Applied Machine Learning - Basic Principals

Max Kuhn (RStudio)

Introduction

In this section, we will introduce concepts that are useful for any type of machine learning model:

- *modeling* versus the model
- data splitting
- resampling
- tuning parameters and overfitting
- model tuning

Many of these topics will be put into action in later sections.

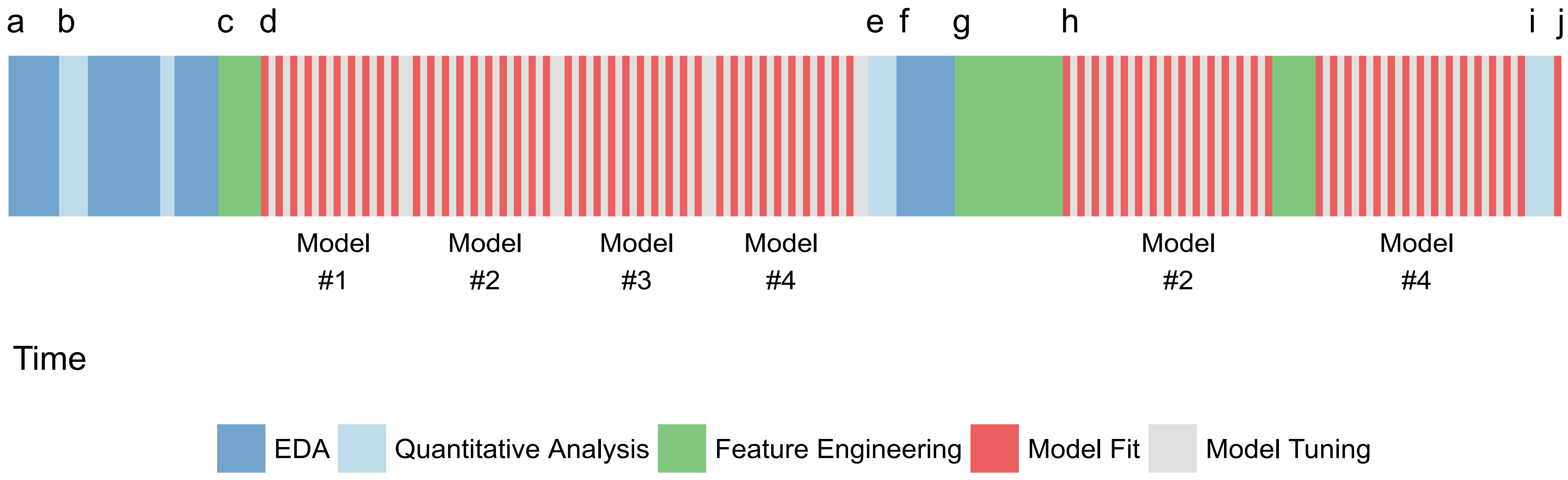
The Modeling *Process*

Common steps during model building are:

- estimating model parameters (i.e. training models)
- determining the values of *tuning parameters* that cannot be directly calculated from the data
- model selection (within a model type) and model comparison (between types)
- calculating the performance of the final model that will generalize to new data

Many books and courses portray predictive modeling as a short sprint. A better analogy would be a marathon or campaign (depending on how hard the problem is).

What the Modeling Process Usually Looks Like



Data Usage

Data Splitting and Spending

How do we "spend" the data to find an optimal model?

We *typically* split data into training and test data sets:

- ***Training Set***: these data are used to estimate model parameters and to pick the values of the complexity parameter(s) for the model.

- ***Test Set***: these data can be used to get an independent assessment of model efficacy. They should not be used during model training.

Data Splitting and Spending

The more data we spend, the better estimates we'll get (provided the data is accurate).

Given a fixed amount of data:

- too much spent in training won't allow us to get a good assessment of predictive performance. We may find a model that fits the training data very well, but is not generalizable (overfitting)

- too much spent in testing won't allow us to get a good assessment of model parameters

Statistically, the best course of action would be to use all the data for model building and use statistical methods to get good estimates of error.

From a non-statistical perspective, many consumers of complex models emphasize the need for an untouched set of samples to evaluate performance.

Large Data Sets

When a large amount of data are available, it might seem like a good idea to put a large amount into the training set. *Personally*, I think that this causes more trouble than it is worth due to diminishing returns on performance and the added cost and complexity of the required infrastructure.

Alternatively, it is probably a better idea to reserve good percentages of the data for specific parts of the modeling process. For example:

- Save a large chunk of data to perform feature selection prior to model building
- Retain data to calibrate class probabilities or determine a cutoff via an ROC curve.

Also, there may be little need for iterative resampling of the data. A single holdout (aka validation set) may be sufficient in some cases if the data are large enough and the data sampling mechanism is solid.

Mechanics of Data Splitting

There are a few different ways to do the split: simple random sampling, *stratified sampling based on the outcome*, by date, or methods that focus on the distribution of the predictors.

For stratification:

- **classification**: this would mean sampling within the classes as to preserve the distribution of the outcome in the training and test sets
- **regression**: determine the quartiles of the data set and samples within those artificial groups

Ames Housing Data

Let's load the example data set and split it. We'll put 75% into training and 25% into testing.

```r
library(AmesHousing)
ames <- make_ames()
nrow(ames)
```

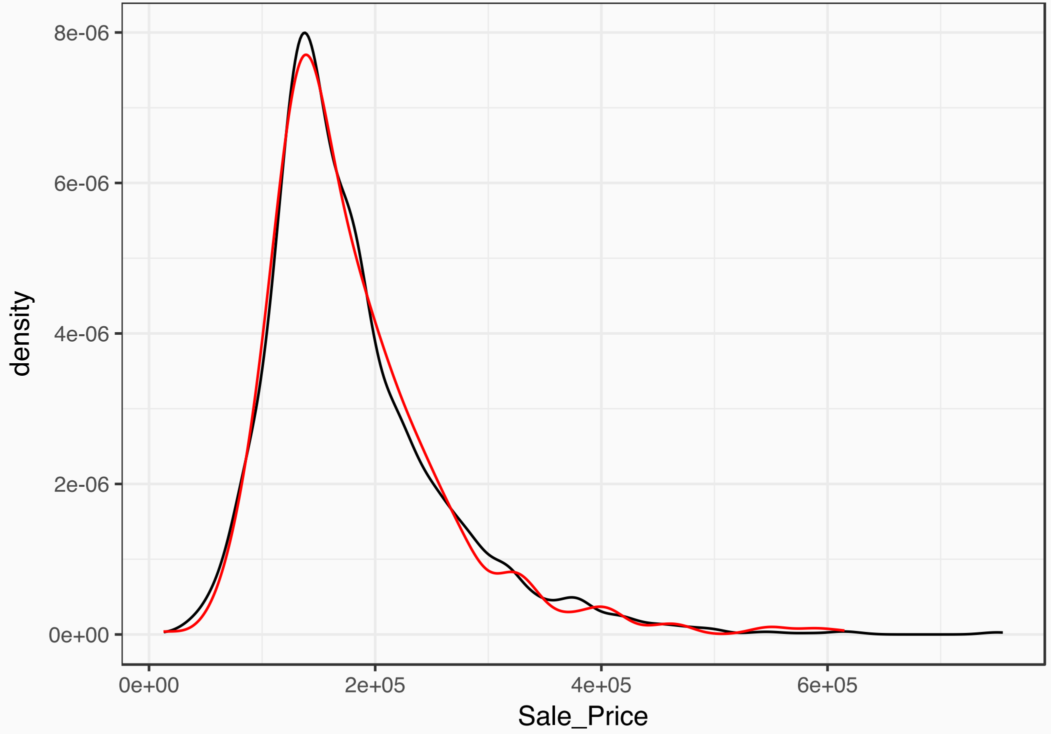
```
## [1] 2930
```

```r
library(rsample)

# Make sure that you get the same random numbers
set.seed(4595)
data_split <- initial_split(ames, strata = "Sale_Price")

ames_train <- training(data_split)
ames_test  <- testing(data_split)

nrow(ames_train)/nrow(ames)
```

```
## [1] 0.7505119
```

Outcome Distributions

```
library(ggplot2)
## Do the distributions line up?
ggplot(ames_train, aes(x = Sale_Price)) +
  geom_line(stat = "density",
            trim = TRUE) +
  geom_line(data = ames_test,
            stat = "density",
            trim = TRUE, col = "red")
```



Creating Models in R

Specifying Models in R Using Formulas

To fit a model to the housing data, the model terms must be specified. Historically, there are two main interfaces for doing this.

The **formula** interface using R formula rules to specify a *symbolic* representation of the terms and variables. For example:

```
foo(Sale_Price ~ Neighborhood + Year_Sold + Neighborhood:Year_Sold, data = ames_train)
```

or

```
foo(Sale_Price ~ ., data = ames_train)
```

or

```
foo(log10(Sale_Price) ~ ns(Longitude, df = 3) + ns(Latitude, df = 3), data = ames_train)
```

This is very convenient but it has some disadvantages.

Downsides to Formulas

- You can't nest in-line functions such as `foo(y ~ pca(scale(x1), scale(x2), scale(x3)), data = dat)`.
- All the model matrix calculations happen at once and can't be recycled when used in a model function.
- For very *wide* data sets, the formula method can be extremely inefficient.
- There are limited *roles* that variables can take which has led to several re-implementations of formulas.
- Specifying multivariate outcomes
- Not all model functions have a formula method.

Specifying Models Without Formulas

Some modeling function have the non-formula interface. This usually has arguments for the predictors and the outcome(s):

```
# Usually, the variables must all be numeric
pre_vars <- c("Year_Sold", "Longitude", "Latitude")
foo(x = ames_train[, pre_vars],
    y = ames_train$Sale_Price)
```

This is inconvenient if you have transformations, factor variables, interactions, or any other operations to apply to the data prior to modeling.

Overall, it is difficult to predict if a package has one or both of these interfaces. For example, `lm` only has formulas.

There is a **third interface**, using *recipes* that will be discussed later that solves some of these issues.

A Linear Regression Model

Let's start by fitting an ordinary linear regression model to the training set. You can choose the model terms for your model but I will use a very simple model:

```
simple_lm <- lm(log10(Sale_Price) ~ Longitude + Latitude, data = ames_train)
```

Before looking at coefficients, we should do some model checking to see if there is anything obviously wrong with the model.

To get the statistics on the individual data points, we will use the awesome `broom` package:

```
library(broom)
simple_lm_values <- augment(simple_lm)
names(simple_lm_values)
```

```
##  [1] "log10.Sale_Price." "Longitude"         "Latitude"
##  [4] ".fitted"           ".se.fit"           ".resid"
##  [7] ".hat"              ".sigma"            ".cooksd"
## [10] ".std.resid"
```

Hands-On: Some Basic Diagnostics

From these results, let's take 10 minutes and do some visualizations:

- Plot the observed versus fitted values
- Plot the residuals
- Plot the predicted versus residuals

Are there any *downsides* to this approach?

Model Evaluation

Overall Model Statistics

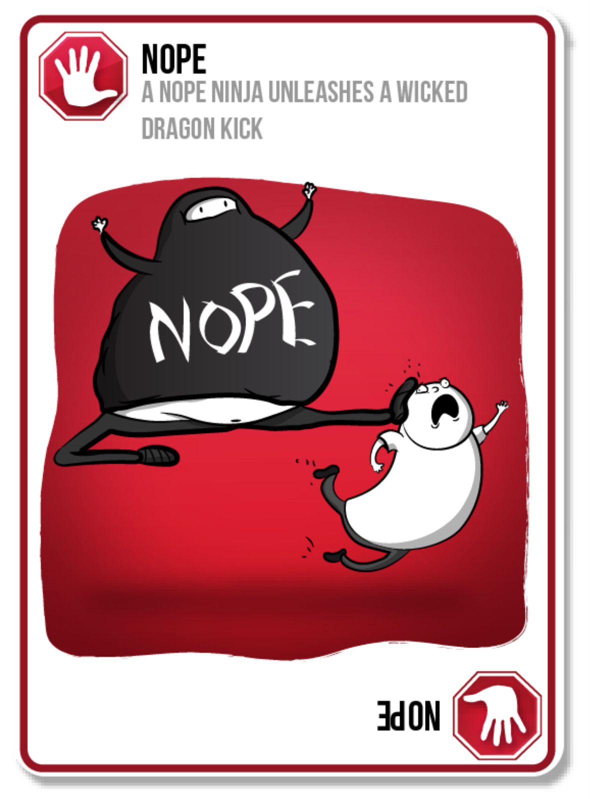
If you use the `summary` method on the `lm` object, the bottom shows some statistics:

```
summary(simple_lm)
```

```
## <snip>
## Residual standard error: 0.1614 on 2196 degrees of freedom
## Multiple R-squared:  0.1808,	Adjusted R-squared:  0.1801
## F-statistic: 242.3 on 2 and 2196 DF,  p-value: < 2.2e-16
```

These statistics are the result of predicting the same data that was used to derive the coefficients. This is problematic because it can lead to optimistic results, especially for models that are extremely flexible.

The tests set is used for assessing performance. **Should we predict the test set** and use those results to estimate these statistics?



(Matthew Inman/Exploding Kittens)

Assessing Models

Save the test set until the very end when you have one or two models that are your favorite.

We'll need to use the training set but....

For some models, it is possible to get very small residuals by predicting the training set.

That's an issue since we will need to make comparisons between models, create diagnostic plots, etc.

If only we had a method for getting honest performance estimates from the *training set*... 🤔

Resampling Methods

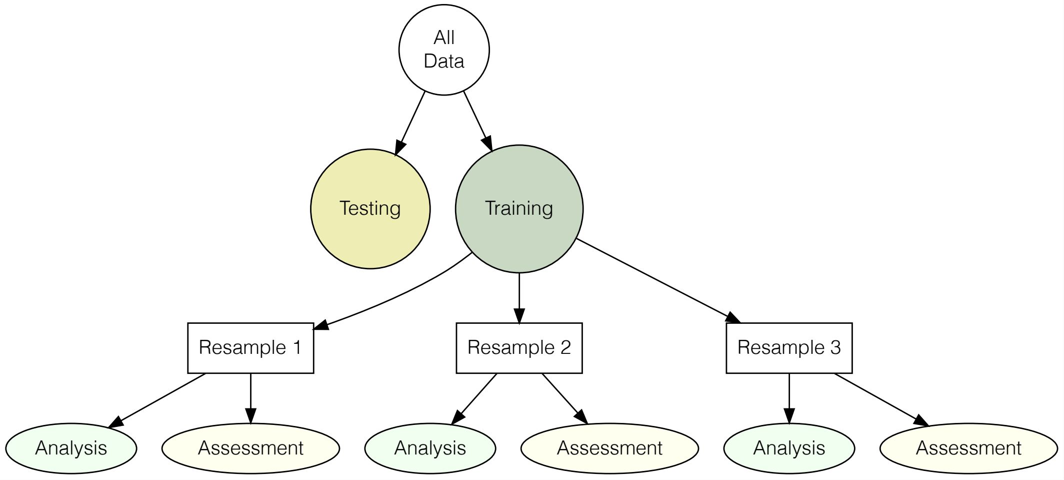
These are additional data splitting schemes that are applied to the *training* set.

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 or resampling but we will focus on two methods.



V-Fold Cross-Validation

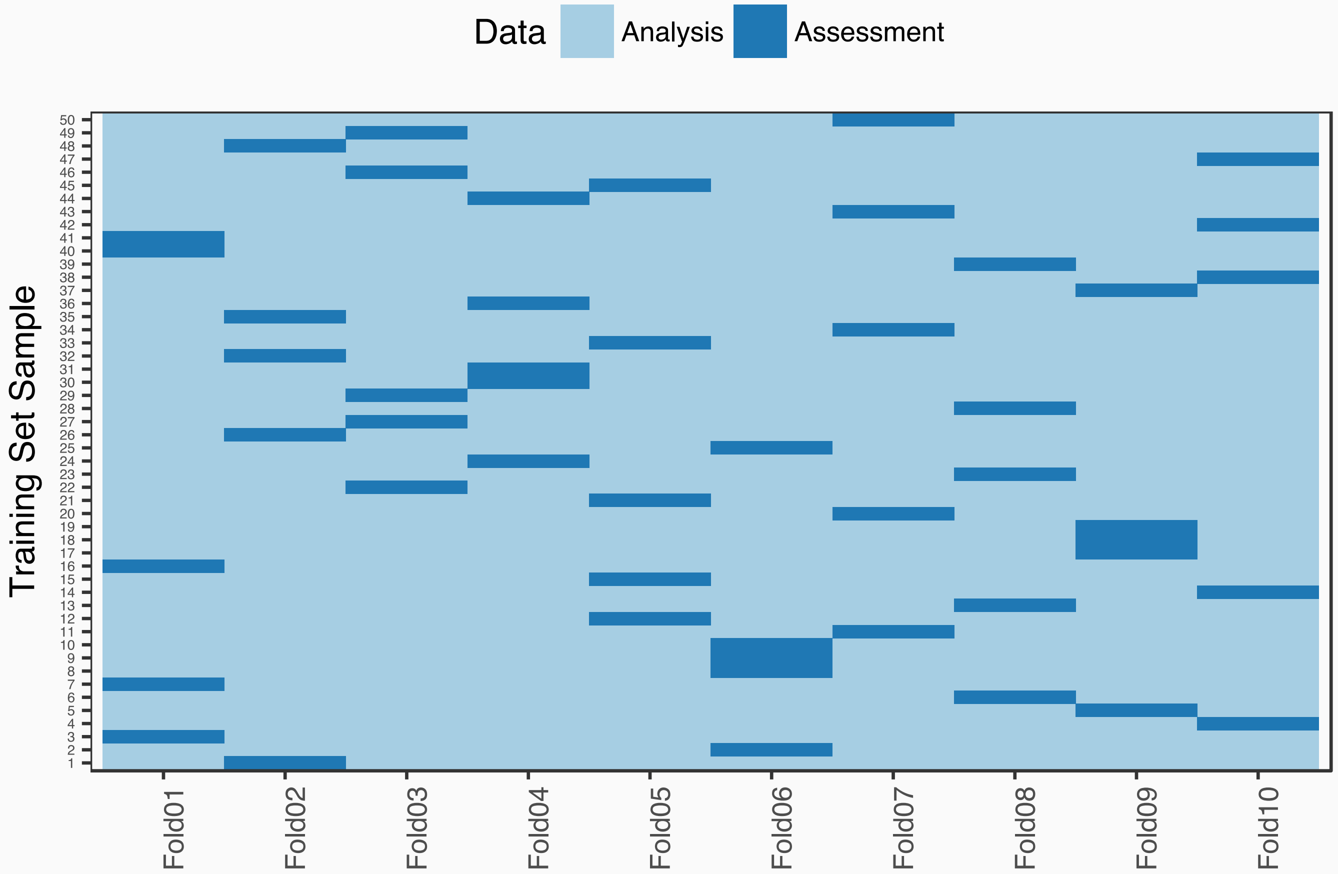
Here, we randomly split the training data into V distinct blocks of roughly equal size.

- 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.

10-Fold Cross-Validation with *n* = 50



Bootstrapping

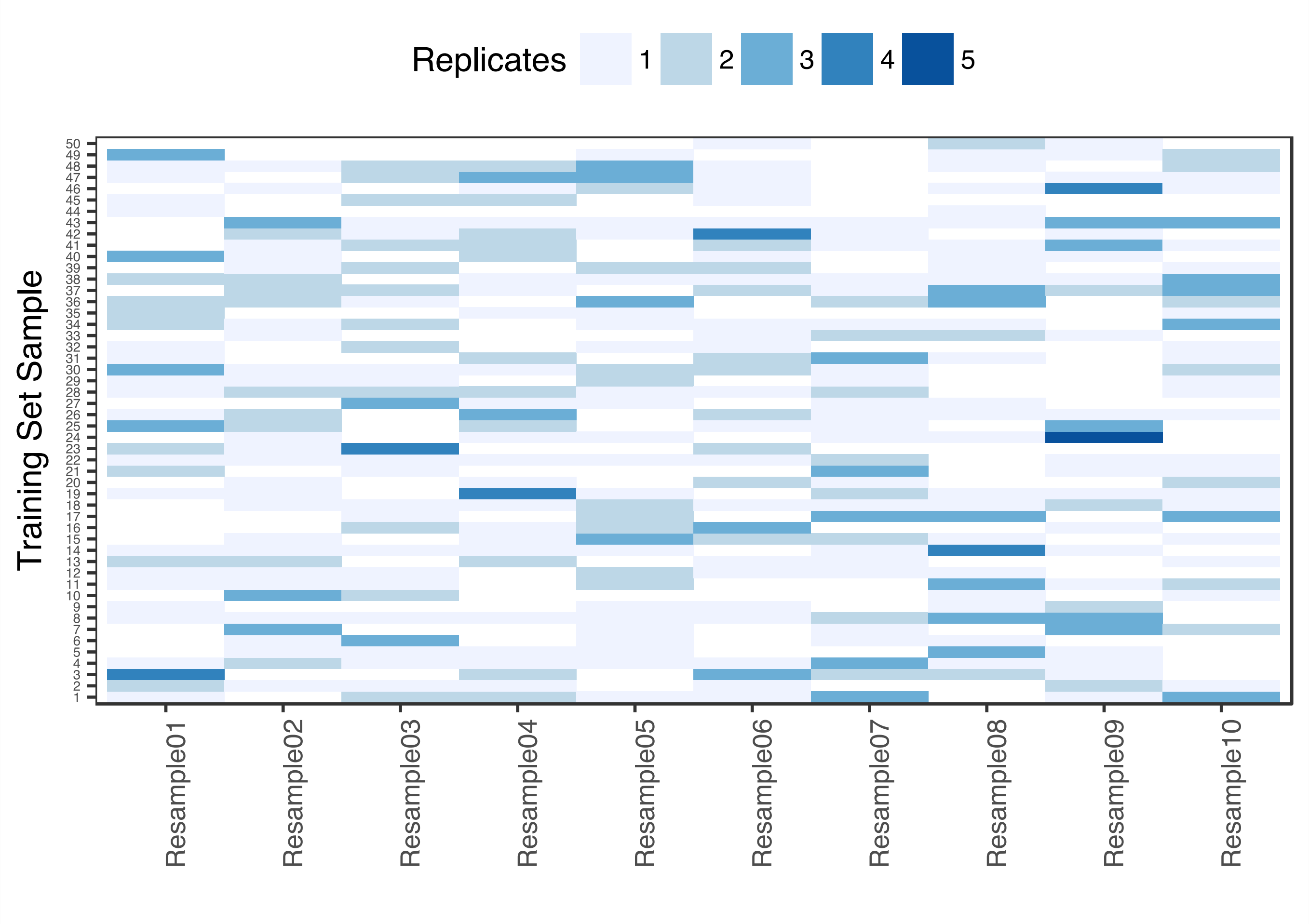
A bootstrap sample is the same size as the training set but each data point is selected *with replacement*.

This means that the analysis set will have more than one replicate of a training set instance.

The assessment set contains all samples that were never included in the bootstrap set. It is often called the "out-of-bag" sample and can vary in size.

On average, 63.2120559% of the training set is contained at least once in the bootstrap sample.

Bootstrapping with *n* = 50



Comparing Resampling Methods

If you think of resampling in the same manner as statistical estimators (e.g. maximum likelihood), this becomes a trade-off between bias and variance:

- Variance is (mostly) driven by the number of resamples (e.g. 5-fold CV has larger variance than 10-fold).
- Bias is (mostly) related to how much data is held back. The bootstrap has large bias compared to 10-fold CV.

There are lengthy blog posts about this subject here and here.

I tend to favor 5 repeats of 10-fold cross-validation unless the size of the assessment data is is "large enough".

For example, 10% of the Ames training set is 219 properties and this is probably good enough to estimate the RMSE and R^2.

Cross-Validating Using `rsample`

```r
library(rsample)
set.seed(2453)
cv_splits <- vfold_cv(ames_train, v = 10, strata = "Sale_Price")
cv_splits
```

```
## #  10-fold cross-validation using stratification
## # A tibble: 10 x 2
##    splits       id
##    <list>       <chr>
##  1 <S3: rsplit> Fold01
##  2 <S3: rsplit> Fold02
##  3 <S3: rsplit> Fold03
##  4 <S3: rsplit> Fold04
##  5 <S3: rsplit> Fold05
##  6 <S3: rsplit> Fold06
##  7 <S3: rsplit> Fold07
##  8 <S3: rsplit> Fold08
##  9 <S3: rsplit> Fold09
## 10 <S3: rsplit> Fold10
```

Cross-Validating Using `rsample`

```r
# The `split` objects contain the information about the sample sizes
cv_splits$splits[[1]]
```

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

```r
# Use the `analysis` and `assessment` functions to get the data
analysis(cv_splits$splits[[1]]) %>% dim()
```

```
## [1] 1977   81
```

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

```
## [1] 222  81
```

Resampling the Linear Model

We'll need to write a function to fit the model to each data set and another to compute performance.

```r
library(yardstick)
lm_fit <- function(data_split, ...)
  lm(..., data = analysis(data_split))

# A formula is also needed for each model:
form <- as.formula(
    log10(Sale_Price) ~ Longitude + Latitude
    )
```

For performance, the first argument should be the `rsplit` object contained in `cv_splits$splits`:

```r
model_perf <- function(data_split, mod_obj) {
  vars <- rsample::form_pred(mod_obj$terms)
  assess_dat <- assessment(data_split) %>%
      select(!!!vars, Sale_Price) %>%
      mutate(
          pred = predict(
              mod_obj,
              newdata = assessment(data_split)
          ),
          Sale_Price = log10(Sale_Price)
      )

  rmse <- assess_dat %>%
      rmse(truth = Sale_Price, estimate = pred)
  rsq <- assess_dat %>%
      rsq(truth = Sale_Price, estimate = pred)
  data.frame(rmse = rmse, rsq = rsq)
}
```

Resampling the Linear Model

The `purrr` package will be used to fit the model to each analysis set. These will be saved in a column called `lm_mod`:

```
library(purrr)
cv_splits <- cv_splits %>%
  mutate(lm_mod = map(splits, lm_fit, formula = form))
cv_splits
```

```
## #  10-fold cross-validation using stratification
## # A tibble: 10 x 3
##    splits       id     lm_mod
##  * <list>       <chr>  <list>
##  1 <S3: rsplit> Fold01 <S3: lm>
##  2 <S3: rsplit> Fold02 <S3: lm>
##  3 <S3: rsplit> Fold03 <S3: lm>
##  4 <S3: rsplit> Fold04 <S3: lm>
##  5 <S3: rsplit> Fold05 <S3: lm>
##  6 <S3: rsplit> Fold06 <S3: lm>
##  7 <S3: rsplit> Fold07 <S3: lm>
##  8 <S3: rsplit> Fold08 <S3: lm>
##  9 <S3: rsplit> Fold09 <S3: lm>
## 10 <S3: rsplit> Fold10 <S3: lm>
```

Resampling the Linear Model

Now, let's compute the two performance measures:

```
# map2 can be used to move over two objects of equal length
library(dplyr)
lm_res <- map2_df(cv_splits$splits, cv_splits$lm_mod, model_perf) %>%
  dplyr::rename(rmse_simple = rmse, rsq_simple = rsq)
head(lm_res, 3)
```

```
##   rmse_simple rsq_simple
## 1   0.1498657  0.3033558
## 2   0.1635438  0.1565590
## 3   0.1604881  0.2151186
```

```
## Merge in results:
cv_splits <- cv_splits %>% bind_cols(lm_res)

## Rename the columns and compute the resampling estimates:
cv_splits %>% select(rmse_simple, rsq_simple) %>% colMeans
```

```
## rmse_simple  rsq_simple
##   0.1612810   0.1833999
```

What Was the Ruckus?

Previously, I mentioned that the performance metrics that were naively calculated from the training set could be optimistic. However, this approach estimates the RMSE to be 0.1614 and cross-validation produced an estimate of 0.1613. What was the big deal?

Linear regression is a *high bias model*. This means that it is fairly incapable at being able to adapt the underlying model function (unless it is linear). For this reason, linear regression is unlikely to **overfit** to the training set and our two estimates are likely to be the same.

We'll consider another model shortly that is *low bias* since it can, theoretically, easily adapt to a wide variety of true model functions.

However, as before, there is also variance to consider. Linear regression is very stable since it leverages all of the data points to estimate parameters. Other methods, such as tree-based models, are not and can drastically change if the training set data is slightly perturbed.

tl;dr: the earlier concern is real but linear regression is less likely to be affected.

Diagnostics Again

Now let's look at diagnostics using the predictions from the assessment sets.

```
get_assessment <- function(splits, model)
  augment(model, newdata = assessment(splits)) %>%
      mutate(.resid = log10(Sale_Price) - .fitted)

holdout_results <- map2_df(cv_splits$splits, cv_splits$lm_mod, get_assessment)
holdout_results %>% dim()
```

```
## [1] 2199   84
```

```
ames_train %>% dim()
```

```
## [1] 2199   81
```

Hands-On: Partial Residual Plots

A partial residual plot is used to diagnose what variables *should* have been in the model.

We can plot the hold-out residuals versus different variables to understand if they should have been in the model

- If the residuals have no pattern in the data, they are likely to be irrelevant.

- If a pattern is seen, it suggests that the variable should have been in the model.

Take 10 min and use `ggplot` to investigate the other predictors using the `holdout_results` data frame. `geom_smooth` might come in handy.

p.s. I have decided that using the `Overall_Qual` variable is cheating; the assessors determine that when they determine the assessment price of the house.

Tuning Parameters and Overfitting

K-Nearest Neighbors Model

Now let's consider a more flexible model that is *low bias*: *K*-nearest neighbors.

The model 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.

When *K* is small, the model might be *too* responsive to the underlying data. When *K* is large, it begins to "over smooth" the neighbors and performance suffers.

Ordinarily, since we are computing a distance, we would want to center and scale the predictors. Our two predictors are already on the same scale so we can skip this step.

5-Nearest Neighbors Model

K-Nearest Neighbors Model

Consider the 2-nearest neighbor model. Would there be a difference in the estimated model performance between re-prediction and cross-validation?

`caret` has a `knnreg` function that can be used (the `kknn` package is another good option). It has a formula method and we'll use this to illustrate the model:

```r
library(caret)

knn_train_mod <- knnreg(log10(Sale_Price) ~ Longitude + Latitude,
                        data = ames_train,
                        k = 2)
repredict <- data.frame(price = log10(ames_train$Sale_Price)) %>%
  mutate(pred =
           predict(knn_train_mod,
                   newdata = ames_train %>% select(Longitude, Latitude)
           )
  )

repredict %>% rsq(truth = "price", estimate = "pred") # <- the ruckus is here
```

```
## [1] 0.892872
```

Resampling a 2-Nearest Neighbor Model

That's pretty good but are we tricking ourselves? One of those two neighbors is always itself...

To resample, let's create another function to fit this model and follow the same resampling process as before:

```
knn_fit <- function(data_split, ...)
  knnreg(..., data = analysis(data_split))

cv_splits <- cv_splits %>%
    mutate(knn_mod = map(splits, knn_fit, formula = form, k = 2))

knn_res <- map2_df(cv_splits$splits, cv_splits$knn_mod, model_perf) %>%
  rename(rmse_knn = rmse, rsq_knn = rsq)

## Merge in results:
cv_splits <- cv_splits %>% bind_cols(knn_res)

colMeans(knn_res)
```

```
##  rmse_knn   rsq_knn
## 0.1032672 0.6745020
```

Making Formal Comparisons

The model appears to be a drastic improvement over simple linear regression but we are definitely getting highly optimistic results by re-predicting the training set.

We can try to make a more formal assessment of the two current models.

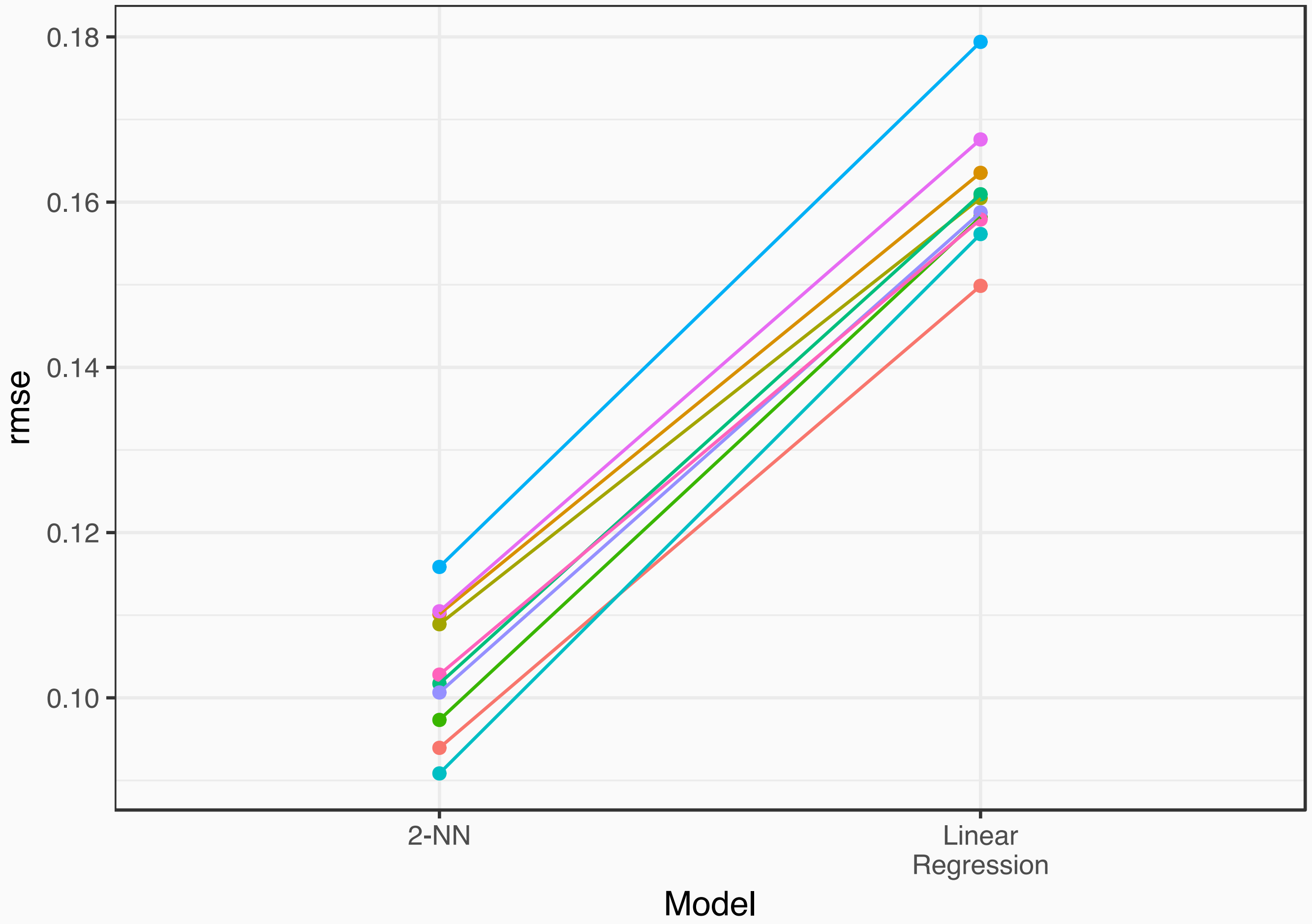
Both models used the *same* resamples, so we have 10 estimates of performance that are matched.

Does the matching mean anything?

Most likely **yes**. It is very common to see that there is a resample effect. Similar to repeated measures designs, we can expect a relationship between models and resamples. For example, some resamples will have the worst performance over different models and so on.

In other words, there is usually a within-resample correlation. For the two models, the estimated correlation in RMSE values is 0.85.

The Resample Effect



Model Comparison Accounting for Resampling

With only two models, a paired *t*-test can be used to estimate the difference in RMSE between the models:

```
t.test(cv_splits$rmse_simple, cv_splits$rmse_knn, paired = TRUE)
```

```
## 
##      Paired t-test
## 
## data:  cv_splits$rmse_simple and cv_splits$rmse_knn
## t = 42.258, df = 9, p-value = 1.161e-11
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  0.05490826 0.06111941
## sample estimates:
## mean of the differences
##              0.05801383
```

Hothorn *et al* (2012) is the original paper on comparing models using resampling.

We'll do more extensive analyses with `tidyposterior` soon.

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.

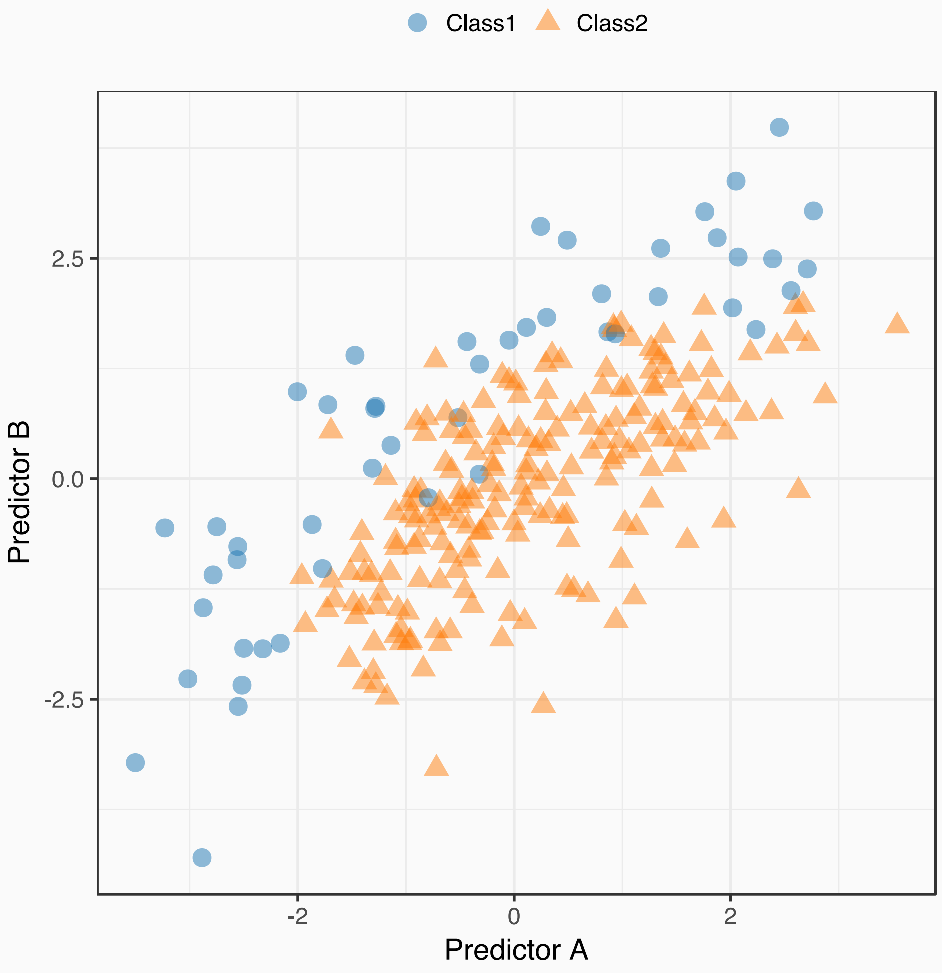
Some models have specific "knobs" to control over-fitting

- neighborhood size in nearest neighbor models is an example
- the number of splits in a tree model

Often, poor choices for these parameters can result in overfitting

For example, the next slide shows a data set with two predictors. We want to be able to produce a line (i.e. decision boundary) that differentiates two classes of data.

Two Class Example

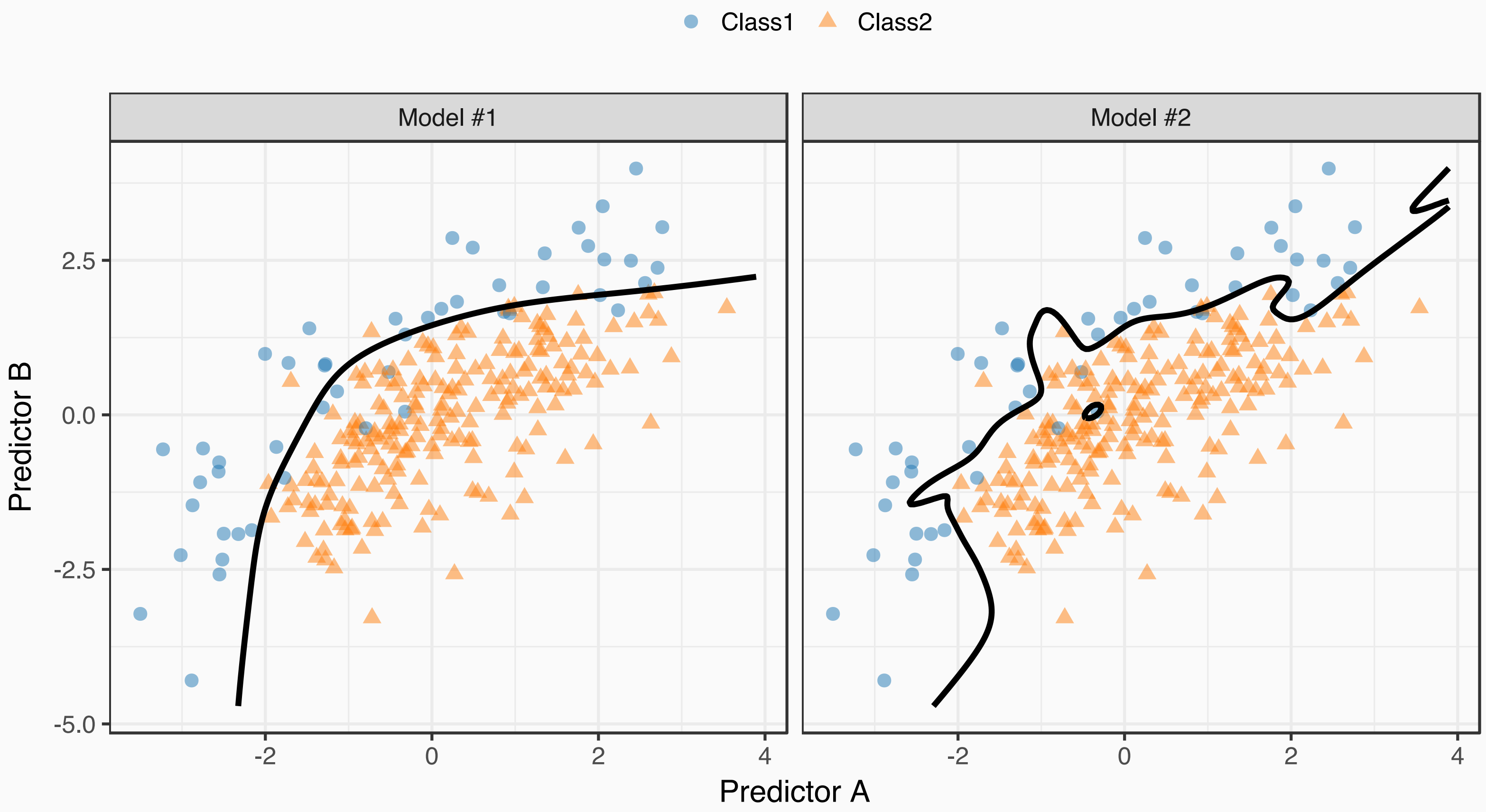


On the next slide, two classification boundaries are shown for the a different model type not yet discussed.

The difference in the two panels is solely due to different choices in tuning parameters.

One overfits the training data.

Two Model Fits



Grid Search to Tune Models

We usually don't have two-dimensional data so a quantitative method for under measuring overfitting is needed. *Resampling* fits that description. A simple method for tuning a model is to used *grid search*:

```
├── Create a set of candidate tuning parameter values
└── For each resample
|    ├── Split the data into analysis and assessment sets
|    ├── [preprocess data]
|    ├── For each tuning parameter value
|    |    ├── Fit the model using the analysis set
|    |    └── Compute the performance on the assessment set and save
├── For each tuning parameter value, average the performance over resamples
├── Determine the best tuning parameter value
└── Create the final model with the optimal parameter(s) on the training set
```

Random search is a similar technique where the candidate set of parameter values are simulated at random across a wide range. Also, an example of *nested resampling* can be found here.

Grid Search Computations

The bad news is that all of the models (except the final model) are discarded.

The good news is that all of the models (except the final model) can be run in parallel.

Let's look at the Ames *K*-NN model and evaluate *K*=1, 2, ..., 20 using the same 10-fold cross-validation as before.

We'll start coding this algorithm from the inside out…

Computing Performance

These steps are:

```
├── Fit the model using the analysis set
└── Compute the performance on the assessment set and save
```

In the code below, `split` will be one of the elements of `cv_splits$splits`

```
knn_rmse <- function(k, split) {
    mod <- knnreg(log10(Sale_Price) ~ Longitude + Latitude,
                            data = analysis(split),
                            k = k)
    # Extract the names of the predictors
    preds <- form_pred(mod$terms)
    data.frame(Sale_Price = log10(assessment(split)$Sale_Price)) %>%
        mutate(pred = predict(mod, assessment(split) %>% select(!!!preds))) %>%
        rmse(Sale_Price, pred)
}
```

The return value is a single number (the RMSE estimate).

Fit the Model Across Values of *K*

```
|    ├── For each tuning parameter value
|    |    └── Run `knn_rmse`
```

```r
knn_grid <- function(split) {
    # Create grid
  tibble(k = 1:20) %>%
    # Execute grid for this resample
    mutate(
      rmse = map_dbl(k, knn_rmse, split = split),
      # Attach the resample indicators using `lables`
      id = labels(split)[[1]]
    )
}
```

The return value here is a tibble with columns for `k`, the RMSE, and the fold ID (e.g. `Fold01`).

Top-Level Iteration Over Resamples

```
└── For each resample
|   └── Run `knn_grid`
```

Here, `resamp` is the resample object `cv_splits`

```
iter_over_resamples <-
    function(resamp)
        map_df(resamp$splits, knn_grid)
```

Running the Code

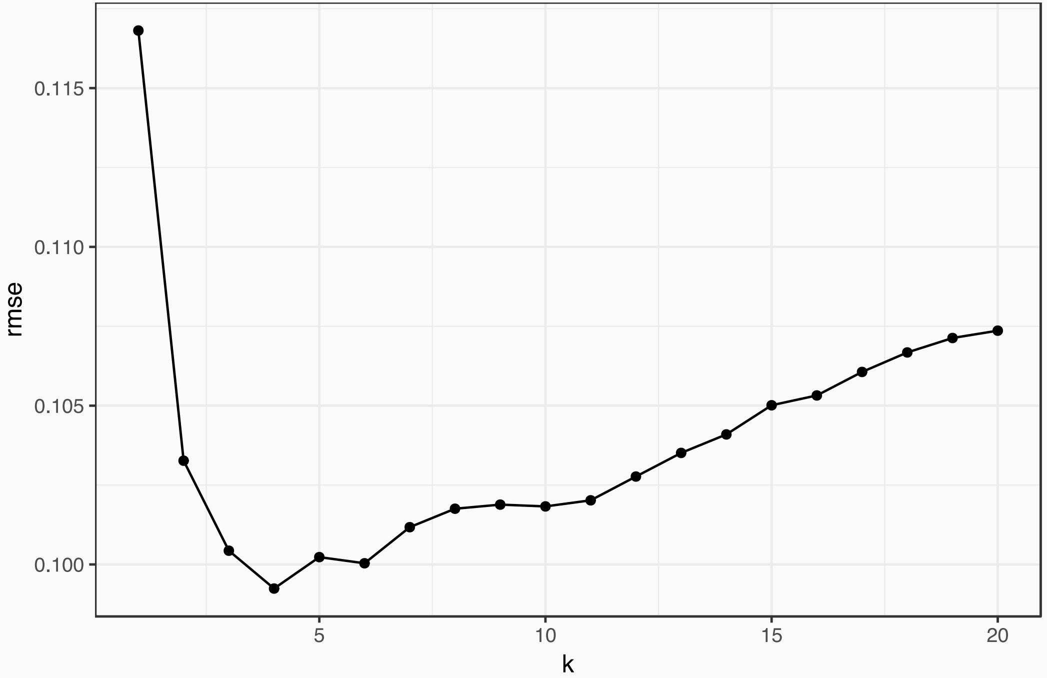
```
knn_tune_res <- iter_over_resamples(cv_splits)
knn_tune_res %>% head(15)
```

```
## # A tibble: 15 x 3
##         k   rmse id
##     <int>  <dbl> <chr>
##   1     1 0.111  Fold01
##   2     2 0.0939 Fold01
##   3     3 0.0930 Fold01
##   4     4 0.0892 Fold01
##   5     5 0.0903 Fold01
##   6     6 0.0920 Fold01
##   7     7 0.0910 Fold01
##   8     8 0.0911 Fold01
##   9     9 0.0901 Fold01
## 10    10 0.0896 Fold01
## 11    11 0.0914 Fold01
## 12    12 0.0926 Fold01
## 13    13 0.0934 Fold01
## 14    14 0.0946 Fold01
## 15    15 0.0952 Fold01
```

The Performance Profile

To summarize the results for each value of *K*:

```
rmse_by_k <- knn_tune_res %>%
  group_by(k) %>%
  summarize(rmse = mean(rmse))

ggplot(rmse_by_k, aes(x = k, y = rmse)) +
  geom_point() + geom_line()
```

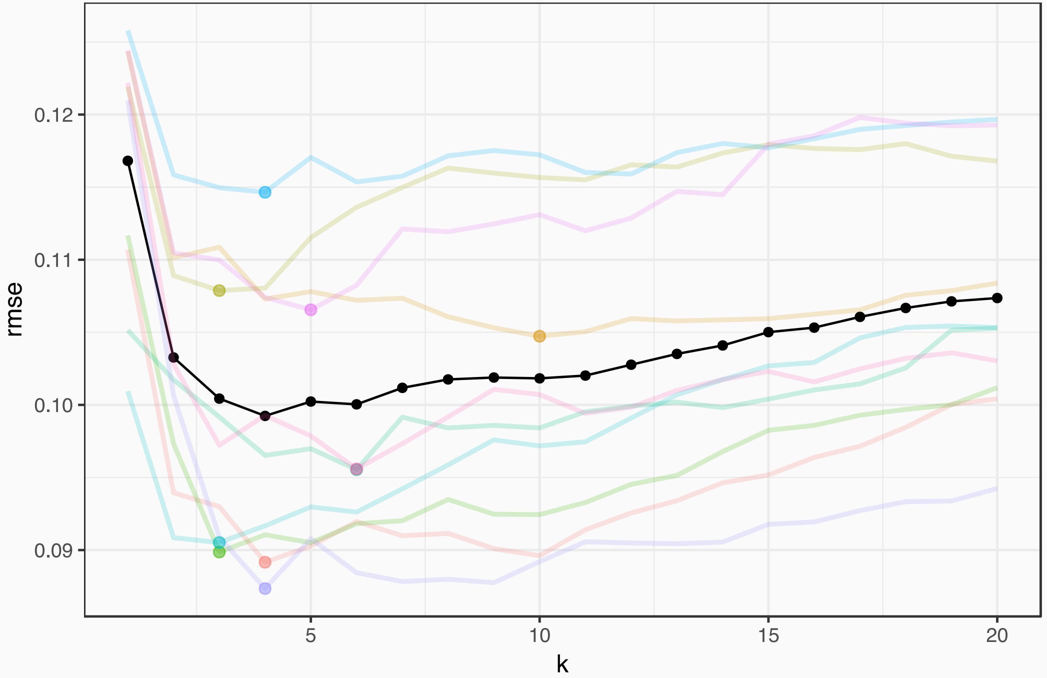


Although it is numerically optimal, we are not required to use a value of 4 neighbors for the final model.

Resampling Variation

How stable is this? We can also plot the individual curves and their minimums.

```
best_k <- knn_tune_res %>%
  group_by(id) %>%
  summarize(k = k[which.min(rmse)],
            rmse = rmse[which.min(rmse)])

ggplot(rmse_by_k, aes(x = k, y = rmse)) +
  geom_point() + geom_line() +
  geom_line(data = knn_tune_res,
            aes(group = id,
                col = id),
            alpha = .2, lwd = 1) +
  geom_point(data = best_k,
             aes(col = id),
             alpha = .5, cex = 2) +
  theme(legend.position = "none")
```



Next Steps

At this point, we would decide on a good value for *K* and then fit the model used going forward:

```
final_knn <- knnreg(log10(Sale_Price) ~ Longitude + Latitude,
                    data = ames_train,
                    k = 4)
```

To reiterate: the previous 200 models created during the grid search are not used once *K* is set.

Later, we will look at a high-level API in `caret` that streamlines almost all of this process for many different models.