Package ‘modeltuning’

October 28, 2025

Title Model Selection and Tuning Utilities

Version 0.1.2

Description Provides a lightweight framework for model selection and hyperparameter tuning in R. The package offers intuitive tools for grid search, cross-validation, and combined grid search with cross-validation that work seamlessly with virtually any modeling package. Designed for flexibility and ease of use, it standardizes tuning workflows while remaining fully compatible with a wide range of model interfaces and estimation functions.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.3

Depends R (>= 4.1.0)

Imports future.apply, progressr, R6, rlang

Suggests e1071, future, parallelly, paws, rsample, rpart, yardstick, knitr, rmarkdown

URL https://www.dmolitor.com/modeltuning/

VignetteBuilder knitr

NeedsCompilation no

Author Daniel Molitor [aut, cre]

Maintainer Daniel Molitor <molitdj97@gmail.com>

Repository CRAN

Date/Publication 2025-10-28 12:50:02 UTC

Contents

`CV` *Predictive Models with Cross Validation*

Description

`CV` allows the user to specify a cross validation scheme with complete flexibility in the model, data splitting function, and performance metrics, among other essential parameters.

Public fields

`learner` Predictive modeling function.

`scorer` List of performance metric functions.

`splitter` Function that splits data into cross validation folds.

Methods

Public methods:

- `CV$fit()`
- `CV$new()`
- `CV$clone()`

Method `fit()`**:** `fit` performs cross validation with user-specified parameters.

Usage:

```
CV$fit(
  formula = NULL,
  data = NULL,
  x = NULL,
  y = NULL,
  response = NULL,
  convert_response = NULL,
  progress = FALSE
)
```

Arguments:

`formula` An object of class formula: a symbolic description of the model to be fitted.

`data` An optional data frame, or other object containing the variables in the model. If `data` is not provided, how `formula` is handled depends on `$learner`.

`x` Predictor data (independent variables), alternative interface to data with formula.

`y` Response vector (dependent variable), alternative interface to data with formula.

`response` String; In the absence of `formula` or `y`, this specifies which element of `learner_args` is the response vector.

`convert_response` Function; This should be a single function that transforms the response vector. E.g. a function converting a numeric binary variable to a factor variable.

`progress` Logical; indicating whether to print progress across cross validation folds.

Details: `fit` follows standard R modeling convention by surfacing a formula modeling interface as well as an alternate matrix option. The user should use whichever interface is supported by the specified `$learner` function.

Returns: An object of class FittedCV.

Examples:

```
if (require(e1071) && require(rpart) && require(yardstick)) {
  iris_new <- iris[sample(1:nrow(iris), nrow(iris)), ]
  iris_new$Species <- factor(iris_new$Species == "virginica")

  ### Decision Tree Example

  iris_cv <- CV$new(
    learner = rpart::rpart,
    learner_args = list(method = "class"),
    splitter = cv_split,
    scorer = list(accuracy = yardstick::accuracy_vec),
    prediction_args = list(accuracy = list(type = "class"))
  )
  iris_cv_fitted <- iris_cv$fit(formula = Species ~ ., data = iris_new)

  ### Example with multiple metric functions

  iris_cv <- CV$new(
    learner = rpart::rpart,
    learner_args = list(method = "class"),
    splitter = cv_split,
    splitter_args = list(v = 3),
    scorer = list(
      f_meas = yardstick::f_meas_vec,
      accuracy = yardstick::accuracy_vec,
      roc_auc = yardstick::roc_auc_vec,
      pr_auc = yardstick::pr_auc_vec
    ),
    prediction_args = list(
      f_meas = list(type = "class"),
      accuracy = list(type = "class"),
      roc_auc = list(type = "prob"),
      pr_auc = list(type = "prob")
    ),
    convert_predictions = list(
      f_meas = NULL,
      accuracy = NULL,
      roc_auc = function(i) i[, "FALSE"],
      pr_auc = function(i) i[, "FALSE"]
```

```
    )
  )
  iris_cv_fitted <- iris_cv$fit(formula = Species ~ ., data = iris_new)

  # Print the mean performance metrics across CV folds
  iris_cv_fitted$mean_metrics

  # Grab the final model fitted on the full dataset
  iris_cv_fitted$model

  ### OLS Example

  mtcars_cv <- CV$new(
    learner = lm,
    splitter = cv_split,
    splitter_args = list(v = 2),
    scorer = list("rmse" = yardstick::rmse_vec, "mae" = yardstick::mae_vec)
  )

  mtcars_cv_fitted <- mtcars_cv$fit(
    formula = mpg ~ .,
    data = mtcars
  )

  ### Matrix interface example - SVM

  mtcars_x <- model.matrix(mpg ~ . - 1, mtcars)
  mtcars_y <- mtcars$mpg

  mtcars_cv <- CV$new(
    learner = e1071::svm,
    learner_args = list(scale = TRUE, kernel = "polynomial", cross = 0),
    splitter = cv_split,
    splitter_args = list(v = 3),
    scorer = list(rmse = yardstick::rmse_vec, mae = yardstick::mae_vec)
  )
  mtcars_cv_fitted <- mtcars_cv$fit(
    x = mtcars_x,
    y = mtcars_y
  )
}
```

Method `new()`**:** Create a new CV object.

Usage:

```
CV$new(
  learner = NULL,
  splitter = NULL,
  scorer = NULL,
```

```
  learner_args = NULL,
  splitter_args = NULL,
  scorer_args = NULL,
  prediction_args = NULL,
  convert_predictions = NULL
)
```

Arguments:

`learner` Function that estimates a predictive model. It is essential that this function support either a formula interface with `formula` and `data` arguments, or an alternate matrix interface with `x` and `y` arguments.

`splitter` A function that computes cross validation folds from an input data set or a pre-computed list of cross validation fold indices. If `splitter` is a function, it must have a `data` argument for the input data, and it must return a list of cross validation fold indices. If `splitter` is a list of integers, the number of cross validation folds is `length(splitter)` and each element contains the indices of the data observations that are included in that fold.

`scorer` A named list of metric functions to evaluate model performance on each cross validation fold. Any provided metric function must have `truth` and `estimate` arguments for true outcome values and predicted outcome values respectively, and must return a single numeric metric value.

`learner_args` A named list of additional arguments to pass to `learner`.

`splitter_args` A named list of additional arguments to pass to `splitter`.

`scorer_args` A named list of additional arguments to pass to `scorer`. `scorer_args` must either be length 1 or `length(scorer)` in the case where different arguments are being passed to each scoring function.

`prediction_args` A named list of additional arguments to pass to `predict`. `prediction_args` must either be length 1 or `length(scorer)` in the case where different arguments are being passed to each scoring function.

`convert_predictions` A list of functions to convert predicted values prior to being evaluated by the metric functions supplied in `scorer`. This list should either be length 1, in which case the same function will be applied to all predicted values, or `length(scorer)` in which case each function in `convert_predictions` will correspond with each function in `scorer`.

Returns: An object of class CV.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## ------------------------------------------------
## Method `CV$fit`
## ------------------------------------------------

if (require(e1071) && require(rpart) && require(yardstick)) {
```

```
iris_new <- iris[sample(1:nrow(iris), nrow(iris)), ]
iris_new$Species <- factor(iris_new$Species == "virginica")

### Decision Tree Example

iris_cv <- CV$new(
  learner = rpart::rpart,
  learner_args = list(method = "class"),
  splitter = cv_split,
  scorer = list(accuracy = yardstick::accuracy_vec),
  prediction_args = list(accuracy = list(type = "class"))
)
iris_cv_fitted <- iris_cv$fit(formula = Species ~ ., data = iris_new)

### Example with multiple metric functions

iris_cv <- CV$new(
  learner = rpart::rpart,
  learner_args = list(method = "class"),
  splitter = cv_split,
  splitter_args = list(v = 3),
  scorer = list(
    f_meas = yardstick::f_meas_vec,
    accuracy = yardstick::accuracy_vec,
    roc_auc = yardstick::roc_auc_vec,
    pr_auc = yardstick::pr_auc_vec
  ),
  prediction_args = list(
    f_meas = list(type = "class"),
    accuracy = list(type = "class"),
    roc_auc = list(type = "prob"),
    pr_auc = list(type = "prob")
  ),
  convert_predictions = list(
    f_meas = NULL,
    accuracy = NULL,
    roc_auc = function(i) i[, "FALSE"],
    pr_auc = function(i) i[, "FALSE"]
  )
)
iris_cv_fitted <- iris_cv$fit(formula = Species ~ ., data = iris_new)

# Print the mean performance metrics across CV folds
iris_cv_fitted$mean_metrics

# Grab the final model fitted on the full dataset
iris_cv_fitted$model

### OLS Example

mtcars_cv <- CV$new(
  learner = lm,
  splitter = cv_split,
```

```
    splitter_args = list(v = 2),
    scorer = list("rmse" = yardstick::rmse_vec, "mae" = yardstick::mae_vec)
  )

  mtcars_cv_fitted <- mtcars_cv$fit(
    formula = mpg ~ .,
    data = mtcars
  )

  ### Matrix interface example - SVM

  mtcars_x <- model.matrix(mpg ~ . - 1, mtcars)
  mtcars_y <- mtcars$mpg

  mtcars_cv <- CV$new(
    learner = e1071::svm,
    learner_args = list(scale = TRUE, kernel = "polynomial", cross = 0),
    splitter = cv_split,
    splitter_args = list(v = 3),
    scorer = list(rmse = yardstick::rmse_vec, mae = yardstick::mae_vec)
  )
  mtcars_cv_fitted <- mtcars_cv$fit(
    x = mtcars_x,
    y = mtcars_y
  )
}
```

---

cv_split *Generate cross-validation fold indices*

Description

Splits row indices of a data frame or matrix into k folds for cross-validation.

Usage

```
cv_split(data, v = 5, seed = NULL)
```

Arguments

data	A data frame or matrix.
v	Integer. Number of folds. Defaults to 5.
seed	Optional integer. Random seed for reproducibility.

Value

A list of length v, where each element is a vector of row indices for that fold.

Examples

```
folds <- cv_split(mtcars, v = 5)
str(folds)
```

FittedCV *Fitted, Cross-Validated Predictive Models*

Description

`FittedCV` is a fitted, cross-validated predictive model object that is returned by `CV$fit()` and contains relevant model components, cross-validation metrics, validation set predicted values, etc.

Public fields

`folds` A list of length `$nfolds` where each element contains the indices of the observations contained in that fold.

`model` Predictive model fitted on the full data set.

`mean_metrics` Numeric list; Cross-validation performance metrics averaged across folds.

`metrics` Numeric list; Cross-validation performance metrics on each fold.

`nfolds` An integer specifying the number of cross-validation folds.

`predictions` A list containing the predicted hold-out values on every fold.

Methods

Public methods:

- `FittedCV$new()`
- `FittedCV$clone()`

Method `new()`**:** Create a new FittedCV object.

Usage:

```
FittedCV$new(folds, model, metrics, nfolds, predictions)
```

Arguments:

`folds` A list of length `$nfolds` where each element contains the indices of the observations contained in that fold.

`model` Predictive model fitted on the full data set.

`metrics` Numeric list; Cross-validation performance metrics on each fold.

`nfolds` An integer specifying the number of cross-validation folds.

`predictions` A list containing the predicted hold-out values on every fold.

Returns: An object of class FittedCV.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`FittedGridSearch` *Fitted Models across a Tuning Grid of Hyper-parameters*

Description

`FittedGridSearch` is an object containing fitted predictive models across a tuning grid of hyper-parameters returned by `GridSearch$fit()` as well as relevant model information such as the best performing model, best hyper-parameters, etc.

Public fields

`best_idx` An integer specifying the index of `$models` that contains the best-performing model.

`best_metric` The performance metric of the best model on the validation data.

`best_model` The best performing predictive model.

`best_params` A named list of the hyper-parameters that result in the optimal predictive model.

`tune_params` Data.frame of the full hyper-parameter grid.

`models` List of predictive models at every value of `$tune_params`.

`metrics` Numeric list; Cross-validation performance metrics on each fold.

`predictions` A list containing the predicted hold-out values on every fold.

Methods

Public methods:

- `FittedGridSearch$new()`
- `FittedGridSearch$clone()`

Method `new()`**:** Create a new FittedGridSearch object.

Usage:

```
FittedGridSearch$new(tune_params, models, metrics, predictions, optimize_score)
```

Arguments:

`tune_params` Data.frame of the full hyper-parameter grid.

`models` List of predictive models at every value of `$tune_params`.

`metrics` List of performance metrics on the validation data for every model in `$models`.

`predictions` A list containing the predicted values on the validation data for every model in `$models`.

`optimize_score` Either "max" or "min" indicating whether or not the specified performance metric was maximized or minimized to find the optimal predictive model.

Returns: An object of class FittedGridSearch.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

FittedGridSearchCV *Fitted Models with Cross Validation across a Tuning Grid of Hyper-parameters*

Description

FittedGridSearchCV is an object containing fitted predictive models across a tuning grid of hyper-parameters returned by GridSearchCV$fit() as well as relevant model information such as the best performing model, best hyper-parameters, etc.

Public fields

best_idx An integer specifying the index of $models that contains the best-performing model.

best_metric The average performance metric of the best model across cross-validation folds.

best_model The best performing predictive model.

best_params A named list of the hyper-parameters that result in the optimal predictive model.

folds A list of length $models where each element contains a list of the cross-validation indices for each fold.

tune_params A data.frame of the full hyper-parameter grid.

models List of predictive models at every value of $tune_params.

metrics Numeric list; Cross-validation performance metrics for every model in $models.

predictions A list containing the cross-validation fold predictions for each model in $models.

Methods

Public methods:

- FittedGridSearchCV$new()
- FittedGridSearchCV$clone()

Method new()**:** Create a new FittedGridSearchCV object.

Usage:

```
FittedGridSearchCV$new(
  tune_params,
  models,
  folds,
  metrics,
  predictions,
  optimize_score
)
```

Arguments:

tune_params Data.frame of the full hyper-parameter grid.

models List of predictive models at every value of $tune_params.

folds List of cross-validation indices at every value of $tune_params.

metrics List of cross-validation performance metrics for every model in $models.

predictions A list containing the predicted values on the cross-validation folds for every model in $models.

optimize_score Either "max" or "min" indicating whether or not the specified performance metric was maximized or minimized to find the optimal predictive model.

Returns: An object of class FittedGridSearchCV.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

GridSearch *Tune Predictive Model Hyper-parameters with Grid Search*

Description

GridSearch allows the user to specify a Grid Search schema for tuning predictive model hyper-parameters with complete flexibility in the predictive model and performance metrics.

Public fields

learner Predictive modeling function.

scorer List of performance metric functions.

tune_params Data.frame of full hyper-parameter grid created from $tune_params

Methods

Public methods:

- GridSearch$fit()
- GridSearch$new()
- GridSearch$clone()

Method fit()**:** fit tunes user-specified model hyper-parameters via Grid Search.

Usage:

```
GridSearch$fit(
  formula = NULL,
  data = NULL,
  x = NULL,
  y = NULL,
  progress = FALSE
)
```

Arguments:

`formula` An object of class formula: a symbolic description of the model to be fitted.

`data` An optional data frame, or other object containing the variables in the model. If `data` is not provided, how `formula` is handled depends on `$learner`.

`x` Predictor data (independent variables), alternative interface to data with formula.

`y` Response vector (dependent variable), alternative interface to data with formula.

`progress` Logical; indicating whether to print progress across cross validation folds.

Details: `fit` follows standard R modeling convention by surfacing a formula modeling interface as well as an alternate matrix option. The user should use whichever interface is supported by the specified `$learner` function.

Returns: An object of class FittedGridSearch.

Examples:

```
if (require(e1071) && require(rpart) && require(yardstick)) {
  iris_new <- iris[sample(1:nrow(iris), nrow(iris)), ]
  iris_new$Species <- factor(iris_new$Species == "virginica")
  iris_train <- iris_new[1:100, ]
  iris_validate <- iris_new[101:150, ]

  ### Decision Tree example

  iris_grid <- GridSearch$new(
    learner = rpart::rpart,
    learner_args = list(method = "class"),
    tune_params = list(
      minsplit = seq(10, 30, by = 5),
      maxdepth = seq(20, 30, by = 2)
    ),
   evaluation_data = list(x = iris_validate[, 1:4], y = iris_validate$Species),
    scorer = list(accuracy = yardstick::accuracy_vec),
    optimize_score = "max",
    prediction_args = list(accuracy = list(type = "class"))
  )
  iris_grid_fitted <- iris_grid$fit(
    formula = Species ~ .,
    data = iris_train
  )

  ### Example with multiple metric functions

  iris_grid <- GridSearch$new(
    learner = rpart::rpart,
    learner_args = list(method = "class"),
    tune_params = list(
      minsplit = seq(10, 30, by = 5),
      maxdepth = seq(20, 30, by = 2)
    ),
    evaluation_data = list(x = iris_validate, y = iris_validate$Species),
    scorer = list(
```

```
    accuracy = yardstick::accuracy_vec,
    auc = yardstick::roc_auc_vec
  ),
  optimize_score = "max",
  prediction_args = list(
    accuracy = list(type = "class"),
    auc = list(type = "prob")
  ),
  convert_predictions = list(
    accuracy = NULL,
    auc = function(i) i[, "FALSE"]
  )
)
iris_grid_fitted <- iris_grid$fit(
  formula = Species ~ .,
  data = iris_train,
)

# Grab the best model
iris_grid_fitted$best_model

# Grab the best hyper-parameters
iris_grid_fitted$best_params

# Grab the best model performance metrics
iris_grid_fitted$best_metric

### Matrix interface example - SVM

mtcars_train <- mtcars[1:25, ]
mtcars_eval <- mtcars[26:nrow(mtcars), ]

mtcars_grid <- GridSearch$new(
  learner = e1071::svm,
  tune_params = list(
    degree = 2:4,
    kernel = c("linear", "polynomial")
  ),
  evaluation_data = list(x = mtcars_eval[, -1], y = mtcars_eval$mpg),
  learner_args = list(scale = TRUE),
  scorer = list(
    rmse = yardstick::rmse_vec,
    mae = yardstick::mae_vec
  ),
  optimize_score = "min"
)
mtcars_grid_fitted <- mtcars_grid$fit(
  x = mtcars_train[, -1],
```

```
    y = mtcars_train$mpg
  )

}
```

Method `new()`: Create a new GridSearch object.

Usage:

```
GridSearch$new(
  learner = NULL,
  tune_params = NULL,
  evaluation_data = NULL,
  scorer = NULL,
  optimize_score = c("min", "max"),
  learner_args = NULL,
  scorer_args = NULL,
  prediction_args = NULL,
  convert_predictions = NULL
)
```

Arguments:

`learner` Function that estimates a predictive model. It is essential that this function support either a formula interface with `formula` and `data` arguments, or an alternate matrix interface with `x` and `y` arguments.

`tune_params` A named list specifying the arguments of `$learner` to tune.

`evaluation_data` A two-element list containing the following elements: `x`, the validation data to generate predicted values with; `y`, the validation response values to evaluate predictive performance.

`scorer` A named list of metric functions to evaluate model performance on `evaluation_data`. Any provided metric function must have `truth` and `estimate` arguments, for true outcome values and predicted outcome values respectively, and must return a single numeric metric value. The last metric function will be the one used to identify the optimal model from the Grid Search.

`optimize_score` One of "max" or "min"; Whether to maximize or minimize the metric defined in `scorer` to find the optimal Grid Search parameters.

`learner_args` A named list of additional arguments to pass to `learner`.

`scorer_args` A named list of additional arguments to pass to `scorer`. `scorer_args` must either be length 1 or `length(scorer)` in the case where different arguments are being passed to each scoring function.

`prediction_args` A named list of additional arguments to pass to `predict`. `prediction_args` must either be length 1 or `length(scorer)` in the case where different arguments are being passed to each scoring function.

`convert_predictions` A list of functions to convert predicted values prior to being evaluated by the metric functions supplied in `scorer`. This list should either be length 1, in which case the same function will be applied to all predicted values, or `length(scorer)` in which case each function in `convert_predictions` will correspond with each function in `scorer`.

Returns: An object of class GridSearch.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Examples

```
## ------------------------------------------------
## Method `GridSearch$fit`
## ------------------------------------------------

if (require(e1071) && require(rpart) && require(yardstick)) {
  iris_new <- iris[sample(1:nrow(iris), nrow(iris)), ]
  iris_new$Species <- factor(iris_new$Species == "virginica")
  iris_train <- iris_new[1:100, ]
  iris_validate <- iris_new[101:150, ]

  ### Decision Tree example

  iris_grid <- GridSearch$new(
    learner = rpart::rpart,
    learner_args = list(method = "class"),
    tune_params = list(
      minsplit = seq(10, 30, by = 5),
      maxdepth = seq(20, 30, by = 2)
    ),
    evaluation_data = list(x = iris_validate[, 1:4], y = iris_validate$Species),
    scorer = list(accuracy = yardstick::accuracy_vec),
    optimize_score = "max",
    prediction_args = list(accuracy = list(type = "class"))
  )
  iris_grid_fitted <- iris_grid$fit(
    formula = Species ~ .,
    data = iris_train
  )

  ### Example with multiple metric functions

  iris_grid <- GridSearch$new(
    learner = rpart::rpart,
    learner_args = list(method = "class"),
    tune_params = list(
      minsplit = seq(10, 30, by = 5),
      maxdepth = seq(20, 30, by = 2)
    ),
    evaluation_data = list(x = iris_validate, y = iris_validate$Species),
    scorer = list(
      accuracy = yardstick::accuracy_vec,
      auc = yardstick::roc_auc_vec
    ),
    optimize_score = "max",
    prediction_args = list(
```

```
    accuracy = list(type = "class"),
    auc = list(type = "prob")
  ),
  convert_predictions = list(
    accuracy = NULL,
    auc = function(i) i[, "FALSE"]
  )
)
iris_grid_fitted <- iris_grid$fit(
  formula = Species ~ .,
  data = iris_train,
)

# Grab the best model
iris_grid_fitted$best_model

# Grab the best hyper-parameters
iris_grid_fitted$best_params

# Grab the best model performance metrics
iris_grid_fitted$best_metric

### Matrix interface example - SVM

mtcars_train <- mtcars[1:25, ]
mtcars_eval <- mtcars[26:nrow(mtcars), ]

mtcars_grid <- GridSearch$new(
  learner = e1071::svm,
  tune_params = list(
    degree = 2:4,
    kernel = c("linear", "polynomial")
  ),
  evaluation_data = list(x = mtcars_eval[, -1], y = mtcars_eval$mpg),
  learner_args = list(scale = TRUE),
  scorer = list(
    rmse = yardstick::rmse_vec,
    mae = yardstick::mae_vec
  ),
  optimize_score = "min"
)
mtcars_grid_fitted <- mtcars_grid$fit(
  x = mtcars_train[, -1],
  y = mtcars_train$mpg
)

}
```

`GridSearchCV` *Tune Predictive Model Hyper-parameters with Grid Search and Cross-Validation*

Description

`GridSearchCV` allows the user to specify a Grid Search schema for tuning predictive model hyper-parameters with Cross-Validation. `GridSearchCV` gives the user complete flexibility in the predictive model and performance metrics.

Public fields

`learner` Predictive modeling function.

`scorer` List of performance metric functions.

`splitter` Function that splits data into cross validation folds.

`tune_params` Data.frame of full hyper-parameter grid created from `$tune_params`

Methods

Public methods:

- `GridSearchCV$fit()`
- `GridSearchCV$new()`
- `GridSearchCV$clone()`

Method `fit()`**:** `fit` tunes user-specified model hyper-parameters via Grid Search and Cross-Validation.

Usage:

```
GridSearchCV$fit(
  formula = NULL,
  data = NULL,
  x = NULL,
  y = NULL,
  response = NULL,
  convert_response = NULL,
  progress = FALSE
)
```

Arguments:

`formula` An object of class formula: a symbolic description of the model to be fitted.

`data` An optional data frame, or other object containing the variables in the model. If `data` is not provided, how `formula` is handled depends on `$learner`.

`x` Predictor data (independent variables), alternative interface to data with formula.

`y` Response vector (dependent variable), alternative interface to data with formula.

`response` String; In the absence of `formula` or `y`, this specifies which element of `learner_args` is the response vector.

`convert_response` Function; This should be a single function that transforms the response vector. E.g. a function converting a numeric binary variable to a factor variable.

`progress` Logical; indicating whether to print progress across the hyper-parameter grid.

Details: `fit` follows standard R modeling convention by surfacing a formula modeling interface as well as an alternate matrix option. The user should use whichever interface is supported by the specified `$learner` function.

Returns: An object of class FittedGridSearchCV.

Examples:

```
\donttest{
if (require(e1071) && require(rpart) && require(yardstick)) {
  iris_new <- iris[sample(1:nrow(iris), nrow(iris)), ]
  iris_new$Species <- factor(iris_new$Species == "virginica")
  iris_train <- iris_new[1:100, ]
  iris_validate <- iris_new[101:150, ]

  ### Decision Tree example

  iris_grid_cv <- GridSearchCV$new(
    learner = rpart::rpart,
    learner_args = list(method = "class"),
    tune_params = list(
      minsplit = seq(10, 30, by = 5),
      maxdepth = seq(20, 30, by = 2)
    ),
    splitter = cv_split,
    splitter_args = list(v = 3),
    scorer = list(accuracy = yardstick::accuracy_vec),
    optimize_score = "max",
    prediction_args = list(accuracy = list(type = "class"))
  )
  iris_grid_cv_fitted <- iris_grid_cv$fit(
    formula = Species ~ .,
    data = iris_train
  )

  ### Example with multiple metric functions

  iris_grid_cv <- GridSearchCV$new(
    learner = rpart::rpart,
    learner_args = list(method = "class"),
    tune_params = list(
      minsplit = seq(10, 30, by = 5),
      maxdepth = seq(20, 30, by = 2)
    ),
    splitter = cv_split,
    splitter_args = list(v = 3),
    scorer = list(
      accuracy = yardstick::accuracy_vec,
      auc = yardstick::roc_auc_vec
    ),
    optimize_score = "max",
    prediction_args = list(
      accuracy = list(type = "class"),
      auc = list(type = "prob")
```

```
    ),
    convert_predictions = list(
      accuracy = NULL,
      auc = function(i) i[, "FALSE"]
    )
  )
  iris_grid_cv_fitted <- iris_grid_cv$fit(
    formula = Species ~ .,
    data = iris_train,
  )

  # Grab the best model
  iris_grid_cv_fitted$best_model

  # Grab the best hyper-parameters
  iris_grid_cv_fitted$best_params

  # Grab the best model performance metrics
  iris_grid_cv_fitted$best_metric

  ### Matrix interface example - SVM

  mtcars_train <- mtcars[1:25, ]
  mtcars_eval <- mtcars[26:nrow(mtcars), ]

  mtcars_grid_cv <- GridSearchCV$new(
    learner = e1071::svm,
    tune_params = list(
      degree = 2:4,
      kernel = c("linear", "polynomial")
    ),
    splitter = cv_split,
    splitter_args = list(v = 2),
    learner_args = list(scale = TRUE),
    scorer = list(
      rmse = yardstick::rmse_vec,
      mae = yardstick::mae_vec
    ),
    optimize_score = "min"
  )
  mtcars_grid_cv_fitted <- mtcars_grid_cv$fit(
    x = mtcars_train[, -1],
    y = mtcars_train$mpg
  )

}
}
```

Method `new()`**:** Create a new GridSearchCV object.

Usage:

```
GridSearchCV$new(
  learner = NULL,
  tune_params = NULL,
  splitter = NULL,
  scorer = NULL,
  optimize_score = c("min", "max"),
  learner_args = NULL,
  splitter_args = NULL,
  scorer_args = NULL,
  prediction_args = NULL,
  convert_predictions = NULL
)
```

Arguments:

`learner` Function that estimates a predictive model. It is essential that this function support either a formula interface with `formula` and `data` arguments, or an alternate matrix interface with `x` and `y` arguments.

`tune_params` A named list specifying the arguments of `$learner` to tune.

`splitter` A function that computes cross validation folds from an input data set or a pre-computed list of cross validation fold indices. If `splitter` is a function, it must have a `data` argument for the input data, and it must return a list of cross validation fold indices. If `splitter` is a list of integers, the number of cross validation folds is `length(splitter)` and each element contains the indices of the data observations that are included in that fold.

`scorer` A named list of metric functions to evaluate model performance on `evaluation_data`. Any provided metric function must have `truth` and `estimate` arguments, for true outcome values and predicted outcome values respectively, and must return a single numeric metric value. The last metric function will be the one used to identify the optimal model from the Grid Search.

`optimize_score` One of "max" or "min"; Whether to maximize or minimize the metric defined in `scorer` to find the optimal Grid Search parameters.

`learner_args` A named list of additional arguments to pass to `learner`.

`splitter_args` A named list of additional arguments to pass to `splitter`.

`scorer_args` A named list of additional arguments to pass to `scorer`. `scorer_args` must either be length 1 or `length(scorer)` in the case where different arguments are being passed to each scoring function.

`prediction_args` A named list of additional arguments to pass to `predict`. `prediction_args` must either be length 1 or `length(scorer)` in the case where different arguments are being passed to each scoring function.

`convert_predictions` A list of functions to convert predicted values prior to being evaluated by the metric functions supplied in `scorer`. This list should either be length 1, in which case the same function will be applied to all predicted values, or `length(scorer)` in which case each function in `convert_predictions` will correspond with each function in `scorer`.

Returns: An object of class GridSearch.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

Examples

```
## ------------------------------------------------
## Method `GridSearchCV$fit`
## ------------------------------------------------


if (require(e1071) && require(rpart) && require(yardstick)) {
  iris_new <- iris[sample(1:nrow(iris), nrow(iris)), ]
  iris_new$Species <- factor(iris_new$Species == "virginica")
  iris_train <- iris_new[1:100, ]
  iris_validate <- iris_new[101:150, ]

  ### Decision Tree example

  iris_grid_cv <- GridSearchCV$new(
    learner = rpart::rpart,
    learner_args = list(method = "class"),
    tune_params = list(
      minsplit = seq(10, 30, by = 5),
      maxdepth = seq(20, 30, by = 2)
    ),
    splitter = cv_split,
    splitter_args = list(v = 3),
    scorer = list(accuracy = yardstick::accuracy_vec),
    optimize_score = "max",
    prediction_args = list(accuracy = list(type = "class"))
  )
  iris_grid_cv_fitted <- iris_grid_cv$fit(
    formula = Species ~ .,
    data = iris_train
  )

  ### Example with multiple metric functions

  iris_grid_cv <- GridSearchCV$new(
    learner = rpart::rpart,
    learner_args = list(method = "class"),
    tune_params = list(
      minsplit = seq(10, 30, by = 5),
      maxdepth = seq(20, 30, by = 2)
    ),
    splitter = cv_split,
    splitter_args = list(v = 3),
    scorer = list(
      accuracy = yardstick::accuracy_vec,
      auc = yardstick::roc_auc_vec
    ),
```

```
    optimize_score = "max",
    prediction_args = list(
      accuracy = list(type = "class"),
      auc = list(type = "prob")
    ),
    convert_predictions = list(
      accuracy = NULL,
      auc = function(i) i[, "FALSE"]
    )
  )
  iris_grid_cv_fitted <- iris_grid_cv$fit(
    formula = Species ~ .,
    data = iris_train,
  )

  # Grab the best model
  iris_grid_cv_fitted$best_model

  # Grab the best hyper-parameters
  iris_grid_cv_fitted$best_params

  # Grab the best model performance metrics
  iris_grid_cv_fitted$best_metric

  ### Matrix interface example - SVM

  mtcars_train <- mtcars[1:25, ]
  mtcars_eval <- mtcars[26:nrow(mtcars), ]

  mtcars_grid_cv <- GridSearchCV$new(
    learner = e1071::svm,
    tune_params = list(
      degree = 2:4,
      kernel = c("linear", "polynomial")
    ),
    splitter = cv_split,
    splitter_args = list(v = 2),
    learner_args = list(scale = TRUE),
    scorer = list(
      rmse = yardstick::rmse_vec,
      mae = yardstick::mae_vec
    ),
    optimize_score = "min"
  )
  mtcars_grid_cv_fitted <- mtcars_grid_cv$fit(
    x = mtcars_train[, -1],
    y = mtcars_train$mpg
  )

}
```

Index