Thomas Dinh

DSE 6211

Final Results

October 20, 2023

Final Results: Leveraging Neural Networks for classification on hotel cancellation

Executive summary:

Data Preparation and Feature Engineering

- Partitioned the data into training and test sets.
- Changed the booking status for both training and test sets to 0 and 1.
- Consolidated infrequent meal plan categories ("meal_plan_2," "meal_plan_3," and "not_selected") into a single "other" category to enhance model stability and generalization.
- Applied one-hot encoding to these columns to optimize the model's ability to process categorical data.
- Excluded non-numeric columns from the dataset as neural networks require numeric input data.
- Performed standardization to address scale differences in the dataset and ensure fair model evaluation.
- Applied PCA for feature engineering and used 7 PCA units, which account for the majority of the variance.

Model Selection and Evaluation

Three neural network models were created and compared. All models utilized ReLU activation functions for non-linearity and "rmsprop" optimization for training. The "binary_crossentropy" loss function was chosen for binary classification.

Model 1 had three layers (100 units, 50 units, and 1 unit). Model 2 had four layers (100 units, 50 units, 25 units, and 1 unit). Model 3 had four layers (25 units, 10 units, 10 units, and 1 unit) and used L1 regularization with a lambda of 0.002 to introduce sparsity into the model.

Model 1 exhibited overfitting after approximately 40 epochs. Model 1 achieved an AUC score of 0.8975, indicating effectiveness.

Model 2 displayed unusual loss and accuracy patterns, possibly due to additional layers. Model 2 achieved an AUC score of 0.89135, also indicating effectiveness.

Model 3 showed signs of both underfitting and overfitting, but the loss and accuracy curves suggest that the model was stopped at the right time to prevent complete overfitting. Model 3 achieved an AUC score of 0.8917, which is comparable to the other two models.

Calibration Plots

All three models' calibration curves were overconfident, suggesting potential issues in probability calibration.

Comparison of Models

	Model 1	**Model 2**	**Model 3**
AUC	0.8975	0.89135	0.8917
Loss	0.3579	0.3849	0.4601
Accuracy	0.8421	0.8234	0.8319
Calibration	Underconfident, away from the line	Underconfident, away from the line	50/50 but close to line
layers	3	4	4
L1 regularization	No	No	Yes

Conclusion

All three models achieved comparable AUC scores, signifying their effectiveness in predicting room cancellations. However, Model 3 holds the advantage of being less complex than the other two models, featuring fewer units and implementing L1 regularization to introduce sparsity. Despite Model 1's better loss and accuracy in comparison to Model 3, Model 3's calibration curve outperforms the other two models, implying its superior ability to calibrate probability predictions.

Final Model and Deployment

I would recommend using Model 3 as the final model. It is a simple and effective model with good calibration performance.

Here is a possible way to deploy the model:

1. Save the trained model to a file.
2. Develop a web service or API that exposes the model.
3. Integrate the web service or API into the hotel's booking system.

This would allow the hotel to use the model to generate predictions for room cancellations in real time. The predictions could be used to make better decisions about room pricing, staffing, and marketing.

Data Processing:

Partitioned the data to create the training and test set.

```r
#Loading Data set
data <- read.csv("project_data.csv")


#Getting the Indexes of 75% of the data
training_ind <- createDataPartition(data$booking_status,
                p = 0.75,
                list = FALSE,
                times = 1)

#Training set data
training_set <- data[training_ind, ]

#Test set data
test_set <- data[-training_ind, ]
```

To account for the impact of seasons on room cancellations, we categorized all dates into seasons. I believe that the seasons play a role in deciding whether or not someone cancels a room reservation. Travel patterns, weather, and holidays can all affect the demand for rooms. To mitigate this, try creating a seasonal pricing model, offering seasonal discounts and promotions, and monitoring your cancellation rates by season.

```r
#Changing to date
training_set$'arrival_date' <- ymd(training_set$'arrival_date')

month <- month(training_set$arrival_date)

# Assign the season to the new column
training_set$season[month %in% c(1, 2, 12)] <- "Winter"
training_set$season[month %in% c(3, 4)] <- "Spring"
training_set$season[month %in% c(5, 6, 7, 8)] <- "Summer"
training_set$season[month %in% c(9, 10, 11)] <- "Fall"

#Changing to date
test_set$'arrival_date' <- ymd(test_set$'arrival_date')

month <- month(test_set$arrival_date)

# Assign the season to the new column
test_set$season[month %in% c(1, 2, 12)] <- "Winter"
test_set$season[month %in% c(3, 4)] <- "Spring"
test_set$season[month %in% c(5, 6, 7, 8)] <- "Summer"
test_set$season[month %in% c(9, 10, 11)] <- "Fall"
```

Changing booking status for both test and training sets to 0 and 1.

```r
#replacing Booking status to 0 and 1
training_set$booking_status <- ifelse(training_set$booking_status == 'canceled', 0, 1)
test_set$booking_status <- ifelse(test_set$booking_status == 'canceled', 0, 1)
```

In the code below, we are consolidating various elements into a unified 'other group.' Specifically, we are grouping together 'meal_plan_2,' 'meal_plan_3,' and 'not_selected.' Given the dataset's size of approximately 36,000 data points, each of these individual meal plans contains fewer than 1,000 occurrences.

The reason for this consolidation is to enhance the stability and generalization of our predictive model. Rare categories, as observed in these infrequent meal plans, can potentially introduce model instability or hinder its ability to generalize effectively. This is due to the limited amount of information available for these rare categories. By combining them into a single 'other' category, we are addressing this challenge and allowing the model to make more reliable predictions by focusing on the predominant categories with richer data.

```r
#checking type of meal plan
result1 <- training_set %>%
  group_by(training_set$type_of_meal_plan) %>%
  summarize(count = n())

# View the result
print(result1)

#replace meal plan 3 and not selected to other

training_set$type_of_meal_plan <- ifelse(training_set$type_of_meal_plan %in% c('meal_plan_2','meal_plan_3', 'not_selected'), 'other_meal_choice',
training_set$type_of_meal_plan)

#Checking count room type reserved
result2 <- training_set %>%
  group_by(training_set$room_type_reserved) %>%
  summarize(count = n())

# View the result
print(result2)

#replace roomtype2,3,5,6,7 with other room
training_set$room_type_reserved <- ifelse(training_set$room_type_reserved %in% c('room_type2',
'room_type3','room_type4','room_type5','room_type6','room_type7'), 'other_room', training_set$room_type_reserved)
```

```r
#checking count market segment type
result3 <- training_set %>%
  group_by(training_set$market_segment_type) %>%
  summarize(count = n())

# View the result
print(result3)

#replace market segment: aviation, comp to other_market
training_set$market_segment_type <- ifelse(training_set$market_segment_type %in% c('aviation', 'complementary', 'corporate'), 'other_market',
training_set$market_segment_type)
```

Following this step, we proceed to perform one-hot encoding on these four columns. One of the notable advantages of one-hot encoding is its ability to facilitate the learning process of the model with greater ease and efficiency. This encoding scheme optimizes the model's ability to process and interpret the data.

```r
# Factored the training and test data
training_set$type_of_meal_plan <- factor(training_set$type_of_meal_plan)
training_set$room_type_reserved <- factor(training_set$room_type_reserved)
training_set$market_segment_type <- factor(training_set$market_segment_type)
training_set$season <- factor(training_set$season)

test_set$type_of_meal_plan <- factor(test_set$type_of_meal_plan)
test_set$room_type_reserved <- factor(test_set$room_type_reserved)
test_set$market_segment_type <- factor(test_set$market_segment_type)
test_set$season <- factor(test_set$season)

#One hot encode training data with type of meal plan, room type reserved and market segment type
onehot_encoder <- dummyVars(~ type_of_meal_plan + room_type_reserved + market_segment_type + season,
                  training_set[, c("type_of_meal_plan", "room_type_reserved", "market_segment_type", "season")],
                  levelsOnly = TRUE,
                  fullRank = TRUE)

#Predict the one hot encode data
onehot_enc_training <- predict(onehot_encoder,
                     training_set[, c("type_of_meal_plan", "room_type_reserved", "market_segment_type", "season")])

#combine the training set and one hot encoded training set
training_set <- cbind(training_set, onehot_enc_training)


#Predict the one hot encode test data
onehot_enc_test <- predict(onehot_encoder,
                       test_set[, c("type_of_meal_plan", "room_type_reserved", "market_segment_type", "season")])

#combine the training set and one hot encoded test set
test_set <- cbind(test_set, onehot_enc_test)
```

I excluded all non-numeric columns from the dataset because neural networks require numeric input data. To ensure compatibility with the neural network, any non-numeric variables, such as text or categorical data, should be appropriately transformed or encoded into numeric representations during preprocessing.

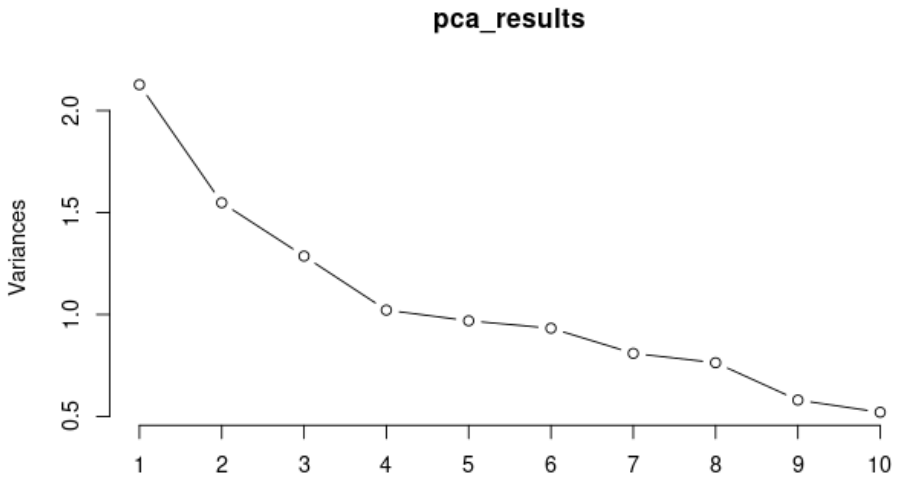
```r
#taking out all non numeric columns from both the training and test data
numeric_columns <- sapply(training_set, is.numeric)
training_set <- training_set[, numeric_columns]


numeric_columns <- sapply(test_set, is.numeric)
test_set <- test_set[, numeric_columns]
```

I standardized my data because the columns in my dataset exhibited significant differences in scale. Such discrepancies in scale could potentially result in certain columns being given greater importance by the model, leading to a more complex model. To address this, I applied standardization, which involves

scaling each feature to have a mean of zero and a standard deviation of one. Additionally, I ensured consistency by using the mean and standard deviation calculated from the training data to standardize the test data, promoting fair and accurate model evaluation.

```r
#scaling both the training and test data
mean <- apply(training_set, 2, mean)
sd <- apply(training_set, 2, sd)
scaled_training_set_features <- scale(training_set, center = mean, scale = sd)
scaled_test_set_features <- scale(test_set, center = mean, scale = sd)



training_features <- array(data = unlist(scaled_training_set_features),
                           dim = c(nrow(scaled_training_set_features), ncol(scaled_training_set_features)))


test_features <- array(data = unlist(scaled_test_set_features),
                       dim = c(nrow(scaled_test_set_features), ncol(scaled_test_set_features)))
```

I applied PCA for feature engineering and used 7 PCA units, which account for the majority of the variance as shown in the graph below. This may improve performance by reducing noise in the data.




```r
pca_results <- prcomp(training_features[, c(1:11)])
summary(pca_results)

screeplot(pca_results, type = "line")

training_rotated <- as.matrix(training_features[, 1:11]) %*% pca_results$rotation

training_features <- cbind(training_features, training_rotated[, 1:6])

test_rotated <- as.matrix(test_features[, 1:11]) %*% pca_results$rotation
test_features <- cbind(test_features, test_rotated[, 1:6])
```

Model Selection:

Previous Model 1 and 2

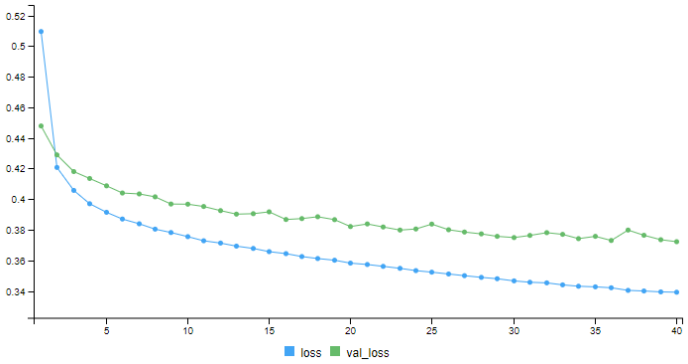
I created 2 models to see which one had better results. What is the same in both of them are the ReLU activation functions, which help introduce non-linearity and are commonly used in hidden layers of neural networks. The sigmoid activation in the output layer is appropriate for binary classification, producing class probabilities. For optimization, the "rmsprop" algorithm was selected, a well-established optimizer. The "binary_crossentropy" loss function was chosen as it's suited for binary classification tasks. These choices aim to achieve good model performance while addressing the specific requirements of the binary classification problem at hand.

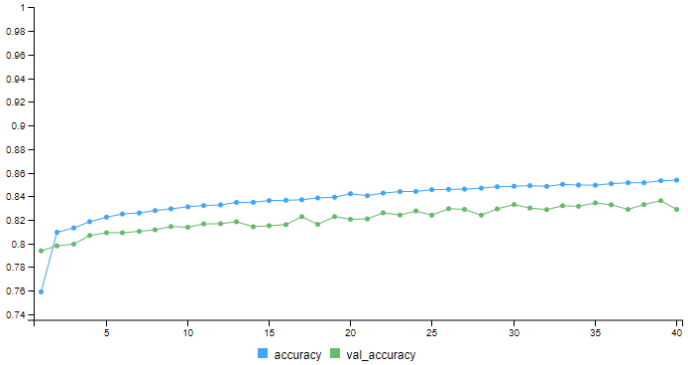
The first model has neural network in this code consists of three layers: a dense layer with 100 units and ReLU activation, a dense layer with 50 units and ReLU activation, and a final dense layer with 1 unit and a sigmoid activation function. The data is fit to 40 epochs, 512 batch-size, and a validation split at 0.2.

```
use_virtualenv("my_tf_workspace")

model <- keras_model_sequential(list(
              layer_dense(units = 75, activation = "relu"),
              layer_dense(units = 50, activation = "relu"),
              layer_dense(units = 1, activation = "sigmoid")
))

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

history <- fit(model, training_features, training_labels,
              epochs = 40, batch_size = 512, validation_split = 0.2)

results <- model %>%
    evaluate(test_features,test_label,verbose = 0)

results
```

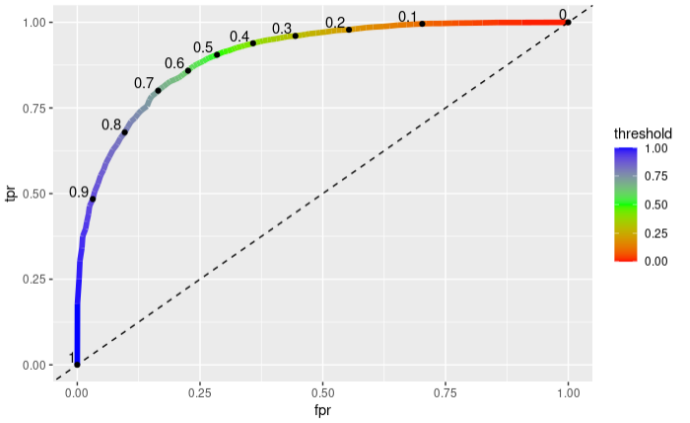
```
> results
      loss  accuracy
0.3579342 0.8421459
```

We see in the loss and accuracy graph after 6 epochs the graphs start to overfit the data.





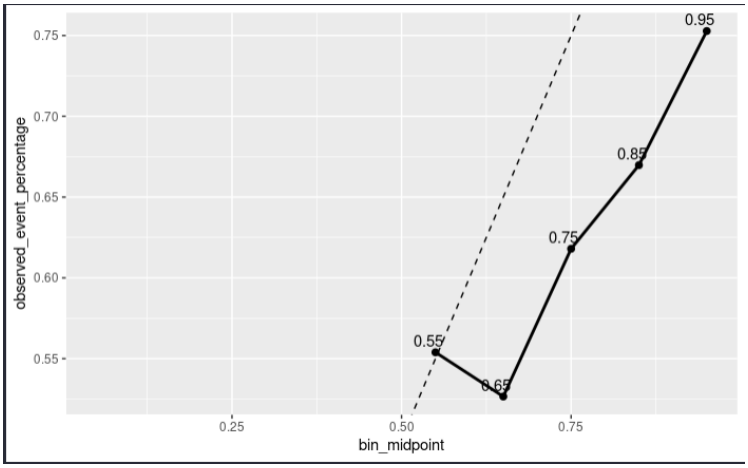
AUC has a score of 0.8975, which indicates that the model is quite effective. It suggests that the model's predicted probabilities or scores for the positive class tend to be higher than those for the negative class, and it can make reasonably accurate predictions.




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

```
[1] 0.8975442
```

The calibration curve is overconfident on the whole curve.



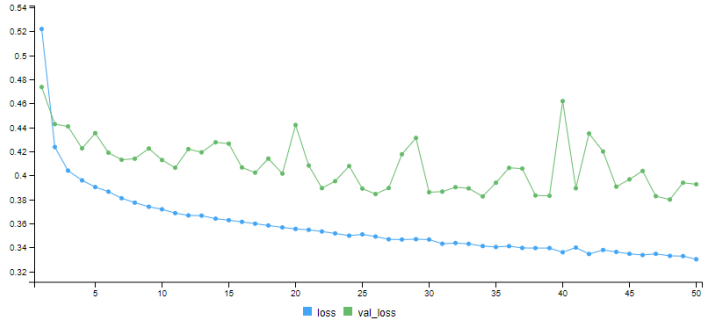
The second model has neural network in this code consists of four layers: a dense layer with 100 units and ReLU activation, a dense layer with 50 units and ReLU activation, a dense layer with 25 units and ReLU activation, and a final dense layer with 1 unit and a sigmoid activation function. The data is fit to 50

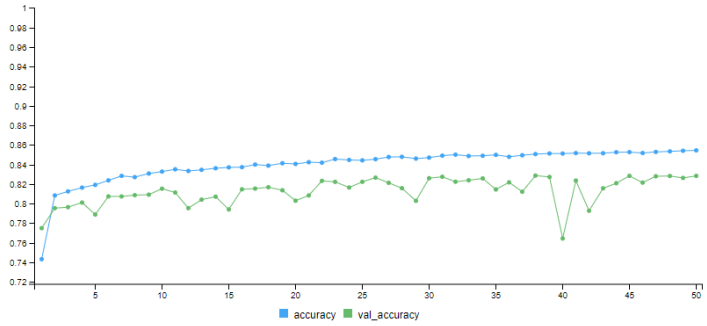
epochs, 1000 batch-size, and a validation split at 0.3.

```
use_virtualenv("my_tf_workspace")

model <- keras_model_sequential(list(
              layer_dense(units = 100, activation = "relu"),
              layer_dense(units = 50, activation = "relu"),
              layer_dense(units = 25, activation = "relu"),
              layer_dense(units = 1, activation = "sigmoid")
))

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

history <- fit(model, training_features, training_labels,
               epochs = 50, batch_size = 1000, validation_split = 0.30)

results <- model %>%
    evaluate(test_features,test_label,verbose = 0)

results
```

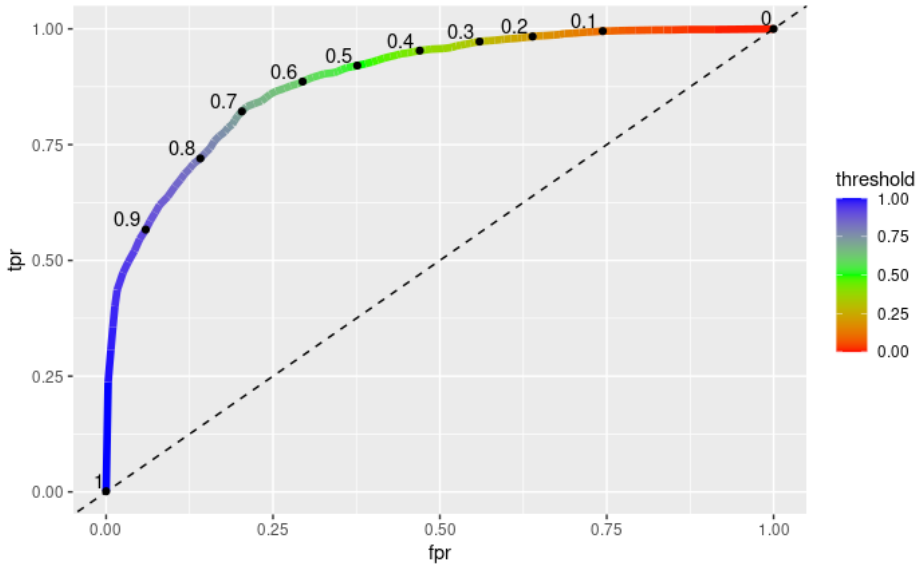
```
      loss  accuracy
0.3849499 0.8234904
```

We observe that the loss and accuracy exhibit some unusual patterns. This could potentially be attributed to the inclusion of an additional layer or the fine-tuning of model parameters.





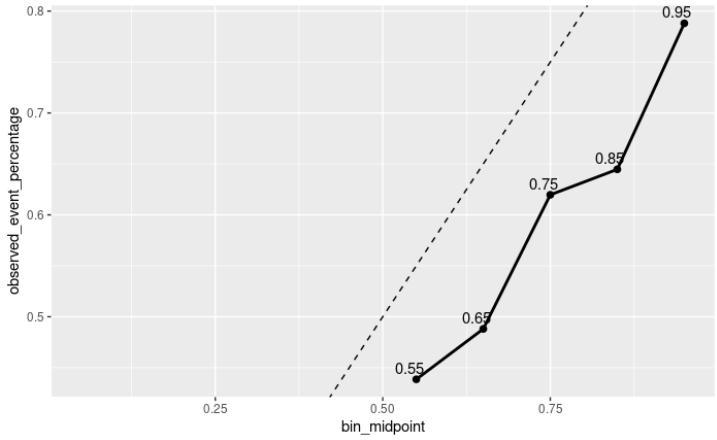
AUC has a score of 0.89135, which indicates that the model is quite effective




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

```
[1] 0.8913571
```

We see this calibration curve is close to the line but is overconfident also.



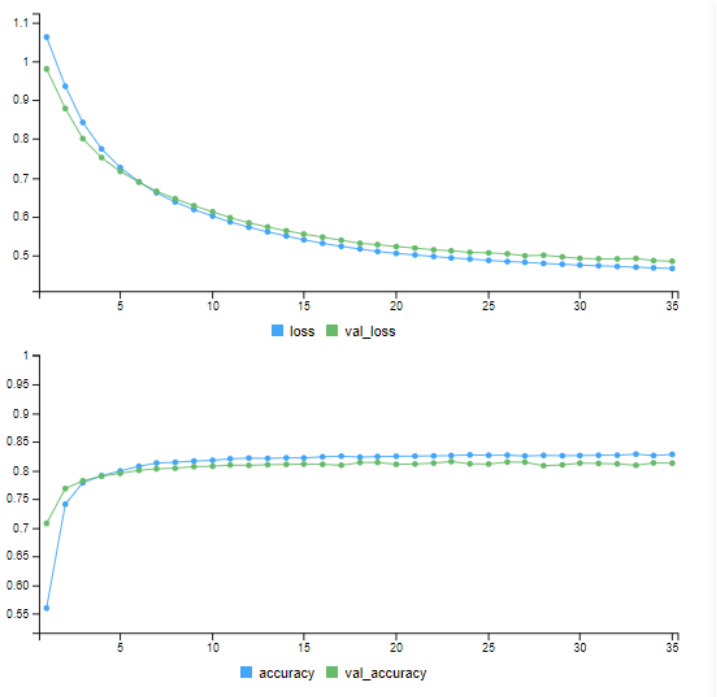
Model 3:

The third model has neural network in this code consists of four layers: a dense layer with 25 units and ReLU activation, a dense layer with 10 units and ReLU activation, a dense layer with 10 units and ReLU activation, and a final dense layer with 1 unit and a sigmoid activation function. The data is fit to 35 epochs, 750 batch-size, and a validation split at 0.3. We see also the loss is 0.46012, while accuracy is 0.83199. I also used l1 regulaziation with a lambda of 0.002 to introduce sparsity into a model.

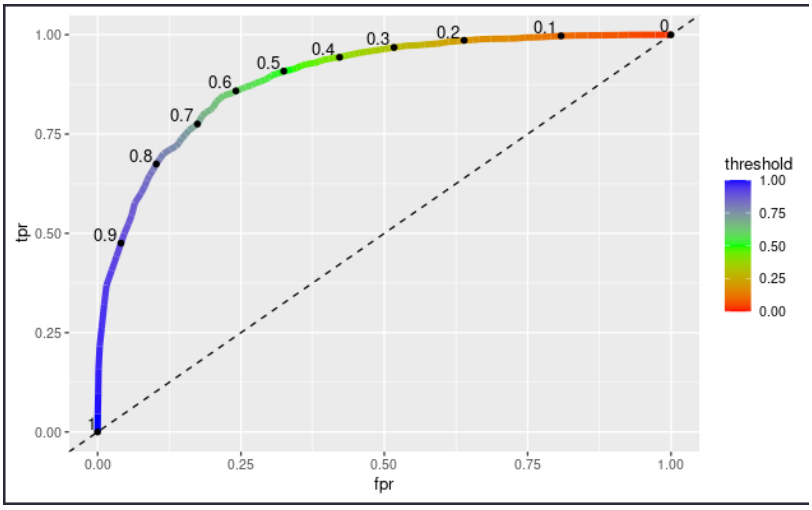
```
use_virtualenv("my_tf_workspace")

lambda1 = 0.002

model <- keras_model_sequential(list(
              layer_dense(units = 25, activation = "relu",
                          kernel_regularizer  = regularizer_l1(l=lambda1)),
              layer_dense(units = 10, activation = "relu",
                          kernel_regularizer = regularizer_l1(l=lambda1)),
              layer_dense(units = 10, activation = "relu",
                          kernel_regularizer = regularizer_l1(l=lambda1)),
              layer_dense(units = 1, activation = "sigmoid")
))

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

history <- fit(model, training_features, training_labels,
              epochs = 35, batch_size = 750, validation_split = 0.30)

results <- model %>%
    evaluate(test_features,test_label,verbose = 0)

results
```

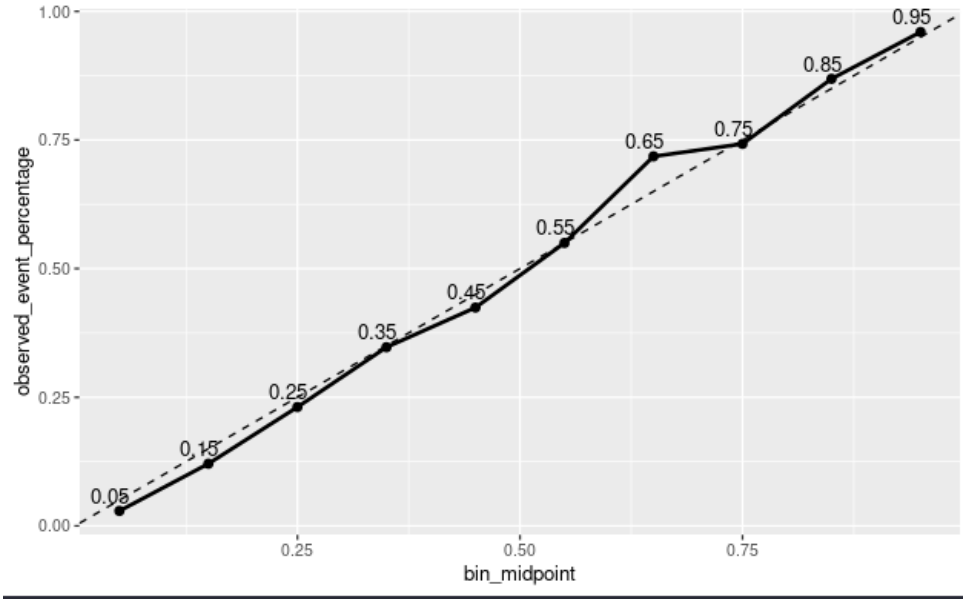
```
     loss  accuracy
0.4601279 0.8319903
```

Our loss and accuracy curves also show underfitting and overfitting, while we stopped at 35 epochs to prevent the model from completely overfitting.



The AUC curve for model 3 is 89.17%, and our calibration curve is nearly identical to the dotted line. This suggests that the model is underconfident for predictions below 0.55 and slightly overconfident for predictions above 0.55.


1.00
0.75
0.50
0.25
0.00
tpr
0.00
0.25
0.50
0.75
1.00
fpr
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
threshold
1.00
0.75
0.50
0.25
0.00


1.00
0.75
0.50
0.25
0.00
observed_event_percentage
0.25
0.50
0.75
bin_midpoint
0.05
0.15
0.25
0.35
0.45
0.55
0.65
0.75
0.85
0.95