04 - A Typical (Supervised) ML Workflow

ml4econ, HUJI 2023

Itamar Caspi

April 30, 2023 (updated: 2023-05-01)

Packages and setup

We will use the following packages during the presentation:

```r
library(tidyverse)   # for data wrangling and visualization
library(tidymodels)  # for data modeling
library(GGally)      # for pairs plot
library(skimr)       # for summary statistics
library(here)        # for referencing folders and files
```

For the presentation, we will select a specific `ggplot` theme (not relevant otherwise):

```r
theme_set(theme_grey(20))
```

The `tidymodels` package

> "`tidymodels` is a "meta-package" for modeling and statistical analysis that share the underlying design philosophy, grammar, and data structures of the tidyverse."

Supervised Machine Learning Workflow

1. Define the Prediction Task
2. Explore the Data
3. Set Model and Tuning Parameters
4. Perform Cross-Validation
5. Evaluate the Model

Step 1: Define the Prediction Task

Welcome to the `BostonHousing` dataset

- Dataset: 506 census tracts from the 1970 Boston census (Harrison & Rubinfeld, 1978)

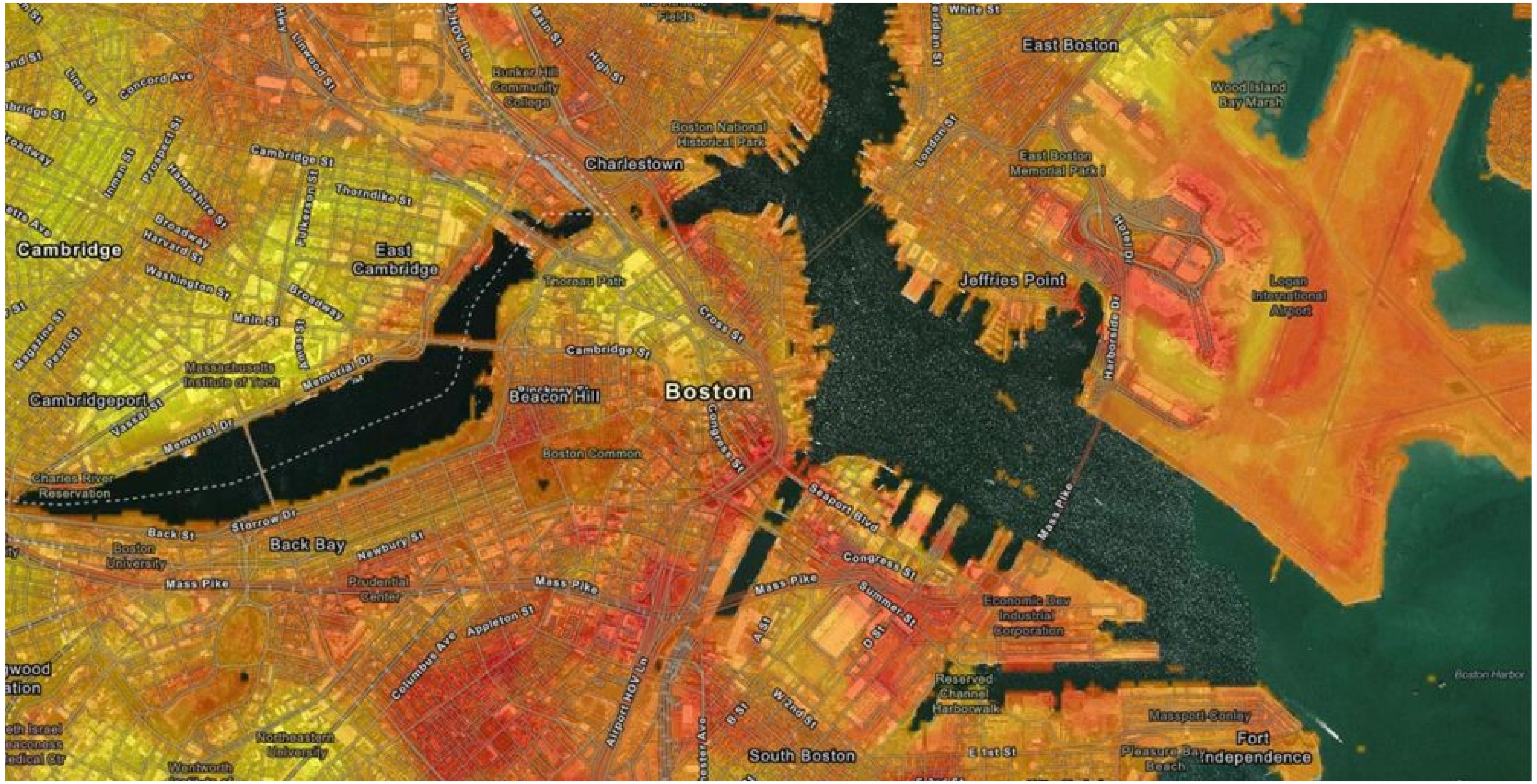
Components:

- `medv` (target): Median home value in thousands of dollars
- `lstat` (predictor): Percentage of lower status population
- `chas` (predictor): Proximity to Charles River (1 = yes, 0 = no)

Objective: Predict `medv` based on the given predictors

Source: https://www.bostonusa.com/

A bird's-eye view of Boston



Source: https://www.wbur.org/news/2019/11/25/heat-mapping-boston-museum-of-science

Load the Data

We will utilize the `read_csv()` function to import the raw dataset.

```
boston_raw <- read_csv(here("04-ml-workflow/data","BostonHousing.csv"))
```

What Type of Data?

For a better understanding of the data structure, apply the `glimpse()` function:

```
glimpse(boston_raw)
```

```
## Rows: 506
## Columns: 14
## $ crim    <dbl> 0.00632, 0.02731, 0.02729, 0.03237, 0.06905, 0.02985, 0.08829, 0.14455, 0.21124, 0.17~
## $ zn      <dbl> 18.0, 0.0, 0.0, 0.0, 0.0, 0.0, 12.5, 12.5, 12.5, 12.5, 12.5, 12.5, 12.5, 0.0, 0.0, 0.~
## $ indus   <dbl> 2.31, 7.07, 7.07, 2.18, 2.18, 2.18, 7.87, 7.87, 7.87, 7.87, 7.87, 7.87, 7.87, 8.14, 8~
## $ chas    <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0~
## $ nox     <dbl> 0.538, 0.469, 0.469, 0.458, 0.458, 0.458, 0.524, 0.524, 0.524, 0.524, 0.524, 0.524, 0~
## $ rm      <dbl> 6.575, 6.421, 7.185, 6.998, 7.147, 6.430, 6.012, 6.172, 5.631, 6.004, 6.377, 6.009, 5~
## $ age     <dbl> 65.2, 78.9, 61.1, 45.8, 54.2, 58.7, 66.6, 96.1, 100.0, 85.9, 94.3, 82.9, 39.0, 61.8, ~
## $ dis     <dbl> 4.0900, 4.9671, 4.9671, 6.0622, 6.0622, 6.0622, 5.5605, 5.9505, 6.0821, 6.5921, 6.346~
## $ rad     <dbl> 1, 2, 2, 3, 3, 3, 5, 5, 5, 5, 5, 5, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4~
## $ tax     <dbl> 296, 242, 242, 222, 222, 222, 311, 311, 311, 311, 311, 311, 311, 307, 307, 307, 307, ~
## $ ptratio <dbl> 15.3, 17.8, 17.8, 18.7, 18.7, 18.7, 15.2, 15.2, 15.2, 15.2, 15.2, 15.2, 15.2, 21.0, 2~
## $ b       <dbl> 396.90, 396.90, 392.83, 394.63, 396.90, 394.12, 395.60, 396.90, 386.63, 386.71, 392.5~
## $ lstat   <dbl> 4.98, 9.14, 4.03, 2.94, 5.33, 5.21, 12.43, 19.15, 29.93, 17.10, 20.45, 13.27, 15.71, ~
## $ medv    <dbl> 24.0, 21.6, 34.7, 33.4, 36.2, 28.7, 22.9, 27.1, 16.5, 18.9, 15.0, 18.9, 21.7, 20.4, 1~
```

The `chas` variable predominantly consists of zeros, which implies that it should be treated as a categorical factor.

Initial Data Filtering

Select `medv` and `lstat`

```
boston <- boston_raw %>%
  as_tibble() %>%
  select(medv, lstat, chas) %>%
  mutate(chas = as_factor(chas))

head(boston)
```

```
## # A tibble: 6 x 3
##    medv lstat chas
##   <dbl> <dbl> <fct>
## 1  24    4.98 0
## 2  21.6  9.14 0
## 3  34.7  4.03 0
## 4  33.4  2.94 0
## 5  36.2  5.33 0
## 6  28.7  5.21 0
```

Step 2: Split the Data

Initial Split

To perform an initial train-test split, we will use the `initial_split()`, `training()`, and `testing()` functions from the rsample package.

Remember to set a seed for reproducibility.

```
set.seed(1203)
```

Initial split:

```
boston_split <- boston %>%
  initial_split(prop = 2/3, strata = medv)

boston_split
```

```
## <Training/Testing/Total>
## <336/170/506>
```

Preparing Training and Test Sets

```r
boston_train_raw <- training(boston_split)
boston_test_raw  <- testing(boston_split)

head(boston_train_raw, 5)
```

```
## # A tibble: 5 x 3
##    medv lstat chas
##   <dbl> <dbl> <fct>
## 1  16.5  29.9 0
## 2  15    20.4 0
## 3  13.6  21.0 0
## 4  15.2  18.7 0
## 5  14.5  19.9 0
```

```r
head(boston_test_raw, 5)
```

```
## # A tibble: 5 x 3
##    medv lstat chas
##   <dbl> <dbl> <fct>
## 1  24     4.98 0
## 2  21.6   9.14 0
## 3  27.1 19.2   0
## 4  18.9 17.1   0
## 5  18.2 10.3   0
```

Step 3: Explore the Data

Summary Statistics Using `skimr`

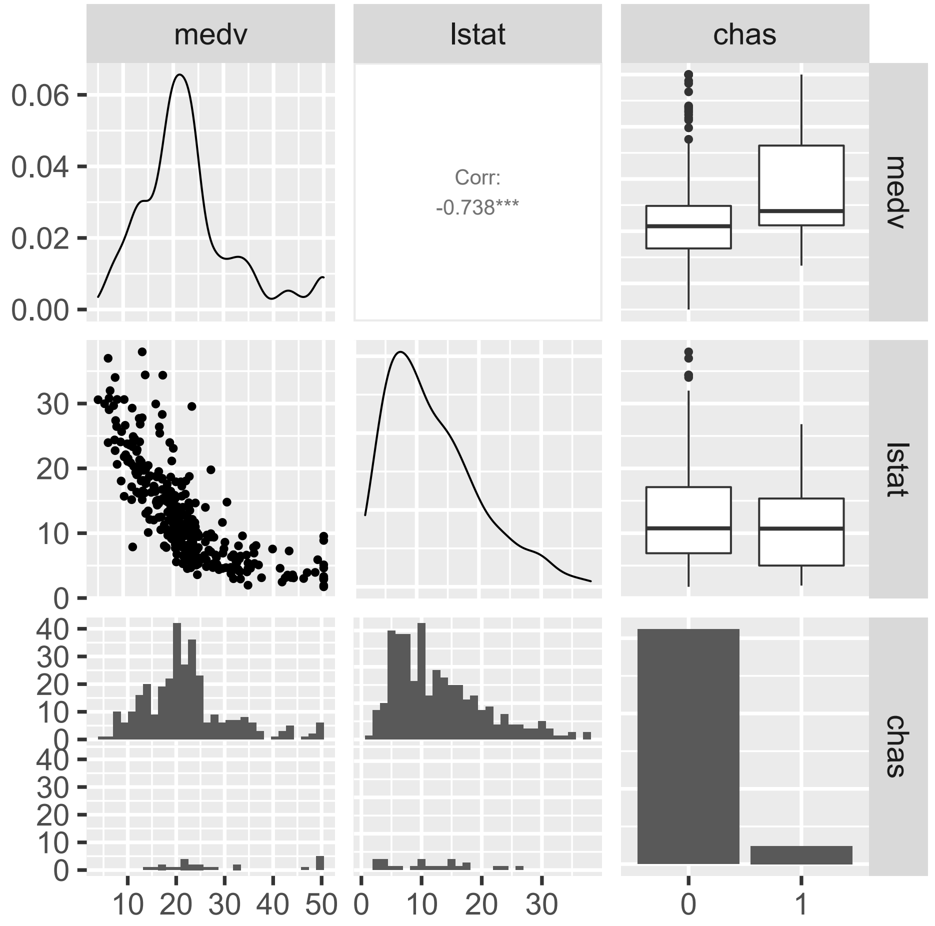
```r
boston_train_raw %>%
  skim()
```

(Not visually appealing on the slides)

Pairs Plot Using `GGally`

We will now create a **pairs plot**, which efficiently displays every variable in a dataset against all the others.

```
boston_train_raw %>% ggpairs()
```



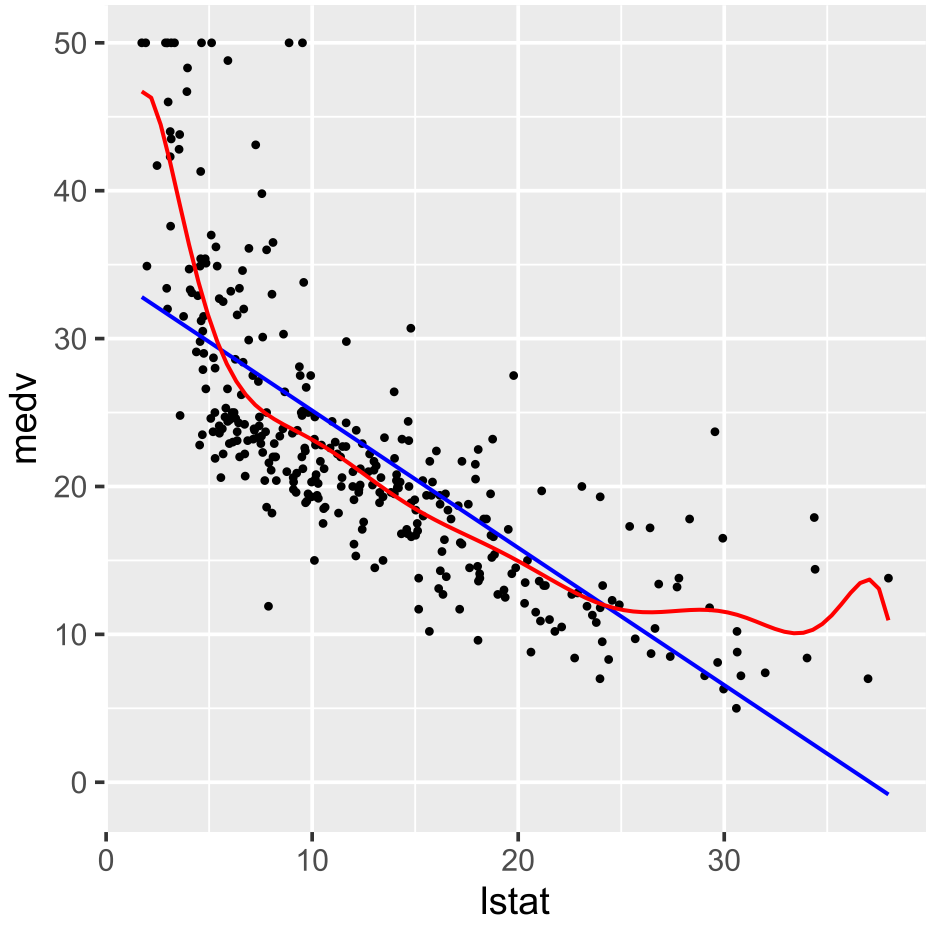
Select a Model

We will select the class of polynomial models, represented as follows:

$$medv_i = \beta_0 + \sum_{j=1}^{\lambda} \beta_j lstat_i^j + \varepsilon_i$$

```r
boston_train_raw %>% ggplot(aes(lstat, medv)) +
  geom_point() +
  geom_smooth(
    method = lm,
    formula = y ~ poly(x,1),
    se = FALSE,
    color = "blue"
  ) +
  geom_smooth(
    method = lm,
    formula = y ~ poly(x,10),
    se = FALSE,
    color = "red"
  )
```

In blue $\lambda = 1$; in red, $\lambda = 10$.



Step 4: Set Model and Tuning Parameters

Data Preprocessing using `recipes`

The `recipes` package is an excellent resource for data preprocessing, seamlessly integrating with the tidy approach to machine learning.

```
boston_rec <-
  recipe(medv ~ lstat + chas, data = boston_train_raw) %>%
  step_poly(lstat, degree = tune("lambda")) %>%
  step_dummy(chas)

boston_rec
```

```
## Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          2
##
## Operations:
##
## Orthogonal polynomials on lstat
## Dummy variables from chas
```

Set a Grid for λ

What are the tuning parameters we need to consider?

```
boston_rec %>% extract_parameter_set_dials()
```

```
## Collection of 1 parameters for tuning
##
##  identifier   type    object
##      lambda degree nparam[+]
```

We must tune the polynomial degree parameter (λ) while constructing our models using the training data. In this example, we will establish a range between 1 and 8:

```
lambda_grid <- expand_grid("lambda" = 1:8)
```

Define the Model

Using the linear regression model:

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

lm_mod
```

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

Note that in this case, there are no tuning parameters involved.

Step 5: Cross-validation

Split the Training Set to 5-folds

We will apply the `vfold_cv()` function from the rsample package to divide the training set into 5-folds:

```
cv_splits <- boston_train_raw %>%
  vfold_cv(v = 5)

cv_splits
```

```
## #  5-fold cross-validation
## # A tibble: 5 x 2
##   splits           id
##   <list>           <chr>
## 1 <split [268/68]> Fold1
## 2 <split [269/67]> Fold2
## 3 <split [269/67]> Fold3
## 4 <split [269/67]> Fold4
## 5 <split [269/67]> Fold5
```

Define the Workflow

Next, we define a `workflow()` that combines a model specification with a recipe or model preprocessor.

```
boston_wf <-
  workflow() %>%
  add_model(lm_mod) %>%
  add_recipe(boston_rec)
```

Note that in this case, there are no tuning parameters involved.

Estimate CV-RMSE Over the λ Grid

We will now calculate the cross-validated root mean squared error (CV-RMSE) for each value of λ.

```
boston_results <-
  boston_wf %>%
  tune_grid(
  resamples = cv_splits,
  grid      = lambda_grid
)

boston_results
```

```
## # Tuning results
## # 5-fold cross-validation
## # A tibble: 5 x 4
##   splits           id    .metrics          .notes
##   <list>           <chr> <list>            <list>
## 1 <split [268/68]> Fold1 <tibble [16 x 5]> <tibble [0 x 3]>
## 2 <split [269/67]> Fold2 <tibble [16 x 5]> <tibble [0 x 3]>
## 3 <split [269/67]> Fold3 <tibble [16 x 5]> <tibble [0 x 3]>
## 4 <split [269/67]> Fold4 <tibble [16 x 5]> <tibble [0 x 3]>
## 5 <split [269/67]> Fold5 <tibble [16 x 5]> <tibble [0 x 3]>
```

Find the Optimal λ

Let's identify the top-3 best-performing models.

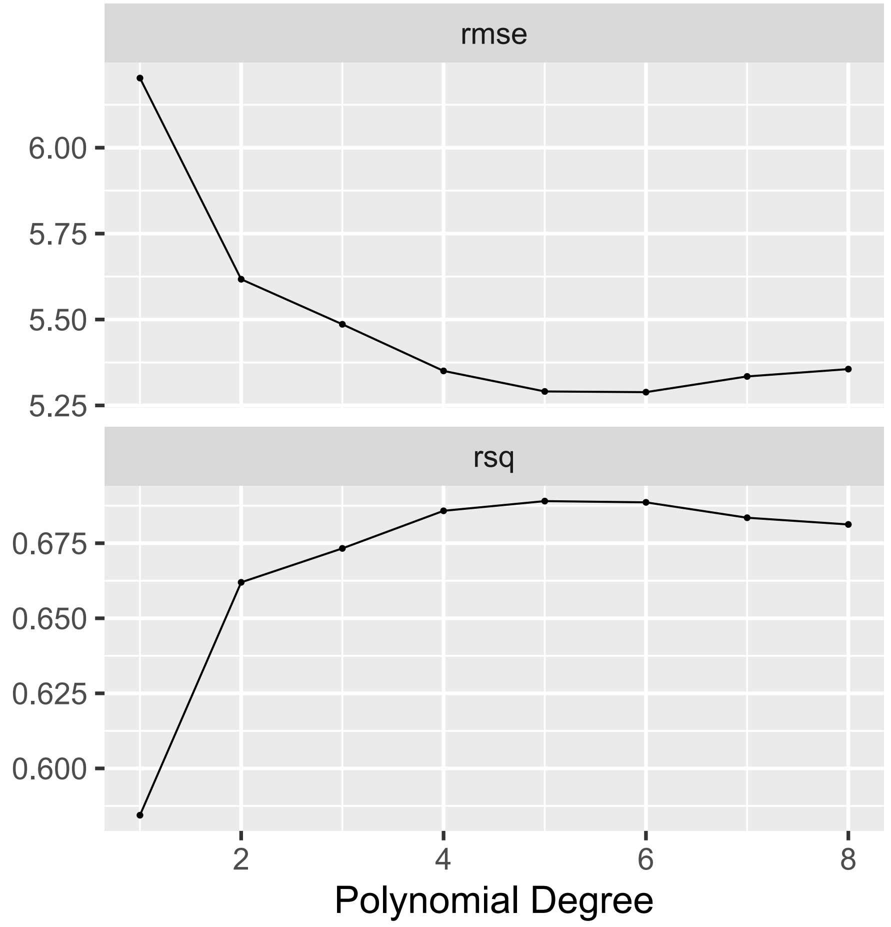
```
boston_results %>%
 show_best(metric = "rmse", n = 3)
```

```
## # A tibble: 3 x 7
##   lambda .metric .estimator  mean     n std_err .config
##    <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1      6 rmse    standard    5.29     5   0.273 Preprocessor6_Model1
## 2      5 rmse    standard    5.29     5   0.279 Preprocessor5_Model1
## 3      7 rmse    standard    5.33     5   0.293 Preprocessor7_Model1
```

> *"[I]n reality there is rarely if ever a true underlying model, and even if there was a true underlying model, selecting that model will not necessarily give the best forecasts..."*
>
> — **Rob J. Hyndman**

And Now Using a Graph

```
boston_results %>%
  autoplot()
```



Step 6: Evaluate the Model

Use the Test Set to Evaluate the Best Model

Choose the optimal value of λ

```
best_lambda <- boston_results %>%
  select_best(metric = "rmse")

best_lambda
```

```
## # A tibble: 1 x 2
##   lambda .config
##    <int> <chr>
## 1      6 Preprocessor6_Model1
```

Create a recipe using the optimal $\lambda = 6$

```
boston_final <- boston_rec %>%
  finalize_recipe(best_lambda)
```

Apply the Recipe to the Training and Test Sets

The `juice()` function applies the recipe to the training set, while the `bake()` function applies it to the test set.

```
boston_train <- boston_final %>%
  prep() %>%
  juice()

boston_test <- boston_final %>%
  prep() %>%
  bake(new_data = boston_test_raw)
```

For instance, let's examine the training set:

```
head(boston_train, 3)
```

```
## # A tibble: 3 x 8
##    medv lstat_poly_1 lstat_poly_2 lstat_poly_3 lstat_poly_4 lstat_poly_5 lstat_poly_6 chas_X1
##   <dbl>        <dbl>        <dbl>        <dbl>        <dbl>        <dbl>        <dbl>   <dbl>
## 1  16.5       0.126        0.0942      -0.0311      -0.118       -0.0932       0.0108       0
## 2  15         0.0565      -0.0399      -0.0549       0.0406       0.0604      -0.0342       0
## 3  13.6       0.0606      -0.0358      -0.0613       0.0335       0.0693      -0.0218       0
```

Fit the Model to the Training Set

Fit the optimal model (with $\lambda = 6$) to the training set:

```
boston_fit <- lm_mod %>%
  fit(medv ~ ., data = boston_train)
```

The following are the estimated coefficients:

```
boston_fit %>% tidy()
```

```
## # A tibble: 8 x 5
##   term         estimate std.error statistic   p.value
##   <chr>           <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept)     22.3      0.295    75.6   1.41e-209
## 2 lstat_poly_1  -126.       5.21    -24.2   4.33e- 75
## 3 lstat_poly_2    52.8      5.21     10.1   3.49e- 21
## 4 lstat_poly_3   -21.4      5.23     -4.09  5.36e-  5
## 5 lstat_poly_4    20.9      5.23      3.99  8.29e-  5
## 6 lstat_poly_5   -14.7      5.23     -2.80  5.34e-  3
## 7 lstat_poly_6     4.22     5.22      0.807 4.20e-  1
## 8 chas_X1          4.45     1.12      3.96  9.27e-  5
```

Make Predictions Using the Test Set

Generate a tibble that includes the predictions and the actual values:

```
boston_pred <- boston_fit %>%
  predict(new_data = boston_test) %>%
  bind_cols(boston_test) %>%
  select(medv, .pred)

head(boston_pred)
```

```
## # A tibble: 6 x 2
##    medv .pred
##   <dbl> <dbl>
## 1  24    31.3
## 2  21.6  23.3
## 3  27.1  15.0
## 4  18.9  16.7
## 5  18.2  22.2
## 6  19.9  24.0
```

It's worth noting that this is the first time we are utilizing the test set!

Test-RMSE

Calculate the root mean square error (RMSE) for the test set (test-RMSE):

```
boston_pred %>%
  rmse(medv, .pred)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard        5.00
```

The above is a measure of our model's performance on "general" data.

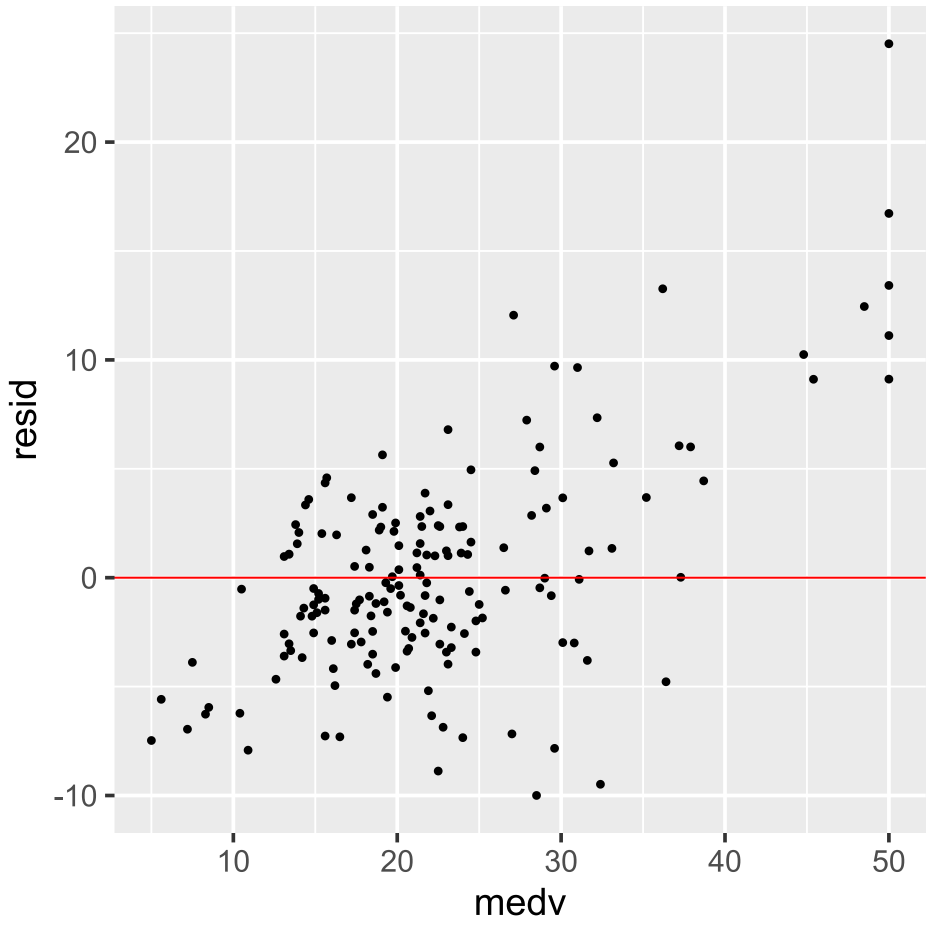
> **NOTE:** The test set RMSE estimates the predicted squared error on unseen data, provided the best model.

Always plot your prediction errors

Plotting the prediction errors $(y_i - \hat{y}_i)$ against the target variable provides critical information regarding prediction quality.

```r
boston_pred %>%
  mutate(resid = medv - .pred) %>%
  ggplot(aes(medv, resid)) +
  geom_point() +
  geom_hline(yintercept = 0, color = "red")
```

For example, our predictions for high-end levels of `medv` are highly biased, indicating that there's potential for improvement...



(A shortcut)

The `last_fit()` function from `tune` is a much quicker way to obtain the test-set RMSE.

Firstly, we need to modify our workflow to utilize the optimal λ value.

```
boston_wf <-
  workflow() %>%
  add_model(lm_mod) %>%
  add_recipe(boston_final)
```

We will now use the optimal model to estimate the out-of-sample RMSE.

```
boston_wf %>%
  last_fit(split = boston_split) %>%
  collect_metrics() %>%
  filter(.metric == "rmse")
```

```
## # A tibble: 1 x 4
##   .metric .estimator .estimate .config
##   <chr>   <chr>          <dbl> <chr>
## 1 rmse    standard        5.00 Preprocessor1_Model1
```

`slides::end()`

Source code