Tidymodels

Nate Wells

Math 243: Stat Learning

November 30th, 2020

Outline

In today's class, we will. . .

- Discuss the `tidymodels` packages for model building in the `tidyverse` framework
- Implement PCA in R and interpret PCA in context

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

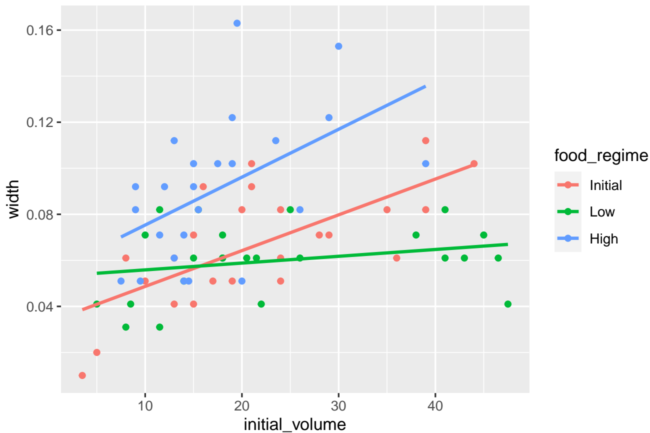
The Data

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

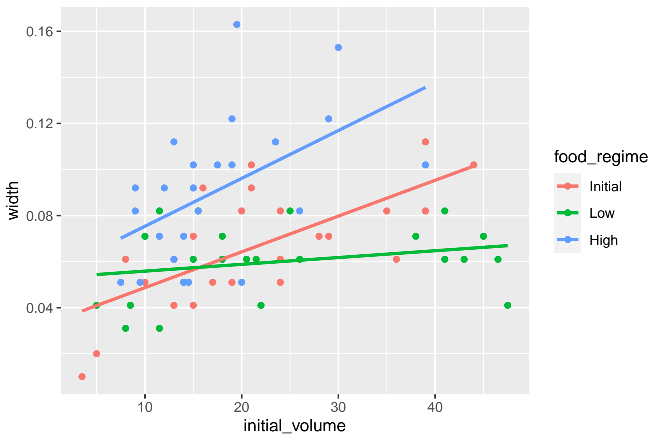
```
sea_urchins<-read_csv("https://tidymodels.org/start/models/urchins.csv") %>%
  setNames(c("food_regime", "initial_volume", "width")) %>%
  mutate(food_regime = factor(food_regime, levels = c("Initial", "Low", "High")))
head(sea_urchins)
```

```
## # 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



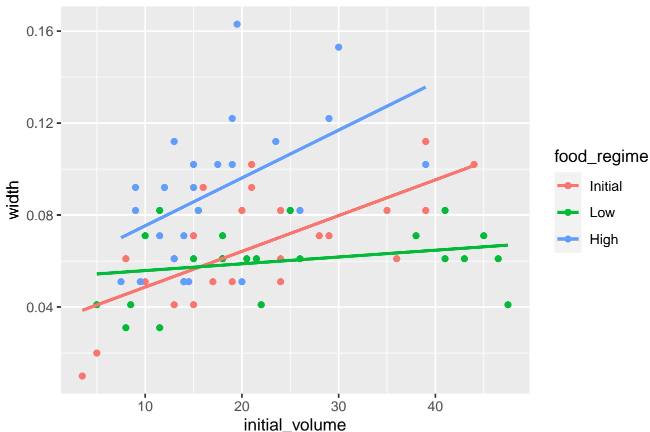
Scatterplot



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:  3ms
##
## 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:

```
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

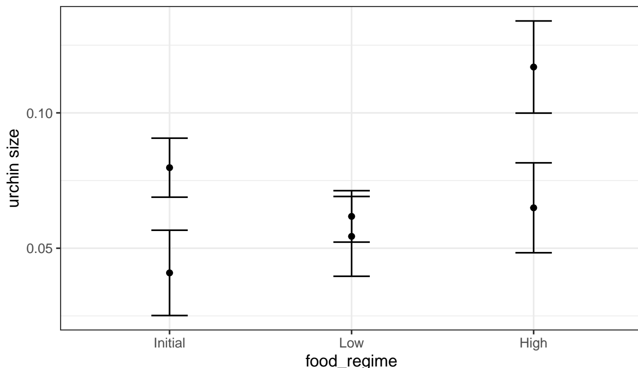
Because the result of `predict()` is tidy, we can easily combine it with the original data:

```
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")
```

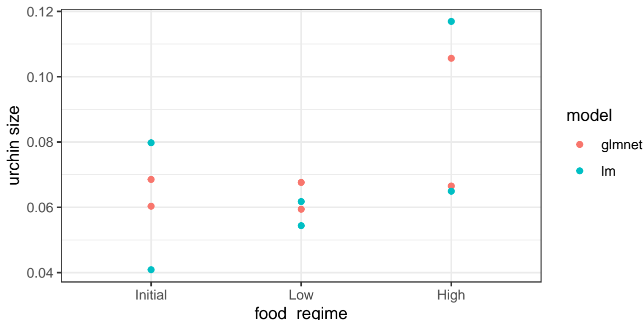
```
glmnet_fit <- glmnet_mod %>% fit(width ~ initial_volume*food_regime,
                                 data = sea_urchins)
tidy(glmnet_fit, penalty = .004) #penalty selects particular value of lambda
```

```
## # 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

NYCFlights

The `flight_data` data contains information on over 300,000 flights departning near New York City in 2013. We'll use it to predict whether a plane will arrive more than 30 minutes late.

```
## # A tibble: 6 x 10
##   dep_time flight origin dest  air_time distance carrier date       arr_delay
##      <int>  <int> <fct>  <fct>    <dbl>    <dbl> <fct>   <date>     <fct>
## 1      517   1545 EWR    IAH        227     1400 UA      2013-01-01 on_time
## 2      533   1714 LGA    IAH        227     1416 UA      2013-01-01 on_time
## 3      542   1141 JFK    MIA        160     1089 AA      2013-01-01 late
## 4      544    725 JFK    BQN        183     1576 B6      2013-01-01 on_time
## 5      554    461 LGA    ATL        116      762 DL      2013-01-01 on_time
## 6      554   1696 EWR    ORD        150      719 UA      2013-01-01 on_time
## # ... with 1 more variable: time_hour <dttm>
```

NYCFlights

The `flight_data` data contains information on over 300,000 flights departning near New York City in 2013. We'll use it to predict whether a plane will arrive more than 30 minutes late.

```
## # A tibble: 6 x 10
##   dep_time flight origin dest  air_time distance carrier date       arr_delay
##      <int>  <int> <fct>  <fct>    <dbl>    <dbl> <fct>   <date>     <fct>
## 1      517   1545 EWR    IAH        227     1400 UA      2013-01-01 on_time
## 2      533   1714 LGA    IAH        227     1416 UA      2013-01-01 on_time
## 3      542   1141 JFK    MIA        160     1089 AA      2013-01-01 late
## 4      544    725 JFK    BQN        183     1576 B6      2013-01-01 on_time
## 5      554    461 LGA    ATL        116      762 DL      2013-01-01 on_time
## 6      554   1696 EWR    ORD        150      719 UA      2013-01-01 on_time
## # ... with 1 more variable: time_hour <dttm>
```

```
flight_data %>%
  count(arr_delay) %>%
  mutate(prop = n/sum(n))
```

```
## # A tibble: 2 x 3
##   arr_delay      n  prop
##   <fct>      <int> <dbl>
## 1 late       52540 0.161
## 2 on_time   273279 0.839
```

Investigate Predictors

Look at the list of variables:

```
names(flight_data)
```

```
##  [1] "dep_time"  "flight"    "origin"    "dest"      "air_time"  "distance"
##  [7] "carrier"   "date"      "arr_delay" "time_hour"
```

Investigate Predictors

Look at the list of variables:

```
names(flight_data)
```

```
##  [1] "dep_time"  "flight"    "origin"    "dest"      "air_time"  "distance"
##  [7] "carrier"   "date"      "arr_delay" "time_hour"
```

- Note that there are two variables `flight` and `time_hour` which are not useful as predictors, but are worth keeping for identification purposes.

Investigate Predictors

Additionally, note that `dest` and `carrier` are factor variables, so should be converted to a collection of dummy variables

```
library(skimr)
flight_data %>% skim(dest, carrier)
```

Table 7: Data summary

Name	Piped data
Number of rows	325819
Number of columns	10
_______________________	
Column type frequency:	
factor	2
________________________	
Group variables	None

Variable type: factor

skim_variable	n_missing	complete_rate	ordered	n_unique	top_counts
dest	0	1	FALSE	104	ATL: 16771, ORD: 16507, LAX: 15942, BOS: 14948
carrier	0	1	FALSE	16	UA: 57489, B6: 53715, EV: 50868, DL: 47465

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(flight_data , 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)
flights_rec <-
  recipe(arr_delay ~ ., data = train_data) %>%
  update_role(flight, time_hour, new_role = "ID")
```

Create a recipe and update roles

We now create a recipe for some data pre-processing

```
library(recipes)
flights_rec <-
  recipe(arr_delay ~ ., data = train_data) %>%
  update_role(flight, time_hour, new_role = "ID")
```

```
summary(flights_rec) %>% kable()
```

variable	type	role	source
dep_time	numeric	predictor	original
flight	numeric	ID	original
origin	nominal	predictor	original
dest	nominal	predictor	original
air_time	numeric	predictor	original
distance	numeric	predictor	original
carrier	nominal	predictor	original
date	date	predictor	original
time_hour	date	ID	original
arr_delay	nominal	outcome	original

Add steps to recipes

Will flight date effect chance of late arrival?

Add steps to recipes

Will flight date effect chance of late arrival?

```
## # A tibble: 6 x 2
##   date       numeric_date
##   <date>            <dbl>
## 1 2013-01-01        15706
## 2 2013-01-02        15707
## 3 2013-01-03        15708
## 4 2013-01-04        15709
## 5 2013-01-05        15710
## 6 2013-01-06        15711
```

Add steps to recipes

Will flight date effect chance of late arrival?

```
## # A tibble: 6 x 2
##   date       numeric_date
##   <date>            <dbl>
## 1 2013-01-01        15706
## 2 2013-01-02        15707
## 3 2013-01-03        15708
## 4 2013-01-04        15709
## 5 2013-01-05        15710
## 6 2013-01-06        15711
```

But its possible that predictors derived from the date will be more beneficial (day of the week, month, holiday)

Add steps to recipes

Will flight date effect chance of late arrival?

```
## # A tibble: 6 x 2
##   date       numeric_date
##   <date>            <dbl>
## 1 2013-01-01        15706
## 2 2013-01-02        15707
## 3 2013-01-03        15708
## 4 2013-01-04        15709
## 5 2013-01-05        15710
## 6 2013-01-06        15711
```

But its possible that predictors derived from the date will be more beneficial (day of the week, month, holiday)

One of the chief benefits of recipes is that they are easy to add to:

```
flights_rec <- flights_rec %>%
  step_date(date, features = c("dow", "month")) %>%
  step_holiday(date, holidays = timeDate::listHolidays("US")) %>%
  step_rm(date)
```

Add steps to recipes

Will flight date effect chance of late arrival?

```
## # A tibble: 6 x 2
##   date       numeric_date
##   <date>            <dbl>
## 1 2013-01-01        15706
## 2 2013-01-02        15707
## 3 2013-01-03        15708
## 4 2013-01-04        15709
## 5 2013-01-05        15710
## 6 2013-01-06        15711
```

But its possible that predictors derived from the date will be more beneficial (day of the week, month, holiday)

One of the chief benefits of recipes is that they are easy to add to:

```
flights_rec <- flights_rec %>%
  step_date(date, features = c("dow", "month")) %>%
  step_holiday(date, holidays = timeDate::listHolidays("US")) %>%
  step_rm(date)
```

- What did each of these verbs do?

Create Dummy Variables

Recall that `dest` and `carrier` are factor variables. Additionally, the newly created `date_dow` and `date_month` variables are factors as well. To create appropriate dummy variables:

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

Create Dummy Variables

Recall that `dest` and `carrier` are factor variables. Additionally, the newly created `date_dow` and `date_month` variables are factors as well. To create appropriate dummy variables:

```
flights_rec <- flights_rec %>% step_dummy(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 `dest` and `carrier` are factor variables. Additionally, the newly created `date_dow` and `date_month` variables are factors as well. To create appropriate dummy variables:

```
flights_rec <- flights_rec %>% step_dummy(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

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

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

Create Dummy Variables

Recall that `dest` and `carrier` are factor variables. Additionally, the newly created `date_dow` and `date_month` variables are factors as well. To create appropriate dummy variables:

```
flights_rec <- flights_rec %>% step_dummy(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

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

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

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