Applied Machine Learning - Feature Engineering and Preprocessing

Max Kuhn (RStudio)

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), etc.

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 an 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 or 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.)

A Bivariate Example

The plot on the right shows two predictors from a real *test* set where the object is to predict the two classes.

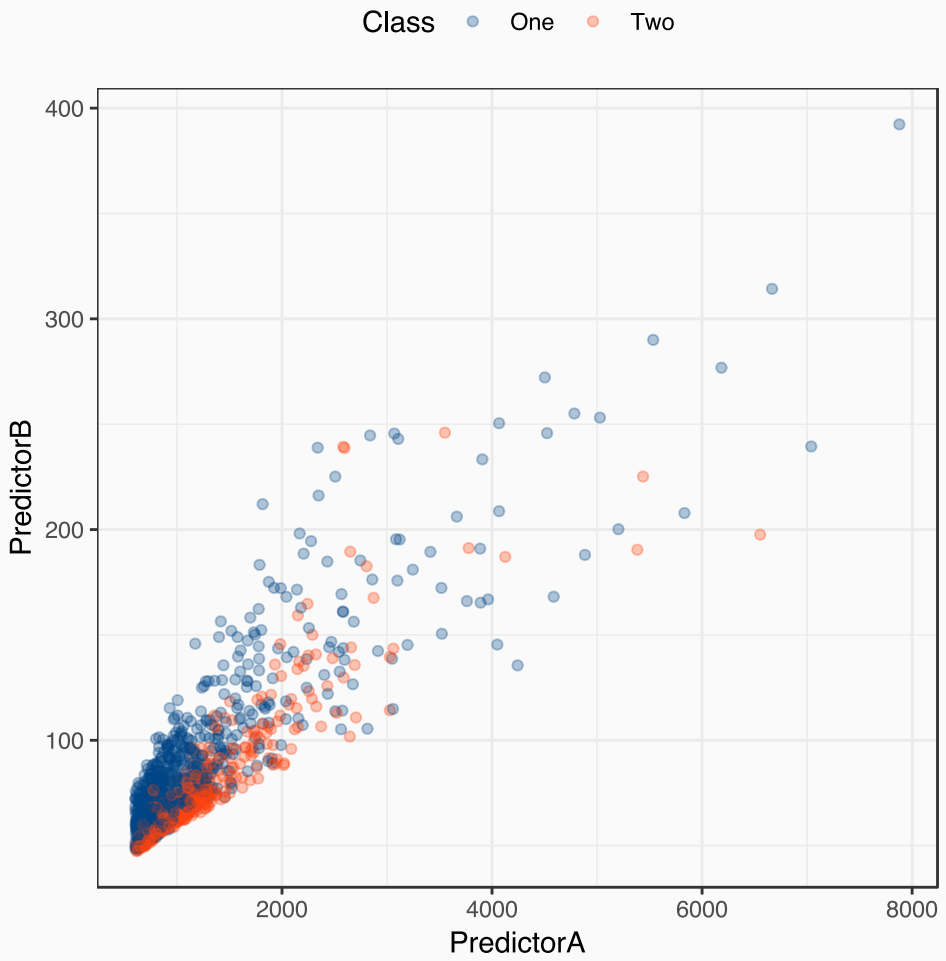
The predictors are strongly correlated and each has a right-skewed distribution.

There appears to be some class separation but only in the bivariate plot; the individual predictors show poor discrimination of the classes.

Some models might be sensitive to highly correlated and/or skewed predictors.

Is there something that we can do to make the predictors *easier for the model to use*?

Any ideas?



A Bivariate Example

We might start by estimating transformations of the predictors to resolve the skewness.

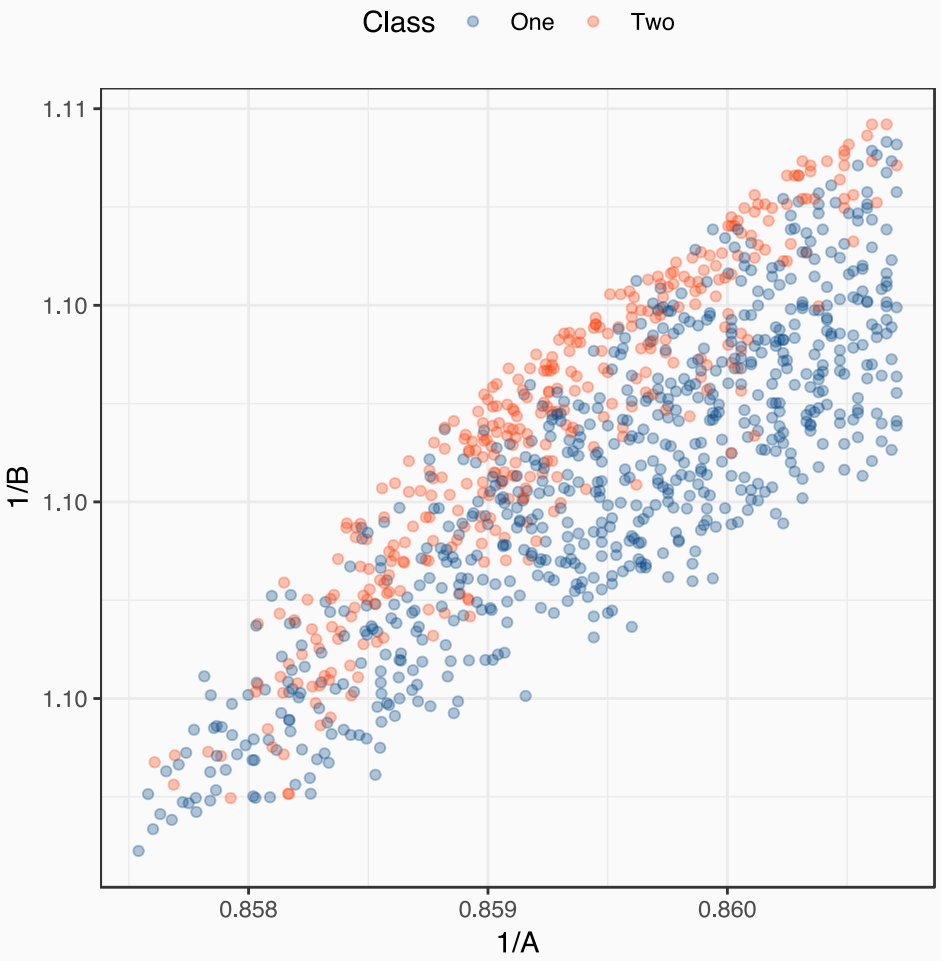
The Box-Cox transformation is a family of transformations originally designed for the outcomes of models. We can use it here for the predictors.

It uses the data to estimate a wide variety of transformations including the inverse, log, sqrt, and polynomial functions.

Using each factor in isolation, both predictors were determined to need inverse transformations (approximately).

The figure on the right shows the data after these transformations have been applied.

A logistic regression model shows a substantial improvement in classifying using the altered data.



Resampling and Preprocessing

It is important to realize that almost all preprocessing steps that involve estimation should be bundled inside of the resampling process so that the performance estimates are not biased.

- **Bad**: preprocess the data, resample the model
- **Good**: resample the preprocessing and modeling

Also:

- Avoid *information leakage* by having all operations in the modeling process occur only on the training set.
- Do not reestimate anything on the test set. For example, to center new data, the training set mean is used.

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 zero for a level and one 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 area 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 the `model.matrix` function 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.

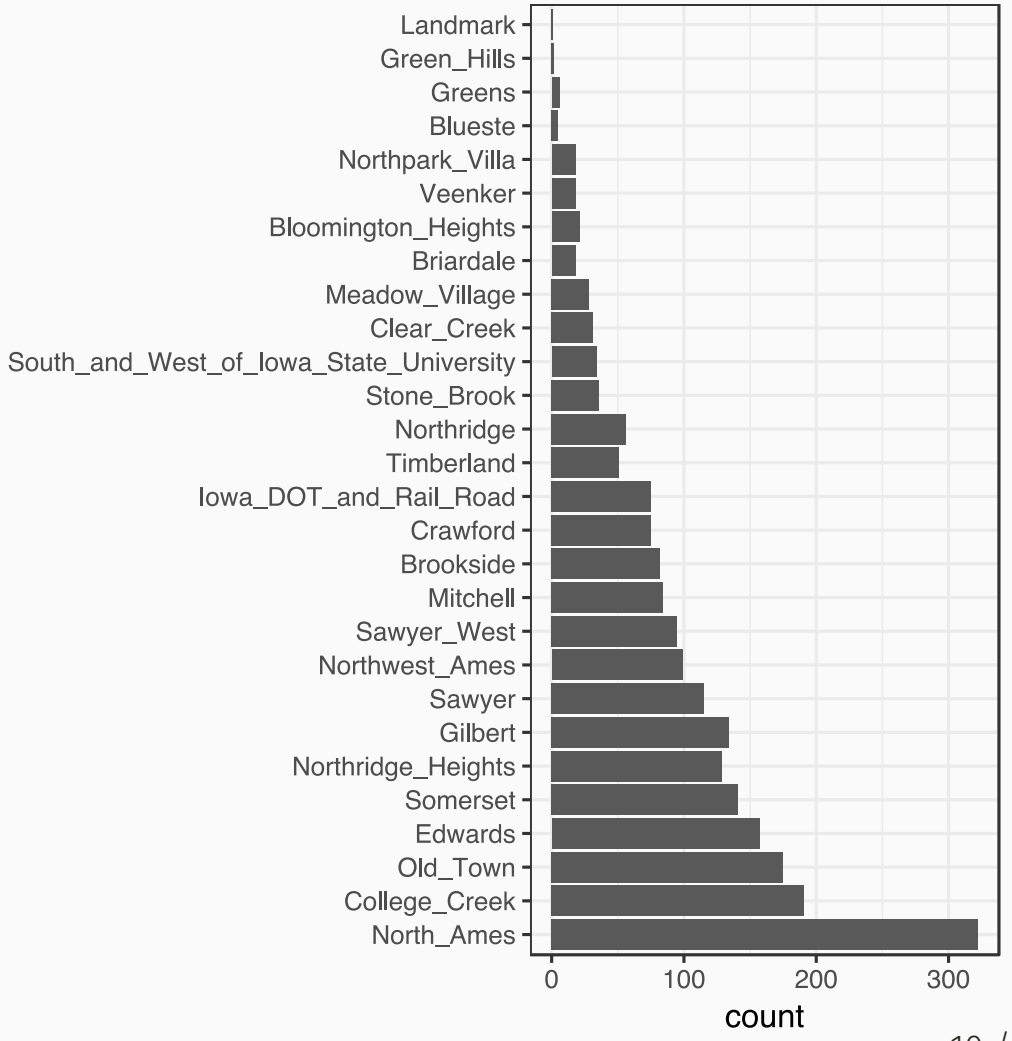
[1] *Almost always* at least. Tree- and rule-based model functions do not. Examples are `randomforest::randomForest`, `ranger::ranger`, `rpart::rpart`, `C50::C5.0`, `Cubist::cubist`, `klaR::NaiveBayes` and others.

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.

For 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.

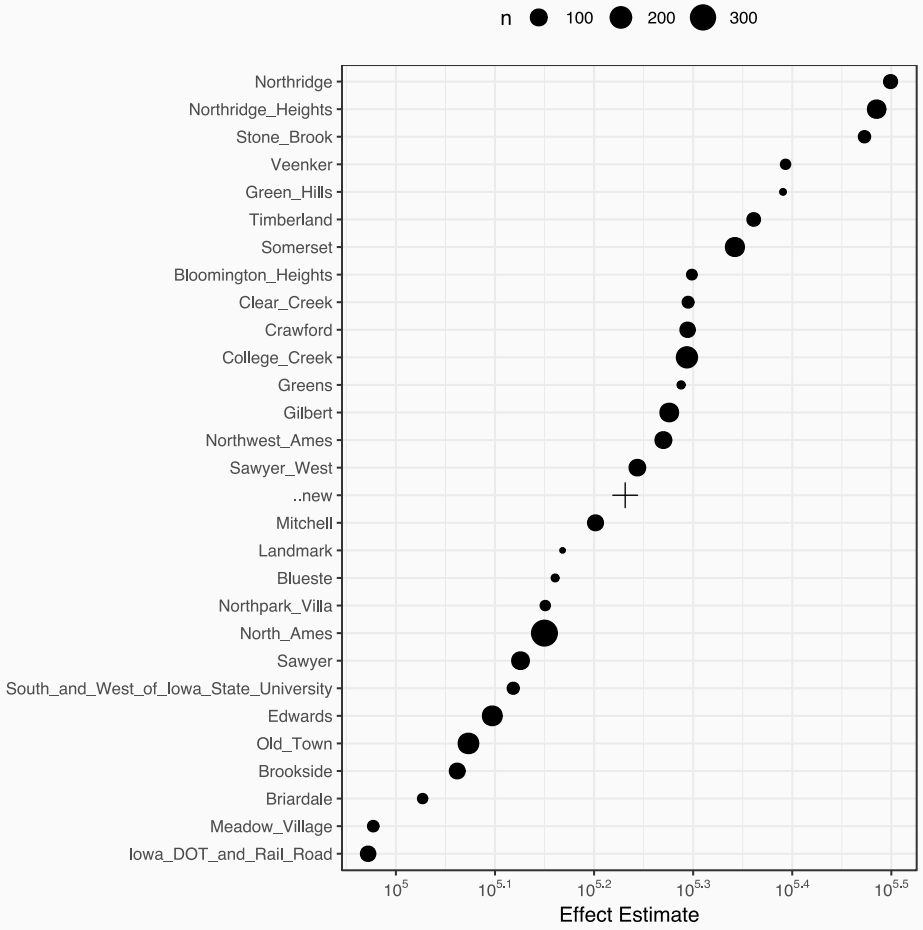
Other Approaches

A few other approaches that might help here:

- *effect* or *likelihood encodings* of categorical predictors estimate the mean effect of the outcome for every factor level in the predictor. These estimates are used in place of the factor levels. Shrinkage/regularization can improve these approaches.

- *entity embeddings* use a neural network to create features that capture the relationships between the categories and the categories and the outcome.

An addd-on to `recipes` called `embed` can be used for these encodings.



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` are predictors
- Log transform the outcome

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

```
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 Workflow

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

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

Recipes and Categorical Predictors

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

```
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())
```

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()`).

Preparing the Recipe

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

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

```
## oper 1 step log [training]
## oper 2 step other [training]
## oper 3 step dummy [training]
```

Here, the "training" is to determine which factors to pool and to enumerate the factor levels of the `Neighborhood` variable,

`retain` 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,newdata = ames_test)
names(ames_test_dummies)
```

```
##  [1] "Sale_Price"                    "Longitude"                     "Latitude"
##  [4] "Neighborhood_College_Creek"    "Neighborhood_Old_Town"         "Neighborhood_Edwards"
##  [7] "Neighborhood_Somerset"         "Neighborhood_Northridge_Heights" "Neighborhood_Gilbert"
## [10] "Neighborhood_Sawyer"           "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` and the default is `everything()`.

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

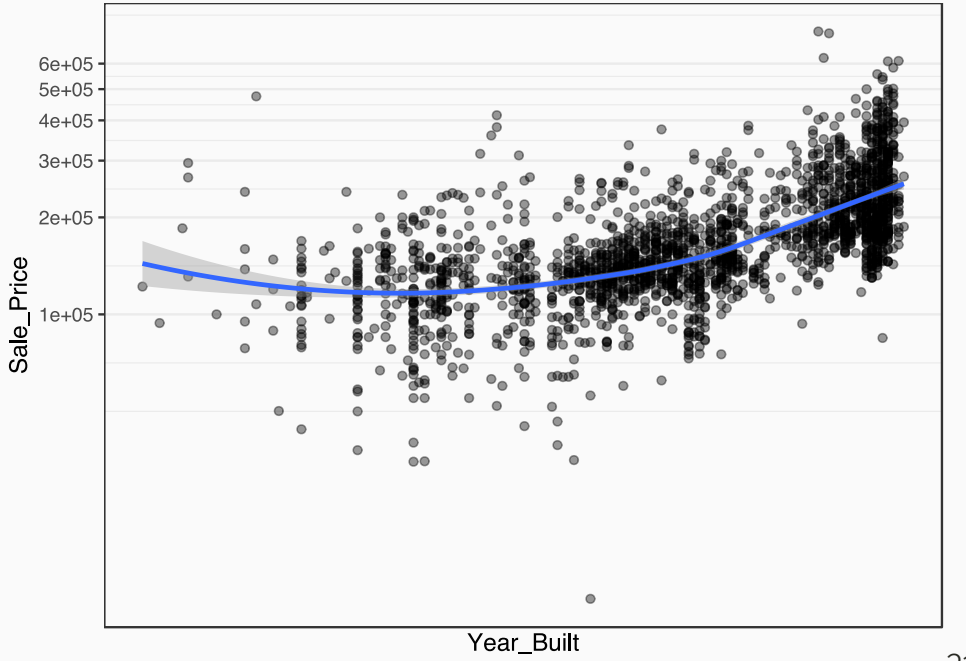
Interactions

An interactions 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")
```

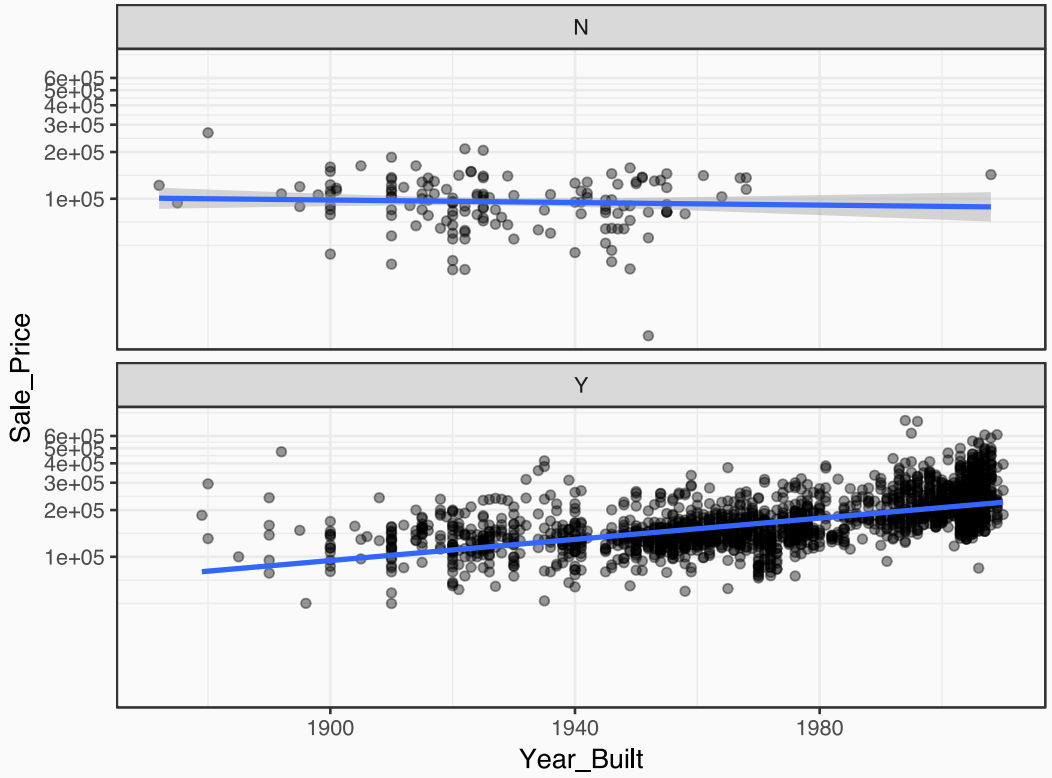


Interactions

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%):

```
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 as 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 41.3
## 2   2195 40.2  1      1.11 60.6 1.1e-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, retain = TRUE) %>%
  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
```

Adding Recipes to our `rsample` Workflows

recipes and rsample

Let's go back to the Ames housing data and work on building models with recipes using code similar to the previous set of notes. Previously:

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

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

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

Linear Models Again

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

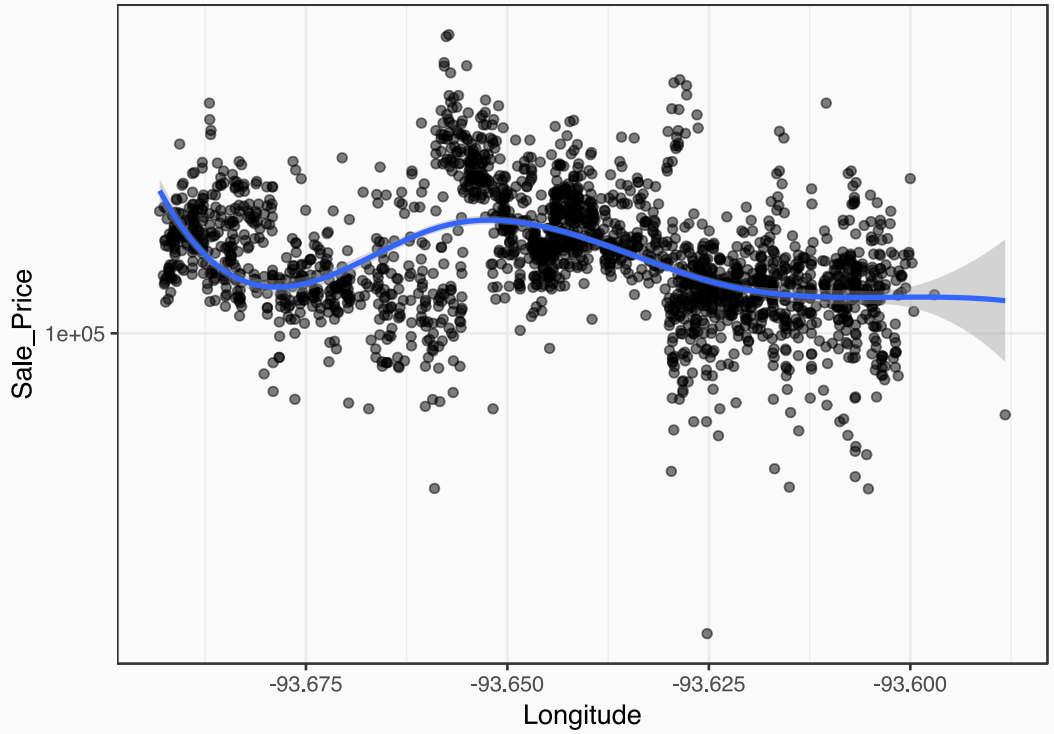
- Two numeric predictors are very skewed and could use a transformation (`Lot_Area` and `Gr_Liv_Area`).
- We'll add neighborhood in as well and a few other house features.
- The *K*-NN model suggests that the coordinates can be helpful but probably require a nonlinear representation. We can add these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

```
lin_coords <- 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_YeoJohnson(Lot_Area, Gr_Liv_Area) %>%
  step_other(Neighborhood, threshold = 0.05)  %>%
  step_dummy(all_nominal()) %>%
  step_interact(~ starts_with("Central_Air"):Year_Built)

coords <- lin_coords %>%
  step_bs(Longitude, Latitude, options = list(df = 5))
```

Longitude

```
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()
```

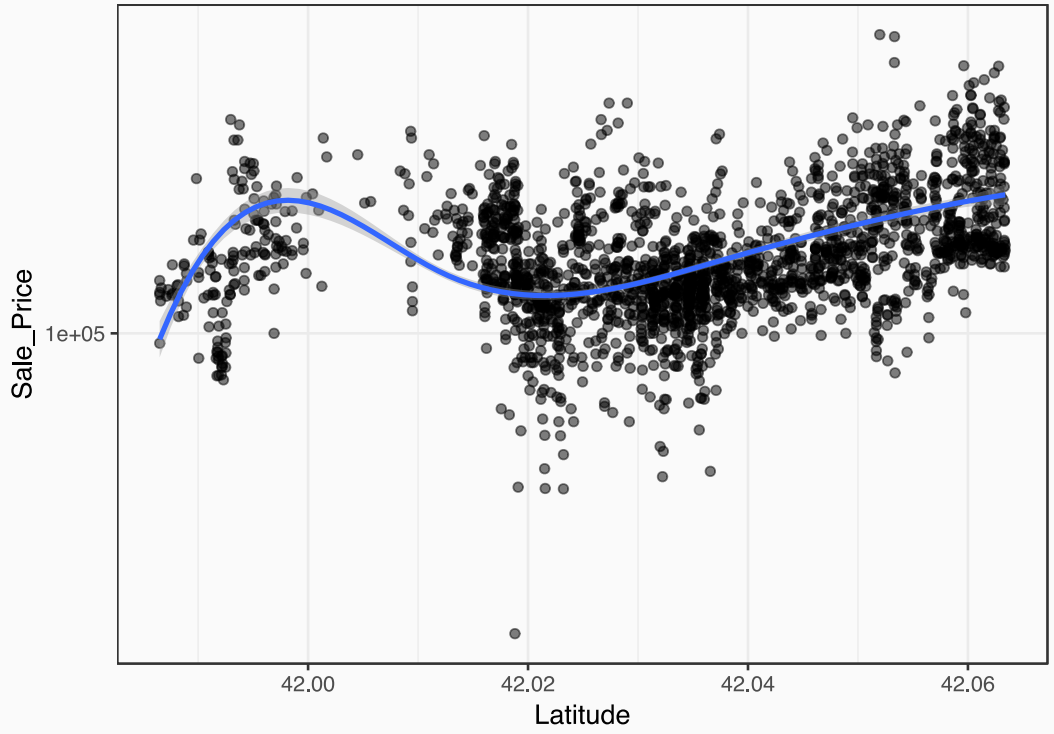


Splines add nonlinear versions of the predictor to a linear model to create smooth and flexible relationships between the predictor and outcome.

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

Latitude

```
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()
```



Preparing the Recipes

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

`rsample` has a function that is a wrapper around `prep()` that can be used to map over the split objects:

```
cv_splits <- cv_splits %>%
  mutate(coords = map(splits, prepper, recipe = coords, retain = TRUE))
```

(note: in the next versions, `prepper` is in the `recipes` package)

Fitting the Models

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

This code will use the recipe object to get the data. Since each analysis set is used to train the recipe, our previous use of `retain = TRUE` means that the processed version of the data is within the recipe.

This can be returned via the `juice` function.

```
lm_fit_rec <- function(rec_obj, ...)
  lm(..., data = juice(rec_obj))

cv_splits <- cv_splits %>%
  mutate(fits = map(coords, lm_fit_rec, Sale_Price ~ .))
glance(cv_splits$fits[[1]])
```

```
## # 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     0.805         0.802 0.0794      276.       0    30  2218. -4374. -4200.     12.3        1947
```

Predicting the Assessment Set

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

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

The function is not too bad:

```
assess_predictions <- function(split_obj, rec_obj, mod_obj) {
  raw_data <- assessment(split_obj)
  proc_x <- bake(rec_obj, newdata = raw_data, all_predictors())
  # Now save _all_ of the columns and add predictions.
  bake(rec_obj, newdata = raw_data, everything()) %>%
    mutate(
      .fitted = predict(mod_obj, newdata = proc_x),
      .resid = Sale_Price - .fitted,  # Sale_Price is already logged by the recipe
      # Save the original row number of the data
      .row = as.integer(split_obj, data = "assessment")
    )
}
```

Predicting the Assessment Set

Since we have three inputs, we will use `purrr`'s `pmap` function to walk along all three columns in the tibble.

```
cv_splits <- cv_splits %>%
  mutate(
    pred =
      pmap(
        lst(split_obj = cv_splits$splits, rec_obj = cv_splits$coords, mod_obj = cv_splits$fits),
        assess_predictions
      )
  )
```

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

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

Predicting the Assessment Set

`yardstick::metrics` will compute a small set of summary statistics for metrics model based on the type of outcome (e.g. regression, classification, etc).

```
# Compute the summary statistics
map_df(cv_splits$pred, metrics, truth = Sale_Price, estimate = .fitted) %>%
  colMeans
```

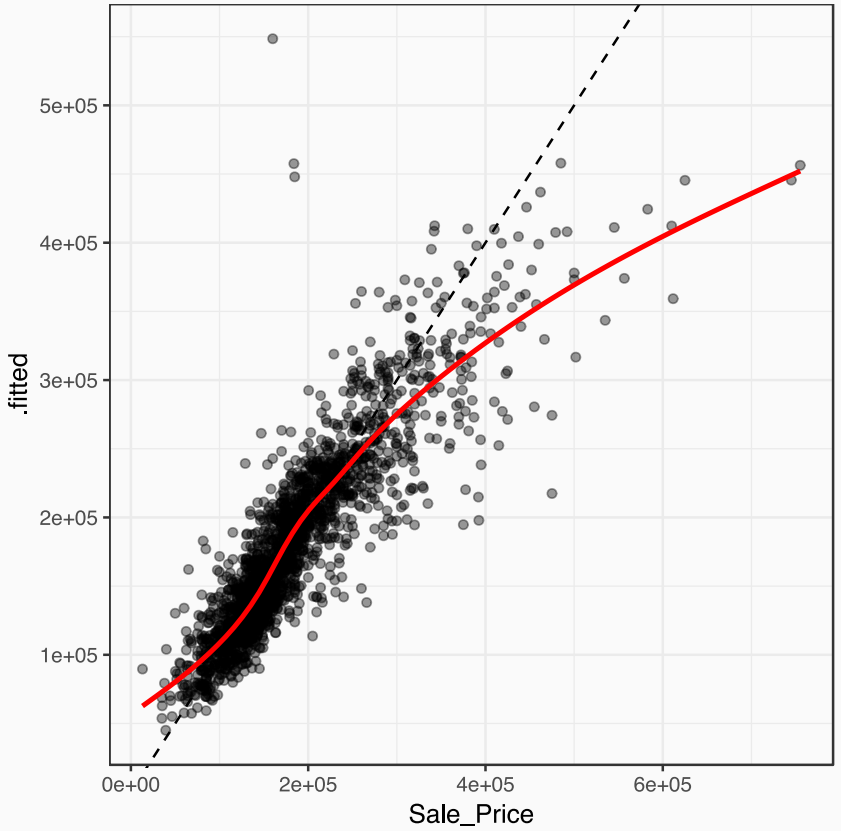
```
##   rmse    rsq
## 0.0798 0.8005
```

These results are better than our *K*-NN model but "*the only way to be comfortable with your results is to never look at them*".

Graphical Checks for Fit

```
assess_pred <- bind_rows(cv_splits$pred) %>%
  mutate(Sale_Price = 10^Sale_Price,
         .fitted = 10^.fitted)

ggplot(assess_pred,
       aes(x = Sale_Price, y = .fitted)) +
  geom_abline(lty = 2) +
  geom_point(alpha = .4)  +
  geom_smooth(se = FALSE, col = "red")
```



Graphical Checks for Fit

The current model is

- drastically *over*-predicting the price of three houses
- significantly *under*-predicting the price of a number of expensive houses.

We would we do next?

1. Try to understand the two big residuals. Are these aberrant houses or does this have something to do with our model? Maybe those extrapolation warnings?
2. Find more predictors that differentiate the more expensive houses with the others.
3. Try a different model.

(I wrote this slide long before the next one)

About Those Five Houses

From Dmytro Perepolkin via `GitHub/topepo/AmesHousing`:

> I did a little bit of research on Kaggle regarding those five houses (three "partial sale" houses in Edwards and two upscale in Northridge):
>
> > *In fact all three of Edwards properties were sold within three months by the same company. The lots are located next to each other in Cochrane Pkwy Ames, IA. I guess the story was that developer was either in trouble or wanted to boost sales, so they were signing sales contracts on half-finished houses in a new development area at a deep discount. Those few houses are likely to have been built first out of the whole residential block that followed.*
> >
> > *Two of the upscale houses in Northridge (sold at 745 and 755 thousand, respectively) are located next to each other. Houses are, of course, eyecandies (at least to my taste). They are outliers with regards to size in their own neighborhoods!*

Sometimes it really pays off to be a forensic statistician.

Coming Attactions for the `tidymodels` repo

- Hash out the principles of modeling functions. Let me know if you'd like to contribute.
- Packages on the horizon:
  - `parsnip`: a unified interface to models. This should significantly reduce the amount of syntactical minutia that you'll need to memorize by having one standardized model function across different packages and by harmonizing the parameter names across models.
  - `embed`: an add-on package for `recipes`. This can be used to efficiently encode high-cardinality categorical predictors using supervised methods such as likelihood encodings and entity embeddings.
  - A pipeline(ish) structure that can contain specifications for a model, recipe, feature filter, and post-processing. This will easily enable a *data analysis process*.
  - A model tuning package with grid search, Bayesian optimization, and other search algorithms.
  - A calibration package for post-processing regression and classification predictions as well as implementing equivocal zones.

Principles of Modeling Packages

We are in the process of developing a set of *guidelines* for making good modeling packages. For example:

- Separate the interface that the **modeler** uses from the code to do the computations. They serve two very different purposes.
- Have multiple interfaces (e.g. formula, x/y, etc).
- The *user-facing interface* should use the most appropriate data structures for the data (as opposed to the computations). For example, factor outcomes versus 0/1 indicators and data frames versus matrices.
- `type = "prob"` for class probabilities 😄.
- Use S3 methods.
- The `predict` method should give standardized, predictable results.

Rather than try to make methodologists into software developers, we will provide GitHub repositories with template packages that can be used to meet these guidelines (along with documentation and examples on *why*).