tidymodels_notes

Software for Modeling

Models are very multifaceted & serve various purposes in statistical methods, but they are ultimately meant to disentangle complicated statistical phenomena.

Software used to create models should:

- have a easy-to-use interface
- protect users from making mistakes

Types of Models

Descriptive Models

Descriptive models are used generally to describe data attributes/traits, more of an opportunity to get to know your data better rather than make any kind of inference about a relationship related to the data

Inferential Models

Models of this nature are use to test a particular claim of a relationship between variables, a particular hypothesis, etc. After this model is created, a state of statistical significance/insignificance is produced

Output from these models usually includes a p-value, confidence intervals, etc. to estimate probabilites (i.e., of committing Type 1 error, etc.)

Predictive Models

These models deal with questions of estimation as opposed to inference

This type of model is used to provide accurate predictions utilizing previous/historical data

Building Predictive Models

- Mechanistic models: using predetermined equations/assumption to derive a model equation, can make statements about how the model will perform based on how it performs with existing data
- Empirically driven models: machine learning models, example: K-nearest neighbor (given a dataset, a new sample is speculated using the K most similar data in the set.)
  - note: "the primary method of evaluating the appropriateness of the model is to assess its accuracy using existing data"

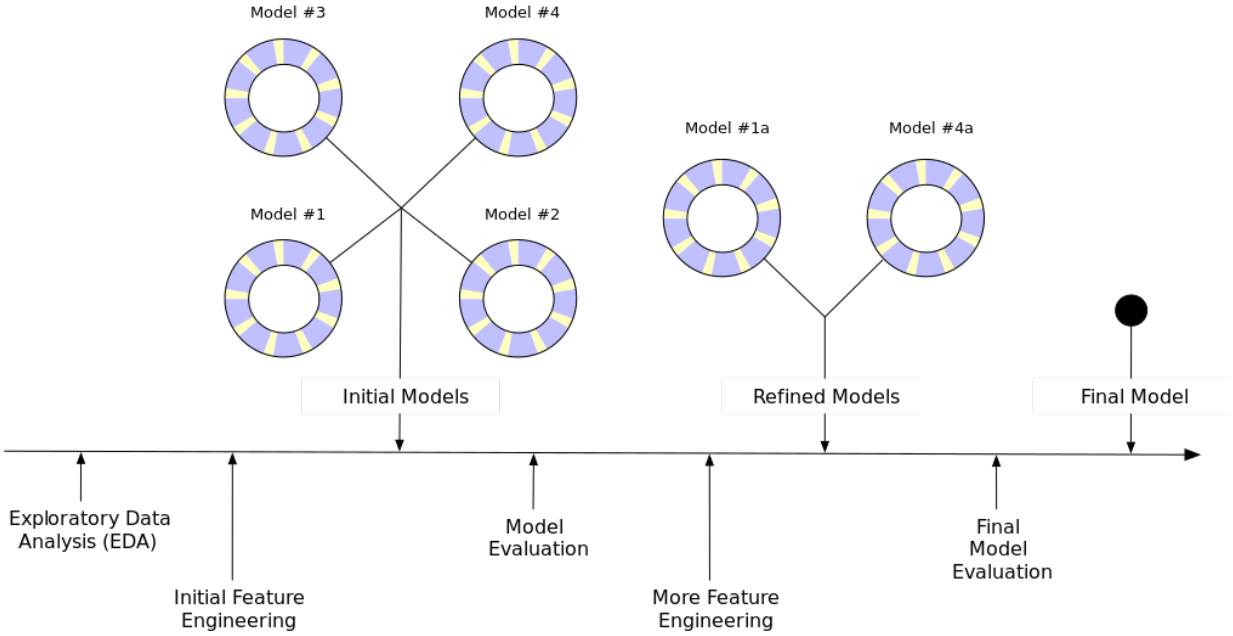
You may generate models that are statistically significant, but that doesn't necessarily mean its the best model for your data/represents your data appropriately

Terminology

- Unsupervised models: those models that learn patterns, clusters or other characteristics of the data but lack an outcome
- Supervised models: those models that have an outcome variable
  - Regression: predicts numeric outcome
  - Classification: produces ordered/unordered qualitative values

Phases of Data Analysis

- Cleaning data
  - involves looking closely at your data to see how it is presented/aligns with the research questions you are trying to answer
- Exploratory data analysis (EDA)
  - understanding how the data came about (sampling information, etc.); should also consider how appropriate/relevant the data are to your research
  - one should outline a clear goal for the model and how success with the model will be judged



Process for choosing appropriate model

- Exploratory data analysis (EDA)
  - oscillating between numerical analysis & data visualization
- Feature engineering
  - refiguring predictors to make them easier to model
- Model tuning and selection
  - creating initial models to assess & compare; usually involves parameter tuning, etc.
- Model evaluation
  - using formal evaluation techniques to compare model results, differences, etc.

Tidyverse notes

Gives broad explanation as to the differences between the tidyverse & base R, essentially providing evidence as to how tidyverse is more intutive & user-friendly for folks that are not super familiar with coding/programming. Goes over syntax, the importance of tibbles & pipes in the tidyverse and provides examples of how the tidyverse is super helpful in the modeling process

Review of R Modeling Fundamentals

Summarizes the workings of the R formula and typical functions that are utilized in model creation. As general principle, tidymodels relies heavily on the strength of R's existing functions (object-oriented programming) and emphasizes understandable defaults to function arguments, consistency across function names & arguments and user-friendliness (not restricting functions to specific data structures)

Briefly talks about combining base R w/ tidyverse that might be helpful in creating your models.

Tidymodels is essentially considered a "metapackage" with a core set of tidymodels & packages. It splits packages by their function (ex: data splitting/resampling w/ rsample, measuring performance w/ yardstick)

Basics (using Ames housing data)

models sale prices of homes in Ames, IA using a histograms (with transformed/not transformed data; i.e. log data)

conducting EDA (exploratory data analysis) to determine distributions of predictors, correlations between predictors or possible associations between predictors & outcomes

Spending our data

Steps to creating useful model:

- parameter estimation
- model selection
- tuning
- performance assessment

data spending: allocating specific subsets of data to do different things

common ways to split/"spend" data:

- split the larger dataset into two, using one as a training set to develop & optimize the model
  - the second set is the test set, used after 1-2 models are selected and the test set is used to produce these models & determine their efficacy
- r function initial_split() will perform this action for you, takes the dataset & desired ratio as arguments
  - training() and testing() will produce the necessary datasets with the object you just created with initial_split()

can either use simple random sampling/stratified sampling to contribute to the model (depends on the level of imbalance in the data) - the strata argument can be added to the initial_split() function to conduct stratified random sampling

**for time series data: initial_time_split() will save most recent data for the test set

validation sets can also be part of this data splitting process, getting a sense of model performance before using the test set

with multi-level data, have to keep in mind that data splitting should be happening according to the independent unit level of the data

Fitting models with parsnip

Parsnip is a tidymodels package that can be used to produce a variety of different models - the functions fit() and predict() are the most commonly used with a parsnip object

example: modeling linear regression

- base R: lm() for ordinary linear regression and stan_glm() for regularized linear regression (Bayesian); for non-Bayesian approach & regularized regression –> glmnet()
- tidymodels unifies this process, eliminates need to memorized different syntax for functions
  1) identify model type (linear regression, random forest, etc.)
  2) identify engine for fitting the model (i.e., software package)
  3) identify the mode of the model (regression/classification)

Example code:

```
linear_reg() %>% set_engine("lm")
# linear regression model specification, computational engine: lm

linear_reg() %>% set_engine("glmnet")
# linear regression model specification, computational engine: glmnet
```

you can use fit() or fit_xy() to begin model estimation; fit() is more appropriately used with a formula, fit_xy() is more appropriate if data is pre-processed

translate() will explain the process of the user's code being converted to package syntax

Example code:

```
linear_reg() %>% set_engine("lm") %>% translate()
# Linear Regression Model Specification (regression)
# Computational engine: lm
# Model fit template:
# stats::lm(formula = missing_arg(), data = missing_arg(), weights = missing_arg())

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

lm_form_fit <-
  lm_model %>%

fit(Sale_Price ~ Longitude + Latitude, data = ames_train)

lm_xy_fit <-
  lm_model %>%
  fit_xy(
      x = ames_train %>% select(Longitude, Latitude),
      y = ames_train %>% pull(Sale_Price)
    )

lm_form_fit

# Call:
# stats::lm(formula = Sale_Price ~ Longitude + Latitude, data = data)

# Coefficients:
# (Intercept)    Longitude     Latitude
#     -302.97        -2.07         2.71
#

lm_xy_fit

# parsnip model object
#
#
# Call:
# stats::lm(formula = ..y ~ ., data = data)
#
# Coefficients:
```

```
# (Intercept)    Longitude     Latitude
#     -302.97        -2.07         2.71
```

parsnip also provides consistency in model arguments ex: random forest function w/ parsnip (arguments)

- number of sampled predictors (mtry)
- number of trees (trees)
- number of data points to split (min_n)

model arguments in parsnip are separated into two different categories: - main arguments: commonly used & available across engines - engine arguments: specific to a particular engine, can be specified in set_engine()

Using Model Results

Examining model output using extract_fit_engine(), adding vcov() will return model output as matrix

However, matrices are not super malleable/easy to change & manipulate

- the broom package in tidymodels convert all different kinds of objects to a tidy structure
  - column names are standardized across models
  - row names converted to columns

Make predictions

predict() function

- returns results in tibble format
- column names are always predictable
- as many rows in the tibble as there are in the input data

these features make the merging of predictions & original data much easier

Table 6.4: The tidymodels mapping of prediction types and column names.

type value	column name(s)
numeric	.pred
class	.pred_class
prob	.pred_{class levels}
conf_int	.pred_lower, .pred_upper
pred_int	.pred_lower, .pred_upper

other packages can be used with parsnip, list found here

parsnip_addin() will provide a list of possible models for each model mode

A model workflow

Above is the typical structural equation for a linear model, including predictors (x) and outcome variable (y).

$$\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_{i1} + \ldots + \hat{\beta}_p x_{ip}$$

Figure 1: Model equation

a feature selection algorithm is mentioned to determine the minimum predictors needed for the model, wish i knew what that was lmao

model workflow: pre-processing steps, model fit and potential post-processing activities

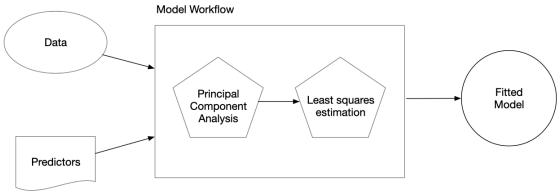


Figure 2: Correct mental model of where model estimation occurs in the data analysis process.

workflows package in R:

Example code:

```
lm_wflow <-
  workflow() %>%
  add_model(lm_model)

lm_wflow
#>   Workflow
#> Preprocessor: None
#> Model: linear_reg()
#>
#>   Model
#> Linear Regression Model Specification (regression)
#>
#> Computational engine: lm
```

*with a simple model, a formula would be added as a preprocessor using add_formula()

- fit() can be used to create the model; takes created workflow & data as arguments, use predict() after on the fitted workflow
- use update_formula() to update/remove model and/or preprocessor

you can add raw variables to your workflow by using add_variables(); takes outcomes & predictors as arguments - any outcome variables accidentally used in the predictors will be removed; fit() then takes the ready workflow & training set data

Formulas & Workflow

- different kinds of encoding procedures for formulas are necessary for different statistical methods
- example: tree-based models
  - when fitting a random forest model using ranger/randomForest, the workflow will know to leave the predictors columns that are factors alone
- determinations regarding the need to create dummy variables, etc. is made for each model & engine combination

  - ex: add_model() can take a formula argument to specify model formula

Multiple Workflows

can be helpful when trying to build predictive models (evaluate different model types)

good for sequential testing (beginning with full model and gradually taking out predictors)

workflowset creates combos of components of workflow (list of preprocessors & model specifications that be matched)

Example code:

```
  location <- list(
    longitude = Sale_Price ~ Longitude,
    latitude = Sale_Price ~ Latitude,
    coords = Sale_Price ~ Longitude + Latitude,
    neighborhood = Sale_Price ~ Neighborhood
    )

  library(workflowsets)
  location_models <- workflow_set(preproc = location, models = list(lm = lm_model))

  location_models
#> # A workflow set/tibble: 4 × 4
#>   wflow_id        info             option    result
#>   <chr>           <list>           <list>    <list>
#> 1 longitude_lm    <tibble [1 × 4]> <opts[0]> <list [0]>
#> 2 latitude_lm     <tibble [1 × 4]> <opts[0]> <list [0]>
#> 3 coords_lm       <tibble [1 × 4]> <opts[0]> <list [0]>
#> 4 neighborhood_lm <tibble [1 × 4]> <opts[0]> <list [0]>

location_models$info[[1]]
#> # A tibble: 1 × 4
#>   workflow   preproc model      comment
#>   <list>     <chr>   <chr>      <chr>
#> 1 <workflow> formula linear_reg ""
extract_workflow(location_models, id = "coords_lm")

#>   Workflow
#> Preprocessor: Formula
#> Model: linear_reg()
#>
#>   Preprocessor
#> Sale_Price ~ Longitude + Latitude
#>
#>   Model
#> Linear Regression Model Specification (regression)
#>
#> Computational engine: lm
```

Feature engineering with recipes

- feature engineering involves reformatting predictors to make them easier to use in a model
  - can be done with recipes package; combines preprocessing/feature engineering tasks into one object to then apply to different data sets

- a recipe essentially works like a formula without actually executing anything
  - it lists out what should be done

Formula

Example code:

```
lm(Sale_Price ~ Neighborhood + log10(Gr_Liv_Area) + Year_Built + Bldg_Type, data = ames)
```

Recipe

Example code:

```
library(tidymodels) # Includes the recipes package
tidymodels_prefer()

simple_ames <-
  recipe(Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type,
         data = ames_train) %>%
  step_log(Gr_Liv_Area, base = 10) %>%
  step_dummy(all_nominal_predictors())

simple_ames

#> Recipe
#>
#> Inputs:
#>
#>       role #variables
#>    outcome          1
#>  predictor          4
#>
#> Operations:
#>
#> Log transformation on Gr_Liv_Area
#> Dummy variables from all_nominal_predictors()
```

step_dummy() converts qualitative data to dummy/indicator variables; all_nominal_predictors() is a function that captures predictor columns that are nominal in nature

advantages of recipes:

- computations can be reused across models
- recipes offer broader set of choices for data processing not available with formulas
- compact syntax
- data processing can be captured in single R object

if adding recipes to existing workflow: - remember to remove variables if previous preprocessing method was used: remove_variables(), then use add_recipe()

use new workflow & training data to call fit() and use the predict() function to test data; extract_recipe() can be used to retrieve recipe details after it's been estimated

Example code:

```
  lm_fit %>%
    extract_recipe(estimated = TRUE)
#> Recipe
#>
```

```
#> Inputs:
#>
#>       role #variables
#>    outcome          1
#>  predictor          4
#>
#> Training data contained 2342 data points and no missing data.
#>
#> Operations:
#>
#> Log transformation on Gr_Liv_Area [trained]
#> Dummy variables from Neighborhood, Bldg_Type [trained]

lm_fit %>%
  # This returns the parsnip object:
  extract_fit_parsnip() %>%
  # Now tidy the linear model object:
  tidy() %>%
  slice(1:5)
#> # A tibble: 5 × 5
#>   term                        estimate std.error statistic   p.value
#>   <chr>                          <dbl>     <dbl>     <dbl>     <dbl>
#> 1 (Intercept)                -0.669    0.231        -2.90 3.80e-  3
#> 2 Gr_Liv_Area                 0.620    0.0143       43.2  2.63e-299
#> 3 Year_Built                  0.00200  0.000117     17.1  6.16e- 62
#> 4 Neighborhood_College_Creek  0.0178   0.00819       2.17 3.02e-  2
#> 5 Neighborhood_Old_Town      -0.0330   0.00838      -3.93 8.66e-  5
```

How data are used by the recipe

1) when calling recipe(), data serves as an argument of the function to determine the data types for each column and facilitate future use of selectors like all_numeric(), etc.
2) when calling fit(), the training data are used for estimation operations
3) when calling predict(), takes new test data and values are standardized based on training data values

Recipes package capabilities

-encoding qualitative data (ex: step_other() can be used to group infrequent values into a catch-all level) - step_dummy() to create dummy/indicator variables for factor predictors - also allows for user to customize the naming of the dummy variable created

Interactions

specified using step_interact(~ interaction term) *can use selectors in interaction term*

Example code:

```
simple_ames <-
  recipe(Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type,
         data = ames_train) %>%
  step_log(Gr_Liv_Area, base = 10) %>%
  step_other(Neighborhood, threshold = 0.01) %>%
  step_dummy(all_nominal_predictors()) %>%

  # Gr_Liv_Area is on the log scale from a previous step
```

```
  step_interact( ~ Gr_Liv_Area:starts_with("Bldg_Type_") )
```

Spline functions

spline terms replace existing numeric predictors with a set of columns that allow the model to emulate a non-linear relationship

Example code:

```
recipe(Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type + Latitude,
       data = ames_train) %>%
  step_log(Gr_Liv_Area, base = 10) %>%
  step_other(Neighborhood, threshold = 0.01) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_interact( ~ Gr_Liv_Area:starts_with("Bldg_Type_") ) %>%
  step_ns(Latitude, deg_free = 20)
```

Feature extraction

- if creating new features from existing predictors (a la PCA), step_pca() can be used in recipe
  - step_normalize() might be needed before if the columns used are not all measured the same

Row sampling steps

- downsampling: keeps minority class & takes random sample of majority so class frequencies are balanced
- upsampling: replicates samples from minority class to balance the classes
- hybrid: combination of both

themis package has recipe steps for this, ex. from it: step_downsample(outcome_column_name)

More

- step_mutate() can be used for general transformations
- textrecipes package can be used to apply natural language processing methods to data (ex: split string of text into separate words, etc.)
- subsampling processes, etc. should not be applied to the data being predicted
  - each step function mentioned has a skip option; set it to TRUE so that it will be ignored by the predict function
  - all steps are then applied when using fit()

Tidy a recipe

when using tidy() along with your recipe object, a summary of the recipe steps will appear

Example code:

```
tidy(ames_rec)
#> # A tibble: 5 × 6
#>   number operation type     trained skip  id
#>    <int> <chr>     <chr>    <lgl>   <lgl> <chr>
#> 1      1 step      log      FALSE   FALSE log_66JTU
#> 2      2 step      other    FALSE   FALSE other_ePfcw
#> 3      3 step      dummy    FALSE   FALSE dummy_Z18Cl
#> 4      4 step      interact FALSE   FALSE interact_JLU36
#> 5      5 step      ns       FALSE   FALSE ns_rvsqQ
```

can specific id column if wanting to tidy() it

Example code:

```
ames_rec <-
  recipe(Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type +
           Latitude + Longitude, data = ames_train) %>%
  step_log(Gr_Liv_Area, base = 10) %>%
  step_other(Neighborhood, threshold = 0.01, id = "my_id") %>%
  step_dummy(all_nominal_predictors()) %>%
  step_interact( ~ Gr_Liv_Area:starts_with("Bldg_Type_") ) %>%
  step_ns(Latitude, Longitude, deg_free = 20)
```

refit workflow with new recipe, call tidy() with new object

Example code:

```
estimated_recipe <-
  lm_fit %>%
  extract_recipe(estimated = TRUE)

tidy(estimated_recipe, id = "my_id")
```

```
#> # A tibble: 22 × 3
#>   terms        retained           id
#>   <chr>        <chr>              <chr>
#> 1 Neighborhood North_Ames         my_id
#> 2 Neighborhood College_Creek      my_id
#> 3 Neighborhood Old_Town           my_id
#> 4 Neighborhood Edwards            my_id
#> 5 Neighborhood Somerset           my_id
#> 6 Neighborhood Northridge_Heights my_id
#> # … with 16 more rows
```

step numbers can also be used in the tidy() call

Column Roles

- columns used in the formula in recipe() are usually assigned a "predictor" or "outcome" depending on which side of the tilde they're on
  - however, if there's a column you'd like to keep that does not fall into one of those categories, you can use add_roles(), update_roles(), remove_roles()
  - can be helpful when data are resampled

  Example code:

  ames_rec %>% update_role(address, new_role = "street address")

all possible recipe steps found here

Judging model effectiveness

tidymodels emphasizes empirical validation (using data not used to create the model to measure effectiveness) - performance metrics are typically used in empirical validation - it's important to choose an appropriate method, as different metrics demonstrate different things (ex: RMSE measures accuracy, R-squared measures correlation) - model results should always be accompanied by some measure of fidelity to the data

Regression metrics

using yardstick tidymodels package, have prediction functions

- uses test data set & already created linear regression model to compare predicted & observed sale prices of housing data

Example code:

```
ames_test_res <- predict(lm_fit, new_data = ames_test %>% select(-Sale_Price))
ames_test_res
```

```
#> # A tibble: 588 × 1
#>   .pred
#>   <dbl>
#> 1  5.07
#> 2  5.31
#> 3  5.28
#> 4  5.33
#> 5  5.30
#> 6  5.24
#> # … with 582 more rows
```

```
ames_test_res <- bind_cols(ames_test_res, ames_test %>% select(Sale_Price))
ames_test_res
#> # A tibble: 588 × 2
#>   .pred Sale_Price
#>   <dbl>      <dbl>
#> 1  5.07       5.02
#> 2  5.31       5.39
#> 3  5.28       5.28
#> 4  5.33       5.28
#> 5  5.30       5.28
#> 6  5.24       5.26
#> # … with 582 more rows
```

```
ggplot(ames_test_res, aes(x = Sale_Price, y = .pred)) +
# Create a diagonal line:
  geom_abline(lty = 2) +
  geom_point(alpha = 0.5) +
  labs(y = "Predicted Sale Price (log10)", x = "Sale Price (log10)") +
  # Scale and size the x- and y-axis uniformly:
  coord_obs_pred()
```

can compute RMSE using rmse() function

Example code:

```
rmse(ames_test_res, truth = Sale_Price, estimate = .pred)
#> # A tibble: 1 × 3
#>   .metric .estimator .estimate
#>   <chr>   <chr>          <dbl>
#> 1 rmse    standard      0.0736
```

a metric set can be created to compare multiple metrics

Example code:

```
ames_metrics <- metric_set(rmse, rsq, mae)
ames_metrics(ames_test_res, truth = Sale_Price, estimate = .pred)
#> # A tibble: 3 × 3
#>   .metric .estimator .estimate
#>   <chr>   <chr>          <dbl>
#> 1 rmse    standard      0.0736
#> 2 rsq     standard      0.836
#> 3 mae     standard      0.0549
```

Binary classification metrics

using modeldata package w/ example predictions from test data set

Example code:

```
data(two_class_example)
str(two_class_example)
#> 'data.frame':    500 obs. of  4 variables:
#>  $ truth    : Factor w/ 2 levels "Class1","Class2": 2 1 2 1 2 1 1 1 2 2 ...
#>  $ Class1   : num  0.00359 0.67862 0.11089 0.73516 0.01624 ...
#>  $ Class2   : num  0.996 0.321 0.889 0.265 0.984 ...
#>  $ predicted: Factor w/ 2 levels "Class1","Class2": 2 1 2 1 2 1 1 1 2 2 ...
```

hard class prediction functions:

- conf_mat()
- accuracy()
- mcc() for Matthews correlation coefficient
- f_meas() for F1 metric

classification_metrics() for combining metrics together

Example code:

```
# A confusion matrix:
conf_mat(two_class_example, truth = truth, estimate = predicted)
#>           Truth
#> Prediction Class1 Class2
#>     Class1    227     50
#>     Class2     31    192

# Accuracy:
accuracy(two_class_example, truth, predicted)
#> # A tibble: 1 × 3
#>   .metric  .estimator .estimate
#>   <chr>    <chr>          <dbl>
#> 1 accuracy binary         0.838

# Matthews correlation coefficient:
mcc(two_class_example, truth, predicted)
#> # A tibble: 1 × 3
#>   .metric .estimator .estimate
#>   <chr>   <chr>          <dbl>
#> 1 mcc     binary         0.677

# F1 metric:
f_meas(two_class_example, truth, predicted)
#> # A tibble: 1 × 3
```

```
#>   .metric .estimator .estimate
#>   <chr>   <chr>          <dbl>
#> 1 f_meas  binary         0.849

# Combining these three classification metrics together
classification_metrics <- metric_set(accuracy, mcc, f_meas)
classification_metrics(two_class_example, truth = truth, estimate = predicted)
#> # A tibble: 3 × 3
#>   .metric  .estimator .estimate
#>   <chr>    <chr>          <dbl>
#> 1 accuracy binary         0.838
#> 2 mcc      binary         0.677
#> 3 f_meas   binary         0.849
```

can specify event level when, for example, the second class is the event

using predicted probabilities as inputs

- ROC (receiver operating characteristic) curve computes sensitivity/specificity over continuum of different event thresholds
- two yardstick functions: roc_curve() to compute data points that make up the curve and roc_auc() that computes the area under the curve

Example code:

```
two_class_curve <- roc_curve(two_class_example, truth, Class1)
two_class_curve
#> # A tibble: 502 × 3
#>   .threshold specificity sensitivity
#>        <dbl>       <dbl>       <dbl>
#> 1 -Inf           0                 1
#> 2    1.79e-7     0                 1
#> 3    4.50e-6     0.00413           1
#> 4    5.81e-6     0.00826           1
#> 5    5.92e-6     0.0124            1
#> 6    1.22e-5     0.0165            1
#> # … with 496 more rows

roc_auc(two_class_example, truth, Class1)
#> # A tibble: 1 × 3
#>   .metric .estimator .estimate
#>   <chr>   <chr>          <dbl>
#> 1 roc_auc binary         0.939
```

autoplot() will plot details of curve

functions that use probability estimates

- gain_curve()
- lift_curve()
- pr_curve()

Multi-class classification metrics

same functions as the binary classification metrics

Example code:

```
accuracy(hpc_cv, obs, pred)
```

```
#> # A tibble: 1 × 3
#>   .metric  .estimator .estimate
#>   <chr>    <chr>          <dbl>
#> 1 accuracy multiclass     0.709

mcc(hpc_cv, obs, pred)
#> # A tibble: 1 × 3
#>   .metric .estimator .estimate
#>   <chr>   <chr>          <dbl>
#> 1 mcc     multiclass     0.515
```

wrapper methods can be used to apply sensitivity to outcomes that are more than 2 classes:

- macro-averaging
- macro-weighted averaging
- micro-averaging

Example code:

```
sensitivity(hpc_cv, obs, pred, estimator = "macro")

#> # A tibble: 1 × 3
#>   .metric     .estimator .estimate
#>   <chr>       <chr>          <dbl>
#> 1 sensitivity macro          0.560

sensitivity(hpc_cv, obs, pred, estimator = "macro_weighted")

#> # A tibble: 1 × 3
#>   .metric     .estimator     .estimate
#>   <chr>       <chr>              <dbl>
#> 1 sensitivity macro_weighted     0.709

sensitivity(hpc_cv, obs, pred, estimator = "micro")

#> # A tibble: 1 × 3
#>   .metric     .estimator .estimate
#>   <chr>       <chr>          <dbl>
#> 1 sensitivity micro          0.709
```

roc_auc() can also be used, need to pass all class probability columns into the function

Example code:

```
roc_auc(hpc_cv, obs, VF, F, M, L)
#> # A tibble: 1 × 3
#>   .metric .estimator .estimate
#>   <chr>   <chr>          <dbl>
#> 1 roc_auc hand_till      0.829

roc_auc(hpc_cv, obs, VF, F, M, L, estimator = "macro_weighted")
#> # A tibble: 1 × 3
#>   .metric .estimator     .estimate
#>   <chr>   <chr>              <dbl>
#> 1 roc_auc macro_weighted     0.868
```

metrics can also be computed using dplyr grouping mechanisms

Example code:

```
hpc_cv %>%
  group_by(Resample) %>%
  accuracy(obs, pred)
#> # A tibble: 10 × 4
#>   Resample .metric  .estimator .estimate
#>   <chr>    <chr>    <chr>          <dbl>
#> 1 Fold01   accuracy multiclass     0.726
#> 2 Fold02   accuracy multiclass     0.712
#> 3 Fold03   accuracy multiclass     0.758
#> 4 Fold04   accuracy multiclass     0.712
#> 5 Fold05   accuracy multiclass     0.712
#> 6 Fold06   accuracy multiclass     0.697
#> # … with 4 more rows

# Four 1-vs-all ROC curves for each fold
hpc_cv %>%
  group_by(Resample) %>%
  roc_curve(obs, VF, F, M, L) %>%
  autoplot()
```

Resampling for evaluating performance

making model assessments with resubstitution approach comparing linear & random forest models

Example code:

```
estimate_perf <- function(model, dat) {
  # Capture the names of the objects used
  cl <- match.call()
  obj_name <- as.character(cl$model)
  data_name <- as.character(cl$dat)
  data_name <- gsub("ames_", "", data_name)

  # Estimate these metrics:
  reg_metrics <- metric_set(rmse, rsq)

  model %>%
    predict(dat) %>%
    bind_cols(dat %>% select(Sale_Price)) %>%
    reg_metrics(Sale_Price, .pred) %>%
    select(-.estimator) %>%
    mutate(object = obj_name, data = data_name)
}
```

code creates new function to estimate model performance, using RMSE & rsq to compare

Example code:

```
estimate_perf(rf_fit, ames_train)

#> # A tibble: 2 × 4
#>   .metric .estimate object data
#>   <chr>       <dbl> <chr>  <chr>
#> 1 rmse       0.0365 rf_fit train
```

```
#> 2 rsq        0.960  rf_fit train

estimate_perf(lm_fit, ames_train)
#> # A tibble: 2 × 4
#>   .metric .estimate object data
#>   <chr>       <dbl> <chr>  <chr>
#> 1 rmse       0.0754 lm_fit train
#> 2 rsq        0.816  lm_fit train
```

random forest RMSE metric is better, use random forest with test set

Example code:

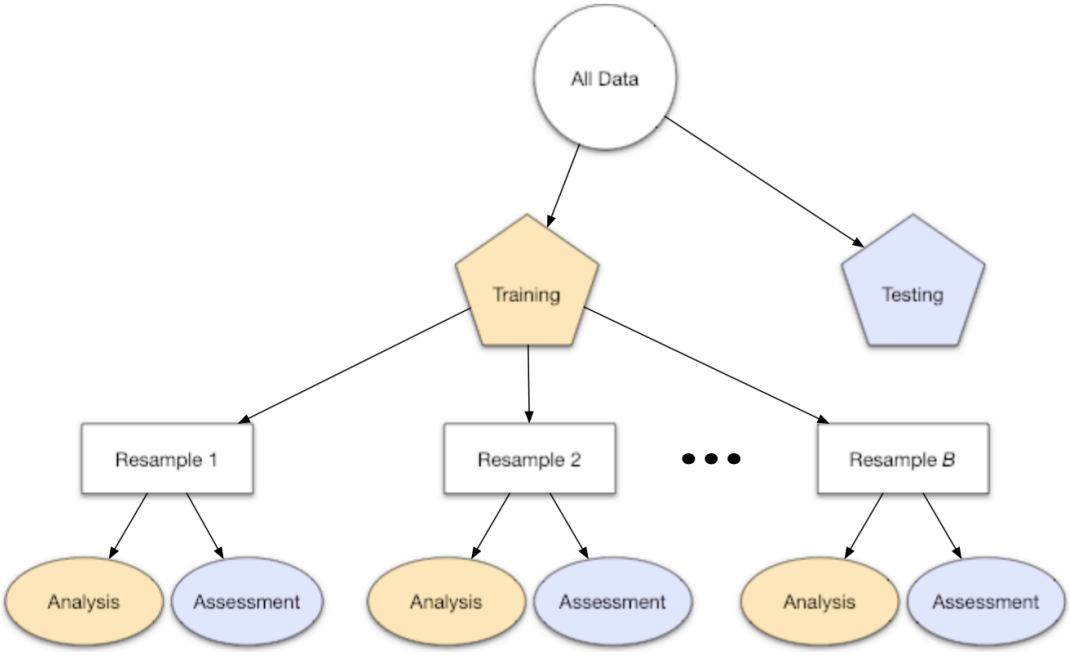
```
estimate_perf(rf_fit, ames_test)
#> # A tibble: 2 × 4
#>   .metric .estimate object data
#>   <chr>       <dbl> <chr>  <chr>
#> 1 rmse       0.0704 rf_fit test
#> 2 rsq        0.852  rf_fit test
```

RMSE is way worse: why?

- complexity of random forest allows for the model to basically memorize training set data

solution: resampling methods

Resampling methods



resampling is conducted on the training set:

- data is divided into two subsamples
  - model fit w/ analysis set
  - model evaluated w/ assessment set

Cross-validation

most common: V-fold cross-validation

- data are partitioned into V sets of roughly equal size (called "folds")
- for example, in 3-fold validation, three iterations are created and one fold is held out for each iteration for assessment statistics

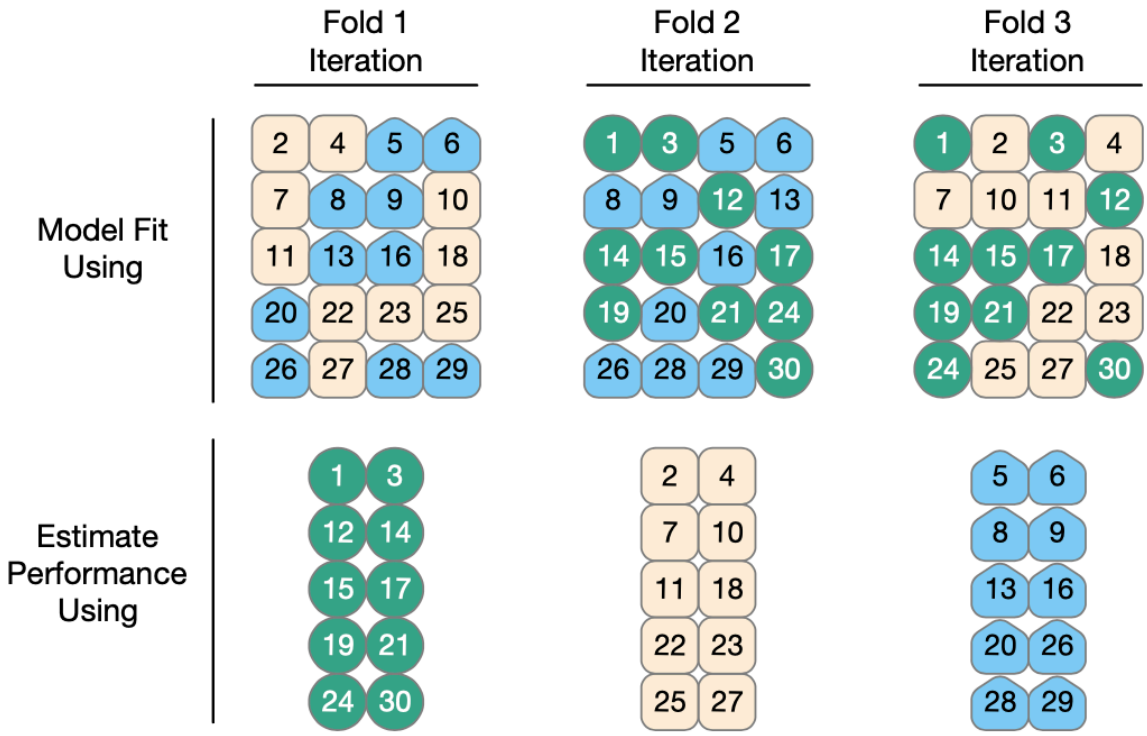


Figure 10.3: V-fold cross-validation data usage.

Figure 3: V-fold cross-validation data usage.

values of v should generally be larger in practice, most often 5 or 10

Example code:

```
set.seed(1001)

ames_folds <- vfold_cv(ames_train, v = 10)

ames_folds
#> #  10-fold cross-validation

#> # A tibble: 10 × 2

#>   splits             id
#>   <list>             <chr>
#> 1 <split [2107/235]> Fold01
#> 2 <split [2107/235]> Fold02
#> 3 <split [2108/234]> Fold03
#> 4 <split [2108/234]> Fold04
#> 5 <split [2108/234]> Fold05
```

```
#> 6 <split [2108/234]> Fold06
#> # … with 4 more rows
```

- splits column holds information regarding data split
- the functions analysis() and assessment() return corresponding data frames

Example:

```
# For the first fold:

ames_folds$splits[[1]] %>% analysis() %>% dim()

#> [1] 2107   74
```

Repeated cross-validation

creating R repeats of V-fold cross-validation will decrease noisiness in initial estimate

- repeat the fold generation R time to generate R collections of V partitions (V x R produces the final resampling estimate)
- the summ. statistics from each model tend toward normal distribution (CLT)

Example code:

```
vfold_cv(ames_train, v = 10, repeats = 5)
#> #  10-fold cross-validation repeated 5 times
#> # A tibble: 50 × 3
#>   splits             id      id2
#>   <list>             <chr>   <chr>
#> 1 <split [2107/235]> Repeat1 Fold01
#> 2 <split [2107/235]> Repeat1 Fold02
#> 3 <split [2108/234]> Repeat1 Fold03
#> 4 <split [2108/234]> Repeat1 Fold04
#> 5 <split [2108/234]> Repeat1 Fold05
#> 6 <split [2108/234]> Repeat1 Fold06
#> # … with 44 more rows
```

Leave-one-out cross-validation

V is the number of data points in the training set; if there are n samples of traning set data, n models are fit using n-1 rows of the training set

- each model will predict excluded point, and the n predictions are pooled at the end of the resampling
- computationally excessive tbh

Monte Carlo cross-validation

data is randomly selected to the assessment sets

Example code:

```
mc_cv(ames_train, prop = 9/10, times = 20)

#> # Monte Carlo cross-validation (0.9/0.1) with 20 resamples

#> # A tibble: 20 × 2

#>   splits             id
#>   <list>             <chr>
```

```
#> 1 <split [2107/235]> Resample01
#> 2 <split [2107/235]> Resample02
#> 3 <split [2107/235]> Resample03
#> 4 <split [2107/235]> Resample04
#> 5 <split [2107/235]> Resample05
#> 6 <split [2107/235]> Resample06
#> # … with 14 more rows
```

Validation sets

usually used when original data set is super large; data is first split into testing/non testing data and the non testing data is then split into training and validation sets

Example code:

```
set.seed(1002)

val_set <- validation_split(ames_train, prop = 3/4)

val_set
```

```
#> # Validation Set Split (0.75/0.25)
#> # A tibble: 1 × 2
#>   splits             id
#>   <list>             <chr>
#> 1 <split [1756/586]> validation
```

Bootstrapping

taking a bootstrap sample (i.e. sampling w/ replacement) that is the same size as the training set

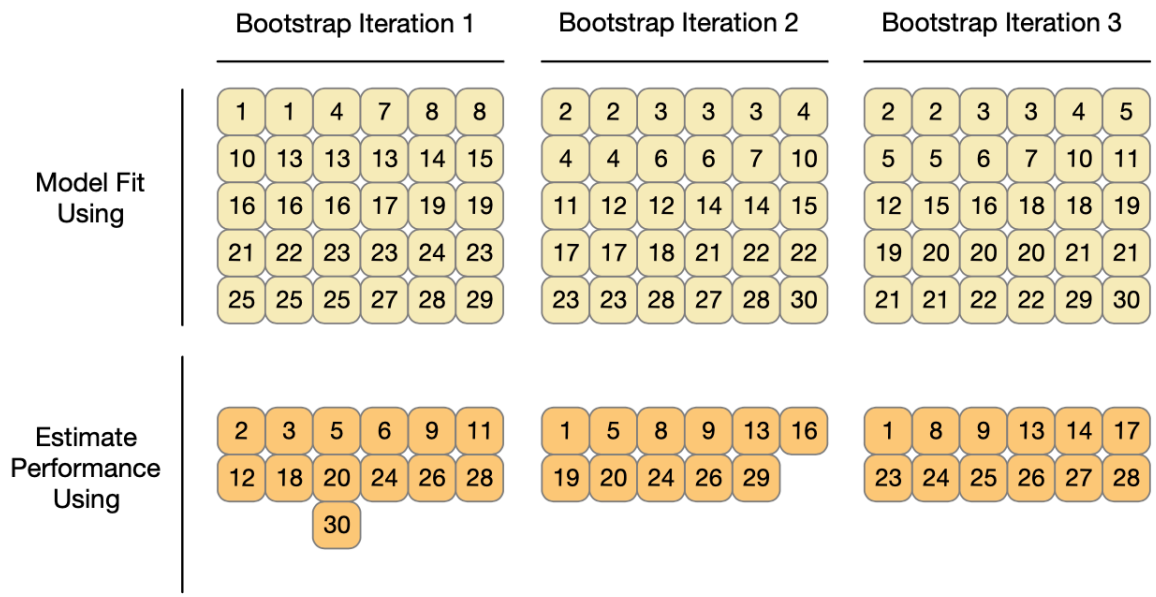


Figure 10.7: Bootstrapping data usage.

Example code:

```
bootstraps(ames_train, times = 5)
```

```
#> # Bootstrap sampling
#> # A tibble: 5 × 2
#>   splits             id
#>   <list>             <chr>
#> 1 <split [2342/858]> Bootstrap1
#> 2 <split [2342/855]> Bootstrap2
#> 3 <split [2342/852]> Bootstrap3
#> 4 <split [2342/851]> Bootstrap4
#> 5 <split [2342/867]> Bootstrap5
```

Rolling forecasting origin resampling

used when data has a time component (when models need to estimate seasonal/temporal trends)

- estimates model using historical data and evaluating with recent data

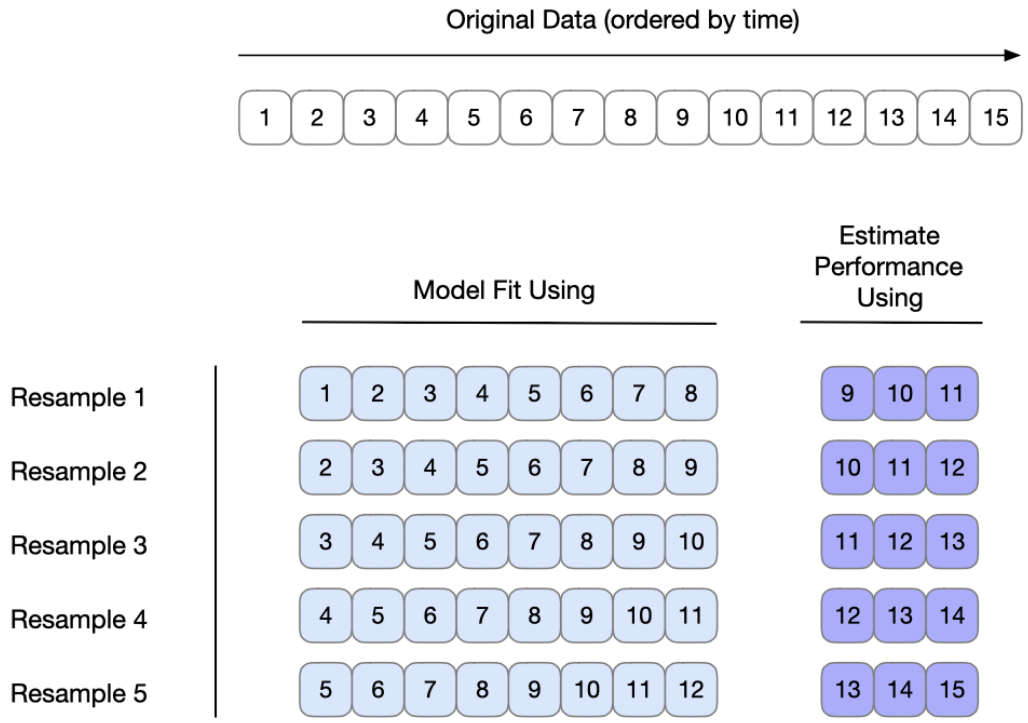


Figure 10.8: Data usage for rolling forecasting origin resampling.

Figure 4: Data usage for rolling forecasting origin resampling.

Example code:

```
time_slices <-
  tibble(x = 1:365) %>%
  rolling_origin(initial = 6 * 30, assess = 30, skip = 29, cumulative = FALSE)

data_range <- function(x) {
  summarize(x, first = min(x), last = max(x))
}
```

```
map_dfr(time_slices$splits, ~   analysis(.x) %>% data_range())

#> # A tibble: 6 × 2
#>   first  last
#>   <int> <int>
#> 1     1   180
#> 2    31   210
#> 3    61   240
#> 4    91   270
#> 5   121   300
#> 6   151   330

map_dfr(time_slices$splits, ~ assessment(.x) %>% data_range())

#> # A tibble: 6 × 2
#>   first  last
#>   <int> <int>
#> 1   181   210
#> 2   211   240
#> 3   241   270
#> 4   271   300
#> 5   301   330
#> 6   331   360
```

Estimating performance

- analysis set is used to preprocess data and use preprocessed data to fit the model
- stats produced by the analysis set are applied to assessment set; predictions from this set estimate model performance

Example code:

```
model_spec %>% fit_resamples(formula,  resamples, ...)
model_spec %>% fit_resamples(recipe,   resamples, ...)
workflow   %>% fit_resamples(          resamples, ...)
```

-optional arguments: metrics (metric set of performance stats), control

- arguments for control: verbose (logical for printing logging), extract (function for retaining objects from each model iteration), save_pred (logical for saving assessment set predictions)

Example code:

```
keep_pred <- control_resamples(save_pred = TRUE, save_workflow = TRUE)

set.seed(1003)

rf_res <-
  rf_wflow %>%
  fit_resamples(resamples = ames_folds, control = keep_pred)

rf_res

#> # Resampling results
#> # 10-fold cross-validation
#> # A tibble: 10 × 5
```

```
#>   splits             id     .metrics         .notes           .predictions
#>   <list>             <chr>  <list>           <list>           <list>
#> 1 <split [2107/235]> Fold01 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [235 × 4]>
#> 2 <split [2107/235]> Fold02 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [235 × 4]>
#> 3 <split [2108/234]> Fold03 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [234 × 4]>
#> 4 <split [2108/234]> Fold04 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [234 × 4]>
#> 5 <split [2108/234]> Fold05 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [234 × 4]>
#> 6 <split [2108/234]> Fold06 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [234 × 4]>
#> # … with 4 more rows
```

to collect performance metric tibble (summarized over replicates):

```
collect_metrics(rf_res)
```

```
#> # A tibble: 2 × 6

#>   .metric .estimator   mean     n std_err .config
#>   <chr>   <chr>       <dbl> <int>   <dbl> <chr>
#> 1 rmse    standard   0.0721    10 0.00305 Preprocessor1_Model1
#> 2 rsq     standard   0.831     10 0.0108  Preprocessor1_Model1
```

to collect predictions from assessment set:

```
assess_res <- collect_predictions(rf_res)

assess_res
```

```
#> # A tibble: 2,342 × 5
#>   id     .pred  .row Sale_Price .config
#>   <chr>  <dbl> <int>      <dbl> <chr>
#> 1 Fold01  5.10    10       5.09 Preprocessor1_Model1
#> 2 Fold01  4.92    27       4.90 Preprocessor1_Model1
#> 3 Fold01  5.21    47       5.08 Preprocessor1_Model1
#> 4 Fold01  5.13    52       5.10 Preprocessor1_Model1
#> 5 Fold01  5.13    59       5.10 Preprocessor1_Model1
#> 6 Fold01  5.13    63       5.11 Preprocessor1_Model1
#> # … with 2,336 more rows
```

Example code (to compare observed and held-out predicted values):

```
assess_res %>%
  ggplot(aes(x = Sale_Price, y = .pred)) +
  geom_point(alpha = .15) +
  geom_abline(color = "red") +
  coord_obs_pred() +
  ylab("Predicted")
```

Example code (to isolate overpredicted data point):

```
over_predicted <-
  assess_res %>%
  mutate(residual = Sale_Price - .pred) %>%
  arrange(desc(abs(residual))) %>%
  slice(1)
over_predicted
```

```
#> # A tibble: 1 × 6
#>   id     .pred  .row Sale_Price .config              residual
```

```
#>   <chr>  <dbl> <int>      <dbl> <chr>                  <dbl>
#> 1 Fold09  4.97    32       4.11 Preprocessor1_Model1   -0.858

ames_train %>%
  slice(over_predicted$.row) %>%
  select(Gr_Liv_Area, Neighborhood, Year_Built, Bedroom_AbvGr, Full_Bath)
#> # A tibble: 1 × 5
#>   Gr_Liv_Area Neighborhood Year_Built Bedroom_AbvGr Full_Bath
#>         <int> <fct>             <int>         <int>     <int>
#> 1         832 Old_Town           1923             2         1
```

Example code (for using a validation set instead of cross-validation):

```
val_res <- rf_wflow %>% fit_resamples(resamples = val_set)
val_res
#> # Resampling results
#> # Validation Set Split (0.75/0.25)
#> # A tibble: 1 × 4
#>   splits             id         .metrics         .notes
#>   <list>             <chr>      <list>           <list>
#> 1 <split [1756/586]> validation <tibble [2 × 4]> <tibble [0 × 3]>

collect_metrics(val_res)
#> # A tibble: 2 × 6
#>   .metric .estimator   mean     n std_err .config
#>   <chr>   <chr>       <dbl> <int>   <dbl> <chr>
#> 1 rmse    standard   0.0695     1      NA Preprocessor1_Model1
#> 2 rsq     standard   0.843      1      NA Preprocessor1_Model1
```

Parallel processing

in tidymodels: the *tune* package uses the *foreach* package to do parallel computations

- on a single computer, the # of possible "worker processes" is determined by the package *parallel*

Example code:

```
# The number of physical cores in the hardware:
```

parallel::detectCores(logical = FALSE) #> [1] 2

```
# The number of possible independent processes that can
# be simultaneously used:
parallel::detectCores(logical = TRUE)
#> [1] 2
```

need to use below code to register a parallel backend package and then execute parallel processing:

```
# Unix and macOS only
library(doMC)
registerDoMC(cores = 2)

# Now run fit_resamples()...
```

(you can use registerDoSEQ() to reset computations to sequential)

alternative option, using network sockets:

```
# All operating systems
library(doParallel)
```

```
# Create a cluster object and then register:
cl <- makePSOCKcluster(2)
registerDoParallel(cl)

# Now run fit_resamples()`...

stopCluster(cl)
```

be aware of how parallel processing can slow down your computations considerably

Saving the resampled objects

models created during resampling are not typically saved, but there is an option to extract them (extract option in control_resamples())

Example code:

```
ames_rec <-
  recipe(Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type +
           Latitude + Longitude, data = ames_train) %>%
  step_other(Neighborhood, threshold = 0.01) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_interact( ~ Gr_Liv_Area:starts_with("Bldg_Type_") ) %>%
  step_ns(Latitude, Longitude, deg_free = 20)

lm_wflow <-
  workflow() %>%
  add_recipe(ames_rec) %>%
  add_model(linear_reg() %>% set_engine("lm"))

lm_fit <- lm_wflow %>% fit(data = ames_train)

# Select the recipe:
extract_recipe(lm_fit, estimated = TRUE)
#> Recipe
#>
#> Inputs:
#>
#>       role #variables
#>    outcome          1
#>  predictor          6
#>
#> Training data contained 2342 data points and no missing data.
#>
#> Operations:
#>
#> Collapsing factor levels for Neighborhood [trained]
#> Dummy variables from Neighborhood, Bldg_Type [trained]
#> Interactions with Gr_Liv_Area:(Bldg_Type_TwoFmCon + Bldg_Type_Duplex + B... [trained]
#> Natural splines on Latitude, Longitude [trained]

get_model <- function(x) {
  extract_fit_parsnip(x) %>% tidy()
}
```

```
# Test it using:
# get_model(lm_fit)
```

extracting resampled fits:

```
ctrl <- control_resamples(extract = get_model)

lm_res <- lm_wflow %>%  fit_resamples(resamples = ames_folds, control = ctrl)

lm_res
```

```
#> # Resampling results
#> # 10-fold cross-validation
#> # A tibble: 10 × 5
#>   splits             id     .metrics         .notes           .extracts
#>   <list>             <chr>  <list>           <list>           <list>
#> 1 <split [2107/235]> Fold01 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [1 × 2]>
#> 2 <split [2107/235]> Fold02 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [1 × 2]>
#> 3 <split [2108/234]> Fold03 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [1 × 2]>
#> 4 <split [2108/234]> Fold04 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [1 × 2]>
#> 5 <split [2108/234]> Fold05 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [1 × 2]>
#> 6 <split [2108/234]> Fold06 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [1 × 2]>
#> # … with 4 more rows
```

getting replicates for a single predictor:

```
all_coef <- map_dfr(lm_res$.extracts, ~ .x[[1]][[1]])

# Show the replicates for a single predictor:

filter(all_coef, term == "Year_Built")
```

```
#> # A tibble: 10 × 5
#>   term       estimate std.error statistic  p.value
#>   <chr>         <dbl>     <dbl>     <dbl>    <dbl>
#> 1 Year_Built  0.00180  0.000149      12.1 1.57e-32
#> 2 Year_Built  0.00180  0.000151      12.0 6.45e-32
#> 3 Year_Built  0.00185  0.000150      12.3 1.00e-33
#> 4 Year_Built  0.00183  0.000147      12.5 1.90e-34
#> 5 Year_Built  0.00184  0.000150      12.2 2.47e-33
#> 6 Year_Built  0.00180  0.000150      12.0 3.35e-32
#> # … with 4 more rows
```

Comparing models with resampling