Applied Machine Learning - 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   0.1.6
## ✔ dplyr     0.8.3       ✔ rsample   0.0.5
## ✔ ggplot2   3.2.0       ✔ tibble    2.1.3
## ✔ infer     0.4.0.1     ✔ yardstick 0.0.3
## ✔ parsnip   0.0.3
```

```
## ── 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)

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

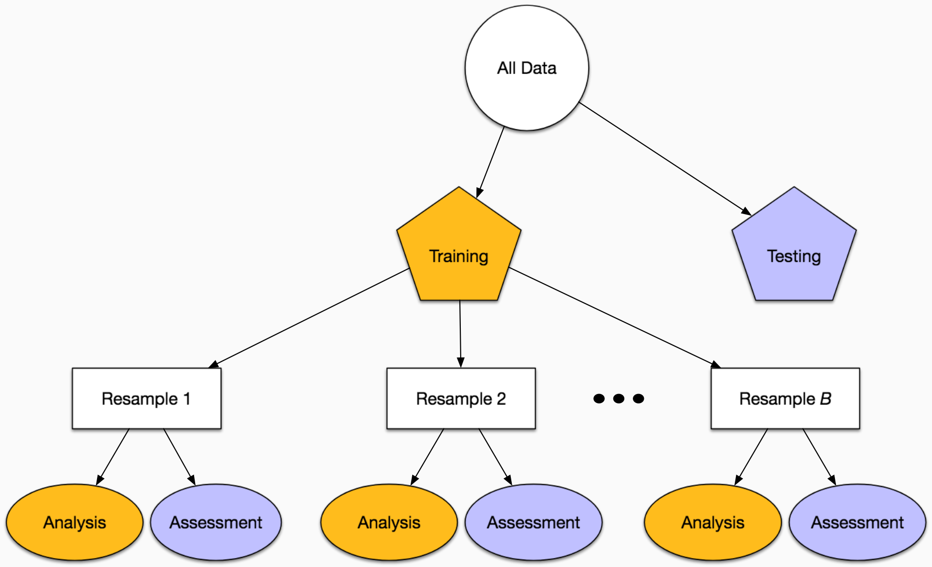
These are additional data splitting schemes that are applied to the *training* set and are used for **estimating model performance**.

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*.

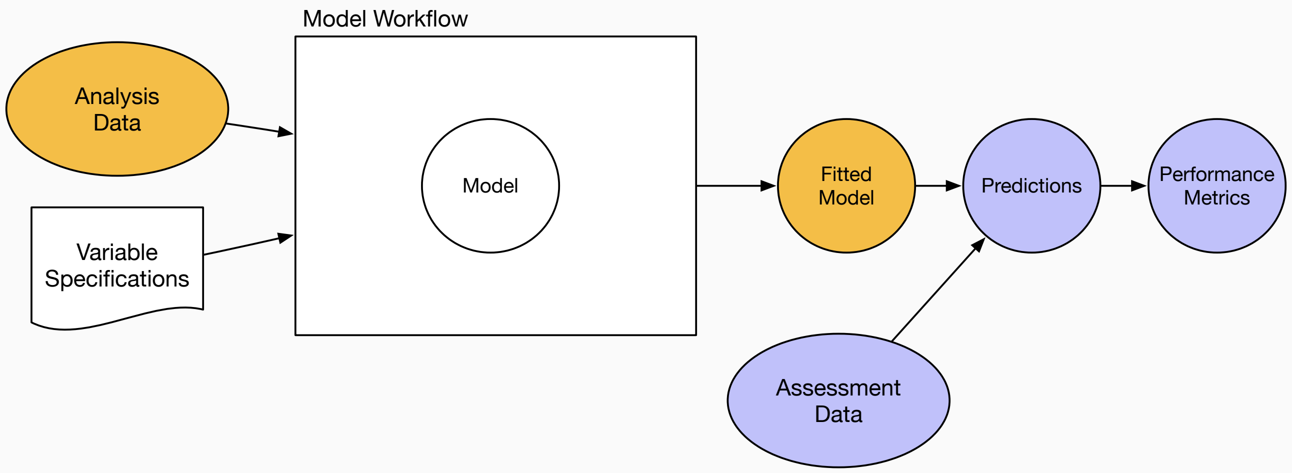
This process is repeated many times.

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:



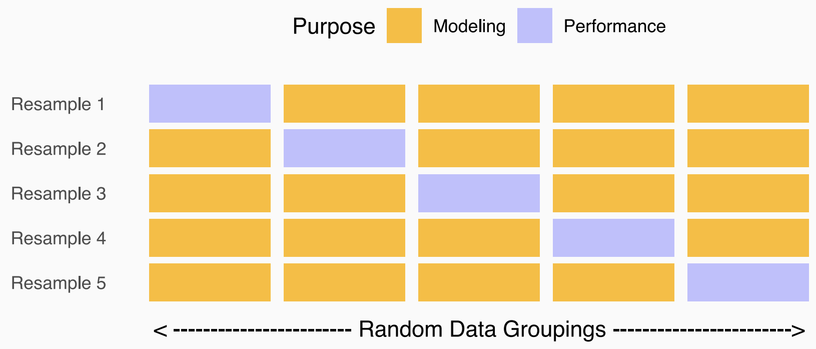
The final resampling estimate is the average of all of the estimated metrics (e.g. RMSE, etc).

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

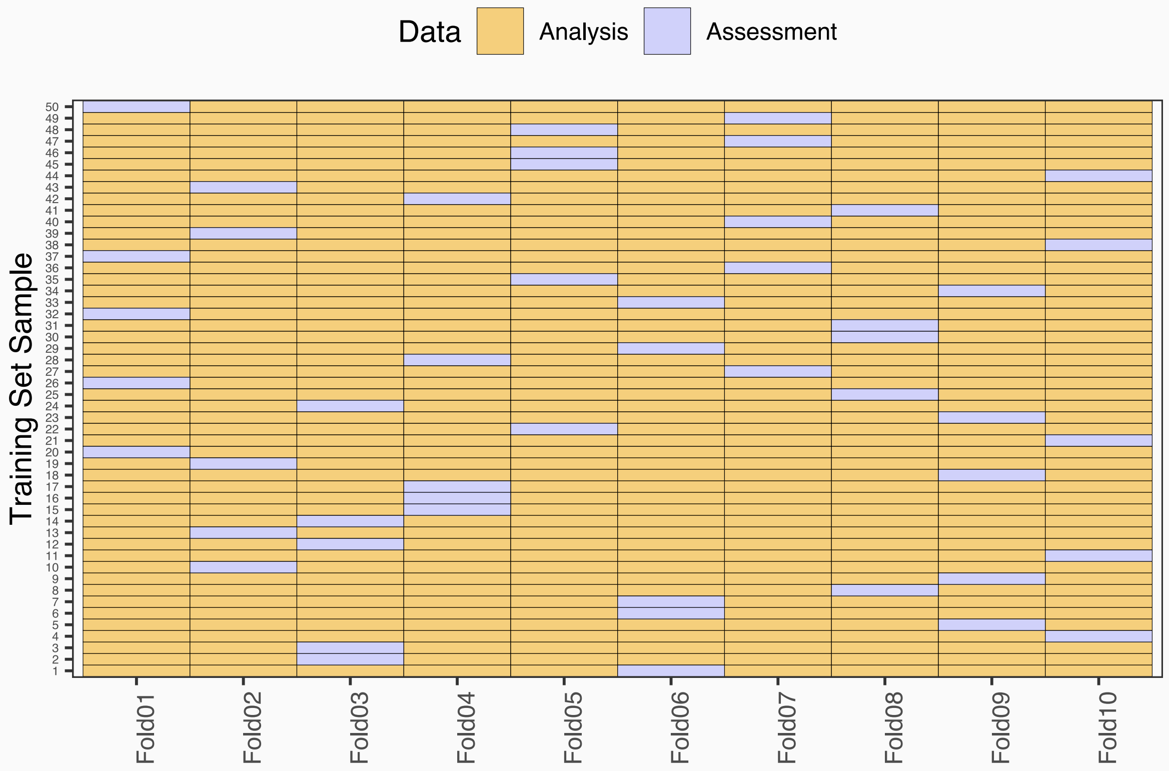
The final performance is based on the hold-out predictions by *averaging* the statistics from the *V* blocks.



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(ames_train) #10-fold is default
cv_splits
```

```
## #  10-fold cross-validation
## # A tibble: 10 x 2
##    splits           id
##    <named list>     <chr>
##  1 <split [2K/220]> Fold01
##  2 <split [2K/220]> Fold02
##  3 <split [2K/220]> Fold03
##  4 <split [2K/220]> Fold04
##  5 <split [2K/220]> Fold05
##  6 <split [2K/220]> Fold06
##  7 <split [2K/220]> Fold07
##  8 <split [2K/220]> Fold08
##  9 <split [2K/220]> Fold09
## 10 <split [2K/219]> Fold10
```

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

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

```
## <1979/220/2199>
```

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

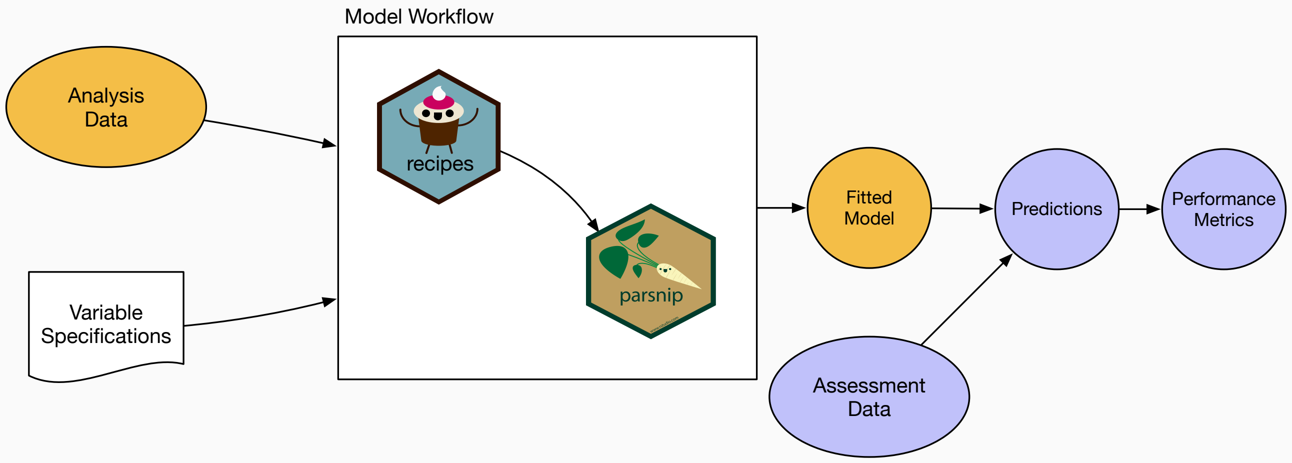
```
## [1] 1979   74
```

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

```
## [1] 220  74
```

Working with Preprocessing and Recipes

We'll use our last recipe along with a K-NN model? Within each analysis set:



This means that, for each resample, there are potentially different preprocessing results (e.g. Box-Cox transformations, etc.) and *this is a good thing*.

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 %>%
  mutate(recipes = map(splits, prepper, recipe = ames_rec, fres
cv_splits
```

```
## #  10-fold cross-validation
## # A tibble: 10 x 3
##    splits          id     recipes
##  * <named list>    <chr>  <named list>
##   1 <split [2K/220]> Fold01 <recipe>
##   2 <split [2K/220]> Fold02 <recipe>
##   3 <split [2K/220]> Fold03 <recipe>
##   4 <split [2K/220]> Fold04 <recipe>
##   5 <split [2K/220]> Fold05 <recipe>
##   6 <split [2K/220]> Fold06 <recipe>
##   7 <split [2K/220]> Fold07 <recipe>
##   8 <split [2K/220]> Fold08 <recipe>
```

As an example of measuring the pre-processing variability:

```
map_dfr(cv_splits$recipes, tidy, number = 2) %>%
  dplyr::filter(terms == "Lot_Area") %>%
  slice(1:6)
```

```
## # A tibble: 6 x 3
##   terms    value id
##   <chr>    <dbl> <chr>
## 1 Lot_Area 0.166 BoxCox_LEmKG
## 2 Lot_Area 0.119 BoxCox_LEmKG
## 3 Lot_Area 0.143 BoxCox_LEmKG
## 4 Lot_Area 0.120 BoxCox_LEmKG
## 5 Lot_Area 0.163 BoxCox_LEmKG
## 6 Lot_Area 0.125 BoxCox_LEmKG
```

Fitting the Models

A simple function is used to fit the model using the results of the recipe.

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.

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

knn_mod <-
  nearest_neighbor(mode = "regression", neighbors = 5) %>%
  set_engine("kknn")

cv_splits <-
  cv_splits %>%
  mutate(knn = map(recipes, parsnip_fit, model = knn_mod))
```

```
cv_splits %>% slice(1:6)
```

```
## #  10-fold cross-validation
## # A tibble: 6 x 4
##   splits             id     recipes  knn
## * <list>             <chr>  <list>   <list>
## 1 <split [2K/220]> Fold01 <recipe> <fit[+]>
## 2 <split [2K/220]> Fold02 <recipe> <fit[+]>
## 3 <split [2K/220]> Fold03 <recipe> <fit[+]>
## 4 <split [2K/220]> Fold04 <recipe> <fit[+]>
## 5 <split [2K/220]> Fold05 <recipe> <fit[+]>
## 6 <split [2K/220]> Fold06 <recipe> <fit[+]>
```

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 K-NN model (for predictions)

```
cv_splits %>%
  select(splits, recipes, knn) %>%
  slice(1:4)
```

```
## #  10-fold cross-validation
## # A tibble: 4 x 3
##   splits            recipes  knn
## * <list>            <list>   <list>
## 1 <split [2K/220]> <recipe> <fit[+]>
## 2 <split [2K/220]> <recipe> <fit[+]>
## 3 <split [2K/220]> <recipe> <fit[+]>
## 4 <split [2K/220]> <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.

```r
cv_splits <- cv_splits %>%
  mutate(
    metrics = pmap(
      list(
        split  = splits,
        recipe = recipes,
        model  = knn
      ),
      parsnip_metrics
    )
  )
```

```r
cv_splits %>%
  select(splits, recipes, knn, metrics) %>%
  slice(1:4)
```

```
## #  10-fold cross-validation
## # A tibble: 4 x 4
##   splits            recipes  knn      metrics
## * <list>            <list>   <list>   <list>
## 1 <split [2K/220]> <recipe> <fit[+]> <tibble [3 × 3]>
## 2 <split [2K/220]> <recipe> <fit[+]> <tibble [3 × 3]>
## 3 <split [2K/220]> <recipe> <fit[+]> <tibble [3 × 3]>
## 4 <split [2K/220]> <recipe> <fit[+]> <tibble [3 × 3]>
```

```r
cv_splits$metrics[[1]]
```

```
## # A tibble: 3 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard      0.0986
## 2 rsq     standard      0.724
## 3 ccc     standard      0.834
```

Predicting the Assessment Set

```
cv_splits %>%
  unnest(metrics) %>%
  group_by(.metric) %>%
  summarise(
    mean = mean(.estimate),
    std_err = sd(.estimate)/sqrt(sum(!is.na(.estimate)))
  )
```

```
## # A tibble: 3 x 3
##   .metric   mean std_err
##   <chr>    <dbl>   <dbl>
## 1 ccc     0.871  0.00780
## 2 rmse    0.0846 0.00258
## 3 rsq     0.777  0.0112
```

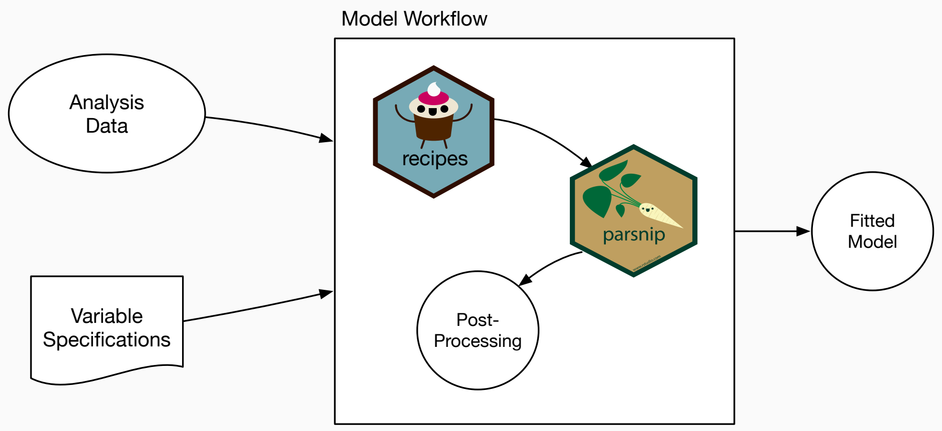
Workflows

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.



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).

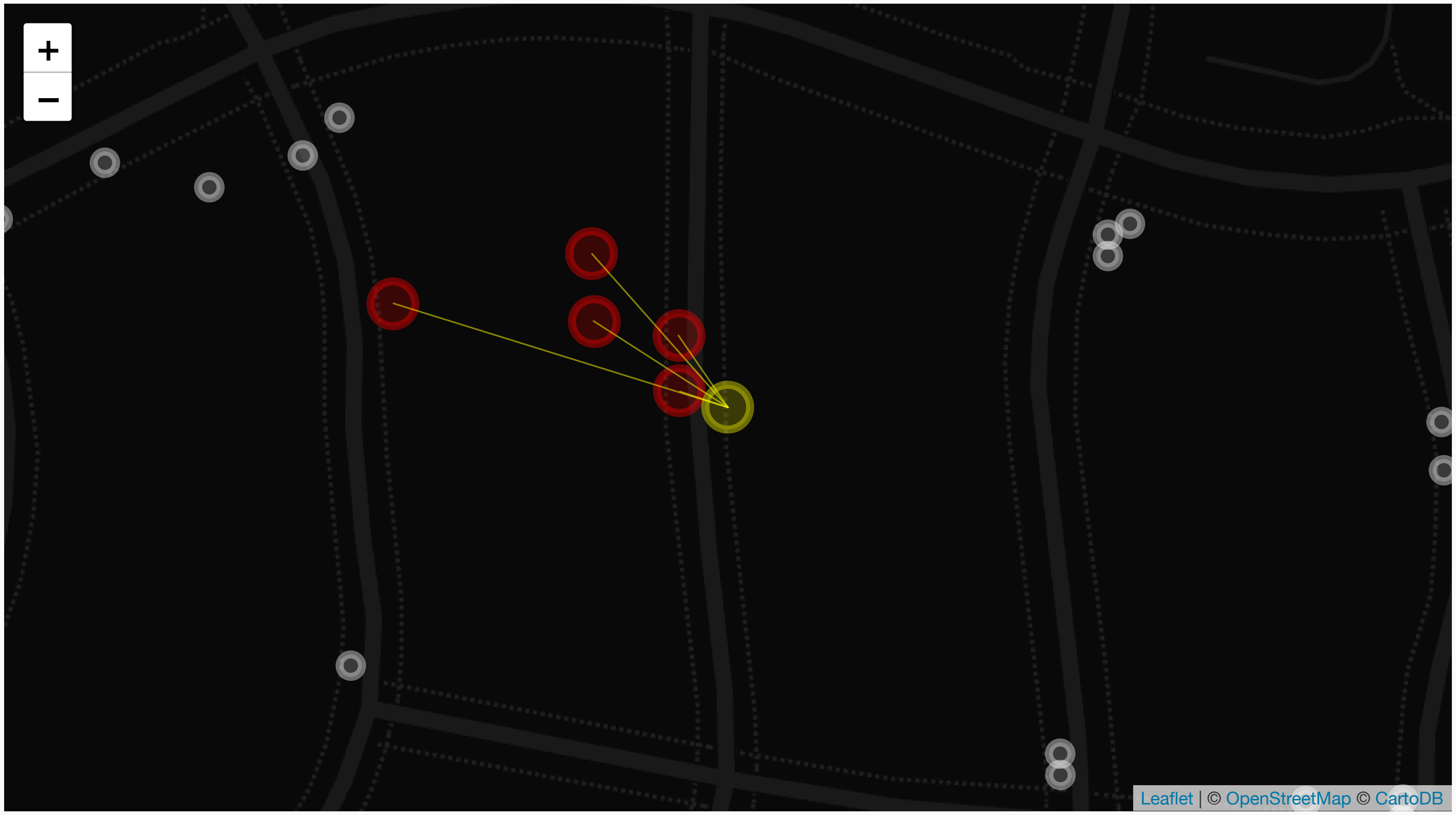
When a new sample is predicted, K training set points are found that are most similar to the new sample being predicted.

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.

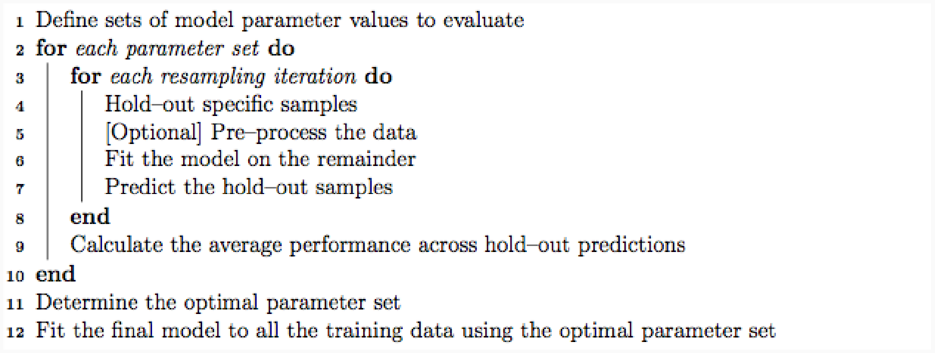
Also, how would you evaluate this model by re-predicting the training set? Those values would be optimistic since one of your neighbors is always you.

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
```

Bad news, good news

- *Bad*: By the end of the year, we should have a nice interface to tuning model and recipe parameters.
- *Good*: For K-NN, there is a simple way of tuning the model.

For some models (e.g. PLS, boosted trees, MARS, glmnet, and others), we can make multiple predictions from the same fitted model.

For example, if you do 100 boosting iterations with trees, you can get predictions for iterations < 100 for nearly free.

We can do this for K-NN using a function called `multi_predict()`.

K-NN Multiple Predictions

```
one_to_five <-
  multi_predict(
    cv_splits$knn[[1]],
    new_data = juice(cv_splits$recipes[[1]]) %>% slice(1:2),
    neighbors = 1:5
  )
one_to_five
```

```
## # A tibble: 2 x 1
##   .pred
##   <list>
## 1 <tibble [5 × 2]>
## 2 <tibble [5 × 2]>
```

```
one_to_five %>%
  add_rowindex() %>%
  unnest()
```

```
## # A tibble: 10 x 3
##     .row neighbors .pred
##    <int>     <int> <dbl>
##  1     1         1  5.24
##  2     1         2  5.23
##  3     1         3  5.22
##  4     1         4  5.20
##  5     1         5  5.19
##  6     2         1  5.39
##  7     2         2  5.37
##  8     2         3  5.36
##  9     2         4  5.34
## 10     2         5  5.33
```

Putting it Together

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

  resample_name <- labels(split)$id

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

```r
cv_splits <-
  cv_splits %>%
  mutate(
    tuning_metrics =
      pmap(list(splits, recipes, knn), more_metrics)
  )

cv_splits %>% select(tuning_metrics)
```

```
## # A tibble: 10 x 1
##    tuning_metrics
##    <named list>
##  1 <tibble [150 × 5]>
##  2 <tibble [150 × 5]>
##  3 <tibble [150 × 5]>
##  4 <tibble [150 × 5]>
##  5 <tibble [150 × 5]>
##  6 <tibble [150 × 5]>
##  7 <tibble [150 × 5]>
##  8 <tibble [150 × 5]>
##  9 <tibble [150 × 5]>
## 10 <tibble [150 × 5]>
```

Resampling Results

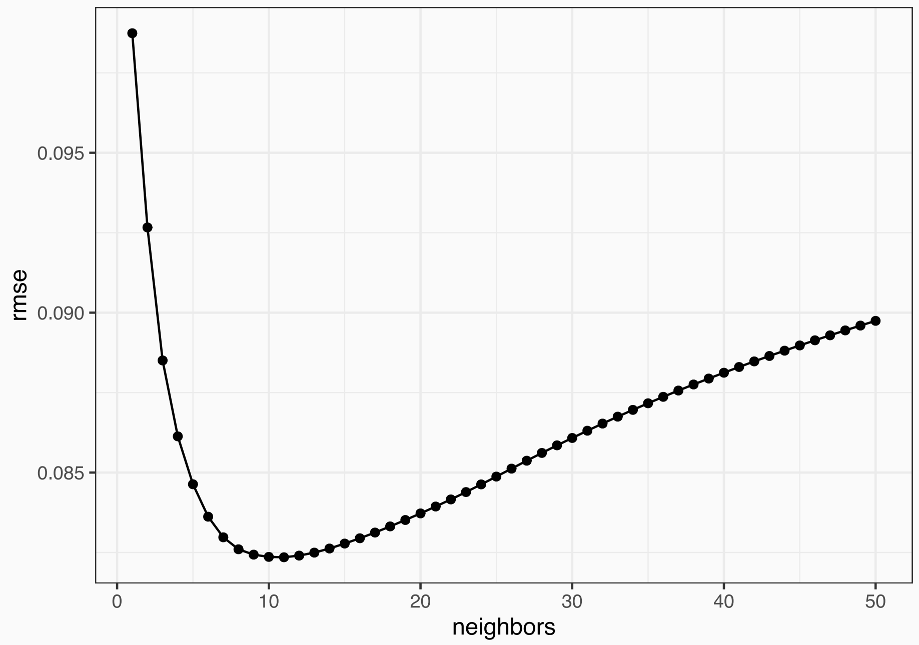
```r
rs_rmse <-
  cv_splits %>%
  unnest(tuning_metrics) %>%
  dplyr::filter(.metric == "rmse") %>%
  group_by(neighbors) %>%
  dplyr::summarize(rmse = mean(.estimate, na.rm = TRUE))

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

```
## # A tibble: 3 x 2
##   neighbors   rmse
##       <int>  <dbl>
## 1         1 0.0987
## 2         2 0.0927
## 3         3 0.0885
```

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

```
## # A tibble: 1 x 2
##   neighbors   rmse
##       <int>  <dbl>
## 1        11 0.0824
```

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



The Final Model

Simpler models are better so I'll finalize the model using `neighbors = 11`:

```
# The entire training set is used for the recipe and model
ames_rec_final <- prep(ames_rec)

knn_mod_final <-
  update(knn_mod, neighbors = 11) %>%
  fit(Sale_Price ~ ., data = juice(ames_rec_final))
```

Note that all of the models in the `knn` 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`)