ptype is used to set ptypes for the blueprint.
formula Either NULL, or a formula that specifies how the predictors and outcomes should be preprocessed. This argument is set automatically at mold() time.
indicators A single character string. Control how factors are expanded into dummy variable indicator columns. One of:
The default. Create dummy variables using the traditional `model.matrix()` infrastructure. Generally this creates \( K -1 \) indicator columns for each factor, where \( K \) is the number of levels in that factor.

- "none" - Leave factor variables alone. No expansion is done.
- "one-hot" - Create dummy variables using a one-hot encoding approach that expands unordered factors into all \( K \) indicator columns, rather than \( K -1 \).

composition

Either "tibble", "matrix", or "dgCMatrix" for the format of the processed predictors. If "matrix" or "dgCMatrix" are chosen, all of the predictors must be numeric after the preprocessing method has been applied; otherwise an error is thrown.

... Name-value pairs for additional elements of blueprints that subclass this blueprint.

subclass A character vector. The subclasses of this blueprint.

fresh Should already trained operations be re-trained when `prep()` is called?

recipe Either NULL, or an unprepped recipe. This argument is set automatically at `mold()` time.

Value

A preprocessing blueprint, which is a list containing the inputs used as arguments to the function, along with a class specific to the type of blueprint being created.

Mold Functions

`blueprint$mold` should be a named list with two elements, both of which are functions:

- clean: A function that performs initial cleaning of the user’s input data to be used in the model.
  
  - Arguments:
    
    - If this is an xy blueprint, `blueprint`, `x` and `y`.
    - Otherwise, `blueprint` and `data`.
  
  - Output: A named list of three elements:
    
    - `blueprint`: The blueprint, returned and potentially updated.
    - `x`: The cleaned predictor data.
    - `y`: The cleaned outcome data.

- process: A function that performs the actual preprocessing of the data.

  - Arguments:
    
    - If this is an xy blueprint, `blueprint`, `x` and `y`.
    - Otherwise, `blueprint` and `data`.
  
  - Output: A named list of 5 elements:
    
    - `blueprint`: The blueprint, returned and potentially updated.
    - `predictors`: A tibble of predictors.
    - `outcomes`: A tibble of outcomes.
**new_model**

- **ptypes**: A named list with 2 elements, `predictors` and `outcomes`, where both elements are 0-row tibbles.
- **extras**: Varies based on the blueprint. If the blueprint has no extra information, `NULL`. Otherwise a named list of the extra elements returned by the blueprint.

Both `blueprint$mold$clean()` and `blueprint$mold$process()` will be called, in order, from `mold()`.

**Forge Functions**

`blueprint$forge` should be a named list with two elements, both of which are functions:

- **clean**: A function that performs initial cleaning of `new_data`:
  - **Arguments**:
    - `blueprint`, `new_data`, and `outcomes`.
  - **Output**: A named list of the following elements:
    - `blueprint`: The blueprint, returned and potentially updated.
    - `predictors`: A tibble containing the cleaned predictors.
    - `outcomes`: A tibble containing the cleaned outcomes.
    - `extras`: A named list of any extras obtained while cleaning. These are passed on to the `process()` function for further use.

- **process**: A function that performs the actual preprocessing of the data using the known information in the `blueprint`.
  - **Arguments**:
    - `blueprint`, `new_data`, `outcomes`, `extras`.
  - **Output**: A named list of the following elements:
    - `blueprint`: The blueprint, returned and potentially updated.
    - `predictors`: A tibble of the predictors.
    - `outcomes`: A tibble of the outcomes, or `NULL`.
    - `extras`: Varies based on the blueprint. If the blueprint has no extra information, `NULL`. Otherwise a named list of the extra elements returned by the blueprint.

Both `blueprint$forge$clean()` and `blueprint$forge$process()` will be called, in order, from `forge()`.

---

### Description

A **model** is a scalar object, as classified in *Advanced R*. As such, it takes uniquely named elements in `...` and combines them into a list with a class of `class`. This entire object represent a single model.

### Usage

```r
new_model(..., blueprint = default_xy_blueprint(), class = character())
```
refresh_blueprint

Arguments

... Name-value pairs for elements specific to the model defined by class.
blueprint A preprocessing blueprint returned from a call to mold().
class A character vector representing the class of the model.

Details

Because every model should have multiple interfaces, including formula and recipes interfaces, all models should have a blueprint that can process new data when predict() is called. The easiest way to generate an blueprint with all of the information required at prediction time is to use the one that is returned from a call to mold().

Value

A new scalar model object, represented as a classed list with named elements specified in ...

Examples

new_model(
  custom_element = "my-elem",
  blueprint = default_xy_blueprint(),
  class = "custom_model"
)

refresh_blueprint

Refresh a preprocessing blueprint

Description

refresh_blueprint() is a developer facing generic function that is called at the end of update_blueprint(). It simply is a wrapper around the method specific new_*_blueprint() function that runs the updated blueprint through the constructor again to ensure that all of the elements of the blueprint are still valid after the update.

Usage

refresh_blueprint(blueprint)

Arguments

blueprint A preprocessing blueprint.

Details

If you implement your own custom blueprint, you should export a refresh_blueprint() method that just calls the constructor for your blueprint and passes through all of the elements of the blueprint to the constructor.

Value

blueprint is returned after a call to the corresponding constructor.
Examples

```r
blueprint <- default_xy_blueprint()

# This should never be done manually, but is essentially
# what `update_blueprint(blueprint, intercept = TRUE)` does for you
blueprint$intercept <- TRUE

# Then update_blueprint() will call refresh_blueprint()
# to ensure that the structure is correct
refresh_blueprint(blueprint)

# So you can’t do something like...
blueprint_bad <- blueprint
blueprint_bad$intercept <- 1

# …because the constructor will catch it
try(refresh_blueprint(blueprint_bad))

# And update_blueprint() catches this automatically
try(update_blueprint(blueprint, intercept = 1))
```

---

**run_mold**  
*Call mold$clean() and mold$process()*

**Description**

This is a purely developer facing function, that is *only* used if you are creating a completely new blueprint inheriting only from `new_blueprint()`, and not from one of the more common:  
`new_xy_blueprint()`, `new_recipe_blueprint()`, `new_formula_blueprint()`.  

**Usage**

```r
run_mold(blueprint, ...)
```

**Arguments**

- **blueprint** A preprocessing blueprint.
- **...** Not used. Required for extensibility.

**Details**

Because `mold()` has different interfaces (like XY and formula), which require different arguments (x and y vs data), their corresponding blueprints also have different arguments for the `blueprint$mold$clean()` and `blueprint$mold$process()` functions. The sole job of `run_mold()` is simply to call these two functions with the right arguments.  

The only time you need to implement a method for `run_mold()` is if you are creating a `new_blueprint()` that does not follow one of the three core blueprint types. In that special case, create a method for `run_mold()` with your blueprint type, and pass through whatever arguments are necessary to call your blueprint specific `clean()` and `process()` functions.
If you go this route, you will also need to create a `mold()` method if `x` is not a data frame / matrix, recipe, or formula. If `x` is one of those types, then `run_mold()` will be called for you by the existing `mold()` method, you just have to supply the `run_mold()` method for your blueprint.

**Value**

The preprocessed result, as a named list.

---

### scream

**Description**

`scream()` ensures that the structure of `data` is the same as prototype, `ptype`. Under the hood, `vctrs::vec_cast()` is used, which casts each column of `data` to the same type as the corresponding column in `ptype`.

This casting enforces a number of important structural checks, including but not limited to:

- **Data Classes** - Checks that the class of each column in `data` is the same as the corresponding column in `ptype`.
- **Novel Levels** - Checks that the factor columns in `data` don’t have any new levels when compared with the `ptype` columns. If there are new levels, a warning is issued and they are coerced to `NA`. This check is optional, and can be turned off with `allow_novel_levels = TRUE`.
- **Level Recovery** - Checks that the factor columns in `data` aren’t missing any factor levels when compared with the `ptype` columns. If there are missing levels, then they are restored.

**Usage**

`scream(data, ptype, allow_novel_levels = FALSE)`

**Arguments**

- `data` A data frame containing the new data to check the structure of.
- `ptype` A data frame prototype to cast `data` to. This is commonly a 0-row slice of the training set.
- `allow_novel_levels` Should novel factor levels in `data` be allowed? The safest approach is the default, which throws a warning when novel levels are found, and coerces them to `NA` values. Setting this argument to `TRUE` will ignore all novel levels. This argument does not apply to ordered factors. Novel levels are not allowed in ordered factors because the level ordering is a critical part of the type.
Details

`scream()` is called by `forge()` after `shrink()` but before the actual processing is done. Generally, you don’t need to call `scream()` directly, as `forge()` will do it for you.

If `scream()` is used as a standalone function, it is good practice to call `shrink()` right before it as there are no checks in `scream()` that ensure that all of the required column names actually exist in `data`. Those checks exist in `shrink()`.

Value

A tibble containing the required columns after any required structural modifications have been made.

Factor Levels

`scream()` tries to be helpful by recovering missing factor levels and warning about novel levels. The following graphic outlines how `scream()` handles factor levels when coercing from a column in `data` to a column in `ptype`.

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Factor</th>
<th>Ordered</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Same levels</td>
<td>New levels (allowed)</td>
<td>New levels (prohibited)</td>
</tr>
<tr>
<td>Factor</td>
<td>No change</td>
<td>No change</td>
<td>Missing: New levels coerced to NA</td>
</tr>
<tr>
<td>Ordered</td>
<td>Error</td>
<td>Error</td>
<td>Error</td>
</tr>
<tr>
<td>Character</td>
<td>Coerce to factor</td>
<td>Coerce to factor</td>
<td>Coerce to factor</td>
</tr>
</tbody>
</table>

Note that ordered factor handing is much stricter than factor handling. Ordered factors in `data` must have exactly the same levels as ordered factors in `ptype`.

Examples

```r
# Setup
train <- iris[1:100,]
test <- iris[101:150,]

# mold() is run at model fit time
# and a formula preprocessing blueprint is recorded
x <- mold(log(Sepal.Width) ~ Species, train)

# Inside the result of mold() are the prototype tibbles
# for the predictors and the outcomes
ptype_pred <- x$blueprint$otypes$predictors
ptype_out <- x$blueprint$otypes$outcomes

# Pass the test data, along with a prototype, to
# shrink() to extract the prototype columns
test_shrunk <- shrink(test, ptype_pred)
```
# Now pass that to `scream()` to perform validation checks
# If no warnings / errors are thrown, the checks were
# successful!
scream(test_shrunk, ptype_pred)

# ---------------------------------------------------------------
# Outcomes

# To also extract the outcomes, use the outcome prototype
test_outcome <- shrink(test, ptype_out)
scream(test_outcome, ptype_out)

# ---------------------------------------------------------------
# Casting

# `scream()` uses `vctrs::vec_cast()` to intelligently convert
# new data to the prototype automatically. This means
# it can automatically perform certain conversions, like
# coercing character columns to factors.
test2 <- test
test2$Species <- as.character(test2$Species)

test2_shrunk <- shrink(test2, ptype_pred)
scream(test2_shrunk, ptype_pred)

# It can also recover missing factor levels.
# For example, it is plausible that the test data only had the
# "virginica" level
test3 <- test
test3$Species <- factor(test3$Species, levels = "virginica")

test3_shrunk <- shrink(test3, ptype_pred)
test3_fixed <- scream(test3_shrunk, ptype_pred)

# `scream()` recovered the missing levels
levels(test3_fixed$Species)

# ---------------------------------------------------------------
# Novel levels

# When novel levels with any data are present in `data`, the default
# is to coerce them to `NA` values with a warning.
test4 <- test
test4$Species <- as.character(test4$Species)
test4$Species[1] <- "new_level"

test4$Species <- factor(
  test4$Species,  # Modify to factor
  levels = c(levels(test$Species), "new_level")
)

test4 <- shrink(test4, ptype_pred)

# Warning is thrown
test4_removed <- scream(test4, ptype_pred)

# Novel level is removed
```r
levels(test4_removed$Species)

# No warning is thrown
test4_kept <- scream(test4, ptype_pred, allow_novel_levels = TRUE)

# Novel level is kept
levels(test4_kept$Species)
```

### Description

`shrink()` subsets data to only contain the required columns specified by the prototype, `ptype`.

### Usage

```r
shrink(data, ptype)
```

### Arguments

- `data`: A data frame containing the data to subset.
- `ptype`: A data frame prototype containing the required columns.

### Details

`shrink()` is called by `forge()` before `scream()` and before the actual processing is done.

### Value

A tibble containing the required columns.

### Examples

```r
# ---------------------------------------------------------------------------
# Setup

train <- iris[1:100,]
test <- iris[101:150,]

# ---------------------------------------------------------------------------
# shrink()

# mold() is run at model fit time
# and a formula preprocessing blueprint is recorded
x <- mold(log(Sepal.Width) ~ Species, train)

# Inside the result of mold() are the prototype tibbles
# for the predictors and the outcomes
ptype_pred <- x$blueprint$ptypes$predictors
ptype_out <- x$blueprint$ptypes$outcomes
```
Pass the test data, along with a prototype, to shrink() to extract the prototype columns

shrink(test, ptype_pred)

To extract the outcomes, just use the outcome prototype

shrink(test, ptype_out)

shrink() makes sure that the columns required by `ptype` actually exist in the data and errors nicely when they don’t.

test2 <- subset(test, select = -Species)
try(shrink(test2, ptype_pred))

---

### spruce

**Spruce up predictions**

**Description**
The family of spruce.*() functions convert predictions into a standardized format. They are generally called from a prediction implementation function for the specific type of prediction to return.

**Usage**

- spruce_numeric(pred)
- spruce_class(pred_class)
- spruce_prob(pred_levels, prob_matrix)

**Arguments**

- `pred` *(type = "numeric")* A numeric vector of predictions.
- `pred_class` *(type = "class")* A factor of "hard" class predictions.
- `pred_levels, prob_matrix` *(type = "prob")*
  - `pred_levels` should be a character vector of the original levels of the outcome used in training.
  - `prob_matrix` should be a numeric matrix of class probabilities with as many columns as levels in `pred_levels`, and in the same order.

**Details**

After running a spruce.*() function, you should always use the validation function `validate_prediction_size()` to ensure that the number of rows being returned is the same as the number of rows in the input `new_data`.

**Value**

A tibble, ideally with the same number of rows as the `new_data` passed to `predict()`. The column names and number of columns vary based on the function used, but are standardized.
standardize

**Standardize the outcome**

**Description**

Most of the time, the input to a model should be flexible enough to capture a number of different input types from the user. `standardize()` focuses on capturing the flexibility in the outcome.

**Usage**

```r
standardize(y)
```

**Arguments**

- **y**
  - The outcome. This can be:
    - A factor vector
    - A numeric vector
    - A 1D numeric array
    - A numeric matrix with column names
    - A 2D numeric array with column names
    - A data frame with numeric or factor columns

**Details**

`standardize()` is called from `mold()` when using an XY interface (i.e. a `y` argument was supplied).

**Value**

All possible values of `y` are transformed into a tibble for standardization. Vectors are transformed into a tibble with a single column named ".outcome".

**Examples**

```r
standardize(1:5)
standardize(factor(letters[1:5]))
mat <- matrix(1:10, ncol = 2)
colnames(mat) <- c("a", "b")
standardize(mat)
df <- data.frame(x = 1:5, y = 6:10)
standardize(df)
```
update_blueprint  |  Update a preprocessing blueprint

Description

update_blueprint() is the correct way to alter elements of an existing blueprint object. It has two benefits over just doing blueprint$elem <- new_elem.

- The name you are updating must already exist in the blueprint. This prevents you from accidentally updating non-existent elements.
- The constructor for the blueprint is automatically run after the update by refresh_blueprint() to ensure that the blueprint is still valid.

Usage

update_blueprint(blueprint, ...)

Arguments

- blueprint  An existing preprocessing blueprint.
- ...  Name-value pairs of existing elements in blueprint that should be updated.

Examples

```r
blueprint <- default_xy_blueprint()
  # 'intercept' defaults to FALSE
  blueprint
  update_blueprint(blueprint, intercept = TRUE)
  # Can't update non-existent elements
  try(update_blueprint(blueprint, intercept = TRUE))
  # Can't add non-valid elements
  try(update_blueprint(blueprint, intercept = 1))
```

validate_column_names  |  Ensure that data contains required column names

Description

validate - asserts the following:

- The column names of data must contain all original_names.

check - returns the following:

- ok  A logical. Does the check pass?
- missing_names  A character vector. The missing column names.
validate_column_names

Usage

validate_column_names(data, original_names)

check_column_names(data, original_names)

Arguments

data A data frame to check.

original_names A character vector. The original column names.

Details

A special error is thrown if the missing column is named ".outcome". This only happens in the case where `mold()` is called using the xy-method, and a `vector` y value is supplied rather than a data frame or matrix. In that case, y is coerced to a data frame, and the automatic name ".outcome" is added, and this is what is looked for in `forge()`. If this happens, and the user tries to request outcomes using `forge(..., outcomes = TRUE)` but the supplied `new_data` does not contain the required ".outcome" column, a special error is thrown telling them what to do. See the examples!

Value

validate_column_names() returns `data` invisibly.

check_column_names() returns a named list of two components, `ok`, and `missing_names`.

Validation

hardhat provides validation functions at two levels.

- `check_*()`: check a condition, and return a list. The list always contains at least one element, `ok`, a logical that specifies if the check passed. Each check also has check specific elements in the returned list that can be used to construct meaningful error messages.
- `validate_*()`: check a condition, and error if it does not pass. These functions call their corresponding check function, and then provide a default error message. If you, as a developer, want a different error message, then call the check_*() function yourself, and provide your own validation function.

See Also

Other validation functions: `validate_no_formula_duplication()`, `validate_outcomes_are_binary()`, `validate_outcomes_are_factors()`, `validate_outcomes_are_numeric()`, `validate_outcomes_are_univariate()`, `validate_prediction_size()`, `validate_predictors_are_numeric()`

Examples

# ---------------------------------------------------------------------------

original_names <- colnames(mtcars)

test <- mtcars

bad_test <- test[, -c(3, 4)]

# All good
check_column_names(test, original_names)

# Missing 2 columns
check_column_names(bad_test, original_names)

# Will error
try(validate_column_names(bad_test, original_names))

# Special error when `.outcome` is missing

train <- iris[1:100,]
test <- iris[101:150,]

train_x <- subset(train, select = -Species)
train_y <- train$Species

# Here, y is a vector
processed <- mold(train_x, train_y)

# So the default column name is `".outcome``
processed$outcomes

# It doesn't affect forge() normally
forge(test, processed$blueprint)

# But if the outcome is requested, and `".outcome``
# is not present in `new_data`, an error is thrown
# with very specific instructions
try(forge(test, processed$blueprint, outcomes = TRUE))

# To get this to work, just create an `.outcome` column in new_data
test$.outcome <- test$Species
forge(test, processed$blueprint, outcomes = TRUE)

---

validate_no_formula_duplication

Ensure no duplicate terms appear in formula

Description

validate - asserts the following:

- formula must not have duplicates terms on the left and right hand side of the formula.

check - returns the following:

- ok A logical. Does the check pass?
- duplicates A character vector. The duplicate terms.
validate_no_formula_duplication

Usage

validate_no_formula_duplication(formula, original = FALSE)

check_no_formula_duplication(formula, original = FALSE)

Arguments

- **formula**: A formula to check.
- **original**: A logical. Should the original names be checked, or should the names after processing be used? If FALSE, \( y \sim \log(y) \) is allowed because the names are "y" and "log(y)"; if TRUE, \( y \sim \log(y) \) is not allowed because the original names are both "y".

Value

- validate_no_formula_duplication() returns formula invisibly.
- check_no_formula_duplication() returns a named list of two components, ok and duplicates.

Validation

hardhat provides validation functions at two levels.

- **check.*()**: *check a condition, and return a list*. The list always contains at least one element, ok, a logical that specifies if the check passed. Each check also has check specific elements in the returned list that can be used to construct meaningful error messages.
- **validate.*()**: *check a condition, and error if it does not pass*. These functions call their corresponding check function, and then provide a default error message. If you, as a developer, want a different error message, then call the check.*() function yourself, and provide your own validation function.

See Also

Other validation functions: validate_column_names(), validate_outcomes_are_binary(), validate_outcomes_are_factors(), validate_outcomes_are_numeric(), validate_outcomes_are_univariate(), validate_prediction_size(), validate_predictors_are_numeric()

Examples

# All good
check_no_formula_duplication(y ~ x)

# Not good!
check_no_formula_duplication(y ~ y)

# This is generally okay
check_no_formula_duplication(y ~ log(y))

# But you can be more strict
check_no_formula_duplication(y ~ log(y), original = TRUE)

# This would throw an error
try(validate_no_formula_duplication(log(y) ~ log(y)))
### Description

validate - asserts the following:

- outcomes must have binary factor columns.

check - returns the following:

- ok A logical. Does the check pass?
- bad.cols A character vector. The names of the columns with problems.
- num.levels An integer vector. The actual number of levels of the columns with problems.

### Usage

```r
validate_outcomes_are_binary(outcomes)
check_outcomes_are_binary(outcomes)
```

### Arguments

- outcomes An object to check.

### Details

The expected way to use this validation function is to supply it the $outcomes element of the result of a call to mold().

### Value

- `validate_outcomes_are_binary()` returns outcomes invisibly.
- `check_outcomes_are_binary()` returns a named list of three components, ok, bad.cols, and num.levels.

### Validation

hardhat provides validation functions at two levels.

- check.*(): check a condition, and return a list. The list always contains at least one element, ok, a logical that specifies if the check passed. Each check also has check specific elements in the returned list that can be used to construct meaningful error messages.
- validate.*(): check a condition, and error if it does not pass. These functions call their corresponding check function, and then provide a default error message. If you, as a developer, want a different error message, then call the check.*() function yourself, and provide your own validation function.
See Also

Other validation functions: `validate_column_names()`, `validate_no_formula_duplication()`, `validate_outcomes_are_factors()`, `validate_outcomes_are_numeric()`, `validate_outcomes_are_univariate()`, `validate_prediction_size()`, `validate_predictors_are_numeric()`

Examples

```r
# Not a binary factor. 0 levels
check_outcomes_are_binary(data.frame(x = 1))

# Not a binary factor. 1 level
check_outcomes_are_binary(data.frame(x = factor("A")))

# All good
check_outcomes_are_binary(data.frame(x = factor(c("A", "B"))))
```

---

```r
validate_outcomes_are_factors

Ensure that the outcome has only factor columns
```

Description

validate - asserts the following:

- outcomes must have factor columns.

check - returns the following:

- ok A logical. Does the check pass?
- bad_classes A named list. The names are the names of problematic columns, and the values are the classes of the matching column.

Usage

```r
validate_outcomes_are_factors(outcomes)
check_outcomes_are_factors(outcomes)
```

Arguments

- outcomes An object to check.

Details

The expected way to use this validation function is to supply it the $outcomes element of the result of a call to mold().

Value

- `validate_outcomes_are_factors()` returns outcomes invisibly.
- `check_outcomes_are_factors()` returns a named list of two components, ok and bad_classes.
Validation

hardhat provides validation functions at two levels.

- `check.*()`: *check a condition, and return a list*. The list always contains at least one element, `ok`, a logical that specifies if the check passed. Each check also has check specific elements in the returned list that can be used to construct meaningful error messages.

- `validate.*()`: *check a condition, and error if it does not pass*. These functions call their corresponding check function, and then provide a default error message. If you, as a developer, want a different error message, then call the check.*() function yourself, and provide your own validation function.

See Also

Other validation functions: `validate_column_names()`, `validate_no_formula_duplication()`, `validate_outcomes_are_binary()`, `validate_outcomes_are_numeric()`, `validate_outcomes_are_univariate()`, `validate_prediction_size()`, `validate_predictors_are_numeric()`

Examples

```r
# Not a factor column.
check_outcomes_are_factors(data.frame(x = 1))

# All good
check_outcomes_are_factors(data.frame(x = factor(c("A", "B"))))
```

---

### validate_outcomes_are_numeric

*Ensure outcomes are all numeric*

---

Description

`validate` - asserts the following:

- `outcomes` must have numeric columns.

`check` - returns the following:

- `ok` A logical. Does the check pass?

- `bad_classes` A named list. The names are the names of problematic columns, and the values are the classes of the matching column.

Usage

```r
validate_outcomes_are_numeric(outcomes)

check_outcomes_are_numeric(outcomes)
```

Arguments

- `outcomes` An object to check.
validate_outcomes_are_univariate

Details
The expected way to use this validation function is to supply it the outcomes element of the result of a call to mold().

Value
validate_outcomes_are_numeric() returns outcomes invisibly.
check_outcomes_are_numeric() returns a named list of two components, ok and bad_classes.

Validation
hardhat provides validation functions at two levels.
• check_(): check a condition, and return a list. The list always contains at least one element, ok, a logical that specifies if the check passed. Each check also has check specific elements in the returned list that can be used to construct meaningful error messages.
• validate_(): check a condition, and error if it does not pass. These functions call their corresponding check function, and then provide a default error message. If you, as a developer, want a different error message, then call the check_() function yourself, and provide your own validation function.

See Also
Other validation functions: validate_column_names(), validate_no_formula_duplication(), validate_outcomes_are_binary(), validate_outcomes_are_factors(), validate_outcomes_are_univariate(), validate_prediction_size(), validate_predictors_are_numeric()

Examples
# All good
check_outcomes_are_numeric(mtcars)

# Species is not numeric
check_outcomes_are_numeric(iris)

# This gives an intelligent error message
try(validate_outcomes_are_numeric(iris))

validate_outcomes_are_univariate

Ensure that the outcome is univariate

Description
validate - asserts the following:
• outcomes must have 1 column. Atomic vectors are treated as 1 column matrices.
check - returns the following:
• ok A logical. Does the check pass?
• n_cols A single numeric. The actual number of columns.
validate_outcomes_are_univariate

Usage

validate_outcomes_are_univariate(outcomes)

check_outcomes_are_univariate(outcomes)

Arguments

outcomes
An object to check.

Details

The expected way to use this validation function is to supply it the $outcomes element of the result of a call to mold().

Value

validate_outcomes_are_univariate() returns outcomes invisibly.

check_outcomes_are_univariate() returns a named list of two components, ok and n_cols.

Validation

hardhat provides validation functions at two levels.

- check_*(): check a condition, and return a list. The list always contains at least one element, ok, a logical that specifies if the check passed. Each check also has check specific elements in the returned list that can be used to construct meaningful error messages.
- validate_*(): check a condition, and error if it does not pass. These functions call their corresponding check function, and then provide a default error message. If you, as a developer, want a different error message, then call the check_*() function yourself, and provide your own validation function.

See Also

Other validation functions: validate_column_names(), validate_no_formula_duplication(), validate_outcomes_are_binary(), validate_outcomes_are_factors(), validate_outcomes_are_numeric(), validate_prediction_size(), validate_predictors_are_numeric()

Examples

validate_outcomes_are_univariate(data.frame(x = 1))

try(validate_outcomes_are_univariate(mtcars))
validate_prediction_size

Ensure that predictions have the correct number of rows

Description

validate - asserts the following:

- The size of `pred` must be the same as the size of `new_data`.

check - returns the following:

- `ok` A logical. Does the check pass?
- `size_new_data` A single numeric. The size of `new_data`.
- `size_pred` A single numeric. The size of `pred`.

Usage

```r
validate_prediction_size(pred, new_data)
check_prediction_size(pred, new_data)
```

Arguments

- `pred` A tibble. The predictions to return from any prediction type. This is often created using one of the spruce functions, like `spruce_numeric()`.
- `new_data` A data frame of new predictors and possibly outcomes.

Details

This validation function is one that is more developer focused rather than user focused. It is a final check to be used right before a value is returned from your specific `predict()` method, and is mainly a "good practice" sanity check to ensure that your prediction blueprint always returns the same number of rows as `new_data`, which is one of the modeling conventions this package tries to promote.

Value

- `validate_prediction_size()` returns `pred` invisibly.
- `check_prediction_size()` returns a named list of three components, `ok`, `size_new_data`, and `size_pred`.

Validation

hardhat provides validation functions at two levels.

- `check_*()`: _check a condition, and return a list_. The list always contains at least one element, `ok`, a logical that specifies if the check passed. Each check also has check specific elements in the returned list that can be used to construct meaningful error messages.
- `validate_*()`: _check a condition, and error if it does not pass_. These functions call their corresponding check function, and then provide a default error message. If you, as a developer, want a different error message, then call the check_*() function yourself, and provide your own validation function.
See Also

Other validation functions: `validate_column_names()`, `validate_no_formula_duplication()`, `validate_outcomes_are_binary()`, `validate_outcomes_are_factors()`, `validate_outcomes_are_numeric()`, `validate_outcomes_are_univariate()`, `validate_predictors_are_numeric()`

Examples

```r
# Say new_data has 5 rows
new_data <- mtcars[1:5,]

# And somehow you generate predictions
# for those 5 rows
pred_vec <- 1:5

# Then you use `spruce_numeric()` to clean
# up these numeric predictions
pred <- spruce_numeric(pred_vec)

pred

# Use this check to ensure that
# the number of rows or pred match new_data
check_prediction_size(pred, new_data)

# An informative error message is thrown
# if the rows are different
try(validate_prediction_size(spruce_numeric(1:4), new_data))
```

---

`validate_predictors_are_numeric`  
Ensure predictors are all numeric

Description

- validate - asserts the following:
  - predictors must have numeric columns.
- check - returns the following:
  - ok A logical. Does the check pass?
  - bad_classes A named list. The names are the names of problematic columns, and the values are the classes of the matching column.

Usage

```r
validate_predictors_are_numeric(predictors)
```

```r
check_predictors_are_numeric(predictors)
```

Arguments

- predictors An object to check.
validate_predictors_are_numeric

Details

The expected way to use this validation function is to supply it the $predictors element of the result of a call to mold().

Value

validate_predictors_are_numeric() returns predictors invisibly.

cHECK_predictors_are_numeric() returns a named list of two components, ok, and bad_classes.

Validation

hardhat provides validation functions at two levels.

- check_*(): *check a condition, and return a list. The list always contains at least one element, ok, a logical that specifies if the check passed. Each check also has check specific elements in the returned list that can be used to construct meaningful error messages.
- validate_*(): *check a condition, and error if it does not pass. These functions call their corresponding check function, and then provide a default error message. If you, as a developer, want a different error message, then call the check_*() function yourself, and provide your own validation function.

See Also

Other validation functions: validate_column_names(), validate_no_formula_duplication(), validate_outcomes_are_binary(), validate_outcomes_are_factors(), validate_outcomes_are_numeric(), validate_outcomes_are_univariate(), validate_prediction_size()

Examples

# All good
check_predictors_are_numeric(mtcars)

# Species is not numeric
check_predictors_are_numeric(iris)

# This gives an intelligent error message
try(validate_predictors_are_numeric(iris))
Index

* datasets
  hardhat-example-data, 19

* validation functions
  validate_column_names, 40
  validate_no_formula_duplication, 42
  validate_outcomes_are_binary, 44
  validate_outcomes_are_factors, 45
  validate_outcomes_are_numeric, 46
  validate_outcomes_are_univariate, 47
  validate_prediction_size, 49
  validate_predictors_are_numeric, 50

  add_intercept_column, 4

  check_column_names
    (validate_column_names), 40
  check_no_formula_duplication
    (validate_no_formula_duplication), 42
  check_outcomes_are_binary
    (validate_outcomes_are_binary), 44
  check_outcomes_are_factors
    (validate_outcomes_are_factors), 45
  check_outcomes_are_numeric
    (validate_outcomes_are_numeric), 46
  check_outcomes_are_univariate
    (validate_outcomes_are_univariate), 47
  check_prediction_size
    (validate_prediction_size), 49
  check_predictors_are_numeric
    (validate_predictors_are_numeric), 50

  create_modeling_package
    (modeling-package), 2

  default_formula_blueprint, 4
  default_formula_blueprint(), 5, 16, 25
  default_recipe_blueprint, 10
  default_recipe_blueprint(), 11, 16, 25

  default_xy_blueprint, 13
  default_xy_blueprint(), 15, 16, 25
  delete_response, 15

  example_test (hardhat-example-data), 19
  example_train (hardhat-example-data), 19
  extract_fit_engine (hardhat-extract), 20
  extract_fit_parsnip (hardhat-extract), 20
  extract_mold (hardhat-extract), 20
  extract_preprocessor
    (hardhat-extract), 20
  extract_recipe (hardhat-extract), 20
  extract_spec_parsnip (hardhat-extract), 20
  extract_workflow (hardhat-extract), 20

  forge, 16
  forge(), 31, 35, 37, 41

  get_data_classes, 17
  get_levels, 18
  get_outcome_levels (get_levels), 18

  hardhat-example-data, 19
  hardhat-extract, 20

  is_blueprint, 21

  model.matrix(), 5, 27, 30
  model_frame, 21
  model_frame(), 23
  model_matrix, 22
  model_matrix(), 22
  model_offset, 24
  model_offset(), 6, 7
  modeling-package, 2
  mold, 25
  mold(), 7, 14, 20, 27–32, 39, 41, 44, 45, 47, 48, 51
  mold.data.frame (default_xy_blueprint), 13
mold.formula
  (default_formula_blueprint), 4
mold.matrix
  (default_xy_blueprint), 13
mold.recipe
  (default_recipe_blueprint), 10

new-blueprint (new_formula_blueprint), 28
new-default-blueprint
  (new_default_formula_blueprint), 26
new-blueprint
  (new_formula_blueprint), 28
new_default_recipe_blueprint, 26
new_recipe_blueprint
  (new_default_formula_blueprint), 26
new_default_xy_blueprint
  (new_default_formula_blueprint), 26
new_formula_blueprint, 28
new_formula_blueprint()
  (new_formula_blueprint), 28
new_model, 31
new_recipe_blueprint
  (new_formula_blueprint), 28
new_recipe_blueprint()
  (new_formula_blueprint), 33
new_xy_blueprint
  (new_formula_blueprint), 28
new_xy_blueprint()
  (new_formula_blueprint), 33

parsnip::linear_reg(), 20

recipes::bake(), 11
recipes::juice(), 11
recipes::prep(), 11
recipes::recipe(), 7, 11
refresh_blueprint, 32
run_mold, 33

scream, 34
scream(), 5, 6, 11, 13, 14, 22, 27, 29, 37
shrink, 37
shrink(), 6, 11, 14, 35
spruce, 38
spruce_class (spruce), 38
spruce_numeric (spruce), 38
spruce_numeric(), 49
spruce_prob (spruce), 38
standardize, 39
standardize(), 14, 18
stats::getXlevels(), 19
stats::contrasts(), 23

stats::lm(), 4
stats::model.frame(), 6, 7, 21
stats::model.matrix(), 6, 7, 22–24
stats::model.offset(), 24
stats::offset(), 6, 24
stats::poly(), 7

update_blueprint, 40
update_blueprint(), 32
use_modeling_deps
  (modeling-package), 2
use_modeling_files
  (modeling-package), 2

validate_column_names, 40, 43, 45–48, 50, 51
validate_no_formula_duplication, 41, 42, 45–48, 50, 51
validate_outcomes_are_binary, 41, 43, 44, 46–48, 50, 51
validate_outcomes_are_factors, 41, 43, 45, 47, 48, 50, 51
validate_outcomes_are_numeric, 41, 43, 45, 46, 48, 50, 51
validate_outcomes_are_univariate, 41, 43, 45–47, 47, 50, 51
validate_prediction_size, 41, 43, 45–48, 49, 51
validate_predictors_are_numeric, 41, 43, 45–48, 50, 50
vctrs::vec_cast(), 34