3 Linear Regression

This lab will go over how to perform linear regression. This will include simple linear regression and multiple linear regression in addition to how you can apply transformations to the predictors. This chapter will use parsnip for model fitting and recipes and workflows to perform the transformations.

3.1 Libraries

We load tidymodels and ISLR and MASS for data sets.

```
library(MASS) # For Boston data set
library(tidymodels)
library(ISLR)
```

3.2 Simple linear regression

The `Boston` data set contains various statistics for 506 neighborhoods in Boston. We will build a simple linear regression model that related the median value of owner-occupied homes (`medv`) as the response with a variable indicating the percentage of the population that belongs to a lower status (`lstat`) as the predictor.

> **Important**
>
> The `Boston` data set is quite outdated and contains some really unfortunate variables.

We start by creating a parsnip specification for a linear regression model.

```
lm_spec <- linear_reg() %>%
  set_mode("regression") %>%
  set_engine("lm")
```

While it is unnecessary to set the mode for a linear regression since it can only be regression, we continue to do it in these labs to be explicit.

The specification doesn't perform any calculations by itself. It is just a specification of what we want to do.

```
lm_spec
```

```
Linear Regression Model Specification (regression)

Computational engine: lm
```

Once we have the specification we can `fit` it by supplying a formula expression and the data we want to fit the model on. The formula is written on the form `y ~ x` where `y` is the name of the response and `x` is the name of the predictors. The names used in the formula should match the names of the variables in the data set passed to `data`.

```
lm_fit <- lm_spec %>%
  fit(medv ~ lstat, data = Boston)

lm_fit
```

```
parsnip model object



Call:
stats::lm(formula = medv ~ lstat, data = data)

Coefficients:
(Intercept)        lstat
      34.55        -0.95
```

The result of this fit is a parsnip model object. This object contains the underlying fit as well as some parsnip-specific information. If we want to look at the underlying fit object we can access it with `lm_fit$fit` or with

```
lm_fit %>%
  pluck("fit")
```

```

Call:
stats::lm(formula = medv ~ lstat, data = data)

Coefficients:
(Intercept)        lstat
      34.55        -0.95
```

The `lm` object has a nice `summary()` method that shows more information about the fit, including parameter estimates and lack-of-fit statistics.

```
lm_fit %>%
  pluck("fit") %>%
  summary()
```

```

Call:
stats::lm(formula = medv ~ lstat, data = data)

Residuals:
    Min      1Q  Median      3Q     Max
```

```
-15.168  -3.990  -1.318   2.034  24.500

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 34.55384    0.56263   61.41   <2e-16 ***
lstat       -0.95005    0.03873  -24.53   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 6.216 on 504 degrees of freedom
Multiple R-squared:  0.5441,    Adjusted R-squared:  0.5432
F-statistic: 601.6 on 1 and 504 DF,  p-value: < 2.2e-16
```

We can use packages from the broom package to extract key information out of the model objects in tidy formats.

the `tidy()` function returns the parameter estimates of a `lm` object

```
tidy(lm_fit)
```

```
# A tibble: 2 × 5
  term        estimate std.error statistic   p.value
  <chr>          <dbl>     <dbl>     <dbl>     <dbl>
1 (Intercept)   34.6      0.563       61.4 3.74e-236
2 lstat         -0.950    0.0387     -24.5 5.08e- 88
```

and `glance()` can be used to extract the model statistics.

```
glance(lm_fit)
```

```
# A tibble: 1 × 12
  r.squared adj.r.squared sigma statistic  p.value    df logLik   AIC   BIC
      <dbl>         <dbl> <dbl>     <dbl>    <dbl> <dbl>  <dbl> <dbl> <dbl>
1     0.544         0.543  6.22      602. 5.08e-88     1 -1641. 3289. 3302.
# … with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

Suppose that we like the model fit and we want to generate predictions, we would typically use the `predict()` function like so:

```
predict(lm_fit)
```

```
Error in predict_numeric(object = object, new_data = new_data, ...): argument "new_data" is
missing, with no default
```

But this produces an error when used on a parsnip model object. This is happening because we need to explicitly supply the data set that the predictions should be performed on via the `new_data` argument

```
predict(lm_fit, new_data = Boston)
```

```
# A tibble: 506 × 1
   .pred
   <dbl>
 1 29.8
 2 25.9
 3 30.7
 4 31.8
 5 29.5
 6 29.6
 7 22.7
 8 16.4
 9  6.12
10 18.3
# … with 496 more rows
```

Notice how the predictions are returned as a tibble. This will always be the case for parsnip models, no matter what engine is used. This is very useful since consistency allows us to combine data sets easily.

We can also return other types of predicts by specifying the `type` argument. Setting `type = "conf_int"` return a 95% confidence interval.

```
predict(lm_fit, new_data = Boston, type = "conf_int")
```

```
# A tibble: 506 × 2
   .pred_lower .pred_upper
         <dbl>       <dbl>
 1       29.0        30.6
 2       25.3        26.5
 3       29.9        31.6
 4       30.8        32.7
 5       28.7        30.3
 6       28.8        30.4
 7       22.2        23.3
 8       15.6        17.1
 9        4.70        7.54
10       17.7        18.9
# … with 496 more rows
```

> **Note**
>
> Not all engines can return all types of predictions.

If you want to evaluate the performance of a model, you might want to compare the observed value and the predicted value for a data set. You

```
bind_cols(
  predict(lm_fit, new_data = Boston),
  Boston
) %>%
  select(medv, .pred)
```

```
# A tibble: 506 × 2
    medv .pred
   <dbl> <dbl>
 1  24    29.8
 2  21.6  25.9
 3  34.7  30.7
 4  33.4  31.8
 5  36.2  29.5
 6  28.7  29.6
 7  22.9  22.7
 8  27.1  16.4
 9  16.5   6.12
10  18.9  18.3
# … with 496 more rows
```

You can get the same results using the `augment()` function to save you a little bit of typing.

```
augment(lm_fit, new_data = Boston) %>%
  select(medv, .pred)
```

```
# A tibble: 506 × 2
    medv .pred
   <dbl> <dbl>
 1  24    29.8
 2  21.6  25.9
 3  34.7  30.7
 4  33.4  31.8
 5  36.2  29.5
 6  28.7  29.6
 7  22.9  22.7
 8  27.1  16.4
 9  16.5   6.12
10  18.9  18.3
# … with 496 more rows
```

3.3 Multiple linear regression

The multiple linear regression model can be fit in much the same way as the simple linear regression model. The only difference is how we specify the predictors. We are using the same formula expression `y ~ x`, but we can specify multiple values by separating them with `+`s.

```
lm_fit2 <- lm_spec %>%
  fit(medv ~ lstat + age, data = Boston)

lm_fit2
```

```
parsnip model object
```

```
Call:
stats::lm(formula = medv ~ lstat + age, data = data)

Coefficients:
(Intercept)        lstat          age
   33.22276     -1.03207      0.03454
```

Everything else works the same. From extracting parameter estimates

```
tidy(lm_fit2)
```

```
# A tibble: 3 × 5
  term        estimate std.error statistic   p.value
  <chr>          <dbl>     <dbl>     <dbl>     <dbl>
1 (Intercept)  33.2       0.731      45.5  2.94e-180
2 lstat        -1.03      0.0482    -21.4  8.42e- 73
3 age           0.0345    0.0122      2.83 4.91e-  3
```

to predicting new values

```
predict(lm_fit2, new_data = Boston)
```

```
# A tibble: 506 × 1
   .pred
   <dbl>
 1 30.3
 2 26.5
 3 31.2
 4 31.8
 5 29.6
 6 29.9
 7 22.7
 8 16.8
 9  5.79
10 18.5
# … with 496 more rows
```

A shortcut when using formulas is to use the form `y ~ .` which means; set `y` as the response and set the remaining variables as predictors. This is very useful if you have a lot of variables and you don't want to type them out.

```
lm_fit3 <- lm_spec %>%
  fit(medv ~ ., data = Boston)

lm_fit3
```

```
parsnip model object
```

```
Call:
stats::lm(formula = medv ~ ., data = data)

Coefficients:
(Intercept)         crim           zn        indus         chas          nox
  3.646e+01   -1.080e-01    4.642e-02    2.056e-02    2.687e+00   -1.777e+01
         rm          age          dis          rad          tax      ptratio
  3.810e+00    6.922e-04   -1.476e+00    3.060e-01   -1.233e-02   -9.527e-01
      black        lstat
  9.312e-03   -5.248e-01
```

For more formula syntax look at `?formula`.

3.4 Interaction terms

Adding interaction terms is quite easy to do using formula expressions. However, the syntax used to describe them isn't accepted by all engines so we will go over how to include interaction terms using recipes as well.

There are two ways on including an interaction term; `x:y` and `x * y`

- `x:y` will include the interaction between `x` and `y`,
- `x * y` will include the interaction between `x` and `y`, `x`, and `y`, i.e. it is short for `x:y + x + y`.

with that out of the way let expand `lm_fit2` by adding an interaction term

```
lm_fit4 <- lm_spec %>%
  fit(medv ~ lstat * age, data = Boston)

lm_fit4
```

```
parsnip model object


Call:
stats::lm(formula = medv ~ lstat * age, data = data)

Coefficients:
(Intercept)        lstat          age    lstat:age
 36.0885359   -1.3921168   -0.0007209    0.0041560
```

note that the interaction term is named `lstat:age`.

Sometimes we want to perform transformations, and we want those transformations to be applied, as part of the model fit as a pre-processing step. We will use the recipes package for this task.

We use the `step_interact()` to specify the interaction term. Next, we create a workflow object to combine the linear regression model specification `lm_spec` with the pre-processing specification `rec_spec_interact` which can then be fitted much like a parsnip model specification.

```r
rec_spec_interact <- recipe(medv ~ lstat + age, data = Boston) %>%
  step_interact(~ lstat:age)

lm_wf_interact <- workflow() %>%
  add_model(lm_spec) %>%
  add_recipe(rec_spec_interact)

lm_wf_interact %>% fit(Boston)
```

```
══ Workflow [trained] ══════════════════════════════════════════════════════════
Preprocessor: Recipe
Model: linear_reg()

── Preprocessor ────────────────────────────────────────────────────────────────
1 Recipe Step

• step_interact()

── Model ───────────────────────────────────────────────────────────────────────

Call:
stats::lm(formula = ..y ~ ., data = data)

Coefficients:
(Intercept)        lstat          age  lstat_x_age
 36.0885359   -1.3921168   -0.0007209    0.0041560
```

Notice that since we specified the variables in the recipe we don't need to specify them when fitting the workflow object. Furthermore, take note of the name of the interaction term. `step_interact()` tries to avoid special characters in variables.

3.5 Non-linear transformations of the predictors

Much like we could use recipes to create interaction terms between values are we able to apply transformations to individual variables as well. If you are familiar with the dplyr package then you know how to `mutate()` which works in much the same way using `step_mutate()`.

You would want to keep as much of the pre-processing inside recipes such that the transformation will be applied consistently to new data.

```r
rec_spec_pow2 <- recipe(medv ~ lstat, data = Boston) %>%
  step_mutate(lstat2 = lstat ^ 2)

lm_wf_pow2 <- workflow() %>%
  add_model(lm_spec) %>%
  add_recipe(rec_spec_pow2)
```

```
lm_wf_pow2 %>% fit(Boston)
```

```
══ Workflow [trained] ══════════════════════════════════════════════════════════
Preprocessor: Recipe
Model: linear_reg()

── Preprocessor ────────────────────────────────────────────────────────────────
1 Recipe Step

• step_mutate()

── Model ───────────────────────────────────────────────────────────────────────

Call:
stats::lm(formula = ..y ~ ., data = data)

Coefficients:
(Intercept)        lstat       lstat2
   42.86201     -2.33282      0.04355
```

You don't have to hand-craft every type of linear transformation since recipes have a bunch created already here such as `step_log()` to take logarithms of variables.

```
rec_spec_log <- recipe(medv ~ lstat, data = Boston) %>%
  step_log(lstat)

lm_wf_log <- workflow() %>%
  add_model(lm_spec) %>%
  add_recipe(rec_spec_log)

lm_wf_log %>% fit(Boston)
```

```
══ Workflow [trained] ══════════════════════════════════════════════════════════
Preprocessor: Recipe
Model: linear_reg()

── Preprocessor ────────────────────────────────────────────────────────────────
1 Recipe Step

• step_log()

── Model ───────────────────────────────────────────────────────────────────────

Call:
stats::lm(formula = ..y ~ ., data = data)

Coefficients:
(Intercept)        lstat
      52.12       -12.48
```

3.6 Qualitative predictors

We will now turn our attention to the `Carseats` data set. We will attempt to predict `Sales` of child car seats in 400 locations based on a number of predictors. One of these variables is `ShelveLoc` which is a qualitative predictor that indicates the quality of the shelving location. `ShelveLoc` takes on three possible values

- Bad
- Medium
- Good

If you pass such a variable to `lm()` it will read it and generate dummy variables automatically using the following convention.

```
Carseats %>%
  pull(ShelveLoc) %>%
  contrasts()
```

```
       Good Medium
Bad       0      0
Good      1      0
Medium    0      1
```

So we have no problems including qualitative predictors when using `lm` as the engine.

```
lm_spec %>%
  fit(Sales ~ . + Income:Advertising + Price:Age, data = Carseats)
```

```
parsnip model object


Call:
stats::lm(formula = Sales ~ . + Income:Advertising + Price:Age,
    data = data)

Coefficients:
       (Intercept)           CompPrice              Income         Advertising
         6.5755654           0.0929371           0.0108940           0.0702462
        Population               Price       ShelveLocGood     ShelveLocMedium
         0.0001592          -0.1008064           4.8486762           1.9532620
               Age           Education            UrbanYes               USYes
        -0.0579466          -0.0208525           0.1401597          -0.1575571
Income:Advertising           Price:Age
         0.0007510           0.0001068
```

However, as with so many things, we can not always guarantee that the underlying engine knows how to deal with qualitative variables. Recipes can be used to handle this as well. The `step_dummy()` will perform the same transformation of turning 1 qualitative with `C` levels into `C-1` indicator variables. While this might seem unnecessary right now, some of the engines, later on, do not handle qualitative variables and this step would be necessary. We are also using the `all_nominal_predictors()` selector to select all character and factor predictor variables. This allows us to select by type rather than having to type out the names.

```
rec_spec <- recipe(Sales ~ ., data = Carseats) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_interact(~ Income:Advertising + Price:Age)

lm_wf <- workflow() %>%
  add_model(lm_spec) %>%
  add_recipe(rec_spec)

lm_wf %>% fit(Carseats)
```

```
══ Workflow [trained] ══════════════════════════════════════════════════════════
Preprocessor: Recipe
Model: linear_reg()

── Preprocessor ────────────────────────────────────────────────────────────────
2 Recipe Steps

• step_dummy()
• step_interact()

── Model ───────────────────────────────────────────────────────────────────────

Call:
stats::lm(formula = ..y ~ ., data = data)

Coefficients:
        (Intercept)            CompPrice               Income
          6.5755654            0.0929371            0.0108940
        Advertising           Population                Price
          0.0702462            0.0001592           -0.1008064
                Age            Education       ShelveLoc_Good
         -0.0579466           -0.0208525            4.8486762
   ShelveLoc_Medium            Urban_Yes               US_Yes
          1.9532620            0.1401597           -0.1575571
Income_x_Advertising          Price_x_Age
          0.0007510            0.0001068
```

3.7 Writing functions

This book will not talk about how to write functions in R. If you still want to know how to write functions, we recommend the Functions chapter of the R for Data Science (2e).