13
Logistic Regression with R

In this chapter, we will introduce logistic regression, covering its theoretical construct, connection with linear regression, and practical implementation. As it is an important classification model that is widely used in areas where interpretability matters, such as credit risk modeling, we will focus on its modeling process in different contexts, along with extensions such as adding regularization to the loss function and predicting more than two classes.

By the end of this chapter, you will learn the fundamentals of logistic regression model and its comparison with linear regression, including extended concepts such as the sigmoid function, odds ratio, and cross-entropy loss. You will also grasp the commonly used evaluation metrics in the classification setting, as well as enhancements that deal with imbalanced datasets and multiple classes in the target variable.

In this chapter, we will cover:

- Introducing logistic regression
- Comparing logistic regression with linear regression
- More on Log odds and odds ratio
- Introducing the Cross entropy loss
- Evaluating a logistic regression model
- Dealing with imbalanced dataset
- Penalized logistic regression
- Extending to multi-class classification

Technical Requirements

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

- `caret` - 6.0.94.
- `tibble` - 3.2.1.
- `dplyr` - 1.0.10.
- `pROC` - 1.18.2.
- `nnet` - 7.3.18.

- `glmnet` - 4.1.7.

  The versions mentioned along with the packages in the preceding list are the latest ones while I am writing this book.

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_13/working.R.

Introducing logistic regression

Logistic regression is a binary classification model. It is still a linear model, but now the output is constrained to be a binary variable, taking the value of 0 or 1, instead of modeling a continuous outcome as in the case of linear regression. In other words, we will observe and model the outcome $y = 1$ or $y = 0$. For example, in the case of credit risk modeling, $y = 0$ refers to a non-default loan application, while $y = 1$ indicates a default loan.

However, instead of directly predicting the binary outcome, the logistic regression model predicts the probability of y taking a specific value, such as $P(y = 1)$. The probability of assuming the other category is $P(y = 0) = 1 - P(y = 1)$, since the total probability should always sum to one. The final prediction would be the winner of the two, taking the value of 1 if $P(y = 1) > P(y = 0)$, and 0 otherwise. In the credit risk example, $P(y = 1)$ would be interpreted as the probability of a loan to default.

In logistic regression, the term "logistic" is related to "logit", which refers to the log odds. The odds is another way to describe the probability; instead of specifying the individual $P(y = 1)$ and $P(y = 0)$, it refers to the ratio of $P(y = 1)$ to $P(y = 0)$. Thus the log odds is calculated via $\log \frac{P(y=1)}{P(y=0)}$.
Therefore, we can simply use the term odds to describe the probability of an event happening ($y = 1$) over the probability of it not occurring ($y = 0$).

First, let us look at how the logistic regression model transforms a continuous output (as in linear regression) into a probability score, which is a number bounded between 0 and 1.

Understanding the sigmoid function

The sigmoid function is the key ingredient that maps any continuous number (from negative infinity to positive infinity) to a probability. Also known as the logistic function, the sigmoid function is characterized by an S-shaped curve, taking any real number as input and mapping it into a score between 0 and 1, which happen to be the range of a valid probability score.

The standard sigmoid function takes the following form:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Note that this is a nonlinear function. That is, the input values will get disproportionate scaling when going through the transformation using this function. It is also a continuous function (thus differentiable) and monotone ($f(x)$ will increase as x increases), thus enjoying high popularity as the go-to activation function to be used at the last layer of a typical neural network model for binary classification tasks.

Let us try to visualize this function. In the following code snippet, we use `seq(-10, 10, by = 0.1)` to create a sequence of equally spaced numbers from -10 to 10, with a step size of 0.1. For

each number, we calculate the corresponding output using the sigmoid function. Here, we directly pass all the numbers of the function, which then calculates all the outputs in a parallel mode called vectorization. Here, vectorization refers to the process of applying an operation to an entire vector simultaneously instead of looping over each element one by one as in a for loop. Finally, we plot the function to show the characteristic S-shaped curve of the sigmoid function and add the gridlines using the `grid()` function.

```
# Create a range of equally spaced numbers between -10 and 10
x = seq(-10, 10, by = 0.1)

# Calculate the output value for each number using sigmoid function
sigmoid = 1 / (1 + exp(-x))

# Plot the sigmoid function
plot(x, sigmoid, type = "l", lwd = 2,
     main = "Sigmoid Function",
     xlab = "x",
     ylab = "f(x)",
     col = "red")

# Add grid lines
grid()
```

Running the preceding codes generates the output shown in *Figure 13.1*. The plot shows a different level of steepness across the whole domain, where the function is more sensitive in the middle region and becomes more saturated at the two extremes.

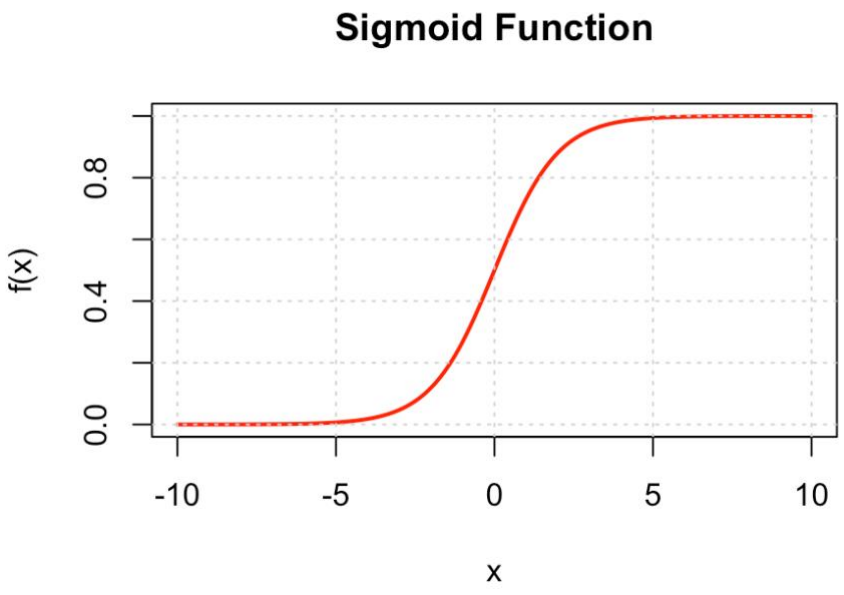


Figure 13.1 – Visualizing the sigmoid function.

With the sigmoid function in perspective, now let us look at the mathematical construct of the logistic regression model.

Grokking the logistic regression model

The logistic regression model is essentially a linear regression model generalized to the setting where the dependent outcome variable is binary. In other words, it is a linear regression model that models the log odds of the probability of an event.

To see this, let us first recall the following linear regression model, where we use a total of p features to model the target output variable z.

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

Here, z is interpreted as the log odds, or logit, of the event of $y = 1$. We are interested in estimating the parameters from β_0 to β_p.

Now, we know that the variable z is unbounded, meaning it can vary from negative infinity to positive infinity. We need a way to bound this output and convert it to a probability score valued between 0 and 1. This is achieved via an additional transformation using the sigmoid function, which happens to satisfy all our needs. Mathematically, we have:

$$P(y = 1) = \frac{1}{1 + e^{-z}}$$

Plugging in the definition of z gives the full logistic regression model:

$$P(y = 1) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p)}}$$

where $P(y = 1)$ refers to the probability of having a success of $y = 1$ (this is a general statement), and correspondingly, $P(y = 0)$ indicates the probability of having a failure.

Note that we can equivalently express the model as the following:

$$\log \frac{P(y = 1)}{P(y = 0)} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p$$

Where the term $\log \frac{P(y=1)}{P(y=0)}$ stands for the log odds.

A key change here is the introduction of the sigmoid function. This makes the relationship between the predictors and the resulting probability no longer linear, but sigmoidal instead. To observe the subtlety here, we can look at the different regions of the sigmoid function across the domain. For example, when looking at the region around 0, a small change in the input would result in a relatively large change in the resulting probability output. However, the same change in the input will cause a very small change in the output when located on the two extreme sides of the function. Also, as the input becomes more extreme, either towards the negative or positive side, the resulting probability will gradually approach 0 or 1. *Figure 13.2* recaps the characteristics of the logistic regression model.

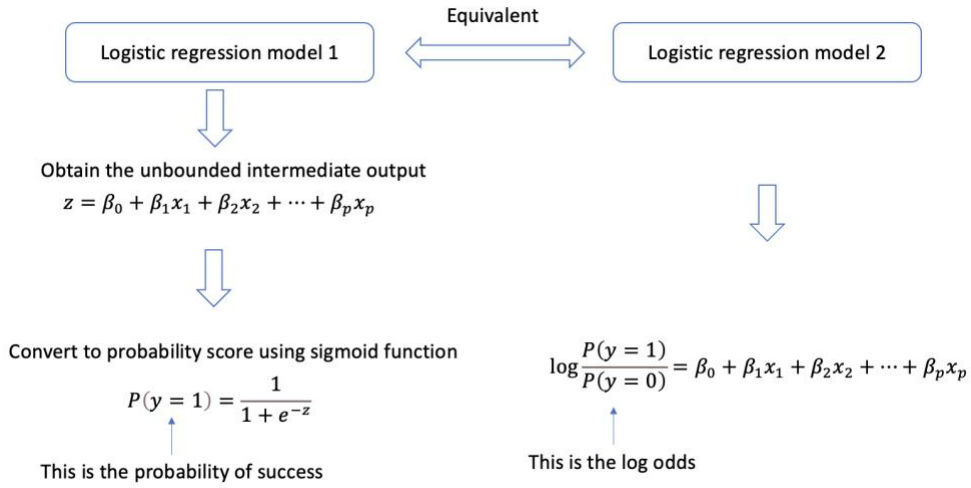


Figure 13.2 – Summarizing the logistic regression model.

Note that a logistic regression model shares similar assumptions with the linear regression model. Specifically, it assumes that the observations are independent of each other, and the target outcome follows a Bernoulli distribution parameterized by p; that is, $y \sim Bernourlli(p)$. Given this, we do not assume a linear regression between the output variable and the input predicts; instead, we use the logistic link function to transform and introduce nonlinearity to the input variables.

The following section further compares logistic regression with linear regression.

Comparing logistic regression with linear regression

In this section, we will focus on a binary credit classification task using the German Credit dataset, which contains 1,000 observations and 20 columns. Each observation denotes a customer who had a loan application from the bank and is labelled as either good or bad in terms of the credit risk. The dataset is available in the `caret` package in R.

For our study, we will attempt to predict the target binary variable `Class` based on `Duration`, and compare the difference in the prediction outcome between linear regression and logistic regression. We specifically choose one predictor only so that we can visualize and compare the decision boundaries of the resultant model in a two-dimensional plot.

Exercise 13.1 Comparing linear regression with logistic regression

In this exercise, we will demonstrate the advantage of using a logistic regression model in producing a probabilistic output compared to the unbounded output using a linear regression model.

1. Load the German Credit dataset from the `caret` package. Convert the target variable (`Class`) to numeric.

```
# install the caret package if you haven't done so
install.packages("caret")
# load the caret package
library(caret)
```

```
# load the German Credit dataset

data(GermanCredit)

GermanCredit$Class_num = ifelse(GermanCredit$Class == "Bad",
1, 0)
```

Here, we create a new target variable called `Class_num` to map the original `Class` variable to `1` if it takes on the value of `"Bad"` and `0` otherwise. This is necessary as both linear regression and logistic regression models cannot accept a string-based variable as the target (or predictor).

2. Build a linear regression model to regress `Class_num` against `Duration`.

```
lm_model = lm(Class_num ~ Duration, data=GermanCredit)

coefs = coefficients(lm_model)

intercept = coefs[1]

slope = coefs[2]
```

Here, we use the `lm()` function to build the linear regression model and `coefficients()` to extract the model coefficients, including the intercept and slope.

3. Visualize the prediction and the target.

```
ggplot(GermanCredit,

       aes(Duration, Class_num)) +

  geom_point() +

  geom_abline(intercept=intercept, slope=slope) +

  theme(axis.title.x = element_text(size = 18),

        axis.title.y = element_text(size = 18))
```

Here, we plot the observed target variable as a scatter plot and use the `geom_abline()` function to plot the model as a straight line based on the estimated slope and intercept.

Running the preceding codes generates *Figure 13.3*.

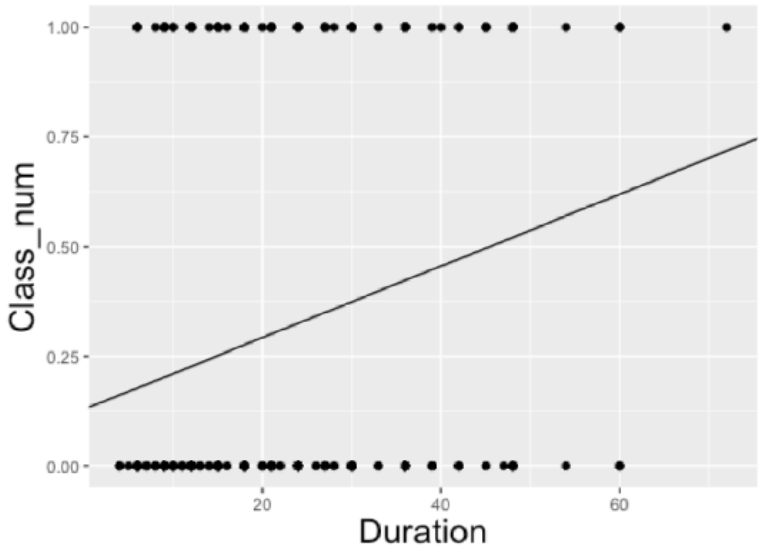


Figure 13.3 – Visualizing the linear regression model.

Since all target values are `0` or `1`, we can think of the predictions as probabilities valued between `0` and `1`. However, as we zoom out, the problem of having an unbounded probability would surface, as shown in the following steps.

4. Re-plot the graph by zooming out to a wide domain of `(-30, 120)` for the x-axis and `(-0.5, 1.5)` for the y-axis.

```
ggplot(GermanCredit,
       aes(Duration, Class_num)) +
  geom_point() +
  geom_abline(intercept=intercept, slope=slope) +
  xlim(-30, 120) +
  ylim(-0.5, 1.5) +
  theme(axis.title.x = element_text(size = 18),
        axis.title.y = element_text(size = 18))
```

Here, we enlarged the range of possible values for the x-axis and y-axis using the `xlim()` and `ylim()` functions.

Running the preceding codes generates the output shown in *Figure 13.4*, which shows that the predicted values are outside the range of `[0,1]` when the value of Duration becomes more extreme, a situation called extrapolation beyond the observed range of values. This means that the predicted probabilities would be smaller than zero or bigger than one, which is obviously an invalid output. This calls for a generalized linear regression model called logistic regression, where the response will follow a logistic, S-shaped curve based on the transformation of the sigmoid function.

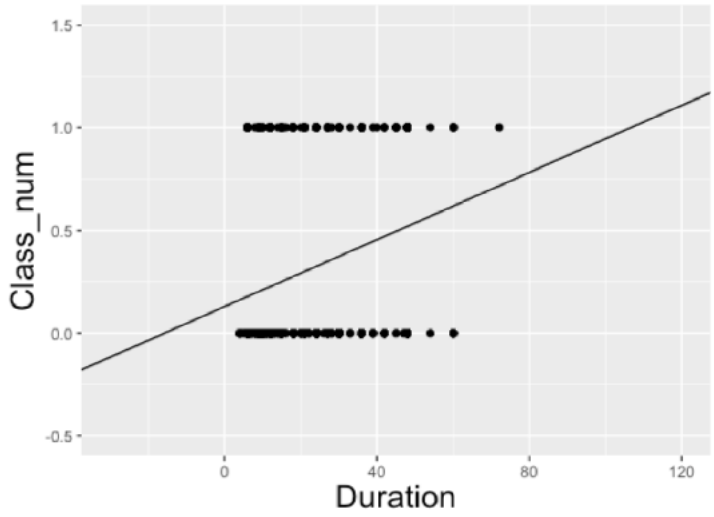


Figure 13.4 – Visualizing the linear regression model with extended range.

5. Build a logistic regression model using the `glm()` function.

```
glm_model = glm(Class_num ~ Duration, data=GermanCredit,
family=binomial)

>>> glm_model

Call:  glm(formula = Class_num ~ Duration, family = binomial,
data = GermanCredit)


Coefficients:

(Intercept)      Duration

   -1.66635       0.03754


Degrees of Freedom: 999 Total (i.e. Null);  998 Residual

Null Deviance:       1222

Residual Deviance: 1177     AIC: 1181
```

The result shows the estimated intercept and slope, along with the residual deviance, a measure of goodness of fit for the logistic regression model.

6. Plot the estimated logistic curve on the previous plot.

```
ggplot(GermanCredit,

       aes(Duration, Class_num)) +

  geom_point() +

  geom_abline(intercept=intercept, slope=slope) +

  geom_smooth(
```

```
    method = "glm",
    se = FALSE,
    method.args = list(family=binomial)
  ) +
  theme(axis.title.x = element_text(size = 18),
        axis.title.y = element_text(size = 18))
```

Running the preceding codes will generate the output as shown in *Figure 13.5*, which suggests a slight deviation between the linear regression line and the logistic regression curve.

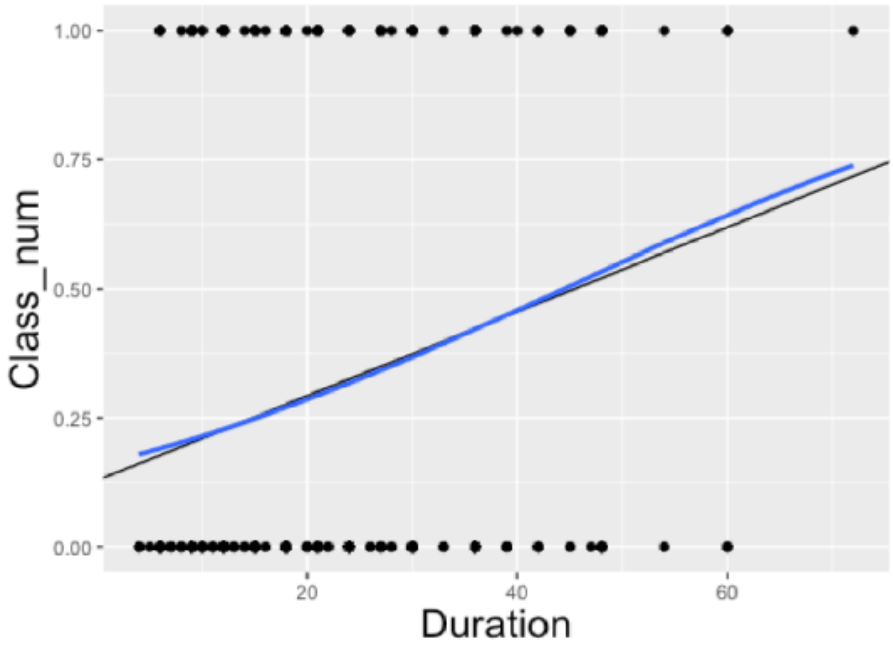


Figure 13.5 – Visualizing the logistic regression curve.

Again, we can zoom out the figure and focus on the difference when going beyond the observed range of possible values in the dataset.

7. Plot the logistic curve within a wider range of values beyond the observed range.

```
# Get coefficients from logistic model
intercept_glm = coef(glm_model)[1]
slope_glm = coef(glm_model)[2]

# Generate sequence of x-values
x_values = seq(from = min(GermanCredit$Duration) - 150,
               to = max(GermanCredit$Duration) + 150,
               by = 0.1)
```

```
# Compute probabilities using logistic function
y_values = 1 / (1 + exp(-(intercept_glm + slope_glm *
x_values)))


# Data frame for plot
plot_df = data.frame(x = x_values, y = y_values)


# Plot
ggplot() +
  geom_point(data = GermanCredit, aes(Duration, Class_num)) +
  geom_abline(intercept=intercept, slope=slope) +
  geom_line(data = plot_df, aes(x, y), color = "blue") +
  theme_minimal() +
  xlim(-30, 120) +
  ylim(-0.5, 1.5) +
  theme(axis.title.x = element_text(size = 18),
        axis.title.y = element_text(size = 18))
```

Here, we first extract the coefficients for the logistic regression model, followed by generating a sequence of values for the input and the corresponding output using the sigmoid function transformation. Lastly, we plot the logistic curve together with the linear fit in the same plot with the observed data.

Running the preceding codes generates the output shown in *Figure 13.6*, which shows that the logistic regression curve gets gradually saturated as the input value becomes more extreme. In addition, all values are now bounded in the range of `[0,1]`, making it a valid candidate for interpretation as probabilities instead of an unbounded value.

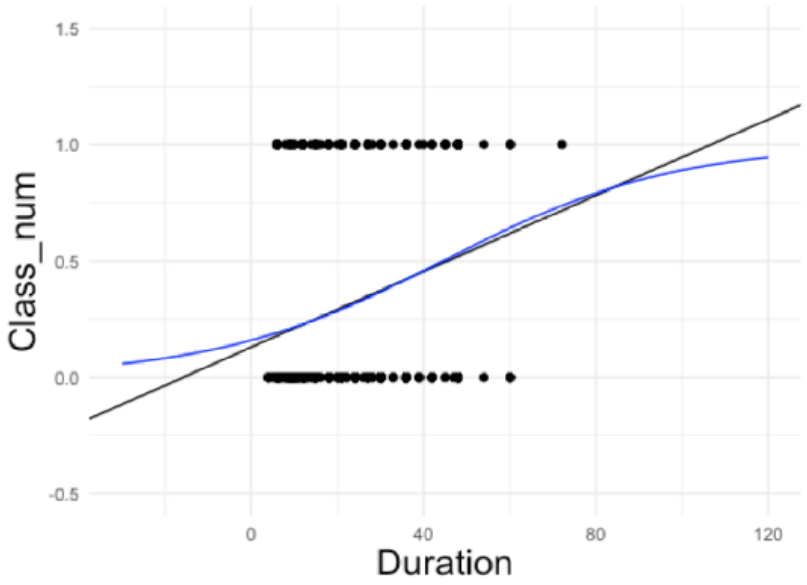


Figure 13.6 – Visualizing the logistic regression model with extended range.

The next section looks at how to make predictions using a logistic regression model.

Making predictions using the logistic regression model

As discussed in the previous section, the direct predictions from a logistic regression model take the form of probabilities valued between 0 and 1. To convert them into binary predictions, we could take the most probable prediction by rounding the probability using a threshold of 0.5. For example, if the predicted probability is $P(y = 1) = 0.8$, the rounding operation will lead to a final binary prediction of $y = 1$. On the other hand, if $P(y = 1) = 0.3$, rounding will result in $y = 0$.

Let us go through the following exercise to understand how to perform predictions using the logistic regression model.

Exercise 13.2 Performing predictions using the logistic regression model

We have seen how to perform predictions using the explicit sigmoid transformation after extracting the slope and intercept of the logistic regression model. In this exercise, we will explore a more convenient approach using the `predict()` function.

1. Generate a sequence of `Duration` values ranging from `5` to `80` with a step size of `2` and predict the corresponding probabilities for the sequence using the `predict()` function based on the previous logistic regression model.

```
library(tibble)
library(dplyr)


# making predictions
pred_df = tibble(
  Duration = seq(5, 80, 2)
```

```
)


pred_df = pred_df %>%
  mutate(
    pred_prob = predict(glm_model, pred_df, type="response")
  )
```

Here, we use the `seq()` function to create the equally spaced vector and store it in a `tibble` object called `pred_df`. We then use the `predict()` to predict the corresponding probabilities by specifying `type="response"`.

2. Visualize the predicted probabilities together with the raw data.

```
ggplot() +
  geom_point(data = GermanCredit, aes(Duration, Class_num)) +
  geom_point(data = pred_df, aes(Duration, pred_prob) ,
color="blue") +
  theme(axis.title.x = element_text(size = 18),
        axis.title.y = element_text(size = 18))
```

The preceding code will generate the following output:

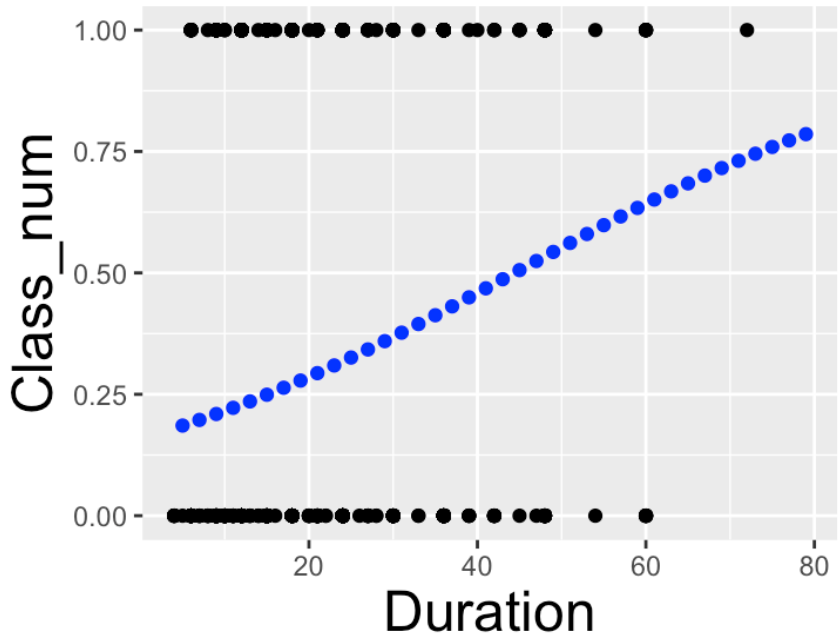


Figure 13.7 – Visualizing the predicted probabilities.

3. Convert the probabilities to binary outcomes using the `round()` function.

```
# getting the most likely outcome
pred_df = pred_df %>%
```

```
  mutate(
    most_likely_outcome = round(pred_prob)
  )
```

Here, we round the predicted probabilities using a default threshold of `0.5`.

4. Add the binary outcomes as green points to the previous graph.

```
ggplot() +
  geom_point(data = GermanCredit, aes(Duration, Class_num)) +
  geom_point(data = pred_df, aes(Duration, pred_prob) ,
color="blue") +
  geom_point(data = pred_df, aes(Duration,
most_likely_outcome) , color="green") +
  theme(axis.title.x = element_text(size = 18),
        axis.title.y = element_text(size = 18))
```

Running the preceding codes will generates *the following output*, which suggests that all predicted probabilities above 0.5 are converted to 1, and those below 0.5 are converted to 0.

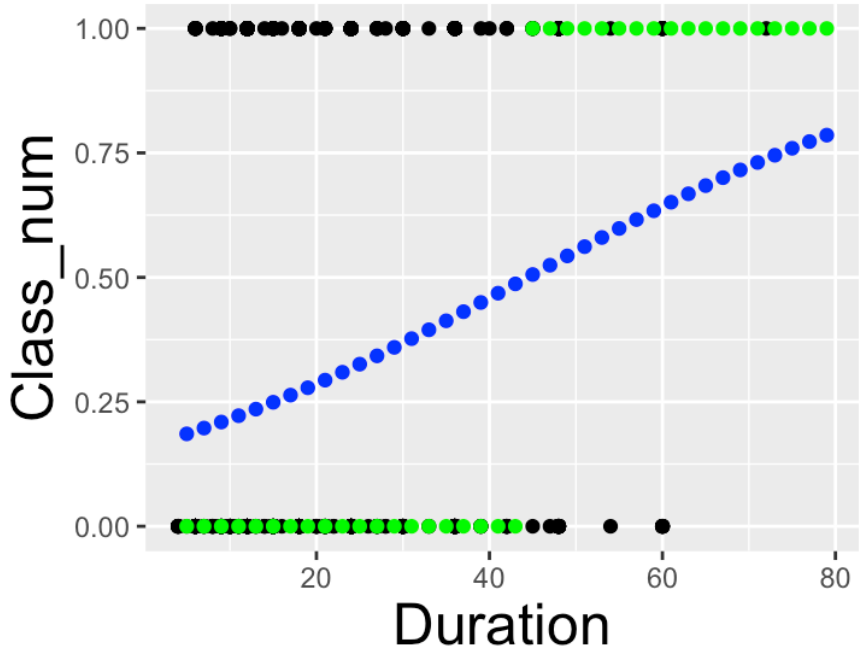


Figure 13.8 – Visualizing the predicted binary outcomes.

The next section discusses more on the log odds.

More on log odds and odds ratio

Recall that the odds refers to the ratio of the probability of an event happening over its complement:

$$odds = \frac{probability\ of\ event\ happening}{probability\ of\ event\ not\ happening} = \frac{p}{1-p} = \frac{P(y=1)}{P(y=0)}$$

Here, the probability is calculated as:

$$p = P(y=1) = \frac{1}{1+e^{-z}}$$

$$1 - p = 1 - \frac{1}{1+e^{-z}} = \frac{e^{-z}}{1+e^{-z}}$$

Plugging in the definition of p and $1-p$ gives:

$$odds = \frac{p}{1-p} = e^{z}$$

Instead of directly working with the odds, we would often use the log odds, or logit. This term is typically modeled as a linear combination of predictors in a logistic regression model via the following:

$$\log \frac{P(y=1)}{P(y=0)} = z = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p$$

Here, we can interpret each coefficient β_j as the expected change in the log odds for a one-unit increase in the jth predictor x_j, while keeping all other predictors constant. This equation essentially says that the log odds of the target value y being 1 is linearly related to the input variables.

Now suppose x_i is a binary input variable, making x_i = 1 or 0. We can calculate the odds of $x_i = 1$ as:

$$\frac{p_1}{1-p_1} = e^{z_1}$$

Which measures the chance of an event for $x_i = 1$, over the chance of a non-event. Similarly, we can calculate the odds of $x_i = 0$ as:

$$\frac{p_0}{1-p_0} = e^{z_0}$$

Which measures the chance of an event for $x_i = 0$ over the chance of a non-event.

We can then calculate the odds ratio of x_i, which is the ratio of the odds $x_i = 1$ to the odds of $x_i = 0$:

$$\frac{\frac{p_1}{1-p_1}}{\frac{p_0}{1-p_0}} = \frac{e^{\beta_0+\beta_1 x_1+\cdots\beta_i*1+\cdots+\beta_p x_p}}{e^{\beta_0+\beta_1 x_1+\cdots\beta_i*0+\cdots+\beta_p x_p}} = e^{\beta_i}$$

Here, e^{β_i} measures the quantified impact of the binary input variable x_i on the odds of the outcome y being 1, while all other input variables remain unchanged. This gives us a way to measure the impact of any predictor in a logistic regression model, covering both categorical and numerical input variables.

For categorical input variables, we may use gender (0 for male and 1 for female) to predict if insurance is purchased (1 for yes and 0 for no). We set the base categorical as 0 for male. If the estimated coefficient $\beta_{gender} = 0.2$ for gender, its odds ratio is calculated as $e^{0.2} \approx 1.22$. Therefore, the odds of female customers purchasing the insurance is 1.22 times the odds of their male

counterparts purchasing the insurance, assuming all other variables remain unchanged (ceteris paribus).

For numerical input variables, we may use age to predict if insurance is purchased. There is no need to set the base category in this case. If the estimated coefficient $\beta_{age} = 0.3$, the corresponding odds ratio is calculated as $e^{0.3} \approx 1.35$. This means that the odds of a client purchasing is 1.35 times the odds of similar people who are 1 year younger, assuming ceteris paribus.

Note that we can calculate the log odds using the predicted probabilities, as shown in the following code snippet.

```
# calculate log odds using predicted probabilities
pred_df = pred_df %>%
  mutate(
    log_odds = log(pred_prob / (1 - pred_prob))
  )
```

The next section introduces more on the loss function of the logistic regression model.

Introducing the cross entropy loss

The binary cross-entropy loss, also called the **log loss**, is often used as the cost function in logistic regression. This is the loss that the logistic regression model will attempt to minimize, by moving the parameters. This function takes the predicted probabilities and the corresponding targets as the input and outputs a scalar score, indicating the goodness of fit. For a single observation with target y_i and predicted probability p_i, the loss is calculated as:

$$Q_i(y_i, p_i) = -[y_i \log p_i + (1 - y_i) \log(1 - p_i)]$$

Summing up all individual losses gives the total binary cross-entropy loss:

$$Q(y, p) = \frac{1}{N} \sum_{i}^{N} Q_i = \frac{1}{N} \sum_{i=1}^{N} -[y_i \log p_i + (1 - y_i) \log(1 - p_i)]$$

The binary **cross-entropy loss** (**CEL**) function is a suitable choice for binary classification problems because it heavily penalizes confident but incorrect predictions. For example, as p_i approaches 0 or 1, the resulting cross-entropy loss will go to infinity if the prediction is incorrect. This property thus encourages the learning process to output probabilities that are close to the true probabilities of the targets.

More generally, we use the CEL to model a classification problem with two or more classes in the target variable. For the i^{th} observation $\mathbf{x}_i$, the classification function would produce a probability output, denoted as $p_{i,k} = f(\mathbf{x}_i; \mathbf{w})$ to indicate the likelihood of belonging to the k^{th} class. When we have a classification task with a total of C classes, the CEL for $\mathbf{x}_i$ is defined as $Q_i(\mathbf{w}) = -\sum_{k=1}^{C} y_{i,k} \log(p_{i,k})$, which essentially sums across all classes. Again, $y_{i,k} = 1$ if the target label for the i^{th} observation belongs to the k^{th} class, and $y_{i,k} = 0$ otherwise.

The summation means that the class-wise evaluation (i.e., the term $y_{i,k} \log(p_{i,k})$) is performed for all classes and summed together to produce the total cost for the i^{th} observation. For each observation, there are a total of C predictions corresponding to the respective probability of

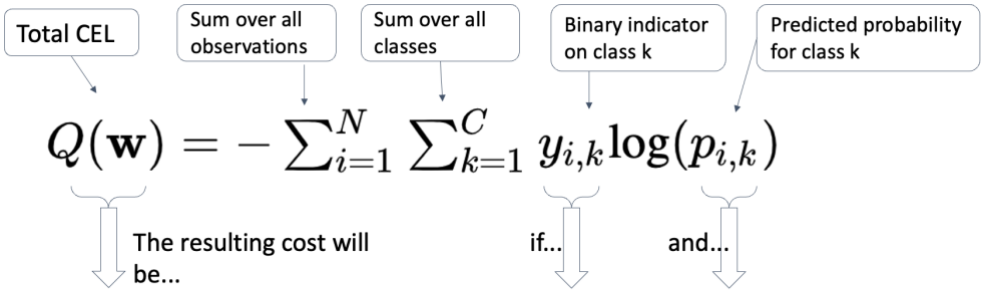
belonging to each class. The CEL thus aggregates the matrix of predicted probabilities by summing them into a single number. In addition, the result is negated to produce a positive number, since $\log(x)$ is negative if x is a probability between 0 and 1.

Note that the target label in the CEL calculation needs to be one-hot encoded. This means that a single categorical label is converted to an array of binary numbers that contains one for the class label and zero otherwise. For example, for a digit image on number eight, instead of passing the original class as the target output, the resulting one-hot encoded target [0, 0, 0, 0, 0, 0, 0, 1, 0, 0] would be used, where the eighth position is activated (i.e., hot) and the rest disabled. The target array would also have the same length as the probability output array, whose elements correspond to each of the classes.

Intuitively, we would hope the predicted probability for the correct class to be close to 1 and for the incorrect class to be 0. That is, the loss should increase as the predicted probabilities diverge from the actual class label. The cross-entropy loss is designed to follow this intuition. Specifically, we can look at the following four scenarios for the i^{th} observation:

- When the target label belongs to the k^{th} class (i.e., $y_{i,k} = 1$) and the predicted probability for the k^{th} class is very strong (i.e., $p_{i,k} \approx 1$), the cost should be low;
- When the target label belongs to the k^{th} class (i.e., $y_{i,k} = 1$) and the predicted probability for the k^{th} class is very weak (i.e., $p_{i,k} \approx 0$), the cost should be high;
- When the target label does not belong to the k^{th} class (i.e., $y_{i,k} = 0$) and the predicted probability for the k^{th} class is very strong (i.e., $p_{i,k} \approx 1$), the cost should be high;
- When the target label does not belong to the k^{th} class (i.e., $y_{i,k} = 0$) and the predicted probability for the k^{th} class is very weak (i.e., $p_{i,k} \approx 0$), the cost should be low.

To calculate the CEL, we first calculate the weighted sum between the vector of binary labels and the vector of predicted probabilities across all classes for each observation. The results of all observations are added together, followed by a minus sign to reverse the cost to a positive number. The CEL is designed to match the intuition for the cost: the cost would be low when the prediction and the target closely match and high otherwise. In other words, to calculate the total cost, we would simply sum individual costs, leading to $Q(\mathbf{w}) = -\sum_{i=1}^{N}\sum_{k=1}^{C} y_{i,k}\log(p_{i,k})$. *Figure 13.9* summarizes the above discussion on the CEL.



Low	1	≈1
High	1	≈0
High	0	≈1
Low	0	≈0

Figure 13.9 – Illustrating the cross entropy loss.

Note that the last two scenarios do not contribute at all to the loss calculation since the target value is equal to zero; any value multiplied by zero becomes zero. Also, observe that the predicted probabilities of a specific observation for these two classes need to add up to one, making it sufficient to only focus on the predicted probability of one class (mostly class 1).

The next section introduces how to evaluate a logistic regression model.

Evaluating a logistic regression model

There are multiple metrics we can use to evaluate a logistic regression model. These are the metrics we use to determine the goodness of fit (over the test set), which needs to be differentiated from the cross-entropy loss we use to train the model (over the training set).

The following list provides the commonly used metrics.

- **Accuracy rate**: This is the proportion of the number of correctly predicted observations made by the model out of the count of all observations. Since a correct prediction can be either a true positive or a true negative, the accuracy is calculated by summing up the true positives and true negatives and dividing the total number of observations.
- **Error rate**: This is the proportion of incorrectly predicted observations made by the model over the total observations. An incorrect prediction can be a false positive or a false negative. It is calculated as 1 - Accuracy Rate; that is, the error rate is the complement of the accuracy rate. In other words, it calculated as (false positives + false negatives) / total observations.
- **Precision**: Precision is the proportion of correct positive predictions among all positive predictions. This measure essentially tells us that out of all predicted positive instances, how many of them are correct. Thus it indicates the model's accuracy in predicting positive observations, and is calculated as true Positives / (true positives + false positives). In the denominator, we note that among all positive instances, some are true positives and the rest are false positives.
- **Recall**: Recall refers to the proportion of actual positive instances that the model correctly predicts. Also called sensitivity or **True Positive Rate** (**TPR**), the recall measures the model's ability to detect positive observations. It is calculated as true positives / (true positives + false negatives), where the formulae differs in the denominator compared with precision.
- **Specificity**: Also called **True Negative Rate** (**TNR**), specificity measures the proportion of actual negative instances correctly predicted by the model. This is the opposite of sensitivity, which focuses on the model's ability to capture the true positives. In other words, specificity measures the model's ability to identify negative instances or non-events correctly. It is calculated as true negatives / (true negatives + false positives).

- **Area Under the Curve** (**AUC**): The AUC is directly determined by the **Receiver Operating Characteristic** (**ROC**) curve, which plots the true positive rate (sensitivity) against the false positive rate (1 - specificity) at different thresholds for binary classification. As the name suggests, AUC measures the area under the ROC curve in the form of a proportion valued between 0 and 1, indicating the degree of separability between two classes. A perfect model with 100% correct predictions an AUC of 1, while a model with completely wrong predictions has an AUC of 0. A model that performs random guesses (choosing 0 or 1 with 50% probability) corresponds to an AUC of 0.5, suggesting no class separation capacity.

Note that we would exercise the principle of parsimony upon assessing two models with equally good evaluation metrics. The principle of parsimony says that if two competing models provide a similar level of fit to the data, the one with fewer input variables should be picked, thus preferring simplicity over complexity. The underlying assumption is that the most accurate model is not necessarily the best model.

Figure 13.10 describes the process of laying out the confusion matrix that captures the prediction results in different scenarios, along with the details on calculating the aforementioned evaluation metrics.

Confusion matrix

	Predicted $\hat{y} = 0$	Predicted $\hat{y} = 1$	Total
Non-event $y = 0$	a	c	$a + c$
Event $y = 0$	b	d	$b + d$
Total	$a + b$	$c + d$	n

Accuracy = $\frac{a+d}{n}$

Error rate = $\frac{b+c}{n}$

Recall = $\frac{d}{b+d}$

Precision = $\frac{d}{c+d}$

Specificity = $\frac{a}{a+c}$

Figure 13.10 – Illustrating the confusion matrix and common evaluation metrics for binary classification tasks.

Note that we will only be able to calculate these metrics after selecting a threshold to cut off predicted probabilities. Specifically, an observation with a predicted probability greater than the cutoff threshold will be classified as positive, and negative otherwise.

Precision and recall usually have an inverse relationship with respect to the adjustment of the classification threshold. Reviewing both precision and recall is useful for cases where there is a huge imbalance in the target variable's values. As precision and recall are two related but different metrics, which should we optimize for?

To answer this question, we need to assess the relative impact of making a false negative prediction. This is measured by the false negative rate, which is the opposite (or complement) of recall. As

Figure 13.14 suggests, failing to spot a spam email is less risky than missing a fraudulent transaction or a positive cancer patient. We should aim to optimize for precision (so that the model's predictions are more precise and targeted) for the first case and recall (so that we minimize the chance of missing a potentially positive case) for the second case.

	Positive class	Impact of having a False Negative (FN)	How bad is FN?	Optimize for?
Spam filter	▪ Spam	▪ Spam goes to the inbox	▪ Acceptable	▪ Precision
Fraudulent transaction	• Fraud	▪ Fraudulent transactions pass through	▪ Very bad	▪ Recall
Cancer Diagnose	▪ Cancer	▪ Patient walks away without knowing he/she has cancer	▪ Very bad	▪ Recall

Figure 13.14 – Three cases with different impacts of having a false negative prediction.

As for the AUC, or area under the ROC curve, there is no need to select a specific threshold as it is calculated by evaluating a sequence of thresholds from 0 to 1. The ROC curve, or receiver operating characteristic curve, plots the sensitivity on the y-axis and (1-specificity) on the x-axis. This also corresponds to plotting TPR vs. (1-TNR), or TPR vs. FPR. As the classification threshold goes up, FPR goes down, leading to a leftward movement of the curve.

A perfect binary classifier has AUC score of 1. This means that FPR, or 1 – specificity, is 0. That is, there is no false positive, and all negative cases are not predicted as positive. In addition, the sensitivity, or TPR, is 1, meaning all positive cases are predicted as positive correctly.

Figure 13.15 illustrates three different AUC curves. The Topmost curve (in green) corresponds to a better model due to the highest AUC. Both models, represented by the green and red curves, perform better than random guessing, as indicated by the straight off-diagonal line in blue.

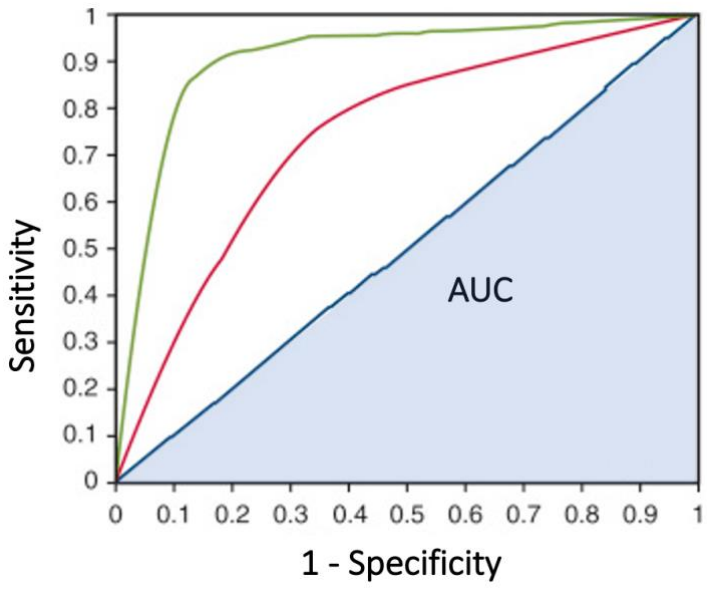


Figure 13.15 Illustrating three different AUC curves.

Continuing the previous exercise, we can now calculate the corresponding evaluation metrics. First, we use the trained logistic regression model to score all observations in the training set and obtain the corresponding probabilities using the `predict()` function and setting `type="response"`, as shown in the following code snippet. Note that we need to pass in a DataFrame with the corresponding feature names as input to the model.

```
# Create new data frame with all durations

new_data = data.frame(Duration = GermanCredit$Duration)

# Calculate predicted classes based on predicted probabilities

predicted_probs = predict(glm_model, new_data, type="response")
```

Next, we set a single cutoff threshold (0.5 in this case) to convert the predicted probabilities into the corresponding binary outcomes using the `ifelse()` function.

```
# Convert to binary outcomes

predicted_classes = ifelse(predicted_probs > 0.5, 1, 0)
```

With the binary outcomes and true target labels, we can obtain the confusion matrix as follows.

```
# Create confusion matrix

conf_matrix = table(predicted = predicted_classes, actual =
GermanCredit$Class_num)

>>> conf_matrix

         actual

predicted   0   1

        0 670 260

        1  30  40
```

Here, the confusion matrix provides a breakdown of the correct and incorrect classifications from the model. Within the confusion matrix, the top left cell means true negatives, the top right means false positives, the bottom left means false negatives, and the bottom right true positives.

Based on the confusion matrix, we can calculate the evaluation metrics as follows.

```
# Accuracy

accuracy = sum(diag(conf_matrix)) / sum(conf_matrix)

>>> print(paste("Accuracy: ", accuracy))

"Accuracy:  0.71"

# Error rate

error_rate = 1 - accuracy

>>> print(paste("Error rate: ", error_rate))

"Error rate:  0.29"
```

```
# Precision
precision = conf_matrix[2,2] / sum(conf_matrix[2,])
print(paste("Precision: ", precision))
"Precision:  0.571428571428571"
# Recall / Sensitivity
recall = conf_matrix[2,2] / sum(conf_matrix[,2])
print(paste("Recall: ", recall))
>>> "Recall:  0.133333333333333"
# Specificity
specificity = conf_matrix[1,1] / sum(conf_matrix[,1])
print(paste("Specificity: ", specificity))
>>> "Specificity:  0.957142857142857"
```

Here, we extract the corresponding items of the confusion matrix and plug in the definition of different evaluation metrics to complete the calculations.

We can also calculate the AUC, starting by calculating the ROC curve using the `pROC` package.

```
library(pROC)
# Calculate ROC curve
roc_obj = roc(GermanCredit$Class_num, predicted_probs)
# Plot ROC curve
>>> plot(roc_obj)
```

Running the codes generates *Figure 13.16*, which suggests that the model is doing marginally better than random guessing.

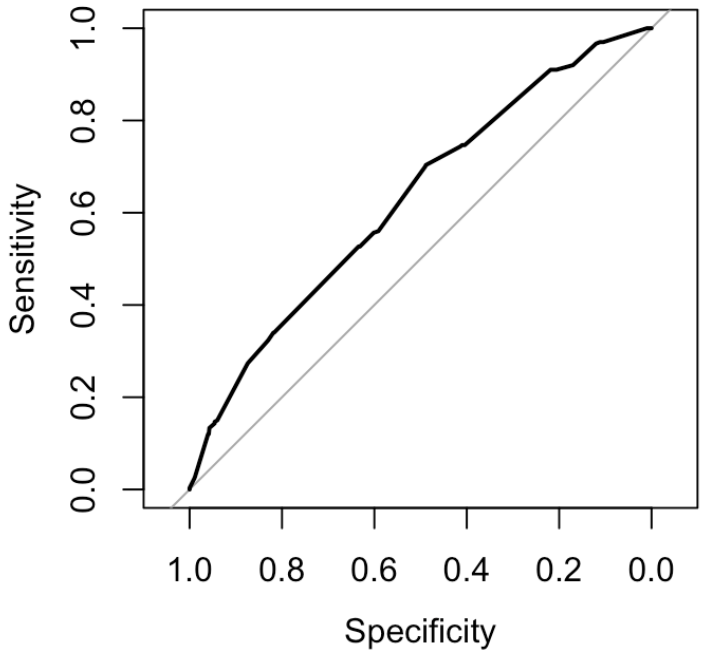


Figure 13.16 – Visualizing the ROC curve.

To calculate the AUC, we can call the `auc()` function.

```
# Calculate AUC

auc = auc(roc_obj)

>>> print(paste("AUC: ", auc))

"AUC:  0.628592857142857"
```

Note that when we have no preference for precision or recall, we can use the F1 score, defined as follows:

$$\text{F1 score} = \frac{2(\text{Precision} \times \text{Recall})}{\text{Precision} + \text{Recall}}$$

The next section discusses a challenging modeling situation when we have an imbalanced dataset to begin with.

Dealing with an imbalanced dataset

When building a logistic regression model using a dataset whose target is a binary outcome, it could be the case that the target values are not equally distributed. This means that we would observe more non-events ($y = 0$) than events ($y = 1$), as is often the case in applications such as fraudulent transactions in banks, spam/phishing emails for corporate employees, identification of rare diseases such as cancer, and natural disasters such as earthquakes. In these situations, the classification performance may be dominated by the majority class.

Such domination can result in misleadingly high accuracy scores, which correspond to poor predictive performance. To see this, suppose we are developing a default prediction model using a dataset that consists of 1,000 observations, where only 10 (or 1%) of them are default cases. A naive model would simply predict every observation as non-default, resulting in a 99% accuracy.

When we encounter an imbalanced dataset, we are often more interested in the minority class they it represents the outcome to be detected in a classification problem. Since the signal on the minority is relatively weak, we would need to rely on good modeling techniques to recognize a good pattern so that the signal can be correctly detected.

There are multiple techniques we can use to address the challenge due to an imbalanced dataset. We introduce a popular approach called data resampling, which requires oversampling and/or under-sampling the original dataset to make the overall distribution less imbalanced. Resampling includes oversampling the minority class, undersampling the majority class, or using a combination of them, as represented by **SMOTE**: **Synthetic Minority Over-sampling Technique**. However, such remedy does not come without a risk. Here, oversampling may lead to overfitting due to more samples added to the original dataset, while undersampling may result in loss of information due to some majority observations being removed from the original dataset.

Figure 13.17 illustrates the process of oversampling the minority class (the left panel) and undersampling the majority class (the right panel). Note that once the model is built based on the balanced dataset, we would still need to calibrate it on a new test set so that it performs well on a new real dataset as well.

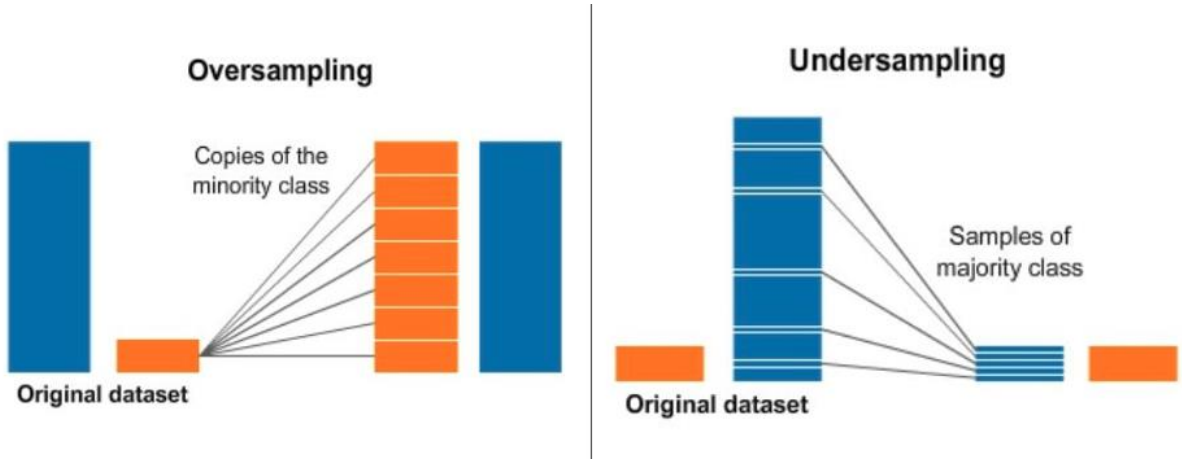


Figure 13.17 Illustrating the process of oversampling the minority and undersampling the majority.

Let us go through an exercise to understand how to perform undersampling or oversampling in a logistic regression setting.

Exercise 13.3 Performing undersampling and oversampling

In this exercise, we will create two artificial datasets based on undersampling and oversampling. We will then assess the performance of the resulting logistic regression model using the confusion matrix.

1. Divide the raw dataset into training (70%) and test (30%) sets.

```
set.seed(2)
index = sample(1:nrow(GermanCredit), nrow(GermanCredit)*0.7)
train = GermanCredit[index, ]
test = GermanCredit[-index, ]
```

Here, we randomly sample a set of indexes used to select observations for the training set, and allocate the rest to the test set.

2. Check the class ratio in the training set.

```
>>> table(train$Class_num)

  0   1

504 196
```

The result shows that the majority (class 0) is more than twice the size of the minority (class 1).

3. Separate the training set into a majority set and a minority set based on the class label.

```
# separate the minority and majority classes

table(train$Class_num)

minority_data = train[train$Class_num == 1,]

majority_data = train[train$Class_num == 0,]
```

We will then use these two datasets to perform undersampling and oversampling.

4. Undersample the majority class and combine the undersampled majority class with the minority class. Check the resulting class ratio.

```
# undersample the majority class

undersampled_majority =
majority_data[sample(1:nrow(majority_data),
nrow(minority_data)),]

# combine undersampled majority class and minority class

undersampled_data = rbind(minority_data,
undersampled_majority)

>>> table(undersampled_data$Class_num)

  0   1

196 196
```

The class ratio is balanced now. Let us perform oversampling for the minority class.

5. Oversample the minority class and combine the oversampled minority class with the majority class. Check the resulting class ratio.

```
# oversample the minority class

oversampled_minority =
minority_data[sample(1:nrow(minority_data),
nrow(majority_data), replace = TRUE),]

# combine majority class and oversampled minority class
```

```
oversampled_data = rbind(majority_data, oversampled_minority)

>>> table(oversampled_data$Class_num)

  0   1

504 504
```

6. Fit logistic regression models on both the undersampled and oversampled datasets.

```
# fit logistic regression models on undersampled and
oversampled data

undersampled_model = glm(Class_num ~ Duration, family =
binomial(link = 'logit'), data = undersampled_data)

oversampled_model = glm(Class_num ~ Duration, family =
binomial(link = 'logit'), data = oversampled_data)
```

7. Obtain the predicted probabilities on the test set.

```
# get the predicted probabilities on the test set

undersampled_pred = predict(undersampled_model, newdata =
test, type = "response")

oversampled_pred = predict(oversampled_model, newdata = test,
type = "response")
```

8. Apply a threshold of 0.5 to convert the probabilities into binary classes.

```
# apply threshold to convert the probabilities into binary
classes

undersampled_pred_class = ifelse(undersampled_pred > 0.5, 1,
0)

oversampled_pred_class = ifelse(oversampled_pred > 0.5, 1, 0)
```

9. Calculate the confusion matrix.

```
# calculate the confusion matrix

undersampled_cm = table(predicted = undersampled_pred_class,
actual = test$Class_num)

oversampled_cm = table(predicted = oversampled_pred_class,
actual = test$Class_num)

>>> undersampled_cm

         actual

predicted   0   1

        0 117  59

        1  79  45

>>> oversampled_cm
```

```
          actual
predicted   0   1
        0 115  59
        1  81  45
```

The result shows that both models deliver a similar performance using the undersampled or oversampled training dataset. Again, we would use another validation dataset, which can be taken from the original training set, to calibrate the model parameters further so that it performs better on the test set.

It turns out that we can also add a lasso or ridge penalty in a logistic regression model, as discussed in the following section.

Penalized logistic regression

As the name suggests, a penalized logistic regression model includes an additional penalty term in the loss function of the usual logistic regression model. Recall that a standard logistic regression model seeks to minimize the negative log-likelihood function (or equivalently, maximize the log-likelihood function), defined as follows:

$$Q(\boldsymbol{\beta}) = \frac{1}{N}\sum_{i=1}^{N} -[y_i \log p_i + (1-y_i)\log(1-p_i)]$$

Where $p_i = \frac{1}{1+e^{-(\beta_0+\beta_1 x_1^{(i)}+\beta_2 x_2^{(i)}+\cdots+\beta_p x_p^{(i)})}}$ is the predicted probability for input $x^{(i)}$, y_i is the corresponding target label, and $\boldsymbol{\beta} = \{\beta_0, \beta_1, \ldots, \beta_p\}$ are model parameters to be estimated. Note that we now express the loss as a function of the coefficient vector as it is directly determined by the set of parameters used in the model.

Since the penalty term aims to shrink the magnitude of the estimated coefficients, we would add it to the loss function so that the penalty terms will be relatively small (subject to a tuning hyperparameter λ). For the case of ridge penalty, we would add up the squared coefficients, resulting in the following penalized negative log-likelihood function:

$$Q_{ridge}(\boldsymbol{\beta}) = Q(\boldsymbol{\beta}) + \lambda\left||\boldsymbol{\beta}|\right|_2^2 = Q(\boldsymbol{\beta}) + \lambda\sum_{j=1}^{p}\beta_j^2$$

Correspondingly, the penalized negative log-likelihood function using the lasso regularization term takes the following form:

$$Q_{lasso}(\boldsymbol{\beta}) = Q(\boldsymbol{\beta}) + \lambda|\boldsymbol{\beta}| = Q(\boldsymbol{\beta}) + \lambda\sum_{j=1}^{p}|\beta_j|$$

In both cases, the penalty term has the potential effect of shrinking the magnitude of the coefficients toward zero relative to the original maximum likelihood estimates. This can help to prevent overfitting by controlling the complexity of the model estimation process. The tuning hyperparameter λ controls the amount of shrinkage. In particular, a larger λ adds more weight to the

penalty term and thus leads to more shrinkage effect, while a smaller λ puts less weight on the overall magnitude of the estimated coefficients.

Let us illustrate the process of development of penalized logistic regression models. The task can be achieved using the `glmnet` package, which supports both lasso and ridge penalties. In the following code snippet, we use the first nine columns as predictors to model the binary outcome.

```
# Create a matrix of predictors and a response vector

# For glmnet, we need to provide our data as matrices/vectors

X = GermanCredit[1:nrow(GermanCredit), 1:9]

y = GermanCredit$Class_num

# Define an alpha value: 0 for ridge, 1 for lasso, between 0 and 1
for elastic net

alpha_value = 1 # for lasso

# Run the glmnet model

fit = glmnet(X, y, family = "binomial", alpha = alpha_value)
```

Here, we set `alpha=1` to enable the lasso penalty. Setting `alpha=0` enables the ridge penalty, and those valued between 0 and 1 correspond to the elastic net penalty.

Note that this procedure would evaluate a sequence of values for the hyperparameter λ, giving us an idea of the level of impact on the resulting coefficients based on the penalty. In particular, we can plot out the coefficient paths, as shown in the following, indicating the resulting coefficients for different values of λ.

```
# plot coefficient paths

>>> plot(fit, xvar = "lambda", label = TRUE)
```

Running the codes generates *Figure 13.18*, which suggests that more parameters are shrunken to zero when λ gets big.

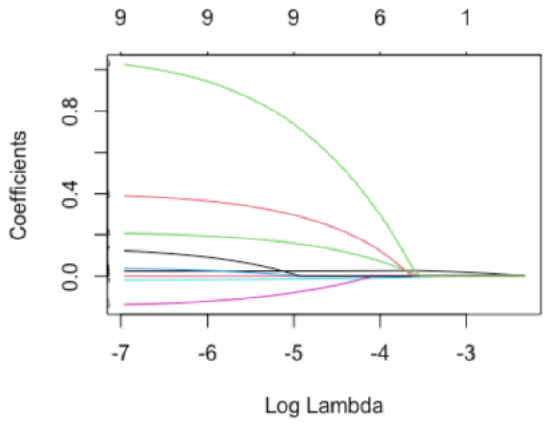


Figure 13.18 – Visualizing the coefficient paths using lasso penalized logistic regression.

The next section discusses a more general setting: the multinomial logistic regression, a model class for multi-class classification.

Extending to multi-class classification

Many problems feature more than two classes. For example, the S&P bond rating includes multiple classes, such as AAA, AA, A, and more like these. Corporate client accounts in a bank are categorized into good credit, past due, overdue, doubtful, or loss. Such settings require the multinomial logistic regression model, which is a generalization of the binomial logistic regression model in the multi-class classification context. Essentially, the target variable y can take more than two possible discrete outcomes and allows for more than two categorical values.

Assume that the target variable can take on three values, giving $y \in \{0,1,2\}$. Let us choose the class 0 as the pivot value or the baseline. We will model the odds of the probabilities of the other categories (class 1 and 2) relative to this baseline. In other words, we have:

$$\frac{p(y=1)}{p(y=0)} = e^{z_1}$$

$$\frac{p(y=2)}{p(y=0)} = e^{z_2}$$

Therefore, we relative ratio of the predicted probabilities for each class is

$$p(y=2) : p(y=1) : p(y=0) = e^{z_2} : e^{z_1} : 1$$

Since we know that

$$p(y=2) + p(y=1) + p(y=0) = 1$$

Thus we have

$$p(y=2) = \frac{e^{z_2}}{e^{z_2} + e^{z_1} + 1}$$

$$p(y=1) = \frac{e^{z_1}}{e^{z_2} + e^{z_1} + 1}$$

$$p(y=0) = \frac{1}{e^{z_2} + e^{z_1} + 1}$$

Again, one of the main assumptions in a multinomial logistic regression model is that the log odds consist of a linear combination of the predictor variables. This is the same assumption as in the binary logistic regression. The corresponding interpretation of the coefficients in a multinomial logistic regression will also change slightly. In particular, each coefficient now represents the change in the log odds of the corresponding category relative to the baseline category for a one unit change in the corresponding predictor variable, while holding all other predictors constant.

We can rely on the `multinom()` function from the `nnet` package to create a multinomial logistic regression model. In the following code snippet, we use the previous `mtcars` dataset and convert the `gear` variable to a factor as the target variable.

```
library(nnet)

# convert gear to factor

mtcars$gear = as.factor(mtcars$gear)

>>> table(mtcars$gear)
```

```
3  4  5
15 12  5
```

The frequency count shows a total of three categories in the variable. Next, we use `mpg`, `hp`, and `disp` to predict `gear` in a multinomial logistic regression model.

```
# fit the model
multinom_model = multinom(gear ~ mpg + hp + disp, data = mtcars)
# weights:  15 (8 variable)
initial  value 35.155593
iter  10 value 10.945783
iter  20 value 9.011992
iter  30 value 8.827997
iter  40 value 8.805003
iter  50 value 8.759821
iter  60 value 8.742738
iter  70 value 8.737492
iter  80 value 8.736569
final  value 8.735812
converged
```

The output message suggests that we have a total of eight variables in the model. This makes sense since we have four variables (intercept, `mpg`, `hp`, and `disp`) to model the difference between 4-gear and 3-gear cars in one submodel, and another four variables to model the difference between 5-gear and 3-gear cars in another submodel.

Let us view the summary of the model.

```
# view summary of the model
>>> summary(multinom_model)
Call:
multinom(formula = gear ~ mpg + hp + disp, data = mtcars)

Coefficients:
  (Intercept)       mpg         hp        disp
4   0.3892548 0.2707320 0.02227133 -0.04428756
5 -17.6837050 0.6115097 0.15511207 -0.08815984

Std. Errors:
```

```
  (Intercept)       mpg         hp       disp
4    17.30456 0.5917790 0.05813736 0.02735148
5    15.46373 0.5754793 0.08651377 0.06060359


Residual Deviance: 17.47162
AIC: 33.47162
```

As expected, the summary includes two sets of coefficients for the two submodels (indexed by 4 and 5, respectively).

Lastly, let us make predictions using the multinomial logistic regression model and calculate the confusion matrix.

```
# make prediction
predicted_gears = predict(multinom_model, newdata = mtcars)
# view the confusion matrix
>>> table(Predicted = predicted_gears, Actual = mtcars$gear)
         Actual
Predicted  3  4  5
        3 14  0  0
        4  1 12  1
        5  0  0  4
```

The result suggests a decent classification performance with only two misclassifications.

Summary

In this chapter, we delved into the world of logistic regression, its theoretical underpinnings and practical applications. We started by exploring the fundamental construct of logistic regression and its comparison with linear regression. We then introduced the concept of the sigmoid transformation, a crucial element in logistic regression, which ensures the output of our model is bounded between 0 and 1. This section helped us better understand the advantages of logistic regression for binary classification tasks.

Next, we delved into the concept of log odds and odds ratio, two critical components of the logistic regression model. Understanding these allowed us to comprehend the real-world implications of the model's predictions and to interpret its parameters effectively. The chapter then introduced the cross-entropy loss, the cost function used in logistic regression. Specifically, we discussed how this loss function ensures our model learns to predict accurate probabilities for the binary classes.

When it comes to evaluating a logistic regression, we learned about various metrics, including accuracy, error rate, precision, recall, sensitivity, specificity, and AUC. This understanding will allow us to assess the performance of our logistic regression model accurately.

An important discussion revolved around the handling of imbalanced datasets, a common scenario in real-world data. We understood the effects of data imbalance on our model and learned about strategies, such as resampling techniques, to handle such situations effectively. Further, we discussed penalized logistic regression where we incorporate L1 (Lasso) or L2 (Ridge) regularization into our logistic regression model. This penalization technique helps us prevent overfitting by keeping the magnitude of the model weights small and creating simpler models when dealing with high-dimensional data.

Finally, we touched upon multinomial logistic regression, an extension of logistic regression used for multi-class classification problems. This part provided insight into handling situations where the target variable consists of more than two classes.

By the end of this chapter, we have gathered an extensive understanding of logistic regression, its implementation, and its nuances. This knowledge lays the groundwork for diving deeper into more complex classification methods and strategies.

In the next chapter, we will cover Bayesian statistics, another major branch of statistical modeling.