Applied Machine Learning - Resampling and Grid Search

Max Kuhn (RStudio)

Loading

```r
library(tidymodels)
```

```
## Registered S3 method overwritten by 'xts':
##   method     from
##   as.zoo.xts zoo
```

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

```
## ✔ broom     0.5.1          ✔ purrr     0.3.3
## ✔ dials     0.0.3.9002     ✔ recipes   0.1.7.9001
## ✔ dplyr     0.8.3          ✔ rsample   0.0.5
## ✔ ggplot2   3.2.1          ✔ tibble    2.1.3
## ✔ infer     0.4.0          ✔ yardstick 0.0.4
## ✔ parsnip   0.0.4
```

```
## ── Conflicts ──────────────────────────────────── tidymodels_conflicts() ──
## ✖ purrr::discard()  masks scales::discard()
## ✖ dplyr::filter()   masks stats::filter()
## ✖ dplyr::lag()      masks stats::lag()
## ✖ ggplot2::margin() masks dials::margin()
## ✖ dials::offset()   masks stats::offset()
```

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, deg_free = 5)
```

Resampling

Resampling Methods

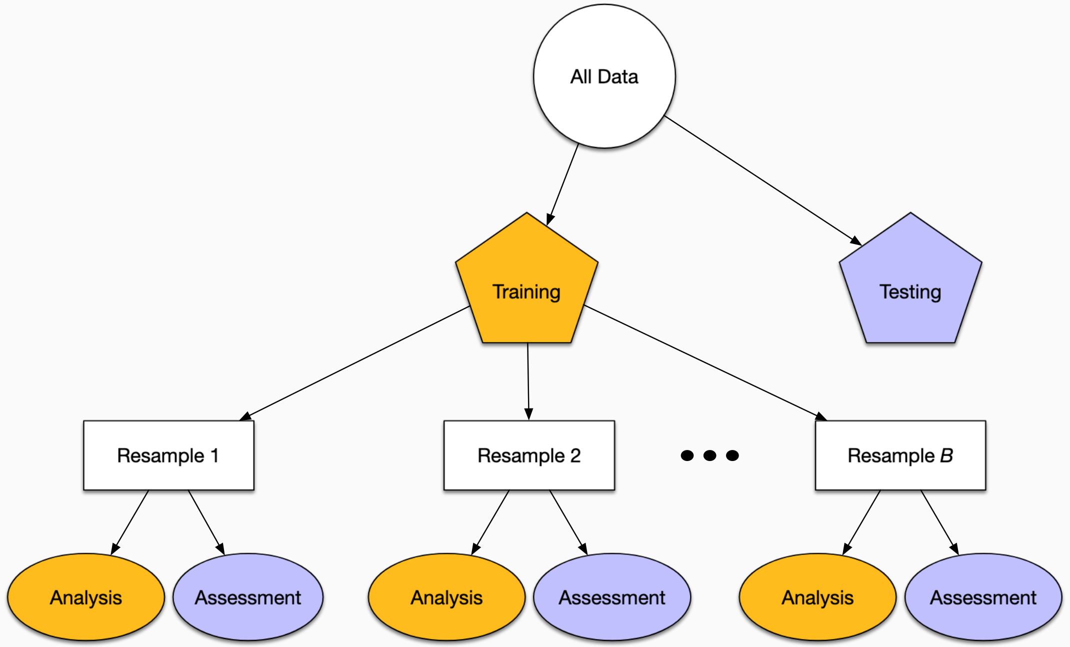
These are additional data splitting schemes that are applied to the *training* set and are used for **estimating model performance**.

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*.

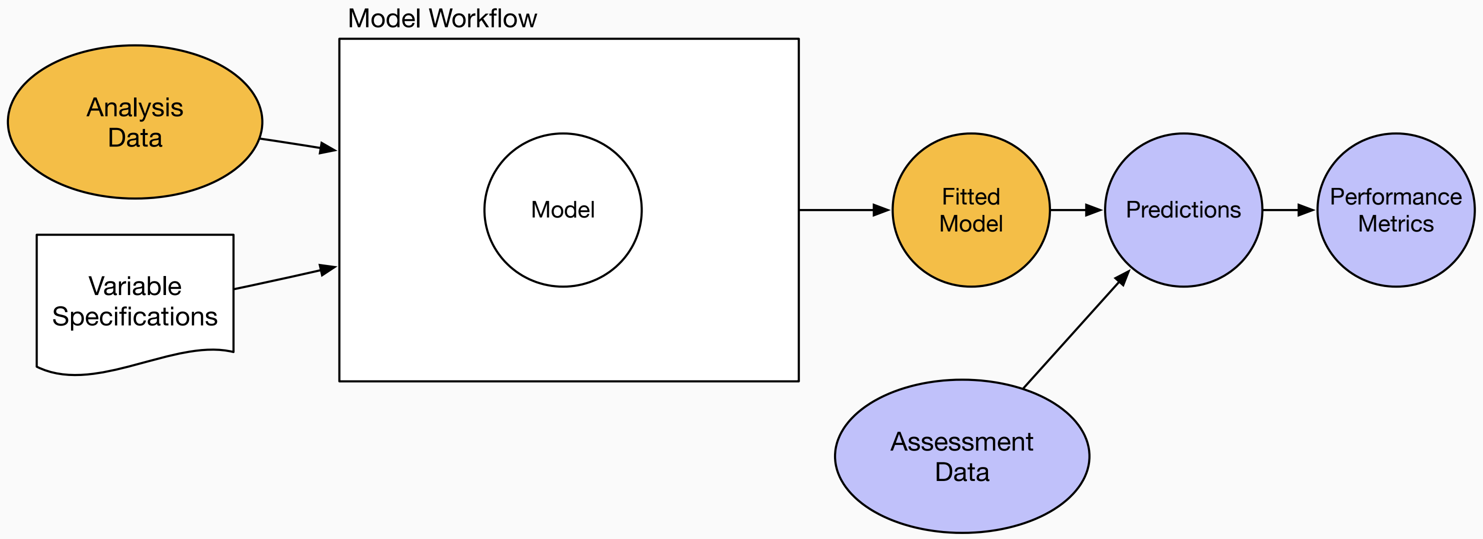
This process is repeated many times.

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:



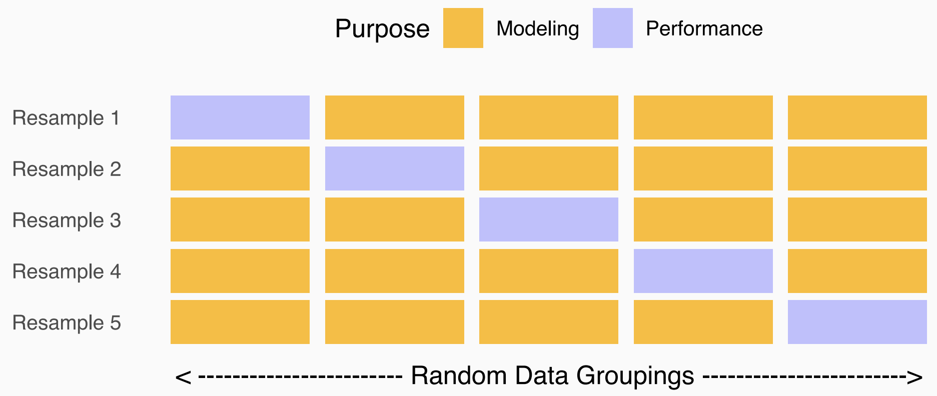
The final resampling estimate is the average of all of the estimated metrics (e.g. RMSE, etc).

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

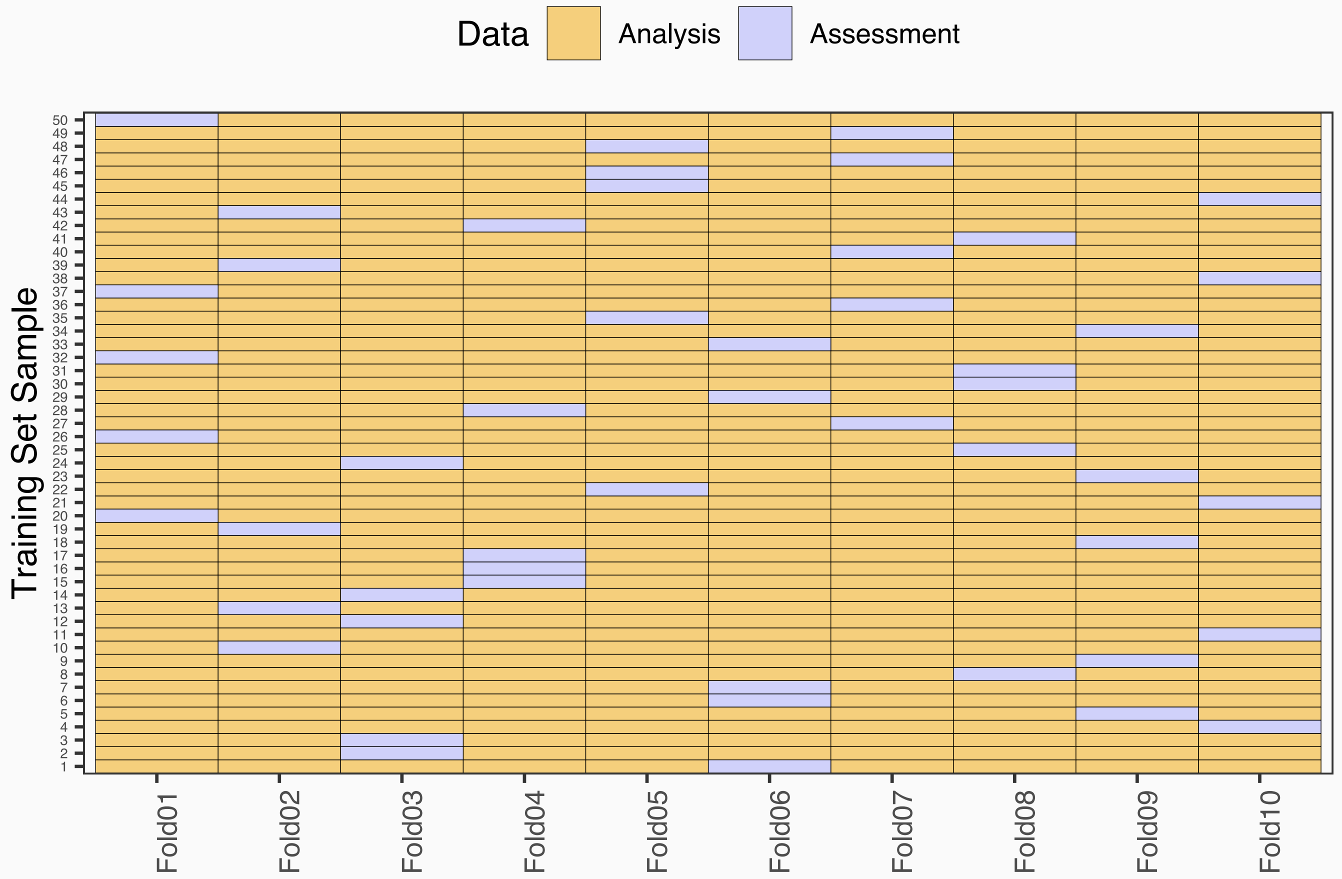
The final performance is based on the hold-out predictions by *averaging* the statistics from the *V* blocks.



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`

```
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.

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

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

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

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

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

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

K-Nearest Neighbors Model

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

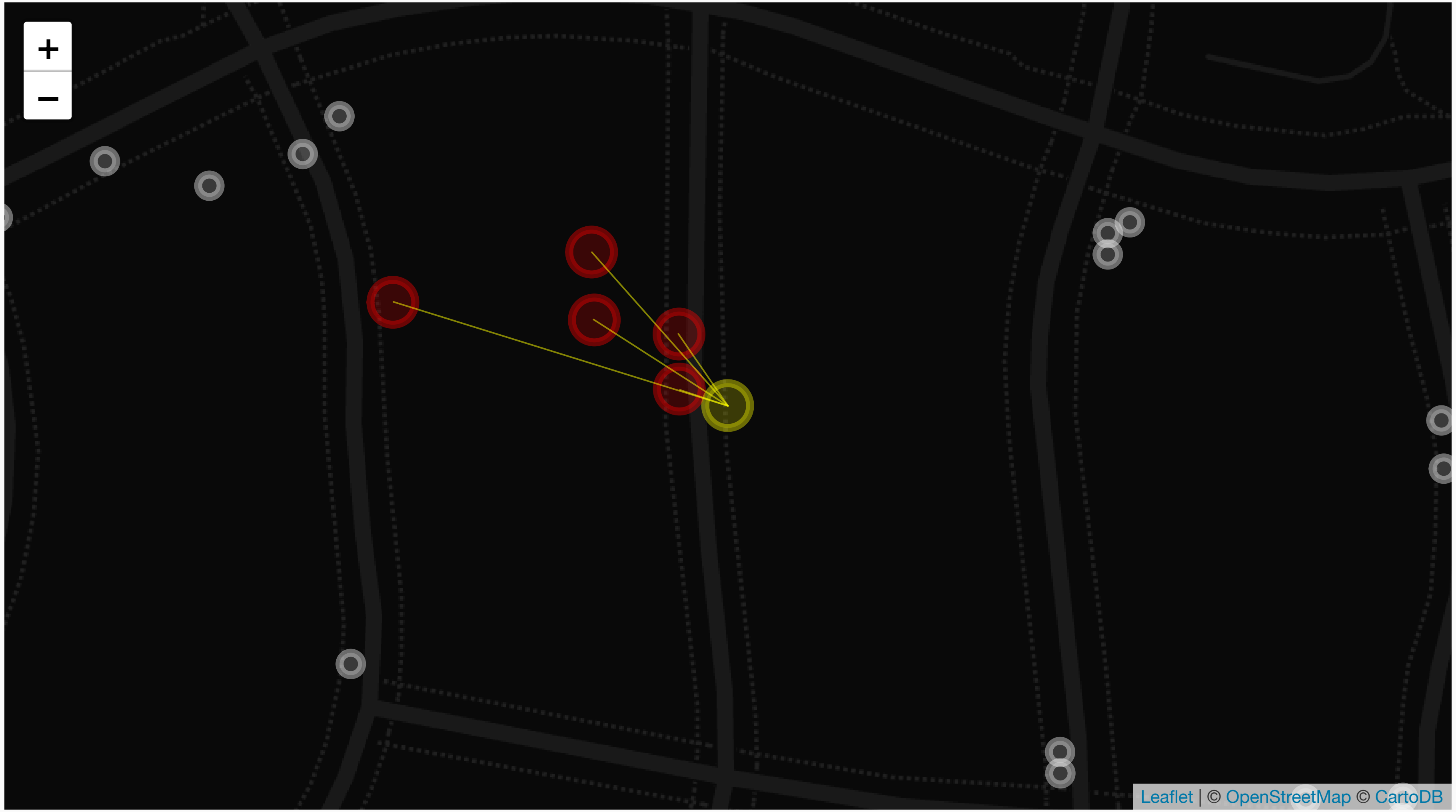
When a new sample is predicted, *K* training set points are found that are most similar to the new sample being predicted.

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

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

Let's try a 5-neighbor model on the Ames data.

5-Nearest Neighbors Model



Resampling a 5-NN model

```
five_nn <-
  nearest_neighbor(neighbors = 5) %>%
  set_engine("kknn") %>%
  set_mode("regression")
```

If we were using this model with the training set:

```
five_nn %>% fit(log10(Sale_Price) ~ Longitude + Latitude, data = ames_train)
```

```
## parsnip model object
##
## Fit in:  31ms
## Call:
## kknn::train.kknn(formula = formula, data = data, ks = ~5)
##
## Type of response variable: continuous
## minimal mean absolute error: 0.06753097
## Minimal mean squared error: 0.009633708
## Best kernel: optimal
## Best k: 5
```

Resampling a 5-NN model

Let's repeat that but using each of the 10 analysis sets:

```
cv_splits_knn <-
  cv_splits %>%
  mutate(
    fits =
      map(
        splits,
        ~ five_nn %>% fit(log10(Sale_Price) ~ Longitude + Latitude, data = analysis(.x))
      )
  )
cv_splits_knn %>% slice(1:3)
```

```
## #  10-fold cross-validation
## # A tibble: 3 x 3
##   splits           id     fits
## * <list>           <chr>  <list>
## 1 <split [2K/220]> Fold01 <fit[+]>
## 2 <split [2K/220]> Fold02 <fit[+]>
## 3 <split [2K/220]> Fold03 <fit[+]>
```

Predictions for each model

```r
knn_pred <-
  map2_dfr(cv_splits_knn$fits, cv_splits_knn$splits,
           ~ predict(.x, assessment(.y)),
           .id = "fold")
```

```
## # A tibble: 2,199 x 2
##    fold  .pred
##    <chr> <dbl>
##  1 1      5.36
##  2 1      5.28
##  3 1      5.27
##  4 1      5.06
##  5 1      5.37
##  6 1      5.30
##  7 1      5.14
##  8 1      5.08
##  9 1      5.10
## 10 1      5.24
## # … with 2,189 more rows
```

Predictions for each model

```r
knn_pred <-
  map2_dfr(cv_splits_knn$fits, cv_splits_knn$splits,
           ~ predict(.x, assessment(.y)),
           .id = "fold")

prices <-
  map_dfr(cv_splits_knn$splits,
          ~ assessment(.x) %>% select(Sale_Price))
```

```
## # A tibble: 2,199 x 1
##    Sale_Price
##         <int>
##  1     216000
##  2     160000
##  3     216500
##  4     133000
##  5     254000
##  6     218500
##  7     108000
##  8     105000
##  9     132000
## 10     155000
## # … with 2,189 more rows
```

Predictions for each model

```r
knn_pred <-
  map2_dfr(cv_splits_knn$fits, cv_splits_knn$splits,
           ~ predict(.x, assessment(.y)),
           .id = "fold")

prices <-
  map_dfr(cv_splits_knn$splits,
          ~ assessment(.x) %>% select(Sale_Price)) %>%
  mutate(Sale_Price = log10(Sale_Price))
```

```
## # A tibble: 2,199 x 1
##    Sale_Price
##         <dbl>
##  1       5.33
##  2       5.20
##  3       5.34
##  4       5.12
##  5       5.40
##  6       5.34
##  7       5.03
##  8       5.02
##  9       5.12
## 10       5.19
## # … with 2,189 more rows
```

Compute RMSE for each model

```r
knn_pred <-
  map2_dfr(cv_splits_knn$fits, cv_splits_knn$splits,
           ~ predict(.x, assessment(.y)),
           .id = "fold")

prices <-
  map_dfr(cv_splits_knn$splits,
          ~ assessment(.x) %>% select(Sale_Price)) %>%
  mutate(Sale_Price = log10(Sale_Price))

rmse_estimates <-
  knn_pred %>%
  bind_cols(prices)
```

```
## # A tibble: 2,199 x 3
##    fold  .pred Sale_Price
##    <chr> <dbl>      <dbl>
##  1 1      5.36       5.33
##  2 1      5.28       5.20
##  3 1      5.27       5.34
##  4 1      5.06       5.12
##  5 1      5.37       5.40
##  6 1      5.30       5.34
##  7 1      5.14       5.03
##  8 1      5.08       5.02
##  9 1      5.10       5.12
## 10 1      5.24       5.19
## # … with 2,189 more rows
```

Compute RMSE for each model

```r
knn_pred <-
  map2_dfr(cv_splits_knn$fits, cv_splits_knn$splits,
           ~ predict(.x, assessment(.y)),
           .id = "fold")

prices <-
  map_dfr(cv_splits_knn$splits,
          ~ assessment(.x) %>% select(Sale_Price)) %>%
  mutate(Sale_Price = log10(Sale_Price))

rmse_estimates <-
  knn_pred %>%
  bind_cols(prices) %>%
  group_by(fold)
```

```
## # A tibble: 2,199 x 3
## # Groups:   fold [10]
##    fold  .pred Sale_Price
##    <chr> <dbl>      <dbl>
##  1 1      5.36       5.33
##  2 1      5.28       5.20
##  3 1      5.27       5.34
##  4 1      5.06       5.12
##  5 1      5.37       5.40
##  6 1      5.30       5.34
##  7 1      5.14       5.03
##  8 1      5.08       5.02
##  9 1      5.10       5.12
## 10 1      5.24       5.19
## # … with 2,189 more rows
```

Compute RMSE for each model

```r
knn_pred <-
  map2_dfr(cv_splits_knn$fits, cv_splits_knn$splits,
           ~ predict(.x, assessment(.y)),
           .id = "fold")

prices <-
  map_dfr(cv_splits_knn$splits,
          ~ assessment(.x) %>% select(Sale_Price)) %>%
  mutate(Sale_Price = log10(Sale_Price))

rmse_estimates <-
  knn_pred %>%
  bind_cols(prices) %>%
  group_by(fold) %>%
  do(rmse = rmse(., Sale_Price, .pred))
```

```
## Source: local data frame [10 x 2]
## Groups: <by row>
##
## # A tibble: 10 x 2
##    fold  rmse
##  * <chr> <list>
##  1 1     <tibble [1 × 3]>
##  2 10    <tibble [1 × 3]>
##  3 2     <tibble [1 × 3]>
##  4 3     <tibble [1 × 3]>
##  5 4     <tibble [1 × 3]>
##  6 5     <tibble [1 × 3]>
##  7 6     <tibble [1 × 3]>
##  8 7     <tibble [1 × 3]>
##  9 8     <tibble [1 × 3]>
## 10 9     <tibble [1 × 3]>
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard       0.113
```

Compute RMSE for each model

```r
knn_pred <-
  map2_dfr(cv_splits_knn$fits, cv_splits_knn$splits,
           ~ predict(.x, assessment(.y)),
           .id = "fold")

prices <-
  map_dfr(cv_splits_knn$splits,
          ~ assessment(.x) %>% select(Sale_Price)) %>%
  mutate(Sale_Price = log10(Sale_Price))

rmse_estimates <-
  knn_pred %>%
  bind_cols(prices) %>%
  group_by(fold) %>%
  do(rmse = rmse(., Sale_Price, .pred)) %>%
  unnest(cols = c(rmse))
```

```
## # A tibble: 10 x 4
##    fold  .metric .estimator .estimate
##    <chr> <chr>   <chr>          <dbl>
##  1 1     rmse    standard      0.113
##  2 10    rmse    standard      0.0874
##  3 2     rmse    standard      0.0895
##  4 3     rmse    standard      0.102
##  5 4     rmse    standard      0.0888
##  6 5     rmse    standard      0.0911
##  7 6     rmse    standard      0.101
##  8 7     rmse    standard      0.103
##  9 8     rmse    standard      0.0971
## 10 9     rmse    standard      0.113
```

Compute Overall RMSE estimate

```r
knn_pred <-
  map2_dfr(cv_splits_knn$fits, cv_splits_knn$splits,
           ~ predict(.x, assessment(.y)),
           .id = "fold")

prices <-
  map_dfr(cv_splits_knn$splits,
          ~ assessment(.x) %>% select(Sale_Price)) %>%
  mutate(Sale_Price = log10(Sale_Price))

rmse_estimates <-
  knn_pred %>%
  bind_cols(prices) %>%
  group_by(fold) %>%
  do(rmse = rmse(., Sale_Price, .pred)) %>%
  unnest(cols = c(rmse))

mean(rmse_estimates$.estimate)
```

```
## [1] 0.09851353
```

How you probably feel right now

Resampling Notes

This is actually a *simple case*:

- There is no attached recipe to be prepared for each resample
- It assumes that 5 neighbors is optimal.

Common Questions:

- Q: What happens to these 10 models? Is this some ensemble thing?
    - A: They are only used for measuring performance so `/dev/null`
- Q: What were we measuring again?
    - A: The resampling estimate is how we think a 5-NN model fit on the *entire data set* would preform.
- Q: Do you actually expect us to do anything like this again?
    - A: No.

Please don't be mad about the next few slides

Easy resampling using the {tune} package

There is a `fit_resamples()` function in the `tune` package that does all of this for you.

```r
library(tune)
easy_eval <-
  fit_resamples(
    log10(Sale_Price) ~ Longitude + Latitude,
    five_nn,
    resamples = cv_splits,
    control = control_resamples(save_pred = TRUE)
  )
```

This does the same process that we did manually except the resamples models are not saved (but you could save them).

```r
easy_eval
```

```
## #  10-fold cross-validation
## # A tibble: 10 x 5
##    splits             id     .metrics          .notes            .pre
##  * <list>             <chr>  <list>            <list>            <lis
##  1 <split [2K/220]> Fold01 <tibble [2 × 3]> <tibble [0 × 1]> <tib
##  2 <split [2K/220]> Fold02 <tibble [2 × 3]> <tibble [0 × 1]> <tib
##  3 <split [2K/220]> Fold03 <tibble [2 × 3]> <tibble [0 × 1]> <tib
##  4 <split [2K/220]> Fold04 <tibble [2 × 3]> <tibble [0 × 1]> <tib
##  5 <split [2K/220]> Fold05 <tibble [2 × 3]> <tibble [0 × 1]> <tib
##  6 <split [2K/220]> Fold06 <tibble [2 × 3]> <tibble [0 × 1]> <tib
##  7 <split [2K/220]> Fold07 <tibble [2 × 3]> <tibble [0 × 1]> <tib
##  8 <split [2K/220]> Fold08 <tibble [2 × 3]> <tibble [0 × 1]> <tib
##  9 <split [2K/220]> Fold09 <tibble [2 × 3]> <tibble [0 × 1]> <tib
## 10 <split [2K/219]> Fold10 <tibble [2 × 3]> <tibble [0 × 1]> <tib
```

Getting the statistics and predictions

```r
collect_predictions(easy_eval) %>%
  arrange(.row) %>%
  slice(1:5)
```

```
## # A tibble: 5 x 4
##   id     .pred  .row Sale_Price
##   <chr>  <dbl> <int>      <dbl>
## 1 Fold10  5.22     1       5.24
## 2 Fold10  5.34     2       5.39
## 3 Fold10  5.28     3       5.28
## 4 Fold09  5.28     4       5.29
## 5 Fold04  5.43     5       5.33
```

```r
collect_metrics(easy_eval)
```

```
## # A tibble: 2 x 5
##   .metric .estimator   mean     n std_err
##   <chr>   <chr>       <dbl> <int>   <dbl>
## 1 rmse    standard   0.0985    10 0.00298
## 2 rsq     standard   0.698     10 0.0153
```

```r
collect_metrics(easy_eval, summarize = FALSE) %>%
  slice(1:10)
```

```
## # A tibble: 10 x 4
##    id     .metric .estimator .estimate
##    <chr>  <chr>   <chr>          <dbl>
##  1 Fold01 rmse    standard      0.113
##  2 Fold01 rsq     standard      0.640
##  3 Fold02 rmse    standard      0.0895
##  4 Fold02 rsq     standard      0.724
##  5 Fold03 rmse    standard      0.102
##  6 Fold03 rsq     standard      0.713
##  7 Fold04 rmse    standard      0.0888
##  8 Fold04 rsq     standard      0.767
##  9 Fold05 rmse    standard      0.0911
## 10 Fold05 rsq     standard      0.717
```

Note that the outcome data are already logged.

Model Tuning

Tuning Parameters

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

For example:

- The number of neighbors in a K-NN models.
- The depth of a classification tree.
- The link function in a generalized linear model (e.g. logit, probit, etc).
- The covariance structure in a linear mixed 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, $K = 1$ neighbors is much more likely to overfit the data than larger values since they average more values.

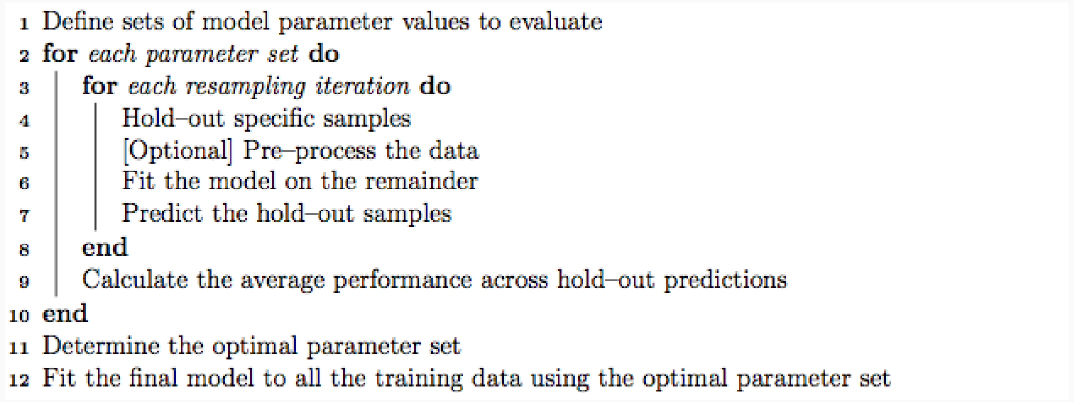
Also, how would you evaluate this model by re-predicting the training set? Those values would be optimistic since one of your neighbors is always you.

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
```

Better API's using the tune package

The `tune` package has more general functions for tuning models. There are two main strategies used:

- Grid search (as shown above) where all of the candidate models are known at the start. We pick the best of these.

- Iterative search where each iterations finds novel tuning parameter values to evaluate.

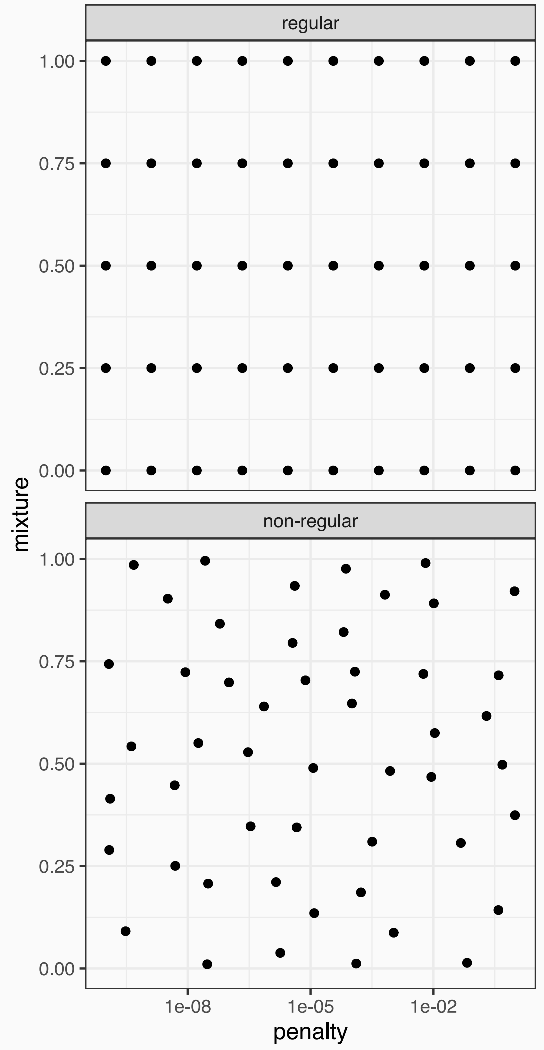
Both have their advantages and disadvantages. At first, we will focus on grid search.

Elements Required for Grid Search

- A set of candidate values to evaluate.
- Measure of model performance.
- A resampling scheme to reliably estimate model performance

We've discussed the last two. Now let's focus on the grid.

There are two main types of grids: regular and non-regular.



About Regular Grids

Usually combinatorial representation of vectors of tuning parameter values. Note that:

- The number of values don't have to be the same per parameter.
- The values can be regular on a transformed scale (e.g. log-10 for penalty).
- Quantitative and qualitative parameters can be combined.
- As the number of parameters increase, the curse of dimensionality kicks in.
- *Thought* to be really inefficient but not in all cases (see the sub-model trick and `multi_predict()`).
- Bad when performance plateaus over a range of one or more parameters.

Making Regular Grids

You can use `expand.grid()` to make a data frame where parameters are in columns and candidate values are in rows.

`dials` has functions to create parameters and parameter sets. For example:

```
penalty()
```

```
## Amount of Regularization  (quantitative)
## Transformer:  log-10
## Range (transformed scale): [-10, 0]
```

```
mixture()
```

```
## Proportion of lasso Penalty  (quantitative)
## Range: [0, 1]
```

```
glmn_grid <-
  grid_regular(glmn_param, levels = c(10, 5))
glmn_grid %>% slice(1:4)
```

```
## # A tibble: 4 x 2
##          penalty mixture
## *          <dbl>   <dbl>
## 1 0.0000000001       0
## 2 0.00000000129      0
## 3 0.0000000167       0
## 4 0.000000215        0
```

Note that the grid for `penalty` is created in the log-10 space but the values in the data frame are in the original units.

Non-Regular Grids

There are two main methods that we have to make these:

- Random grids uniformly sample the parameter space (that might already be on a different scale).

- Space-filling designs (SFD) are based on statistical experimental design principles and try to keep candidate values away from one another while encompassing the entire parameter space.

There's no real downside to using the space-filling designs, so we will focus on these.

The code is easy when using a parameter set:

```r
set.seed(7454)
glmn_sfd <- grid_max_entropy(glmn_param, size = 50)
glmn_sfd %>% slice(1:4)
```

```
## # A tibble: 4 x 2
##      penalty mixture
##        <dbl>   <dbl>
## 1 0.00000362   0.172
## 2 0.0102       0.414
## 3 0.00000372   0.346
## 4 0.000709     0.544
```

```r
# grid_latin_hypercube() can also be used

# grid_random() too
```

Modifying Parameter Sets

```r
# The names can be changed:
glmn_set <- parameters(lambda = penalty(), mixture())

# The ranges can also be set by their name:
glmn_set <-
  update(glmn_set, lambda = penalty(c(-5, -1)))
```

```r
# Some parameters depend on data dimensions:
mtry()
```

```
## # Randomly Selected Predictors  (quantitative)
## Range: [1, ?]
```

```r
rf_set <- parameters(mtry(), trees())
```

```r
rf_set
```

```
## Collection of 2 parameters for tuning
##
##     id parameter type object class
##   mtry           mtry    nparam[?]
##  trees          trees    nparam[+]
##
## Parameters needing finalization:
##    # Randomly Selected Predictors ('mtry')
##
## See `?dials::finalize` or `?dials::update.parameters` for m
```

```r
# Sets the range of mtry to be the number of predictors
finalize(rf_set, mtcars %>% select(-mpg))
```

```
## Collection of 2 parameters for tuning
##
##     id parameter type object class
##   mtry           mtry    nparam[+]
##  trees          trees    nparam[+]
```

Hands-On: K-NN Grids

Looking at the help file `?nearest_neighbors` and find the names of the three tuning parameters.

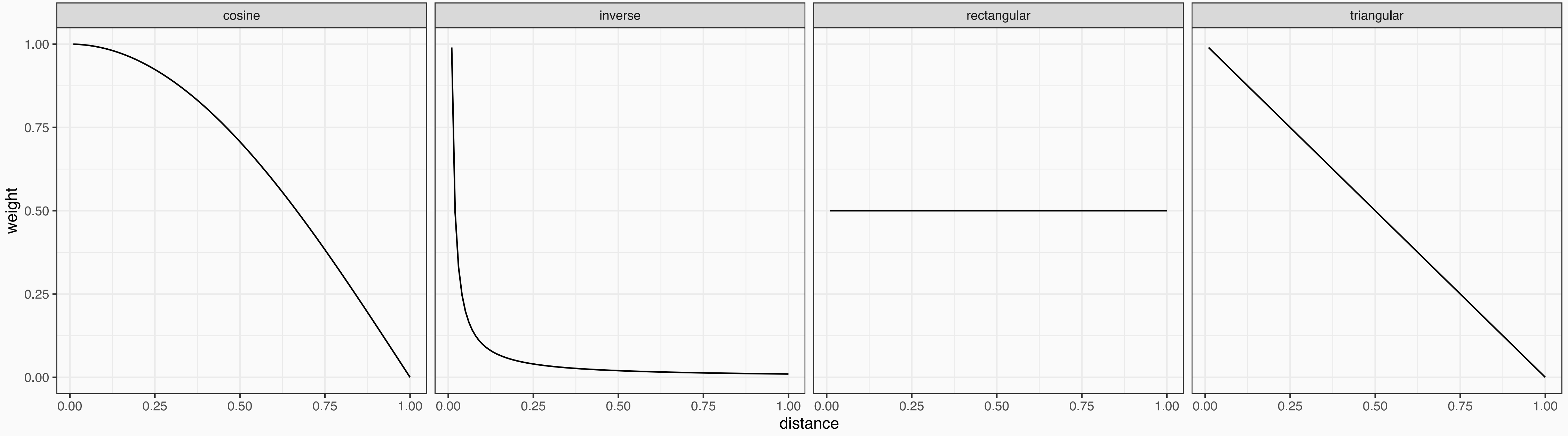
Create a parameter set for these three, make at least one grid, and plot them.

10:00

K-NN Tuning parameters

The `dist_power` parameter is related to the type of distance calculation. `dist_power = 2` is everyday Euclidean distance, `dist_power = 1` is Manhattan distance, and so on. Fractional values are acceptable.

`weight_func` determines how much influence neighbors have based on their distance:



Tagging Tuning parameters

To tune the model, the first step is to *tag* the parameters that will be optimized. The `tune` package has a function called `tune()` that can be used to do this:

```r
library(tune)
knn_mod <-
  nearest_neighbor(neighbors = tune(), weight_func = tune()) %>%
  set_engine("kknn") %>%
  set_mode("regression")
```

`parameters` can detect these arguments:

```r
parameters(knn_mod)
```

```
## Collection of 2 parameters for tuning
##
##           id parameter type object class
##    neighbors      neighbors    nparam[+]
##  weight_func    weight_func    dparam[+]
```

Tagging Tuning parameters

The `tune()` function has an optimal argument called `id` that can be used to name the parameters:

```
nearest_neighbor(neighbors = tune("K"), weight_func = tune("weights")) %>%
  set_engine("kknn") %>%
  set_mode("regression") %>%
  parameters()
```

```
## Collection of 2 parameters for tuning
##
##       id parameter type object class
##        K      neighbors    nparam[+]
##  weights    weight_func    dparam[+]
```

This mainly comes in handy when the same parameter type shows up in two different places (an example is shown later).

Recipe objects can also have `tune()` in their arguments.

Grid Search

Let's tune the model over a regular grid and optimize `neighbors` and `weight_func`. First we make the grid, then use the `tune_grid()` function:

```
set.seed(522)
knn_reg <- knn_mod %>% parameters() %>% grid_regular(levels = c(15, 5))
ctrl <- control_grid(verbose = TRUE)

knn_reg_search <-
  tune_grid(ames_rec, model = knn_mod, resamples = cv_splits, grid = knn_reg, control = ctrl)
```

We are optimizing 75 models over 10 resamples. The output looks like:

```
✓ Fold04: recipe
❯ Fold04: model 1/5
✓ Fold04: model 1/5
! Fold04: model 1/5 (predictions): some 'x' values beyond boundary knots may cause ill-conditioned bases
❯ Fold04: model 2/5
✓ Fold04: model 2/5
! Fold04: model 2/5 (predictions): some 'x' values beyond boundary knots may cause ill-conditioned bases
```

Some Notes

- If we are evaluating 75 models, why does it say `model 1/5`?

- The recipe is made *once* per resample to avoid redundant computations.

- The warnings about "ill-conditioned bases" is related to using splines for longitude and latitude. Remember those outlying houses on Lincoln Highway?

    - These are printed *as they happen* and are labeled so that you know which sub-model had the issue.

- Unfortunately, these messages don't show up when parallel processing is used (more on that later). We are working on this but it is a tough problem that I've been trying to solve for about a decade.

The Results

A tibble is returned that looks like the `rsample` object with an extra list column called `.metrics`:

```
knn_reg_search
```

```
## #  10-fold cross-validation
## # A tibble: 10 x 4
##    splits            id     .metrics             .notes
##  * <list>            <chr>  <list>               <list>
##  1 <split [2K/220]> Fold01 <tibble [150 × 5]> <tibble [0 × .
##  2 <split [2K/220]> Fold02 <tibble [150 × 5]> <tibble [0 × .
##  3 <split [2K/220]> Fold03 <tibble [150 × 5]> <tibble [0 × .
##  4 <split [2K/220]> Fold04 <tibble [150 × 5]> <tibble [5 × .
##  5 <split [2K/220]> Fold05 <tibble [150 × 5]> <tibble [0 × .
##  6 <split [2K/220]> Fold06 <tibble [150 × 5]> <tibble [0 × .
##  7 <split [2K/220]> Fold07 <tibble [150 × 5]> <tibble [5 × .
##  8 <split [2K/220]> Fold08 <tibble [150 × 5]> <tibble [5 × .
##  9 <split [2K/220]> Fold09 <tibble [150 × 5]> <tibble [5 × .
## 10 <split [2K/219]> Fold10 <tibble [150 × 5]> <tibble [0 × .
```

```
# results for the first fold:
knn_reg_search$.metrics[[1]]
```

```
## # A tibble: 150 x 5
##    neighbors weight_func .metric .estimator .estimate
##        <int> <chr>       <chr>   <chr>          <dbl>
##  1         1 biweight    rmse    standard      0.111
##  2         2 biweight    rmse    standard      0.103
##  3         3 biweight    rmse    standard      0.0982
##  4         4 biweight    rmse    standard      0.0962
##  5         5 biweight    rmse    standard      0.0950
##  6         6 biweight    rmse    standard      0.0940
##  7         7 biweight    rmse    standard      0.0925
##  8         8 biweight    rmse    standard      0.0923
##  9         9 biweight    rmse    standard      0.0926
## 10        10 biweight    rmse    standard      0.0929
## # … with 140 more rows
```

150 rows = 75 models x 2 metrics

Resampled Performance Estimates

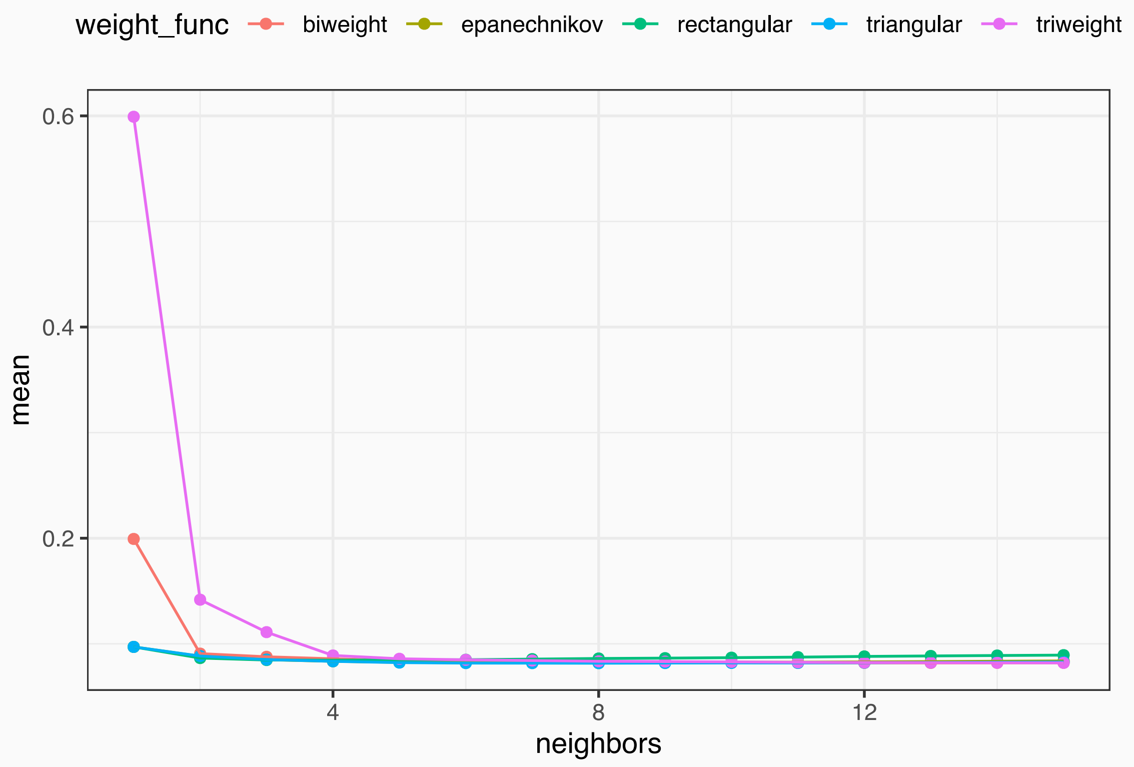
To get the overall resampling estimate (averaged over folds) for each parameter combination:

```
knn_perf <- collect_metrics(knn_reg_search)
knn_perf %>% slice(1:10)
```

```
## # A tibble: 10 x 7
##    neighbors weight_func  .metric .estimator   mean     n std_err
##        <int> <chr>        <chr>   <chr>       <dbl> <int>   <dbl>
##  1         1 biweight     rmse    standard   0.199     10 0.0512
##  2         1 biweight     rsq     standard   0.508     10 0.0920
##  3         1 epanechnikov rmse    standard   0.0971    10 0.00315
##  4         1 epanechnikov rsq     standard   0.717     10 0.0169
##  5         1 rectangular  rmse    standard   0.0971    10 0.00315
##  6         1 rectangular  rsq     standard   0.717     10 0.0169
##  7         1 triangular   rmse    standard   0.0971    10 0.00315
##  8         1 triangular   rsq     standard   0.717     10 0.0169
##  9         1 triweight    rmse    standard   0.599     10 0.0557
## 10         1 triweight    rsq     standard   0.0516    10 0.0111
```

Resampled Performance Estimates

```r
knn_perf %>%
  dplyr::filter(.metric == "rmse") %>%
  ggplot(aes(x = neighbors, y = mean, col = weight_func)) +
  geom_point() +
  geom_line()
```



Zoomed

```r
knn_perf %>%
  dplyr::filter(.metric == "rmse") %>%
  ggplot(aes(x = neighbors, y = mean, col = weight_func)) +
  geom_point() +
  geom_line() +
  ylim(c(0.08, 0.10))
```

Finalizing the objects

If we felt that this MARS model was the best one, the recipe and model would be updated with the final parameter values and train them *on the entire training set*. Choosing the numerically best model:

```
best_knn <- select_best(knn_reg_search, metric = "rmse", maximize = FALSE)
best_knn
```

```
## # A tibble: 1 x 2
##   neighbors weight_func
##       <int> <chr>
## 1         8 triangular
```

```
final_ames_rec <-
  ames_rec %>%
  prep()

final_mars_mod <-
  knn_mod %>%
  finalize_model(best_knn) %>%
  # Recall that sale price has been transformed by the recipe.
  fit(Sale_Price ~ ., data = juice(final_ames_rec))
```