Modeling in the Tidyverse - Resampling and Parameter Tuning

Max Kuhn (RStudio)

Loading

```r
library(tidymodels)
```

```
## ── Attaching packages ─────────────────────────────────────────── tidymodels 0.0.2 ──
```

```
## ✔ broom     0.5.2     ✔ purrr     0.3.2
## ✔ dials     0.0.2     ✔ recipes   0.1.5
## ✔ dplyr     0.8.1     ✔ rsample   0.0.4
## ✔ ggplot2   3.1.0     ✔ tibble    2.1.1
## ✔ infer     0.4.0     ✔ yardstick 0.0.2
## ✔ parsnip   0.0.2
```

```
## ── Conflicts ────────────────────────────────────────────── tidymodels_conflicts() ──
## ✖ purrr::discard() masks scales::discard()
## ✖ dplyr::filter()  masks stats::filter()
## ✖ dplyr::lag()     masks stats::lag()
## ✖ recipes::step()  masks stats::step()
```

Previously

```r
library(AmesHousing)

ames <-
  make_ames() %>%
  dplyr::select(-matches("Qu"))

set.seed(4595)
data_split <- initial_split(ames, strata = "Sale_Price")
ames_train <- training(data_split)
ames_test  <- testing(data_split)

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

perf_metrics <- metric_set(rmse, rsq, ccc)
```

```r
ames_rec <-
  recipe(Sale_Price ~ Bldg_Type + Neighborhood + Year_Built +
           Gr_Liv_Area + Full_Bath + Year_Sold + Lot_Area +
           Central_Air + Longitude + Latitude,
         data = ames_train) %>%
  step_log(Sale_Price, base = 10) %>%
  step_BoxCox(Lot_Area, Gr_Liv_Area) %>%
  step_other(Neighborhood, threshold = 0.05)  %>%
  step_dummy(all_nominal()) %>%
  step_interact(~ starts_with("Central_Air"):Year_Built) %>%
  step_bs(Longitude, Latitude, options = list(df = 5))
```

Resampling

Resampling Methods

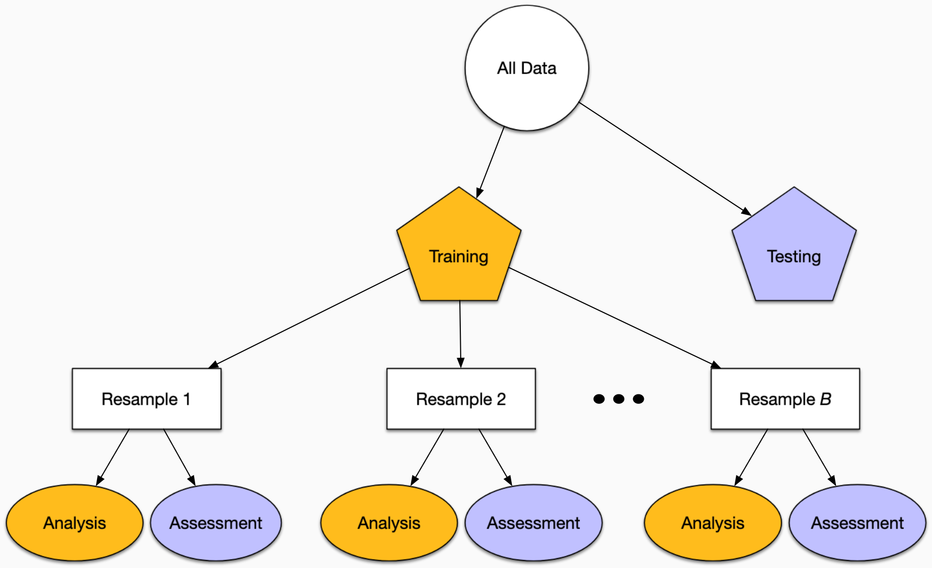
These are additional data splitting schemes that are applied to the *training* set and are used for **estimating model performance**.

They attempt to simulate slightly different versions of the training set. These versions of the original are split into two model subsets:

- The *analysis set* is used to fit the model (analogous to the training set).
- Performance is determined using the *assessment set*.

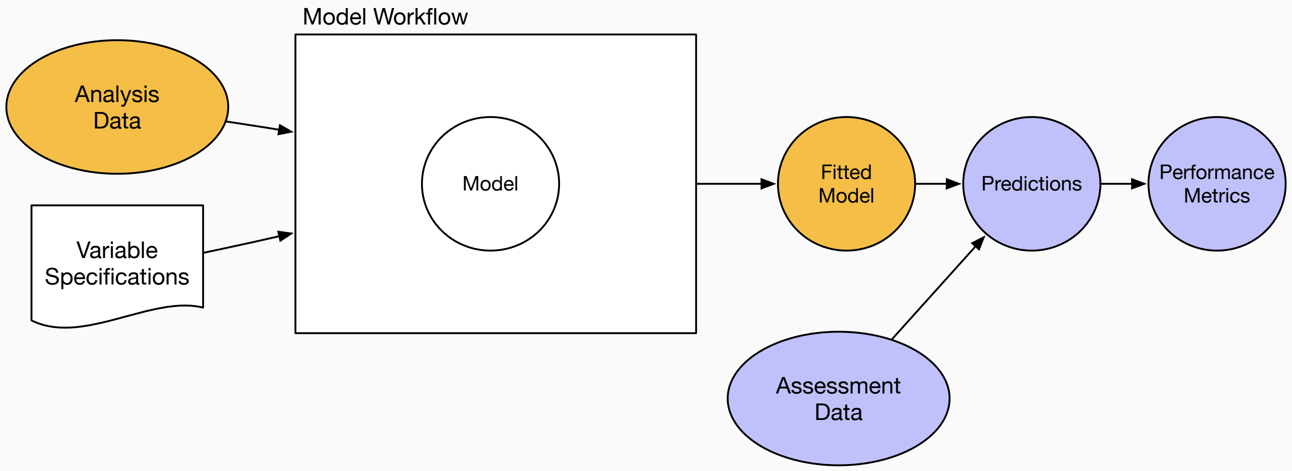
This process is repeated many times.

There are different flavors of resampling but we will focus on one method in these notes.



The Model Workflow and Resampling

All resampling methods repeat this process multiple times:



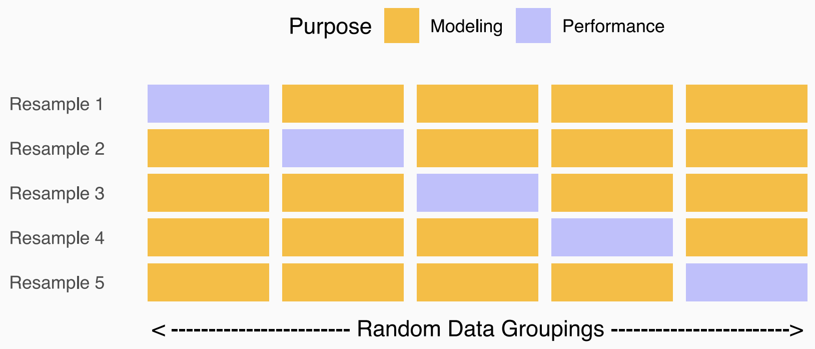
The final resampling estimate is the average of all of the estimated metrics (e.g. RMSE, etc).

V-Fold Cross-Validation

Here, we randomly split the training data into *V* distinct blocks of roughly equal size (AKA the "folds").

- We leave out the first block of analysis data and fit a model.
- This model is used to predict the held-out block of assessment data.
- We continue this process until we've predicted all *V* assessment blocks

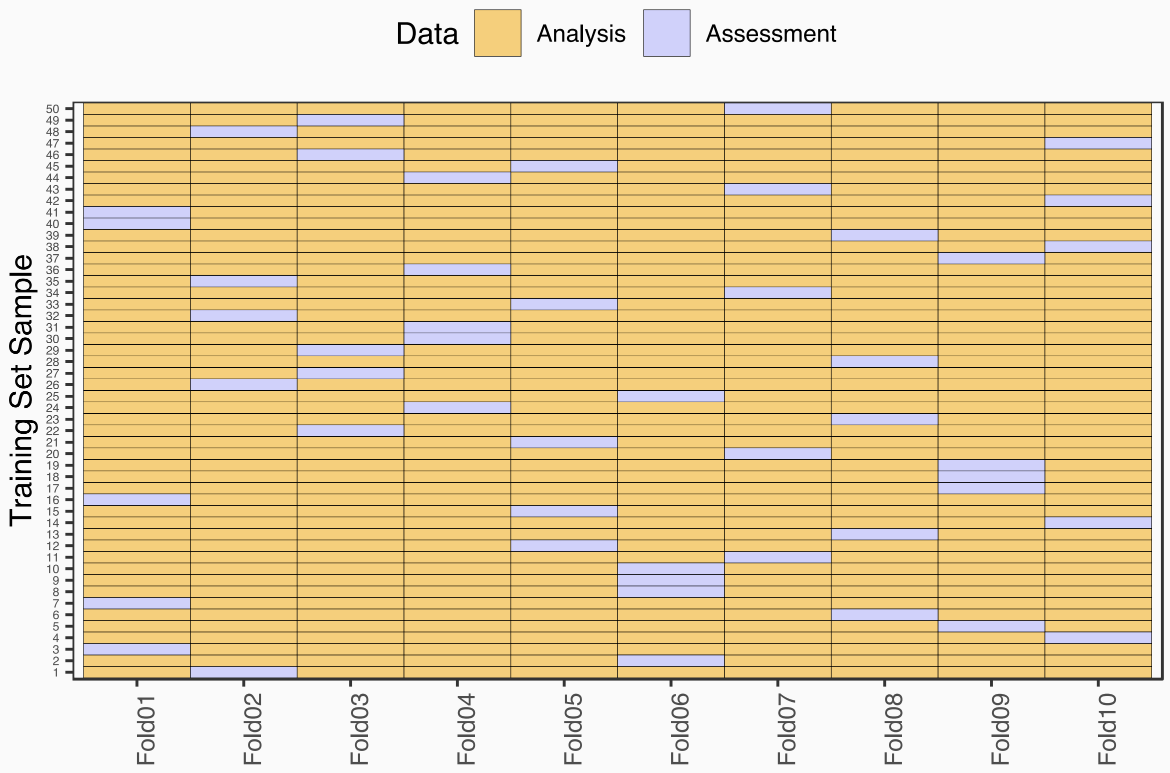
The final performance is based on the hold-out predictions by *averaging* the statistics from the *V* blocks.



V is usually taken to be 5 or 10 and leave-one-out cross-validation has each sample as a block.

Repeated CV can be used when trianing set sizes are small. 5 repeats of 10-fold CV averages 50 sets of metrics.

10-Fold Cross-Validation with n = 50



Resampling Results

The goal of resampling is to produce a single estimate of perforamnce for a model.

Even though we end up estimating V models (for V-fold CV), these models are discarded after we have our performance estimate.

Resampling is basically an *emprical simulation system* used to understand how well the model would work on *new data*.

Cross-Validating Using `rsample`

```r
set.seed(2453)
cv_splits <- vfold_cv(
  data = ames_train,
  v = 10,
  strata = "Sale_Price"
)
cv_splits %>% slice(1:6)
```

```
## #  10-fold cross-validation using stratification
## # A tibble: 6 x 2
##   splits           id
## * <list>           <chr>
## 1 <split [2K/222]> Fold01
## 2 <split [2K/222]> Fold02
## 3 <split [2K/222]> Fold03
## 4 <split [2K/222]> Fold04
## 5 <split [2K/222]> Fold05
## 6 <split [2K/219]> Fold06
```

Each individual split object is similar to the `initial_split()` example.

```r
cv_splits$splits[[1]]
```

```
## <1977/222/2199>
```

```r
cv_splits$splits[[1]] %>% analysis() %>% dim()
```

```
## [1] 1977   74
```

```r
cv_splits$splits[[1]] %>% assessment() %>% dim()
```

```
## [1] 222  74
```

Resampling the Linear Model

Working with resample tibbles generally involves two things:

1) Small functions that perform an action on a single split.

2) The `purrr` package for `map()` ping over splits.

```
geo_form <- log10(Sale_Price) ~ Latitude + Longitude

# Fit on a single analysis resample
fit_model <- function(split, spec) {
  fit(
    object = spec,
    formula = geo_form,
    # Pull out the data used for estimation:
    data = analysis(split)
  )
}

# For each resample, call fit_model()
cv_splits <- cv_splits %>%
  mutate(models_lm = map(splits, fit_model, spec_lm))

cv_splits
```

```
## #  10-fold cross-validation using stratification
## # A tibble: 10 x 3
##    splits             id     models_lm
##  * <list>             <chr>  <list>
##  1 <split [2K/222]> Fold01 <fit[+]>
##  2 <split [2K/222]> Fold02 <fit[+]>
##  3 <split [2K/222]> Fold03 <fit[+]>
##  4 <split [2K/222]> Fold04 <fit[+]>
##  5 <split [2K/222]> Fold05 <fit[+]>
##  6 <split [2K/219]> Fold06 <fit[+]>
##  7 <split [2K/219]> Fold07 <fit[+]>
##  8 <split [2K/217]> Fold08 <fit[+]>
##  9 <split [2K/217]> Fold09 <fit[+]>
## 10 <split [2K/217]> Fold10 <fit[+]>
```

Resampling the Linear Model

Next, we will attach the predictions for each resample:

```r
lm_pred <- function(split, model) {

  # Extract the assessment set
  assess <- assessment(split) %>%
    mutate(Sale_Price_Log = log10(Sale_Price))

  # Compute predictions (a df is returned)
  pred <- predict(model, new_data = assess)

  bind_cols(assess, pred)
}
```

```r
cv_splits <- cv_splits %>%
  mutate(
    pred_lm = map2(splits, models_lm, lm_pred)
  )
cv_splits
```

```
## #  10-fold cross-validation using stratification
## # A tibble: 10 x 4
##    splits             id     models_lm pred_lm
##  * <list>             <chr>  <list>    <list>
##  1 <split [2K/222]> Fold01 <fit[+]>  <tibble [222 × 76]>
##  2 <split [2K/222]> Fold02 <fit[+]>  <tibble [222 × 76]>
##  3 <split [2K/222]> Fold03 <fit[+]>  <tibble [222 × 76]>
##  4 <split [2K/222]> Fold04 <fit[+]>  <tibble [222 × 76]>
##  5 <split [2K/222]> Fold05 <fit[+]>  <tibble [222 × 76]>
##  6 <split [2K/219]> Fold06 <fit[+]>  <tibble [219 × 76]>
##  7 <split [2K/219]> Fold07 <fit[+]>  <tibble [219 × 76]>
##  8 <split [2K/217]> Fold08 <fit[+]>  <tibble [217 × 76]>
##  9 <split [2K/217]> Fold09 <fit[+]>  <tibble [217 × 76]>
## 10 <split [2K/217]> Fold10 <fit[+]>  <tibble [217 × 76]>
```

Resampling the Linear Model

Now, let's compute performance measures as before:

```r
lm_metrics <- function(pred_df) {
  # Use our previous yardstick function
  perf_metrics(
    pred_df,
    truth = Sale_Price_Log,
    estimate = .pred
  )
}
```

```r
cv_splits <- cv_splits %>%
  mutate(perf_lm = map(pred_lm, lm_metrics))

select(cv_splits, pred_lm, perf_lm)
```

```
## # A tibble: 10 x 2
##    pred_lm             perf_lm
##    <list>              <list>
##  1 <tibble [222 × 76]> <tibble [3 × 3]>
##  2 <tibble [222 × 76]> <tibble [3 × 3]>
##  3 <tibble [222 × 76]> <tibble [3 × 3]>
##  4 <tibble [222 × 76]> <tibble [3 × 3]>
##  5 <tibble [222 × 76]> <tibble [3 × 3]>
##  6 <tibble [219 × 76]> <tibble [3 × 3]>
##  7 <tibble [219 × 76]> <tibble [3 × 3]>
##  8 <tibble [217 × 76]> <tibble [3 × 3]>
##  9 <tibble [217 × 76]> <tibble [3 × 3]>
## 10 <tibble [217 × 76]> <tibble [3 × 3]>
```

Resampling the Linear Model

And finally, let's compute the average of each metric over the resamples:

```
cv_splits$perf_lm[[1]]
```

```
## # A tibble: 3 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard       0.150
## 2 rsq     standard       0.303
## 3 ccc     standard       0.391
```

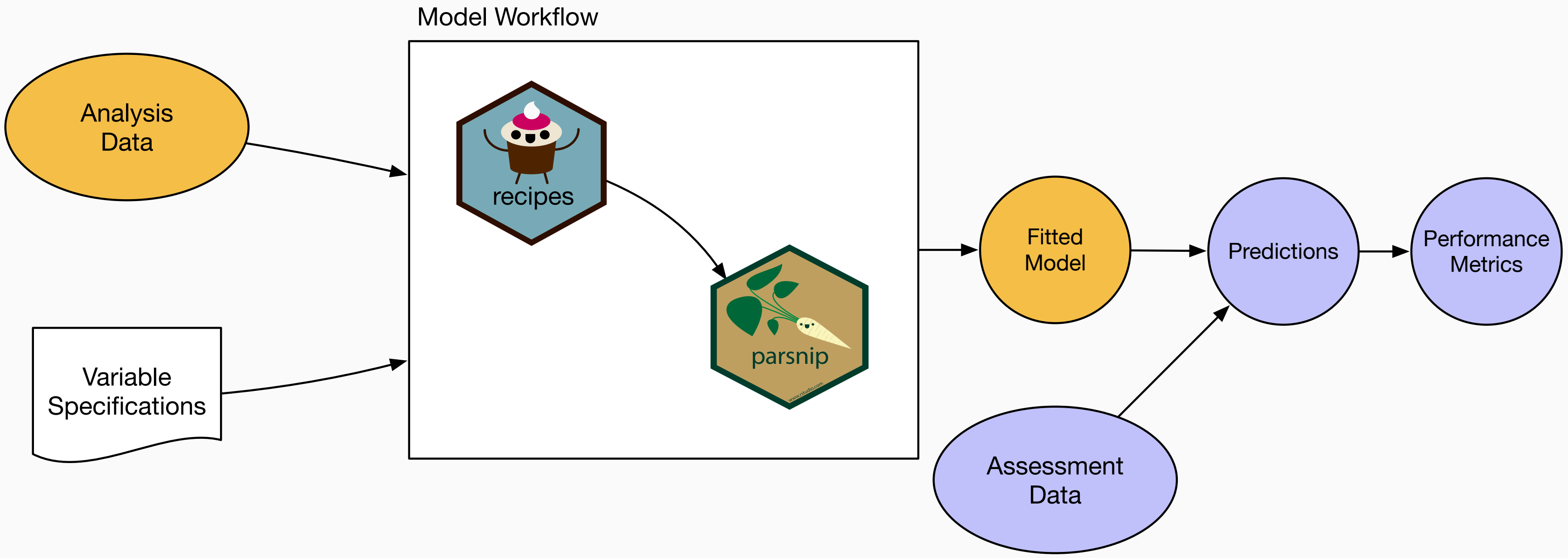
```
cv_splits %>%
  unnest(perf_lm) %>%
  group_by(.metric) %>%
  summarise(
    .avg = mean(.estimate),
    .sd = sd(.estimate)
  )
```

```
## # A tibble: 3 x 3
##   .metric  .avg     .sd
##   <chr>   <dbl>   <dbl>
## 1 ccc     0.303 0.0471
## 2 rmse    0.161 0.00787
## 3 rsq     0.183 0.0569
```

Adding Recipes

Working with Preprocessing and Recipes

How do we incorporate a recipe? We prepare the recipe within each analysis set:



Preparing the Recipes

Our first step is the run `prep()` on the `ames_rec` recipe but using each of the analysis sets.

`recipes` has a function that is a wrapper around `prep()` that can be used to map over the split objects, prepping on the analysis set of each one:

```
cv_splits <-
  cv_splits %>%
  # remove previous linear model stuff
  dplyr::select(-models_lm, -pred_lm, -perf_lm ) %>%
  mutate(lm_rec = map(splits, prepper, recipe = ames_rec))
cv_splits %>% slice(1:4)
```

```
## #  10-fold cross-validation using stratification
## # A tibble: 4 x 3
##   splits            id     lm_rec
## * <list>            <chr>  <list>
## 1 <split [2K/222]> Fold01 <recipe>
## 2 <split [2K/222]> Fold02 <recipe>
## 3 <split [2K/222]> Fold03 <recipe>
## 4 <split [2K/222]> Fold04 <recipe>
```

Fitting the Models

We can use code that is very similar to the previous section.

This code will use the recipe object to get the data. Since each analysis set is used to train the recipe, our previous use of `retain = TRUE` means that the processed version of the data is within the recipe. This can be returned via the `juice()` function.

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

parsnip_fit <- function(rec_obj, model) {
  fit(model, Sale_Price ~ ., data = juice(rec_obj))
}
```

```
cv_splits <- cv_splits %>%
  mutate(
    lm = map(lm_rec, parsnip_fit, model = spec_lm)
  )

glance(cv_splits$lm[[1]]$fit)
```

```
## # A tibble: 1 x 11
##   r.squared adj.r.squared  sigma statistic p.value    df logLik    AIC    BIC deviance df.residual
##       <dbl>         <dbl>  <dbl>     <dbl>   <dbl> <int>  <dbl>  <dbl>  <dbl>    <dbl>       <int>
## 1     0.803         0.801 0.0796      319.       0    26  2211. -4368. -4217.     12.4        1951
```

Predicting the Assessment Set

This is a little more complex. We need three elements contained in our tibble:

- the split object (to get the assessment data)
- the recipe object (to process the data)
- the linear model (for predictions)

```
cv_splits %>%
  select(splits, lm_rec, lm) %>%
  slice(1:4)
```

```
## #  10-fold cross-validation using stratification
## # A tibble: 4 x 3
##   splits             lm_rec   lm
## * <list>             <list>   <list>
## 1 <split [2K/222]> <recipe> <fit[+]>
## 2 <split [2K/222]> <recipe> <fit[+]>
## 3 <split [2K/222]> <recipe> <fit[+]>
## 4 <split [2K/222]> <recipe> <fit[+]>
```

The function is not too bad:

```
parsnip_metrics <- function(split, recipe, model) {
  raw_assessment <- assessment(split)
  processed <- bake(recipe, new_data = raw_assessment)

  model %>%
    predict(new_data = processed) %>%
    # Add the baked assessment data back in
    bind_cols(processed) %>%
    perf_metrics(Sale_Price, .pred)
}
```

How to do iterate over three columns at once? `map`, `map2`, ... ?

Predicting the Assessment Set

Since we have three inputs, we will use `purrr::pmap()` to walk along all three columns in the tibble.

```
cv_splits <- cv_splits %>%
  mutate(
    lm_res = pmap(
      list(
        split  = splits,
        recipe = lm_rec,
        model  = lm
      ),
      parsnip_metrics
    )
  )
```

```
cv_splits %>%
  select(splits, lm_rec, lm, lm_res) %>%
  slice(1:4)
```

```
## #  10-fold cross-validation using stratification
## # A tibble: 4 x 4
##   splits            lm_rec   lm       lm_res
## * <list>            <list>   <list>   <list>
## 1 <split [2K/222]> <recipe> <fit[+]> <tibble [3 × 3]>
## 2 <split [2K/222]> <recipe> <fit[+]> <tibble [3 × 3]>
## 3 <split [2K/222]> <recipe> <fit[+]> <tibble [3 × 3]>
## 4 <split [2K/222]> <recipe> <fit[+]> <tibble [3 × 3]>
```

We do get some warnings that the assessment data are outside the range of the analysis set values:

```
## Warning in bs(x = c(-93.6235954, -93.636372, -93.627536, -93.65332,
## -93.649447, : some 'x' values beyond boundary knots may cause ill-
## conditioned bases
```

Where are we Extrapolating?

Lincoln Highway
Nevada
240th Street
240th Street
240th Street
240th Street
+
−
Leaflet | Map tiles by Stamen Design, CC BY 3.0 — Map data © OpenStreetMap

Predicting the Assessment Set

```
cv_splits %>%
  unnest(lm_res) %>%
  group_by(.metric) %>%
  summarise(
    resampled_estimate = mean(.estimate)
  )
```

```
## # A tibble: 3 x 2
##   .metric resampled_estimate
##   <chr>                <dbl>
## 1 ccc                 0.884
## 2 rmse                0.0799
## 3 rsq                 0.800
```

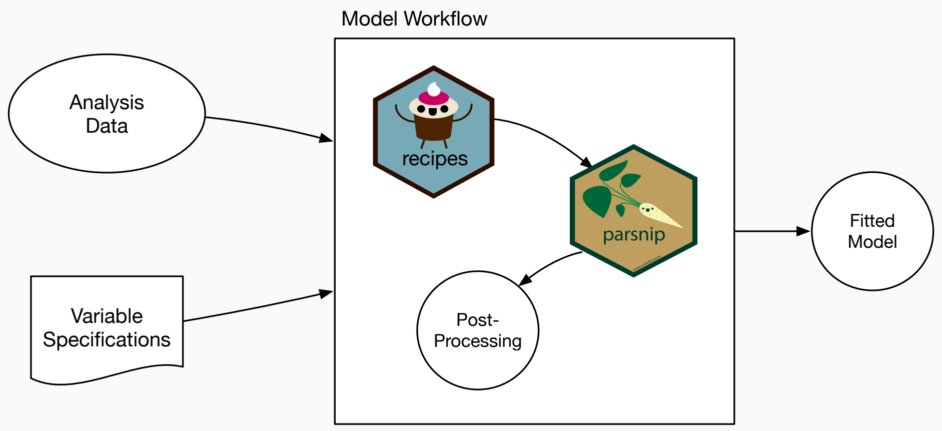
Workflows

We are in the process of making a new package called "`workflows`" that bundles together the operations that go "in the model process box". This would include:

- pre-processing steps (via a recipe)
- model estimation (from `parsnip`)
- post-processing steps (cut-point optimization, calibration, etc)

A workflow would have its own `fit` function that would prepare the recipe, fit the model, etc. A simple `prediction` function would also be available.

This will make the code much higher-level.



Hands-On

The tibble `cv_splits` has a column of resampled recipes (`lm_rec`).

`broom::tidy()` can be used to get nicely formatted versions of some objects.

Read `tidy.recipe()` to understand how to use this with a recipe object.

Question: What was the range of the Box-Cox transformation parameter for `Gr_Liv_Area` over the resamples?

10:00

Model Tuning

Tuning Parameters

There sre some models with parameters that *cannot be directly estimated from the data*.

For example, K-nearest neighbors stores the training set (including the outcome).

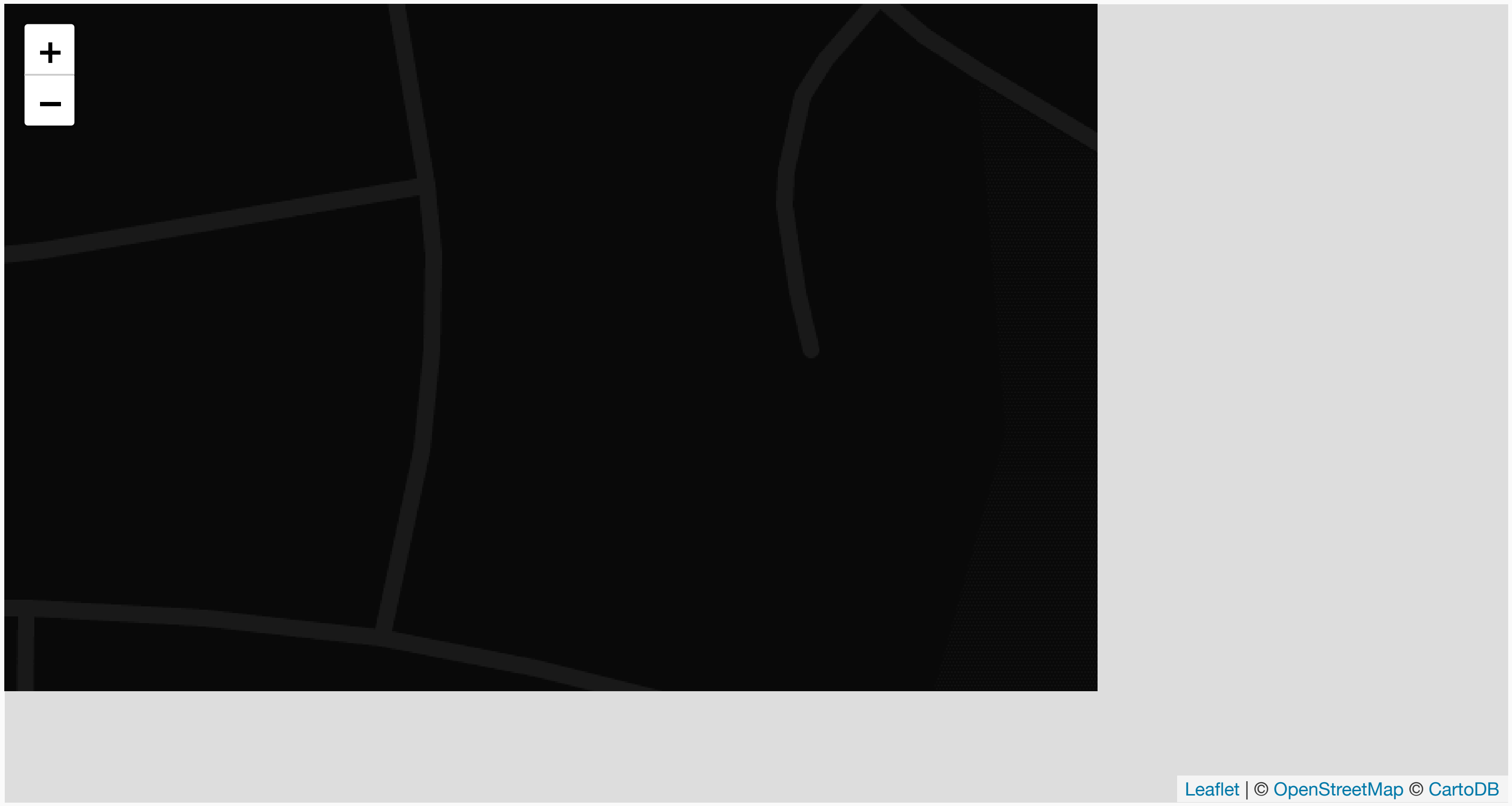
When a new sample is predicted, K training set points are found that are most similar to the new sample being predicted.

The predicted value for the new sample is some summary statistic of the neighbors, usually:

- the mean for regression, or
- the mode for classification.

There is no formula that will tell you what K should be. We need a way to estimate it.

5-Nearest Neighbors Model



Tuning Parameters and Overfitting

Overfitting occurs when a model inappropriately picks up on trends in the training set that do not generalize to new samples.

When this occurs, assessments of the model based on the training set can show good performance that does not reproduce in future samples.

For example, $K = 1$ is much more likely to overfit the data than larger values since they average more values.

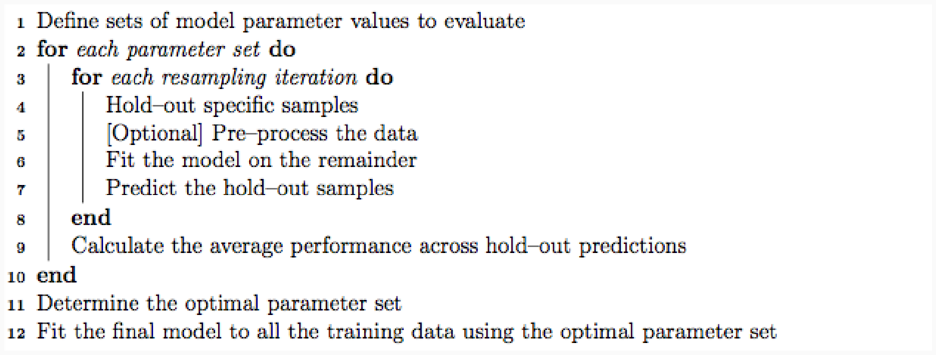
Also, how would you evaluate this model by re-predicting the training set? Those values would be optimistic since one of your neighbors is always you.

Model Tuning

Unsurprisingly, we will evaluate a tuning parameter by fitting a model on one set of data and assessing it with another.

Grid search uses a pre-defined set of candidate tuning parameter values and evaluates their performance so that the best values can be used in the final model.

We'll use resampling to do this. If there are *B* resamples and *C* tuning parameter combinations, we end up fitting $B \times C$ models (but these can be done in parallel).



```
Define sets of model parameter values to evaluate
for each parameter set do
    for each resampling iteration do
        Hold–out specific samples
        [Optional] Pre–process the data
        Fit the model on the remainder
        Predict the hold–out samples
    end
    Calculate the average performance across hold–out predictions
end
Determine the optimal parameter set
Fit the final model to all the training data using the optimal parameter set
```

Fitting a Boosted Tree

Boosting is an approach to ensembling regression trees where a series of trees are created such that each one is related to prior trees

In boosting, a common approach is for the current trees to up-weight samples that were poorly predicted in previous fits. `parsnip` has boosting methods for trees. A common engine used in R is the `xgboost` package.

Boosted trees have very low overhead for pre-processing. However, `xgboost` (unlike other boosting packages) requires the use of dummy variables. We'll create a simpler version of our recipe for this model.

We will start by including 50 trees in the ensemble using the `trees` parameter.

Fitting a Boosted Tree

```r
bst_mod <-
  boost_tree(mode = "regression", trees = 50) %>%
  set_engine("xgboost")

# No need for nonlinear terms, interactions, or "other groups"
bst_rec <-
  recipe(Sale_Price ~ Bldg_Type + Neighborhood + Year_Built +
           Gr_Liv_Area + Full_Bath + Year_Sold + Lot_Area +
           Central_Air + Longitude + Latitude,
         data = ames_train) %>%
  step_log(Sale_Price, base = 10) %>%
  step_BoxCox(Lot_Area, Gr_Liv_Area) %>%
  step_dummy(all_nominal())
```

Helper Functions

The previous `parsnip_fit` function is used for model training

```
parsnip_fit
```

```
## function(rec_obj, model) {
##   fit(model, Sale_Price ~ ., data = juice(rec_obj))
## }
## <bytecode: 0x7fd283ef2d90>
```

For prediction...

An interesting aspect of some models is that you can use the same R object to get predictions on multiple *sub-models*.

In our case, if you create 50 trees in the boosting ensemble, you can get predictions on the same model using ensemble sizes smaller than 50.

Not all models can do this and it depends on how the package is implemented.

Other R packages that enable sub-models includes `glmnet`, `pls`, `C50`, `rpart`, `earth`, and many others.

`parsnip` enables this is a separate function called `multi_predict()`.

Multiple Predictions at Once

```r
ex_fit <-
  bst_mod %>%
  fit(log10(Sale_Price) ~ Longitude + Latitude,
      data = ames_train)

smol_test <-
  ames_test %>%
  select(Longitude, Latitude) %>%
  slice(1:3)

ex_pred <- multi_predict(ex_fit, smol_test, trees = 20:21)

ex_pred
```

```
## # A tibble: 3 x 1
##   .pred
##   <list>
## 1 <tibble [2 × 2]>
## 2 <tibble [2 × 2]>
## 3 <tibble [2 × 2]>
```

Recall that we always return as many predictions as original rows:

```r
ex_pred$.pred[[1]]
```

```
## # A tibble: 2 x 2
##   trees .pred
##   <int> <dbl>
## 1    20  5.16
## 2    21  5.16
```

```r
unnest(ex_pred)
```

```
## # A tibble: 6 x 2
##   trees .pred
##   <int> <dbl>
## 1    20  5.16
## 2    21  5.16
## 3    20  5.15
## 4    21  5.15
## 5    20  5.23
## 6    21  5.23
```

Putting it Together

```r
bst_metrics <- function(split, recipe, fit) {
  holdout_dat <- bake(recipe, new_data = assessment(split))

  pred_vars <- fit$fit$feature_names
  resample_name <- labels(split)$id

  multi_predict(fit,
                new_data = holdout_dat %>%
                  select(one_of(pred_vars)),
                trees = 1:50) %>%
    bind_cols(dplyr::select(holdout_dat, Sale_Price)) %>%
    unnest() %>%
    group_by(trees) %>%
    perf_metrics(Sale_Price, .pred) %>%
    mutate(resample = resample_name)
}
```

```r
bst_splits <-
  cv_splits %>%
  dplyr::select(splits, id, lm_res) %>%
  mutate(
    recipes = map(splits, prepper, recipe = bst_rec),
    boosting = map(recipes, parsnip_fit, model = bst_mod),
    boosting_res = pmap(list(splits, recipes, boosting),
                        bst_metrics)
  )

bst_splits %>% select(-splits, -lm_res)
```

```
## # A tibble: 10 x 4
##    id     recipes  boosting boosting_res
##    <chr>  <list>   <list>   <list>
##  1 Fold01 <recipe> <fit[+]> <tibble [150 × 5]>
##  2 Fold02 <recipe> <fit[+]> <tibble [150 × 5]>
##  3 Fold03 <recipe> <fit[+]> <tibble [150 × 5]>
##  4 Fold04 <recipe> <fit[+]> <tibble [150 × 5]>
##  5 Fold05 <recipe> <fit[+]> <tibble [150 × 5]>
##  6 Fold06 <recipe> <fit[+]> <tibble [150 × 5]>
##  7 Fold07 <recipe> <fit[+]> <tibble [150 × 5]>
##  8 Fold08 <recipe> <fit[+]> <tibble [150 × 5]>
##  9 Fold09 <recipe> <fit[+]> <tibble [150 × 5]>
## 10 Fold10 <recipe> <fit[+]> <tibble [150 × 5]>
```

Resampling Results

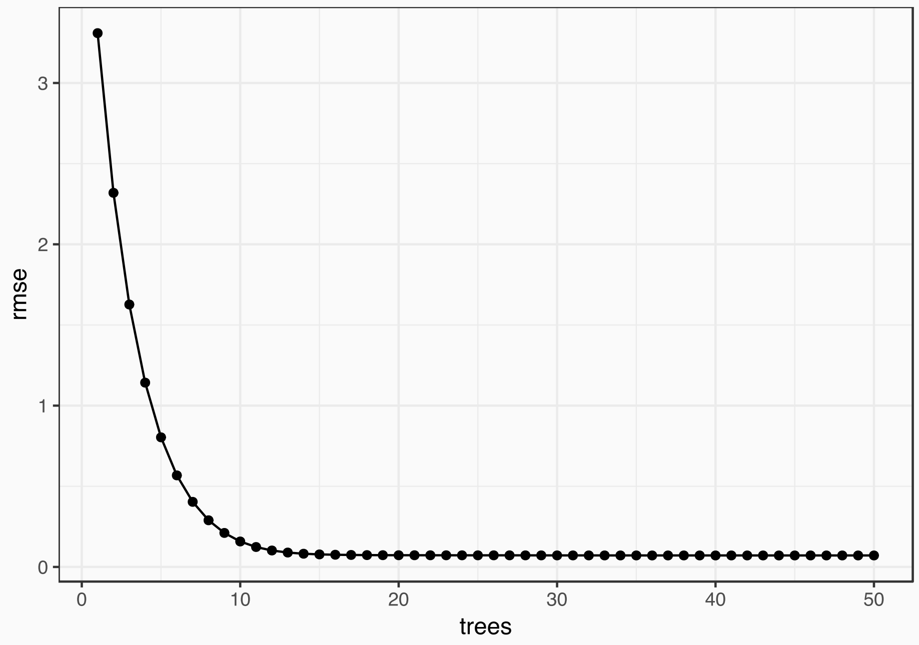
```
rs_rmse <-
  bst_splits %>%
  unnest(boosting_res) %>%
  dplyr::filter(.metric == "rmse") %>%
  group_by(trees) %>%
  dplyr::summarize(rmse = mean(.estimate, na.rm = TRUE))

rs_rmse %>% slice(1:3)
```

```
## # A tibble: 3 x 2
##   trees  rmse
##   <int> <dbl>
## 1     1  3.31
## 2     2  2.32
## 3     3  1.63
```

```
rs_rmse %>% arrange(rmse) %>% slice(1)
```

```
## # A tibble: 1 x 2
##   trees   rmse
##   <int>  <dbl>
## 1    44 0.0713
```

```
ggplot(rs_rmse, aes(x = trees,y = rmse)) +
  geom_point() +
  geom_path()
```



The Final Model

Simpler models are better so I'll finalize the model using `trees = 20`:

```r
# The entire training set is used for the recipe and model
bst_rec_final <- prep(bst_rec)

bst_mod_final <-
  update(bst_mod, trees = 20) %>%
  fit(Sale_Price ~ ., data = juice(bst_rec_final))
```

Note that all of the models in the `bst_mod` column were only used to measure performance and can be deleted.

Workflows and Tuning Packages

Similar to the discussion about `workflow` objects, we have an upcoming package that will make tuning simple.

Within a workflow, you will be able to tag parameters that should be optimized.

- This can also include parameters in the recipe and post-processing operations

A variety of tuning methods such as grid search, random search, and Bayesian optimization will then find optimal values.

The API will be much higher level (similar to what `caret` does).

Other Interesting Packages for Modeling

- For predictors with many (or novel) **categories**, supervised encodings (`embed`) may make the model simpler.
- When working with **tuning parameters**, pre-defined objects can make this easier (`dials`) (experimental)
- When you don't want to make a prediction, **equivocal zone** data structures are available (`probably`).
- If you are making your own modeling package, `hardhat` makes the **behind-the-scene code** simple.
- Feature engineering for **text data** is easy (`tidytext` and `textrecipes`)
- A beutiful API for hypothesis testing (`infer`)