Package: workflows (via r-universe)

October 4, 2024

Title Modeling Workflows

Version 1.1.4.9000

Description Managing both a 'parsnip' model and a preprocessor, such as a model formula or recipe from 'recipes', can often be challenging. The goal of 'workflows' is to streamline this process by bundling the model alongside the preprocessor, all within the same object.

License MIT + file LICENSE

URL `https://github.com/tidymodels/workflows`,
`https://workflows.tidymodels.org`

BugReports `https://github.com/tidymodels/workflows/issues`

Depends R (>= 4.0)

Imports cli (>= 3.3.0), generics (>= 0.1.2), glue (>= 1.6.2), hardhat (>= 1.3.1.9000), lifecycle (>= 1.0.3), modelenv (>= 0.1.0), parsnip (>= 1.2.1.9000), rlang (>= 1.0.3), tidyselect (>= 1.2.0), sparsevctrs (>= 0.1.0.9002), vctrs (>= 0.4.1), withr

Suggests butcher (>= 0.2.0), tailor (>= 0.0.0.9001), covr, dials (>= 1.0.0), glmnet, knitr, magrittr, Matrix, methods, modeldata (>= 1.0.0), recipes (>= 1.0.10.9000), rmarkdown, testthat (>= 3.0.0)

VignetteBuilder knitr

Config/Needs/website dplyr, ggplot2, tidyr, tidyverse/tidytemplate, yardstick

Remotes tidymodels/recipes, tidymodels/parsnip, tidymodels/tailor, r-lib/sparsevctrs

Config/testthat/edition 3

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

Repository https://tidymodels.r-universe.dev

RemoteUrl https://github.com/tidymodels/workflows

RemoteRef HEAD

RemoteSha bc637fa6d64358a25592715573a827effde75eae

Contents

`add_case_weights` *Add case weights to a workflow*

Description

This family of functions revolves around selecting a column of `data` to use for *case weights*. This column must be one of the allowed case weight types, such as `hardhat::frequency_weights()` or `hardhat::importance_weights()`. Specifically, it must return `TRUE` from `hardhat::is_case_weights()`. The underlying model will decide whether or not the type of case weights you have supplied are applicable or not.

- `add_case_weights()` specifies the column that will be interpreted as case weights in the model. This column must be present in the `data` supplied to fit().
- `remove_case_weights()` removes the case weights. Additionally, if the model has already been fit, then the fit is removed.
- `update_case_weights()` first removes the case weights, then replaces them with the new ones.

Usage

```
add_case_weights(x, col)

remove_case_weights(x)

update_case_weights(x, col)
```

Arguments

`x` A workflow

`col` A single unquoted column name specifying the case weights for the model. This must be a classed case weights column, as determined by `hardhat::is_case_weights()`.

Details

For formula and variable preprocessors, the case weights `col` is removed from the data before the preprocessor is evaluated. This allows you to use formulas like `y ~ .` or tidyselection like `everything()` without fear of accidentally selecting the case weights column.

For recipe preprocessors, the case weights `col` is not removed and is passed along to the recipe. Typically, your recipe will include steps that can utilize case weights.

Examples

```
library(parsnip)
library(magrittr)
library(hardhat)

mtcars2 <- mtcars
mtcars2$gear <- frequency_weights(mtcars2$gear)

spec <- linear_reg() %>%
  set_engine("lm")

wf <- workflow() %>%
  add_case_weights(gear) %>%
  add_formula(mpg ~ .) %>%
  add_model(spec)

wf <- fit(wf, mtcars2)

# Notice that the case weights (gear) aren't included in the predictors
extract_mold(wf)$predictors

# Strip them out of the workflow, which also resets the model
remove_case_weights(wf)
```

add_formula *Add formula terms to a workflow*

Description

- add_formula() specifies the terms of the model through the usage of a formula.
- remove_formula() removes the formula as well as any downstream objects that might get created after the formula is used for preprocessing, such as terms. Additionally, if the model has already been fit, then the fit is removed.
- update_formula() first removes the formula, then replaces the previous formula with the new one. Any model that has already been fit based on this formula will need to be refit.

Usage

```
add_formula(x, formula, ..., blueprint = NULL)

remove_formula(x)

update_formula(x, formula, ..., blueprint = NULL)
```

Arguments

x A workflow

formula A formula specifying the terms of the model. It is advised to not do preprocessing in the formula, and instead use a recipe if that is required.

... Not used.

blueprint A hardhat blueprint used for fine tuning the preprocessing.

If NULL, hardhat::default_formula_blueprint() is used and is passed arguments that best align with the model present in the workflow.

Note that preprocessing done here is separate from preprocessing that might be done by the underlying model. For example, if a blueprint with indicators = "none" is specified, no dummy variables will be created by hardhat, but if the underlying model requires a formula interface that internally uses stats::model.matrix(), factors will still be expanded to dummy variables by the model.

Details

To fit a workflow, exactly one of add_formula(), add_recipe(), or add_variables() *must* be specified.

Value

x, updated with either a new or removed formula preprocessor.

Formula Handling

Note that, for different models, the formula given to add_formula() might be handled in different ways, depending on the parsnip model being used. For example, a random forest model fit using ranger would not convert any factor predictors to binary indicator variables. This is consistent with what ranger::ranger() would do, but is inconsistent with what stats::model.matrix() would do.

The documentation for parsnip models provides details about how the data given in the formula are encoded for the model if they diverge from the standard model.matrix() methodology. Our goal is to be consistent with how the underlying model package works.

How is this formula used?:

To demonstrate, the example below uses lm() to fit a model. The formula given to add_formula() is used to create the model matrix and that is what is passed to lm() with a simple formula of body_mass_g ~ .:

```
library(parsnip)
library(workflows)
library(magrittr)
library(modeldata)
library(hardhat)

data(penguins)

lm_mod <- linear_reg() %>%
  set_engine("lm")

lm_wflow <- workflow() %>%
  add_model(lm_mod)

pre_encoded <- lm_wflow %>%
  add_formula(body_mass_g ~ species + island + bill_depth_mm) %>%
  fit(data = penguins)

pre_encoded_parsnip_fit <- pre_encoded %>%
  extract_fit_parsnip()

pre_encoded_fit <- pre_encoded_parsnip_fit$fit

# The `lm()` formula is *not* the same as the `add_formula()` formula:
pre_encoded_fit
```

```
##
## Call:
## stats::lm(formula = ..y ~ ., data = data)
##
## Coefficients:
##      (Intercept)  speciesChinstrap     speciesGentoo
##        -1009.943             1.328          2236.865
##      islandDream   islandTorgersen     bill_depth_mm
##            9.221           -18.433           256.913
```

This can affect how the results are analyzed. For example, to get sequential hypothesis tests, each individual term is tested:

```
anova(pre_encoded_fit)
```

```
## Analysis of Variance Table
##
## Response: ..y
##                   Df    Sum Sq   Mean Sq  F value Pr(>F)
## speciesChinstrap   1  18642821  18642821 141.1482 <2e-16 ***
## speciesGentoo      1 128221393 128221393 970.7875 <2e-16 ***
## islandDream        1     13399     13399   0.1014 0.7503
## islandTorgersen    1       255       255   0.0019 0.9650
## bill_depth_mm      1  28051023  28051023 212.3794 <2e-16 ***
## Residuals        336  44378805    132080
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Overriding the default encodings:

Users can override the model-specific encodings by using a hardhat blueprint. The blueprint can specify how factors are encoded and whether intercepts are included. As an example, if you use a formula and would like the data to be passed to a model untouched:

```
minimal <- default_formula_blueprint(indicators = "none", intercept = FALSE)

un_encoded <- lm_wflow %>%
  add_formula(
    body_mass_g ~ species + island + bill_depth_mm,
    blueprint = minimal
  ) %>%
  fit(data = penguins)

un_encoded_parsnip_fit <- un_encoded %>%
  extract_fit_parsnip()

un_encoded_fit <- un_encoded_parsnip_fit$fit

un_encoded_fit
```

```
##
## Call:
## stats::lm(formula = ..y ~ ., data = data)
##
## Coefficients:
##      (Intercept)     bill_depth_mm  speciesChinstrap
##        -1009.943           256.913             1.328
##    speciesGentoo       islandDream   islandTorgersen
##         2236.865             9.221           -18.433
```

While this looks the same, the raw columns were given to `lm()` and that function created the dummy variables. Because of this, the sequential ANOVA tests groups of parameters to get column-level p-values:

```
anova(un_encoded_fit)
```

```
## Analysis of Variance Table
##
## Response: ..y
##                 Df    Sum Sq  Mean Sq F value Pr(>F)
## bill_depth_mm   1  48840779 48840779 369.782 <2e-16 ***
## species         2 126067249 63033624 477.239 <2e-16 ***
## island          2     20864    10432   0.079 0.9241
## Residuals     336  44378805   132080
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Overriding the default model formula:

Additionally, the formula passed to the underlying model can also be customized. In this case, the `formula` argument of `add_model()` can be used. To demonstrate, a spline function will be used for the bill depth:

```
library(splines)

custom_formula <- workflow() %>%
  add_model(
    lm_mod,
    formula = body_mass_g ~ species + island + ns(bill_depth_mm, 3)
  ) %>%
  add_formula(
    body_mass_g ~ species + island + bill_depth_mm,
    blueprint = minimal
  ) %>%
  fit(data = penguins)

custom_parsnip_fit <- custom_formula %>%
  extract_fit_parsnip()

custom_fit <- custom_parsnip_fit$fit

custom_fit
```

```
##
## Call:
## stats::lm(formula = body_mass_g ~ species + island + ns(bill_depth_mm,
##     3), data = data)
##
## Coefficients:
##           (Intercept)      speciesChinstrap         speciesGentoo
##              1959.090                 8.534              2352.137
##          islandDream       islandTorgersen  ns(bill_depth_mm, 3)1
##                2.425               -12.002               1476.386
## ns(bill_depth_mm, 3)2  ns(bill_depth_mm, 3)3
##             3187.839              1686.996
```

Altering the formula:

Finally, when a formula is updated or removed from a fitted workflow, the corresponding model fit is removed.

```
custom_formula_no_fit <- update_formula(custom_formula, body_mass_g ~ species)

try(extract_fit_parsnip(custom_formula_no_fit))

## Error in extract_fit_parsnip(custom_formula_no_fit) :
##   Can't extract a model fit from an untrained workflow.
## i Do you need to call `fit()`?
```

Examples

```
workflow <- workflow()
workflow <- add_formula(workflow, mpg ~ cyl)
workflow

remove_formula(workflow)

update_formula(workflow, mpg ~ disp)
```

---

add_model *Add a model to a workflow*

Description

- add_model() adds a parsnip model to the workflow.
- remove_model() removes the model specification as well as any fitted model object. Any extra formulas are also removed.
- update_model() first removes the model then adds the new specification to the workflow.

Usage

```
add_model(x, spec, ..., formula = NULL)

remove_model(x)

update_model(x, spec, ..., formula = NULL)
```

Arguments

x A workflow.

spec A parsnip model specification.

... These dots are for future extensions and must be empty.

formula An optional formula override to specify the terms of the model. Typically, the terms are extracted from the formula or recipe preprocessing methods. However, some models (like survival and bayesian models) use the formula not to preprocess, but to specify the structure of the model. In those cases, a formula specifying the model structure must be passed unchanged into the model call itself. This argument is used for those purposes.

Details

add_model() is a required step to construct a minimal workflow.

Value

x, updated with either a new or removed model.

Indicator Variable Details

Some modeling functions in R create indicator/dummy variables from categorical data when you use a model formula, and some do not. When you specify and fit a model with a workflow(), parsnip and workflows match and reproduce the underlying behavior of the user-specified model's computational engine.

Formula Preprocessor:

In the modeldata::Sacramento data set of real estate prices, the type variable has three levels: "Residential", "Condo", and "Multi-Family". This base workflow() contains a formula added via add_formula() to predict property price from property type, square footage, number of beds, and number of baths:

```
set.seed(123)

library(parsnip)
library(recipes)
library(workflows)
library(modeldata)

data("Sacramento")

base_wf <- workflow() %>%
  add_formula(price ~ type + sqft + beds + baths)
```

This first model does create dummy/indicator variables:

```
lm_spec <- linear_reg() %>%
  set_engine("lm")
```

```
base_wf %>%
  add_model(lm_spec) %>%
  fit(Sacramento)
```

```
## == Workflow [trained] ================================================
## Preprocessor: Formula
## Model: linear_reg()
##
## -- Preprocessor ------------------------------------------------------
## price ~ type + sqft + beds + baths
##
## -- Model -------------------------------------------------------------
##
## Call:
## stats::lm(formula = ..y ~ ., data = data)
##
## Coefficients:
##      (Intercept)  typeMulti_Family   typeResidential
##          32919.4          -21995.8           33688.6
##             sqft              beds             baths
##            156.2          -29788.0            8730.0
```

There are **five** independent variables in the fitted model for this OLS linear regression. With this model type and engine, the factor predictor `type` of the real estate properties was converted to two binary predictors, `typeMulti_Family` and `typeResidential`. (The third type, for condos, does not need its own column because it is the baseline level).

This second model does not create dummy/indicator variables:

```
rf_spec <- rand_forest() %>%
  set_mode("regression") %>%
  set_engine("ranger")

base_wf %>%
  add_model(rf_spec) %>%
  fit(Sacramento)
```

```
## == Workflow [trained] ================================================
## Preprocessor: Formula
## Model: rand_forest()
##
## -- Preprocessor ------------------------------------------------------
## price ~ type + sqft + beds + baths
##
## -- Model -------------------------------------------------------------
## Ranger result
##
## Call:
##  ranger::ranger(x = maybe_data_frame(x), y = y, num.threads = 1,     verbose = FALSE, seed = sample.i
##
## Type:                             Regression
## Number of trees:                  500
## Sample size:                      932
## Number of independent variables:  4
## Mtry:                             2
## Target node size:                 5
## Variable importance mode:         none
## Splitrule:                        variance
## OOB prediction error (MSE):       7058847504
## R squared (OOB):                  0.5894647
```

Note that there are **four** independent variables in the fitted model for this ranger random forest. With this model type and engine, indicator variables were not created for the `type` of real estate property being sold. Tree-based models such as random forest models can handle factor predictors directly, and don't need any conversion to numeric binary variables.

Recipe Preprocessor:

When you specify a model with a `workflow()` and a recipe preprocessor via `add_recipe()`, the *recipe* controls whether dummy variables are created or not; the recipe overrides any underlying behavior from the model's computational engine.

Examples

```
library(parsnip)

lm_model <- linear_reg()
lm_model <- set_engine(lm_model, "lm")

regularized_model <- set_engine(lm_model, "glmnet")

workflow <- workflow()
workflow <- add_model(workflow, lm_model)
workflow

workflow <- add_formula(workflow, mpg ~ .)
workflow

remove_model(workflow)

fitted <- fit(workflow, data = mtcars)
fitted

remove_model(fitted)

remove_model(workflow)

update_model(workflow, regularized_model)
update_model(fitted, regularized_model)
```

add_recipe *Add a recipe to a workflow*

Description

- add_recipe() specifies the terms of the model and any preprocessing that is required through the usage of a recipe.
- remove_recipe() removes the recipe as well as any downstream objects that might get created after the recipe is used for preprocessing, such as the prepped recipe. Additionally, if the model has already been fit, then the fit is removed.
- update_recipe() first removes the recipe, then replaces the previous recipe with the new one. Any model that has already been fit based on this recipe will need to be refit.

Usage

```
add_recipe(x, recipe, ..., blueprint = NULL)

remove_recipe(x)

update_recipe(x, recipe, ..., blueprint = NULL)
```

Arguments

x A workflow

recipe A recipe created using recipes::recipe(). The recipe should not have been trained already with recipes::prep(); workflows will handle training internally.

... Not used.

blueprint A hardhat blueprint used for fine tuning the preprocessing.

If NULL, hardhat::default_recipe_blueprint() is used.

Note that preprocessing done here is separate from preprocessing that might be done automatically by the underlying model.

Details

To fit a workflow, exactly one of add_formula(), add_recipe(), or add_variables() *must* be specified.

Value

x, updated with either a new or removed recipe preprocessor.

Examples

```
library(recipes)
library(magrittr)

recipe <- recipe(mpg ~ cyl, mtcars) %>%
  step_log(cyl)

workflow <- workflow() %>%
  add_recipe(recipe)

workflow

remove_recipe(workflow)

update_recipe(workflow, recipe(mpg ~ cyl, mtcars))
```

---

add_tailor *Add a tailor to a workflow*

Description

- `add_tailor()` specifies post-processing steps to apply through the usage of a tailor.
- `remove_tailor()` removes the tailor as well as any downstream objects that might get created after the tailor is used for post-processing, such as the fitted tailor.
- `update_tailor()` first removes the tailor, then replaces the previous tailor with the new one.

Usage

```
add_tailor(x, tailor, ...)

remove_tailor(x)

update_tailor(x, tailor, ...)
```

Arguments

`x`	A workflow
`tailor`	A tailor created using `tailor::tailor()`. The tailor should not have been trained already with `tailor::fit()`; workflows will handle training internally.
`...`	Not used.

Value

`x`, updated with either a new or removed tailor postprocessor.

Data Usage

While preprocessors and models are trained on data in the usual sense, postprocessors are training on *predictions* on data. When a workflow is fitted, the user typically supplies training data with the `data` argument. When workflows don't contain a postprocessor that requires training, users can pass all of the available data to the `data` argument to train the preprocessor and model. However, in the case where a postprocessor must be trained as well, allotting all of the available data to the `data` argument to train the preprocessor and model would leave no data to train the postprocessor with—if that were the case, workflows would need to `predict()` from the preprocessor and model on the same `data` that they were trained on, with the postprocessor then training on those predictions. Predictions on data that a model was trained on likely follow different distributions than predictions on unseen data; thus, workflows must split up the supplied `data` into two training sets, where the first is used to train the preprocessor and model and the second, called the "calibration set," is passed to that trained postprocessor and model to generate predictions, which then form the training data for the postprocessor.

When fitting a workflow with a postprocessor that requires training (i.e. one that returns `TRUE` in `.workflow_includes_calibration(workflow)`), users must pass two data arguments–the usual `fit.workflow(data)` will be used to train the preprocessor and model while `fit.workflow(calibration)` will be used to train the postprocessor.

In some situations, randomly splitting `fit.workflow(data)` (with `rsample::initial_split()`, for example) is sufficient to prevent data leakage. However, `fit.workflow(data)` could also have arisen as:

```
boots <- rsample::bootstraps(some_other_data)
split <- rsample::get_rsplit(boots, 1)
data <- rsample::assessment(split)
```

In this case, some of the rows in `data` will be duplicated. Thus, randomly allotting some of them to train the preprocessor and model and others to train the preprocessor would likely result in the same rows appearing in both datasets, resulting in the preprocessor and model generating predictions on rows they've seen before. Similarly problematic situations could arise in the context of other resampling situations, like time-based splits. In general, use the `rsample::inner_split()` function to prevent data leakage when resampling; when workflows with postprocessors that require training are passed to the tune package, this is handled internally.

Examples

```
library(tailor)
library(magrittr)

tailor <- tailor()
tailor_1 <- adjust_probability_threshold(tailor, .1)

workflow <- workflow() %>%
  add_tailor(tailor_1)

workflow

remove_tailor(workflow)
```

```
update_tailor(workflow, adjust_probability_threshold(tailor, .2))
```

add_variables *Add variables to a workflow*

Description

- `add_variables()` specifies the terms of the model through the usage of tidyselect::select_helpers for the `outcomes` and `predictors`.
- `remove_variables()` removes the variables. Additionally, if the model has already been fit, then the fit is removed.
- `update_variables()` first removes the variables, then replaces the previous variables with the new ones. Any model that has already been fit based on the original variables will need to be refit.
- `workflow_variables()` bundles `outcomes` and `predictors` into a single variables object, which can be supplied to `add_variables()`.

Usage

```
add_variables(x, outcomes, predictors, ..., blueprint = NULL, variables = NULL)

remove_variables(x)

update_variables(
  x,
  outcomes,
  predictors,
  ...,
  blueprint = NULL,
  variables = NULL
)

workflow_variables(outcomes, predictors)
```

Arguments

`x` A workflow

`outcomes, predictors`
Tidyselect expressions specifying the terms of the model. `outcomes` is evaluated first, and then all outcome columns are removed from the data before `predictors` is evaluated. See tidyselect::select_helpers for the full range of possible ways to specify terms.

`...` Not used.

blueprint A hardhat blueprint used for fine tuning the preprocessing.

If NULL, hardhat::default_xy_blueprint() is used.

Note that preprocessing done here is separate from preprocessing that might be done by the underlying model.

variables An alternative specification of outcomes and predictors, useful for supplying variables programmatically.

- If NULL, this argument is unused, and outcomes and predictors are used to specify the variables.
- Otherwise, this must be the result of calling workflow_variables() to create a standalone variables object. In this case, outcomes and predictors are completely ignored.

Details

To fit a workflow, exactly one of add_formula(), add_recipe(), or add_variables() *must* be specified.

Value

- add_variables() returns x with a new variables preprocessor.
- remove_variables() returns x after resetting any model fit and removing the variables preprocessor.
- update_variables() returns x after removing the variables preprocessor, and then re-specifying it with new variables.
- workflow_variables() returns a 'workflow_variables' object containing both the outcomes and predictors.

Examples

```
library(parsnip)

spec_lm <- linear_reg()
spec_lm <- set_engine(spec_lm, "lm")

workflow <- workflow()
workflow <- add_model(workflow, spec_lm)

# Add terms with tidyselect expressions.
# Outcomes are specified before predictors.
workflow1 <- add_variables(
  workflow,
  outcomes = mpg,
  predictors = c(cyl, disp)
)

workflow1 <- fit(workflow1, mtcars)
workflow1

# Removing the variables of a fit workflow will also remove the model
```

```
remove_variables(workflow1)

# Variables can also be updated
update_variables(workflow1, mpg, starts_with("d"))

# The `outcomes` are removed before the `predictors` expression
# is evaluated. This allows you to easily specify the predictors
# as "everything except the outcomes".
workflow2 <- add_variables(workflow, mpg, everything())
workflow2 <- fit(workflow2, mtcars)
extract_mold(workflow2)$predictors

# Variables can also be added from the result of a call to
# `workflow_variables()`, which creates a standalone variables object
variables <- workflow_variables(mpg, c(cyl, disp))
workflow3 <- add_variables(workflow, variables = variables)
fit(workflow3, mtcars)
```

---

augment.workflow *Augment data with predictions*

Description

This is a generics::augment() method for a workflow that calls augment() on the underlying parsnip model with new_data.

x must be a trained workflow, resulting in fitted parsnip model to augment() with.

new_data will be preprocessed using the preprocessor in the workflow, and that preprocessed data will be used to generate predictions. The final result will contain the original new_data with new columns containing the prediction information.

Usage

```
## S3 method for class 'workflow'
augment(x, new_data, eval_time = NULL, ...)
```

Arguments

x	A workflow
new_data	A data frame of predictors
eval_time	For censored regression models, a vector of time points at which the survival probability is estimated. See parsnip::augment.model_fit() for more details.
...	Arguments passed on to methods

Value

new_data with new prediction specific columns.

Examples

```
if (rlang::is_installed("broom")) {

library(parsnip)
library(magrittr)
library(modeldata)

data("attrition")

model <- logistic_reg() %>%
  set_engine("glm")

wf <- workflow() %>%
  add_model(model) %>%
  add_formula(
    Attrition ~ BusinessTravel + YearsSinceLastPromotion + OverTime
  )

wf_fit <- fit(wf, attrition)

augment(wf_fit, attrition)

}
```

---

control_workflow *Control object for a workflow*

Description

control_workflow() holds the control parameters for a workflow.

Usage

```
control_workflow(control_parsnip = NULL)
```

Arguments

control_parsnip
: A parsnip control object. If NULL, a default control argument is constructed from parsnip::control_parsnip().

Value

A control_workflow object for tweaking the workflow fitting process.

Examples

```
control_workflow()
```

extract-workflow *Extract elements of a workflow*

Description

These functions extract various elements from a workflow object. If they do not exist yet, an error is thrown.

- `extract_preprocessor()` returns the formula, recipe, or variable expressions used for preprocessing.
- `extract_spec_parsnip()` returns the parsnip model specification.
- `extract_fit_parsnip()` returns the parsnip model fit object.
- `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_mold()` returns the preprocessed "mold" object returned from `hardhat::mold()`. It contains information about the preprocessing, including either the prepped recipe, the formula terms object, or variable selectors.
- `extract_recipe()` returns the recipe. The `estimated` argument specifies whether the fitted or original recipe is returned.
- `extract_parameter_dials()` returns a single dials parameter object.
- `extract_parameter_set_dials()` returns a set of dials parameter objects.
- `extract_fit_time()` returns a tibble with elapsed fit times. The fit times correspond to the time for the parsnip engine or recipe steps to fit (or their sum if `summarize = TRUE`) and do not include other portions of the elapsed time in `fit.workflow()`.

Usage

```
## S3 method for class 'workflow'
extract_spec_parsnip(x, ...)

## S3 method for class 'workflow'
extract_recipe(x, ..., estimated = TRUE)

## S3 method for class 'workflow'
extract_fit_parsnip(x, ...)

## S3 method for class 'workflow'
extract_fit_engine(x, ...)

## S3 method for class 'workflow'
extract_mold(x, ...)

## S3 method for class 'workflow'
```

```
extract_preprocessor(x, ...)

## S3 method for class 'workflow'
extract_postprocessor(x, estimated = TRUE, ...)

## S3 method for class 'workflow'
extract_parameter_set_dials(x, ...)

## S3 method for class 'workflow'
extract_parameter_dials(x, parameter, ...)

## S3 method for class 'workflow'
extract_fit_time(x, summarize = TRUE, ...)
```

Arguments

`x` A workflow

`...` Not currently used.

`estimated` A logical for whether the original (unfit) recipe or the fitted recipe should be returned. This argument should be named.

`parameter` A single string for the parameter ID.

`summarize` A logical for whether the elapsed fit time should be returned as a single row or multiple rows.

Details

Extracting the underlying engine fit can be helpful for describing the model (via `print()`, `summary()`, `plot()`, etc.) or for variable importance/explainers.

However, users should not invoke the `predict()` method on an extracted model. There may be preprocessing operations that `workflows` has executed on the data prior to giving it to the model. Bypassing these can lead to errors or silently generating incorrect predictions.

Good:

```
workflow_fit %>% predict(new_data)
```

Bad:

```
workflow_fit %>% extract_fit_engine()  %>% predict(new_data)
# or
workflow_fit %>% extract_fit_parsnip() %>% predict(new_data)
```

Value

The extracted value from the object, `x`, as described in the description section.

Examples

```
library(parsnip)
library(recipes)
library(magrittr)

model <- linear_reg() %>%
  set_engine("lm")

recipe <- recipe(mpg ~ cyl + disp, mtcars) %>%
  step_log(disp)

base_wf <- workflow() %>%
  add_model(model)

recipe_wf <- add_recipe(base_wf, recipe)
formula_wf <- add_formula(base_wf, mpg ~ cyl + log(disp))
variable_wf <- add_variables(base_wf, mpg, c(cyl, disp))

fit_recipe_wf <- fit(recipe_wf, mtcars)
fit_formula_wf <- fit(formula_wf, mtcars)

# The preprocessor is a recipe, formula, or a list holding the
# tidyselect expressions identifying the outcomes/predictors
extract_preprocessor(recipe_wf)
extract_preprocessor(formula_wf)
extract_preprocessor(variable_wf)

# The `spec` is the parsnip spec before it has been fit.
# The `fit` is the fitted parsnip model.
extract_spec_parsnip(fit_formula_wf)
extract_fit_parsnip(fit_formula_wf)
extract_fit_engine(fit_formula_wf)

# The mold is returned from `hardhat::mold()`, and contains the
# predictors, outcomes, and information about the preprocessing
# for use on new data at `predict()` time.
extract_mold(fit_recipe_wf)

# A useful shortcut is to extract the fitted recipe from the workflow
extract_recipe(fit_recipe_wf)

# That is identical to
identical(
  extract_mold(fit_recipe_wf)$blueprint$recipe,
  extract_recipe(fit_recipe_wf)
)
```

fit-workflow *Fit a workflow object*

Description

Fitting a workflow currently involves three main steps:

- Preprocessing the data using a formula preprocessor, or by calling `recipes::prep()` on a recipe.
- Fitting the underlying parsnip model using `parsnip::fit.model_spec()`.
- Postprocessing predictions from the model using `tailor::tailor()`.

Usage

```
## S3 method for class 'workflow'
fit(object, data, ..., calibration = NULL, control = control_workflow())
```

Arguments

`object` A workflow

`data` A data frame of predictors and outcomes to use when fitting the preprocessor and model.

`...` Not used

`calibration` A data frame of predictors and outcomes to use when fitting the postprocessor. See the "Data Usage" section of `add_tailor()` for more information.

`control` A `control_workflow()` object

Value

The workflow `object`, updated with a fit parsnip model in the `object$fit$fit` slot.

Indicator Variable Details

Some modeling functions in R create indicator/dummy variables from categorical data when you use a model formula, and some do not. When you specify and fit a model with a `workflow()`, parsnip and workflows match and reproduce the underlying behavior of the user-specified model's computational engine.

Formula Preprocessor:

In the modeldata::Sacramento data set of real estate prices, the `type` variable has three levels: `"Residential"`, `"Condo"`, and `"Multi-Family"`. This base `workflow()` contains a formula added via `add_formula()` to predict property price from property type, square footage, number of beds, and number of baths:

```
set.seed(123)

library(parsnip)
library(recipes)
library(workflows)
library(modeldata)

data("Sacramento")
```

```
base_wf <- workflow() %>%
  add_formula(price ~ type + sqft + beds + baths)
```

This first model does create dummy/indicator variables:

```
lm_spec <- linear_reg() %>%
  set_engine("lm")

base_wf %>%
  add_model(lm_spec) %>%
  fit(Sacramento)
```

```
## == Workflow [trained] ================================================
## Preprocessor: Formula
## Model: linear_reg()
##
## -- Preprocessor ------------------------------------------------------
## price ~ type + sqft + beds + baths
##
## -- Model -------------------------------------------------------------
##
## Call:
## stats::lm(formula = ..y ~ ., data = data)
##
## Coefficients:
##      (Intercept)  typeMulti_Family   typeResidential
##          32919.4          -21995.8           33688.6
##             sqft              beds             baths
##            156.2          -29788.0            8730.0
```

There are **five** independent variables in the fitted model for this OLS linear regression. With this model type and engine, the factor predictor `type` of the real estate properties was converted to two binary predictors, `typeMulti_Family` and `typeResidential`. (The third type, for condos, does not need its own column because it is the baseline level).

This second model does not create dummy/indicator variables:

```
rf_spec <- rand_forest() %>%
  set_mode("regression") %>%
  set_engine("ranger")

base_wf %>%
  add_model(rf_spec) %>%
  fit(Sacramento)
```

```
## == Workflow [trained] ================================================
## Preprocessor: Formula
## Model: rand_forest()
##
## -- Preprocessor ------------------------------------------------------
## price ~ type + sqft + beds + baths
```

```
##
## -- Model -------------------------------------------------------------
## Ranger result
##
## Call:
##  ranger::ranger(x = maybe_data_frame(x), y = y, num.threads = 1,     verbose = FALSE, seed = sample.i
##
## Type:                             Regression
## Number of trees:                  500
## Sample size:                      932
## Number of independent variables:  4
## Mtry:                             2
## Target node size:                 5
## Variable importance mode:         none
## Splitrule:                        variance
## OOB prediction error (MSE):       7058847504
## R squared (OOB):                  0.5894647
```

Note that there are **four** independent variables in the fitted model for this ranger random forest. With this model type and engine, indicator variables were not created for the `type` of real estate property being sold. Tree-based models such as random forest models can handle factor predictors directly, and don't need any conversion to numeric binary variables.

Recipe Preprocessor:

When you specify a model with a `workflow()` and a recipe preprocessor via `add_recipe()`, the *recipe* controls whether dummy variables are created or not; the recipe overrides any underlying behavior from the model's computational engine.

Examples

```
library(parsnip)
library(recipes)
library(magrittr)

model <- linear_reg() %>%
  set_engine(”lm”)

base_wf <- workflow() %>%
  add_model(model)

formula_wf <- base_wf %>%
  add_formula(mpg ~ cyl + log(disp))

fit(formula_wf, mtcars)

recipe <- recipe(mpg ~ cyl + disp, mtcars) %>%
  step_log(disp)

recipe_wf <- base_wf %>%
  add_recipe(recipe)
```

```
fit(recipe_wf, mtcars)
```

glance.workflow *Glance at a workflow model*

Description

This is a generics::glance() method for a workflow that calls glance() on the underlying parsnip model.

x must be a trained workflow, resulting in fitted parsnip model to glance() at.

Usage

```
## S3 method for class 'workflow'
glance(x, ...)
```

Arguments

x	A workflow
...	Arguments passed on to methods

Examples

```
if (rlang::is_installed("broom")) {

library(parsnip)
library(magrittr)
library(modeldata)

data("attrition")

model <- logistic_reg() %>%
  set_engine("glm")

wf <- workflow() %>%
  add_model(model) %>%
  add_formula(
    Attrition ~ BusinessTravel + YearsSinceLastPromotion + OverTime
  )

# Workflow must be trained to call `glance()`
try(glance(wf))

wf_fit <- fit(wf, attrition)

glance(wf_fit)

}
```

is_trained_workflow *Determine if a workflow has been trained*

Description

A trained workflow is one that has gone through fit(), which preprocesses the underlying data, and fits the parsnip model.

Usage

```
is_trained_workflow(x)
```

Arguments

x	A workflow.

Value

A single logical indicating if the workflow has been trained or not.

Examples

```
library(parsnip)
library(recipes)
library(magrittr)

rec <- recipe(mpg ~ cyl, mtcars)

mod <- linear_reg()
mod <- set_engine(mod, "lm")

wf <- workflow() %>%
  add_recipe(rec) %>%
  add_model(mod)

# Before any preprocessing or model fitting has been done
is_trained_workflow(wf)

wf <- fit(wf, mtcars)

# After all preprocessing and model fitting
is_trained_workflow(wf)
```

predict-workflow *Predict from a workflow*

Description

This is the predict() method for a fit workflow object. The nice thing about predicting from a workflow is that it will:

- Preprocess new_data using the preprocessing method specified when the workflow was created and fit. This is accomplished using hardhat::forge(), which will apply any formula preprocessing or call recipes::bake() if a recipe was supplied.
- Call parsnip::predict.model_fit() for you using the underlying fit parsnip model.

Usage

```
## S3 method for class 'workflow'
predict(object, new_data, type = NULL, opts = list(), ...)
```

Arguments

object A workflow that has been fit by fit.workflow()

new_data A data frame containing the new predictors to preprocess and predict on. If using a recipe preprocessor, you should not call recipes::bake() on new_data before passing to this function.

type A single character value or NULL. Possible values are "numeric", "class", "prob", "conf_int", "pred_int", "quantile", "time", "hazard", "survival", or "raw". When NULL, predict() will choose an appropriate value based on the model's mode.

opts A list of optional arguments to the underlying predict function that will be used when type = "raw". The list should not include options for the model object or the new data being predicted.

... Additional parsnip-related options, depending on the value of type. Arguments to the underlying model's prediction function cannot be passed here (use the opts argument instead). Possible arguments are:

- interval: for type equal to "survival" or "quantile", should interval estimates be added, if available? Options are "none" and "confidence".
- level: for type equal to "conf_int", "pred_int", or "survival", this is the parameter for the tail area of the intervals (e.g. confidence level for confidence intervals). Default value is 0.95.
- std_error: for type equal to "conf_int" or "pred_int", add the standard error of fit or prediction (on the scale of the linear predictors). Default value is FALSE.
- quantile: for type equal to quantile, the quantiles of the distribution. Default is (1:9)/10.
- eval_time: for type equal to "survival" or "hazard", the time points at which the survival probability or hazard is estimated.

Value

A data frame of model predictions, with as many rows as `new_data` has.

Examples

```
library(parsnip)
library(recipes)
library(magrittr)

training <- mtcars[1:20, ]
testing <- mtcars[21:32, ]

model <- linear_reg() %>%
  set_engine("lm")

workflow <- workflow() %>%
  add_model(model)

recipe <- recipe(mpg ~ cyl + disp, training) %>%
  step_log(disp)

workflow <- add_recipe(workflow, recipe)

fit_workflow <- fit(workflow, training)

# This will automatically `bake()` the recipe on `testing`,
# applying the log step to `disp`, and then fit the regression.
predict(fit_workflow, testing)
```

`tidy.workflow` *Tidy a workflow*

Description

This is a `generics::tidy()` method for a workflow that calls `tidy()` on either the underlying parsnip model or the recipe, depending on the value of `what`.

`x` must be a fitted workflow, resulting in fitted parsnip model or prepped recipe that you want to tidy.

Usage

```
## S3 method for class 'workflow'
tidy(x, what = "model", ...)
```

Arguments

`x`	A workflow
`what`	A single string. Either `"model"` or `"recipe"` to select which part of the workflow to tidy. Defaults to tidying the model.
`...`	Arguments passed on to methods

Details

To tidy the unprepped recipe, use `extract_preprocessor()` and `tidy()` that directly.

`workflow` *Create a workflow*

Description

A `workflow` is a container object that aggregates information required to fit and predict from a model. This information might be a recipe used in preprocessing, specified through `add_recipe()`, or the model specification to fit, specified through `add_model()`, or a tailor used in postprocessing, specified through `add_tailor()`.

The `preprocessor` and `spec` arguments allow you to add components to a workflow quickly, without having to go through the `add_*()` functions, such as `add_recipe()` or `add_model()`. However, if you need to control any of the optional arguments to those functions, such as the `blueprint` or the model `formula`, then you should use the `add_*()` functions directly instead.

Usage

```
workflow(preprocessor = NULL, spec = NULL, postprocessor = NULL)
```

Arguments

`preprocessor` An optional preprocessor to add to the workflow. One of:

- A formula, passed on to `add_formula()`.
- A recipe, passed on to `add_recipe()`.
- A `workflow_variables()` object, passed on to `add_variables()`.

`spec` An optional parsnip model specification to add to the workflow. Passed on to `add_model()`.

`postprocessor` An optional `tailor::tailor()` defining post-processing steps to add to the workflow. Passed on to `add_tailor()`.

Value

A new `workflow` object.

Indicator Variable Details

Some modeling functions in R create indicator/dummy variables from categorical data when you use a model formula, and some do not. When you specify and fit a model with a `workflow()`, parsnip and workflows match and reproduce the underlying behavior of the user-specified model's computational engine.

Formula Preprocessor:

In the modeldata::Sacramento data set of real estate prices, the type variable has three levels: "Residential", "Condo", and "Multi-Family". This base workflow() contains a formula added via add_formula() to predict property price from property type, square footage, number of beds, and number of baths:

```
set.seed(123)

library(parsnip)
library(recipes)
library(workflows)
library(modeldata)

data("Sacramento")

base_wf <- workflow() %>%
  add_formula(price ~ type + sqft + beds + baths)
```

This first model does create dummy/indicator variables:

```
lm_spec <- linear_reg() %>%
  set_engine("lm")

base_wf %>%
  add_model(lm_spec) %>%
  fit(Sacramento)
```

```
## == Workflow [trained] ================================================
## Preprocessor: Formula
## Model: linear_reg()
##
## -- Preprocessor ------------------------------------------------------
## price ~ type + sqft + beds + baths
##
## -- Model -------------------------------------------------------------
##
## Call:
## stats::lm(formula = ..y ~ ., data = data)
##
## Coefficients:
##      (Intercept)  typeMulti_Family   typeResidential
##          32919.4          -21995.8           33688.6
##             sqft              beds             baths
##            156.2          -29788.0            8730.0
```

There are **five** independent variables in the fitted model for this OLS linear regression. With this model type and engine, the factor predictor type of the real estate properties was converted to two binary predictors, typeMulti_Family and typeResidential. (The third type, for condos, does not need its own column because it is the baseline level).

This second model does not create dummy/indicator variables:

```
rf_spec <- rand_forest() %>%
  set_mode("regression") %>%
  set_engine("ranger")

base_wf %>%
  add_model(rf_spec) %>%
  fit(Sacramento)
```

```
## == Workflow [trained] ================================================
## Preprocessor: Formula
## Model: rand_forest()
##
## -- Preprocessor ------------------------------------------------------
## price ~ type + sqft + beds + baths
##
## -- Model -------------------------------------------------------------
## Ranger result
##
## Call:
##  ranger::ranger(x = maybe_data_frame(x), y = y, num.threads = 1,     verbose = FALSE, seed = sample.i
##
## Type:                             Regression
## Number of trees:                  500
## Sample size:                      932
## Number of independent variables:  4
## Mtry:                             2
## Target node size:                 5
## Variable importance mode:         none
## Splitrule:                        variance
## OOB prediction error (MSE):       7058847504
## R squared (OOB):                  0.5894647
```

Note that there are **four** independent variables in the fitted model for this ranger random forest. With this model type and engine, indicator variables were not created for the `type` of real estate property being sold. Tree-based models such as random forest models can handle factor predictors directly, and don't need any conversion to numeric binary variables.

Recipe Preprocessor:

When you specify a model with a `workflow()` and a recipe preprocessor via `add_recipe()`, the *recipe* controls whether dummy variables are created or not; the recipe overrides any underlying behavior from the model's computational engine.

Examples

```
library(parsnip)
library(recipes)
library(magrittr)
library(modeldata)

data("attrition")
```

```
model <- logistic_reg() %>%
  set_engine("glm")

formula <- Attrition ~ BusinessTravel + YearsSinceLastPromotion + OverTime

wf_formula <- workflow(formula, model)

fit(wf_formula, attrition)

recipe <- recipe(Attrition ~ ., attrition) %>%
  step_dummy(all_nominal(), -Attrition) %>%
  step_corr(all_predictors(), threshold = 0.8)

wf_recipe <- workflow(recipe, model)

fit(wf_recipe, attrition)

variables <- workflow_variables(
  Attrition,
  c(BusinessTravel, YearsSinceLastPromotion, OverTime)
)

wf_variables <- workflow(variables, model)

fit(wf_variables, attrition)
```

---

workflow-butcher *Butcher methods for a workflow*

Description

These methods allow you to use the butcher package to reduce the size of a workflow. After calling `butcher::butcher()` on a workflow, the only guarantee is that you will still be able to `predict()` from that workflow. Other functions may not work as expected.

Usage

```
axe_call.workflow(x, verbose = FALSE, ...)

axe_ctrl.workflow(x, verbose = FALSE, ...)

axe_data.workflow(x, verbose = FALSE, ...)

axe_env.workflow(x, verbose = FALSE, ...)

axe_fitted.workflow(x, verbose = FALSE, ...)
```

Arguments

`x`	A workflow.
`verbose`	Should information be printed about how much memory is freed from butchering?
`...`	Extra arguments possibly used by underlying methods.

Index