Tidymodels

Prof Wells

STA 295: Stat Learning

May 2nd, 2024

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 and want predicted probabilities.

Why `tidymodels`?

- Suppose we plan to classify data with a binary response and want predicted probabilities.
  - Several different models are available:

Model	Function	Code
Logistic	`glm`	`predict(object, type = "response")`
Penalized Logistic	`glmnet`	`predict(object, s, type = "response")`
KNN	`kknn`	`kknn(...)$prob`
Naive Bayes	`naiveBayes`	`predict(object, type = "raw")`
Tree	`rpart`	`predict(object, type = "prob")`
Random Forest	`randomForest`	`predict(object, type = "prob")`
Boosted Tree	`gbm`	`predict(object, type = "response", n.trees)`

Why tidymodels?

- Suppose we plan to classify data with a binary response and want predicted probabilities.
  - Several different models are available:

Model	Function	Code
Logistic	`glm`	`predict(object, type = "response")`
Penalized Logistic	`glmnet`	`predict(object, s, type = "response")`
KNN	`kknn`	`kknn(...)$prob`
Naive Bayes	`naiveBayes`	`predict(object, type = "raw")`
Tree	`rpart`	`predict(object, type = "prob")`
Random Forest	`randomForest`	`predict(object, type = "prob")`
Boosted Tree	`gbm`	`predict(object, type = "response", n.trees)`

- Each model has different methods for making class probability predictions

Why tidymodels?

- Suppose we plan to classify data with a binary response and want predicted probabilities.
    - Several different models are available:

Model	Function	Code
Logistic	`glm`	`predict(object, type = "response")`
Penalized Logistic	`glmnet`	`predict(object, s, type = "response")`
KNN	`kknn`	`kknn(...)$prob`
Naive Bayes	`naiveBayes`	`predict(object, type = "raw")`
Tree	`rpart`	`predict(object, type = "prob")`
Random Forest	`randomForest`	`predict(object, type = "prob")`
Boosted Tree	`gbm`	`predict(object, type = "response", n.trees)`

- Each model has 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 packages.

tidymodels goals

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

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 packages.

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 (`glmnet`, `rpart`, `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

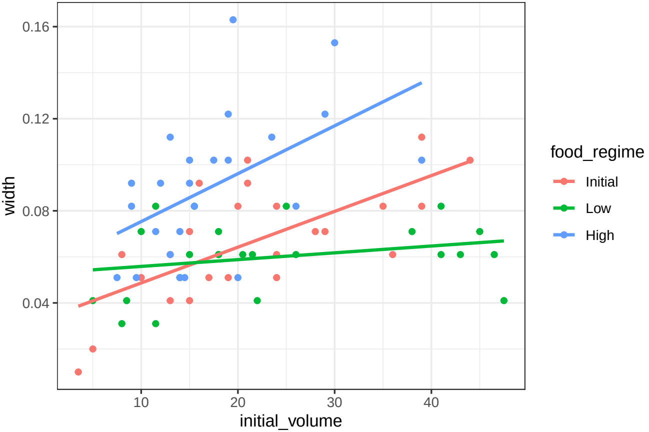
The sea_urchins data set explores the relationship between feeding regimes and size of sea urchins over time:

```
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
```

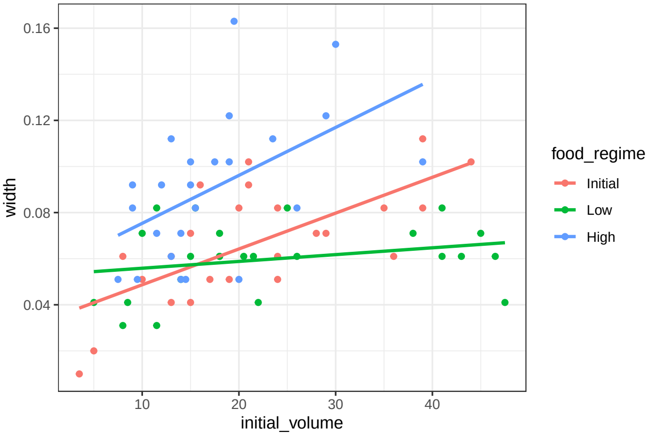
Scatterplot

```
sea_urchins %>%
  ggplot(aes(x = initial_volume, y = width, group = food_regime, color = food_regime)) +
  geom_point() + geom_smooth(method = lm, se = FALSE)
```



Scatterplot

```r
sea_urchins %>%
  ggplot(aes(x = initial_volume, y = width, group = food_regime, color = food_regime)) +
  geom_point() + geom_smooth(method = lm, se = FALSE)
```



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

Build it!

We'll consider an interaction model, which takes the form:

```
width ~ initial_volume + food_regime + initial_volume:food_regime
```

Build it!

We'll consider an interaction model, which takes the form:

```
width ~ initial_volume + food_regime + initial_volume:food_regime
```

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

Build it!

We'll consider an interaction model, which takes the form:

```
width ~ initial_volume + food_regime + initial_volume:food_regime
```

- We need to specify the model's functional form using the `parsnip` package.
- 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!

We'll consider an interaction model, which takes the form:

```
width ~ initial_volume + food_regime + initial_volume:food_regime
```

- We need to specify the model's functional form using the `parsnip` package.
- 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!

We'll consider an interaction model, which takes the form:

```
width ~ initial_volume + food_regime + initial_volume:food_regime
```

- We need to specify the model's functional form using the `parsnip` package.
- 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
##
##
## 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(digits = 3)
```

term	estimate	std.error	statistic	p.value
(Intercept)	0.033	0.010	3.443	0.001
initial_volume	0.002	0.000	3.908	0.000
food_regimeLow	0.020	0.013	1.523	0.133
food_regimeHigh	0.021	0.015	1.473	0.145
initial_volume:food_regimeLow	-0.001	0.001	-2.469	0.016
initial_volume:food_regimeHigh	0.001	0.001	0.748	0.457

Summary Table

To get the traditional `summary` table:

```
tidy(lm_fit) %>% kable(digits = 3)
```

term	estimate	std.error	statistic	p.value
(Intercept)	0.033	0.010	3.443	0.001
initial_volume	0.002	0.000	3.908	0.000
food_regimeLow	0.020	0.013	1.523	0.133
food_regimeHigh	0.021	0.015	1.473	0.145
initial_volume:food_regimeLow	-0.001	0.001	-2.469	0.016
initial_volume:food_regimeHigh	0.001	0.001	0.748	0.457

We can get goodness-of-fit measures using `glance`

```
glance(lm_fit) %>% kable(digits = 3)
```

r.squared	adj.r.squared	sigma	statistic	p.value	df	logLik	AIC	BIC	deviance	df.residual	nobs
0.462	0.421	0.021	11.345	0	5	178.594	-343.188	-327.251	0.03	66	72

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)
new_preds %>% kable(digits = 3)
```

.pred
0.041
0.080
0.054
0.062
0.065
0.117

Combining Data and Predictions

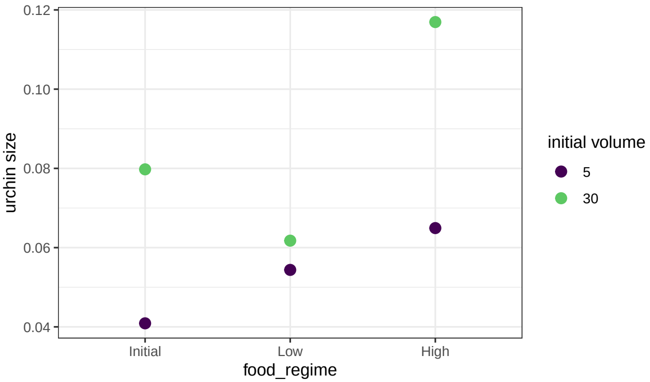
Because the result of `predict()` is tidy, we can easily combine it with the original data:

```
combined_data <- new_urchins %>% cbind(new_preds)
combined_data %>% kable(digits = 3)
```

initial_volume	food_regime	.pred
5	Initial	0.041
30	Initial	0.080
5	Low	0.054
30	Low	0.062
5	High	0.065
30	High	0.117

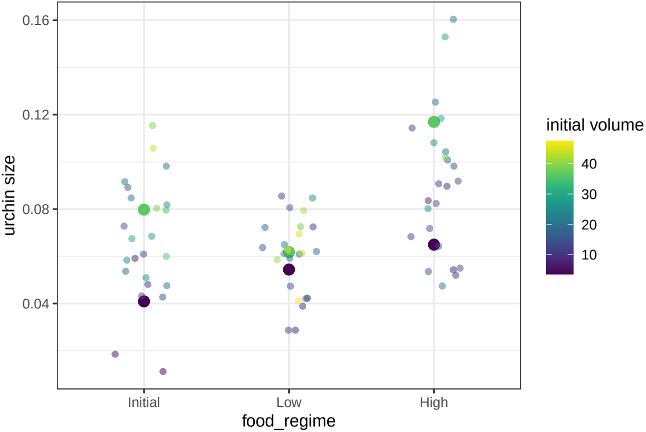
Predictions Plot

```r
ggplot(combined_data, aes(x = food_regime)) +
  geom_point(aes(y = .pred, color = initial_volume))
```



Predictions Plot

We can compare our predictions to the original data:



Using a different engine

LASSO?

- 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(penalty = 0.01, mixture = 1) %>% set_engine("glmnet")
```

- `mixture = 1` indicates LASSO (`mixture = 0` is used for Ridge Regression)
- `glmnet` requires us to indicate a value of `penalty` parameter λ to make predictions.
    - Here, we choose `penalty = 0.01` entirely arbitrarily; in any case, `glmnet` will still create models for all λ regardless of penalty selected

Using a different engine

LASSO?

- 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(penalty = 0.01, mixture = 1) %>% set_engine("glmnet")
```

- `mixture = 1` indicates LASSO (`mixture = 0` is used for Ridge Regression)
- `glmnet` requires us to indicate a value of `penalty` parameter λ to make predictions.
  - Here, we choose `penalty = 0.01` entirely arbitrarily; in any case, `glmnet` will still create models for all λ regardless of penalty selected

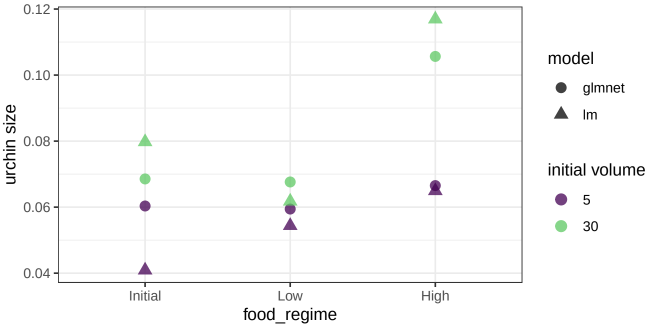
```
glmnet_fit <- glmnet_mod %>% fit(width ~ initial_volume*food_regime, data = sea_urchins)
tidy(glmnet_fit, penalty = .004) #penalty selects particular value of lambda; can be anything
```

```
## # 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) %>%
  mutate(model = rep(c("glmnet","lm"), each = 6))
```

```
ggplot(two_models, aes(x = food_regime)) +
  geom_point(aes(y = .pred, color = initial_volume, shape = model))
```



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

```
dim(house)
```

```
## [1] 200  31
```

```
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"
```

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
- By setting `strata = SalePrice`, we ensure that SalePrice values are balanced across the test and training sets.

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
- By setting `strata = SalePrice`, we ensure that SalePrice values are balanced across the test and training sets.

```
library(rsample)
set.seed(1221)
data_split <- initial_split(house , prop = 3/4, strata = SalePrice)
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

```
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

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

- The variable `Id` is not useful as a predictor, but is useful for referring to houses in the data set. By setting it to the `ID` role, it will not be used for fitting models.

Create a recipe and update roles

- We now create a recipe for some data pre-processing

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

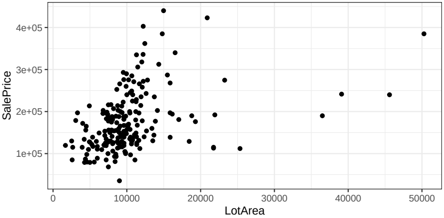
- The variable Id is not useful as a predictor, but is useful for referring to houses in the data set. By setting it to the ID role, it will not be used for fitting models.

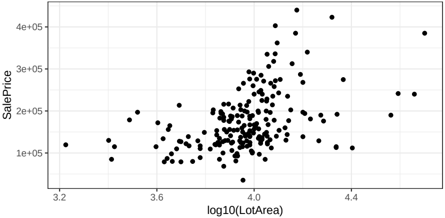
```
summary(house_rec)
```

```
## # A tibble: 31 x 4
##    variable      type      role      source
##    <chr>         <list>    <chr>     <chr>
##  1 SalePrice     <chr [2]> outcome   original
##  2 Id            <chr [2]> ID        original
##  3 Functional    <chr [3]> predictor original
##  4 BldgType      <chr [3]> predictor original
##  5 Foundation    <chr [3]> predictor original
##  6 LotShape      <chr [3]> predictor original
##  7 LandSlope     <chr [3]> predictor original
##  8 SaleCondition <chr [3]> predictor original
##  9 RoofMatl      <chr [3]> predictor original
## 10 ScreenPorch   <chr [2]> predictor original
## # i 21 more rows
```

Add steps to recipes

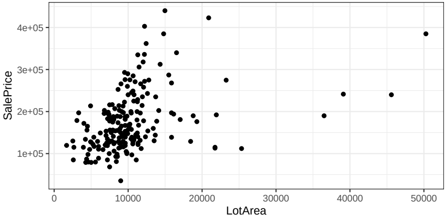
- Consider the relationship between of sale price and lot area:

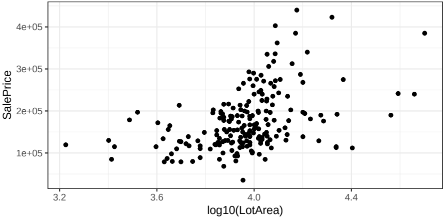




Add steps to recipes

- Consider the relationship between of sale price and lot area:

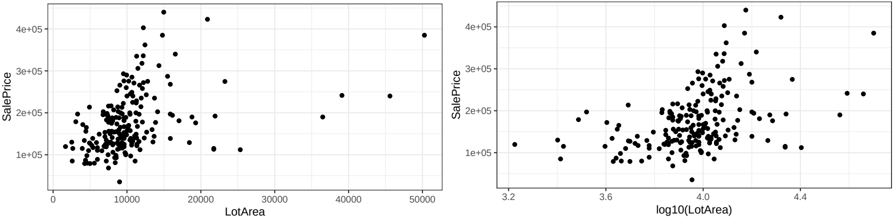




- Accuracy of a linear model may improve by performing log transformation on `LotArea`:

Add steps to recipes

- Consider the relationship between of sale price and lot area:

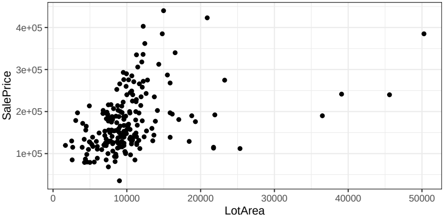


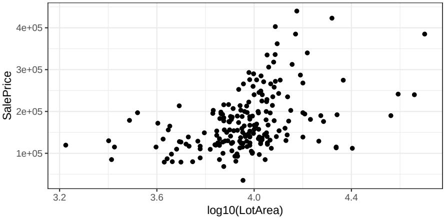
- Accuracy of a linear model may improve by performing log transformation on `LotArea`:
- Let's update our recipe to take the log of `LotArea`:

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

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`:
- Let's update our recipe to take the log of `LotArea`:

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

Mutating Data

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

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

Mutating Data

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)
```

Mutating Data

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)
```

- Here, we also removed the two original variables `FullBath` and `HalfBath`

Mutating Data

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)
```

- Here, we also removed the two original variables `FullBath` and `HalfBath`

Imbalanced Predictors

- Note that for a few categorical variables, some levels are very underrepresented.

```
house %>% count(RoofMatl)
```

```
##   RoofMatl   n
## 1  CompShg 195
## 2  Membran   1
## 3  Tar&Grv   2
## 4  WdShake   1
## 5  WdShngl   1
```

Imbalanced Predictors

- Note that for a few categorical variables, some levels are very underrepresented.

```
house %>% count(RoofMatl)
```

```
##   RoofMatl   n
## 1  CompShg 195
## 2  Membran   1
## 3  Tar&Grv   2
## 4  WdShake   1
## 5  WdShngl   1
```

- This can be particularly problematic if some of these levels only appear in the test set, but not the training set (since models won't know how to handle new variables)

Imbalanced Predictors

- Note that for a few categorical variables, some levels are very underrepresented.

```
house %>% count(RoofMatl)
```

```
##   RoofMatl   n
## 1  CompShg 195
## 2  Membran   1
## 3  Tar&Grv   2
## 4  WdShake   1
## 5  WdShngl   1
```

- This can be particularly problematic if some of these levels only appear in the test set, but not the training set (since models won't know how to handle new variables)
- To fix, we add `step_novel` to our recipe, which takes any new (previously unseen) factor level and groups them into a new factor called "new"

Imbalanced Predictors

- Note that for a few categorical variables, some levels are very underrepresented.

```
house %>% count(RoofMatl)
```

```
##   RoofMatl   n
## 1  CompShg 195
## 2  Membran   1
## 3  Tar&Grv   2
## 4  WdShake   1
## 5  WdShngl   1
```

- This can be particularly problematic if some of these levels only appear in the test set, but not the training set (since models won't know how to handle new variables)
- To fix, we add `step_novel` to our recipe, which takes any new (previously unseen) factor level and groups them into a new factor called "new"

```
house_rec <- house_rec %>% step_novel(all_nominal())
```

Imbalanced Predictors

- Note that for a few categorical variables, some levels are very underrepresented.

```
house %>% count(RoofMatl)
```

```
##   RoofMatl   n
## 1  CompShg 195
## 2  Membran   1
## 3  Tar&Grv   2
## 4  WdShake   1
## 5  WdShngl   1
```

- This can be particularly problematic if some of these levels only appear in the test set, but not the training set (since models won't know how to handle new variables)
- To fix, we add `step_novel` to our recipe, which takes any new (previously unseen) factor level and groups them into a new factor called “new”

```
house_rec <- house_rec %>% step_novel(all_nominal())
```

- Here, we only apply this step to the nominal (i.e. categorical) variables

Creating Dummy Variables

Recall 7 of our variables are categorical, which will need to be converted to dummy variables for many models:

```
house %>% select_if(is.character) %>% names()
```

```
## [1] "Functional"    "BldgType"      "Foundation"    "LotShape"
## [5] "LandSlope"     "SaleCondition" "RoofMatl"
```

Creating Dummy Variables

Recall 7 of our variables are categorical, which will need to be converted to dummy variables for many models:

```
house %>% select_if(is.character) %>% names()
```

```
## [1] "Functional"    "BldgType"      "Foundation"    "LotShape"
## [5] "LandSlope"     "SaleCondition" "RoofMatl"
```

To create appropriate dummy variables:

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

Creating Dummy Variables

Recall 7 of our variables are categorical, which will need to be converted to dummy variables for many models:

```
house %>% select_if(is.character) %>% names()
```

```
## [1] "Functional"    "BldgType"      "Foundation"    "LotShape"
## [5] "LandSlope"     "SaleCondition" "RoofMatl"
```

To create appropriate dummy variables:

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

- Here, `all_nominal` selects all variables that are either factors or characters, while `-all_outcomes` removes any response variables from this step

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_novel()
## * step_dummy()
##
## -- Model ------------------------------------------------------------------------
## Linear Regression Model Specification (regression)
##
## Computational engine: lm
```

Fitting Models with Workflows

When we are ready to actually fit the model, we apply `fit` to the workflow:

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

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

```
## # A tibble: 55 x 5
##    term          estimate  std.error statistic p.value
##    <chr>            <dbl>      <dbl>     <dbl>   <dbl>
##  1 (Intercept)  3341085.  3204406.       1.04  0.300
##  2 ScreenPorch       86.4       49.5      1.75  0.0834
##  3 MSSubClass      -209.       126.      -1.66  0.0997
##  4 GarageCars      4149.      6471.       0.641 0.523
##  5 BedroomAbvGr    -413.      3693.      -0.112 0.911
##  6 TotalBsmtSF       17.3        7.91     2.18  0.0312
##  7 LotArea        13247.     16943.       0.782 0.436
##  8 OpenPorchSF      -42.8       38.0     -1.13  0.263
##  9 BsmtFullBath   13913.      4542.       3.06  0.00279
## 10 WoodDeckSF        16.1       16.4      0.984 0.327
## # i 45 more rows
```

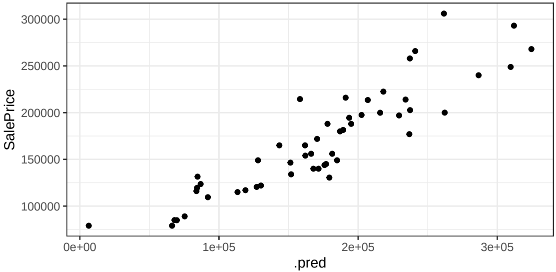
Making predictions with workflow

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

```
## # A tibble: 52 x 1
##      .pred
##      <dbl>
##  1 189262.
##  2 262209.
##  3 184852.
##  4 162068.
##  5 261673.
##  6 236809.
##  7  86811.
##  8 218140.
##  9 175821.
## 10  66268.
## # i 42 more rows
```

Evaluate performance

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




```r
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   30658.
## 2 rsq     standard       0.813
```

Section 4

Resampling

Resampling with `rsample`

- We previously built a linear model for `SalePrice` as a function of predictors in the `house` data and found the following accuracy measures on **test** data:

```
## # A tibble: 2 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard   30658.
## 2 rsq     standard       0.813
```

Resampling with rsample

- We previously built a linear model for SalePrice as a function of predictors in the house data and found the following accuracy measures on **test** data:

```
## # A tibble: 2 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard   30658.
## 2 rsq     standard       0.813
```

- But how typical are these estimates? Let's perform cross-validation.

```r
set.seed(271)
library(rsample)
folds <- vfold_cv(train_data, v = 10)
```

Delving Deeper

- Which observations are in each fold?

```
folds$splits[[1]]
```

```
## <Analysis/Assess/Total>
## <133/15/148>
```

```
folds$splits[[1]] %>% analysis() %>% head() %>% select(1:5)
```

```
##   SalePrice  Id Functional BldgType Foundation
## 1     80000  69        Typ     1Fam     CBlock
## 2     98600  92        Typ     1Fam     CBlock
## 3     87000 128        Typ     1Fam     BrkTil
## 4     97000 224        Typ     1Fam     CBlock
## 5    113000 240        Typ     1Fam     CBlock
## 6     85000 345        Typ   TwnhsE     CBlock
```

```
folds$splits[[1]] %>% assessment() %>% head() %>% select(1:5)
```

```
##   SalePrice   Id Functional BldgType Foundation
## 1    113000  423        Typ     1Fam     CBlock
## 2     92900 1091        Typ   Duplex       Slab
## 3    112000 1384        Typ     1Fam     BrkTil
## 4    144000   43        Typ     1Fam     CBlock
## 5    155000  118        Typ     1Fam      PConc
## 6    139000  575        Typ     1Fam     CBlock
```

Adding resampling to workflow

```
house_fit_resamples <- house_wflow %>% fit_resamples(folds)
house_fit_resamples
```

```
## # Resampling results
## # 10-fold cross-validation
## # A tibble: 10 x 4
##    splits           id     .metrics         .notes
##    <list>           <chr>  <list>           <list>
##  1 <split [133/15]> Fold01 <tibble [2 x 4]> <tibble [1 x 3]>
##  2 <split [133/15]> Fold02 <tibble [2 x 4]> <tibble [1 x 3]>
##  3 <split [133/15]> Fold03 <tibble [2 x 4]> <tibble [1 x 3]>
##  4 <split [133/15]> Fold04 <tibble [2 x 4]> <tibble [1 x 3]>
##  5 <split [133/15]> Fold05 <tibble [2 x 4]> <tibble [1 x 3]>
##  6 <split [133/15]> Fold06 <tibble [2 x 4]> <tibble [1 x 3]>
##  7 <split [133/15]> Fold07 <tibble [2 x 4]> <tibble [1 x 3]>
##  8 <split [133/15]> Fold08 <tibble [2 x 4]> <tibble [1 x 3]>
##  9 <split [134/14]> Fold09 <tibble [2 x 4]> <tibble [1 x 3]>
## 10 <split [134/14]> Fold10 <tibble [2 x 4]> <tibble [1 x 3]>
##
## There were issues with some computations:
##
##   - Warning(s) x10: prediction from a rank-deficient fit may be misleading
##
## Run `show_notes(.Last.tune.result)` for more information.
```

Metrics

- Let's look at the results:

```
house_fit_resamples$.metrics[[1]]
```

```
## # A tibble: 2 x 4
##   .metric .estimator .estimate .config
##   <chr>   <chr>          <dbl> <chr>
## 1 rmse    standard   27154.    Preprocessor1_Model1
## 2 rsq     standard       0.894 Preprocessor1_Model1
```

```
house_fit_resamples$.metrics[[2]]
```

```
## # A tibble: 2 x 4
##   .metric .estimator .estimate .config
##   <chr>   <chr>          <dbl> <chr>
## 1 rmse    standard   24092.    Preprocessor1_Model1
## 2 rsq     standard       0.932 Preprocessor1_Model1
```

```
house_fit_resamples$.metrics[[3]]
```

```
## # A tibble: 2 x 4
##   .metric .estimator .estimate .config
##   <chr>   <chr>          <dbl> <chr>
## 1 rmse    standard   31683.    Preprocessor1_Model1
## 2 rsq     standard       0.899 Preprocessor1_Model1
```

CV Performance

- How do the models do overall?

```
#Baseline
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   30658.
## 2 rsq     standard       0.813
```

CV Performance

- How do the models do overall?

```
#Baseline
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   30658.
## 2 rsq     standard       0.813
```

- Cross-validation:

```
collect_metrics(house_fit_resamples)
```

```
## # A tibble: 2 x 6
##   .metric .estimator      mean     n   std_err .config
##   <chr>   <chr>          <dbl> <int>     <dbl> <chr>
## 1 rmse    standard   27350.       10 1669.     Preprocessor1_Model1
## 2 rsq     standard       0.878    10    0.0187 Preprocessor1_Model1
```

Section 5

Tuning Hyperparameters

Building a LASSO model

- The linear model did fine. But can we improve our results using penalized regression?

Building a LASSO model

- The linear model did fine. But can we improve our results using penalized regression?
  - Note that our data pre-processing recipe `house_rec` is still valid (although we could change it)

Building a LASSO model

- The linear model did fine. But can we improve our results using penalized regression?
  - Note that our data pre-processing recipe `house_rec` is still valid (although we could change it)
- If we wanted a LASSO model with particular penalty (say $\lambda = 4$) we could use

```
house_lasso_mod <- linear_reg(penalty =4 ) %>% set_engine("glmnet")
```

Building a LASSO model

- The linear model did fine. But can we improve our results using penalized regression?
  - Note that our data pre-processing recipe `house_rec` is still valid (although we could change it)
- If we wanted a LASSO model with particular penalty (say $\lambda = 4$) we could use

```
house_lasso_mod <- linear_reg(penalty =4 ) %>% set_engine("glmnet")
```

- But we are really interested in finding the **BEST** value of λ. So instead

```
house_lasso_mod <- linear_reg(penalty = tune() ) %>% set_engine("glmnet")
```

Building a LASSO model

- The linear model did fine. But can we improve our results using penalized regression?
  - Note that our data pre-processing recipe `house_rec` is still valid (although we could change it)
- If we wanted a LASSO model with particular penalty (say $\lambda = 4$) we could use

```
house_lasso_mod <- linear_reg(penalty =4 ) %>% set_engine("glmnet")
```

- But we are really interested in finding the **BEST** value of λ. So instead

```
house_lasso_mod <- linear_reg(penalty = tune() ) %>% set_engine("glmnet")
```

- Let's fit the model and tune

```
lasso_grid <- grid_regular(penalty() %>% range_set(c(-4,4)), levels = 10)
lasso_wf <- workflow() %>% add_model(house_lasso_mod) %>% add_recipe(house_rec)
lasso_res <- lasso_wf %>% tune_grid(grid = lasso_grid, resamples = folds)
```

Results

```r
collect_metrics(lasso_res)
```

```
## # A tibble: 20 x 7
##          penalty .metric .estimator      mean     n   std_err .config
##            <dbl> <chr>   <chr>          <dbl> <int>     <dbl> <chr>
##  1      0.0001   rmse    standard   27066.       10 1622.     Preprocessor1_Mode~
##  2      0.0001   rsq     standard       0.881    10    0.0179 Preprocessor1_Mode~
##  3      0.000774 rmse    standard   27066.       10 1622.     Preprocessor1_Mode~
##  4      0.000774 rsq     standard       0.881    10    0.0179 Preprocessor1_Mode~
##  5      0.00599  rmse    standard   27066.       10 1622.     Preprocessor1_Mode~
##  6      0.00599  rsq     standard       0.881    10    0.0179 Preprocessor1_Mode~
##  7      0.0464   rmse    standard   27066.       10 1622.     Preprocessor1_Mode~
##  8      0.0464   rsq     standard       0.881    10    0.0179 Preprocessor1_Mode~
##  9      0.359    rmse    standard   27066.       10 1622.     Preprocessor1_Mode~
## 10      0.359    rsq     standard       0.881    10    0.0179 Preprocessor1_Mode~
## 11      2.78     rmse    standard   27066.       10 1622.     Preprocessor1_Mode~
## 12      2.78     rsq     standard       0.881    10    0.0179 Preprocessor1_Mode~
## 13     21.5      rmse    standard   27066.       10 1622.     Preprocessor1_Mode~
## 14     21.5      rsq     standard       0.881    10    0.0179 Preprocessor1_Mode~
## 15    167.       rmse    standard   26669.       10 1619.     Preprocessor1_Mode~
## 16    167.       rsq     standard       0.883    10    0.0165 Preprocessor1_Mode~
## 17   1292.       rmse    standard   26728.       10 2117.     Preprocessor1_Mode~
## 18   1292.       rsq     standard       0.880    10    0.0148 Preprocessor1_Mode~
## 19  10000        rmse    standard   32100.       10 2500.     Preprocessor1_Mode~
## 20  10000        rsq     standard       0.845    10    0.0209 Preprocessor1_Mode~
```

Results

```
collect_metrics(lasso_res)
```

```
## # A tibble: 20 x 7
##         penalty .metric .estimator     mean     n   std_err .config
##           <dbl> <chr>   <chr>         <dbl> <int>     <dbl> <chr>
##  1      0.0001  rmse    standard   27066.      10 1622.     Preprocessor1_Mode~
##  2      0.0001  rsq     standard       0.881   10    0.0179 Preprocessor1_Mode~
##  3      0.000774 rmse   standard   27066.      10 1622.     Preprocessor1_Mode~
##  4      0.000774 rsq    standard       0.881   10    0.0179 Preprocessor1_Mode~
##  5      0.00599 rmse    standard   27066.      10 1622.     Preprocessor1_Mode~
##  6      0.00599 rsq     standard       0.881   10    0.0179 Preprocessor1_Mode~
##  7      0.0464  rmse    standard   27066.      10 1622.     Preprocessor1_Mode~
##  8      0.0464  rsq     standard       0.881   10    0.0179 Preprocessor1_Mode~
##  9      0.359   rmse    standard   27066.      10 1622.     Preprocessor1_Mode~
## 10      0.359   rsq     standard       0.881   10    0.0179 Preprocessor1_Mode~
## 11      2.78    rmse    standard   27066.      10 1622.     Preprocessor1_Mode~
## 12      2.78    rsq     standard       0.881   10    0.0179 Preprocessor1_Mode~
## 13     21.5     rmse    standard   27066.      10 1622.     Preprocessor1_Mode~
## 14     21.5     rsq     standard       0.881   10    0.0179 Preprocessor1_Mode~
## 15    167.      rmse    standard   26669.      10 1619.     Preprocessor1_Mode~
## 16    167.      rsq     standard       0.883   10    0.0165 Preprocessor1_Mode~
## 17   1292.      rmse    standard   26728.      10 2117.     Preprocessor1_Mode~
## 18   1292.      rsq     standard       0.880   10    0.0148 Preprocessor1_Mode~
## 19  10000       rmse    standard   32100.      10 2500.     Preprocessor1_Mode~
## 20  10000       rsq     standard       0.845   10    0.0209 Preprocessor1_Mode~
```

Which penalties?

- Focus just on optimal penalties for rmse:

```
lasso_res %>%
  show_best(metric = "rmse")
```

```
## # A tibble: 5 x 7
##       penalty .metric .estimator   mean     n std_err .config
##         <dbl> <chr>   <chr>       <dbl> <int>   <dbl> <chr>
## 1  167.       rmse    standard   26669.    10   1619. Preprocessor1_Model08
## 2 1292.       rmse    standard   26728.    10   2117. Preprocessor1_Model09
## 3    0.0001   rmse    standard   27066.    10   1622. Preprocessor1_Model01
## 4    0.000774 rmse    standard   27066.    10   1622. Preprocessor1_Model02
## 5    0.00599  rmse    standard   27066.    10   1622. Preprocessor1_Model03
```

Which penalties?

- Focus just on optimal penalties for rmse:

```
lasso_res %>%
  show_best(metric = "rmse")
```

```
## # A tibble: 5 x 7
##       penalty .metric .estimator   mean     n std_err .config
##         <dbl> <chr>   <chr>       <dbl> <int>   <dbl> <chr>
## 1  167.        rmse    standard   26669.    10   1619. Preprocessor1_Model08
## 2 1292.        rmse    standard   26728.    10   2117. Preprocessor1_Model09
## 3    0.0001    rmse    standard   27066.    10   1622. Preprocessor1_Model01
## 4    0.000774 rmse    standard   27066.    10   1622. Preprocessor1_Model02
## 5    0.00599  rmse    standard   27066.    10   1622. Preprocessor1_Model03
```

- Let's choose the best model:

```
best_lasso <- lasso_res %>% select_best(metric = "rmse")
best_lasso
```

```
## # A tibble: 1 x 2
##   penalty .config
##     <dbl> <chr>
## 1    167. Preprocessor1_Model08
```

Finalize the model

- We update or finalize our workflow with the values from `select_best`:

```
final_lasso_wf <- lasso_wf %>% finalize_workflow(best_lasso)
final_lasso_wf
```

```
## == Workflow ==================================================================
## Preprocessor: Recipe
## Model: linear_reg()
##
## -- Preprocessor --------------------------------------------------------------
## 5 Recipe Steps
##
## * step_log()
## * step_mutate()
## * step_rm()
## * step_novel()
## * step_dummy()
##
## -- Model ---------------------------------------------------------------------
## Linear Regression Model Specification (regression)
##
## Main Arguments:
##   penalty = 166.810053720006
##
## Computational engine: glmnet
```

Fit the Best Model

- Thus far, we've just focused on finding the best parameter. But we haven't actually built a LASSO model on training data, nor predicted on the test data. Let's do just that:

Fit the Best Model

- Thus far, we've just focused on finding the best parameter. But we haven't actually built a LASSO model on training data, nor predicted on the test data. Let's do just that:

```
final_lasso_fit<-final_lasso_wf %>% last_fit(data_split )
final_lasso_fit$.metrics[[1]]
```

```
## # A tibble: 2 x 4
##   .metric .estimator .estimate .config
##   <chr>   <chr>          <dbl> <chr>
## 1 rmse    standard   28961.    Preprocessor1_Model1
## 2 rsq     standard       0.823 Preprocessor1_Model1
```

```
final_lasso_fit$.predictions[[1]] %>% head()
```

```
## # A tibble: 6 x 4
##     .pred  .row SalePrice .config
##     <dbl> <int>     <int> <chr>
## 1 187702.     1    181500 Preprocessor1_Model1
## 2 256420.     3    200000 Preprocessor1_Model1
## 3 182292.     4    149000 Preprocessor1_Model1
## 4 160247.     5    154000 Preprocessor1_Model1
## 5 262237.     7    306000 Preprocessor1_Model1
## 6 234814.     9    177000 Preprocessor1_Model1
```