# Modeling in the Tidyverse

Max Kuhn (RStudio)

## Workshop Overview

> *Basic familiarity with R and the tidyverse is required.*

The *goal* is for you to be able to easily build predictive/machine learning models in R using a variety of packages and model types.

- "Models that are focused on prediction"... what does that mean?
- "Machine Learning"... so this is deep learning with massive data sets, right?

The course is broken up into sections for *fitting models* and *preprocessing data*.

## Why R for Modeling?

- *R has cutting edge models.*

  Machine learning developers in some domains use R as their primary computing environment and their work often results in R packages.

- *It is easy to port or link to other applications.*

  R doesn't try to be everything to everyone. If you prefer models implemented in C, C++, `tensorflow`, `keras`, `python`, `stan`, or `Weka`, you can access these applications without leaving R.

- *R and R packages are built by people who **do** data analysis.*

- *The S language is very mature.*

- The machine learning environment in R is extremely rich.

## Downsides to Modeling in R

- R is a data analysis language and is not C or Java. If a high performance deployment is required, R can be treated like a prototyping language.
- R is mostly memory-bound. There are plenty of exceptions to this though.

The main issue is one of *consistency of interface*. For example:

- There are two methods for specifying what terms are in a model[1]. Not all models have both.
- 99% of model functions automatically generate dummy variables.
- Sparse matrices can be used (unless they can't).

[1] There are now three but the last one is brand new and will be discussed later.

## Syntax for Computing Predicted Class Probabilities

| Function | Package | Code |
|---|---|---|
| `lda` | `MASS` | `predict(obj)` |
| `glm` | `stats` | `predict(obj, type = "response")` |
| `gbm` | `gbm` | `predict(obj, type = "response", n.trees)` |
| `mda` | `mda` | `predict(obj, type = "posterior")` |
| `rpart` | `rpart` | `predict(obj, type = "prob")` |
| `Weka` | `RWeka` | `predict(obj, type = "probability")` |
| `logitboost` | `LogitBoost` | `predict(obj, type = "raw", nIter)` |

We'll see a solution for this later in the class.

## `tidymodels` Collection of Packages

```r
library(tidymodels)
```

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

```
## ✔ broom     0.5.1          ✔ purrr     0.3.2
## ✔ dials     0.0.2          ✔ recipes   0.1.4.9000
## ✔ dplyr     0.8.0.1        ✔ rsample   0.0.4.9000
## ✔ ggplot2   3.1.1          ✔ 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()
```

Plus `tidypredict`, `tidyposterior`, `tidytext`, and more in development.

## Example Data Set - House Prices

For our examples, we will use the Ames IA housing data. There are 2,930 properties in the data.

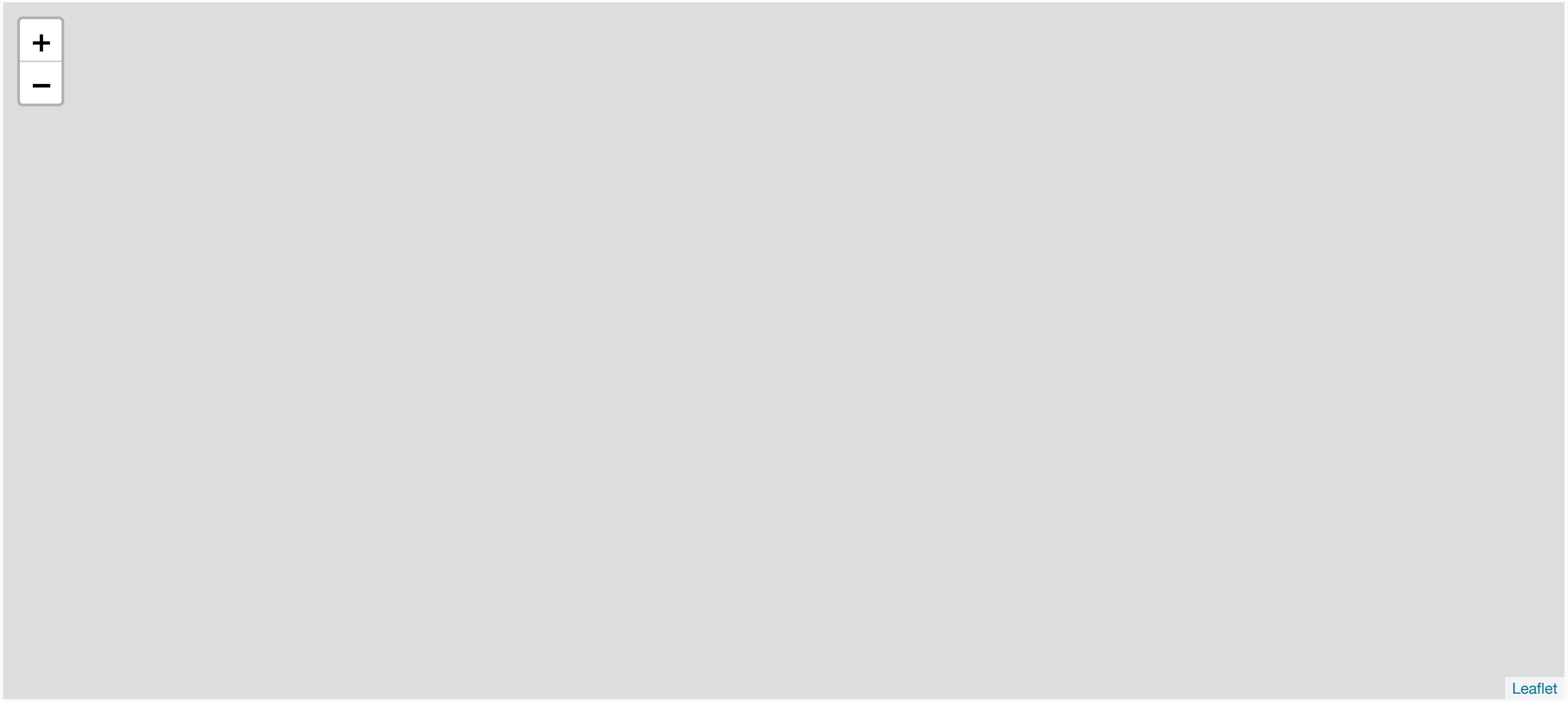
The Sale Price was recorded along with 81 predictors, including:

- Location (e.g. neighborhood) and lot information.
- House components (garage, fireplace, pool, porch, etc.).
- General assessments such as overall quality and condition.
- Number of bedrooms, baths, and so on.

More details can be found in De Cock (2011, Journal of Statistics Education).

The raw data are at `http://bit.ly/2whgsQM` but we will use a processed version found in the `AmesHousing` package.

## Example Data Set - House Prices



## Tidyverse Syntax

Many tidyverse functions have syntax unlike base R code. For example:

- Vectors of variable names are eschewed in favor of *functional programming*. For example:

```
contains("Sepal")

# instead of

c("Sepal.Width", "Sepal.Length")
```

- The *pipe* operator is preferred. For example:

```
merged <- inner_join(a, b)

# is equal to

merged <- a %>%
  inner_join(b)
```

- Functions are more *modular* than their traditional analogs (`dplyr`'s `filter()` and `select()` vs `base::subset()`)

## Some Example Data Manipulation Code

```r
library(tidyverse)

ames_prices <- "http://bit.ly/2whgsQM" %>%
  read_delim(delim = "\t", guess_max = 2000) %>%
  rename_at(vars(contains(' ')), list(~gsub(' ', '_', .))) %>%
  dplyr::rename(Sale_Price = SalePrice) %>%
  dplyr::filter(!is.na(Electrical)) %>%
  dplyr::select(-Order, -PID, -Garage_Yr_Blt)

ames_prices %>%
  group_by(Alley) %>%
  summarize(
    mean_price = mean(Sale_Price / 1000),
    n = sum(!is.na(Sale_Price))
  )
```

```
## # A tibble: 3 x 3
##   Alley mean_price     n
##   <chr>      <dbl> <int>
## 1 <NA>        183.  2731
## 2 Grvl        124.   120
## 3 Pave        177.    78
```

# Examples of `purrr::map*`

purrr contains functions that *iterate over lists* without the explicit use of loops. They are similar to the family of apply functions in base R, but are type stable.

```r
# purrr loaded with tidyverse or tidymodels package

mini_ames <- ames_prices %>%
  dplyr::select(Alley, Sale_Price, Yr_Sold) %>%
  dplyr::filter(!is.na(Alley))

head(mini_ames, n = 5)
```

```
## # A tibble: 5 x 3
##   Alley Sale_Price Yr_Sold
##   <chr>      <int>   <int>
## 1 Pave      190000    2010
## 2 Pave      155000    2010
## 3 Pave      151000    2010
## 4 Pave      149500    2010
## 5 Pave      152000    2010
```

```r
by_alley <- split(mini_ames, mini_ames$Alley)
# map(.x, .f, ...)
map(by_alley, head, n = 2)
```

```
## $Grvl
## # A tibble: 2 x 3
##   Alley Sale_Price Yr_Sold
##   <chr>      <int>   <int>
## 1 Grvl       96500    2010
## 2 Grvl      109500    2010
##
## $Pave
## # A tibble: 2 x 3
##   Alley Sale_Price Yr_Sold
##   <chr>      <int>   <int>
## 1 Pave      190000    2010
## 2 Pave      155000    2010
```

## Examples of `purrr::map*`

```
map(by_alley, nrow)
```

```
## $Grvl
## [1] 120
##
## $Pave
## [1] 78
```

`map()` always returns a list. Use suffixed versions for simplification of the result.

```
map_int(by_alley, nrow)
```

```
## Grvl Pave
##  120   78
```

Complex operations can be specified using a *formula notation*. Access the current thing you are iterating over with `.x`.

```
map(
  by_alley,
  ~summarise(.x, max_price = max(Sale_Price))
)
```

```
## $Grvl
## # A tibble: 1 x 1
##   max_price
##       <dbl>
## 1    256000
##
## $Pave
## # A tibble: 1 x 1
##   max_price
##       <dbl>
## 1    345000
```

## `purrr` and list-columns

Rather than using `split()`, we can `tidyr::nest()` by `Alley` to get a data frame with a *list-column*. We often use these when working with *multiple models*.

```r
ames_lst_col <- nest(mini_ames, -Alley)
ames_lst_col
```

```
## # A tibble: 2 x 2
##   Alley data
##   <chr> <list>
## 1 Pave  <tibble [78 × 2]>
## 2 Grvl  <tibble [120 × 2]>
```

```r
ames_lst_col %>%
  mutate(
    n_row = map_int(data, nrow),
    max   = map_dbl(data, ~max(.x$Sale_Price))
  )
```

```
## # A tibble: 2 x 4
##   Alley data               n_row    max
##   <chr> <list>             <int>  <dbl>
## 1 Pave  <tibble [78 × 2]>     78 345000
## 2 Grvl  <tibble [120 × 2]>   120 256000
```

## Quick Data Investigation

To get warmed up, let's load the real Ames data and do some basic investigations into the variables, such as exploratory visualizations or summary statistics. The idea is to get a feel for the data.

Let's take 10 minutes to work on your own or with someone next to you. Collaboration is highly encouraged!

To get the data:

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

## Resources

- `http://www.tidyverse.org/`
- R for Data Science
- Jenny's `purrr` tutorial or Happy R Users Purrr
- Programming with `dplyr` vignette
- Selva Prabhakaran's `ggplot2` tutorial
- `caret` package documentation
- CRAN Machine Learning Task View

About these slides.... they were created with Yihui's `xaringan` and the stylings are a slightly modified version of Patrick Schratz's Metropolis theme.

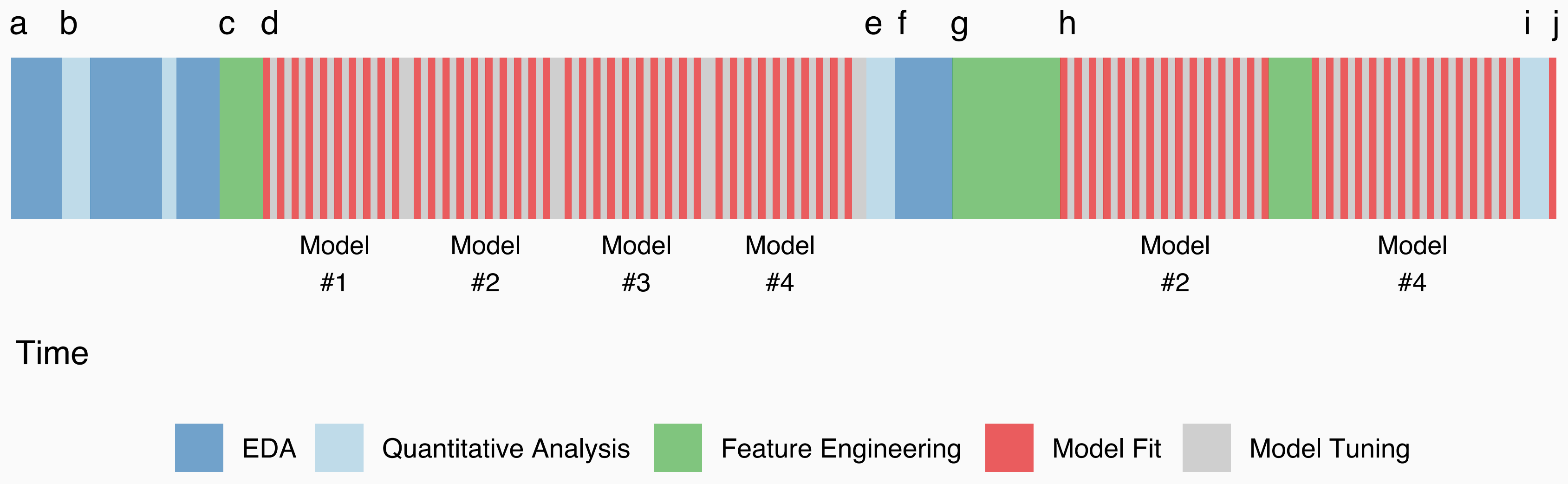
## The Modeling *Process*

Common steps during model building are:

- estimating model parameters (i.e. training models)
- determining the values of *tuning parameter*s 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.

## 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 to preserve the distribution of the outcome in the training and test sets
- **regression**: determine the quartiles of the data set and sample 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
# resample loaded with tidyverse or tidymodels package
ames <-
  make_ames() %>%
  # Remove quality-related predictors
  dplyr::select(-matches("Qu"))
nrow(ames)
```

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

```r
# resample functions
# 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
```

## Ames Housing Data

What do these objects look like?

```
# result of initial_split()
# <training / testing / total>
data_split
```

```
## <2199/731/2930>
```

```
training(data_split)
```

```
## # A tibble: 2,199 x 81
##    MS_SubClass MS_Zoning Lot_Frontage Lot_Area Street Alley Lot_Shape Land_Contour Utilities Lot_Config Land_Slope
##    <fct>       <fct>            <dbl>    <int> <fct>  <fct> <fct>     <fct>        <fct>     <fct>      <fct>
##  1 One_Story_… Resident…          141    31770 Pave   No_A… Slightly… Lvl          AllPub    Corner     Gtl
##  2 Two_Story_… Resident…           74    13830 Pave   No_A… Slightly… Lvl          AllPub    Inside     Gtl
##  3 Two_Story_… Resident…           78     9978 Pave   No_A… Slightly… Lvl          AllPub    Inside     Gtl
##  4 One_Story_… Resident…           43     5005 Pave   No_A… Slightly… HLS          AllPub    Inside     Gtl
##  5 One_Story_… Resident…           39     5389 Pave   No_A… Slightly… Lvl          AllPub    Inside     Gtl
## # … and many more rows and columns
## # …
```

## 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:

Variables + interactions

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

Shorthand for all predictors

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

Inline functions / transformations

```
model_fn(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 `model_fn(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 is clunky and inelegant.
- Not all modeling functions have a formula method (consistency!).

## Specifying Models Without Formulas

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

```r
# Usually, the variables must all be numeric
pre_vars <- c("Year_Sold", "Longitude", "Latitude")
model_fn(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:

```
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?

## parsnip

- A tidy unified *interface* to models
- `lm()` isn't the only way to perform linear regression
  - `glmnet` for regularized regression
  - `stan` for Bayesian regression
  - `keras` for regression using tensorflow
- But...remember the consistency slide?
  - Each interface has its own minutae to remember
  - `parsnip` standardizes all that!

## parsnip in Action

1) Create specification

2) Set the engine

3) Fit the model

```
spec_lin_reg <- linear_reg()
spec_lin_reg
```

```
## Linear Regression Model Specification (regression)
```

```
spec_lm <- set_engine(spec_lin_reg, "lm")
spec_lm
```

```
## Linear Regression Model Specification (regression)
##
## Computational engine: lm
```

```
fit_lm <- fit(
  spec_lm,
  log10(Sale_Price) ~ Longitude + Latitude,
  data = ames_train
)

fit_lm
```

```
## parsnip model object
##
##
## Call:
## stats::lm(formula = formula, data = data)
##
## Coefficients:
## (Intercept)    Longitude     Latitude
##    -316.153       -2.079        3.014
```

## Different interfaces

`parsnip` is not picky about the interface used to specify terms. Remember, `lm()` only allowed the formula interface!

```
ames_train_log <- ames_train %>%
  mutate(Sale_Price_Log = log10(Sale_Price))

fit_xy(
  spec_lm,
  y = ames_train_log$Sale_Price_Log,
  x = ames_train_log %>% select(Latitude, Longitude)
)
```

```
## parsnip model object
##
##
## Call:
## stats::lm(formula = formula, data = data)
##
## Coefficients:
## (Intercept)     Latitude    Longitude
##    -316.153        3.014       -2.079
```

## Alternative Engines

With `parsnip`, it is easy to switch to a different engine, like Stan, to run the same model with alternative backends.

```r
spec_stan <-
  spec_lin_reg %>%
  # Engine specific arguments are passed through here
  set_engine("stan", chains = 4, iter = 1000)

# Otherwise, looks exactly the same!
fit_stan <- fit(
  spec_stan,
  log10(Sale_Price) ~ Longitude + Latitude,
  data = ames_train
)
```

```r
coef(fit_stan$fit)
```

```
## (Intercept)   Longitude    Latitude
## -316.040671   -2.079419    3.007612
```

```r
coef(fit_lm$fit)
```

```
## (Intercept)   Longitude    Latitude
## -316.152846   -2.079171    3.013530
```

## Different models

Switching *between* models is easy since the interfaces are homogenous.

For example, to fit a 5-nearest neighbor model:

```r
fit_knn <-
  nearest_neighbor(mode = "regression", neighbors = 5) %>%
  set_engine("kknn") %>%
  fit(log10(Sale_Price) ~ Longitude + Latitude, data = ames_train)
fit_knn
```

```
## parsnip model object
##
##
## Call:
## kknn::train.kknn(formula = formula, data = data, ks = ~5)
##
## Type of response variable: continuous
## minimal mean absolute error: 0.06874768
## Minimal mean squared error: 0.009443873
## Best kernel: optimal
## Best k: 5
```

## DANGER

In a real scenario, we would use *resampling* methods (e.g. cross- \validation, bootstrapping, etc) or a validation set to evaluate how well the model is doing.

`tidymodels` has a great infrastructure to do this with the `rsample` package.

In this 90 min tutorial, there's not enough time to go through those methods.

In general, we would **not** want to predict the test set at this point. I'll do so to illustrate how the code to make predictions works.

## Predictions

Now, let's compute predictions and performance measures:

```
# Numeric predictions always in a df
# with column `.pred`
test_pred <- fit_lm %>%
  predict(ames_test) %>%
  bind_cols(ames_test) %>%
  mutate(log_price = log10(Sale_Price))

test_pred %>%
  dplyr::select(log_price, .pred) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 2
##   log_price .pred
##       <dbl> <dbl>
## 1      5.02  5.23
## 2      5.24  5.22
## 3      5.39  5.22
```

```
# yardstick loaded my tidymodels

# yardstick has many metrics for assessing performance.
# They can be bundled together
perf_metrics <- metric_set(rmse, rsq, ccc)

# A tidy result back:
test_pred  %>%
  perf_metrics(truth = log_price, estimate = .pred)
```

```
## # A tibble: 3 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard       0.160
## 2 rsq     standard       0.148
## 3 ccc     standard       0.282
```

For the KNN model, just change to `fit_knn`.

## Feature Engineering

## Preprocessing and Feature Engineering

This part mostly concerns what we can *do* to our variables to make the models more effective.

This is mostly related to the predictors. Operations that we might use are:

- transformations of individual predictors or groups of variables
- alternate encodings of a variable
- elimination of predictors (unsupervised)

In statistics, this is generally called *preprocessing* the data. As usual, the computer science side of modeling has a much flashier name: *feature engineering*.

## Reasons for Modifying the Data

- Some models ($K$-NN, SVMs, PLS, neural networks) require that the predictor variables have the same units. **Centering** and **scaling** the predictors can be used for this purpose.
- Other models are very sensitive to correlations between the predictors and **filters** or **PCA signal extraction** can improve the model.
- As we'll see in an example, changing the scale of the predictors using a **transformation** can lead to a big improvement.
- In other cases, the data can be **encoded** in a way that maximizes its effect on the model. Representing the date as the day of the week can be very effective for modeling public transportation data.
- Many models cannot cope with missing data so **imputation** strategies might be necessary.
- Development of new *features* that represent something important to the outcome (e.g. compute distances to public transportation, university buildings, public schools, etc.)

## Preprocessing Categorical Predictors

## Dummy Variables

One common procedure for modeling is to create numeric representations of categorical data. This is usually done via *dummy variables*: a set of binary 0/1 variables for different levels of an R factor.

For example, the Ames housing data contains a predictor called `Alley` with levels: 'Gravel', 'No_Alley_Access', 'Paved'.

Most dummy variable procedures would make *two* numeric variables from this predictor that are 1 when the observation has that level, and 0 otherwise.

| **Data** | **Dummy Variables** | |
|---|---|---|
| | **No_Alley_Access** | **Paved** |
| Gravel | 0 | 0 |
| No_Alley_Access | 1 | 0 |
| Paved | 0 | 1 |

## Dummy Variables

If there are *C* levels of the factor, only *C*-1 dummy variables are created since the last can be inferred from the others. There are different contrast schemes for creating the new variables.

For ordered factors, *polynomial* contrasts are used. See this blog post for more details.

How do you create them in R?

The formula method does this for you[1]. Otherwise, the traditional method is to use `model.matrix()` to create a matrix. However, there are some caveats to this that can make things difficult.

We'll show another method for making them shortly.

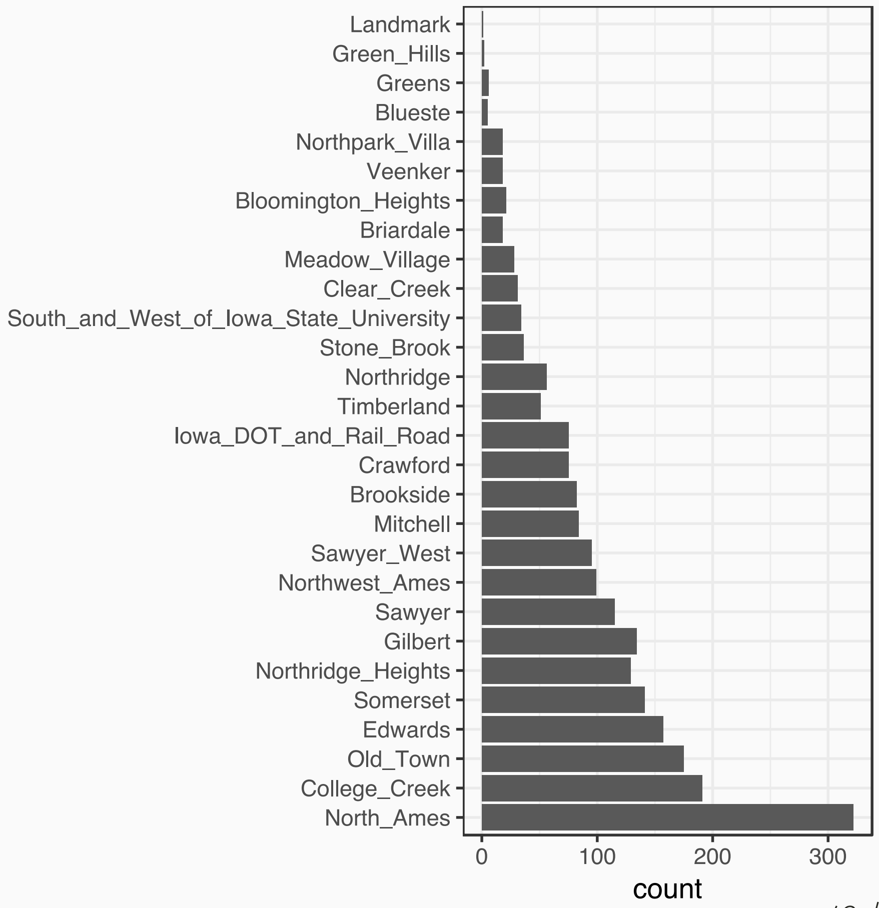
[1] *Almost always* at least. Tree- and rule-based model functions do not. Examples are `randomforest::randomForest`, `ranger::ranger`, `rpart::rpart`, `C50::C5.0`, `Cubist::cubist`, `klaR::NaiveBayes` and others.

## Infrequent Levels in Categorical Factors

One issue is: what happens when there are very few values of a level?

Consider the Ames training set and the `Neighborhood` variable.

If these data are resampled, what would happen to Landmark and similar locations when dummy variables are created?



## Infrequent Levels in Categorical Factors

A *zero-variance* predictor that has only a single value (zero) would be the result.

Many models (e.g. linear/logistic regression, etc.) would find this numerically problematic and issue a warning and `NA` values for that coefficient. Trees and similar models would not notice.

There are two main approaches to dealing with this:

- Run a filter on the training set predictors prior to running the model and remove the zero-variance predictors.

- Recode the factor so that infrequently occurring predictors (and possibly new values) are pooled into an "other" category.

However, `model.matrix()` and the formula method are incapable of doing either of these.

## Recipes

Recipes are an alternative method for creating the data frame of predictors for a model.

They allow for a sequence of *steps* that define how data should be handled.

Recall the previous part where we used the formula `log10(Sale_Price) ~ Longitude + Latitude`? These steps are:

- Assign `Sale_Price` to be the outcome
- Assign `Longitude` and `Latitude` as predictors
- Log transform the outcome

To start using a recipe, these steps can be done using

```r
# recipes loaded by tidymodels
mod_rec <- recipe(Sale_Price ~ Longitude + Latitude, data = ames_train) %>%
  step_log(Sale_Price, base = 10)
```

This creates the recipe for data processing (but does not execute it yet)

## Recipes and Categorical Predictors

To deal with the dummy variable issue, we can expand the recipe with more steps:

```r
mod_rec <- recipe(
    Sale_Price ~ Longitude + Latitude + Neighborhood,
    data = ames_train
  ) %>%
  step_log(Sale_Price, base = 10) %>%

  # Lump factor levels that occur in
  # <= 5% of data as "other"
  step_other(Neighborhood, threshold = 0.05) %>%

  # Create dummy variables for _any_ factor variables
  step_dummy(all_nominal())
```

```r
mod_rec
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          3
##
## Operations:
##
## Log transformation on Sale_Price
## Collapsing factor levels for Neighborhood
## Dummy variables from all_nominal()
```

Note that we can use standard `dplyr` selectors as well as some new ones based on the data type (`all_nominal()`) or by their role in the analysis (`all_predictors()`).

## Using Recipes

```
      recipe   -->   prepare    --> bake/juice

     (define) --> (estimate) -->   (apply)
```

## Preparing the Recipe

Now that we have a preprocessing *specification*, let's run it on the training set to *prepare* the recipe:

```r
mod_rec_trained <- prep(mod_rec, training = ames_train, verbose = TRUE)
```

```
## oper 1 step log [training]
## oper 2 step other [training]
## oper 3 step dummy [training]
## The retained training set is ~ 0.19 Mb  in memory.
```

Here, the "training" is to determine which levels to lump together and to enumerate the factor levels of the `Neighborhood` variable.

An unused option, `retain = TRUE`, that keeps the processed version of the training set around so we don't have to recompute it.

## Preparing the Recipe

```
mod_rec_trained
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          3
##
## Training data contained 2199 data points and no missing data.
##
## Operations:
##
## Log transformation on Sale_Price [trained]
## Collapsing factor levels for Neighborhood [trained]
## Dummy variables from Neighborhood [trained]
```

## Getting the Values

Once the recipe is prepared, it can be applied to any data set using `bake()`:

```
ames_test_dummies <- bake(mod_rec_trained, new_data = ames_test)
names(ames_test_dummies)
```

```
##  [1] "Sale_Price"                        "Longitude"
##  [3] "Latitude"                          "Neighborhood_College_Creek"
##  [5] "Neighborhood_Old_Town"             "Neighborhood_Edwards"
##  [7] "Neighborhood_Somerset"             "Neighborhood_Northridge_Heights"
##  [9] "Neighborhood_Gilbert"              "Neighborhood_Sawyer"
## [11] "Neighborhood_other"
```

If `retain = TRUE`, the training set does not need to be "rebaked". The `juice()` function can return the processed version of the training data.

Selectors can be used with `bake()` to only extract relevant columns and the default is `everything()`.

## How Data Are Used

Note that we have:

```
recipe(..., data = data_set)
prep(...,   training = data_set)
bake(...,   new_data = data_set)
```

In the first case, `data` is used by the `recipe` function only to determine the column names and types (e.g. factor, numeric, etc). A small subset can be passed via `head` or `slice`.

For `prep`, the `training` argument should have all of the data used to estimate parameters and other quantities.

For `bake`, `new_data` is the data that the pre-processing should be applied to.

## Hands-On: Zero-Variance Filter

Instead of using `step_other()`, take 10 minutes and research how to eliminate any zero-variance predictors using the `recipe` reference site.

Re-run the recipe with this step.

What were the results?

Do you prefer either of these approaches to the other?

## Interaction Effects

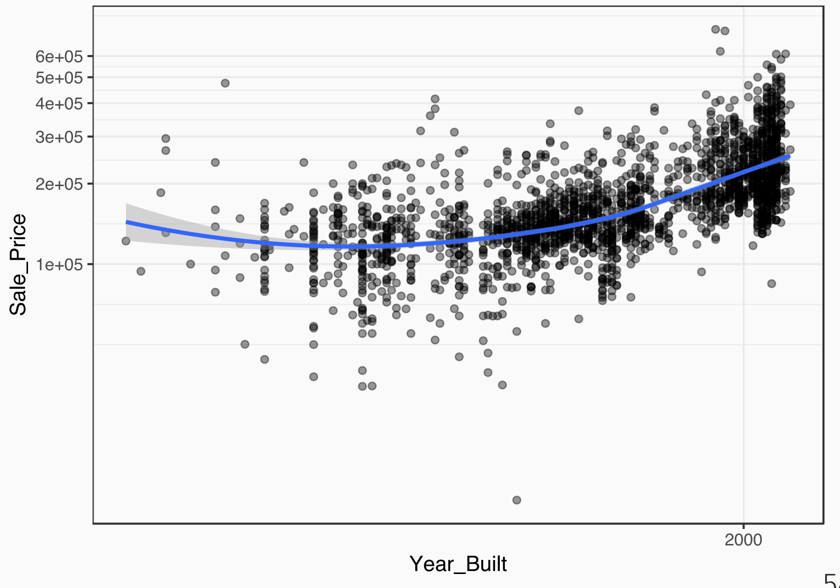
## Interactions

An **interaction** between two predictors indicates that the relationship between the predictors and the outcome cannot be describe using only one of the variables.

For example, let's look at the relationship between the price of a house and the year in which it was built. The relationship appears to be slightly nonlinear, possibly quadratic:

```
price_breaks <- (1:6)*(10^5)

ggplot(
    ames_train,
    aes(x = Year_Built, y = Sale_Price)
  ) +
  geom_point(alpha = 0.4) +
  scale_x_log10() +
  scale_y_continuous(
    breaks = price_breaks,
    trans = "log10"
  ) +
  geom_smooth(method = "loess")
```

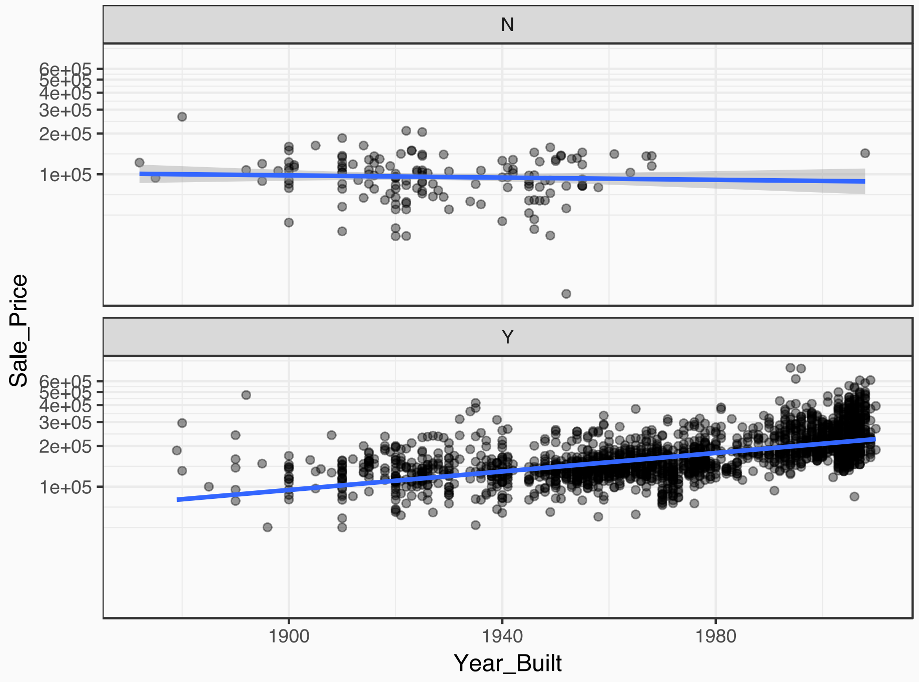


## Interactions

However... what if we seperate this trend based on whether the property has air conditioning (93.4% of the training set) or not (6.6%):

```r
library(MASS) # to get robust linear regression model

ggplot(
    ames_train,
    aes(x = Year_Built,
        y = Sale_Price)
  ) +
  geom_point(alpha = 0.4) +
  scale_y_continuous(
    breaks = price_breaks,
    trans = "log10"
  ) +
  facet_wrap(~ Central_Air, nrow = 2) +
  geom_smooth(method = "rlm")
```



## Interactions

It appears as though the relationship between the year built and the sale price is somewhat *different* for the two groups.

- When there is no AC, the trend is perhaps flat or slightly decreasing.
- With AC, there is a linear increasing trend or is perhaps slightly quadratic with some outliers at the low end.

```
mod1 <- lm(log10(Sale_Price) ~ Year_Built + Central_Air, data = ames_train)
mod2 <- lm(log10(Sale_Price) ~ Year_Built + Central_Air + Year_Built:Central_Air, data = ames_train)
anova(mod1, mod2)
```

```
## Analysis of Variance Table
##
## Model 1: log10(Sale_Price) ~ Year_Built + Central_Air
## Model 2: log10(Sale_Price) ~ Year_Built + Central_Air + Year_Built:Central_Air
##   Res.Df    RSS Df Sum of Sq      F    Pr(>F)
## 1   2196 41.349
## 2   2195 40.239  1    1.1106 60.583 1.078e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

# Interactions in Recipes

We first create the dummy variables for the qualitative predictor (`Central_Air`) then use a formula to create the interaction using the `:` operator in an additional step:

```
recipe(Sale_Price ~ Year_Built + Central_Air, data = ames_train) %>%
  step_log(Sale_Price) %>%
  step_dummy(Central_Air) %>%
  step_interact(~ starts_with("Central_Air"):Year_Built) %>%
  prep(training = ames_train) %>%
  juice() %>%
  # select a few rows with different values
  slice(153:157)
```

```
## # A tibble: 5 x 4
##   Year_Built Sale_Price Central_Air_Y Central_Air_Y_x_Year_Built
##        <int>      <dbl>         <dbl>                      <dbl>
## 1       1912       12.0             1                       1912
## 2       1930       10.9             0                          0
## 3       1900       11.8             1                       1900
## 4       1959       12.1             1                       1959
## 5       1917       11.6             0                          0
```

## Linear Models Again

Let's add a few extra predictors and some preprocessing.

- Two numeric predictors are very skewed and could use a transformation ( `Lot_Area` and `Gr_Liv_Area` ).
- We'll add neighborhood in as well and a few other house features.
- The *K*-NN model suggests that the coordinates can be helpful but probably require a nonlinear representation. We can add these using *B-splines* with 5 degrees of freedom. To evaluate this, we will create two versions of the recipe to evaluate this hypothesis.

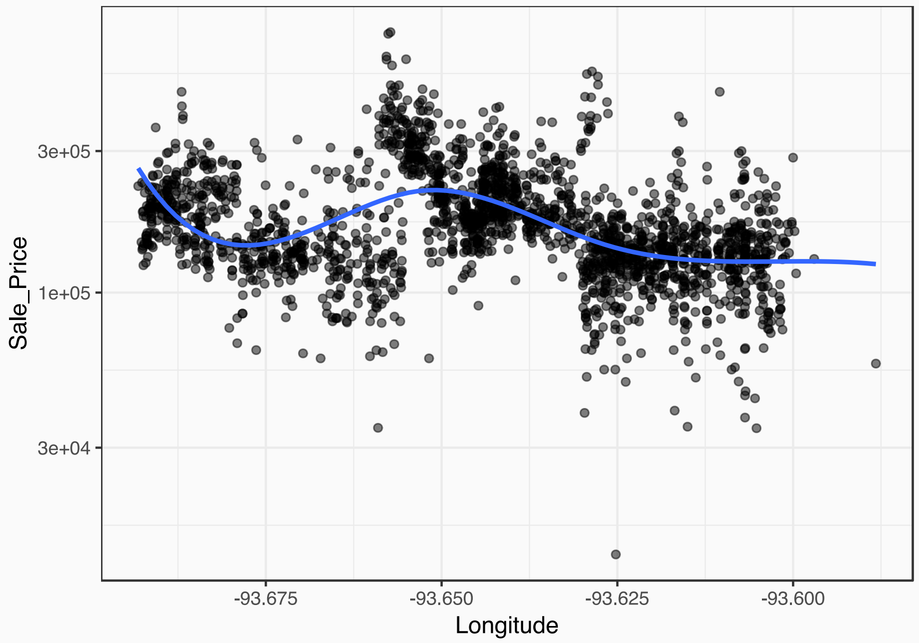
```
ames_rec <-
  recipe(Sale_Price ~ Bldg_Type + Neighborhood + Year_Built +
           Gr_Liv_Area + Full_Bath + Year_Sold + Lot_Area +
           Central_Air + Longitude + Latitude,
         data = ames_train) %>%
  step_log(Sale_Price, base = 10) %>%
  step_BoxCox(Lot_Area, Gr_Liv_Area) %>%
  step_other(Neighborhood,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

# Longitude

```r
ggplot(ames_train,
       aes(x = Longitude, y = Sale_Price)) +
  geom_point(alpha = .5) +
  geom_smooth(
    method = "lm",
    formula = y ~ splines::bs(x, 5),
    se = FALSE
  ) +
  scale_y_log10()
```

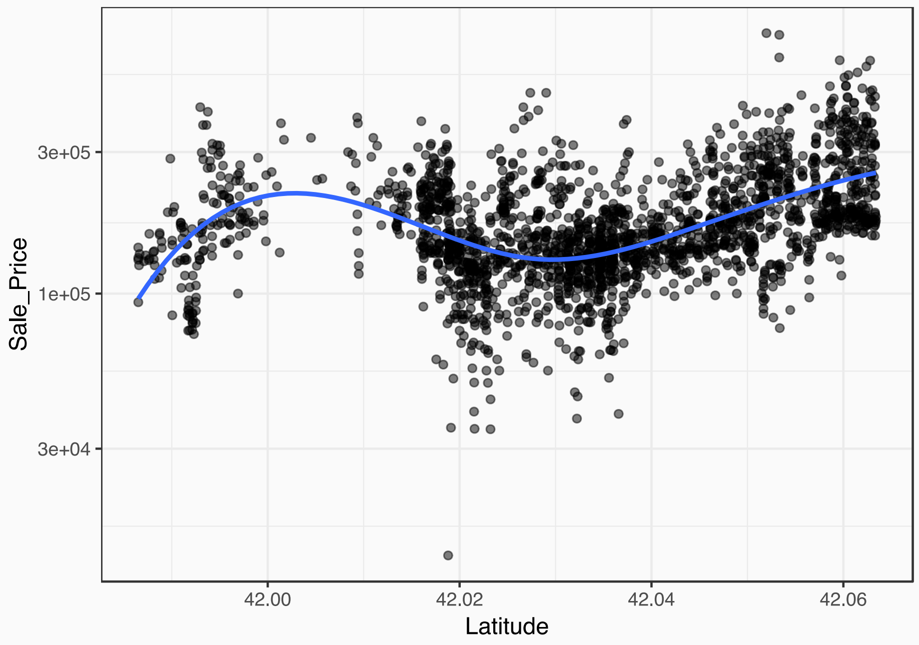
Splines add nonlinear versions of the predictor to a linear model to create smooth and flexible relationships between the predictor and outcome.

This "basis expansion" technique will be seen again in the regression section of the workshop.



# Latitude

```r
ggplot(ames_train,
       aes(x = Latitude, y = Sale_Price)) +
  geom_point(alpha = .5) +
  geom_smooth(
    method = "lm",
    formula = y ~ splines::bs(x, 5),
    se = FALSE
  ) +
  scale_y_log10()
```



## A More Complex Linear Model

```r
ames_rec <- prep(ames_rec)

train_data <- juice(ames_rec)             # Since we want the training set back so use `juice()`
test_data  <- bake(ames_rec, ames_test)  # For other data sets

lm_fit <-
  linear_reg() %>%
  set_engine("lm") %>%
  fit(Sale_Price ~ ., data = train_data)

broom::glance(lm_fit$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     0.804         0.802 0.0793      357.       0    26  2467. -4880. -4726.     13.7        2173
```

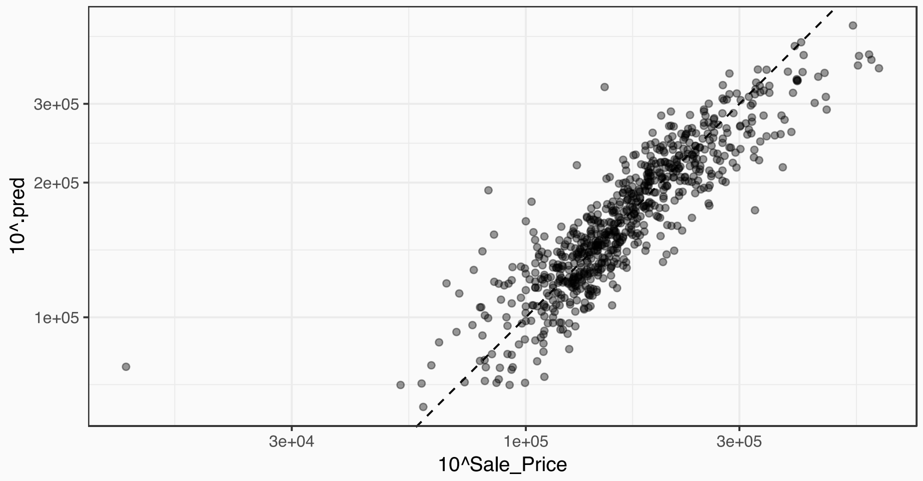
## Test Set Results

```r
test_data <-
  test_data %>%
  bind_cols(predict(lm_fit, test_data))

test_data %>%
  perf_metrics(truth = Sale_Price, estimate = .pred)
```

```
## # A tibble: 3 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard      0.0809
## 2 rsq     standard      0.782
## 3 ccc     standard      0.878
```

```r
ggplot(test_data, aes(x = 10^Sale_Price, y = 10^.pred))
  geom_point(alpha = .4) +
  geom_abline(lty = 2) +
  scale_x_log10()  + scale_y_log10() +
  coord_equal()
```



## Next Steps

- **Resampling** ( `rsample` ) methods can be used to estimate model performance before going to the test set.
- To compare performance between-models, **Bayesian analysis** ( `tidyposterior` ) can be used.
- 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` )
- 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` )