

DSE6211 Final Project

Daniel Jackson

2024-06-30

Project Overview

Customer: ABC Hotels.

Business Need: ABC Hotels would like to identify bookings that have a high risk of cancellation. The risk of cancellation should be a value between 0 and 1, so it can be interpreted as the probability of cancellation. With this capability, hotel management can target bookings that have a high risk (i.e., probability) of cancellation with additional advertisements and/or offers in an effort to prevent them from being cancelled.

Data: ABC Hotels has provided a data set containing over 35,000 bookings for which it is known whether or not the booking was cancelled. Students are required to use this data set provided in the zipped folder below.

Questions to Answer

The first step in the machine learning process is to carefully consider the objectives (i.e., the business needs) in the context of the available data, appropriate and applicable machine learning methods, as well as the expected analytic and informational outcomes. Consequently, the Analytic Plan is a detailed and specific outline addressing at least the following:

- 1.) What is the label (i.e., the target or dependent variable) for the supervised classification problem?
- 2.) What data processing is needed and how will it be performed?
- 3.) What features will be initially included?
- 4.) What are the expected analytic and informational outcomes to be produced?
- 5.) How will the model be used in practice?

Below, we included the R code of our exploratory data analysis. Within this analysis, we will be addressing and answering the questions above.

Analytic Plan R Code

```
# Libraries Used  
library(dplyr)
```

```

library(lubridate)

library(purrr)
library(caret)

library(ggplot2)
library(MESS)
library(reticulate)

library(tensorflow)

library(keras)

use_virtualenv("my_tf_workspace", required = TRUE)

# Read in CSV
hotel_df = read.csv("project_data.csv")

# Check for NA values
colSums(is.na(hotel_df))

##           Booking_ID           no_of_adults
##           0           0
##      no_of_children      no_of_weekend_nights
##           0           0
##      no_of_week_nights      type_of_meal_plan
##           0           0
##      required_car_parking_space      room_type_reserved
##           0           0
##           lead_time      arrival_date
##           0           0
##      market_segment_type      repeated_guest
##           0           0
##      no_of_previous_cancellations no_of_previous_bookings_not_canceled
##           0           0
##           avg_price_per_room      no_of_special_requests
##           0           0
##           booking_status
##           0

# No NA values in data frame. This is great!

# Check unique values of booking status. This will be our response variable
# for our analysis.
unique(hotel_df$booking_status)

## [1] "not_canceled" "canceled"

```

Our response variable will be whether the customer canceled, or did not cancel their reservation. We want our response variable to be a binary variable. Let us have 0 represent not canceled and 1 represent canceled and let us convert booking_status to binary response variable. We also will

remove the Booking_ID column as we will not be interested in what their booking identification number will be for our analysis.

```
hotel_df = hotel_df %>%  
  mutate(booking_status = ifelse(booking_status == "canceled", 1, 0))  
  
# We will not need the booking ID code for each observation. Let us remove  
# that  
# variable  
hotel_df = hotel_df[, -which(names(hotel_df) == "Booking_ID")]
```

Now, we are left with 16 total variables. With 1 response variable, we have 15 predictors to use in our modeling. Let us look at a summary of our variables.

Variable Summary

We will now run through what each of our variables in the data set represent. We will identify what type of variable each one is and make any changes we see necessary to help make simplify syntax.

booking_status: This is our response variable. It is a binary variable with values 0 and 1. This is a classification variable, meaning we will be constructing qualitative predictive models in our analysis.

no_of_adults: This represents the number of adults in the reservation. This is a discrete quantitative variable. Let us change the name of the variable from no_of_adults to just adults to help with syntax during modeling. We will have a snippet of code below for all of the variables that get a column name change.

no_of_children: This represents the number of children in the reservation. This is a discrete quantitative variable. We will change name to children for easier syntax.

no_of_weekend_nights: This represents the number of weekend nights in the reservation. This is a discrete quantitative variable. We will change name to weekend_nights for easier syntax.

no_of_week_nights: This represents the number of weeknights in the reservation. This is a discrete quantitative variable. We will change name to week_nights for easier syntax.

type_of_meal_plan: This variable represents type of meal plan for each reservation. Let us look at the values.

```
unique(hotel_df$type_of_meal_plan)  
  
## [1] "meal_plan_1" "not_selected" "meal_plan_2" "meal_plan_3"  
  
# There is four options: not selected, 1, 2, or 3  
# Check counts of each value  
hotel_df %>%  
  count(type_of_meal_plan)
```

```

##   type_of_meal_plan    n
## 1      meal_plan_1 27802
## 2      meal_plan_2  3302
## 3      meal_plan_3     5
## 4      not_selected 5129

# type_of_meal_plan    n
# 1      meal_plan_1 27802
# 2      meal_plan_2  3302
# 3      meal_plan_3     5
# 4      not_selected 5129
# Let us change values to none, one, two or three
hotel_df = hotel_df %>%
  mutate(type_of_meal_plan = ifelse(type_of_meal_plan ==
                                    "not_selected", "meal_none",
type_of_meal_plan)) %>%
  mutate(type_of_meal_plan = ifelse(type_of_meal_plan ==
                                    "meal_plan_1", "meal_one",
type_of_meal_plan)) %>%
  mutate(type_of_meal_plan = ifelse(type_of_meal_plan ==
                                    "meal_plan_2", "meal_two",
type_of_meal_plan)) %>%
  mutate(type_of_meal_plan = ifelse(type_of_meal_plan ==
                                    "meal_plan_3", "meal_three",
type_of_meal_plan))
# Check count again to make sure code worked
hotel_df %>%
  count(type_of_meal_plan)

##   type_of_meal_plan    n
## 1      meal_none  5129
## 2      meal_one 27802
## 3      meal_three     5
## 4      meal_two  3302

# type_of_meal_plan    n
# 1      none  5129
# 2      one 27802
# 3      three     5
# 4      two  3302

# The code worked.
# This variable is a qualitative variable with four unique values.
# Let us change variable to meal_plan

```

required_car_parking_space: This represents number of parking spaces needed. This is a discrete quantitative variable. Let us change the variable name to parking_space.

room_type_reserved: This represents type of room reserved. Let us check values.

```
unique(hotel_df$room_type_reserved)
```

```
## [1] "room_type1" "room_type4" "room_type2" "room_type6" "room_type5"
## [6] "room_type7" "room_type3"
```

Check counts of each

```
hotel_df %>%
```

```
  count(room_type_reserved)
```

```
##   room_type_reserved     n
## 1      room_type1 28105
## 2      room_type2   692
## 3      room_type3     7
## 4      room_type4  6049
## 5      room_type5   263
## 6      room_type6   964
## 7      room_type7   158
```

```
# room_type_reserved     n
# 1      room_type1 28105
# 2      room_type2   692
# 3      room_type3     7
# 4      room_type4  6049
# 5      room_type5   263
# 6      room_type6   964
# 7      room_type7   158
```

There are 7 unique values. Let us change the names of unique values

```
hotel_df = hotel_df %>%
```

```
  mutate(room_type_reserved = ifelse(room_type_reserved ==
                                     "room_type1", "room_one",
```

```
room_type_reserved)) %>%
```

```
  mutate(room_type_reserved = ifelse(room_type_reserved ==
                                     "room_type2", "room_two",
```

```
room_type_reserved)) %>%
```

```
  mutate(room_type_reserved = ifelse(room_type_reserved ==
                                     "room_type3", "room_three",
```

```
room_type_reserved)) %>%
```

```
  mutate(room_type_reserved = ifelse(room_type_reserved ==
                                     "room_type4", "room_four",
```

```
room_type_reserved)) %>%
```

```
  mutate(room_type_reserved = ifelse(room_type_reserved ==
                                     "room_type5", "room_five",
```

```
room_type_reserved)) %>%
```

```
  mutate(room_type_reserved = ifelse(room_type_reserved ==
                                     "room_type6", "room_six",
```

```
room_type_reserved)) %>%
```

```
  mutate(room_type_reserved = ifelse(room_type_reserved ==
                                     "room_type7", "room_seven",
```

```
room_type_reserved))
```

```
hotel_df %>%
```

```
  count(room_type_reserved)
```

```
##   room_type_reserved      n
## 1      room_five      263
## 2      room_four    6049
## 3      room_one   28105
## 4      room_seven    158
## 5      room_six     964
## 6      room_three      7
## 7      room_two     692
```

Code worked.

This is a qualitative variable with seven different unique values.

We will change the variable name to room_type

lead_time: We will be removing the lead_time predictor. We will treat this as we did with the booking ID variable. This is more of a time stamp observation on the reservation. Therefore we will remove it.

```
hotel_df = hotel_df[, -which(names(hotel_df) == "lead_time")]
```

arrival_date: This represents arrival date of each customer. Let us pull just the months of the arrival time and group our observations into four seasons rather than months. We will group them into Spring, Summer, Fall and Winter.

We can use month function from the lubridate package for this

```
hotel_df = hotel_df %>%
```

```
  mutate(arrival_season = case_when(
    month(arrival_date) %in% c(1, 2, 12) ~ "winter",
    month(arrival_date) %in% c(3, 4, 5) ~ "spring",
    month(arrival_date) %in% c(6, 7, 8) ~ "summer",
    month(arrival_date) %in% c(9, 10, 11) ~ "fall"))
```

Let us drop arrival date column now

```
hotel_df = hotel_df[, -which(names(hotel_df) == "arrival_date")]
```

market_segment_type: This is a qualitative variable with five unique values that represents how the engaged customer booked.

```
unique(hotel_df$market_segment_type)
```

```
## [1] "offline"      "online"      "corporate"   "aviation"
## [5] "complementary"
```

The customer either booked offline (phone call), online, corporate, aviation or complementary. Let us change the variable to market_type

repeated_guest: This is a binary qualitative variable with 0 representing not a repeat guest and 1 representing repeat guest

```
unique(hotel_df$repeated_guest)
```

```
## [1] 0 1
```

Let us change variable name to repeat_guest.

no_of_previous_cancellations: This represents the number of previous cancellations by customer.

```
unique(hotel_df$no_of_previous_cancellations)
```

```
## [1] 0 3 1 2 11 4 5 13 6
```

This is a continuous quantitative variable. Let us change name to previous_cancellations

no_of_previous_bookings_not_canceled: This represents number of previous bookings that were not canceled.

```
unique(hotel_df$no_of_previous_bookings_not_canceled)
```

```
## [1] 0 5 1 3 4 12 19 2 15 17 7 20 16 50 13 6 14 34 18 8 10 23 11
49 47
## [26] 53 9 33 22 24 52 21 48 28 39 25 31 38 26 51 42 37 35 56 44 27 32 55
45 30
## [51] 57 46 54 43 58 41 29 40 36
```

This is a continuous quantitative variable. Let us change name to prev_not_cancel.

avg_price_per_room: This represents average booking price per room. This is a continuous quantitative variable. We will not change this variable for now.

no_of_special_requests: This represents number of special requests made by each customer. This is a continuous quantitative variable. We will change this variable to spec_requests

Let us change the variable names:

```
hotel_df = hotel_df %>%
  rename(adults = no_of_adults,
         children = no_of_children,
         weekend_nights = no_of_weekend_nights,
         week_nights = no_of_week_nights,
         meal_plan = type_of_meal_plan,
         parking_spaces = required_car_parking_space,
         room_type = room_type_reserved,
         market_type = market_segment_type,
         repeat_guest = repeated_guest,
         prev_cancel = no_of_previous_cancellations,
         prev_not_cancel = no_of_previous_bookings_not_canceled,
         spec_requests = no_of_special_requests)
```

We have now summarized and started our data pre-processing. There will be more data pre-processing in the following feature engineering section.

Feature Engineering

One-Hot Encoding Categorical Variables

Now, let us focus on one-hot encoding all our categorical variables. Let us first convert all of our character variables to factors. Once the variables are converted to factors, we can then one-hot encode them. Once that is done, we then want to scale all of the non-response variables.

```
# Let us find all of the character variables
character_variables = hotel_df %>%
  keep(is.character) %>%
  names()
print(character_variables)

## [1] "meal_plan"      "room_type"      "market_type"    "arrival_season"

# Now, let us convert all the character variables to factors
hotel_df = hotel_df %>%
  mutate(meal_plan = as.factor(meal_plan),
         room_type = as.factor(room_type),
         market_type = as.factor(market_type),
         arrival_season = as.factor(arrival_season))

# Confirm variables were converted to factors
factor_variables = hotel_df %>%
  keep(is.factor) %>%
  names()
print(factor_variables)

## [1] "meal_plan"      "room_type"      "market_type"    "arrival_season"

# Now, let us one-hot encode our factor variables
dummy_vars = dummyVars(~ meal_plan + room_type + market_type +
  arrival_season,
                      data = hotel_df,
                      levelsOnly = TRUE,
                      fullRank = TRUE)
encoded_hotel_df = predict(dummy_vars, newdata = hotel_df)

# Now let us combine data hotel_df and encoded_hotel_df
hotel_df = cbind(hotel_df, encoded_hotel_df)

# Now let us remove our original categorical variables
hotel_df = hotel_df[, -which(names(hotel_df) == "meal_plan")]
hotel_df = hotel_df[, -which(names(hotel_df) == "room_type")]
hotel_df = hotel_df[, -which(names(hotel_df) == "market_type")]
hotel_df = hotel_df[, -which(names(hotel_df) == "arrival_season")]

# Now, let us scale all of the predictors in the data frame. We will not be
scaling the response variable. This will conclude our data pre-processing.
After that, we can start building our neural networks
```



```

response_var = hotel_df$booking_status
predictors_scaled = scale(hotel_df[, -which(names(hotel_df) ==
"booking_status")])
hotel_df = cbind(response_var, as.data.frame(predictors_scaled))
hotel_df = hotel_df %>%
  rename(booking_status = response_var)

```

Analysis Gameplan

This will be a qualitative regression analysis. Our response variable is a binary qualitative variable with two values: 0 for not-cancelled and 1 for cancelled reservation. We have 26 predictor variables and the 1 response variable for our analysis.

What is our goal? Our goal is to fit a predictive model to help the ABC Hotels to identify bookings that have a high risk of cancellation using the data set given. The risk of cancellation will be a value between 0 and 1. The closer the probability is to 1, the higher risk of cancellation. Since we are trying to predict a qualitative response variable, we will fit at least one dense neural network model. In this portion of the project, we will specify the following aspects of the neural network(s) and discuss why/how they were chosen: number of layers, number of units for each layer, activation functions for each layer, loss function and optimization algorithm.

We have already started our data pre-processing above. We have one-hot encoded our categorical variables and scaled all our predictors.

We will create a training and test data set. We will train the model using the training data set and will use that trained model to predict the response variable in the test set. We will then be using confidence matrices, ROC, AUC, Confidence Curves, etc. to measure accuracy.

We will evaluate the neural networks using learning curves on training and validation sets. Is the model(s) underfitting or overfitting? Based on this, changes will be made to the architecture of the dense neural network(s). Based on the evaluation of the preliminary model(s), we will provide further data processing and feature engineering steps that will be implemented and investigated for the Final Report.

Training and Test Data

Let us look at the dimension of our data set:

```

dim(hotel_df)
## [1] 36238    27

```

There are 36238 observations and 27 variables. Let us define our training and test data. Since we have a good amount of observations, will be using 70% of our data to be the training data and 30% to be our test data.

```
set.seed(1)
train = sample(nrow(hotel_df), 0.7 * nrow(hotel_df))
train_df = hotel_df[train, ]
test_df = hotel_df[-train, ]
```

We have 25366 observations in our training data and 10872 observations in our test data. These data sets will be used to train our neural networks. We will use the trained neural network models to predict our booking_status response variables in the the test data.

Preliminary Neural Network Model

Let us fit a neural network model on the training data and use that to predict our test data. First we will need to separate the training features and labels and the test features and labels. Once that is done, we can choose how many nodes we want in each layer. We will use the rectified linear unit (relu) as the activation function in the intermediate layers. The final layer will use a sigmoid activation. This will output a probability between 0 and 1, indicating how likely a customer is to the target 1, which would represent them cancelling their reservation. Since our problem is a binary classification problem, we will use the binary_crossentropy loss function. We will use the rmsprop optimizer. We will use accuracy as our metric.

We will start with two layers. The first layer will have 20 nodes and the second layer will have 10. We will start with 100 epochs, a batch size of 100 and a validation split of 0.3. This means that 70% of our training data will be used to train the model and 30% of our training data will be used to validate the trained model.

We will then used the train model to predict the booking status labels in our test data.

```
# Training data
train_feat = train_df[, -which(names(train_df) == "booking_status")]
train_labels = as.matrix(train_labels)

train_labels = train_df[, which(names(train_df) == "booking_status")]
train_labels = as.matrix(train_labels)

# Test data
test_feat = test_df[, -which(names(test_df) == "booking_status")]
test_labels = as.matrix(test_labels)

test_labels = test_df[, which(names(test_df) == "booking_status")]
test_labels = as.matrix(test_labels)

# Fit neural network model
model = keras_model_sequential(list(
  layer_dense(units = 20, activation = "relu"),
  layer_dense(units = 10, activation = "relu"),
  layer_dense(units = 1, activation = "sigmoid")
))

compile(model,
```

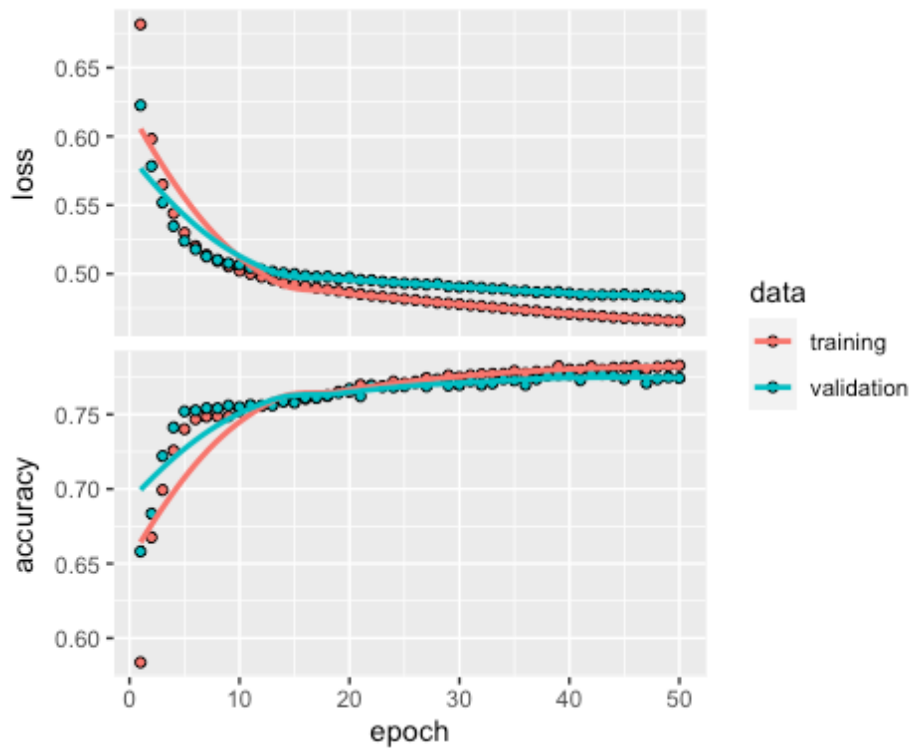
```

optimizer = "rmsprop",
loss = "binary_crossentropy",
metrics = "accuracy")

history = fit(model, train_feat, train_labels,
              epochs = 50, batch_size = 500, validation_split = 0.3)

plot(history)

```



```

set.seed(123)
results = model %>%
  evaluate(test_feat, test_labels)

## 340/340 - 0s - loss: 0.4706 - accuracy: 0.7808 - 274ms/epoch - 807us/step

print(results)

##      loss  accuracy
## 0.4706424 0.7808131

```

We used 50 epochs and a batch size of 500. We used a validation split of 30%.

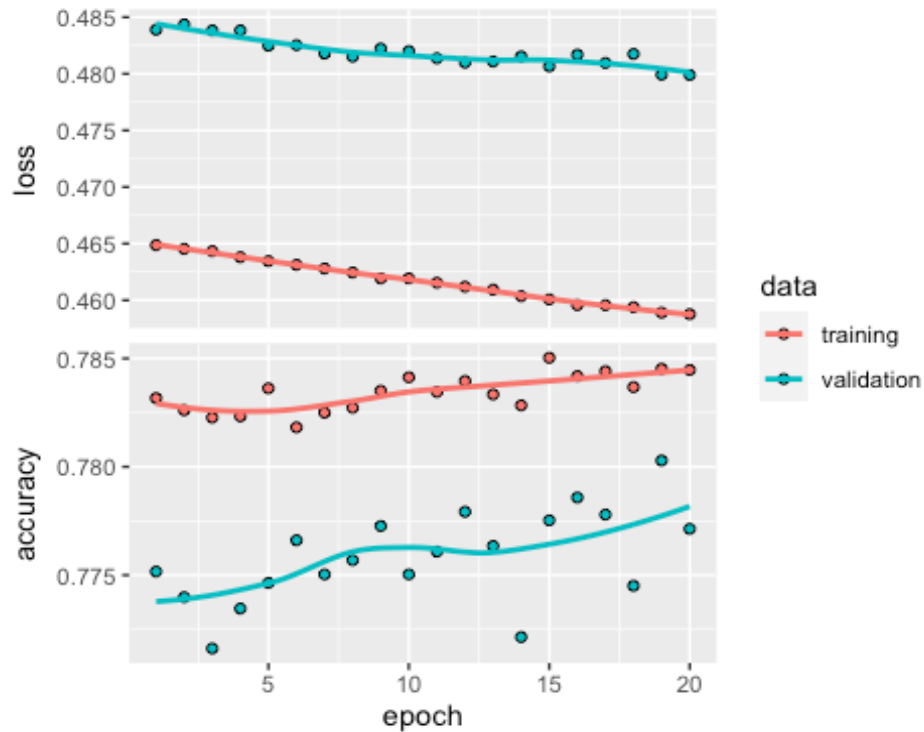
We see that our validation accuracy starts to level out around 20 epochs. This model produced an accuracy of 78% when predicting the test labels. Let us change the epoch size to 20.

```

history = fit(model, train_feat, train_labels,
              epochs = 20, batch_size = 500, validation_split = 0.3)

plot(history)

```



```
set.seed(123)
results = model %>%
  evaluate(test_feat, test_labels)

## 340/340 - 0s - loss: 0.4673 - accuracy: 0.7820 - 319ms/epoch - 939us/step

print(results)

##      loss  accuracy
## 0.4673166 0.7820088
```

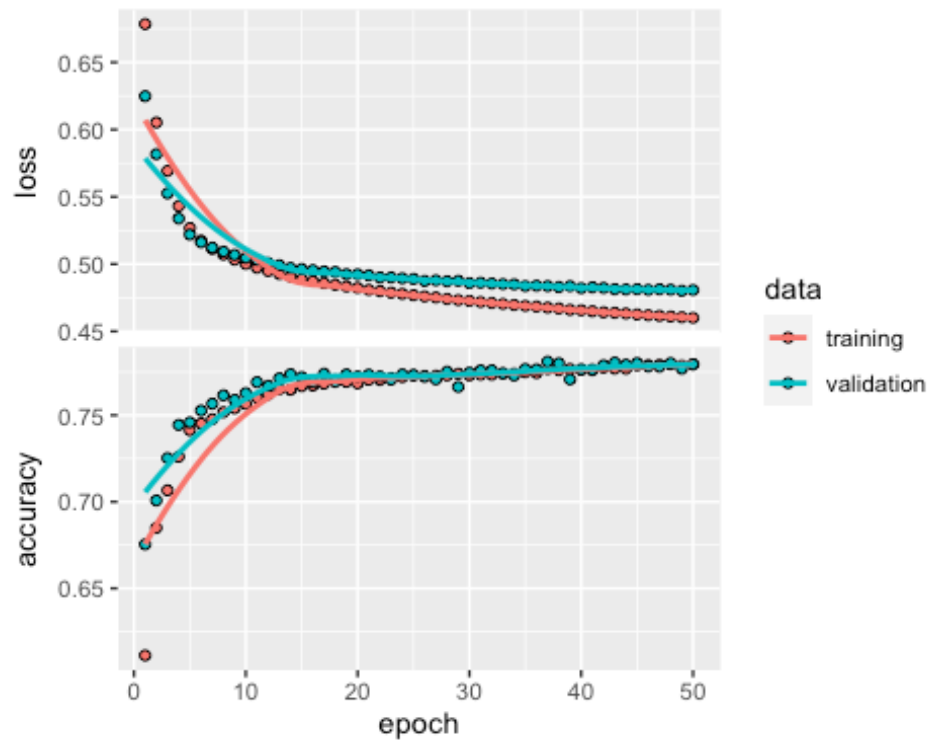
Now let us add another layer into our neural network model to see if we can increase the accuracy of predicting our test labels. Let us use 5 nodes in the layer that we added.

```
model_1 = keras_model_sequential(list(
  layer_dense(units = 20, activation = "relu"),
  layer_dense(units = 10, activation = "relu"),
  layer_dense(units = 5, activation = "relu"),
  layer_dense(units = 1, activation = "sigmoid")
))

compile(model_1,
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = "accuracy")

history = fit(model_1, train_feat, train_labels,
  epochs = 50, batch_size = 500, validation_split = 0.3)
```

```
plot(history)
```



```
set.seed(123)
results = model_1 %>%
  evaluate(test_feat, test_labels)

## 340/340 - 0s - loss: 0.4690 - accuracy: 0.7812 - 287ms/epoch - 845us/step

print(results)

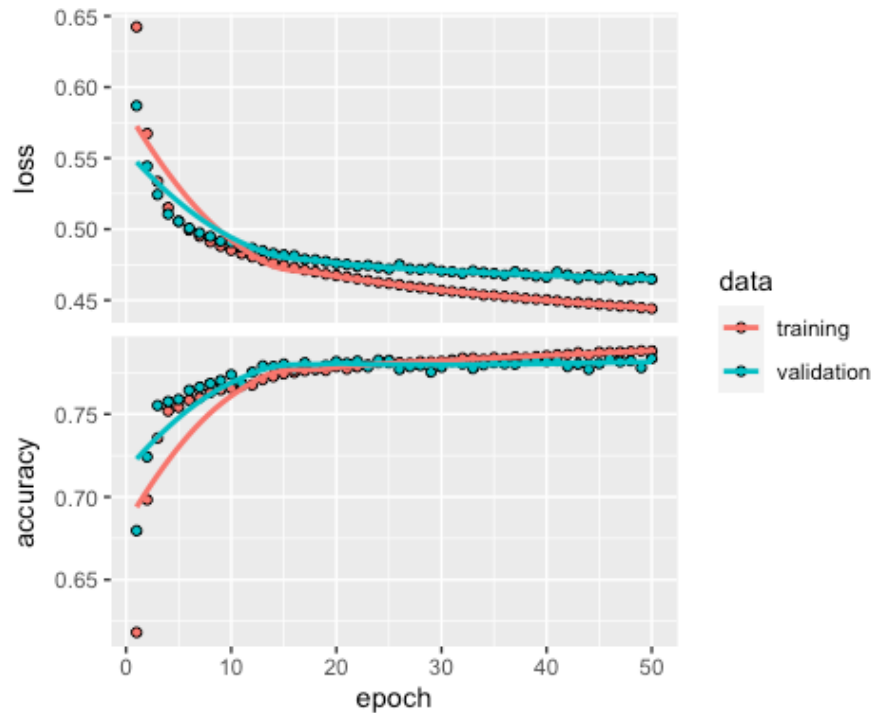
##      loss accuracy
## 0.469017 0.781181
```

This produced a similar test accuracy rate of 78%. Since the first model was less complex, let us revert back to that and double the nodes in each layer to see if that increases accuracy.

```
model_2 = keras_model_sequential(list(
  layer_dense(units = 40, activation = "relu"),
  layer_dense(units = 20, activation = "relu"),
  layer_dense(units = 1, activation = "sigmoid")
))

compile(model_2,
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = "accuracy")
```

```
history = fit(model_2, train_feat, train_labels,
              epochs = 50, batch_size = 500, validation_split = 0.3)
plot(history)
```



```
set.seed(123)
results = model_2 %>%
  evaluate(test_feat, test_labels)

## 340/340 - 0s - loss: 0.4584 - accuracy: 0.7888 - 273ms/epoch - 804us/step

print(results)

##      loss  accuracy
## 0.4583837 0.7888153
```

This produced a slightly better test accuracy of approximately 79%. Not much improvement. Since this model was slightly better, let us re-fit our neural network model with a decrease in batch size to see if that helps.

```
model_3 = keras_model_sequential(list(
  layer_dense(units = 40, activation = "relu"),
  layer_dense(units = 20, activation = "relu"),
  layer_dense(units = 1, activation = "sigmoid")
))

compile(model_3,
        optimizer = "rmsprop",
        loss = "binary_crossentropy",
```

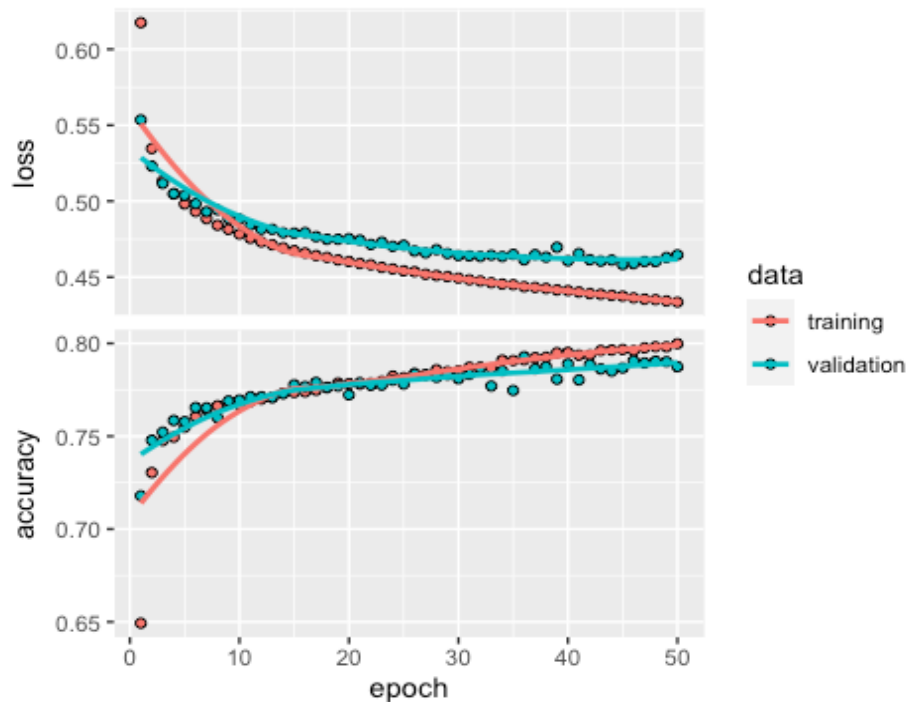
```

metrics = "accuracy")

history = fit(model_3, train_feat, train_labels,
              epochs = 50, batch_size = 250, validation_split = 0.3)

plot(history)

```



```

set.seed(123)
results = model_3 %>%
  evaluate(test_feat, test_labels)

## 340/340 - 0s - loss: 0.4570 - accuracy: 0.7898 - 307ms/epoch - 903us/step

print(results)

##      loss  accuracy
## 0.4569745 0.7898271

```

This also produced a test accuracy of about 79%.

Measure Neural Network Model

Let us use the model that had 50 epochs, 500 batch size and the three activation layers, with the final layer being our output. This is model_2 from above. Here we will measure neural network model and make assumptions on how to increase model performance. Let us use a ROC curve here to measure the model performance. Since our classification is binary, we will use 0.5 as our threshold to convert the probability from our model to predictions. If a prediction is greater than 0.5, it would belong to the canceled reservation classification. If the predictions is less than 0.5,

it would belong to the non-canceled classification. We will calculate the false positive rate (FPR) and true positive rate (TPR) for these predictions. The FPR is the percentage of incorrect predictions we make for observations that belong to the negative class. Thus, we divide the number of observations in `over_threshold` that belong to class 0 by the total number of observations in the test set that belong to class 0.

```
predictions = predict(model_2, test_feat)

## 340/340 - 0s - 330ms/epoch - 972us/step

test_df$p_prob = predictions[, 1]
head(predictions, 10)

##           [,1]
## [1,] 0.67100543
## [2,] 0.12005132
## [3,] 0.13033767
## [4,] 0.42847282
## [5,] 0.78556085
## [6,] 0.05349032
## [7,] 0.90943331
## [8,] 0.64474297
## [9,] 0.03761552
## [10,] 0.31770593

over_threshold = test_df[test_df$p_prob >= 0.5, ]
fpr = sum(over_threshold$booking_status == 0)/sum(test_df$booking_status == 0)
fpr

## [1] 0.1055993
```

Our model produced a false positive rate of approximately 11%. Now, let us find the true positive rate (TPR), which is the percentage of correct predictions we make for observations that belong to the positive class (customers that actually cancel their reservation). Thus, we divide the number of observations in `over_threshold` that belong to class 1 by the total number of observations in the test set that belong to class 1.

```
tpr = sum(over_threshold$booking_status == 1)/sum(test_df$booking_status == 1)
tpr

## [1] 0.5677291
```

This produced a true positive rate of approximately 57%. Let us plot a ROC curve to show the FPR versus the TPR for different thresholds ranging from 0 to 1.

```
roc_data = data.frame(threshold=seq(1, 0, -0.01), fpr = 0, tpr = 0)
for (i in roc_data$threshold) {
  over_threshold = test_df[test_df$p_prob >= i, ]
  fpr = sum(over_threshold$booking_status == 0)/sum(test_df$booking_status == 0)
}
```

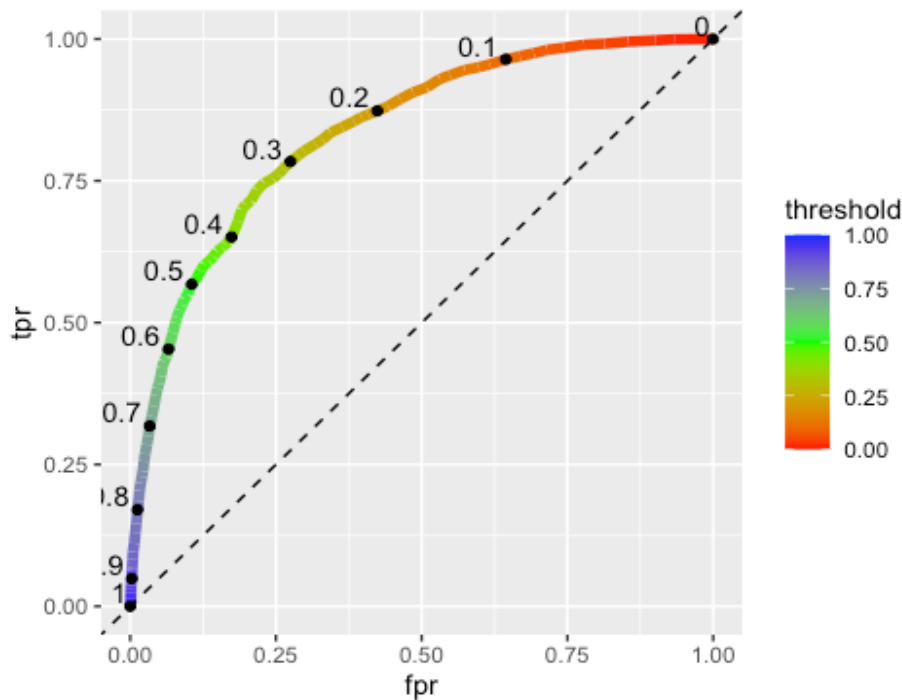


```

roc_data[roc_data$threshold == i, "fpr"] = fpr
tpr = sum(over_threshold$booking_status == 1)/sum(test_df$booking_status ==
1)
roc_data[roc_data$threshold == i, "tpr"] = tpr
}
ggplot() +
  geom_line(data = roc_data, aes(x = fpr, y = tpr, color = threshold), size =
2) +
  scale_color_gradientn(colors = rainbow(3)) +
  geom_abline(intercept = 0, slope = 1, lty = 2) +
  geom_point(data = roc_data[seq(1, 101, 10), ], aes(x = fpr, y = tpr)) +
  geom_text(data = roc_data[seq(1, 101, 10), ],
    aes(x = fpr, y = tpr, label = threshold, hjust = 1.2, vjust = -
0.2))

## Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
## Please use `linewidth` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.

```



To sum up the ROC curve with one number, we calculate the area under the ROC curve (AUC). The closer to 1 that our AUC is, the more accurate the model.

```

auc = auc(x = roc_data$fpr, y = roc_data$tpr, type = "spline")

## Warning in regularize.values(x, y, ties, missing(ties)): collapsing to
unique
## 'x' values

```

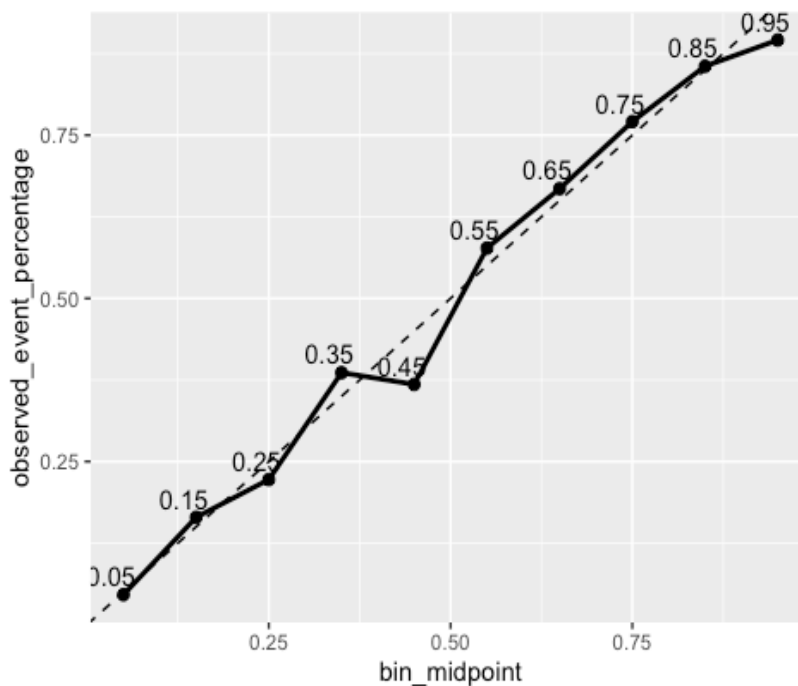
```
auc
```

```
## [1] 0.835557
```

The AUC of our ROC is approximately 83.5.

Now, let us look at the calibration curve. The calibration curve shows the observed event percentage for any set of intervals that partition the interval (0, 1), which is the range of possible predicted probability values.

```
calibration_data = data.frame(bin_midpoint=seq(0.05, 0.95, 0.1),
                              observed_event_percentage = 0)
for (i in seq(0.05, 0.95, 0.1)) {
  in_interval = test_df[test_df$p_prob >= (i-0.05) & test_df$p_prob <=
(i+0.05), ]
  oep = nrow(in_interval[in_interval$booking_status == 1,
])/nrow(in_interval)
  calibration_data[calibration_data$bin_midpoint == i,
"observed_event_percentage"] = oep
}
ggplot(data = calibration_data, aes(x = bin_midpoint, y =
observed_event_percentage)) +
  geom_line(size = 1) +
  geom_abline(intercept = 0, slope = 1, lty = 2) +
  geom_point(size = 2) +
  geom_text(aes(label = bin_midpoint), hjust = 0.75, vjust = -0.5)
```



We want the calibration curve to lie near the dashed diagonal line, which corresponds to a well-calibrated classifier. Values that lie above the dashed diagonal line correspond to under-confident probabilities and values that lie below correspond to over-confident probabilities.

Preliminary Result Conclusion

We fit a neural network model with three activation layers. Our first layer had 40 nodes using the relu (rectified linear unit) activation function. The second layer had 20 nodes using the relu activation function. The third layer used the sigmoid activation function for our binary classification output. Using 20 epochs, a batch size of 500, and a validation split of 30%, this model produced a test label accuracy rate of approximately 79% when identifying if a customer cancelled their reservation or not. This model had a false positive rate (FPR) of approximately 11% and a true positive rate (TPR) of approximately 57%. The ROC curve did hug the upper left side of the plot when plotting FPR versus TPR. The AUC of the model's ROC was approximately 83.5%. This means that our model performed about 33.5% better than a random guessing rate of 50%. Our calibration curve did lie near the dashed diagonal line in the graph above. This corresponds to a well-calibrated classifier from our model.

Improving Model Fit

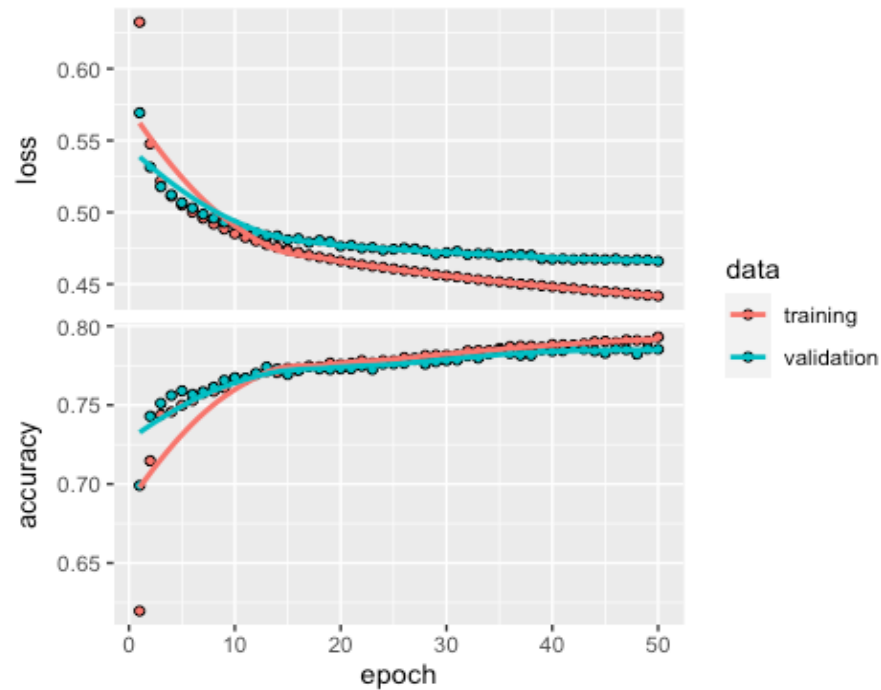
Now, we want to try and achieve a perfect fit. In order to do so, we need to must first overfit our neural network model. We want to cross the overfitting boundary line in order to find it. Once we cross it, we will then implement some steps to to fight the overfitting of the model. We will start by tuning the key gradient descent parameters using large learning rate of 1. We will be using the model that we chose in the preliminary result conclusion section above.

```
# Model we selected from preliminary results: Model 2
model_2 = keras_model_sequential(list(
    layer_dense(units = 40, activation = "relu"),
    layer_dense(units = 20, activation = "relu"),
    layer_dense(units = 1, activation = "sigmoid")
))

compile(model_2,
        optimizer = "rmsprop",
        loss = "binary_crossentropy",
        metrics = "accuracy")

history = fit(model_2, train_feat, train_labels,
              epochs = 50, batch_size = 500, validation_split = 0.3)

plot(history)
```



```
set.seed(123)
results = model_2 %>%
  evaluate(test_feat, test_labels)

## 340/340 - 0s - loss: 0.4566 - accuracy: 0.7909 - 275ms/epoch - 808us/step

print(results)

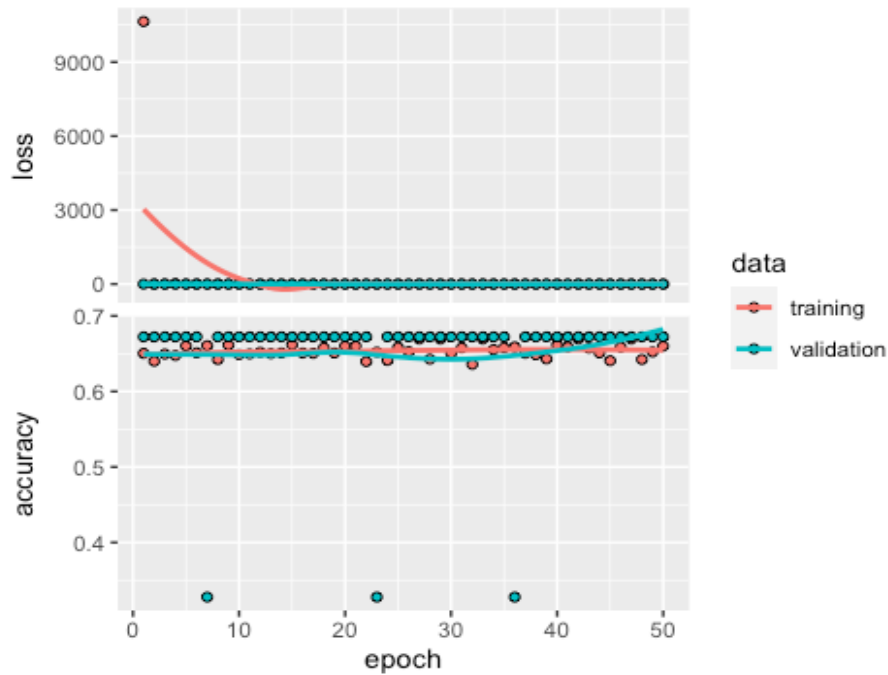
##      loss  accuracy
## 0.4566251 0.7909308

# Now let us use an inappropriately large Learning rate of 1. We will also
# add in 2 more Layers.
model_ofit = keras_model_sequential(list(
  layer_dense(units = 25, activation = "relu"),
  layer_dense(units = 20, activation = "relu"),
  layer_dense(units = 15, activation = "relu"),
  layer_dense(units = 10, activation = "relu"),
  layer_dense(units = 5, activation = "relu"),
  layer_dense(units = 1, activation = "sigmoid")
))

compile(model_ofit,
  optimizer = optimizer_rmsprop(1),
  loss = "binary_crossentropy",
  metrics = "accuracy")

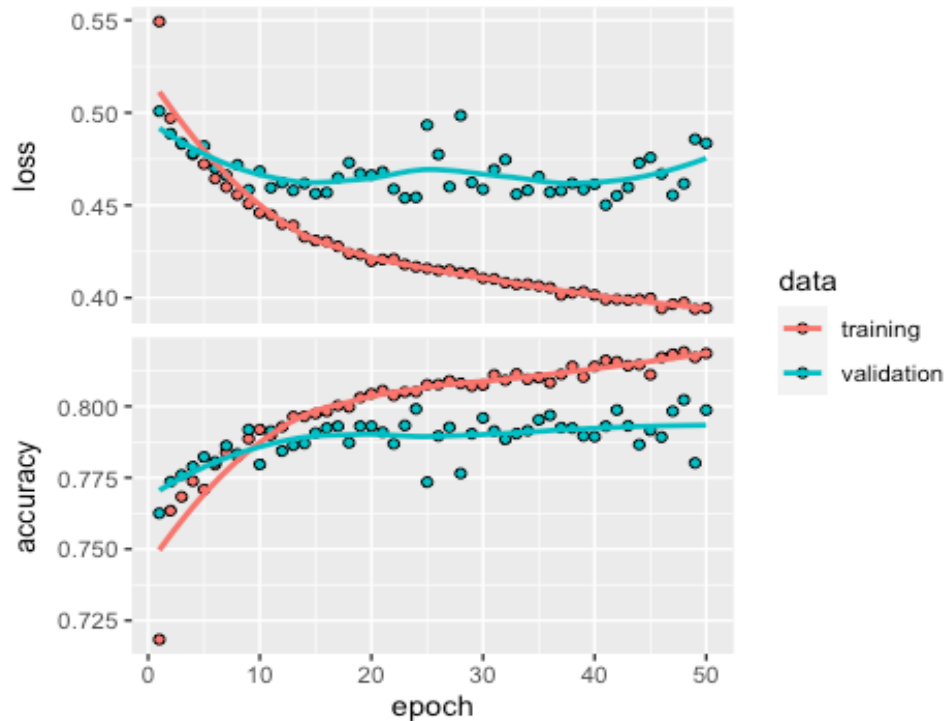
history = fit(model_ofit, train_feat, train_labels,
  epochs = 50, batch_size = 500, validation_split = 0.3)
```

```
plot(history)
```



Now, let us try to lower the training rate to a more reasonable value of $1e-2$.

```
model_ofit = keras_model_sequential(list(  
    layer_dense(units = 25, activation = "relu"),  
    layer_dense(units = 20, activation = "relu"),  
    layer_dense(units = 15, activation = "relu"),  
    layer_dense(units = 10, activation = "relu"),  
    layer_dense(units = 5, activation = "relu"),  
    layer_dense(units = 1, activation = "sigmoid")  
))  
  
compile(model_ofit,  
        optimizer = optimizer_rmsprop(1e-2),  
        loss = "binary_crossentropy",  
        metrics = "accuracy")  
  
history = fit(model_ofit, train_feat, train_labels,  
              epochs = 50, batch_size = 500, validation_split = 0.3)  
  
plot(history)
```



Now, let us try to increase the batch size to 1000.

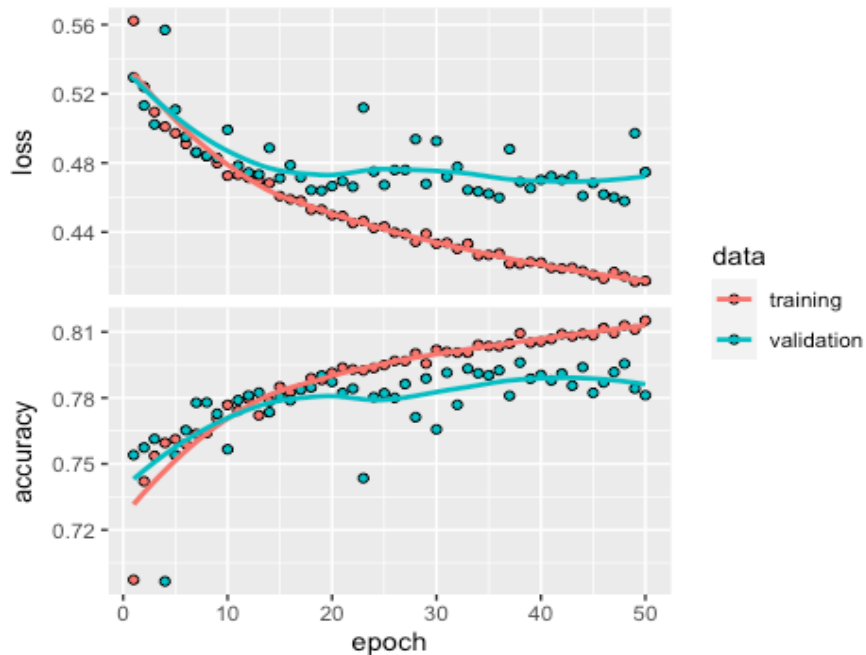
Now, let us try to increase the batch size to 1000

```
model_ofit = keras_model_sequential(list(
    layer_dense(units = 25, activation = "relu"),
    layer_dense(units = 20, activation = "relu"),
    layer_dense(units = 15, activation = "relu"),
    layer_dense(units = 10, activation = "relu"),
    layer_dense(units = 5, activation = "relu"),
    layer_dense(units = 1, activation = "sigmoid")
))

compile(model_ofit,
        optimizer = optimizer_rmsprop(1e-2),
        loss = "binary_crossentropy",
        metrics = "accuracy")

history = fit(model_ofit, train_feat, train_labels,
              epochs = 50, batch_size = 1000, validation_split = 0.3)

plot(history)
```



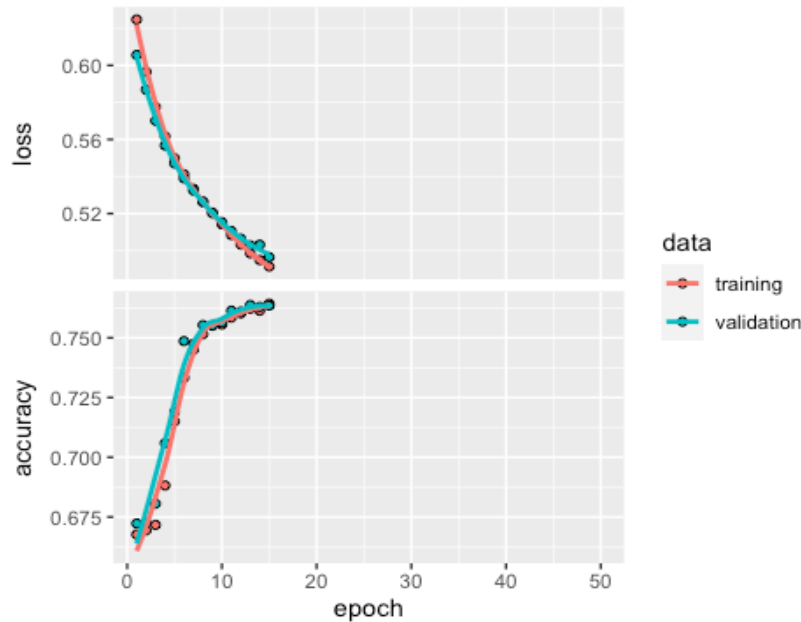
Now, let us use the `callback_early_stopping()` function to stop the training when you measure that the validation loss is no longer improving.

```
set.seed(123)
callback = callback_early_stopping(monitor = "val_accuracy", patience = 2)
model_ofit = keras_model_sequential(list(
  layer_dense(units = 25, activation = "relu"),
  layer_dense(units = 20, activation = "relu"),
  layer_dense(units = 15, activation = "relu"),
  layer_dense(units = 10, activation = "relu"),
  layer_dense(units = 5, activation = "relu"),
  layer_dense(units = 1, activation = "sigmoid")
))
```

```
set.seed(123)
compile(model_ofit,
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = "accuracy")
```

```
set.seed(123)
history = fit(model_ofit, train_feat, train_labels,
  epochs = 50, batch_size = 1000,
  validation_split = 0.3, callbacks = callback)
```

```
plot(history)
```



```
set.seed(123)
results = model_ofit %>%
  evaluate(test_feat, test_labels)

## 340/340 - 0s - loss: 0.4896 - accuracy: 0.7660 - 385ms/epoch - 1ms/step

print(results)

##      loss  accuracy
## 0.4896206 0.7660044
```

We see that this model's validation accuracy did not improve after the 15th epoch. This means that overfitting began to occur at the 15th epoch. Using the neural network model with 15 epochs and batch size of 1000, our model produced a test accuracy of approximately 77% on our test features. This is slightly less than the 79% test label accuracy from model_2 that we selected earlier.

Measure Overfitted Neural Network Model

Now let us look at at ROC, AUC and calibration curve for this model.

```
predictions = predict(model_ofit, test_feat)

## 340/340 - 0s - 409ms/epoch - 1ms/step

test_df$p_prob = predictions[, 1]

over_threshold = test_df[test_df$p_prob >= 0.5, ]
fpr = sum(over_threshold$booking_status == 0)/sum(test_df$booking_status == 0)
fpr
```

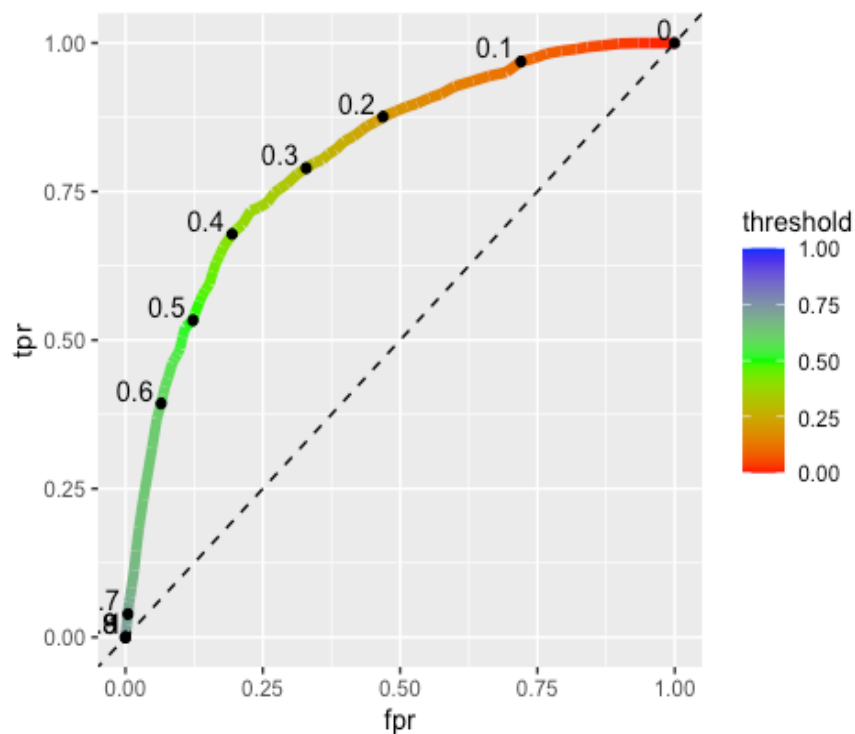


```
## [1] 0.1228595

tpr = sum(over_threshold$booking_status == 1)/sum(test_df$booking_status == 1)
tpr

## [1] 0.5332954

roc_data = data.frame(threshold=seq(1, 0, -0.01), fpr = 0, tpr = 0)
for (i in roc_data$threshold) {
  over_threshold = test_df[test_df$p_prob >= i, ]
  fpr = sum(over_threshold$booking_status == 0)/sum(test_df$booking_status == 0)
  roc_data[roc_data$threshold == i, "fpr"] = fpr
  tpr = sum(over_threshold$booking_status == 1)/sum(test_df$booking_status == 1)
  roc_data[roc_data$threshold == i, "tpr"] = tpr
}
ggplot() +
  geom_line(data = roc_data, aes(x = fpr, y = tpr, color = threshold), size = 2) +
  scale_color_gradientn(colors = rainbow(3)) +
  geom_abline(intercept = 0, slope = 1, lty = 2) +
  geom_point(data = roc_data[seq(1, 101, 10), ], aes(x = fpr, y = tpr)) +
  geom_text(data = roc_data[seq(1, 101, 10), ],
            aes(x = fpr, y = tpr, label = threshold, hjust = 1.2, vjust = -0.2))
```



The overfitted model produced a false positive rate (FPR) of approximately 12% and a true positive rate (TPR) of approximately 53%.

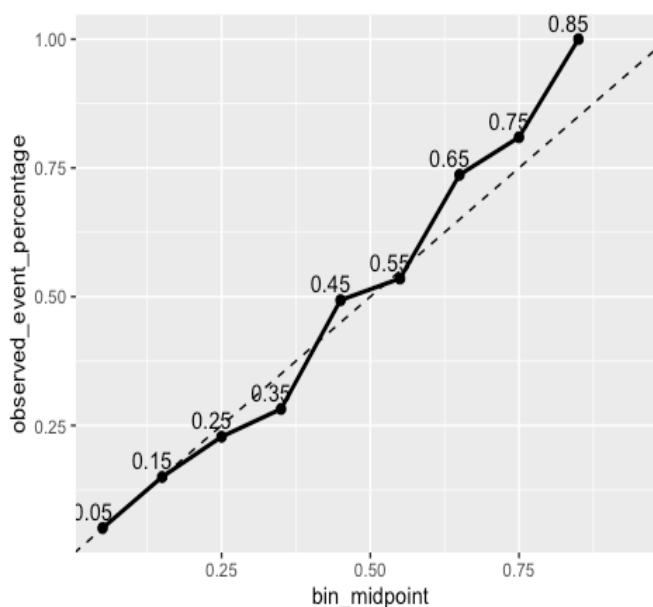
AUC:

```
auc = auc(x = roc_data$fpr, y = roc_data$tpr, type = "spline")
auc
## [1] 0.8105066
# 0.81
```

The AUC value from this model was 0.81.

Calibration Curve:

```
calibration_data = data.frame(bin_midpoint=seq(0.05, 0.95, 0.1),
                              observed_event_percentage = 0)
for (i in seq(0.05, 0.95, 0.1)) {
  in_interval = test_df[test_df$p_prob >= (i-0.05) & test_df$p_prob <=
(i+0.05), ]
  oep = nrow(in_interval[in_interval$booking_status == 1,
])/nrow(in_interval)
  calibration_data[calibration_data$bin_midpoint == i,
"observed_event_percentage"] = oep
}
ggplot(data = calibration_data, aes(x = bin_midpoint, y =
observed_event_percentage)) +
  geom_line(size = 1) +
  geom_abline(intercept = 0, slope = 1, lty = 2) +
  geom_point(size = 2) +
  geom_text(aes(label = bin_midpoint), hjust = 0.75, vjust = -0.5)
```



The ROC did hug the upper left side of the plot, but not as much as model_2's ROC did. The AUC value is 81.1, which is lower than the AUC value of 83.5 from model_2. In the calibration curve above, we see that each midpoint did hug the diagonal line, but not as well as model_2's calibration curve did. When compared to model_2, this overfitted model does not correspond to a well-calibrated classifier. Based on that information, let us move on to our conclusion and any recommendations that we can make based on what we learn.

Final Conclusion/Recommendations

In our preliminary results section, we fit a neural network model that included three activation layers. This neural network model produced a test label accuracy of 79% when identifying if a customer cancelled their reservation or not.

In our overfitting model section above, we produced an overfitted model that produced a lower test label accuracy rate of 77%.

The ROC curve of our overfit model did hug the upper left side of the plot when plotting FPR versus TPR. However, the AUC of the overfit model was 81.1 compared to the 83.5 from our preliminary result conclusion.

Our calibration curve of our overfitted model did lie near the dashed diagonal line, but not as well as the calibration curve from model_2.

What that being said, we will choose model_2. This was selected in our preliminary results conclusion to help ABC Hotel predict whether a customer will cancel their reservation. This model had a better test label prediction rate, had a larger AUC value and had a calibration curve with each FPR midpoint value along the diagonal line. Therefore, we want to use this model over the one that we intentionally overfit.

Ultimately, this model performed better than the overfit model that we built. This model best represented our data; therefore, we will recommend this model to ABC Hotel. This will hopefully allow them to target customers who are most likely to cancel to help them minimize their reservation cancellations.

As time goes on, ABC Hotel will get more and more data about customers. Whether that is more customers themselves, or more features that they may want to track and record. With that being said, customers are forever changing, meaning that our model may not properly evolve with those evolving customers. Based on our analysis, the model that we selected best represents the customer data set that we were given at this time and should be used as foundation for future analyses. Using this framework should allow future data scientists to tweak and manipulate the neural network model to better help ABC Hotel predict whether customers will cancel their reservation or not.