Deep Learning with R

Chenshu Liu

April 2022

Contents

DL 01 Regression as a first step in deep learning

Prediction from linear regression

```
# Scenario: predicting sales performance
sales <- c(3, 4, 2, 4, 5, 6, 3, 9, 1, 12)
```

In the simplest way possible, we can simple find the mean of all the observations, and use the mean as our prediction for future sales performances

$$Performance = \frac{\Sigma_{i=1}^{n} x_i}{n}$$

```
mean.sales <- mean(sales)
mean.sales
```

```
## [1] 4.9
```

How to measure error in prediction

However, using only the mean as future prediction is not close to accurate, we are bound to make errors. Thus, to measure the magnitude of the error we are making using the mean, we introduce the concept of *sum of squared errors (SSE)*:

$$SSE = \Sigma_{i=1}^{n} (x_i - \bar{X})^2$$

Where $\bar{X}$ is the mean of the observations.

NOTE: There the differences are squared because there can be positive and negative numbers in the differences, and summing positive and negative differences together will make 0. Thus, we need to **square** the differences to make them significant.

```
SSE = sum((sales - mean.sales)^2)
SSE
```

```
## [1] 100.9
```

However, the measurement of deviation from actual observations has some limitations:

1. In the calculation of SSE, we summed all the differences together, which means the sample size plays a role in the calculation of the deviation. Thus, in order to rule out the effect of sample size on SSE, we introduce the idea of **variance**: $var = \frac{\Sigma_{i=1}^{n} (x_i - \bar{X})^2}{n-1}$, where $n-1$ is the degrees of freedom. Now, the standard deviation is a square-root version of the measurement of deviation, which makes more sense.
2. Because the differences are squared, it is hard for us to interpret the deviation, thus, we further introduce the idea of *standard deviation*: $s = \sqrt{\frac{\Sigma_{i=1}^{n} (x_i - \bar{X})^2}{n-1}}$

Both the variance and standard deviation describes how bad the model's prediction is

```r
variance <- (sum((sales - mean.sales)^2))/(length(sales) - 1)
variance
```

```
## [1] 11.21111
```

```r
# base R function
var(sales)
```

```
## [1] 11.21111
```

```r
standard.deviation <- sqrt((sum((sales - mean.sales)^2))/(length(sales) - 1))
standard.deviation
```

```
## [1] 3.3483
```

```r
# base R function
sd(sales)
```

```
## [1] 3.3483
```

So, predictions are bound to have errors:

$$y_i = \bar{X} + \epsilon_i$$
$$target = model + error$$

Take away: After learning about how to measure the accuracy of a prediction model, we now need to know how to optimize the model and letting it to generate more accurate predictions. The task of optimizing the model that minimizes prediction error is the in the scope of Deep Learning!

How to potentially reduce prediction error

Assume that we have a base-line linear regression model that has a SSE we call *sst* (unsystematic variance), if we tune the slope and intercept of the base-line linear regression model, it has a new SSE we call *ssm* (systematic variance). Thus, in order to **measure the improvement in prediction accuracy brought by tuning the parameters in the model**, we introduce the measurement of R-squared:

$$R^2 = \frac{ssm}{sst}$$

R^2 describes the amount of variance that can be described by the new, improved model, with respect to the base-line model.

Preview on next section

Thought question: we know that tuning the parameters in the model can lead to improvement in the prediction performance, but how can we find the optimal parameter that can minimize the prediction error (**core** goal in deep learning). This is what we will be covering in the next section

Reference video: https://www.youtube.com/watch?v=0F2bBZiirlg&list=PLH5_eZVldmtUCZWp-eL0lVL7SA6qyDIf9&index=1

DL 02 Linear regression as a Simple Learner "SL"

Related functions/calculations in prediction

Prediction function

From last section, we introduced that predictions can be made with a linear function in the form of:

$$\hat{y_i}(x_i) = \beta_0 + \beta_1 x_i$$

Where $\hat{y}$ is the predicted value based on the prediction function

Loss function

Loss function is used to measure the amount of error in **one of** our predictions using the model:

$$L(x_i) = [\hat{y_i}(x_i) - y_i]^2$$

The loss function is just the SSE that we introduced in the last section, where we take the square of the differences between observed and predicted value, at the ith position (because there are many observations and predictions)

Cost function

The cost function is a little from the loss function because the cost function is calculated from the persepective of the overall prediction, instead of each individual prediction's deviation (calculated by loss function):

$$\begin{aligned} C(\beta_0, \beta_1) &= \frac{1}{n}\Sigma_{i=1}^{n} L \\ &= \frac{1}{n}\Sigma_{i=1}^{n} [\hat{y_i}(x_i) - y_i]^2 \\ &= \frac{1}{n}\Sigma_{i=1}^{n} [\beta_0 + \beta_1 x_i - y_i]^2 \end{aligned}$$

How to find the optimial parameters

Say we find a cost function that is:

$$\begin{aligned} C &= \frac{1}{5} \times [\beta_0 + \beta_1(1.3) - 0.7]^2 + \cdots + [\beta_0 + \beta_1(3.3) - 3.5]^2 \\ &= 6.55 - 4.68\beta_0 + \beta_0^2 - 13.132\beta_1 + 5.08\beta_0\beta_1 + 7.002\beta_1^2 \end{aligned}$$

We can see that the cost function is composed of the two parameters β_0 and β_1, and we also know that we want to minimize the cost function. So, our goal now is to find optimal values of β_0 and β_1 so that the cost function is at its minimum. The way to achieve so, its through using **partial derivatives**

$$\frac{\partial C}{\partial \beta_0} = 2\beta_0 + 5.08\beta_1 - 4.68 \tag{1}$$

$$\frac{\partial C}{\partial \beta_1} = 5.08\beta_0 + 14.004\beta_1 - 13.132 \tag{2}$$

By solving (1) and (2), we can obtain the following augmented matrix for the linear system:

$$\begin{bmatrix} 2 & 5.08 & 4.68 \\ 5.08 & 14.004 & 13.132 \end{bmatrix}$$

Then, we can reduce the augmented matrix which give us the final values for $\beta_0 = -0.532267$ and $\beta_1 = 1.13081$

Gradient descent

The case above is a two dimensional (having to variables), which is rather simple. However, in real-life scenarios, we often have many parameters and we still need to find the set of parameters that minimizes the cost function, so we ought to find a more generalized way to find optimal parameters, so we introduce the idea of **gradient descent**

1. Randomly choose a point in space and find the derivative (i.e. slope) of the cost function at that particular point: $slope_x$
2. Define a learning rate (LR), which is a predefined value for gradient descent
3. Calculate step: $x_{new} = x - LR \times slope_x$
4. Repeat the same process from step1-3 for x_{new}

Reference video: https://www.youtube.com/watch?v=FrceOv_oJac&list=PLH5_eZVldmtUCZWp-eL0lVL7SA6qyDIf9&index=2

DL 03 Linear regression as a Shallow Neural Network "SNN"

Multiple linear regression in R

```
df <- read.csv("MultipleLinearRegression.csv")
df
```

```
##      x1   x2  x3     y
## 1  20.1 39.3 1.3 394.5
## 2  23.6 31.6 1.5 211.4
## 3  29.2 36.9 1.4 251.4
## 4  29.3 34.1 1.2  85.4
## 5  30.0 37.2 1.2 248.6
## 6  22.9 39.3 1.9  46.0
## 7  25.1 33.0 1.3 252.5
## 8  27.7 36.0 2.0 315.4
## 9  24.7 34.5 1.3 120.5
## 10 24.2 39.8 1.5 110.1
```

After preliminary view of the data, we can see that there are three independent variables x_1, x_2, x_3, so we are dealing with a multiple linear regression:

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 \approx y$$

Now, there are four parameters (i.e. $\beta_0, \beta_1, \beta_2, \beta_3$) that we need to tune, in order to optimize prediction, so we have the loss function for the four parameters:

$$L^{(i)}(\beta_0, \beta_1, \beta_2, \beta_3) = (\beta_0 + \beta_1 x_1^{(i)} + \beta_2 x_2^{(i)} + \beta_3 x_3^{(i)} - y^{(i)})^2$$

```
# multiple linear regression
mlr <- lm(y ~., data = df)
summary(mlr)
```

```
##
## Call:
## lm(formula = y ~ ., data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -151.55  -96.65   22.22   56.09  164.10
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 342.7300   777.7247   0.441    0.675
## x1           -3.5850    14.2899  -0.251    0.810
## x2            0.2326    16.6230   0.014    0.989
## x3          -38.0145   166.3937  -0.228    0.827
##
## Residual standard error: 134.4 on 6 degrees of freedom
## Multiple R-squared:  0.01687,    Adjusted R-squared:  -0.4747
## F-statistic: 0.03433 on 3 and 6 DF,  p-value: 0.9906
```

Single layer neural network

Based on the multiple linear regression we conducted in last subsection, we are actually getting into neural network layers. The multiple linear regression itself can be considered as a single layer network (Figure 1: Single Layer Neural Network), where the inputs are taken into hidden layers and multiplied with the weights (the parameters), and then output the prediction in the output layer

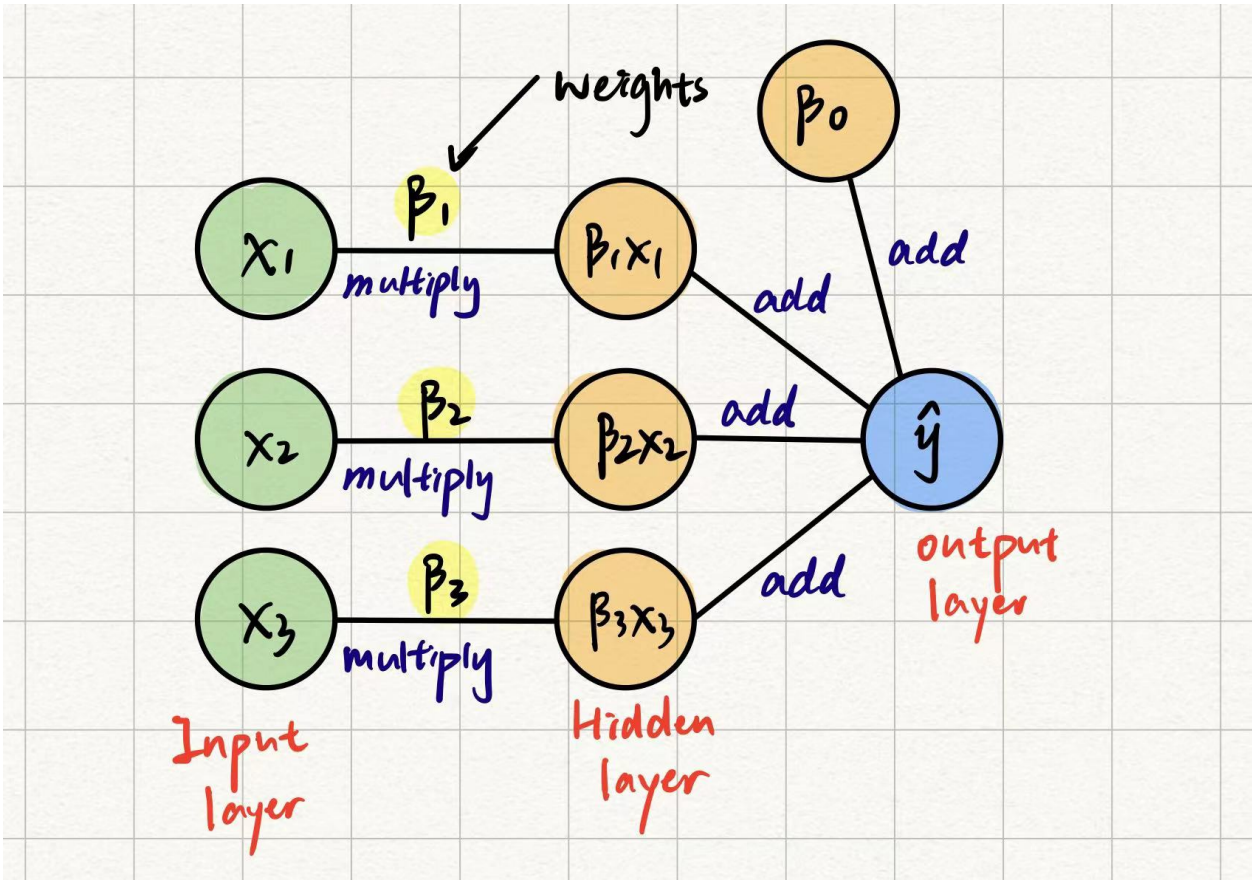


Figure 1: Single Layer Neural Network

Now, here's when things get interesting:

1. We can use **forward propagation** to find output prediction (i.e. feeding in input values, and calculate for output)
2. We can also use **backward propagation** to better tune the parameters (i.e. using the predicted values, we can go back to tune the parameters and allow for better prediction performance)

Reference video: https://www.youtube.com/watch?v=ZX4YSidnQaI&list=PLH5_eZVldmtUCZWp-eL0lVL7SA6qyDIf9&index=6

DL 04 Logistic regression as a Neural Network

Why do we need logistic regression?

So far, we have looked at ways which we can find predictions about numerical outcome, but there are real-life cases where the prediction of binary outcomes is needed. It's where logistic regression comes into play. Logistic regression is specifically designed to deal with prediction of binary outcomes

Mechanism of logistic regression

Logistic regression is similar to simple linear regression in every way except for the outcome variable. Simple linear regression produces numerical outcomes while logistic regression produces binary outcome. Because the rest of the algorithm is just the same as simple linear regression, logistic regression also has parameters $\beta_0, \beta_1, \ldots, \beta_n$, and the way in which we can achieve the best prediction model is to find values for the parameters that can minimize the cost function

For logistic regression, the solution is a sigmoidal function:

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

Where z in the expression can be expressed as $z(\beta_0, \beta_1, \beta_2, \beta_3, \beta_4) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4$. One of the key property of sigmoidal function is that its solution is always between 0 and 1, and we can define a threshold (i.e. cutoff) to determine which side the outcome belongs to (i.e. 0 or 1), which fits the purpose of deriving binary outcomes

Reference video: https://www.youtube.com/watch?v=7TjN1kuJidA&list=PLH5_eZVldmtUCZWp-eL0lVL7SA6qyDIf9&index=7

DL 05 Deep Neural Network in R - Example

Libaries & Data

```
# import spreadsheet files
library(readr)
# deep learning package
library(keras)
# dynamic interactive tables
library(DT)

data <- read_csv("SimulatedBinaryClassificationDataset.csv",
                 col_names = TRUE)
summary(data)
```

```
##       Var1                Var2                Var3                Var4
##  Min.   :-4.404214   Min.   :-4.413886   Min.   :-4.374043   Min.   :-4.25136
##  1st Qu.:-0.680166   1st Qu.:-0.677792   1st Qu.:-0.671550   1st Qu.:-0.68108
##  Median :-0.000612   Median :-0.005248   Median : 0.004296   Median :-0.01094
##  Mean   :-0.001853   Mean   :-0.005588   Mean   : 0.005143   Mean   :-0.00506
##  3rd Qu.: 0.676390   3rd Qu.: 0.667976   3rd Qu.: 0.675082   3rd Qu.: 0.67401
##  Max.   : 3.766234   Max.   : 4.562115   Max.   : 3.826255   Max.   : 3.82499
##       Var5                Var6                Var7
##  Min.   :-6.562910   Min.   :-4.015382   Min.   :-4.004546
##  1st Qu.:-0.712045   1st Qu.:-1.061773   1st Qu.:-0.675130
##  Median : 0.082631   Median :-0.063422   Median :-0.001151
##  Mean   : 0.001955   Mean   :-0.004981   Mean   :-0.000119
##  3rd Qu.: 1.303538   3rd Qu.: 1.001450   3rd Qu.: 0.668163
##  Max.   : 3.347969   Max.   : 6.161397   Max.   : 3.878217
##       Var8                Var9               Var10              Target
##  Min.   :-4.003598   Min.   :-3.601650   Min.   :-3.84684   Min.   :0.0000
##  1st Qu.:-0.668504   1st Qu.:-1.023422   1st Qu.:-0.73220   1st Qu.:0.0000
##  Median : 0.012026   Median :-0.244104   Median : 0.07816   Median :1.0000
##  Mean   : 0.006157   Mean   : 0.002998   Mean   :-0.00422   Mean   :0.5001
##  3rd Qu.: 0.677135   3rd Qu.: 1.002847   3rd Qu.: 0.82045   3rd Qu.:1.0000
##  Max.   : 4.202026   Max.   : 5.387780   Max.   : 3.58177   Max.   :1.0000
```

Data preprocessing

Because deep learning is achieved through manipulation of matrices, we cannot pass in dataframe format data, so we need to change the format of the data to matrix during preprocessing

```
# data.frame --> matrix
data  <- as.matrix(data)
# remove the row and col names, leaving only numerical values
dimnames(data) = NULL
mode(data)
```

```
## [1] "numeric"
```

Also, deep learning involves separate training and testing phases in order optimize the model's performance, so we need tp prepare for training and testing sets during preprocessing

```r
# train and test split index
set.seed(123)
index <- sample(2,
                nrow(data),
                replace = TRUE,
                prob = c(0.9, 0.1))
table(index)
```

```
## index
##     1     2
## 45119  4881
```

```r
# data splitting
x_train <- data[index == 1, 1:10]
x_test <- data[index == 2, 1:10]
y_test_actual <- data[index == 2, 11]
```

In cases where the target variable has multiple categories, our deep learning network wouldn't be able to process raw categorical information, so we need to convert the categories into numerical matrix format. One common way of encoding the target variable is through **one-hot encoding**:

Simply put, one-hot encoding would expand the column vector of categories and make each category an independent column. If the original label fits into a category, then the column with the category will be labeled 1, otherwise 0

```r
# use teh to_categorical function in keras package for one-hot encoding
y_train <- to_categorical(data[index == 1, 11])
```

```
## Loaded Tensorflow version 2.8.0
```

```r
y_test <- to_categorical(data[index == 2, 11])
```

Creating a simple model

```r
model <- keras_model_sequential()

model %>%
  # layer_dense means a densely connected layer
  layer_dense(name = "DeepLayer1",
              units = 10, # hyperparameter: the number of nodes
              activation = "relu",
              # the first layer need to have specification about the input dimension
              input_shape = c(10)) %>%
  layer_dense(name = "DeepLayer2",
              units = 10,
              activation = "relu") %>%
  layer_dense(name = "OutputLayer",
              units = 2,
              # softmax function will provide probabilities of the nodes
              activation = "softmax")

summary(model)
```

```
## Model: "sequential"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## ================================================================================
##  DeepLayer1 (Dense)                 (None, 10)                      110
##
##  DeepLayer2 (Dense)                 (None, 10)                      110
##
##  OutputLayer (Dense)                (None, 2)                       22
##
## ================================================================================
## Total params: 242
## Trainable params: 242
## Non-trainable params: 0
## ________________________________________________________________________________
```

According to the summary table of the deep learning model, we can see that the number of parameters is very large:

1. DeepLayer 1 has 110 parameters after passing the 10 input values, this is because neural network connects all the input with the nodes in the hidden layer which results in $10 \times 10 = 100$, and there is a bias term associating with every input, so the total number of parameters is $10 + 10 \times 10 = 110$
2. DeepLayer 2 also has 110 parameters, which is from the same reason as DeepLayer 1
3. OutputLayer has 22 parameters: $10 \times 2 + 2 = 22$
4. Thus, the total number of parameters in our two hidden layer network is already so large: $110 + 110 + 22 = 242$, so we need to tune 242 parameters for the model through forward/backward propagation, which can make the performance so much better

Compile the model

```
model %>% compile(
  # another way to calculate loss, besides mean-squared-error
  loss = "categorical_crossentropy",
  # a special way of gradient descent
  optimizer = "adam",
  # measurement of model performance - using accuracy to measure
  metrics = c("accuracy"))
```

Fitting the data

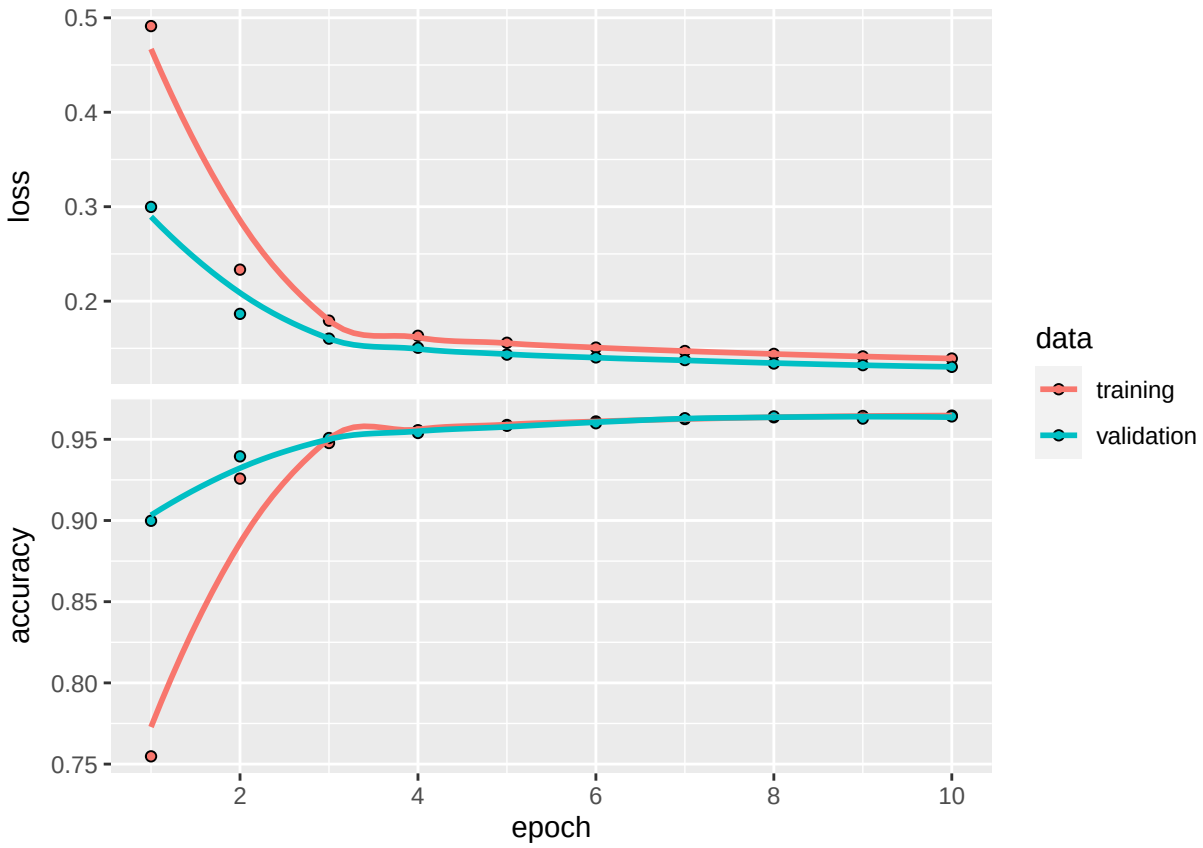
```
history <- model %>%
  fit(x_train,
      y_train,
      # number of full forward & backward propagation
      # (i.e. run 10 times back and forth of all samples)
      epoch = 10,
      # instead of propagating the whole dataset at one go, use smaller batches
      batch_size = 256,
```

```
      # splitting the training set to test itself during training
      validation_split = 0.1,
      verbose = 2)
```

Arguments for fitting the model (hyperparameters):

1. epoch: one forward pass and one backward pass of **all** the training samples
2. batch size: the number of training examples in one forward/backward pass, usually, for better memory performance, we use values that are 2^n

```
# plot the training history
plot(history)
```



Model evaluation

Be aware of the new update in tensorFlow 2.8.0: https://keras.rstudio.com/reference/predict_proba.html

```
model %>%
  evaluate(x_test,
           # NOTE: here we are still using the one-hot encoded y_test
           y_test)
```

```
##      loss  accuracy
## 0.1531303 0.9598443
```

```r
# form predictions
pred <- model %>%
  predict(x_test) %>%
  k_argmax()
# reference for converting tensor to R data types
# https://torch.mlverse.org/technical/tensors/
pred <- as.array(pred)
table(Predicted = pred,
      # NOTE: for confusion matrix, we are using the original y_test_actual, not encoded
      Actual = y_test_actual)
```

```
##           Actual
## Predicted    0    1
##         0 2419  157
##         1   39 2266
```