
3
Applied Machine Learning
Feature Engineering

Loading

```
library(tidymodels)
```

```
## ── Attaching packages ──────────────────────────────────────── tidymodels 0.0.4 ──
```

```
## ✓ broom     0.5.3     ✓ recipes   0.1.9
## ✓ dials     0.0.4     ✓ rsample   0.0.5
## ✓ dplyr     0.8.3     ✓ tibble    2.1.3
## ✓ ggplot2   3.2.1     ✓ tune      0.0.1
## ✓ infer     0.5.1     ✓ workflows 0.1.0
## ✓ parsnip   0.0.5     ✓ yardstick 0.0.5
## ✓ purrr     0.3.3
```

```
## ── Conflicts ─────────────────────────────────────────── tidymodels_conflicts() ──
## x purrr::discard()    masks scales::discard()
## x dplyr::filter()     masks stats::filter()
## x dplyr::lag()        masks stats::lag()
## x ggplot2::margin()   masks dials::margin()
## x recipes::step()     masks stats::step()
## x recipes::yj_trans() masks scales::yj_trans()
```

Previously

```
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)

lm_mod <- linear_reg() %>%
  set_engine("lm")

perf_metrics <- metric_set(rmse, rsq, ccc)
```

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.

Reasons for Modifying the 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.

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.

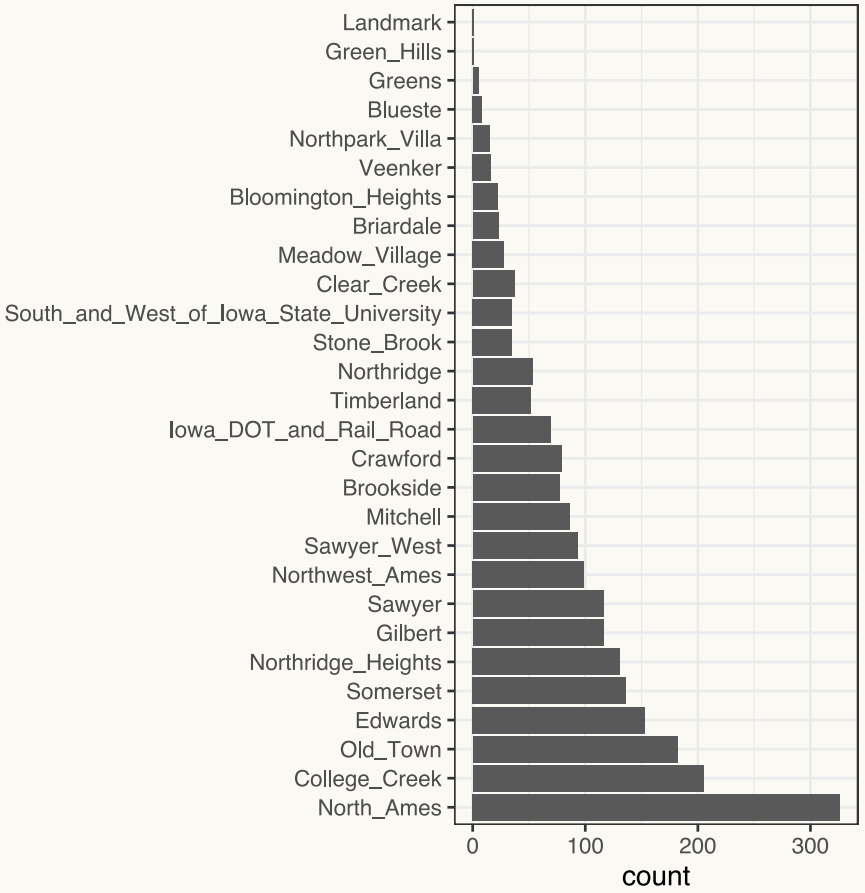
[1] *Almost always* at least. Tree- and rule-based model functions do not. Examples are `randomforest`, `ranger`, `rpart`, `C50`, `Cubist`, `klaR::NaiveBayes` and others.

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 helping you.

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

```
# recipes loaded by tidymodels
mod_rec <- recipe(Sale_Price ~ Longitude + Latitude, 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:

```
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())
```

```
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()` → `prep()` → `bake()` `juice()`

Defines the pre-processing

Calculates statistics from the training set

Applies the pre-processing to data sets

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, 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.

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 - Training

Now that the recipe has been prepared, we can extract the processed training set from it, with all of the steps applied. To do that, we use `juice()`.

```
# Extracts processed version of `ames_train`
juice(mod_rec_trained)
```

```
## # A tibble: 2,199 x 11
##    Longitude Latitude Sale_Price Neighborhood_Co… Neighborhood_Ol… Neighborhood_Ed… Neighborhood_So…
##        <dbl>    <dbl>      <dbl>             <dbl>             <dbl>             <dbl>             <dbl>
##  1     -93.6     42.1       5.24                 0                 0                 0                 0
##  2     -93.6     42.1       5.39                 0                 0                 0                 0
##  3     -93.6     42.1       5.28                 0                 0                 0                 0
##  4     -93.6     42.1       5.29                 0                 0                 0                 0
##  5     -93.6     42.1       5.33                 0                 0                 0                 0
##  6     -93.6     42.1       5.28                 0                 0                 0                 0
##  7     -93.6     42.1       5.37                 0                 0                 0                 0
##  8     -93.6     42.1       5.28                 0                 0                 0                 0
##  9     -93.6     42.1       5.25                 0                 0                 0                 0
## 10     -93.6     42.1       5.26                 0                 0                 0                 0
## # … with 2,189 more rows, and 4 more variables: Neighborhood_Northridge_Heights <dbl>,
## #   Neighborhood_Gilbert <dbl>, Neighborhood_Sawyer <dbl>, Neighborhood_other <dbl>
```

This is what you'd pass on to `fit()` your model.

Getting the Values - Testing

After model fitting, you'll eventually want to make predictions on *new data*. But first, you have to reapply all of the pre-processing steps on it. To do that, use `bake()`.

```
bake(mod_rec_trained, new_data = ames_test)
```

```
## # A tibble: 731 x 11
##    Longitude Latitude Sale_Price Neighborhood_Co… Neighborhood_Ol… Neighborhood_Ed… Neighborhood_So…
##        <dbl>    <dbl>      <dbl>             <dbl>             <dbl>             <dbl>             <dbl>
##  1     -93.6     42.1       5.33                 0                 0                 0                 0
##  2     -93.6     42.1       5.02                 0                 0                 0                 0
##  3     -93.6     42.1       5.27                 0                 0                 0                 0
##  4     -93.6     42.1       5.60                 0                 0                 0                 0
##  5     -93.6     42.1       5.28                 0                 0                 0                 0
##  6     -93.6     42.1       5.17                 0                 0                 0                 0
##  7     -93.6     42.1       5.02                 0                 0                 0                 0
##  8     -93.7     42.1       5.46                 0                 0                 0                 0
##  9     -93.7     42.1       5.44                 0                 0                 0                 0
## 10     -93.7     42.1       5.33                 0                 0                 0                 0
## # … with 721 more rows, and 4 more variables: Neighborhood_Northridge_Heights <dbl>,
## #   Neighborhood_Gilbert <dbl>, Neighborhood_Sawyer <dbl>, Neighborhood_other <dbl>
```

This is what you'd pass on to `predict()`.

`juice()` is used to get the pre-processed training set (basically for free)

`bake()` is used to pre-process a *new* data set

How Data Are Used

Note that we have:

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

- `recipe()` - `data` is used *only* to determine column names and types. A 0-row data frame could even be used.
- `prep()` - `training` is the entire training set, used to estimate parameters in each step (like means or standard deviations).
- `bake()` - `new_data` is data to apply the pre-processing to, using the *same estimated parameters* from when `prep()` was called on the training set.

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?

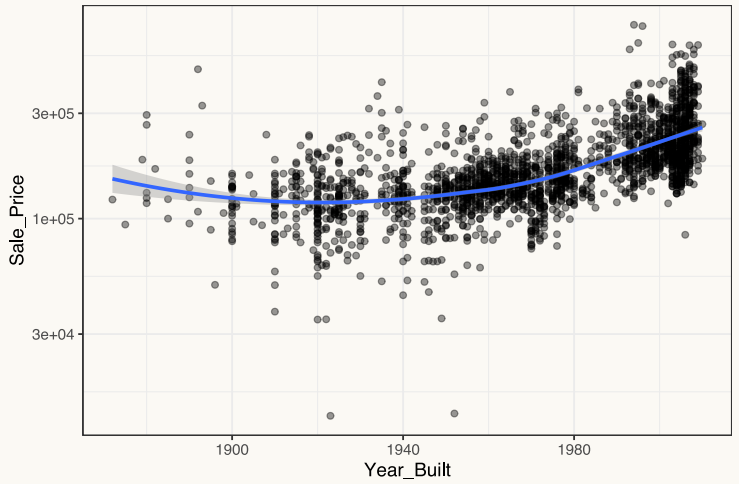
Interaction Effects

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)

ames_train %>%
  ggplot(aes(x = Year_Built, y = Sale_Price)) +
  geom_point(alpha = 0.4) +
  scale_y_log10() +
  geom_smooth(method = "loess")
```



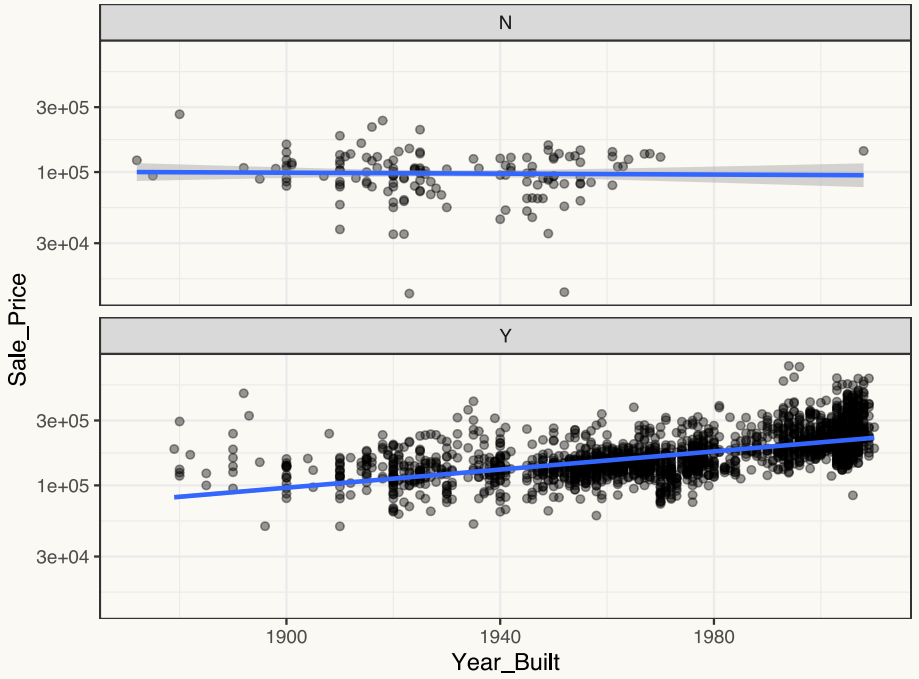
Interactions

However... what if we separate this trend based on whether the property has air conditioning or not.

```
ames_train %>%
  group_by(Central_Air) %>%
  summarise(n = n()) %>%
  mutate(percent = n / sum(n) * 100)
```

```
## # A tibble: 2 x 3
##   Central_Air     n percent
##   <fct>       <int>   <dbl>
## 1 N             141    6.41
## 2 Y            2058   93.6
```

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

ames_train %>%
  ggplot(aes(x = Year_Built, y = Sale_Price)) +
  geom_point(alpha = 0.4) +
  scale_y_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 42.741
## 2   2195 41.733  1    1.0075 52.993 4.64e-13 ***
## ---
## 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:

```
interact_rec <- 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)

interact_rec %>%
  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       1915       11.9             1                       1915
## 2       1912       12.0             1                       1912
## 3       1920       11.7             1                       1920
## 4       1963       11.6             0                          0
## 5       1930       10.9             0                          0
```

Principal Component Analysis

A Bivariate Example

The plot on the right shows two **predictors** from a real *test* set where the objective 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.

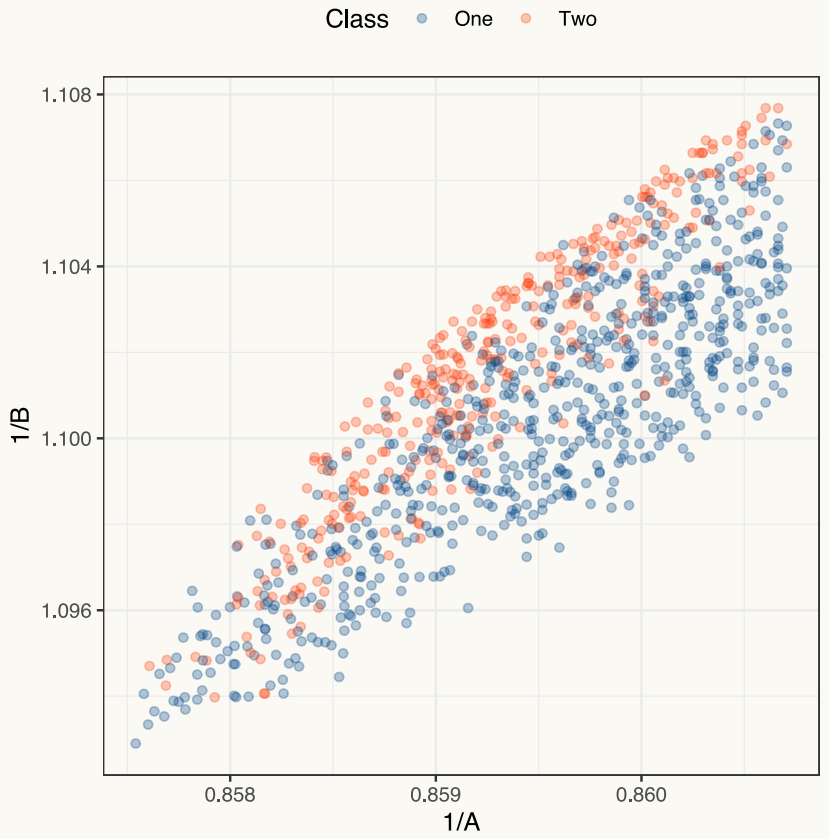
The Box-Cox transformation is a family of transformations originally designed for the outcomes of models. We can use it here for the predictors.

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.



More Recipe Steps

The package has a rich set of steps that can be used including transformations, filters, variable creation and removal, dimension reduction procedures, imputation, and others.

There are also packages like `embed`, `textrecipes`, and `themis` that extend recipes with new steps.

For example, in the previous bivariate data problem, the Box-Cox transformation was conducted using:

```
bivariate_rec <- recipe(Class ~ ., data = bivariate_data_train) %>%
  step_BoxCox(all_predictors())

bivariate_rec <- prep(bivariate_rec, training = bivariate_data_train, verbose = FALSE)

inverse_test_data <- bake(bivariate_rec, new_data = bivariate_data_test)
```

Correlated Predictors

In the Ames data, there are potential clusters of *highly correlated variables*:

- proxies for size: `Lot_Area`, `Gr_Liv_Area`, `First_Flr_SF`, `Bsmt_Unf_SF`, `Full_Bath` etc.
- quality fields: `Overall_Qual`, `Garage_Qual`, `Kitchen_Qual`, `Exter_Qual`, etc.

It would be nice if we could combine/amalgamate the variables in these clusters into a single variable that represents them.

Another way of putting this is that we would like to create artificial features of the data that account for a certain amount of *variation* in the data.

There are a few different methods that can accomplish this; we will focus on principal component analysis (PCA). Another, regularization, will be discussed later.

PCA Signal Extraction

Principal component analysis (PCA) is a multivariate statistical technique that can be used to create artificial new variables from an existing set.

Conceptually, PCA determines which variables account for the most correlation in the data and creates a new variable that is a linear combination of all the predictors.

- This is called the *first principal component* (aka `PC1`).
- This linear combination emphasizes the variables that are the most correlated.

The variables constructing `PC1` are then *removed from the data*.

The second PCA component is the linear combination that accounts for the most left-over correlation in the data (and so on).

PCA Signal Extraction

The main takeaways:

- The components account for as much as the variation in the original data as possible.
- Each component is uncorrelated with the others.
- The new variables are *linear combinations* of all of the input variables and are effectively unitless (It is generally a good idea to center and scale your predictors because of this).

For our purposes, we would use PCA on the *predictors* to:

- Reduce the number of variables exposed to the model (but this is not feature selection).
- Combat excessive correlations between the predictors (aka multicollinearity).

In this way, the procedure is often called *signal extraction* but this is poorly named since there is no guarantee that the new variables will have an association with the outcome.

Back to the Bivariate Example - Transformed Data

Back to the Bivariate Example - Recipes

We can build on our transformed data recipe and add normalization:

```
bivariate_pca <-
  recipe(Class ~ PredictorA + PredictorB, data = bivariate_data_train) %>%
  step_BoxCox(all_predictors()) %>%
  step_normalize(all_predictors()) %>% # center and scale
  step_pca(all_predictors()) %>%
  prep(training = bivariate_data_train)

pca_test <- bake(bivariate_pca, new_data = bivariate_data_test)

# Put components axes on the same range
pca_rng <- extendrange(c(pca_test$PC1, pca_test$PC2))

pca_test %>%
  ggplot(aes(x = PC1, y = PC2, color = Class)) +
  geom_point(alpha = .2, cex = 1.5) +
  theme(legend.position = "top") +
  scale_colour_calc() +
  xlim(pca_rng) + ylim(pca_rng) +
  xlab("Principal Component 1") +
  ylab("Principal Component 2")
```

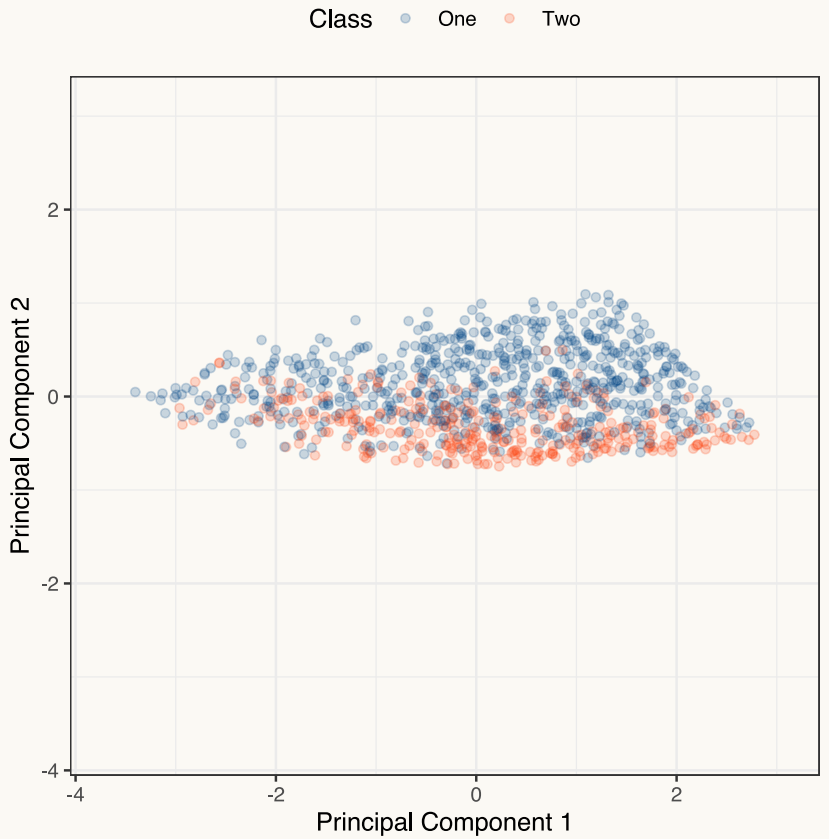
Back to the Bivariate Example

Recall that even after the Box-Cox transformation was applied to our previous example, there was still a high degree of correlation between the predictors.

After the transformation, the predictors were centered and scaled, then PCA was conducted. The plot on the right shows the results.

Since these two predictors are highly correlated, the first component captures 91.7% of the variation in the original data. However...

...recall that PCA does not guarantee that the components are associated with the outcome. In this example, the *least important* component has the association with the outcome.



PCA does a *rotation* of the data so that the *variation* in one dimension is maximized.

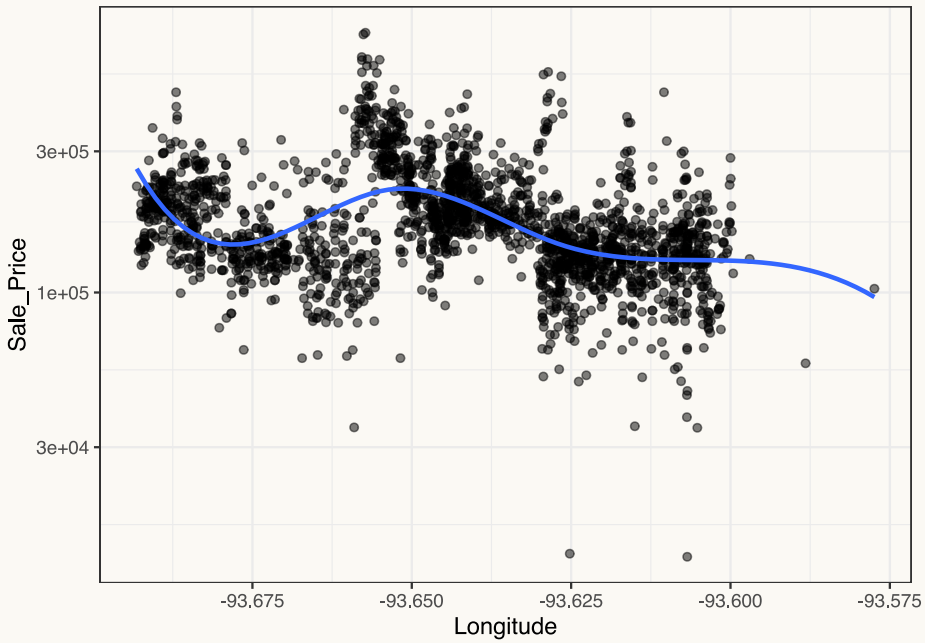
The rotation also makes the new variables *uncorrelated*.


Class
One
Two
New Component #2
2
0
-2
-4
-2
0
2
New Component #1

Recipe and Models

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()
```

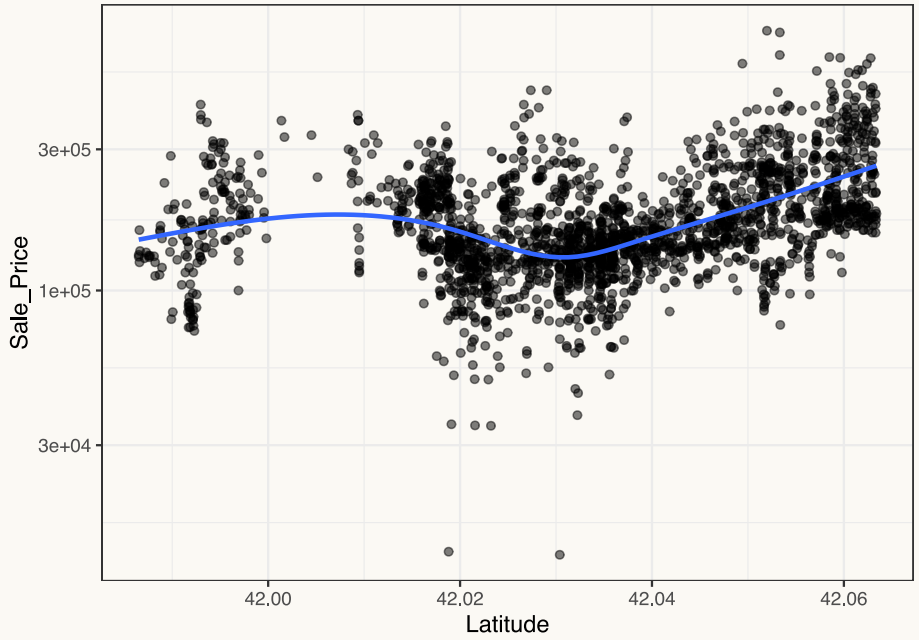


Splines add nonlinear versions of the predictor to a linear model to create smooth and flexible relationships between the predictor and outcome.

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::ns(x, df = 5),
    se = FALSE
  ) +
  scale_y_log10()
```



Linear Models Again

- We'll add neighborhood in as well and a few other house features.
- Our plots 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.
- Two numeric predictors are very skewed and could use a transformation (`Lot_Area` and `Gr_Liv_Area`).

```
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_ns(Longitude, Latitude, deg_free = 5)
```

Combining the Recipe with a Model

- `prep()` - `juice()` - `fit()`

```
ames_rec <- prep(ames_rec)

lm_fit <-
  lm_mod %>%
  fit(Sale_Price ~ ., data = juice(ames_rec))   # The recipe puts Sale_Price on the log scale

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     0.802         0.799 0.0800      303.       0    30  2448. -4834. -4657.     13.9        2169
```

- `bake()` - `predict()`

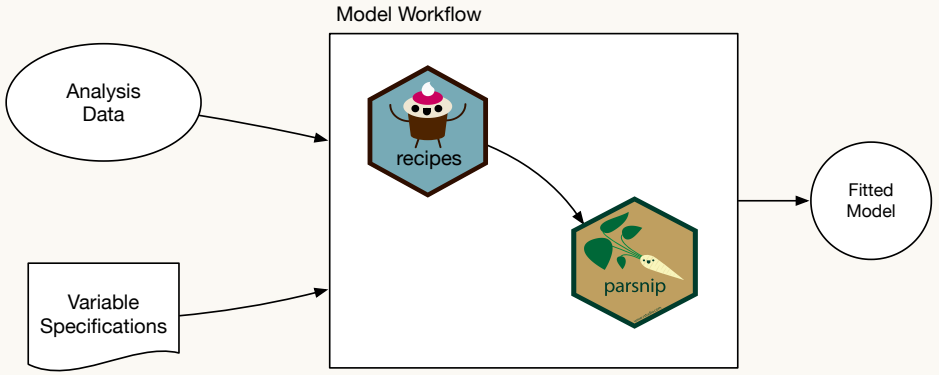
```
ames_test_processed <- bake(ames_rec, ames_test, all_predictors())

# but let's not do this
# predict(lm_fit, new_data = ames_test_processed)
```

That Modeling Process Again

We are

- estimating transformations on some variables,
- making nonlinear features for the geocoding variables, and
- estimating new encodings for some factor variables.

These should definitely be included in our overall modeling process.



Workflows

The `workflows` package enables a handy type of object that can bundle pre-processing and models together.

- You don't have to keep track of separate objects in your workspace.
- The recipe prepping and model fitting can be executed using a single call to `fit()` instead of `prep()`-`juice()`-`fit()`.
- The recipe baking and model predictions are handled with a single call to `predict()` instead of `bake()`-`predict()`.
- Workflows *will* be able to add post-processing operations in upcoming versions. An example of post-processing would be to modify (and tune) the probability cutoff for two-class models.

Workflows


```
ames_wfl <- workflow() %>%
  add_recipe(ames_rec) %>%
  add_model(lm_mod)

ames_wfl
```

```
## == Workflow ====================================================================
## Preprocessor: Recipe
## Model: linear_reg()
##
## -- Preprocessor ----------------------------------------------------------------
## 6 Recipe Steps
##
## ● step_log()
## ● step_BoxCox()
## ● step_other()
## ● step_dummy()
## ● step_interact()
## ● step_ns()
##
## -- Model -----------------------------------------------------------------------
## Linear Regression Model Specification (regression)
##
## Computational engine: lm
```

1-Step fitting and predicting


```
# preps() `ames_train`, juice()s it, then fit()s model
ames_wfl_fit <- fit(ames_wfl, ames_train)

# bake()s `ames_test`, then predict()s
predict(ames_wfl_fit, ames_test) %>% slice(1:5)
```

```
## # A tibble: 5 x 1
##   .pred
##   <dbl>
## 1  5.32
## 2  5.10
## 3  5.14
## 4  5.42
## 5  5.32
```

- `add_formula()` can be used as a simple alternative to `add_recipe()`.
- A *secondary formula* can be given to `add_model()` that will be directly passed to the model. This is helpful when recipes are used with GAMs or mixed-models.
- There are `remove_*()` and `update_*()` functions too.

A Quick Knowledge Check - Match tasks to packages

- Fit a K-NN model
- Extract holidays from dates
- Make a training/test split
- Bundle a recipe and model
- Is high in vitamin A
- Compute R^2
- Bin a predictor (but seriously,...don't)

- `workflows`
- `yardstick`
- `carrot`
- `recipes`
- `parsnip`
- `rsample`
- `ggvis`