# Train and Evaluate Regression Models using Tidymodels

Eric Wanjau

## Regression Challenge

Predicting the selling price of a residential property depends on a number of factors, including the property age, availability of local amenities, and location.

In this challenge, you will use a dataset of real estate sales transactions to predict the price-per-unit of a property based on its features. The price-per-unit in this data is based on a unit measurement of 3.3 square meters.

> **Citation**: The data used in this exercise originates from the following study:
>
> *Yeh, I. C., & Hsu, T. K. (2018). Building real estate valuation models with comparative approach through case-based reasoning. Applied Soft Computing, 65, 260-271.*
>
> It was obtained from the UCI dataset repository (Dua, D. and Graff, C. (2019). UCI Machine Learning Repository. Irvine, CA: University of California, School of Information and Computer Science).

## Review the data

Let's hit the ground running by importing the data and viewing the first few rows.

```r
# Load the core tidyverse and tidymodels in your current R session
suppressPackageStartupMessages({
 library(tidyverse)
 library(tidymodels)
})

# Read the csv file into a tibble
estate_data <- read_csv(file = "https://raw.githubusercontent.com/MicrosoftDocs/ml-basics/master/challen

# Print the first 10 rows of the data
estate_data %>%
 slice_head(n = 10)
```

```
## # A tibble: 10 x 7
##    transaction_date house_age transit_distance local_convenience_stores latitude
##               <dbl>     <dbl>            <dbl>                    <dbl>    <dbl>
##  1            2013.      32               84.9                       10     25.0
##  2            2013.      19.5            307.                         9     25.0
##  3            2014.      13.3            562.                         5     25.0
##  4            2014.      13.3            562.                         5     25.0
##  5            2013.       5              391.                         5     25.0
##  6            2013.       7.1           2175.                         3     25.0
```

```
##  7              2013.      34.5             623.                             7     25.0
##  8              2013.      20.3             288.                             6     25.0
##  9              2014.      31.7            5512.                             1     25.0
## 10              2013.      17.9            1783.                             3     25.0
## # ... with 2 more variables: longitude <dbl>, price_per_unit <dbl>
```

The data consists of the following variables:

- **transaction_date** - the transaction date (for example, 2013.250=2013 March, 2013.500=2013 June, etc.)
- **house_age** - the house age (in years)
- **transit_distance** - the distance to the nearest light rail station (in meters)
- **local_convenience_stores** - the number of convenience stores within walking distance
- **latitude** - the geographic coordinate, latitude
- **longitude** - the geographic coordinate, longitude
- **price_per_unit** house price of unit area (3.3 square meters)

## Train a Regression Model

Your challenge is to explore and prepare the data, identify predictive features that will help predict the **price_per_unit** label, and train a regression model that achieves the lowest **Root Mean Square Error** (RMSE) you can achieve (which must be less than **7**) when evaluated against a test subset of data.

### View the label distribution

Let's start our analysis of the data by examining a few key descriptive statistics. We can use the `summarytools::descr()` function to neatly and quickly summarize the numeric features as well as the **rentals** label column.

```r
# Load summary tools library
library(summarytools)
```

```
##
## Attaching package: 'summarytools'

## The following object is masked from 'package:tibble':
##
##     view
```

```r
# Obtain summary stats for feature and label columns
estate_data %>%
  # Summary stats
  descr(order = "preserve",
        stats = c('mean', 'sd', 'min', 'q1', 'med', 'q3', 'max'),
        round.digits = 6)
```

```
## Descriptive Statistics
## estate_data
## N: 414
##
##                  transaction_date   house_age   transit_distance   local_convenience_stores
## ------------- ------------------ ----------- ------------------ --------------------------
##          Mean        2013.148971   17.712560        1083.885689                   4.094203
##       Std.Dev           0.281967   11.392485        1262.109595                   2.945562
##           Min        2012.667000    0.000000          23.382840                   0.000000
##            Q1        2012.917000    9.000000         289.324800                   1.000000
##        Median        2013.167000   16.100000         492.231300                   4.000000
##            Q3        2013.417000   28.200000        1455.798000                   6.000000
##           Max        2013.583000   43.800000        6488.021000                  10.000000
##
## Table: Table continues below
##
##
##
##                   latitude    longitude   price_per_unit
## ------------- ----------- ------------ ----------------
##          Mean   24.969030   121.533361        37.980193
##       Std.Dev    0.012410     0.015347        13.606488
##           Min   24.932070   121.473530         7.600000
##            Q1   24.962990   121.527600        27.700000
##        Median   24.971100   121.538630        38.450000
##            Q3   24.977460   121.543310        46.600000
##           Max   25.014590   121.566270       117.500000
```

The statistics reveal some information about the distribution of the data in each of the numeric fields, including the number of observations (there are 414 records), the mean, standard deviation, minimum and maximum values, and the quantile values (the threshold values for 25%, 50% - which is also the median, and 75% of the data).

From this, we can see that the mean number of price per unit is around 38. There's a comparatively `small standard deviation`, indicating `not much variance` in the prices per unit.

We might get a clearer idea of the distribution of price values by visualizing the data.

```r
library(patchwork)

# Plot a histogram
theme_set(theme_light())

hist_plt <- estate_data %>%
  ggplot(mapping = aes(x = price_per_unit)) +
  geom_histogram(bins = 100, fill = "midnightblue", alpha = 0.7) +

  # Add lines for mean and median
  geom_vline(aes(xintercept = mean(price_per_unit), color = 'Mean'), linetype = "dashed", size = 1.3) +
  geom_vline(aes(xintercept = median(price_per_unit), color = 'Median'), linetype = "dashed", size = 1.
  xlab("") +
  ylab("Frequency") +
  scale_color_manual(name = "", values = c(Mean = "red", Median = "yellow")) +
  theme(legend.position = c(0.9, 0.9), legend.background = element_blank())
```

```
# Plot a box plot
box_plt <- estate_data %>%
  ggplot(aes(x = price_per_unit, y = 1)) +
  geom_boxplot(fill = "#E69F00", color = "gray23", alpha = 0.7) +
    # Add titles and labels
  xlab("Price_per_unit")+
  ylab("")


# Combine plots using patchwork syntax
(hist_plt / box_plt) +
  plot_annotation(title = 'Price Distribution',
                  theme = theme(
                    plot.title = element_text(hjust = 0.5)))
```


What can you observe from the boxplot? Yes, outliers.

### Remove outliers

We are now set to begin writing some code ourselves. Let's begin by dealing with ouliers. An outlier is a data point that differs significantly from other observations.

**Question 1.**

Starting with the `estate_data` dataset, `filter` to create a subset that contains observations where `price_per_unit` is less than 70.

```
BEGIN QUESTION
name: Question 1
manual: false
```

```r
# Narrow down to observations whose price_per_unit is less than 70
estate_data <- estate_data %>%
  filter(price_per_unit < 70)
```

```r
. = " # BEGIN TEST CONFIG

success_message: Great start! Your tibble dimensions are correct.

failure_message: Almost there! Ensure you have filtered correctly to obtain a subset whose observations
" # END TEST CONFIG

suppressPackageStartupMessages({
  library(testthat)
  library(ottr)
})

## Test ##
test_that('data dimensions correct', {
  expect_equal(dim(estate_data), c(408, 7))

})
```

```
## Test passed
```

```r
. = " # BEGIN TEST CONFIG

success_message: Excellent. You have successfully created a subset whose observations of price_per_unit

failure_message: Let's give this another try. Ensure your subset contains observations where **price_pe
" # END TEST CONFIG




## Test ##
test_that('the range of values for price per unit is within 7.6 and 69.7', {
    expect_equal(range(estate_data$price_per_unit), c(7.6, 69.7))
})
```

```
## Test passed
```

Now let's take a look at the distribution without the outliers.

```r
# Plot a histogram
theme_set(theme_light())
hist_plt <- estate_data %>%
  ggplot(mapping = aes(x = price_per_unit)) +
  geom_histogram(bins = 100, fill = "midnightblue", alpha = 0.7) +
```

```r
  # Add lines for mean and median
  geom_vline(aes(xintercept = mean(price_per_unit), color = 'Mean'), linetype = "dashed", size = 1.3) +
  geom_vline(aes(xintercept = median(price_per_unit), color = 'Median'), linetype = "dashed", size = 1.
  xlab("") +
  ylab("Frequency") +
  scale_color_manual(name = "", values = c(Mean = "red", Median = "yellow")) +
  theme(legend.position = c(0.9, 0.9), legend.background = element_blank())

# Plot a box plot
box_plt <- estate_data %>%
  ggplot(aes(x = price_per_unit, y = 1)) +
  geom_boxplot(fill = "#E69F00", color = "gray23", alpha = 0.7) +
    # Add titles and labels
  xlab("Price_per_unit")+
  ylab("")


# Combine plots using patchwork syntax
(hist_plt / box_plt) +
  plot_annotation(title = 'Price Distribution',
                  theme = theme(
                    plot.title = element_text(hjust = 0.5)))
```


Much better! What can you say about the distribution of the price?

**View numeric correlations**

We can now start to look for relationships between the *features* and the *label* we want to be able to predict.

The *correlation* statistic, *r*, is a value between -1 and 1 that indicates the strength of a linear relationship.

For the numeric features, we can create scatter plots that show the intersection of feature and label values.

**Question 2.**

Starting with the `estate_data` dataset, in a piped sequence: - `pivot_longer` the data (increase the number of rows and decrease the number of columns) such that all the existing column names except price_per_unit now fall under a new column name called **`features`** and their corresponding values under a new column name **`values`**

- group the data by `features`
- add a new column `corr_coef` which calculates the correlation between `values` and `price_per_unit` (hint: the function used for calculating correlation in R is `cor()`)

```
BEGIN QUESTION
name: Question 2
manual: false
```

```r
# Pivot numeric features to a long format
numeric_features_long <- estate_data %>%
  pivot_longer(!price_per_unit, names_to = "features", values_to = "values") %>%
  group_by(features) %>%
  mutate(corr_coef = cor(values, price_per_unit),
         features = paste(features, ' vs price, r = ', round(corr_coef, 2), sep = '')) %>%
  ungroup()

# Print the first few rows of the data
numeric_features_long %>%
  slice_head(n = 10)
```

```
## # A tibble: 10 x 4
##    price_per_unit features                                     values corr_coef
##             <dbl> <chr>                                         <dbl>     <dbl>
##  1           37.9 transaction_date vs price, r = 0.07          2013.     0.0672
##  2           37.9 house_age vs price, r = -0.22                  32      -0.220
##  3           37.9 transit_distance vs price, r = -0.71           84.9    -0.709
##  4           37.9 local_convenience_stores vs price, r = 0.61    10       0.610
##  5           37.9 latitude vs price, r = 0.57                    25.0     0.575
##  6           37.9 longitude vs price, r = 0.56                  122.      0.556
##  7           42.2 transaction_date vs price, r = 0.07          2013.     0.0672
##  8           42.2 house_age vs price, r = -0.22                  19.5    -0.220
##  9           42.2 transit_distance vs price, r = -0.71          307.     -0.709
## 10           42.2 local_convenience_stores vs price, r = 0.61     9       0.610
```

```
. = " # BEGIN TEST CONFIG

success_message: Great start! Your tibble dimensions and corresponding columns are correct.

failure_message: Almost there! Let's give this another shot.
```

```
" # END TEST CONFIG


## Test ##
test_that('data dimensions correct', {
  expect_equal(dim(numeric_features_long), c(2448, 4))
  expect_equal(sort(colnames(numeric_features_long)), c("corr_coef", "features", "price_per_unit", "valu

})
```

```
## Test passed
```

```
. = " # BEGIN TEST CONFIG

success_message: Excellent. You have successfully pivoted the tibble and found the correlation between t

failure_message: Let's give this another try. Ensure you have correctly pivoted the data to obtain two n
" # END TEST CONFIG




## Test ##
test_that('the correlation coefficients are correct', {
    expect_equal(round(range(numeric_features_long$corr_coef), 7), c(-0.7087782, 0.6101017))
})
```

```
## Test passed
```

Fantastic! Now let's use a scatter plot to investigate whether there is any linear relationship between our predictors and outcome variables.

```
# Plot a scatter plot for each feature
numeric_features_long %>%
  ggplot(aes(x = values, y = price_per_unit, color = features)) +
  geom_point(alpha = 0.7, show.legend = F) +
  facet_wrap(~ features, scales = 'free') +
  paletteer::scale_color_paletteer_d("ggthemes::excel_Parallax")
```


Take a moment and go through the scatter plot. How does the correlation between these features and the price vary?

### View categorical features

Now let's compare the categorical features to the label. We'll do this by creating box plots that show the distribution of rental counts for each category.

**transaction_date** and **local_convenience_stores** seem to be discrete values - so might work better if treated as categorical features. Let' get right into it.

**Question 3.**

Starting with the `estate_data` dataset, in a piped sequence:

- only keep columns `transaction_date`, `local_convenience_stores` and `price_per_unit`
- encode columns `transaction_date` and `local_convenience_stores` as categorical (factor)
- `pivot_longer` the data (increase the number of rows and decrease the number of columns) such that all the existing column names except price_per_unit now fall under a new column name called **`features`** and their corresponding values under a new column name **`values`**

```
BEGIN QUESTION
name: Question 3
manual: false
```

```r
# Pivot categorical features to a long format
cat_features_long <- estate_data %>%
  select(transaction_date, local_convenience_stores, price_per_unit) %>%
  # Encode features from numeric to categorical
  mutate(across(c(transaction_date, local_convenience_stores), factor)) %>%
  pivot_longer(!price_per_unit, names_to = "features", values_to = "values")

# Print some observations
cat_features_long %>%
  slice_head(n = 10)
```

```
## # A tibble: 10 x 3
##    price_per_unit features                 values
##             <dbl> <chr>                    <fct>
##  1           37.9 transaction_date         2012.917
##  2           37.9 local_convenience_stores 10
##  3           42.2 transaction_date         2012.917
##  4           42.2 local_convenience_stores 9
##  5           47.3 transaction_date         2013.583
##  6           47.3 local_convenience_stores 5
##  7           54.8 transaction_date         2013.5
##  8           54.8 local_convenience_stores 5
##  9           43.1 transaction_date         2012.833
## 10           43.1 local_convenience_stores 5
```

```r
. = " # BEGIN TEST CONFIG

success_message: Fantastic! Your tibble dimensions and corresponding columns are correct.

failure_message: Almost there! Ensure you have selected columns transaction_date, local_convenience_sto
" # END TEST CONFIG


## Test ##
test_that('data dimensions correct', {
  expect_equal(dim(cat_features_long), c(816, 3))
  expect_equal(sort(colnames(cat_features_long)), c("features", "price_per_unit", "values"))

})
```

```
## Test passed
```

```r
. = " # BEGIN TEST CONFIG

success_message: Congratulations! You have successfully selected the desired columns, encoded some of tl

failure_message: Almost there! Ensure you have selected columns transaction_date, local_convenience_sto
" # END TEST CONFIG


## Test ##
test_that('data contains the correct observations', {
```

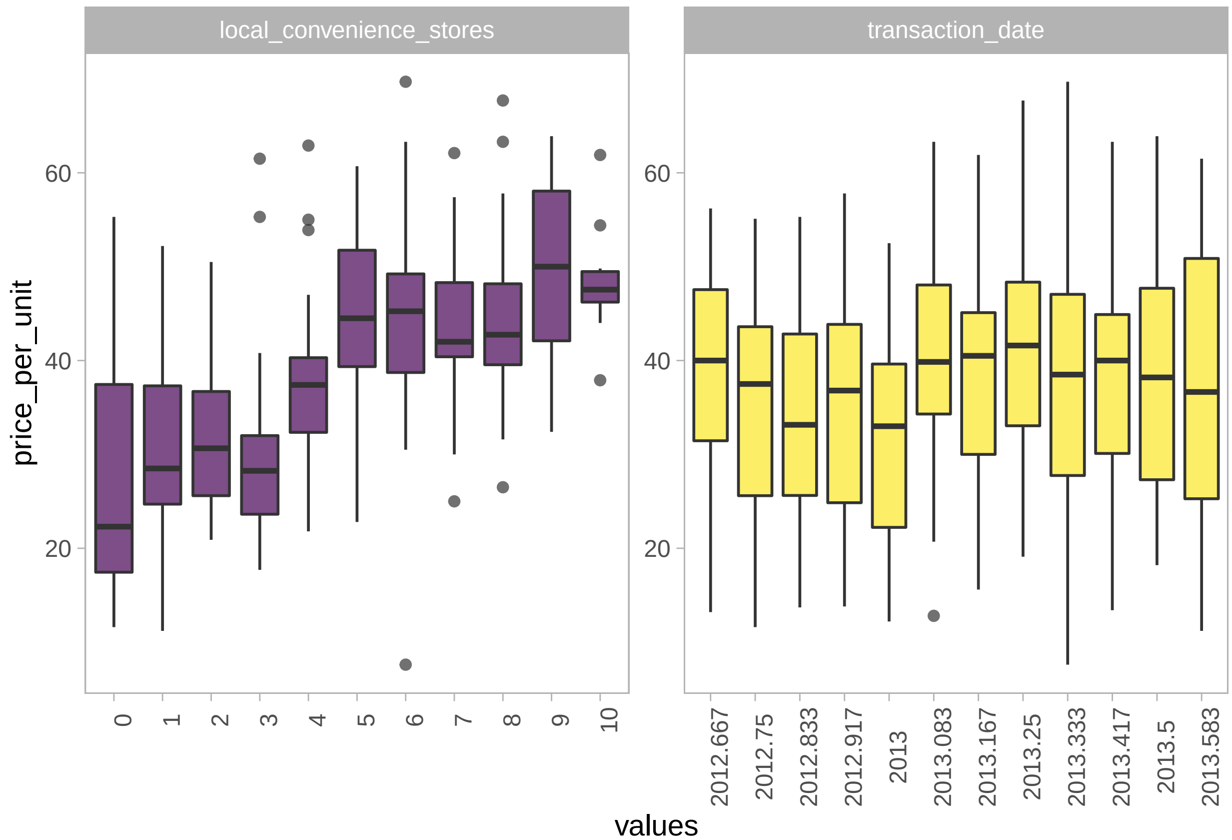
```
  expect_equal(sort(unique(cat_features_long$features)), c("local_convenience_stores", "transaction_dat
  expect_equal(class(cat_features_long$values), "factor")

})
```

```
## Test passed
```

Perfect! Now, for our categorical features, boxplots can be a great way of visualising how the price per unit varies within the levels of the categorical feature.

```
# Plot a box plot for each feature
cat_features_long %>%
  ggplot() +
  geom_boxplot(aes(x = values, y = price_per_unit, fill = features), alpha = 0.7, show.legend = F) +
  facet_wrap(~ features, scales = 'free') +
  scale_fill_viridis_d() +
  theme(panel.grid = element_blank(),
        axis.text.x = element_text(angle = 90))
```




Take a moment and interpret the graphics. How does the price vary with these features?

### Split the data into training and test sets.

Now that we've explored the data, it's time to use it to train a regression model that uses the features we've identified as `potentially predictive` to predict the **price__per__unit** label.

**transaction_date** doesn't seem to be very predictive, so we'll omit it.

Let's begin by splitting the data set such that some goes to training and some goes for validation. This enables us to evaluate how well the model performs in order to get a better estimate of how your models will perform on new data.

**Question 4.**

In this section:

- Make a split specification of `estate_data` such that `70%` goes to training and the rest goes to testing. Save this to a variable name `estate_split`
- Extract the training and testing sets from `estate_split` and save them in `estate_train` and `estate_test` variable names respectively.

```
BEGIN QUESTION
name: Question 4
manual: false
```

```r
# Set seed to ensure reproducibility and consitency of outputs
set.seed(2056)

# Load the tidymodels package
library(tidymodels)

# Split 70% of the data for training and the rest for tesing
estate_split <- estate_data %>%
  initial_split(prop = 0.7)

# Extract the data in each split
estate_train <- training(estate_split)
estate_test <- testing(estate_split)

# Print the number of observations in each split
cat("Training Set", nrow(estate_train), "rows",
    "\nTest Set", nrow(estate_test), "rows")
```

```
## Training Set 285 rows
## Test Set 123 rows
```

```r
. = " # BEGIN TEST CONFIG

success_message: Fantastic! You have successfully split the data and extracted the training (70%) and te

failure_message: Almost there. Let's have a look at this again.Ensure that the splitting specification 
" # END TEST CONFIG


## Test ##
test_that('data dimensions correct', {
  expect_equal(dim(estate_train), c(285, 7))
  expect_equal(dim(estate_test), c(123, 7))

})
```

```
## Test passed
```

Great progress!

## Train a regression model

### Preprocess data using recipes

Often before fitting a model, we may want to reformat the predictor values to make them easier for a model to use effectively. This includes transformations and encodings of the data to best represent their important characteristics. In R,this is done using a `recipe`.

A recipe is an object that defines a series of steps for data processing.

**Question 5.**

In this section, specify a recipe, `estate_recipe`, that will:

- Remove the `transaction_date` feature
- Transform `local_convenience_stores` feature into categorical (factor)
- Center and scale all numeric predictors

```
BEGIN QUESTION
name: Question 5
manual: false
```

```
# Create a preprocessing recipe
estate_recipe <- recipe(price_per_unit ~ ., data = estate_train) %>%
  # Specify the removal of a variable
  step_rm(transaction_date) %>%
  # Specify the encoding of local_convenience_stores as categorical
  step_mutate(
    local_convenience_stores = factor(local_convenience_stores)) %>%
  # Specify the normalization of numeric features
  step_normalize(all_numeric_predictors())

# Print recipe
estate_recipe
```

```
## Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          6
##
## Operations:
##
## Delete terms transaction_date
## Variable mutation
## Centering and scaling for all_numeric_predictors()
```

```
. = " # BEGIN TEST CONFIG

success_message: Good job. You have correctly specified a recipe that will Remove the `transaction_date

failure_message: Almost there. Ensure your recipe specification will Remove the `transaction_date` featu
" # END TEST CONFIG

## Test ##
test_that('recipe specification is correct', {

  # Test for step_rm
  expect_equal(attr(estate_recipe[["steps"]][[1]], "class"), c("step_rm","step"))
  expect_equal(as_label(estate_recipe[["steps"]][[1]][["terms"]][[1]]), "transaction_date")

  # Test for step_mutate
  expect_equal(attr(estate_recipe[["steps"]][[2]], "class"), c("step_mutate","step"))
  expect_equal(as_label(estate_recipe[["steps"]][[2]][["inputs"]][["local_convenience_stores"]]), "facto

  # Test for step_normalize
  expect_equal(attr(estate_recipe[["steps"]][[3]], "class"), c("step_normalize","step"))
  expect_equal(as_label(estate_recipe[["steps"]][[3]][["terms"]][[1]]), "all_numeric_predictors()")


})
```

```
## Test passed
```

Fantastic! We have the data processing in order. Now, let's make a model specification. In this solution, we'll try out a random forest model which applies an averaging function to multiple decision tree models for a better overall model.

**Question 6.**

Create a random forest model specification, `rf_spec`, which uses the `randomForest` package as its engine and then set the mode to `regression`.

```
BEGIN QUESTION
name: Question 6
manual: false
```

```
# Build a random forest model specification
rf_spec <- rand_forest() %>%
  set_engine('randomForest') %>%
  set_mode('regression')
```

```
. = " # BEGIN TEST CONFIG

success_message: Excellent! Your model specification is looking great!

failure_message: Let's have a look at this again. Ensure you have set your engine to **randomForest** an
" # END TEST CONFIG

## Test ##
```

```r
test_that('the model specification is correct', {
  expect_equal(rf_spec$mode, "regression")
  expect_equal(rf_spec$engine, "randomForest")


})
```

```
## Test passed
```

### Create a modeling workflow

The **workflows** package allows the user to bind modeling and preprocessing objects together. You can then fit the entire workflow to the data, so that the model encapsulates all of the preprocessing steps as well as the algorithm.

### Question 7.

Components of a `workflow()` go together like LEGO blocks. In this section, create a workflow container and then add the preprocessing information from our recipe and then add the model specification to be trained.

```
BEGIN QUESTION
name: Question 7
manual: false
```

```r
# Create a workflow that bundles a recipe and model specification
rf_workflow <- workflow() %>%
  add_recipe(estate_recipe) %>%
  add_model(rf_spec)

# Print workflow
rf_workflow
```

```
## == Workflow ====================================================================
## Preprocessor: Recipe
## Model: rand_forest()
##
## -- Preprocessor ----------------------------------------------------------------
## 3 Recipe Steps
##
## * step_rm()
## * step_mutate()
## * step_normalize()
##
## -- Model -----------------------------------------------------------------------
## Random Forest Model Specification (regression)
##
## Computational engine: randomForest
```

```r
. = " # BEGIN TEST CONFIG


" # END TEST CONFIG
```

```r
## Test ##
test_that('workflow specification is correct', {

  # Test for step_rm
  expect_equal(attr(rf_workflow[["pre"]][["actions"]][["recipe"]][["recipe"]][["steps"]][[1]], "class")
  expect_equal(as_label(rf_workflow[["pre"]][["actions"]][["recipe"]][["recipe"]][["steps"]][[1]][["terr

  # Test for step_mutate
  expect_equal(attr(rf_workflow[["pre"]][["actions"]][["recipe"]][["recipe"]][["steps"]][[2]], "class")
  expect_equal(as_label(rf_workflow[["pre"]][["actions"]][["recipe"]][["recipe"]][["steps"]][[2]][["inpu

  # Test for step_normalize
  expect_equal(attr(rf_workflow[["pre"]][["actions"]][["recipe"]][["recipe"]][["steps"]][[3]], "class")
  expect_equal(as_label(rf_workflow[["pre"]][["actions"]][["recipe"]][["recipe"]][["steps"]][[3]][["terr



})
```

```
## Test passed
```

Now that we have everything (recipe + model specification) wrapped together nicely in a workflow, we are ready to train a model. Workflows have a `fit()` method that can be used to train a model.

```r
# For reproducibility
set.seed(2056)

# Train a random forest model
rf_workflow_fit <- rf_workflow %>%
  fit(data = estate_train)

# Print out the fitted workflow
rf_workflow_fit
```

```
## == Workflow [trained] ==========================================================
## Preprocessor: Recipe
## Model: rand_forest()
##
## -- Preprocessor ----------------------------------------------------------------
## 3 Recipe Steps
##
## * step_rm()
## * step_mutate()
## * step_normalize()
##
## -- Model -----------------------------------------------------------------------
##
## Call:
##  randomForest(x = maybe_data_frame(x), y = y)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 1
```

```
##
##           Mean of squared residuals: 39.46666
##                     % Var explained: 73.58
```

Excellent! So we now have a trained random forest model; but is it any good? Let's evaluate its performance! We'll do this by making predictions on the **test data** and then evaluate some performance metrics based on the actual outcomes.

**Question 8.**

- We'll evaluate the model performance based on the `rmse` and `rsq` metrics. Use the `metric_set()` function to combine these metric functions together into a new function, `eval_metrics`, that calculates all of them at once.
- Generate predictions for the test data and then bind them to the test set. Rename the column containing predictions from `.pred` to `predictions`.

```
BEGIN QUESTION
name: Question 8
manual: false
```

```
# Create a metric set
eval_metrics <- metric_set(rmse, rsq)


# Make and bind predictions to test data
results <- rf_workflow_fit %>%
  augment(new_data = estate_test) %>%
  rename(predictions = .pred)
```

```
. = " # BEGIN TEST CONFIG

success_message: Fantastic! You have successfully made predictions and binded them to the test set.

failure_message: Almost there! Generate predictions for the test data and then bind them to the test se
" # END TEST CONFIG

## Test ##
test_that('the model specification is correct', {
  expect_equal(dim(results), c(123, 8))
  expect_equal(sort(colnames(results)), c("house_age", "latitude", "local_convenience_stores", "longitu


})
```

```
## Test passed
```

Awesome work! You have just used your trained model to make predictions on the test set.

How well did the model predict the prices per unit? Let's find out by looking at the metrics.

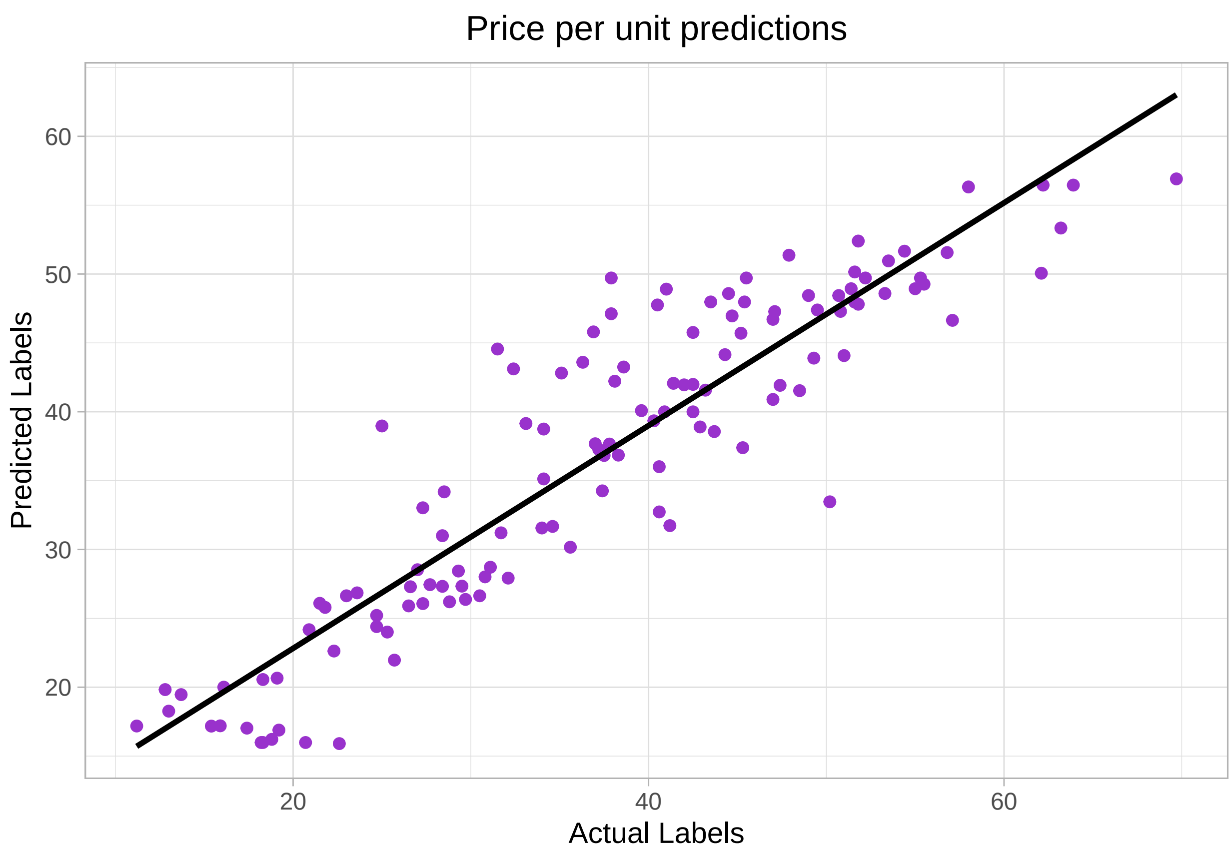
```r
# Evaluate the model
rf_metrics <- eval_metrics(data = results,
                           truth = price_per_unit,
                           estimate = predictions)


# Plot predicted vs actual
rf_plt <- results %>%
  ggplot(mapping = aes(x = price_per_unit, y = predictions)) +
  geom_point(color = 'darkorchid', size = 1.6) +
  # overlay regression line
  geom_smooth(method = 'lm', color = 'black', se = F) +
  ggtitle("Price per unit predictions") +
  xlab("Actual Labels") +
  ylab("Predicted Labels") +
  theme(plot.title = element_text(hjust = 0.5))

# Return evaluations
list(metrics = rf_metrics, evaluation_plot = rf_plt)
```

```
## $metrics
## # A tibble: 2 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard       5.23
## 2 rsq     standard       0.841
##
## $evaluation_plot
```

```
## `geom_smooth()` using formula 'y ~ x'
```




How do you think the model performed. What do the values for `rsq` and `rmse` tell you? You can refer to the corresponding module for this notebook to help answer these questions in case you are stuck.

### Use the Trained Model

Save your trained model, and then use it to predict the price-per-unit for the following real estate transactions:

| **transaction_date** | **house_age** | **transit_distance** | **local_convenience_stores** | **latitude** | **longitude** |
|---|---|---|---|---|---|
| 2013.167 | 16.2 | 289.3248 | 5 | 24.98203 | 121.54348 |
| 2013.000 | 13.6 | 4082.015 | 0 | 24.94155 | 121.50381 |

```
library(here)
```

```
## here() starts at C:/Users/medewan/Documents/GitHub/ml-basics-R
```

```
# Save trained workflow
saveRDS(rf_workflow_fit, "rf_price_model.rds")
```

Now, we can load it whenever we need it, and use it to predict labels for new data. This is often called *`scoring`* or *`inferencing`*.

```r
# Create a tibble for the new real estate samples
new_data <- tibble(
  transaction_date = c(2013.167, 2013.000),
  house_age = c(16.2, 13.6),
  transit_distance = c(289.3248, 4082.015),
  local_convenience_stores = c(5, 0),
  latitude = c(24.98203, 24.94155),
  longitude = c(121.54348, 121.50381))

# Print out new data
new_data
```

```
## # A tibble: 2 x 6
##   transaction_date house_age transit_distance local_convenience_stores latitude
##              <dbl>     <dbl>            <dbl>                    <dbl>    <dbl>
## 1            2013.      16.2             289.                        5     25.0
## 2            2013       13.6            4082.                        0     24.9
## # ... with 1 more variable: longitude <dbl>
```

Now that we have our data, let's load the saved model and make predictions.

```r
# Load the model into the current R session
loaded_model <- readRDS("rf_price_model.rds")

# Make predictions
predictions <- loaded_model %>%
  augment(new_data = new_data)

predictions
```

```
## # A tibble: 2 x 7
##   transaction_date house_age transit_distance local_convenience_stores latitude
##              <dbl>     <dbl>            <dbl>                    <dbl>    <dbl>
## 1            2013.      16.2             289.                        5     25.0
## 2            2013       13.6            4082.                        0     24.9
## # ... with 2 more variables: longitude <dbl>, .pred <dbl>
```

That's it for now. In this notebook, you:

- Explored the data set to understand the relationships between the predictors and outcomes
- Preprocessed the data using recipes to make them easier for a model to use effectively.
- Made a random forest model specifcation.
- Bundles a recipe and model specification into a workflow.
- Trained a model.
- Made predictions on test set and evaluated the model performance.
- Saved the model, loaded it and then used it to predict labels for new data.

Fantastic job for coming this far! Feeling adventurous? Then, be sure to try out other regression models and tune some hyperparameters while at it.

Happy Learning,

Eric, Gold Microsoft Learn Student Ambassador.