12
Linear Regression with R

In this chapter, we will introduce linear regression, a fundamental statistical approach used to model the relationship between a target variable and multiple explanatory (also called independent) variables. We will cover the basics of linear regression, starting with simple linear regression and then extending the concepts to multiple linear regression. We will learn how to estimate the model coefficients, evaluate the goodness of fit, and test the significance of the coefficients using hypothesis testing. Additionally, we will discuss the assumptions underlying linear regression and explore techniques to address potential issues, such as nonlinearity, interaction effect, multicollinearity, and heteroskedasticity. We will also introduce two widely used regularization techniques: the ridge and **lasso** (**Least Absolute Shrinkage and Selection Operator**) penalties.

By the end of this chapter, you will learn the core principles of linear regression, its extensions to regularized linear regression, and the implementation details involved.

In this chapter, we will cover the following topics:

- Introducing linear regression
- Working with ridge regression
- Working with lasso regression

To run the codes in this chapter, you will need to have the latest versions of the following packages:

- `ggplot2` - 3.4.0.
- `tidyr` - 1.2.1.
- `dplyr` -1.0.10.
- `car` -3.1.1.
- `lmtest` -0.9.40.
- `glmnet` -4.1.7.

Please note that the versions of packages mentioned in the preceding list are the latest ones while I am writing this chapter. All the codes and data for this chapter are available at https://github.com/PacktPublishing/The-Statistics-and-Machine-Learning-with-R-Workshop/blob/main/Chapter_12/working.R.

Introducing linear regression

At the core of linear regression is the concept of fitting a straight line, or more generally, a hyperplane, to the data points. Such fitting aims to minimize the deviation between the observed

and predicted values. When it comes to simple linear regression, there is one target variable regressed by one predictor, and the goal is to fit a straight line that best mimics the relationship between the two variables. For multiple linear regression, there is more than one predictor, and the goal is to fit a hyperplane that best describes the relationship among the variables. Both tasks are achieved by minimizing a measure of deviation between the predictions and the corresponding targets.

In linear regression, obtaining an optimal model means identifying the best coefficients that define the relationship between the target variable and the input predictors. These coefficients represent the change in the target associated with a single unit change in the associated predictor, assuming all other variables constant. This allows us to quantify the magnitude (size of the coefficient) and direction (sign of the coefficient) of the relationship between the variables, which can be used for inference (highlighting explainability) and prediction.

When it comes to inference, we often look at the relative impact on the target variable given a unit change to the input variable. Examples of such explanatory modeling include how marketing spend affects quarterly sales, how smoker status affects the insurance premium, and how education affects income. On the other hand, predictive modeling focuses on predicting a target quantity. Examples include predicting quarterly sales given the marketing spend; predicting the insurance premium given a policyholder's profile information such as the age and gender; or predicting income given the education, age, work experience, and industry.

The expected outcome is modeled in linear regression as a weighted sum of all the input variables. It also assumes that the change in the output is linearly proportional to the change in any input variable. This is the simplest form of the regression method.

Let us start with the simple linear regression.

Understanding simple linear regression

Simple linear regression (**SLR**) is a powerful and widely-used statistical model that specifies the relationship between two continuous variables, including one input and one output. It allows us to understand how a response variable (also referred to as the dependent or target variable) changes as the explanatory variable (also called the independent variable or the input variable) varies. By fitting a straight line to the observed data, SLR quantifies the strength and direction of the linear association between the two variables. This straight line is called the SLR **model**. It enables us to make predictions and infer the impact of the predictor on the target variable.

Specifically, in a SLR model, we assume a linear relationship between the target variable (y) and the input variable (x). The model can be represented mathematically as:

$$y = \beta_0 + \beta_1 x + \epsilon$$

Here, y is called the response variable, dependent variable, explained variable, predicted variable, target variable, or regressand. x is called the explanatory variable, independent variable, control variable, predictor variable, or regressor. β_0 is the intercept of the linear line that represents the expected value of y when x is 0. β_1 is the slope that represents the change in y for a one-unit increase in x. Finally, ϵ is the random error term that accounts for the variability in the target y that the predictor x cannot explain.

The main objective in SLR is to estimate the parameters β_0 and β_1. An optimal set of β_0 and β_1 would minimize the total squared deviations between the observed target values y and the

predicted values $\hat{y}$ using from the model. This is called the **least squares method**, where we seek the optimal β_0 and β_1 that corresponds to the minimum **sum of squared error** (**SSR**):

$$\min SSR = \min \sum_{i=1}^{n} u_i^2 = \min(y_i - \hat{y}_i)^2$$

Here, each residual u_i is the difference between the observation y_i and its fitted value $\hat{y}_i$. In simple terms, the objective is to locate the straight line that is closest to the data points given. *Figure 12.1* illustrates a collection of data points (in blue) and the linear model (in red).

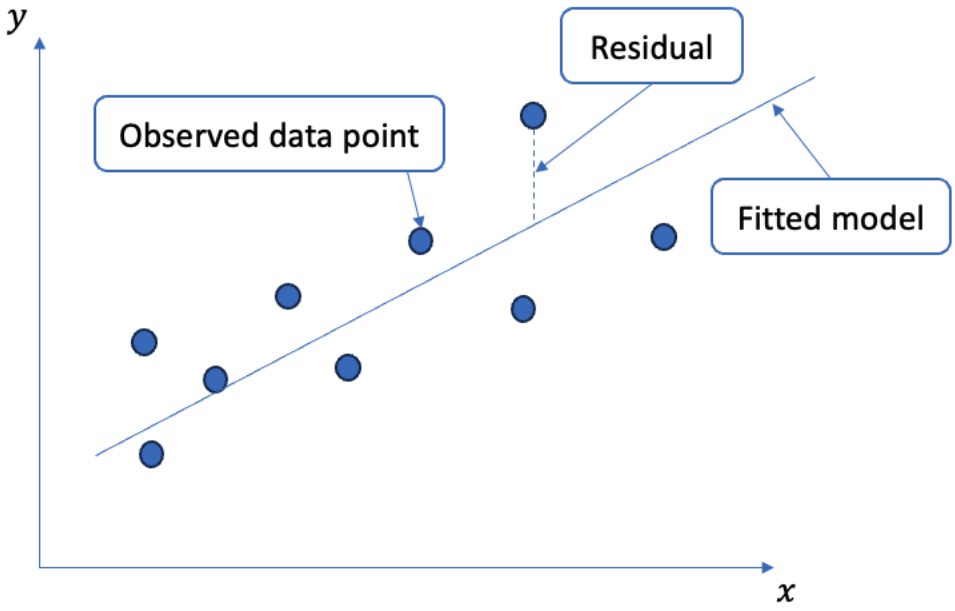

Figure 12.1 – Illustrating the SLR model, where the linear model appears as a line and is trained by minimizing the SSR

Once we have estimated the model coefficients β_0 and β_1, we can use the model to make predictions and make inferences on the intensity of the linear relationship between the variables. Such a linear relationship indicates the goodness of fit, often measured using the coefficient of determination, or the R^2. The R^2 ranges from 0 to 1 and quantifies the proportion of the total variation in y that can be explained by x. It is defined as the following:

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}$$

Where $\bar{y}$ denotes the average value of the observed target variable y.

Besides, we can also use hypothesis testing to test the significance of the resulting coefficients β_0 and β_1, thus helping us determine whether the observed relationship between the variables is statistically significant.

Let us go through an example of building a simple linear model using a simulated dataset.

Exercise 12.1 Building an SLR model

In this exercise, we will demonstrate the implementation of a SLR model in R. We'll be using a combination of built-in functions and packages to accomplish this task using a simulated dataset.

1. Simulate a dataset such that the response variable `Y` is linearly dependent on the explanatory variable `X` with some added noise.

```
# Set seed for reproducibility

set.seed(123)

# Generate independent variable X

X = runif(100, min = 1, max = 100) # 100 random uniform
numbers between 1 and 100

# Generate some noise

noise = rnorm(100, mean = 0, sd = 10) # 100 random normal
numbers with mean 0 and standard deviation 10

# Generate dependent variable Y

Y = 5 + 0.5 * X + noise

# Combine X and Y into a data frame

data = data.frame(X, Y)
```

Here, we use the `runif()` function to generate the independent variable `X`, a vector of random uniform numbers. We then add some 'noise' to the dependent variable `Y`, making the observed data more realistic and less perfectly linear. This is achieved using the `rnorm()` function, which creates a vector of random normal numbers. The target variable `Y` is then created as a function of `X`, plus the noise.

Besides, we use a seed (`set.seed(123)`) at the beginning to ensure reproducibility. This means that we will get the same set of random numbers every time we run this code. Each run would produce a different list of random numbers without setting a seed.

In this simulation, the true intercept (β_0) is 5, the true slope (β_1) is 0.5, and the noise is normally distributed with a mean of 0 and a standard deviation of 10.

2. Train a linear regression model based on the simulated dataset using the `lm()` function.

```
# Fit a simple linear regression model

model = lm(Y ~ X, data = data)

# Print the model summary

>>> summary(model)

Call:

lm(formula = Y ~ X, data = data)


Residuals:

     Min       1Q   Median       3Q      Max
```

```
-22.3797  -6.1323  -0.1973   5.9633  22.1723


Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  4.91948    1.99064   2.471   0.0152 *
X            0.49093    0.03453  14.218   <2e-16 ***
---
Signif. codes:
0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1


Residual standard error: 9.693 on 98 degrees of freedom
Multiple R-squared:  0.6735,     Adjusted R-squared:  0.6702
F-statistic: 202.2 on 1 and 98 DF,  p-value: < 2.2e-16
```

Here, we use the `lm()` function to fit the data, where `lm` stands for 'linear model'. This function creates our SLR model. The syntax `Y ~ X` is how we specify our model: it tells the function that `Y` is being modeled as a function of `X`.

The `summary()` function then provides a comprehensive overview of the model, including the estimated coefficients, the standard errors, the t-values, and the p-values, among other statistics. Since the resulting p-value is extremely low, we can conclude that the input variable is predictive with strong statistical significance.

3. Use the `plot()` and `abline()` functions to visualize the data and the fitted regression line.

```
# Plot the data
plot(data$X, data$Y, main = "Simple Linear Regression", xlab =
"X", ylab = "Y")
# Add the fitted regression line
abline(model, col = "red")
```

Running the codes generates *Figure 12.2*.

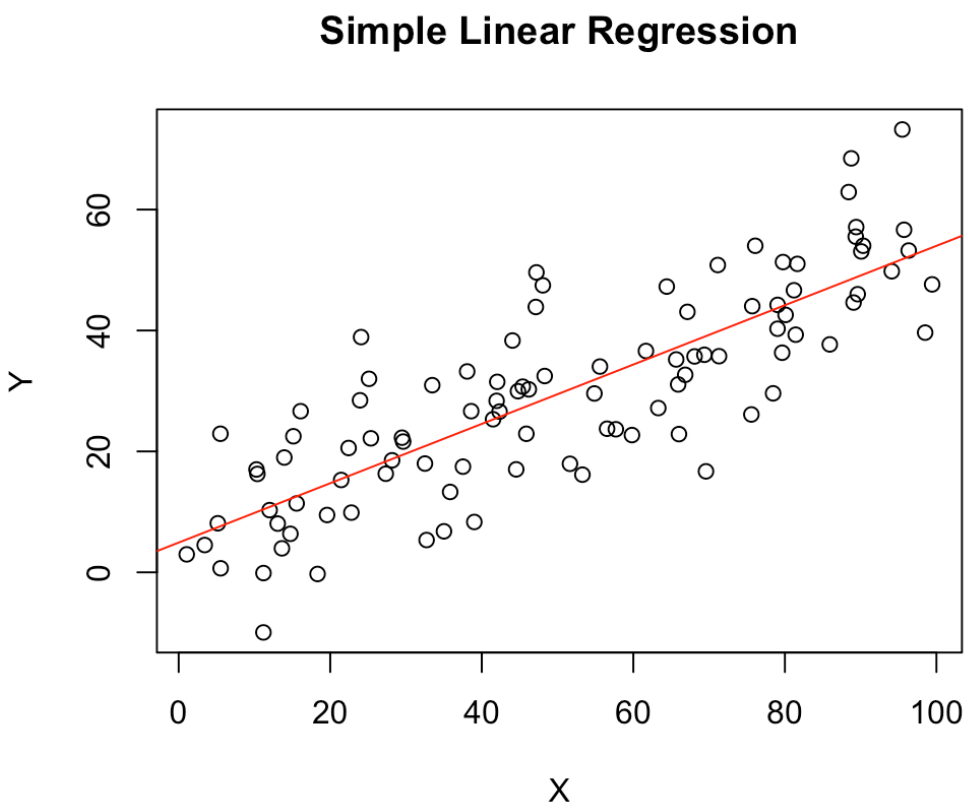

Figure 12.2 – Visualizing the data and the fitted regression line.

Here, the `plot()` function creates a scatter plot of our data, and the `abline()` function adds the regression line to this plot. Such a visual representation is very useful for understanding the quality of the fitting.

We move to the multiple linear regression model in the next section.

Introducing multiple linear regression

Multiple linear regression (**MLR**) expands the single predictor in SLR to predict the target outcome based on multiple predictor variables. Here, the term "multiple" in MLR refers to the multiple predictors in the model, where each feature is given a coefficient. A specific coefficient β represents the change in the outcome variable for a single unit change in the associated predictor variable, assuming all other predictors are held constant.

One of the great advantages of MLR is its ability to include multiple predictors, allowing for a more complex and realistic (linear) representation of the real world. It can provide a holistic view of the connection between the target and all input variables. This is particularly useful in fields where the outcome variable is likely influenced by more than one predictor variable. It is modeled via the following form:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p + \epsilon$$

where we have a total of p features, and therefore, $(p + 1)$ coefficients due to the intercept term. ϵ is the usual noise term that represents the unexplained part. In other words, our prediction using MLR is:

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p$$

We can perform ceteris paribus analysis with this formulation, which is a Latin way of saying all other things are equal, and we only change one input variable to assess its impact on the outcome variable. In other words, MLR allows us to explicitly control (that is, remain unchanged) many other factors that simultaneously affect the target variable and observe the impact of only one factor.

For example, suppose we add a small increment Δx_j to the feature x_j and remain all other features unchanged. The new prediction $\hat{y}_{new}$ is obtained by the following:

$$\hat{y}_{new} = \beta_0 + \beta_1 x_1 + \cdots + \beta_{\mathrm{j}}(x_{\mathrm{j}} + \Delta x_j) + \cdots + \beta_p x_p$$

We know that the original prediction is:

$$\hat{y}_{old} = \beta_0 + \beta_1 x_1 + \cdots + \beta_{\mathrm{j}} x_{\mathrm{j}} + \cdots + \beta_p x_p$$

The difference between these two gives the change in the output variable:

$$\Delta\hat{y} = \hat{y}_{new} - \hat{y}_{old} = \beta_{\mathrm{j}} \Delta x_j$$

What we are doing here is essentially controlling all other input variables but only bumping x_j to see the impact on the prediction $\hat{y}$. The coefficient β_{j} thus measures the sensitivity of the outcome to a specific feature. When we have a unit change, with $\Delta x_j = 1$, the change is exactly the coefficient itself, giving $\Delta\hat{y} = \beta_{\mathrm{j}}$.

The next section discusses the measure of the predictiveness of the MLR model.

Seeking a higher coefficient of determination?

MLR tends to perform better than SLR due to the multiple predictors used in the model, such as a higher coefficient of determination (R^2). However, a regression model with more input variables and a higher R^2 does not necessarily mean that the model is a better fit and can predict better for the test set.

A higher R^2, as a result of more input features, could likely be due to overfitting. Overfitting occurs when a model is excessively complex, including too many predictors or even interaction terms between predictors. In such cases, the model may fit the observed data well (thus leading to a high R^2), but it may perform poorly when applied to new, unseen test data. This is because the model might have learned not only the underlying structure of the training data, but also the random noise specific to the dataset.

Now we look at the metric of R^2 more closely. While R^2 measures how well the model explains the variance in the outcome variable, it has a major limitation: it tends to get bigger as more predictors enter the model, even if those predictors are irrelevant. As a remedy, we can use the adjusted R^2. Unlike R^2, the adjusted R^2 explicitly considers the number of predictors and adjusts the resulting statistic accordingly. If a predictor improves the model substantially, the adjusted R^2 will increase, but if a predictor does not improve the model by a significant amount, the adjusted R^2 may even decrease.

When building statistical models, simpler models are usually preferred when they perform similarly to more complex models. This principle of parsimony, also known as Occam's razor, suggests that among models with similar predictive power, the simplest one should be chosen. In other words, adding more predictors to the model makes it more complex, harder to interpret, and more likely to overfit.

More on adjusted R^2

The adjusted R^2 improves upon R^2 by adjusting for the number of features in the selected model. Specifically, the value of the adjusted R^2 only increases if adding this feature is worth more what would have been expected from adding a random feature. Essentially, the additional predictors added to the model must be meaningful and predictive to lead to a higher adjusted R^2. These additional predictors, however, would always increase the R^2 when added to the model.

Adjusted R^2 addresses this issue by incorporating the model's degree of freedom. Here, the degree of freedom refers to the number of values in a statistical calculation that is free to vary. In the context of regression models, this typically means the number of predictors. The Adjusted R^2 can be expressed as follows:

$$\text{Adjusted } R^2 = 1 - (1 - R^2)\frac{(n-1)}{n-p-1}$$

Where n denotes the number of observations, and p represents the number of features in the model.

The formula works by adjusting the scale of R^2 based on the count of observations and predictors. The term $\frac{(n-1)}{n-p-1}$ is a ratio that reflects the degrees of freedom in the model, where $(n-1)$ represents the total degrees of freedom in the model. We subtract by 1 because we are estimating the mean of the dependent variable from the data. $(n-p-1)$ represents the degrees of freedom for the error, representing the number of observations left over after estimating the model parameters. The whole term $(1-R^2)\frac{(n-1)}{n-p-1}$ denotes the error variance adjusted for the count of predictors.

Subtracting this from 1 results in the proportion of the total variance explained by the model, after adjusting for the number of predictors in the model. In other words, it's a version of R^2 that penalizes the addition of unnecessary predictors. This helps to prevent overfitting and makes adjusted R^2 a more balanced measure of a model's explanatory power when comparing models with different numbers of predictors.

Let us look at how to develop an MLR model in R.

Developing an MLR model

In this section, we will develop an MLR model using the same `lm()` function in R based on the `mtcars` dataset, which comes preloaded with R and was used in previous exercises. Again, the `mtcars` dataset contains measurements for 32 vehicles from a 1974 Motor Trend issue. These measurements include attributes like miles per gallon (`mpg`), number of cylinders (`cyl`), horsepower (`hp`), and weight (`wt`).

Exercise 12.2 Building an MLR model

In this exercise, we will develop an MLR model to predict `mpg` using `cyl`, `hp`, and `wt`. We will then interpret the model results.

1. Load the `mtcars` dataset and build an MLR that predicts mpg based on `cyl`, `hp`, and `wt`.

```
# Load the data
```

```
data(mtcars)
# Build the model
model = lm(mpg ~ cyl + hp + wt, data = mtcars)
```

Here, we first load the `mtcars` dataset using the `data()` function, then construct the MLR model using the `lm()` function. The formula `mpg ~ cyl + hp + wt` is used to specify the model. This formula tells R that we want to model `mpg` as a function of `cyl`, `hp`, and `wt`. The `data = mtcars` argument tells R to look for these variables in the `mtcars` dataset. The `lm()` function fits the model to the data and returns a model object, which we store in the variable model.

2. View the summary of the model.

```
# Print the summary of the model
>>> summary(model)
Call:
lm(formula = mpg ~ cyl + hp + wt, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-3.9290 -1.5598 -0.5311  1.1850  5.8986

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 38.75179    1.78686  21.687  < 2e-16 ***
cyl         -0.94162    0.55092  -1.709 0.098480 .
hp          -0.01804    0.01188  -1.519 0.140015
wt          -3.16697    0.74058  -4.276 0.000199 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 2.512 on 28 degrees of freedom
Multiple R-squared:  0.8431,     Adjusted R-squared:  0.8263
F-statistic: 50.17 on 3 and 28 DF,  p-value: 2.184e-11
```

The summary includes the model's coefficients (the intercept and the slopes for each predictor), the residuals (differences between the actual observations and predicted values

for the target), and several statistics that tell us how well the model fits the data, including R^2 and adjusted R^2.

Let us interpret the output. Each coefficient represents the expected change in `mpg` for a single unit increase in the associated predictor, assuming all other predictors constant. The R^2 value, which is 0.8431, denotes the proportion of variance (over 84%) in mpg that can be explained by the predictors together. Again, the adjusted R^2 value, which is 0.8263, is a modified R^2 that accounts for the number of features in the model.

In addition, the p-values for each predictor test the null hypothesis that the true value of the coefficient is zero. If a predictor's p-value is smaller than a preset significance level (such as 0.05), we would reject this null hypothesis and conclude that the predictor is statistically significant. In this case, `wt` is the only statistically significant factor compared with others using a significance level of 5%.

In the MLR model, all coefficients are negative, indicating a reverse direction of travel between the input variable and the target in the MLR model. However, we cannot conclude that all the predictors negatively correlate with the target variable. The correlation between the individual predictor and the target variable could be positive or negative in an SLR.

The next section provides more context on this phenomenon.

Introducing the Simpson's Paradox

Simpson's Paradox says that a trend appears in different data groups but disappears or changes when combined. In the context of regression analysis, Simpson's Paradox can appear when a variable that seems positively correlated with the outcome might be negatively correlated when we control for other variables.

Essentially, this paradox illustrates the importance of considering confounding variables and not drawing conclusions from aggregated data without understanding the context. The confounding variables are those not among the explanatory variables under consideration but are related to both the target variable and the predictors.

Let us consider a simple example through the following exercise.

Exercise 12.3 Illustrating the Simpson's Paradox

In this exercise, we will look at two scenarios with opposite signs of coefficient values for the same feature in both SLR and MLR.

1. Create a dummy dataset with two predictors and one output variable.

```
set.seed(123)
x1 = rnorm(100)
x2 = -3 * x1 + rnorm(100)
y = 2 + x1 + x2 + rnorm(100)
df = data.frame(y = y, x1 = x1, x2 = x2)
```

Here, `x1` is a set of 100 numbers randomly generated from a standard normal distribution. `x2` is a linear function of `x1` but with a negative correlation, and some random noise is added (via `rnorm(100)`). `y` is then generated as a linear function of `x1` and `x2`, again with some random noise added. All three variables are compiled into a DataFrame `df`.

2. Train a SLR model with `y` as the outcome and `x1` as the input features. Check the summary of the model.

```
# Single linear regression
single_reg = lm(y ~ x1, data = df)
>>> summary(single_reg)
Call:
lm(formula = y ~ x1, data = df)

Residuals:
    Min      1Q  Median      3Q     Max
-2.7595 -0.8365 -0.0564  0.8597  4.3211

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   2.0298     0.1379   14.72   <2e-16 ***
x1           -2.1869     0.1511  -14.47   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.372 on 98 degrees of freedom
Multiple R-squared:  0.6813,    Adjusted R-squared:  0.678
F-statistic: 209.5 on 1 and 98 DF,  p-value: < 2.2e-16
```

The result shows that `x1` is negatively correlated with `y` due to a negative coefficient of -2.1869.

3. Train a SLR model with `y` as the target and `x1` and `x2` as the input features. Check the summary of the model.

```
# Multiple linear regression
multi_reg = lm(y ~ x1 + x2, data = df)
>>> summary(multi_reg)
```

```
Call:
lm(formula = y ~ x1 + x2, data = df)


Residuals:
    Min      1Q  Median      3Q     Max
-1.8730 -0.6607 -0.1245  0.6214  2.0798


Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  2.13507    0.09614  22.208  < 2e-16 ***
x1           0.93826    0.31982   2.934  0.00418 **
x2           1.02381    0.09899  10.342  < 2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1


Residual standard error: 0.9513 on 97 degrees of freedom
Multiple R-squared:  0.8484,     Adjusted R-squared:  0.8453
F-statistic: 271.4 on 2 and 97 DF,  p-value: < 2.2e-16
```

The result shows that the estimated coefficient for `x1` is now a positive quantity. Does it suggest that `x1` is suddenly positively correlated with y? No, since there are likely other confounding variables that lead to a positive coefficient.

The key takeaway is, that we can only make inferences on the (positive or negative) correlation between a predictor and a target outcome in a SLR setting. For example, if we build an SLR model to regress `y` against `x`, we can conclude that `x` and `y` are positively correlated if the resulting coefficient is positive ($\beta > 0$). Similarly, if $\beta > 0$, we can conclude that x and y are positively correlated. The same applies to the case of negative correlation.

However, such inference breaks in an MLR setting. That is, we cannot conclude a positive correlation if $\beta > 0$, and vice versa.

Let us also take the opportunity to interpret the results here. The `Estimate` column shows the estimated regression coefficients. These values indicate how much the `y` variable is expected to increase when the corresponding predictor variable increases by one unit while holding all other features constant. In this case, for each unit increase in `x1`, `y` is expected to increase by approximately 0.93826 units, and for each unit increase in `x2`, `y` is expected to increase by approximately 1.02381 units. The `(Intercept)` row shows the estimated value of `y` when all predictor variables in the model are zero. In this model, the estimated intercept is 2.13507.

The `Std. Error` represents the standard errors for the estimates. Smaller values here indicate more precise estimates. The `t value` column shows the t-statistics for the hypothesis test that the

corresponding population regression coefficient is zero, given that the other predictors are in the model. A larger absolute value of the t-statistic indicates stronger evidence against the null hypothesis. The `Pr(>|t|)` column gives the p-values for the hypothesis tests. In this case, both `x1` and `x2` have p-values below 0.05, indicating that both are statistically significant predictors of `y` at the 5% significance level.

Finally, the Multiple R-squared and Adjusted R-squared values provide measures of how well the model fits the data. The Multiple R-squared value is 0.8484, indicating that this model explains approximately 84.84% of the variability in `y`. The Adjusted R-squared value adjusts this measure for the number of features in the model. As discussed, it is a better measure when comparing models with different numbers of predictors. Here, the Adjusted R-squared value is 0.8453. The F-statistic and its associated p-value are used to test the hypothesis that all population regression coefficients are zero. A small p-value (less than 0.05) indicates that we can reject this hypothesis, and conclude that at least one of the predictors is useful in predicting `y`.

The next section looks at the situation when we have a categorical predictor in the MLR model.

Working with categorical variables

In MLR, the process of including a binary predictor is similar to including a numeric predictor. However, the interpretation differs. Consider a dataset where y is the target variable, x_1 is a numeric predictor, and x_2 is a binary predictor.

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \epsilon$$

In this model, x_2 is coded as 0 and 1, and its corresponding coefficient β_2 represents the difference in the mean values of y between the two groups identified by x_2.

For example, if x_2 is a binary variable representing sex (0 for males, 1 for females), and y is salary, then β_2 represents the average difference in salary between females and males, after accounting for the value of x_1.

Note that the interpretation of the coefficient of a binary predictor is dependent on the other variables in the model. So in the example above, β_2 is the difference in salary between females and males for given values of x_1.

On the implementation side, R automatically creates dummy variables when a factor is used in a regression model. So, if x_2 were a factor with levels "male" and "female", R would handle the conversion to 0 and 1 internally when fitting the model.

Let us look at a concrete example. In the following codes, we build an MLR to predict mpg using `qsec` and `am`.

```
# Fit the model

model <- lm(mpg ~ qsec + am, data = mtcars)

# Display the summary of the model

>>> summary(model)

Call:

lm(formula = mpg ~ qsec + am, data = mtcars)
```

```
Residuals:
    Min      1Q  Median      3Q     Max
-6.3447 -2.7699  0.2938  2.0947  6.9194

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -18.8893     6.5970  -2.863  0.00771 **
qsec          1.9819     0.3601   5.503 6.27e-06 ***
am            8.8763     1.2897   6.883 1.46e-07 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 3.487 on 29 degrees of freedom
Multiple R-squared:  0.6868,     Adjusted R-squared:  0.6652
F-statistic:  31.8 on 2 and 29 DF,  p-value: 4.882e-08
```

Note that the variable `am` is treated as a numeric variable. Since it represents the type of transmission in the car (0 = automatic, 1 = manual), it should have been treated as a categorical variable. This can be achieved by converting it to a factor, as shown in the following.

```
# Convert am to categorical var
mtcars$am_cat = as.factor(mtcars$am)
# Fit the model
model <- lm(mpg ~ qsec + am_cat, data = mtcars)
# Display the summary of the model
>>> summary(model)
Call:
lm(formula = mpg ~ qsec + am_cat, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-6.3447 -2.7699  0.2938  2.0947  6.9194

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
```

```
(Intercept) -18.8893     6.5970  -2.863  0.00771 **
qsec          1.9819     0.3601   5.503 6.27e-06 ***
am_cat1       8.8763     1.2897   6.883 1.46e-07 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1


Residual standard error: 3.487 on 29 degrees of freedom
Multiple R-squared:  0.6868,     Adjusted R-squared:  0.6652
F-statistic:  31.8 on 2 and 29 DF,  p-value: 4.882e-08
```

Note that there is only one variable `am_cat1` created for the categorical variable `am_cat`. This is because `am_cat` is binary, thus we only need one dummy column (keeping only `am_cat1` and removing `am_cat0` in this case) to represent the original categorical variable. In general, for a categorical variable with k categorical, R will automatically create $(k-1)$ dummy variables in the model.

This process is called **one-hot encoding**, which involves converting categorical data into a format usable by the regression model. For categorical variables where there is no ordinal relationship, the integer encoding (as originally used) may not be sufficient. In fact, it could even be misleading to some extent. For these cases, one-hot encoding can be applied, where each unique original value is replaced by a binary variable. Specifically, each of these new variables (also known as "dummy" variables) would have a value of 1 for the observations where am was equal to the corresponding level, and 0 otherwise. This essentially creates a set of indicators that capture the presence or absence of each category. Finally, since we are able to infer the last dummy variable based on the values of the previous (k-1) dummy variables, we can remove the last dummy variable in the resulting one-hot encoded set of variables.

Using a categorical variable introduces a vertical shift to the model estimate, as discussed in the following section. To see this, let us look more closely at the impact of the categorical variable am_cat1. Our MLR model now assumes the following form:

$$\hat{y} = \beta_0 + \beta_1 x_{qsec} + \beta_2 x_{am\_cat}$$

We know that x_{am_cat} is a binary variable. When $x_{am\_cat} = 0$, the prediction becomes:

$$\hat{y} = \beta_0 + \beta_1 x_{qsec}$$

When $x_{am\_cat} = 1$, the prediction is

$$\hat{y} = \beta_0 + \beta_1 x_{qsec} + \beta_2 = (\beta_0 + \beta_2) + \beta_1 x_{qsec}$$

Comparing these two quantities, we find that they are two linear models parallel to each other since the slope is the same and the only difference is β_2 in the intercept term.

A visual illustration helps here. In the following code snippet, we first create a new DataFrame `newdata` that covers the range of `qsec` values in the original data, for each of `am_cat` values (0 and 1). We then use the `predict()` function to get the predicted `mpg` values from the model for this new data. Next, we plot the original data points with `geom_point()`, and add two regression lines with `geom_line()`, where the lines are based on the predicted values in `newdata`. The

`color = am_cat` aesthetic setting adds different colors to the points and lines for the different `am_cat` values, and the labels are adjusted in `scale_color_discrete()` so that 0 corresponds to "Automatic" and 1 corresponds to "Manual".

```
# Load required library
library(ggplot2)
# Create new data frame for the predictions
newdata = data.frame(qsec = seq(min(mtcars$qsec), max(mtcars$qsec),
length.out = 100),
                     am_cat = c(rep(0, 100), rep(1, 100)))
newdata$am_cat = as.factor(newdata$am_cat)
# Get predictions
newdata$mpg_pred = predict(model, newdata)
# Plot the data and the regression lines
ggplot(data = mtcars, aes(x = qsec, y = mpg, color = am_cat)) +
  geom_point() +
  geom_line(data = newdata, aes(y = mpg_pred)) +
  labs(title = "mpg vs qsec by Transmission Type",
       x = "Quarter Mile Time (qsec)",
       y = "Miles per Gallon (mpg)",
       color = "Transmission Type") +
  scale_color_discrete(labels = c("Automatic", "Manual")) +
  theme(text = element_text(size = 16),  # Default text size
        title = element_text(size = 15),  # Title size
        axis.title = element_text(size = 18),  # Axis title size
        legend.title = element_text(size = 16),  # Legend title size
        legend.text = element_text(size = 16), # Legend text size
        legend.position = "bottom")  # Legend position
```

Running the codes generates *Figure 12.3*.

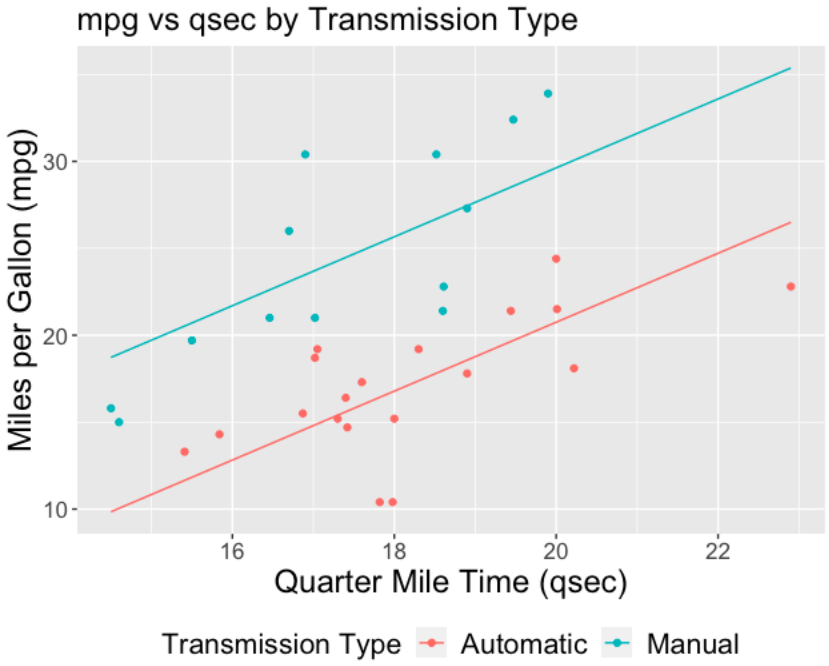

Figure 12.3 – Visualizing the two linear regression models based on different transmission types. These two lines are parallel to each other due to a shift in the intercept term.

What this figure suggests is that manual transmission cars have the same miles per gallon (`mpg`) more than automatic transmission cars given the same quarter-mile time (`qsec`). However, this is unlikely in the case in practice, since different car types (manual versus automatic) will likely have different quarter-mile time. In other words, there is an interaction effect between these two variables.

The following section Introduces the interaction term as a remedy to this situation.

Introducing the interaction term

In a regression analysis, an interaction occurs when the effect of one predictor on the target variable differs depending on the level of another predictor variable. In our running example, we are essentially looking at whether the relationship between `mpg` and `qsec` is different for different values of `am`. In other words, we are testing whether the slope of the line relating `mpg` and `qsec` differs for manual (`am`=1) and automatic (`am`=0) transmissions.

For example, if there is no interaction effect, then the impact of `qsec` on `mpg` is the same regardless of whether the car has a manual or automatic transmission. This would mean that the lines depicting the relationship between `mpg` and `qsec` for manual and automatic cars would be parallel.

If there is an interaction effect, then the effect of `qsec` on `mpg` differs for manual and automatic cars. This would mean that the lines depicting the relationship between `mpg` and `qsec` for manual and automatic cars would not be parallel. They could either cross or, more commonly, just have different slopes.

In order to depict these differences in relationships, we can add an interaction term to the model, which is done using the `*` operator. For example, the formula for a regression model with an interaction between `qsec` and `am_cat` would be `mpg ~ qsec * am_cat`. This is equivalent to

`mpg ~ qsec + am_cat + qsec:am_cat`, where `qsec:am_cat` represents the interaction term. The following codes show the details.

```
# Adding interaction term
model_interaction <- lm(mpg ~ qsec * am_cat, data = mtcars)
# Print model summary
>>> summary(model_interaction)
Call:
lm(formula = mpg ~ qsec * am_cat, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-6.4551 -1.4331  0.1918  2.2493  7.2773

Coefficient s:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   -9.0099     8.2179  -1.096  0.28226
qsec           1.4385     0.4500   3.197  0.00343 **
am_cat1      -14.5107    12.4812  -1.163  0.25481
qsec:am_cat1   1.3214     0.7017   1.883  0.07012 .
Signif. code s:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 3.343 on 28 degrees of freedom
Multiple R-squared:  0.722, Adjusted R-squared:  0.6923
F-statistic: 24.24 on 3 and 28 DF,  p-value: 6.129e-08
```

Let us also plot the updated model that consists of two intersecting lines due to the interaction effect. In the following code snippet, `geom_smooth(method =“"l”", se = FALSE)` is used to fit different linear lines to each group of points (automatic and manual cars). `as.factor(am_cat)` is used to treat `am_cat` as a factor (categorical) variable so that a separate line is fit for each category.

```
# Create scatter plot with two intersecting lines
ggplot(mtcars, aes(x = qsec, y = mpg, color = as.factor(am_cat))) +
  geom_point() +
  geom_smooth(method =“"l”", se = FALSE) + # fit separate regression
lines per group
```

```
  scale_color_discrete(name =""Transmission Typ"",
                       labels = c""Automati"",""Manua"")) +
  labs(x =""Quarter mile time (seconds"",
       y =""Miles per gallo"",
       title =""Separate regression lines fit for automatic and
manual car"") +
  theme(text = element_text(size = 16),
        title = element_text(size = 15),
        axis.title = element_text(size = 20),
        legend.title = element_text(size = 16),
        legend.text = element_text(size = 16))
```

Running the codes generates *Figure 12.4*.

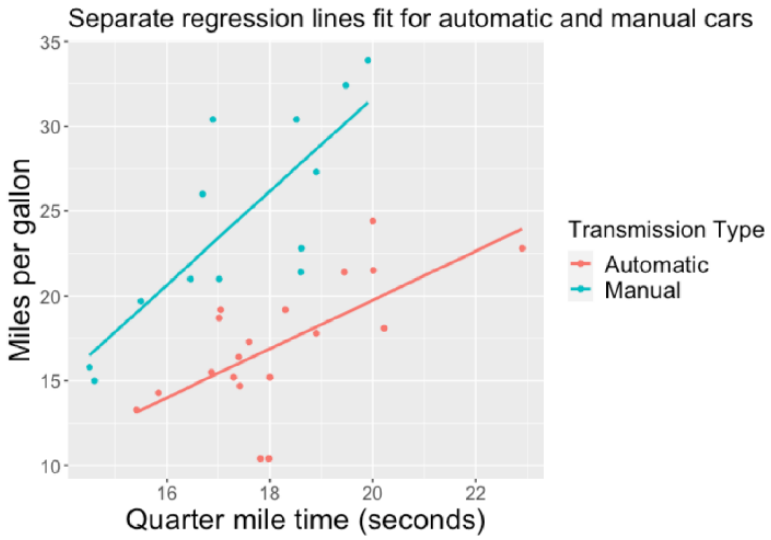

Figure 12.4 – Two intersecting lines due to the intersection term between quarter mile time and transmission type.

The next section focuses on another related topic: nonlinear terms.

Handling nonlinear terms

Linear regression is a widely used model for understanding the linear relationships between a response and explanatory variables. However, not all underlying relationships in the data are linear. In many situations, a feature and a response variable may not have a straight-line relationship, thus necessitating the handling of nonlinear terms in the linear regression model to increase the flexibility of the model.

The simplest way to incorporate nonlinearity, and therefore build a curve instead of a straight line is by including polynomial terms of predictors in the regression model. In a polynomial regression, some or all predictors are raised to a specific polynomial term. For example, transforming a feature x into x^2 or x^3.

Let us go through an exercise to understand the impact of adding polynomial features in a linear regression model.

Exercise 12.4 Adding polynomial features to a linear regression model

In this exercise, we will create a simple dataset where the relationship between the input feature x and the target variable y is not linear, but quadratic. We will first fit a SLR model, then add a quadratic term and compare the results.

1. Generate a sequence of x values ranging from -10 to 10. For each x, compute the corresponding y as the square of x plus some random noise to show a (noisy) quadratic relationship. Put x and y in a DataFrame.

```
# Create a quadratic dataset
set.seed(1)
x = seq(-10, 10, by = 0.5)
y = x^2 + rnorm(length(x), sd = 5)
# Put it in a dataframe
df = data.frame(x = x, y = y)
Fit a simple linear regression to the data and print the
summary.
lm1 <- lm(y ~ x, data = df)
>>> summary(lm1)
Call:
lm(formula = y ~ x, data = df)


Residuals:
    Min      1Q  Median      3Q     Max
-43.060 -29.350  -5.451  19.075  64.187


Coefficient s:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 35.42884    5.04627   7.021 2.01e-08 ***
x           -0.04389    0.85298  -0.051    0.959
Signif. code s:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' '
1


Residual standard error: 32.31 on 39 degrees of freedom
```

```
Multiple R-squared:  6.787e-05,  Adjusted R-squared:  -0.02557
F-statistic: 0.002647 on 1 and 39 DF,  p-value: 0.9592
```

The result suggests a not-so-good model fitting to the data, which possesses a nonlinear relationship.

2. Fit a quadratic model to the data by including x^2 as a predictor using the `I()` function. Print the summary of the model.

```
lm2 <- lm(y ~ x + I(x^2), data = df)
>>> summary(lm2)
Call:
lm(formula = y ~ x + I(x^2), data = df)

Residuals:
    Min      1Q  Median      3Q     Max
-11.700  -2.134  -0.078   2.992   7.247

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.49663    1.05176   0.472    0.639
x           -0.04389    0.11846  -0.370    0.713
I(x^2)       0.99806    0.02241  44.543   <2e-16 ***
Signif. code s:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’
1

Residual standard error: 4.487 on 38 degrees of freedom
Multiple R-squared:  0.9812,     Adjusted R-squared:  0.9802
F-statistic: 992.1 on 2 and 38 DF,  p-value: < 2.2e-16
```

The result shows that the polynomial model fits the data better than the simple linear model. Thus, adding nonlinear terms can improve model fit when the relationship between predictors and the response is not strictly linear.

3. Plot the linear and quadratic models together with the data.

```
ggplot(df, aes(x = x, y = y)) +
  geom_point() +
```

```
  geom_line(aes(y = linear_pred), color =""blu"", linetype
=""dashe"") +
  geom_line(aes(y = quadratic_pred), color =""re"") +
  labs(title =""Scatter plot with linear and quadratic fit"",
       x =""",
       y =""") +
  theme(text = element_text(size = 15)) +
  scale_color_discrete(name =""Mode"",
                       labels = c(""Linear Mode"",""Quadratic
Mode"")) +
  annotate""tex"", x = 0, y = 40, label =""Linear Mode"",
color =""blu"") +
  annotate""tex"", x = 6, y = 80, label =""Quadratic Mode"",
color =""re"")
```

Here, we first calculate the predicted values for both models and add them to the DataFrame. We then create a scatter plot of the data and add two lines representing the predicted values from the linear model (in blue) and the quadratic model (in red).

Running the codes generates *Figure 12.5*. The result suggests that adding a polynomial feature could extend the flexibility of a linear model.

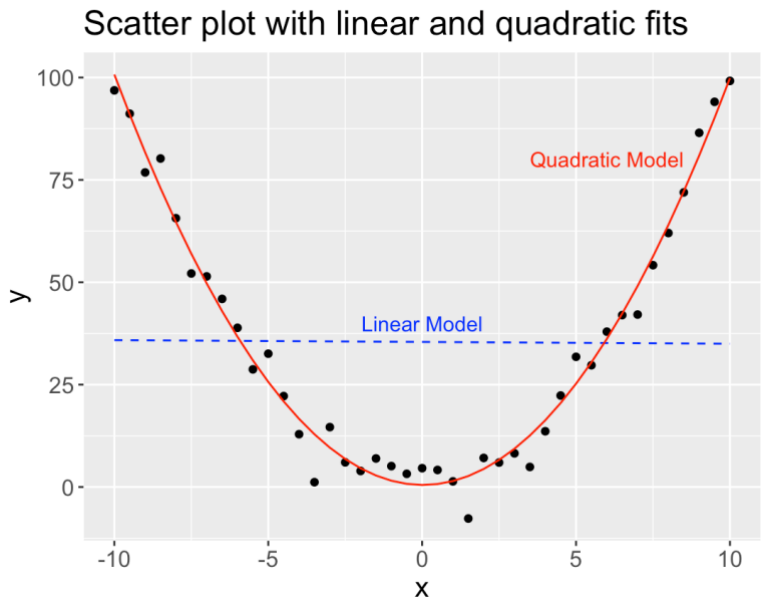

Figure 12.5 – Visualizing the linear and quadratic fits to the nonlinear data.

Other common ways to introduce nonlinearity include the logarithmic transformation ($\log x$) or a square root transformation ($\sqrt{x}$). These transformations can also be applied to the target variable y, and we can have multiple transformed features in the same linear model.

Note that the model with transformed features still remains a linear model. If there is a nonlinear transformation to the coefficients, the model would be a nonlinear one.

The next section sheds more light on a widely used type of transformation: the logarithmic transformation.

More on the logarithmic transformation

The logarithmic transformation, or log transformation, maps an input to the corresponding output based on the logarithmic function, giving $y = \log x$. A popular reason behind using such a transformation is to introduce nonlinearity in the linear regression model. When the relationship between the input features and the target output is nonlinear, applying a transformation can sometimes linearize the relationship, making it possible to model the relationship with a linear regression model. For the logarithmic transformation, it can help when the rate of change in the outcome variable increases or decreases as the value of the predictor increases.

To be specific, the rate of change decreases as the input becomes more extreme. The natural consequence of such transformation is that potential outliers in the input data are squeezed so that they appear less extreme in the transformed column. In other words, the resulting linear regression model will be less sensitive to the original outliers due to the log transformation.

Another side benefit of using log transformation is its ability to deal with heteroscedasticity. Heteroscedasticity is when the variability of the error term in a regression model is not constant across all levels of the predictors. This violates one of the assumptions of linear regression models and can lead to inefficient and biased estimates. In this case, log transformations can stabilize the variance of the error term by shrinking the potential big error terms, making it more constant across different levels of the predictors.

Lastly, when the relationship between predictors and the outcome is multiplicative rather than additive, taking the log of the predictors and/or the outcome variable can convert the relationship into an additive one, which can be modeled using linear regression.

Le''s consider an example where we predict the miles per gallon (`mpg`) from horsepower (`hp`). W''ll compare the model where we predict mpg directly from `hp`, and another model where we predict the log of `mpg` from `hp`, as shown in the following code snippet.

```
# Fit the original model
model_original = lm(mpg ~ hp, data = mtcars)


# Fit the log-transformed model
mtcars$log_mpg = log(mtcars$mpg)
model_log = lm(log_mpg ~ hp, data = mtcars)


# Predictions from the original model
mtcars$pred_original = predict(model_original, newdata = mtcars)
```

```
# Predictions from the log-transformed model (back-transformed to
the original scale using exp)

mtcars$pred_log = exp(predict(model_log, newdata = mtcars))


library(tidyr)

library(dplyr)


# Reshape data to long format

df_long <- mtcars %>%

  gather(key =""Mode”", value =""Predictio”", pred_original,
pred_log)


# Create plot

ggplot(df_long, aes(x = hp, y = mpg)) +

  geom_point(data = mtcars, aes(x = hp, y = mpg)) +

  geom_line(aes(y = Prediction, color = Model)) +

  labs(

    x =""Horsepower (hp)"",

    y =""Miles per gallon (mpg)"",

    color =""Mode""

  ) +

  scale_color_manual(values = c""pred_origina”" =""blu"",""pred_lo""
=""re""")) +

  theme(

    legend.position =""botto"",

    text = element_text(size = 16),

    legend.title = element_text(size = 16),

    axis.text = element_text(size = 16),  # control the font size of
axis labels

    legend.text = element_text(size = 16)  # control the font size
of legend text

  )
```

Running the codes generates *Figure 12.6*, where we notice a slight curvature in the blue:

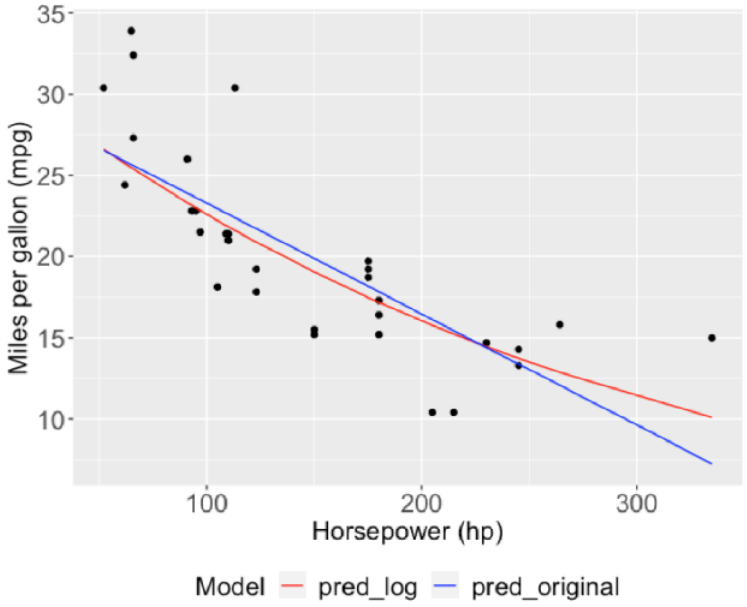

Figure 12.6 – Visualizing the original and log-transformed model.

Note that log transformation can only be applied to positive data. In the case of `mtcars$mpg`, all values are positive, so we can safely apply the log transformation. If the variable included zero or negative values, we would need to consider a different transformation or approach.

The next section focuses on deriving and using the closed-form solution to the linear regression model.

Working with the closed-form solution

When developing a linear regression model, the available training set $(\mathbf{X}, \mathbf{y})$ is given, and the only unknown parameters are the coefficients $\boldsymbol{\beta}$. Here, a bold lower case letter means a vector (such as $\boldsymbol{\beta}$ and $\mathbf{y}$), and a bold upper case letter denotes a matrix (such as $\mathbf{X}$). It turns out that the closed-form solution to a linear regression model can be derived using the concept of Ordinary Least Squares (OLS) estimator, which aims to minimize the sum of the squared residuals in the model. Having the closed-form solution means we can simply plug in the required elements (in this case, $\mathbf{X}$ and $\mathbf{y}$) and perform the calculation to obtain the solution, without resorting to any optimization procedure.

Specifically, given a data matrix $\mathbf{X}$ (which includes a column of ones for the intercept term and is in bold to indicate more than one feature) of dimensions $n \times p$ (where n is the number of observations and p is the number of predictors) and a response vector $\mathbf{y}$ of length n, the OLS estimator for the coefficient vector $\boldsymbol{\beta}$ is given by the formula:

$$\boldsymbol{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

This solution assumes that the term $(\mathbf{X}^T\mathbf{X})$ is invertible, meaning it should be a full-rank matrix. If this is not the case, the solution either does not exist or is not unique.

Now let us look at how to derive this solution. We start with the minimization problem for the least squares: minimizing $(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})$ over $\boldsymbol{\beta}$. This quadratic form can be expanded to $\mathbf{y}^T\mathbf{y} - \boldsymbol{\beta}^T\mathbf{X}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}\boldsymbol{\beta} + \boldsymbol{\beta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\beta}$. Note that $\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{y} = \mathbf{y}^T\mathbf{X}\boldsymbol{\beta}$ since both terms are scalars and therefore are equal to each other after the transpose operation. We can then write the **residual sum of squares** (**RSS**) expression as $\mathbf{y}^T\mathbf{y} - 2\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{y} + \boldsymbol{\beta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\beta}$.

Here we apply the first-order condition to solve for the value of $\boldsymbol{\beta}$ that minimizes this expression (recall that the point that either minimizes or maximizes a graph has a derivative of 0). This means that we would set its first derivative to zero, leading to:

$$\frac{\partial(\mathbf{y}^T\mathbf{y} - 2\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{y} + \boldsymbol{\beta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\beta})}{\partial\boldsymbol{\beta}} = -2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} = \mathbf{0}$$

$$\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} = \mathbf{X}^T\mathbf{y}$$

$$\boldsymbol{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Thus, we have derived the closed-form solution of $\boldsymbol{\beta}$ that minimizes the sum of the squared residuals. Let us go through an example to see how it can be implemented.

Implementing the closed-form solution

Let us look at implementing the **ordinary least squares** (**OLS**) estimation in R for a SLR model. An example using synthetic data is shown in the following code snippet.

```
# Set seed for reproducibility
set.seed(123)
# Generate synthetic data
n = 100 # number of observations
x = runif(n, -10, 10) # predictors
beta0 = 2 # intercept
beta1 = 3 # slope
epsilon = rnorm(n, 0, 2) # random error term
y = beta0 + beta1*x + epsilon # response variable
# Design matrix X
X = cbind(1, x)
```

Here, we generate 100 observations with a single input feature, where the observation and noise-perturbed and follows a process given by $y = \beta_0 + \beta_1 x + \epsilon$. The error term assumes a normal distribution parameterized by a mean of 0 and a standard deviation 2.

Before proceeding to the estimation, note that we also appended a column of 1s on the left of the input feature `x` to form a matrix `X`. This column of 1s is used to indicate the intercept term and is often referred to as the bias trick. That is, the coefficient β_0 for the intercept term will be part of the coefficient vector, and there is no need to create a separate coefficient just for the intercept.

Let us calculate the result using the closed-form solution.

Calculate beta using closed-form solution

```
beta_hat = solve(t(X) %*% X) %*% t(X) %*% y
>>> print(beta_hat)
       [,1]
  1.985344
```

```
x 3.053152
```

Here, `%*%` is used for matrix multiplication, `t(X)` is the transpose of `X`, and `solve()` is used to calculate the inverse of a matrix.

We can also run the linear regression procedure using the `lm()` function for comparison.

```
# Fit linear regression model for comparison
model = lm(y ~ x)
>>> print(coef(model))
(Intercept)           x
   1.985344    3.053152
```

The results are exactly the same as the ones obtained via the manual approach.

The next two sections cover two common issues in linear regression settings: multicollinearity and heteroskedasticity.

Dealing with multicollinearity

Multicollinearity refers to the case when two (or more) predictors are highly correlated in a multiple regression model. This means that one independent variable can be linearly predicted from the others with a high degree of accuracy. This is a situation in which we do not want to fall into. In other words, we would like to see a high degree of correlation between the predictors and the target variable, while a low degree of correlation among these predictors themselves.

In the face of multicollinearity in a linear regression model, the resultant model tends to generate unreliable and unstable estimates of the regression coefficients. It can inflate the coefficients of the parameters, making them statistically insignificant, even though they might be substantively important. In addition, multicollinearity makes it difficult to assess the effect of each independent variable on the dependent variable, as the effects are intertwined with each other. However, it does not affect the predictive power or interpretability of the model; instead, it only changes the calculations for individual features.

Detecting any potential multicollinearity among the predictors can be performed by examining the pair-wise correlation. Alternatively, we can resort to a particular test statistic called the Variance Inflation Factor (VIF), which quantifies how much the variance is increased due to multicollinearity. A VIF of 1 indicates that two variables are not correlated, while a VIF greater than 5 (in many fields) would suggest a problematic amount of multicollinearity.

When multicollinearity exists in the linear regression model, we could choose to keep one predictor only and remove all other highly correlated predictors. We can also combine these correlated variables into a few uncorrelated ones via the Principle Component Analysis (PCA), a widely used technique for dimension reduction. Besides, we can resort to ridge regression to control the magnitude of the coefficients, as introduced in a later section of the chapter.

To check multicollinearity using VIF, we can use the `vif()` function from the `car` package, as shown in the following code snippet.

```
# install the package if not already installed
if(!require(car)) install.packages('car')
```

```
# load the package

library(car)

# fit a linear model

model = lm(mpg ~ hp + wt + disp, data = mtcars)

# calculate VIF

vif_values = vif(model)

>>> print(vif_values)

      hp       wt     disp

2.736633 4.844618 7.324517
```

Looking at the result, `disp` seems to have high multicollinearity (VIF = 7.32 > 5), suggesting that it has a strong correlation with `hp` and `wt`. This implies that `disp` is not providing much information that is not already contained in the other two predictors.

To handle the multicollinearity here, we can consider removing `disp` from your model since it has the highest VIF, applying PCA to combine the three predictors, or using ridge or lasso regression (more on this in the last two sections of this chapter).

The next section focuses on the issue of heteroskedasticity.

Dealing with heteroskedasticity

Heteroskedasticity (or heteroscedasticity) refers to the situation in which the variability of the error term, or residuals, is not the same across all levels of the independent variables. This violates one of the key assumptions of OLS linear regression, which assumes that the residuals have a constant variance. In other words, the residuals are homoskedastic. Violating this assumption could lead to incorrect inferences on the statistical significance of the coefficients, since the resulting standard errors of the regression coefficients could be larger or smaller than they should be.

There are a few ways to handle heteroskedasticity. We can transform the outcome variable using the logarithmic function as introduced earlier. Other functions, such as taking the square root or inverse of the original outcome variable, could also help reduce heteroskedasticity. Advanced regression models such as the **weighted least squares** (**WLS**) or **generalized least squares** (**GLS**) may also be explored to reduce the impact of heteroskedasticity.

To formally test for heteroskedasticity, we can conduct a Breusch-Pagan test using the `bptest()` function from the `lmtest` package. In the following code snippet, we fit an MLR model to predict `mpg` using `wt` and `hp`, followed by performing the Breusch-Pagan test.

```
# Load library

library(lmtest)

# Fit a simple linear regression model on mtcars dataset

model = lm(mpg ~ wt + hp, data = mtcars)

# Perform a Breusch-Pagan test to formally check for
heteroskedasticity
```

```
>>> bptest(model)

      studentized Breusch-Pagan test


data:  model

BP = 0.88072, df = 2, p-value = 0.6438
```

Since the p-value (0.6438) is greater than 0.05, we do not reject the null hypothesis of the Breusch-Pagan test. This suggests that there is not enough evidence to say that heteroskedasticity is present in the regression model. We would then conclude that the variances of the residuals are not significantly different from being constant, or homoskedastic.

The next section shifts to the regularized linear regression models using ridge and lasso penalties.

Introducing penalized linear regression

Penalized regression models, such as Ridge and Lasso, are techniques used to handle problems such as multicollinearity, reduce overfitting, and even perform variable selection, especially when dealing with high-dimensional data with multiple input features.

Ridge regression (also called L2 regularization) is a method that adds a penalty equivalent to the square of the magnitude of coefficients. We would add this term into the loss function after weighting it by an additional hyperparameter, often denoted as λ, to control the strength of the penalty term.

Lasso regression (L1 regularization), on the other hand, is a method that, similar to the ridge regression, adds a penalty for non-zero coefficients, but unlike ridge regression, it can force some coefficients to be exactly equal to zero when the penalty tuning parameter is large enough. The larger the value of the hyperparameter λ, the greater the amount of shrinkage. The penalty on the size of coefficients helps to reduce the model complexity and multicollinearity, leading to a model that can generalize better on unseen data. However, ridge regression includes all the features in the final model, thus it doesn't induce any sparsity. Therefore, it's not particularly useful for variable selection.

In summary, Ridge and Lasso regression are both penalized linear regression methods that add a constraint on the magnitude of the estimated coefficients to the model optimization process, which helps to prevent overfitting, manage multicollinearity and reduce model complexity. However, Ridge tends to include all predictors in the model and helps to reduce their effect, while Lasso can exclude predictors from the model altogether, leading to a simpler and more interpretable model.

Let us start with ridge regression and look at its loss function more closely.

Working with ridge regression

Ridge Regression, also referred to as L2 regularization, is a commonly used technique to alleviate overfitting in linear regression models by penalizing the magnitude of the estimated coefficients in the resulting model.

Recall that in a standard linear regression model, we seek to minimize the sum of the squared differences between our predicted and actual values, which we refer to as the least squares method. The loss function we wish to minimize is the RSS:

$$RSS = \sum_{i=1}^{n} \left(y_i - \left(\beta_0 + \sum_{j=1}^{p} \beta_j x_{ij} \right) \right)^2$$

where y_i is the actual target value, β_0 is the intercept term, $\{\beta_j\}$ are the coefficient estimates for each predictor x_{ij}, and the summations are over all observations and predictors.

Purely minimizing the RSS would tend to end up with an overfitting model, as represented by the high magnitude of the resulting coefficients. As a remedy, we could apply ridge regression by adding a penalty term to this loss function. This penalty term is the sum of the squares of each coefficient multiplied by a tuning parameter λ. The ridge regression loss function (also known as the **cost function**) is then:

$$L_{ridge} = RSS + \lambda \sum_{j=1}^{p} \beta_j^2$$

Here, the λ parameter is a user-defined tuning parameter. A larger λ means a higher penalty, and a smaller λ means less regularization effect. $\lambda = 0$ gives the ordinary least squares regression result, while as λ approaches infinity, the impact of the penalty term dominates, and the coefficient estimates would approach zero.

By adding this penalty term, ridge regression tends to decrease the size of the coefficients, which can help to mitigate the problem of multicollinearity (where predictors are highly correlated). It does this by spreading the coefficient estimates of correlated predictors across each other, which can lead to a more stable and interpretable model.

However, it's important to note that ridge regression does not typically produce sparse solutions and does not perform variable selection. In other words, it will not result in a model where some coefficients are exactly zero (unless λ is infinite), thus all predictors are included in the model. If feature selection is important, methods like Lasso (L1 regularization) or Elastic Net (a combination of L1 and L2 regularization) might be more appropriate.

Note that we would often note penalize the intercept β_0.

Let us go through the following exercise to look at how to develop a ridge regression model.

Exercise 12.5 Implementing ridge regression

In this exercise, we will implement a ridge regression model and compare the estimated coefficients with the OLS model. Our implementation will be based on the `glmnet` package.

1. Install and load the `glmnet` package.

```
# install the package if not already installed
if(!require(glmnet)) install.packages('glmnet')
library(glmnet)
```

   Here, we use an `if-else` statement to detect if the `glmnet` package is installed.

2. Store all columns other than `mpg` as predictors in `X` and the `mpg` as the target variable in `y`.

```
# Prepare data
data(mtcars)
X = as.matrix(mtcars[, -1]) # predictors
y = mtcars[, 1] # response
```

3. Fit a ridge regression model using the `glmnet()` function.

```
# Fit ridge regression model
set.seed(123) # for reproducibility
ridge_model = glmnet(X, y, alpha = 0)
```

Here, the alpha parameter controls the type of model we fit. `alpha = 0` fits a ridge regression model, `alpha = 1` fits a lasso model, and any value in between fits an elastic net model.

4. Use cross-validation to choose the best lambda value.

```
# Use cross-validation to find the optimal lambda
cv_ridge = cv.glmnet(X, y, alpha = 0)
best_lambda = cv_ridge$lambda.min
>>> best_lambda
2.746789
```

Here, we use the cross-validation approach to identify the optimal lambda that gives the lowest error on the cross-validation set on average. All repeated cross-validation steps are completed via the `cv.glmnet()` function.

5. Fit a new ridge regression model using the optimal lambda and extract its coefficients without the intercept.

```
# Fit a new ridge regression model using the optimal lambda
opt_ridge_model = glmnet(X, y, alpha = 0, lambda =
best_lambda)
# Get coefficients
ridge_coefs = coef(opt_ridge_model)[-1]  # remove intercept
>>> ridge_coefs
[1] -0.371840170 -0.005260088 -0.011611491  1.054511975 -
1.233657799  0.162231830
 [7]  0.771141047  1.623031037  0.544153807 -0.547436697
```

6. Fit a linear regression model and extract its coefficients.

```
# Ordinary least squares regression

ols_model = lm(mpg ~ ., data = mtcars)

# Get coefficients

ols_coefs = coef(ols_model)[-1] # remove intercept

>>> ols_coefs

        cyl         disp           hp         drat           wt
qsec           vs

-0.11144048  0.01333524 -0.02148212  0.78711097 -3.71530393
0.82104075  0.31776281

         am         gear         carb

 2.52022689  0.65541302 -0.19941925
```

7. Plot the coefficients of both models on the same graph.

```
plot(1:length(ols_coefs), ols_coefs, type="b", col="blue",
pch=19, xlab="Coefficient", ylab="Value",
ylim=c(min(ols_coefs, ridge_coefs), max(ols_coefs,
ridge_coefs)))

lines(1:length(ridge_coefs), ridge_coefs, type="b", col="red",
pch=19)

legend("bottomright", legend=c("OLS", "Ridge"), col=c("blue",
"red"), pch=19)
```

Running the codes generates *Figure 12.7*.

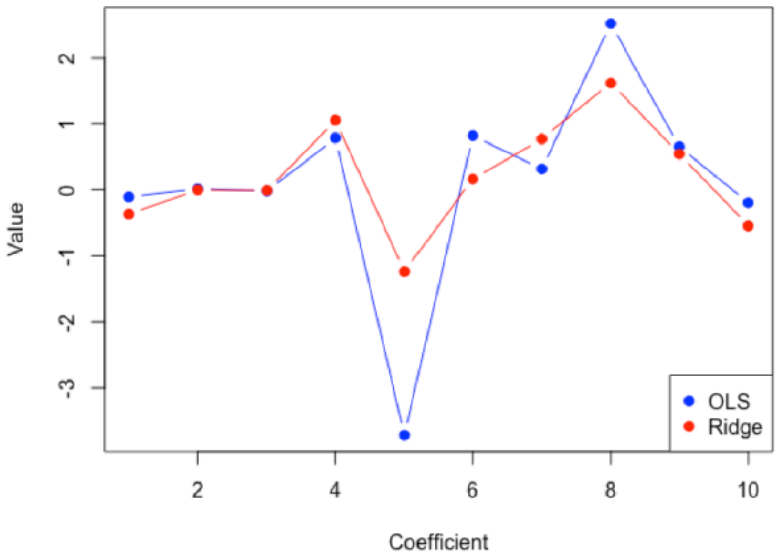

Figure 12.7 – Visualizing the estimated coefficients from the ridge and OLS models.

The plot shows that the estimated coefficients from the ridge regression model are, in general, smaller than those from the OLS model.

The next section focuses on the lasso regression.

Working with lasso regression

The lasso regression is another type of regularized linear regression. It is similar to ridge regression but differs in terms of the specific process of calculating the magnitude of the coefficients. Specifically, it uses the L1 norm of the coefficients, which consists of the total sum of absolute values of the coefficients, as the penalty added to the OLS loss function.

The lasso regression cost function can be written as:

$$L_{lasso} = RSS + \lambda \sum_{j=1}^{p} |\beta_j|$$

The key characteristic of lasso regression is that it can reduce some coefficients exactly to 0, effectively performing the variable selection. This is a consequence of the L1 penalty term and is not the case for ridge regression, which can only shrink coefficients close to 0. Therefore, lasso regression is particularly useful when we believe that only a subset of the predictors actually matters when it comes to predicting the outcome.

In addition, in contrast to ridge regression, which can't perform variable selection and therefore may be less interpretable, lasso regression automatically selects the most important features and discards the rest, which can make the final model easier to interpret.

Like ridge regression, the lasso regression penalty term is also subject to a tuning parameter λ. The optimal λ is typically chosen via cross-validation or a more intelligent search policy such as Bayesian optimization.

Let us go through an exercise to understand how to develop a lasso regression model.

Exercise 12.6 Implementing lasso regression

To implement the lasso regression model, we can follow a similar process as the ridge regression model.

1. To fit a lasso regression model, set alpha=1 in the `glmnet()` function.

```
lasso_model = glmnet(X, y, alpha = 1)
```

2. Use the same cross-validation procedure to identify the optimal value of lambda.

```
# Use cross-validation to find the optimal lambda
cv_lasso = cv.glmnet(X, y, alpha = 1)
best_lambda = cv_lasso$lambda.min
>>> best_lambda
0.8007036
```

3. Fit a new lasso regression model using the optimal lambda.

```
# Fit a new lasso regression model using the optimal lambda
```

```
opt_lasso_model = glmnet(X, y, alpha = 1, lambda =
best_lambda)
```

The resulting coefficients can also be extracted using the `coef()` function, followed by `[-1]` to remove the intercept term.

```
# Get coefficients

lasso_coefs = coef(opt_lasso_model)[-1]  # remove intercept

>>> lasso_coefs

[1] -0.88547684  0.00000000 -0.01169485  0.00000000 -
2.70853300  0.00000000  0.00000000

 [8]  0.00000000  0.00000000  0.00000000
```

4. Plot the estimated coefficients together with the previous two models.

```
plot(1:length(ols_coefs), ols_coefs, type="b", col="blue",
pch=19, xlab="Coefficient", ylab="Value",
ylim=c(min(ols_coefs, ridge_coefs), max(ols_coefs,
ridge_coefs)))

lines(1:length(ridge_coefs), ridge_coefs, type="b", col="red",
pch=19)

lines(1:length(lasso_coefs), lasso_coefs, type="b",
col="green", pch=19)

legend("bottomright", legend=c("OLS", "Ridge", "Lasso"),
col=c("blue", "red", "green"), pch=19)
```

Running the codes generates *Figure 12.8*, suggesting that only two variables are kept in the resultant model. The lasso regression model is thus able to produce a sparse solution by setting the coefficients of some features equal to zero.

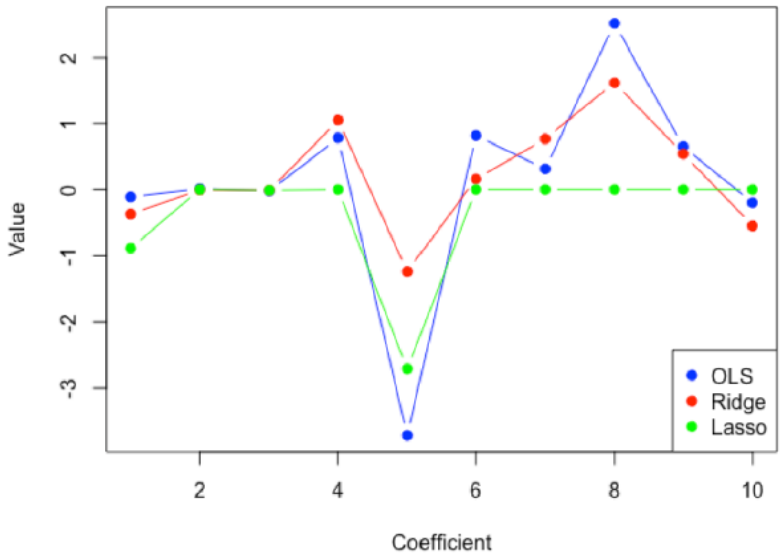

Figure 12.8 – Visualizing the estimated coefficients from the ridge, lasso, and OLS regression models.

In summary, the lasso regression model gives us a sparse model by setting the coefficients of non-significant variables to zero, thus achieving feature selection and model estimation at the same time.

Summary

In this chapter, we covered the nuts and bolts of the linear regression model. We started by introducing the SLR model that consists of only one input variable and one target variable, and then extended to the MLR model with two or more predictors. Both models can be assessed using the R^2, or more preferably, the adjusted R^2 metric. Next, we discussed specific scenarios, such as working with categorical variables and interaction terms, handling nonlinear terms via transformations, working with the closed-form solution, and dealing with multicollinearity and heteroskedasticity. Lastly, we introduced widely used regularization techniques such as ridge and lasso penalties, which can be incorporated into the loss function as a penalty term and generate a regularized model, and, additionally, a sparse solution in the case of lasso regression.

In the next chapter, we will cover another type of widely used linear model: the logistic regression model.