Tidymodels

Nate Wells

Math 243: Stat Learning

December 2nd, 2020

Outline

In today's class, we will. . .

- Discuss the `tidymodels` packages for model building in the `tidyverse` framework

Section 1

Intro to tidymodels

Why `tidymodels`?

Suppose we plan to classify data with a binary response variable. Several models are available:

Why `tidymodels`?

Suppose we plan to classify data with a binary response variable. Several models are available:

Function	Package	Code
`lda`	MASS	`predict(object)`
`glm`	stats	`predict(object, type = "response")`
`gbm`	gbm	`predict(object, type = "response", n.trees)`
`tree`	treet	`predict(object, type = "prob")`
`M5P`	RWeka	`predict(object, type = "probability")`
`knn`	class	`---`

Why tidymodels?

Suppose we plan to classify data with a binary response variable. Several models are available:

Function	Package	Code
`lda`	MASS	`predict(object)`
`glm`	stats	`predict(object, type = "response")`
`gbm`	gbm	`predict(object, type = "response", n.trees)`
`tree`	treet	`predict(object, type = "prob")`
`M5P`	RWeka	`predict(object, type = "probability")`
`knn`	class	`---`

Each method has significantly different methods for making class probability predictions

Why `tidymodels`?

Suppose we plan to classify data with a binary response variable. Several models are available:

Function	Package	Code
`lda`	MASS	`predict(object)`
`glm`	stats	`predict(object, type = "response")`
`gbm`	gbm	`predict(object, type = "response", n.trees)`
`tree`	treet	`predict(object, type = "prob")`
`M5P`	RWeka	`predict(object, type = "probability")`
`knn`	class	`---`

Each method has significantly different methods for making class probability predictions

Additionally, each model takes in different types of data arguments (vectors, model matrices, data frames, model formulas)

`tidymodels` goals

Broadly, `tidymodels` presents collection of modeling packages that share design philosophy, syntax and data structure to make it easy to move between pacakges.

tidymodels goals

Broadly, `tidymodels` presents collection of modeling packages that share design philosophy, syntax and data structure to make it easy to move between pacakges.

Additionally, `tidymodels` fits in the broader `tidyverse` framework:

- Packages and functions should be accessible and easily interpreted
- Outputs should be data frames (or tibbles) whenever possible
- Functions should be compatible with the %>% operator and functional programming
- Model objects should be compatible with `ggplot2`

tidymodels goals

Broadly, `tidymodels` presents collection of modeling packages that share design philosophy, syntax and data structure to make it easy to move between pacakges.

Additionally, `tidymodels` fits in the broader `tidyverse` framework:

- Packages and functions should be accessible and easily interpreted
- Outputs should be data frames (or tibbles) whenever possible
- Functions should be compatible with the %>% operator and functional programming
- Model objects should be compatible with `ggplot2`

`tidymodels` takes the mechanics from each individual model package (`mass`, `tree`, `glm` etc.) and unifies the input and output

The `tidymodel` framework

1. Preprocess data using the `recipes` package
2. Create training-test data splits using the `rsample` package
3. Give a model a functional form and specify fitting method using the `parsnip` package
4. Fit the model, tidy the results, and make predictions using the `fit`, `tidy`, and `predict` functions
5. Estimate model performance using cross-validation from the `rsample` package
6. Tune model parameters by adding model specifications

The `tidymodel` framework

1. Preprocess data using the `recipes` package
2. Create training-test data splits using the `rsample` package
3. Give a model a functional form and specify fitting method using the `parsnip` package
4. Fit the model, tidy the results, and make predictions using the `fit`, `tidy`, and `predict` functions
5. Estimate model performance using cross-validation from the `rsample` package
6. Tune model parameters by adding model specifications

We'll investigate each of these in-depth (although slightly out of order)

Section 2

Build a Model

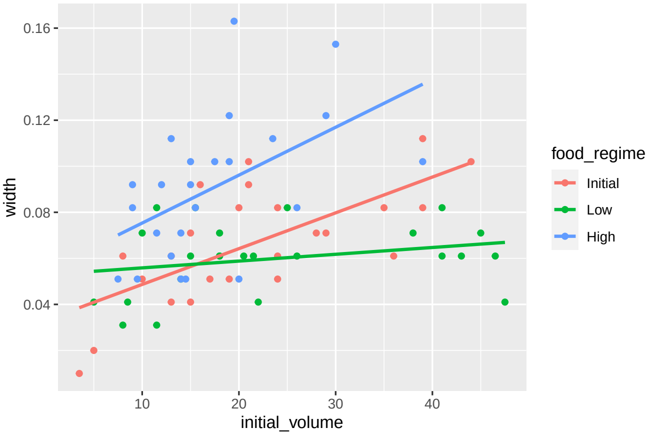
The Data

The sea_urchins data set explores the relationship between feeding regimes and size of sea urchins over time:

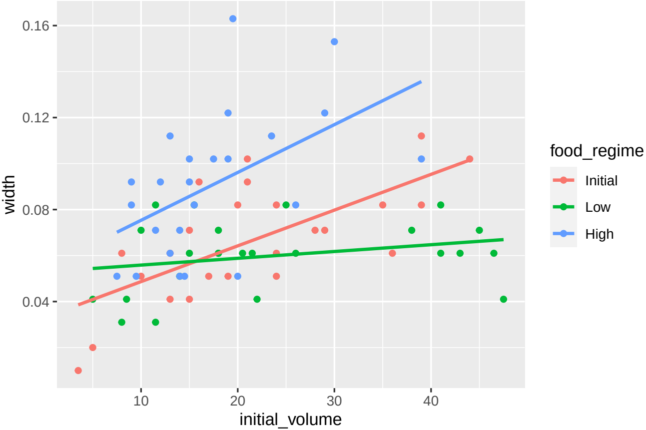
```
sea_urchins<-read_csv("https://tidymodels.org/start/models/urchins.csv") %>%
  setNames(c("food_regime", "initial_volume", "width")) %>%
  mutate(food_regime = factor(food_regime, levels = c("Initial", "Low", "High")))
head(sea_urchins)
```

```
## # A tibble: 6 x 3
##   food_regime initial_volume width
##   <fct>                <dbl> <dbl>
## 1 Initial                3.5 0.01
## 2 Initial                5   0.02
## 3 Initial                8   0.061
## 4 Initial               10   0.051
## 5 Initial               13   0.041
## 6 Initial               13   0.061
```

Scatterplot



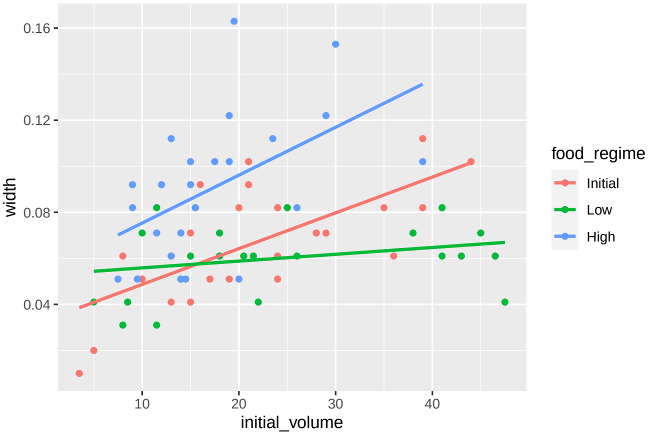
Scatterplot



Goal: Predict width as a function of `food_regime` and `initial_volume`.

- Does an additive model seem appropriate?

Scatterplot



Goal: Predict width as a function of `food_regime` and `initial_volume`.

- Does an additive model seem appropriate?
- One option might be a linear model with interaction terms.

Build it!

Our model formula takes the form `width ~ initial_volume + food_regime + initial_volume:food_regime` (or `width ~ initial_volume*food_regime`)

Build it!

Our model formula takes the form `width ~ initial_volume + food_regime + initial_volume:food_regime` (or `width ~ initial_volume*food_regime`)

We need to specify the model's functional form. Then specify the method for fitting using `set_engine()`

```
library(parsnip)
linear_reg() %>%
  set_engine("lm")
```

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

Build it!

Our model formula takes the form `width ~ initial_volume + food_regime + initial_volume:food_regime` (or `width ~ initial_volume*food_regime`)

We need to specify the model's functional form. Then specify the method for fitting using `set_engine()`

```
library(parsnip)
linear_reg() %>%
  set_engine("lm")
```

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

- Other engines are possible for `linear_reg()`: `glmnet`, `stan`, and more

Build it!

Our model formula takes the form `width ~ initial_volume + food_regime + initial_volume:food_regime` (or `width ~ initial_volume*food_regime`)

We need to specify the model's functional form. Then specify the method for fitting using `set_engine()`

```
library(parsnip)
linear_reg() %>%
  set_engine("lm")
```

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

- Other engines are possible for `linear_reg()`: `glmnet`, `stan`, and more

Now we create the model based on data using the `fit` function:

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

lm_fit<- lm_mod %>%
  fit(width ~ initial_volume*food_regime, data = sea_urchins)
```

Results

The output of our `lm_fit` object:

```
lm_fit
```

```
## parsnip model object
##
## Fit time:  4ms
##
## Call:
## stats::lm(formula = width ~ initial_volume * food_regime, data = data)
##
## Coefficients:
##                    (Intercept)                    initial_volume
##                      0.0331216                         0.0015546
##                 food_regimeLow                   food_regimeHigh
##                      0.0197824                         0.0214111
##  initial_volume:food_regimeLow  initial_volume:food_regimeHigh
##                     -0.0012594                         0.0005254
```

Summary Table

To get the traditional `summary` table:

```
tidy(lm_fit) %>% kable()
```

term	estimate	std.error	statistic	p.value
(Intercept)	0.0331216	0.0096186	3.4434873	0.0010020
initial_volume	0.0015546	0.0003978	3.9077643	0.0002220
food_regimeLow	0.0197824	0.0129883	1.5230864	0.1325145
food_regimeHigh	0.0214111	0.0145318	1.4733993	0.1453970
initial_volume:food_regimeLow	-0.0012594	0.0005102	-2.4685525	0.0161638
initial_volume:food_regimeHigh	0.0005254	0.0007020	0.7484702	0.4568356

Summary Table

To get the traditional `summary` table:

```r
tidy(lm_fit) %>% kable()
```

term	estimate	std.error	statistic	p.value
(Intercept)	0.0331216	0.0096186	3.4434873	0.0010020
initial_volume	0.0015546	0.0003978	3.9077643	0.0002220
food_regimeLow	0.0197824	0.0129883	1.5230864	0.1325145
food_regimeHigh	0.0214111	0.0145318	1.4733993	0.1453970
initial_volume:food_regimeLow	-0.0012594	0.0005102	-2.4685525	0.0161638
initial_volume:food_regimeHigh	0.0005254	0.0007020	0.7484702	0.4568356

Note that the output is a data frame with standard column names

New Data

Suppose we wish to predict the `width` of 6 sea urchins with `initial_volume` 5 and 30 ml, and with each different `food_regime`.

New Data

Suppose we wish to predict the `width` of 6 sea urchins with `initial_volume` 5 and 30 ml, and with each different `food_regime`.

- First, we generate data:

New Data

Suppose we wish to predict the `width` of 6 sea urchins with `initial_volume` 5 and 30 ml, and with each different `food_regime`.

- First, we generate data:

```
new_urchins <- expand.grid(initial_volume  = c(5,30),
                           food_regime = c("Initial", "Low", "High"))
new_urchins %>% kable()
```

initial_volume	food_regime
5	Initial
30	Initial
5	Low
30	Low
5	High
30	High

Make predictions

Then we make predictions

```
new_preds <- predict(lm_fit, new_data = new_urchins)
conf_int_preds<-predict(lm_fit, new_data = new_urchins, type = "conf_int")
new_preds %>% kable()
```

.pred
0.0408948
0.0797608
0.0543803
0.0617621
0.0649329
0.1169338

```
conf_int_preds %>% kable()
```

.pred_lower	.pred_upper
0.0251382	0.0566514
0.0688612	0.0906605
0.0396403	0.0691204
0.0522641	0.0712601
0.0483265	0.0815393
0.0999144	0.1339532

Combining Data and Predictions

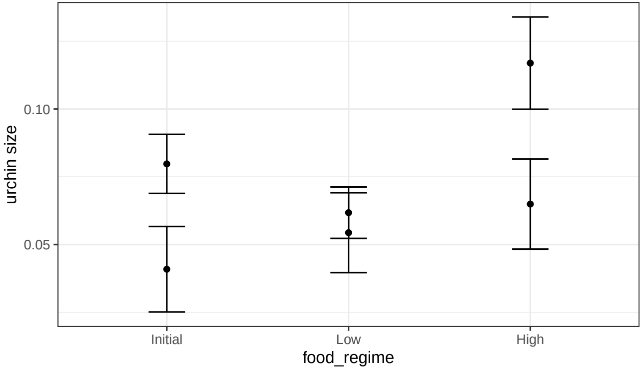
Because the result of `predict()` is tidy, we can easily combine it with the original data:

```
combined_data <- new_urchins %>% cbind(new_preds) %>% cbind(conf_int_preds)
combined_data %>% kable()
```

initial_volume	food_regime	.pred	.pred_lower	.pred_upper
5	Initial	0.0408948	0.0251382	0.0566514
30	Initial	0.0797608	0.0688612	0.0906605
5	Low	0.0543803	0.0396403	0.0691204
30	Low	0.0617621	0.0522641	0.0712601
5	High	0.0649329	0.0483265	0.0815393
30	High	0.1169338	0.0999144	0.1339532

Predictions Plot

```
ggplot(combined_data, aes(x = food_regime)) +
  geom_point(aes(y = .pred)) +
  geom_errorbar(aes(ymin = .pred_lower, ymax = .pred_upper),width = .2) +
  labs(y = "urchin size")+theme_bw()
```



Using a different engine

With only 3 predictors (`food_regime`, `initial_width` and the interaction term), its unlikely our model will be improved by Penalized Regression. But let's try anyway:

```
glmnet_mod<- linear_reg(mixture = 1) %>% #mixture specifies alpha parameter
  set_engine("glmnet")
```

Using a different engine

With only 3 predictors (`food_regime`, `initial_width` and the interaction term), its unlikely our model will be improved by Penalized Regression. But let's try anyway:

```
glmnet_mod<- linear_reg(mixture = 1) %>% #mixture specifies alpha parameter
  set_engine("glmnet")
```

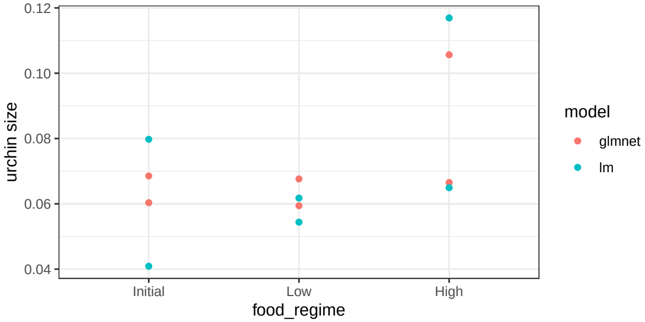
```
glmnet_fit <- glmnet_mod %>% fit(width ~ initial_volume*food_regime,
                                 data = sea_urchins)
tidy(glmnet_fit, penalty = .004) #penalty selects particular value of lambda
```

```
## # A tibble: 6 x 3
##   term                              estimate penalty
##   <chr>                                <dbl>   <dbl>
## 1 (Intercept)                       0.0587     0.004
## 2 initial_volume                    0.000328   0.004
## 3 food_regimeLow                   -0.000918   0.004
## 4 food_regimeHigh                   0          0.004
## 5 initial_volume:food_regimeLow     0          0.004
## 6 initial_volume:food_regimeHigh    0.00124    0.004
```

Results from glmnet

```
new_glmnet_preds <- predict(glmnet_fit, new_data = new_urchins, penalty = 0.004)
combined_glmnet_data <- new_urchins %>% cbind(new_glmnet_preds)
two_models <- rbind(combined_glmnet_data,
                    combined_data %>% select(-.pred_lower, -.pred_upper )) %>%
  mutate(model = rep(c("glmnet","lm"), each = 6))
```

```
ggplot(two_models, aes(x = food_regime)) +
  geom_point(aes(y = .pred, color = model) ) +
  labs(y = "urchin size")+theme_bw()
```



Section 3

Preprocessing with recipes

Recipes

The `recipes` package assists with preprocessing before a model is trained

Recipes

The `recipes` package assists with preprocessing before a model is trained

- Converts qualitative predictors to dummy variables
- Transforms data to be on a different scale
- Transforms several predictors at the same time
- Extracts features from variable

Recipes

The `recipes` package assists with preprocessing before a model is trained

- Converts qualitative predictors to dummy variables
- Transforms data to be on a different scale
- Transforms several predictors at the same time
- Extracts features from variable

The main advance of `recipes` is that it allows us combine several steps at once, in a reproducible fashion

House Prices

The `house` data contains information on 30 predictors for 200 houses in Ames, Iowa

```
glimpse(house)
```

```
## Rows: 200
## Columns: 31
## $ SalePrice     <int> 181500, 223500, 200000, 149000, 154000, 134800, 30600...
## $ Id            <int> 2, 3, 8, 17, 25, 27, 28, 43, 51, 54, 58, 69, 72, 79, ...
## $ Functional    <fct> Typ, Typ, Typ, Typ, Typ, Typ, Typ, Typ, Typ, Typ, Typ...
## $ BldgType      <fct> 1Fam, 1Fam, 1Fam, 1Fam, 1Fam, 1Fam, 1Fam, 1Fam, 1Fam,...
## $ Foundation    <fct> CBlock, PConc, CBlock, CBlock, CBlock, CBlock, PConc,...
## $ LotShape      <fct> Reg, IR1, IR1, IR1, IR1, Reg, Reg, IR1, IR2, IR1, IR1...
## $ LandSlope     <fct> Gtl, Gtl, Gtl, Gtl, Gtl, Gtl, Gtl, Gtl, Gtl, Gtl, Gtl...
## $ SaleCondition <fct> Normal, Normal, Normal, Normal, Normal, Normal, Norma...
## $ RoofMatl      <fct> CompShg, CompShg, CompShg, CompShg, CompShg, CompShg,...
## $ ScreenPorch   <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
## $ MSSubClass    <int> 20, 60, 60, 20, 20, 20, 20, 85, 60, 20, 60, 30, 20, 9...
## $ GarageCars    <int> 2, 2, 2, 2, 1, 2, 3, 2, 2, 3, 2, 1, 2, 0, 0, 2, 0, 2,...
## $ BedroomAbvGr  <int> 3, 3, 3, 2, 3, 3, 3, 2, 3, 0, 3, 2, 2, 4, 3, 2, 3, 2,...
## $ TotalBsmtSF   <int> 1262, 920, 1107, 1004, 1060, 900, 1704, 840, 794, 184...
## $ LotArea       <int> 9600, 11250, 10382, 11241, 8246, 7200, 11478, 9180, 1...
## $ OpenPorchSF   <int> 0, 42, 204, 0, 90, 32, 50, 0, 75, 72, 70, 0, 0, 0, 0,...
## $ BsmtFullBath  <int> 0, 1, 1, 1, 1, 0, 1, 1, 0, 2, 0, 0, 1, 0, 1, 0, 1, 0,...
## $ WoodDeckSF    <int> 298, 0, 235, 0, 406, 222, 0, 240, 0, 857, 0, 0, 0, 0,...
## $ OverallCond   <int> 8, 5, 6, 7, 8, 7, 5, 7, 6, 5, 5, 6, 6, 5, 5, 3, 5, 5,...
## $ YrSold        <int> 2007, 2008, 2009, 2010, 2010, 2010, 2010, 2007, 2007,...
## $ GrLivArea     <int> 1262, 1786, 2090, 1004, 1060, 900, 1704, 884, 1470, 1...
## $ MoSold        <int> 5, 9, 11, 3, 5, 5, 5, 12, 7, 11, 8, 6, 6, 4, 8, 12, 1...
## $ TotRmsAbvGrd  <int> 6, 6, 7, 5, 6, 5, 7, 5, 6, 5, 7, 4, 4, 8, 5, 6, 6, 5,...
## $ PoolArea      <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...
## $ YearBuilt     <int> 1976, 2001, 1973, 1970, 1968, 1951, 2007, 1983, 1997,...
## $ GarageArea    <int> 460, 608, 484, 480, 270, 576, 772, 504, 388, 894, 565...
## $ OverallQual   <int> 6, 7, 7, 6, 5, 5, 8, 5, 6, 9, 7, 4, 4, 4, 4, 5, 4, 5,...
## $ Fireplaces    <int> 1, 1, 2, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,...
## $ EnclosedPorch <int> 0, 0, 228, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Investigate Predictors

Look at the list of variables:

```
names(house)
```

```
##  [1] "SalePrice"     "Id"            "Functional"    "BldgType"
##  [5] "Foundation"    "LotShape"      "LandSlope"     "SaleCondition"
##  [9] "RoofMatl"      "ScreenPorch"   "MSSubClass"    "GarageCars"
## [13] "BedroomAbvGr"  "TotalBsmtSF"   "LotArea"       "OpenPorchSF"
## [17] "BsmtFullBath"  "WoodDeckSF"    "OverallCond"   "YrSold"
## [21] "GrLivArea"     "MoSold"        "TotRmsAbvGrd"  "PoolArea"
## [25] "YearBuilt"     "GarageArea"    "OverallQual"   "Fireplaces"
## [29] "EnclosedPorch" "FullBath"      "HalfBath"
```

Investigate Predictors

Look at the list of variables:

```
names(house)
```

```
##  [1] "SalePrice"     "Id"           "Functional"   "BldgType"
##  [5] "Foundation"    "LotShape"     "LandSlope"    "SaleCondition"
##  [9] "RoofMatl"      "ScreenPorch"  "MSSubClass"   "GarageCars"
## [13] "BedroomAbvGr"  "TotalBsmtSF"  "LotArea"      "OpenPorchSF"
## [17] "BsmtFullBath"  "WoodDeckSF"   "OverallCond"  "YrSold"
## [21] "GrLivArea"     "MoSold"       "TotRmsAbvGrd" "PoolArea"
## [25] "YearBuilt"     "GarageArea"   "OverallQual"  "Fireplaces"
## [29] "EnclosedPorch" "FullBath"     "HalfBath"
```

- Note that the variable `Id` is not useful as a predictor, but is useful for referring to houses in the data set.

Investigate Predictors

Additionally, note that several of the variables are factors, so should be converted to a collection of dummy variables.

Investigate Predictors

Additionally, note that several of the variables are factors, so should be converted to a collection of dummy variables.

Moreover, for a few variables, some levels are very underrepresented.

```
library(skimr)
house %>% skim(RoofMatl)
```

Table 7: Data summary

Name	Piped data
Number of rows	200
Number of columns	31
_______________________	
Column type frequency:	
factor	1
________________________	
Group variables	None

Variable type: factor

skim_variable	n_missing	complete_rate	ordered	n_unique	top_counts
RoofMatl	0	1	FALSE	5	Com: 195, Tar: 2, Mem: 1, WdS: 1

Data Splitting

We can use the `rsample` package to create a test-training split

Data Splitting

We can use the `rsample` package to create a test-training split

- The `rsample` package allows us to create stratified samples in addition to simple random samples

Data Splitting

We can use the `rsample` package to create a test-training split

- The `rsample` package allows us to create stratified samples in addition to simple random samples

```
library(rsample)
set.seed(1221)
data_split <- initial_split(house , prop = 3/4)
train_data <- training(data_split)
test_data <- testing(data_split)
```

Create a recipe and update roles

We now create a recipe for some data pre-processing

```
library(recipes)
house_rec <-
  recipe(SalePrice ~ ., data = train_data) %>%
  update_role(Id, new_role = "ID")
```

Create a recipe and update roles

We now create a recipe for some data pre-processing

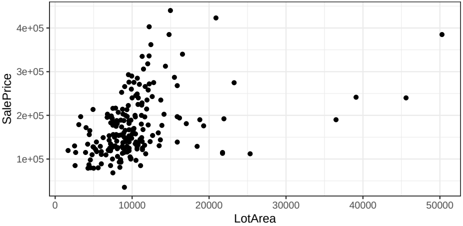
```
library(recipes)
house_rec <-
  recipe(SalePrice ~ ., data = train_data) %>%
  update_role(Id, new_role = "ID")
```

```
summary(house_rec)
```

```
## # A tibble: 31 x 4
##    variable      type    role      source
##    <chr>         <chr>   <chr>     <chr>
##  1 Id            numeric ID        original
##  2 Functional    nominal predictor original
##  3 BldgType      nominal predictor original
##  4 Foundation    nominal predictor original
##  5 LotShape      nominal predictor original
##  6 LandSlope     nominal predictor original
##  7 SaleCondition nominal predictor original
##  8 RoofMatl      nominal predictor original
##  9 ScreenPorch   numeric predictor original
## 10 MSSubClass    numeric predictor original
## # ... with 21 more rows
```

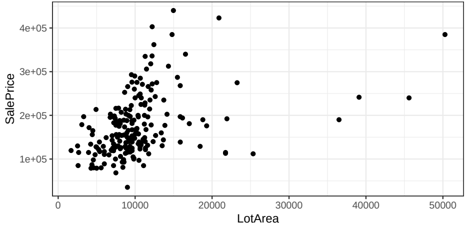
Add steps to recipes

Consider the relationship between of sale price and lot area:

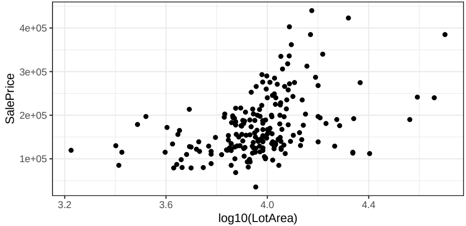


Add steps to recipes

Consider the relationship between of sale price and lot area:



Accuracy of a linear model may improve by performing log transformation on LotArea:



Adding steps to recipes

Let's update our recipe:

```
house_rec <- house_rec %>%
  step_log(LotArea, base = 10)

house_rec
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##         ID          1
##    outcome          1
##  predictor         29
##
## Operations:
##
## Log transformation on LotArea
```

Create New Variables from Old

The original data set contains variables `FullBath` and `HalfBath`. But we want a measure of total number of baths:

$$\text{TotalBath} = \text{FullBath} + \frac{1}{2}\text{HalfBath}$$

Create New Variables from Old

The original data set contains variables `FullBath` and `HalfBath`. But we want a measure of total number of baths:

$$\text{TotalBath} = \text{FullBath} + \frac{1}{2}\text{HalfBath}$$

We can also add a mutate step in our recipe to do just this:

```
house_rec <- house_rec %>%
  step_mutate(TotalBath = FullBath+0.5*HalfBath) %>%
  step_rm(FullBath, HalfBath)

house_rec
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##         ID          1
##    outcome          1
##  predictor         29
##
## Operations:
##
## Log transformation on LotArea
## Variable mutation for TotalBath
## Delete terms FullBath, HalfBath
```

Create Dummy Variables

Recall that 7 of our variables are factors (`Functional`, `BldgType`, `Foundation`, `LotShape`, `LandSlope`, `SaleCondition`, `RoofMatl`). To create appropriate dummy variables:

```
house_rec <- house_rec %>% step_dummy(all_nominal(), -all_outcomes())
house_rec
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##         ID          1
##    outcome          1
##  predictor         29
##
## Operations:
##
## Log transformation on LotArea
## Variable mutation for TotalBath
## Delete terms FullBath, HalfBath
## Dummy variables from all_nominal(), -all_outcomes()
```

Create Dummy Variables

Recall that 7 of our variables are factors (`Functional`, `BldgType`, `Foundation`, `LotShape`, `LandSlope`, `SaleCondition`, `RoofMatl`). To create appropriate dummy variables:

```
house_rec <- house_rec %>% step_dummy(all_nominal(), -all_outcomes())
house_rec
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##         ID          1
##    outcome          1
##  predictor         29
##
## Operations:
##
## Log transformation on LotArea
## Variable mutation for TotalBath
## Delete terms FullBath, HalfBath
## Dummy variables from all_nominal(), -all_outcomes()
```

- The first argument `all_nominal` selects all variables that are either factors or characters
- The second argument `-all_outcomes` removes any response variables from this step

Create Dummy Variables

Recall that 7 of our variables are factors (`Functional`, `BldgType`, `Foundation`, `LotShape`, `LandSlope`, `SaleCondition`, `RoofMatl`). To create appropriate dummy variables:

```
house_rec <- house_rec %>% step_dummy(all_nominal(), -all_outcomes())
house_rec
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##         ID          1
##    outcome          1
##  predictor         29
##
## Operations:
##
## Log transformation on LotArea
## Variable mutation for TotalBath
## Delete terms FullBath, HalfBath
## Dummy variables from all_nominal(), -all_outcomes()
```

- The first argument `all_nominal` selects all variables that are either factors or characters
- The second argument `-all_outcomes` removes any response variables from this step

Remove Problematic Predictors

Finally, to avoid the situation where an infrequently occuring level doesn't exist in the training or test sets:

```
house_rec <- house_rec %>% step_zv(all_predictors())
house_rec
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##         ID          1
##    outcome          1
##  predictor         29
##
## Operations:
##
## Log transformation on LotArea
## Variable mutation for TotalBath
## Delete terms FullBath, HalfBath
## Dummy variables from all_nominal(), -all_outcomes()
## Zero variance filter on all_predictors()
```

Remove Problematic Predictors

Finally, to avoid the situation where an infrequently occuring level doesn't exist in the training or test sets:

```
house_rec <- house_rec %>% step_zv(all_predictors())
house_rec
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##         ID          1
##    outcome          1
##  predictor         29
##
## Operations:
##
## Log transformation on LotArea
## Variable mutation for TotalBath
## Delete terms FullBath, HalfBath
## Dummy variables from all_nominal(), -all_outcomes()
## Zero variance filter on all_predictors()
```

- The `step_zv` verb removes columns from the training data which have a single value

Workflows

Why create a recipe when we could just as easily perform the pre-processing steps using `dplyr`?

Workflows

Why create a recipe when we could just as easily perform the pre-processing steps using `dplyr`?

1. The recipe allows us to apply the same procedures to both test and training data.
2. The recipe gives instructions for processing the data **without actually performing that action**

Workflows

Why create a recipe when we could just as easily perform the pre-processing steps using `dplyr`?

1. The recipe allows us to apply the same procedures to both test and training data.
2. The recipe gives instructions for processing the data **without actually performing that action**

To use our recipe across several steps, we will use a *workflow*, which will

1. Process the recipe using the training set
2. Apply the recipe to the training set
3. Apply the recipe to the test set

Create the workflow

```
house_mod <- linear_reg() %>% set_engine("lm")

house_wflow <- workflow() %>%
  add_model(house_mod) %>%
  add_recipe(house_rec)

house_wflow
```

```
## == Workflow ==================================================================
## Preprocessor: Recipe
## Model: linear_reg()
##
## -- Preprocessor --------------------------------------------------------------
## 5 Recipe Steps
##
## * step_log()
## * step_mutate()
## * step_rm()
## * step_dummy()
## * step_zv()
##
## -- Model ---------------------------------------------------------------------
## Linear Regression Model Specification (regression)
##
```

Fitting Models with Workflows

```r
house_fit <- house_wflow %>% fit(data = train_data)

house_fit %>% pull_workflow_fit() %>% tidy()
```

```
## # A tibble: 46 x 5
##    term           estimate  std.error statistic p.value
##    <chr>             <dbl>      <dbl>     <dbl>   <dbl>
##  1 (Intercept)  2335263.   3579757.    0.652    0.516
##  2 ScreenPorch      112.         57.7   1.93     0.0557
##  3 MSSubClass      -249.        140.   -1.78     0.0781
##  4 GarageCars      -684.       5990.   -0.114    0.909
##  5 BedroomAbvGr   -2812.       4198.   -0.670    0.504
##  6 TotalBsmtSF       17.1         8.79  1.95     0.0543
##  7 LotArea          -15.0     19935.   -0.000752 0.999
##  8 OpenPorchSF      -22.4        45.5  -0.491    0.624
##  9 BsmtFullBath   14277.       5125.    2.79     0.00632
## 10 WoodDeckSF         1.69       18.8   0.0900   0.928
## # ... with 36 more rows
```

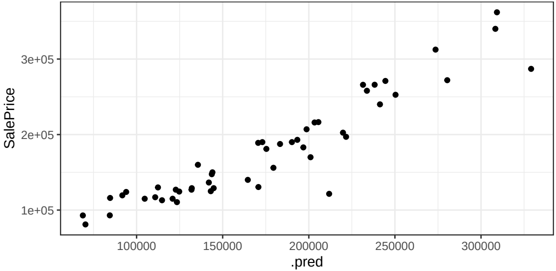
Making predictions with workflow

```
house_preds<- predict(house_fit, test_data)
house_preds
```

```
## # A tibble: 50 x 1
##      .pred
##      <dbl>
##  1 143084.
##  2 131894.
##  3 250360.
##  4 205571.
##  5 114775.
##  6 198707.
##  7 219853.
##  8 179459.
##  9 190201.
## 10 122767.
## # ... with 40 more rows
```

Evaluate performance

```
house_results <- house_preds %>% cbind(test_data)
```




```
rbind(
  rmse(house_results, truth = SalePrice, estimate = .pred),
  rsq(house_results, truth = SalePrice, estimate = .pred)
)
```

```
## # A tibble: 2 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard   24410.
## 2 rsq     standard       0.871
```