Overview

This assignment is focused on binary classification performance metrics. You will calculate the Accuracy, the confusion matrix, and a simple ROC curve manually on a simple example. Afterwards, you will assess the performance of multiple binary classifiers using `caret` to manage the resampling process and the `yardstick` package to calculate several performance metrics in a more realistic application. You will use Accuracy, the ROC curve, and the ROC AUC to compare multiple models and select the best one. Lastly, you will compare the models using the calibration curve.

IMPORTANT: The RMarkdown assumes you have downloaded the 3 data sets (CSV files) to the same directory you saved the template Rmarkdown file. If you do not have the 3 CSV files in the correct location, the data will not be loaded correctly.

IMPORTANT!!!

Certain code chunks are created for you. Each code chunk has `eval=FALSE` set in the chunk options. You **MUST** change it to be `eval=TRUE` in order for the code chunks to be evaluated when rendering the document.

You are free to add more code chunks if you would like.

Load packages

The tidyverse is loaded for you in the code chunk below.

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.2     v readr     2.1.4
## v forcats   1.0.0     v stringr   1.5.0
## v ggplot2   3.4.3     v tibble    3.2.1
## v lubridate 1.9.2     v tidyr     1.3.0
## v purrr     1.0.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

The other packages, `caret` and `yardstick`, will be loaded as needed later in the assignment.

Problem 01

The code chunk below reads in a data set that you will work with in Problems 1, 2 and 3. A glimpse is printed for you which shows three variables, an input `x`, a model predicted event probability, `pred_prob`, and the observed output class, `obs_class`. A binary classifier has already been trained for you. That model's predicted probability (`pred_prob`) is provided with the observed binary outcome (`obs_class`). You will use this data set to get experience with binary classification performance metrics.

```
example_data_path <- "hw03_example_binary_data.csv"

model_pred_df <- readr::read_csv(example_data_path, col_names = TRUE)
```

```
## Rows: 125 Columns: 3
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (1): obs_class
## dbl (2): x, pred_prob
##
## i Use `spec()` to retrieve the full column specification for this data.
```

```
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
model_pred_df %>% glimpse()
```

```
## Rows: 125
## Columns: 3
## $ x         <dbl> -0.4574291, 0.4259475, -0.7846951, -1.9252092, 2.2526171, 1.~
## $ pred_prob <dbl> 0.4174307, 0.2236508, 0.5987937, 0.9903552, 0.8607660, 0.491~
## $ obs_class <chr> "non_event", "non_event", "event", "event", "event", "non_ev~
```

1a)

Pipe the `model_pred_df` data set into the `count()` function to display the number of unique values of the `obs_class` variable.

```
model_pred_df %>% count(obs_class)
```

SOLUTION

```
## # A tibble: 2 x 2
##   obs_class     n
##   <chr>     <int>
## 1 event        66
## 2 non_event    59
```

1b)

You should see that one of the values of `obs_class` is the event of interest and is named `"event"`.

Use the `mean()` function to determine the fraction (or proportion) of the observations that correspond to the event of interest. Is the data set a balanced data set?

```
mean(model_pred_df$obs_class == "event")
```

SOLUTION

```
## [1] 0.528
```

Is the data set balanced?

Yes, the data set is balanced because almost half of the observations is "event".

1c)

In lecture we discussed that regardless of the labels or classes associated with the binary response, we can encode the outcome as `y = 1` if the `"event"` is observed and `y = 0` if the `"non_event"` is observed. You will encode the output with this 0/1 encoding.

The `ifelse()` function can help you perform this operation. The `ifelse()` function is a one-line if-statement which operates similar to the IF function in Excel. The basic syntax is:

```
ifelse(<conditional statement to check>, <value if TRUE>, <value if FALSE>)
```

Thus, the user must specify a condition to check as the first argument to the `ifelse()` function. The second argument is the value to return if the conditional statement is TRUE, and the second argument is the value to return if the conditional statement is FALSE.

You can use the `ifelse()` statement within a `mutate()` call to create a new column in the `model_pred_df` data set.

The code chunk below provides an example using the first 10 rows from the `iris` data set which is loaded with base R. The `Sepal.Width` variable is compared to a value of 3.5. If `Sepal.Width` is greater than 3.5 the new variable, `width_factor`, is set equal to `"greater than"`. However, if it is less than 3.5 the new variable is set to `"less than"`.

```
iris %>%
  slice(1:10) %>%
  select(starts_with("Sepal"), Species) %>%
  mutate(width_factor = ifelse(Sepal.Width > 3.5,
                               "greater than",
                               "less than"))
```

```
##    Sepal.Length Sepal.Width Species width_factor
## 1           5.1         3.5  setosa    less than
## 2           4.9         3.0  setosa    less than
## 3           4.7         3.2  setosa    less than
## 4           4.6         3.1  setosa    less than
## 5           5.0         3.6  setosa greater than
## 6           5.4         3.9  setosa greater than
## 7           4.6         3.4  setosa    less than
## 8           5.0         3.4  setosa    less than
## 9           4.4         2.9  setosa    less than
## 10          4.9         3.1  setosa    less than
```

You will use the `ifelse()` function combined with `mutate()` to add a column to the `model_pred_df` tibble.

Pipe `model_pred_df` into a `mutate()` call in order to create a new column (variable) named `y`. The new variable, `y`, will equal the result of the `ifelse()` function. The conditional statement will be if `obs_class` is equal to the `"event"`. If TRUE assign `y` to equal the value 1. If FALSE, assign `y` to equal the value 0. Assign the result to the variable `model_pred_df` which overwrites the existing value.

```
model_pred_df <- model_pred_df %>%
  mutate(y = ifelse(obs_class == "event", 1, 0))
```

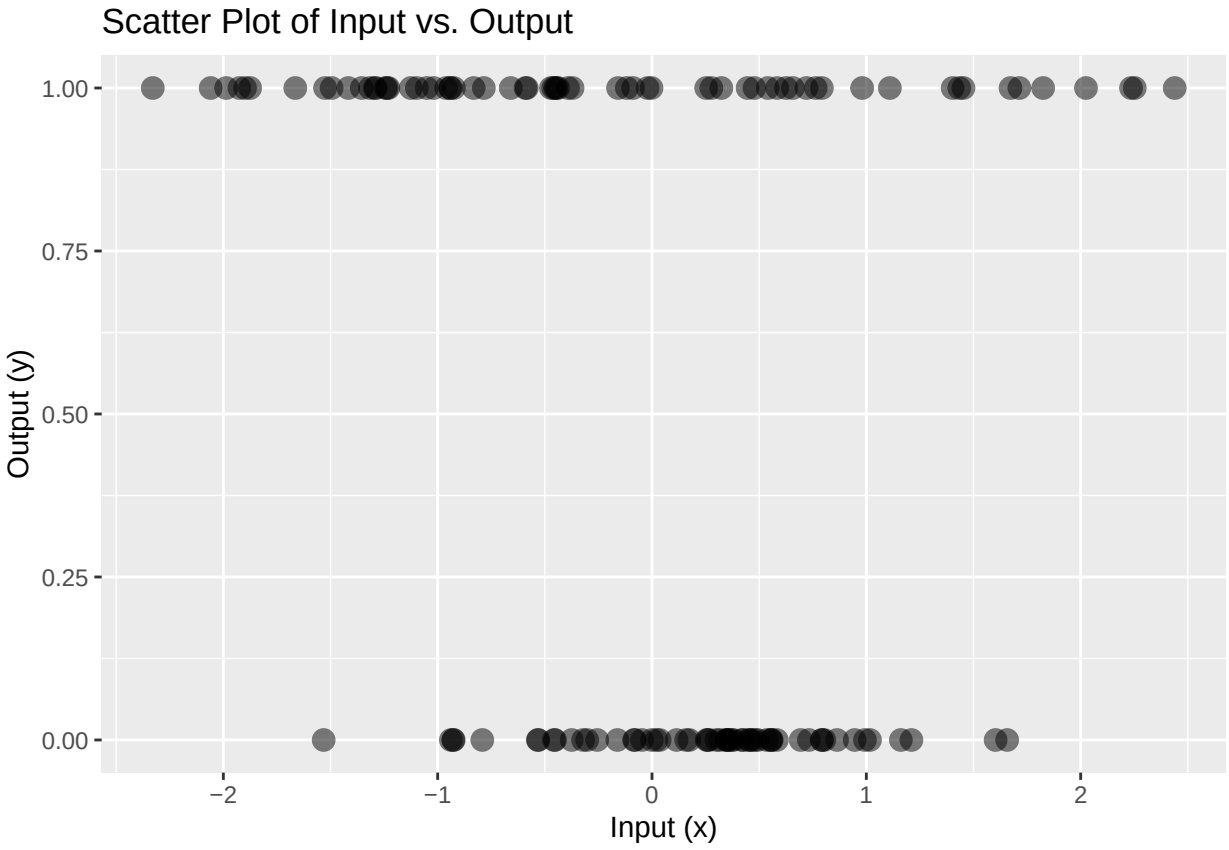
SOLUTION

1d)

You will now visualize the observed binary outcome as encoded by 0 and 1.

Pipe the `model_pred_df` object into `ggplot()`. Create a scatter plot between the encoded output `y` and the input `x`. Set the marker `size` to be 3.5 and the transparency (`alpha`) to be 0.5.

```
scatter_plot <- model_pred_df %>%
  ggplot(aes(x = x, y = y)) +
  geom_point(size = 3.5, alpha = 0.5) +
  labs(x = "Input (x)", y = "Output (y)", title = "Scatter Plot of Input vs. Output")
print(scatter_plot)
```



SOLUTION

1e)

The `model_pred_df` includes a column (variable) for a model predicted event probability.

Use the `summary()` function to confirm that the lower an upper bounds on `pred_prob` are in fact between 0 and 1.

```
summary(model_pred_df$pred_prob)
```

SOLUTION

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.2227  0.2323  0.3218  0.4547  0.6859  0.9988
```

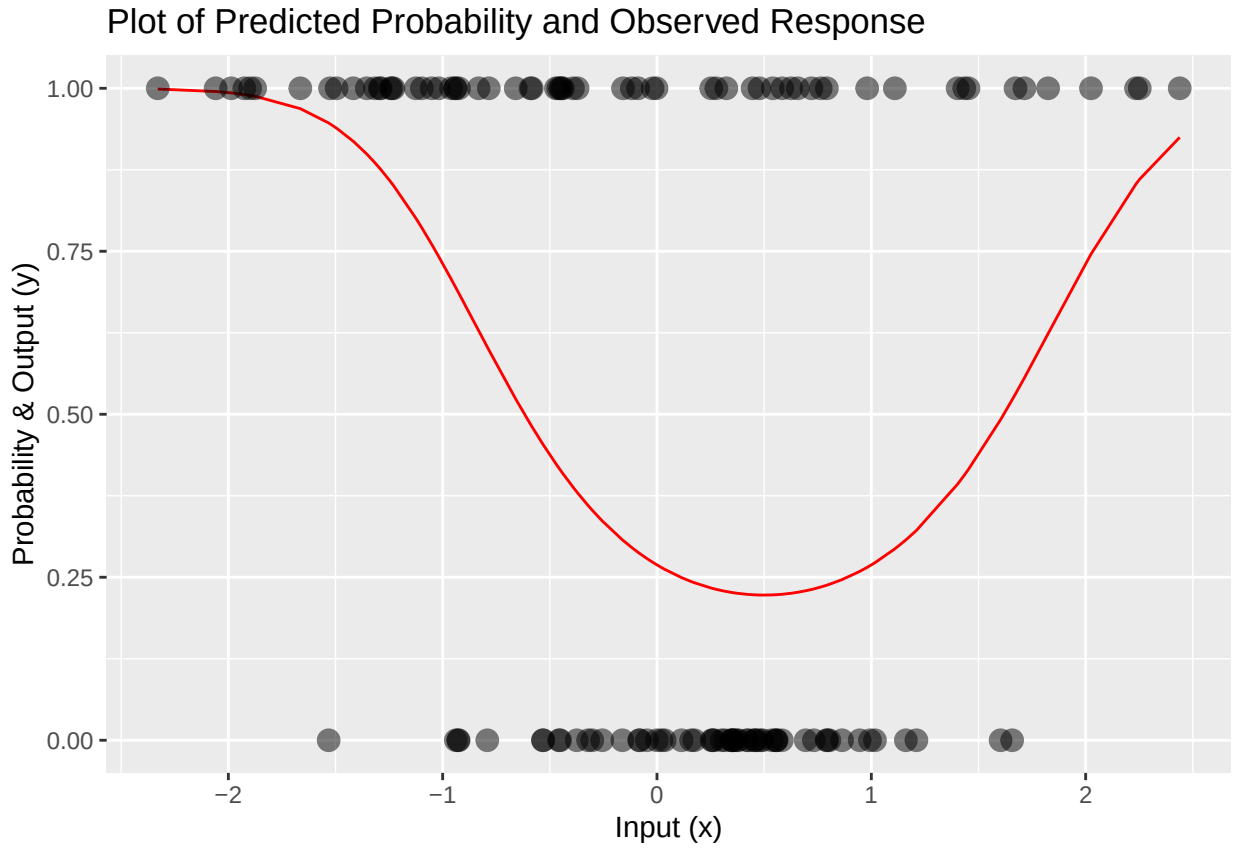
1f)

With the binary outcome encoded as 0/1 within the `y` variable we can overlay the model predicted probability on top of the observed binary response.

Use a `geom_line()` to plot the predicted event probability, `pred_prob`, with respect to the input `x`. Set the line `color` to be `"red"` within the `geom_line()` call. Overlay the binary response with the encoded response `y` as a scatter plot with `geom_point()`. Use the same marker size and transparency that you used for Problem 1d).

```
scatter_prob_plot <- model_pred_df %>%
  ggplot(aes(x = x)) +
  geom_line(aes(y = pred_prob), color = "red") +
```

```
  geom_point(aes(y = y), size = 3.5, alpha = 0.5) +
  labs(x = "Input (x)", y = "Probability & Output (y)", title = "Plot of Predicted Probability and Obse
print(scatter_prob_plot)
```



SOLUTION

1g)

Does the observed binary response "follow" the model predicted probability?

SOLUTION What do you think?

Yes, the observed binary response "follow" the model predicted probability. When the predicted probability is greater that 0.5, "event" observations are more than "non-event" observations. Similarly, when the predicted probability is less that 0.5, "non-event" observations are more than "event" observations.

Problem 02

As you can see from the `model_pred_df` tibble, we have a model predicted probability but we do not have a corresponding classification.

2a)

In order to classify our predictions we must compare the predicted probability against a threshold. You will use `ifelse()` combined with `mutate()` to create a new variable `pred_class`. If the predicted probability, `pred_prob`, is greater than the threshold set the predicted class equal to `"event"`. If the predicted probability, `pred_prob`, is less than the threshold set the predicted class equal to the `"non_event"`.

Use a threshold value of 0.5 and create the new variable `pred_class` such that the classification is `"event"` if the predicted probability is greater than the threshold and `"non_event"`

if the predicted probability is less than the threshold. Assign the result to the new object `model_class_0.5`.

```
model_class_0.5 <- model_pred_df %>%
  mutate(pred_class = ifelse(pred_prob > 0.5,"event","non_event"))
```

SOLUTION

2b)

You should now have a tibble that has a model classification and the observed binary outcome.

Calculate the Accuracy, the fraction of observations where the model classification is correct.

```
sum(model_class_0.5$pred_class == model_class_0.5$obs_class)/nrow(model_class_0.5)
```

SOLUTION

```
## [1] 0.704
```

2c)

We discussed in lecture how there are additional metrics we can consider with binary classification. Specifically, we can consider how a classification is correct, and how a classification is incorrect. A simple way to determine the counts per combination of `pred_class` and `obs_class` is with the `count()` function.

Pipe `model_class_0.5` into `count()` with `pred_class` as the first argument and `obs_class` as the second argument. You should see 4 combinations and the number of rows in the data set associated with each combination (the number or count is given by the `n` variable).

How many observations are associated with False-Positives? How many observations are associated with True-Negatives?

```
combinations <- model_class_0.5 %>%
  count(pred_class, obs_class)
print(combinations)
```

SOLUTION

```
## # A tibble: 4 x 3
##   pred_class obs_class     n
##   <chr>      <chr>     <int>
## 1 event      event        35
## 2 event      non_event     6
## 3 non_event  event        31
## 4 non_event  non_event    53
```

your response here.

6 observations are associated with False-Positives and 53 observations are associated with True-Negatives.

2d)

You will now calculate the Sensitivity and False Positive Rate (FPR) associated with the model predicted classifications based on a threshold of 0.5. This question is left open ended. It is your choice as to how you calculate the Sensitivity and FPR. However, you CANNOT use an

existing function from a library which performs the calculations automatically for you. You are permitted to use `dplyr` data manipulation functions. Include as many code chunks as you feel are necessary.

```
TP_05 <- sum(model_class_0.5$pred_class == "event" & model_class_0.5$obs_class == "event")
FN_05 <- sum(model_class_0.5$pred_class == "non_event" & model_class_0.5$obs_class == "event")
TN_05 <- sum(model_class_0.5$pred_class == "non_event" & model_class_0.5$obs_class == "non_event")
FP_05 <- sum(model_class_0.5$pred_class == "event" & model_class_0.5$obs_class == "non_event")

Sensitivity_05 <- TP_05/(TP_05+FN_05)

FPR_05 <- 1-(TN_05/(FP_05+TN_05))
```

```
print(Sensitivity_05)
```

SOLUTION

```
## [1] 0.530303
```

```
print(FPR_05)
```

```
## [1] 0.1016949
```

2e)

We also discussed the ROC curve in addition to the confusion matrix. You will not have to calculate the ROC curve for a large number of threshold values. You will go through several calculations in order to demonstrate an understanding of the steps necessary to create an ROC curve.

The first action you must perform is to make classifications based on a different threshold compared to the default value of 0.5, which we used previously.

Pipe the `model_pred_df` tibble into a `mutate()` function again, but this time determine the classifications based on a threshold value of 0.7 instead of 0.5. Assign the result to the object `model_class_0.7`.

```
model_class_0.7 <- model_pred_df %>%
  mutate(pred_class = ifelse(pred_prob > 0.7, "event", "non_event"))
```

SOLUTION

2f)

Perform the same action as in Problem 2e), but this time for a threshold value of 0.3. Assign the result to the object `model_class_0.3`.

```
model_class_0.3 <- model_pred_df %>%
  mutate(pred_class = ifelse(pred_prob > 0.3, "event", "non_event"))
```

SOLUTION

Problem 3

You will continue with the binary classification application in this problem.

3a)

Calculate the Accuracy of the model classifications based on the 0.7 threshold. You CANNOT use an existing function that calculates Accuracy automatically for you. You are permitted to use `dplyr` data manipulation functions.

```
sum(model_class_0.7$pred_class == model_class_0.7$obs_class)/nrow(model_class_0.7)
```

SOLUTION

```
## [1] 0.672
```

3b)

Calculate the Sensitivity and Specificity of the model classifications based on the 0.7 threshold. Again you can calculate these however you wish. Except you cannot use a model function library that performs the calculations automatically for you.

```
TP_07 <- sum(model_class_0.7$pred_class == "event" & model_class_0.7$obs_class == "event")
FN_07 <- sum(model_class_0.7$pred_class == "non_event" & model_class_0.7$obs_class == "event")
TN_07 <- sum(model_class_0.7$pred_class == "non_event" & model_class_0.7$obs_class == "non_event")
FP_07 <- sum(model_class_0.7$pred_class == "event" & model_class_0.7$obs_class == "non_event")

Sensitivity_07 <- TP_07/(TP_07+FN_07)

FPR_07 <- 1-(TN_07/(FP_07+TN_07))
```

```
print(Sensitivity_07)
```

SOLUTION

```
## [1] 0.3939394
```

```
print(FPR_07)
```

```
## [1] 0.01694915
```

3c)

Calculate the Accuracy of the model classifications based on the 0.3 threshold.

```
sum(model_class_0.3$pred_class == model_class_0.3$obs_class)/nrow(model_class_0.3)
```

SOLUTION

```
## [1] 0.712
```

3d)

Calculate the Sensitivity and Specificity of the model classifications based on the 0.3 threshold. Again you can calculate these however you wish. Except you cannot use a model function library that performs the calculations automatically for you.

```
TP_03 <- sum(model_class_0.3$pred_class == "event" & model_class_0.3$obs_class == "event")
FN_03 <- sum(model_class_0.3$pred_class == "non_event" & model_class_0.3$obs_class == "event")
TN_03 <- sum(model_class_0.3$pred_class == "non_event" & model_class_0.3$obs_class == "non_event")
FP_03 <- sum(model_class_0.3$pred_class == "event" & model_class_0.3$obs_class == "non_event")

Sensitivity_03 <- TP_03/(TP_03+FN_03)

FPR_03 <- 1-(TN_03/(FP_03+TN_03))
```

```
print(Sensitivity_03)
```

SOLUTION

```
## [1] 0.7272727
```

```
print(FPR_03)
```

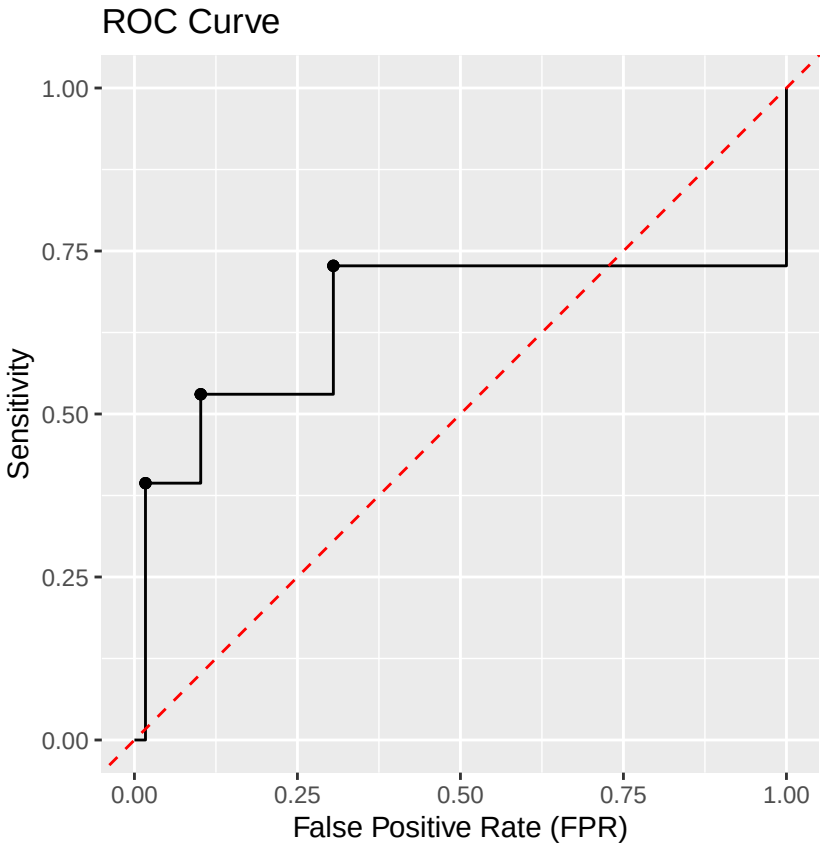
```
## [1] 0.3050847
```

3e)

You have calculated the Sensitivity and FPR at three different threshold values. You will plot your simple 3 point ROC curve and include a "45-degree" line as reference.

Use ggplot2 to plot your simple 3 point ROC curve. You must compile the necessary values into a data.frame or tibble. You must use `geom_point()` to show the markers, `geom_abline()` with `slope=1` and `intercept=0` to show the reference "45-degree" line. And you must use `coord_equal(xlim=c(0,1), ylim=c(0,1))` with your graphic. This way both axes are plotted between 0 and 1 and the axes are equal.

```
roc_data <- data.frame(
  Sensitivity = c(0,0,Sensitivity_07, Sensitivity_07, Sensitivity_05,Sensitivity_05,Sensitivity_03,Sens
  FPR = c(0,FPR_07,FPR_07, FPR_05,FPR_05,FPR_03,FPR_03,1,1),
  Sensitivity2 = c(Sensitivity_07,Sensitivity_05,Sensitivity_03),
  FPR2 = c(FPR_07,FPR_05,FPR_03)
)

roc_plot <- ggplot(roc_data, aes(x = FPR, y = Sensitivity)) +
  geom_line()+
  geom_point(aes(x = FPR2, y = Sensitivity2))+
  geom_abline(slope = 1, intercept = 0, linetype = "dashed", color = "red") +
  coord_equal(xlim = c(0, 1), ylim = c(0, 1)) +
  labs(x = "False Positive Rate (FPR)", y = "Sensitivity",
       title = "ROC Curve")

print(roc_plot)
```



SOLUTION

Problem 04

You have practiced calculating several important binary classification performance metrics. Let's use those metrics in a more realistic application to compare multiple models. The code chunk below reads in data that you will use for model training. The data consists of 4 inputs, `x1` through `x4`, and a binary outcome, `y`. The binary outcome is converted to a factor for you with the appropriate level order for modeling with `caret`.

```
train_data_path <- 'hw03_train_data.csv'

df <- readr::read_csv(train_data_path, col_names = TRUE)
```

```
## Rows: 155 Columns: 5
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (1): y
## dbl (4): x1, x2, x3, x4
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
df <- df %>%
  mutate(y = factor(y, levels = c("event", "non_event")))
```

A glimpse of the data are provided to you below.

```
df %>% glimpse()
```

```
## Rows: 155
## Columns: 5
```

```
## $ x1 <dbl> -0.3260365, 0.5524619, -0.6749438, 0.2143595, 0.3107692, 1.1739663,~
## $ x2 <dbl> -2.823000119, 0.462973827, 2.132869726, -0.270486687, 0.248525349, ~
## $ x3 <dbl> 0.23917695, -0.50232861, -1.70387658, 0.52377224, 1.56417215, -2.46~
## $ x4 <dbl> -1.14362001, -0.30082496, -1.38891427, -2.02747975, 0.50521591, 0.7~
## $ y  <fct> event, event, event, non_event, event, non_event, event, non_event,~
```

4a)

Are the levels of the binary outcome, y, balanced?

SOLUTION Add as many code chunks as you feel are necessary.

```
mean(df$y == "event")
```

```
## [1] 0.483871
```

Yes, the levels are balanced because almost half of the observations is "event".

4b)

Although it is best to explore the data in greater detail when we start a data analysis project, we will jump straight to modeling in this assignment.

Download and install `yardstick` if you have not done so already.

Load in the `caret` package and the `yardstick` packages below. Use a separate code chunk for each package.

SOLUTION Add the code chunks here.

```
library(caret)
```

```
## Loading required package: lattice
```

```
##
## Attaching package: 'caret'
```

```
## The following object is masked from 'package:purrr':
##
##     lift
```

```
library(yardstick)
```

```
##
## Attaching package: 'yardstick'
```

```
## The following objects are masked from 'package:caret':
##
##     precision, recall, sensitivity, specificity
```

```
## The following object is masked from 'package:readr':
##
##     spec
```

4c)

Just as with regression problems, we must first specify the resampling scheme and primary performance metric when we use `caret` for classification problems. All students will use the same primary performance metric in this assignment. We will begin by focusing on the Accuracy. That said, you are free to decide the kind of resampling scheme you wish to use.

The resampling scheme is controlled by the `trainControl()` function, just as it was with regression problems. You must specify the arguments to the `trainControl()` function accordingly in this problem.

Specify the resampling scheme you wish to use and assign the result to the `ctrl_acc` object. Specify the primary performance metric to be Accuracy by assigning `'Accuracy'` to the `metric_acc` argument.

```
ctrl_acc <- trainControl(method = "cv", number = 5)

metric_acc <- "Accuracy"
```

4d)

You are going to train 8 binary classifiers in this problem. The different models will use different features derived from the 4 inputs. The 8 models must have the following features:

- model 1: linear additive features all inputs
- model 2: linear features and quadratic features all inputs
- model 3: linear features and quadratic features just inputs 1 and 2
- model 4: linear features and quadratic features just inputs 1 and 3
- model 5: linear features and quadratic features just inputs 1 and 4
- model 6: linear features and quadratic features just inputs 2 and 3
- model 7: linear features and quadratic features just inputs 2 and 4
- model 8: linear features and quadratic features just inputs 3 and 4

Model 1 is the conventional "linear method" for binary classification. All other models have linear and quadratic terms to allow capturing non-linear relationships with the event probability (just how that works will be discussed later in the semester). Model 2 creates the features from all four inputs. The remaining 6 models use the combinations of just two of the inputs. This scheme is trying to identify the best possible set of inputs to model the binary outcome in a step-wise like fashion. This is **not** an efficient way to identify the best set of features to use. Later in the semester, we will learn a much more efficient *modeling approach* that performs this operation for us!

You must complete the 8 code chunks below. Use the formula interface to create the features in the model, analogous to the approach used in the previous assignment. You must specify the `method` argument in the `train()` function to be `"glm"`. You must specify the remaining arguments to `train()` accordingly.

The variable names and comments within the code chunks specify which model you are working with.

NOTE: The models are trained in separate code chunks that way you can run each model separately from the others.

```
### model 1
set.seed(2021)
mod_1_acc <- train(y ~ x1 + x2 + x3 + x4,
                   data = df,
                   method = "glm",
```

```
                      metric = metric_acc,
                      preProcess = c("center", "scale"),
                      trControl = ctrl_acc)
mod_1_acc
```

SOLUTION

```
## Generalized Linear Model
##
## 155 samples
##   4 predictor
##   2 classes: 'event', 'non_event'
##
## Pre-processing: centered (4), scaled (4)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 124, 124, 124, 124, 124
## Resampling results:
##
##   Accuracy   Kappa
##   0.7483871  0.4960866
```

```
### model 2
set.seed(2021)
mod_2_acc <- train(y ~ x1 + x2 + x3 + x4 + I(x1^2)+ I(x2^2)+ I(x3^2)+ I(x4^2),
                      data = df,
                      method = "glm",
                      metric = metric_acc,
                      preProcess = c("center", "scale"),
                      trControl = ctrl_acc)
mod_2_acc
```

```
## Generalized Linear Model
##
## 155 samples
##   4 predictor
##   2 classes: 'event', 'non_event'
##
## Pre-processing: centered (8), scaled (8)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 124, 124, 124, 124, 124
## Resampling results:
##
##   Accuracy   Kappa
##   0.7870968  0.5729577
```

```
### model 3
set.seed(2021)
mod_3_acc <- train(y ~ x1 + x2 +I(x1^2)+ I(x2^2),
                      data = df,
                      method = "glm",
                      metric = metric_acc,
                      preProcess = c("center", "scale"),
                      trControl = ctrl_acc)
mod_3_acc
```

```
## Generalized Linear Model
```

```
##
## 155 samples
##   2 predictor
##   2 classes: 'event', 'non_event'
##
## Pre-processing: centered (4), scaled (4)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 124, 124, 124, 124, 124
## Resampling results:
##
##   Accuracy  Kappa
##   0.8       0.5979811
```

```
### model 4
set.seed(2021)
mod_4_acc <- train(y ~ x1 + x3 +I(x1^2)+ I(x3^2),
                       data = df,
                       method = "glm",
                       metric = metric_acc,
                       preProcess = c("center", "scale"),
                       trControl = ctrl_acc)
mod_4_acc
```

```
## Generalized Linear Model
##
## 155 samples
##   2 predictor
##   2 classes: 'event', 'non_event'
##
## Pre-processing: centered (4), scaled (4)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 124, 124, 124, 124, 124
## Resampling results:
##
##   Accuracy  Kappa
##   0.6       0.2045638
```

```
### model 5
set.seed(2021)
mod_5_acc <- train(y ~ x1 + x4 +I(x1^2)+ I(x4^2),
                       data = df,
                       method = "glm",
                       metric = metric_acc,
                       preProcess = c("center", "scale"),
                       trControl = ctrl_acc)
mod_5_acc
```

```
## Generalized Linear Model
##
## 155 samples
##   2 predictor
##   2 classes: 'event', 'non_event'
##
## Pre-processing: centered (4), scaled (4)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 124, 124, 124, 124, 124
```

```
## Resampling results:
##
##   Accuracy  Kappa
##   0.6       0.2003237
```

```
### model 6
set.seed(2021)
mod_6_acc <- train(y ~ x2 + x3 +I(x2^2)+ I(x3^2),
                       data = df,
                       method = "glm",
                       metric = metric_acc,
                       preProcess = c("center", "scale"),
                       trControl = ctrl_acc)
mod_6_acc
```

```
## Generalized Linear Model
##
## 155 samples
##   2 predictor
##   2 classes: 'event', 'non_event'
##
## Pre-processing: centered (4), scaled (4)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 124, 124, 124, 124, 124
## Resampling results:
##
##   Accuracy   Kappa
##   0.7806452  0.5592538
```

```
### model 7
set.seed(2021)
mod_7_acc <- train(y ~ x2 + x4 +I(x2^2)+ I(x4^2),
                       data = df,
                       method = "glm",
                       metric = metric_acc,
                       preProcess = c("center", "scale"),
                       trControl = ctrl_acc)
mod_7_acc
```

```
## Generalized Linear Model
##
## 155 samples
##   2 predictor
##   2 classes: 'event', 'non_event'
##
## Pre-processing: centered (4), scaled (4)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 124, 124, 124, 124, 124
## Resampling results:
##
##   Accuracy   Kappa
##   0.7806452  0.5585457
```

```
### model 8
set.seed(2021)
mod_8_acc <- train(y ~ x3 + x4 +I(x3^2)+ I(x4^2),
```

```
                        data = df,
                        method = "glm",
                        metric = metric_acc,
                        preProcess = c("center", "scale"),
                        trControl = ctrl_acc)
mod_8_acc
```

```
## Generalized Linear Model
##
## 155 samples
##   2 predictor
##   2 classes: 'event', 'non_event'
##
## Pre-processing: centered (4), scaled (4)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 124, 124, 124, 124, 124
## Resampling results:
##
##   Accuracy   Kappa
##   0.5612903  0.1140718
```

4e)

You will now compile all resample results together and compare the models based on their Accuracy.

Complete the first code chunk below which assigns the models to the appropriate field within the `resamples()` function.

Then use the `summary()` function to summarize the Accuracy across the resamples and visualize the resample averaged performance with the `dotplot()` function from `caret`. In the function calls to both `summary()` and `dotplot()`, set the `metric` argument equal to `'Accuracy'`.

Which model is the best based on Accuracy? Are you confident it's the best?

HINT: The field names within the list contained in the `resamples()` call correspond to the model object you should use.

```
acc_results <- resamples(list(mod_1 = mod_1_acc,
                              mod_2 = mod_2_acc,
                              mod_3 = mod_3_acc,
                              mod_4 = mod_4_acc,
                              mod_5 = mod_5_acc,
                              mod_6 = mod_6_acc,
                              mod_7 = mod_7_acc,
                              mod_8 = mod_8_acc))
```

SOLUTION Summarize the results across the resamples.

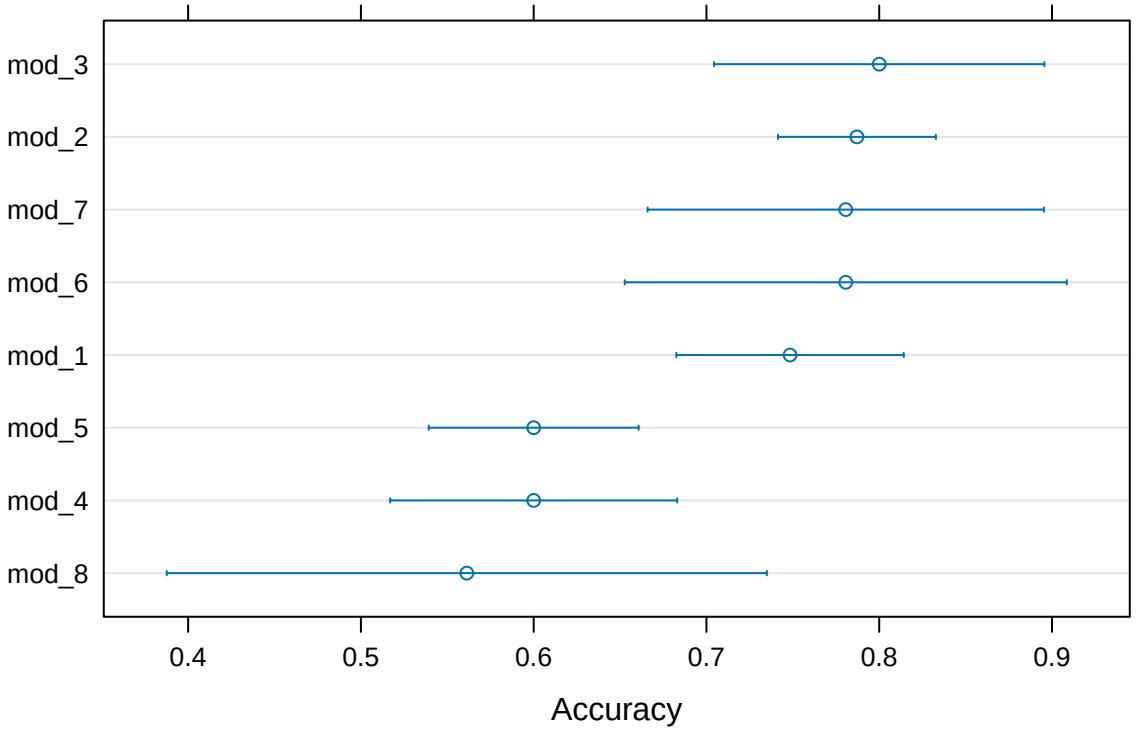
```
summary(acc_results, metric = metric_acc)
```

```
##
## Call:
## summary.resamples(object = acc_results, metric = metric_acc)
##
## Models: mod_1, mod_2, mod_3, mod_4, mod_5, mod_6, mod_7, mod_8
## Number of resamples: 5
```

```
##
## Accuracy
##            Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## mod_1 0.6774194 0.7096774 0.7741935 0.7483871 0.7741935 0.8064516    0
## mod_2 0.7419355 0.7741935 0.7741935 0.7870968 0.8064516 0.8387097    0
## mod_3 0.7096774 0.7419355 0.8064516 0.8000000 0.8387097 0.9032258    0
## mod_4 0.5483871 0.5483871 0.5806452 0.6000000 0.6129032 0.7096774    0
## mod_5 0.5483871 0.5806452 0.5806452 0.6000000 0.6129032 0.6774194    0
## mod_6 0.6451613 0.7096774 0.8064516 0.7806452 0.8387097 0.9032258    0
## mod_7 0.6774194 0.7096774 0.7741935 0.7806452 0.8387097 0.9032258    0
## mod_8 0.3870968 0.5161290 0.5483871 0.5612903 0.5806452 0.7741935    0
```

Visualize the resample averaged Accuracy per model.

```
dotplot(acc_results, metric = metric_acc)
```



Which model is the best?

Based on the Accuracy values, mod_3 has the highest accuaracy of 0.8000000, making it the best-performing model.

4f)

Next, you will consider how a model was correct and how a model was wrong via the confusion matrix. You are allowed to use the `confusionMatrix()` function from the `caret` package in this assignment to create the confusion matrix. A `caret` model object can be passed as the argument to the `confusionMatrix()` function. The function will then calculate the average confusion matrix across all resample test-sets. The resulting confusion matrix is displayed with percentages instead of counts, as shown in the lecture slides. The interpretations however are the same.

Use the `confusionMatrix()` function to display the confusion matrix for the top two and worst

two models according to Accuracy.

How do the False-Positive and False-Negative behavior compare between these four models?

SOLUTION Add as many code chunks as you feel are necessary.

```
confusionMatrix(mod_3_acc)
```

```
## Cross-Validated (5 fold) Confusion Matrix
##
## (entries are percentual average cell counts across resamples)
##
##            Reference
## Prediction  event non_event
##   event      36.1       7.7
##   non_event  12.3      43.9
##
##  Accuracy (average) : 0.8
```

```
confusionMatrix(mod_2_acc)
```

```
## Cross-Validated (5 fold) Confusion Matrix
##
## (entries are percentual average cell counts across resamples)
##
##            Reference
## Prediction  event non_event
##   event      36.8       9.7
##   non_event  11.6      41.9
##
##  Accuracy (average) : 0.7871
```

```
confusionMatrix(mod_4_acc)
```

```
## Cross-Validated (5 fold) Confusion Matrix
##
## (entries are percentual average cell counts across resamples)
##
##            Reference
## Prediction  event non_event
##   event      33.5      25.2
##   non_event  14.8      26.5
##
##  Accuracy (average) : 0.6
```

```
confusionMatrix(mod_8_acc)
```

```
## Cross-Validated (5 fold) Confusion Matrix
##
## (entries are percentual average cell counts across resamples)
##
##            Reference
## Prediction  event non_event
##   event      20.6      16.1
##   non_event  27.7      35.5
##
##  Accuracy (average) : 0.5613
```

The formulas for False Positive Rate (FPR) and False Negative Rate (FNR) are as follows:

FPR = FP / (FP + TN) FNR = FN / (FN + TP)

By utilizing these formulas with the values from confusion matrices, we can observe that both FPR and FNR exhibit an increasing trend, from the best-performing model to the worst-performing model among the four models.

Problem 05

Now that you have compared the models based on Accuracy, it is time to consider another performance metric. The Accuracy is calculated using a single threshold value. You will now examine the model performance across all possible thresholds by studying the ROC curve.

5a)

You will ultimately visually compare the ROC curves for the different models. Unfortunately with `caret`, we need to make several changes to the `trainControl()` function in order to support such comparisons. The code chunk below is started for you by including the necessary arguments you will need to visualize the ROC curves later.

You must complete the code chunk by specifying the same resampling scheme you used in Problem 4c). You must also specify the primary performance metric as `'ROC'`. That is how `caret` knows it must calculate the Area Under the Curve (AUC) for the ROC curve.

```
ctrl_roc <- trainControl(method = "cv", number = 5,
                         summaryFunction = twoClassSummary,
                         classProbs = TRUE,
                         savePredictions = TRUE)

metric_roc <- "ROC"
```

SOLUTION

5b)

You will retrain the same set of 8 models that you trained in Problem 4d), but this time using the ROC AUC as the primary performance metric.

Complete the code chunks below so that you train the 8 models again, but this time focusing on the ROC AUC. The object name and comments within the code chunks specify the model you should use.

```
### model 1
set.seed(2021)
mod_1_roc <- train(y ~ x1 + x2 + x3 + x4,
                   data = df,
                   method = "glm",
                   metric = metric_roc,
                   preProcess = c("center", "scale"),
                   trControl = ctrl_roc)
mod_1_roc
```

SOLUTION

```
## Generalized Linear Model
##
## 155 samples
##   4 predictor
##   2 classes: 'event', 'non_event'
##
## Pre-processing: centered (4), scaled (4)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 124, 124, 124, 124, 124
## Resampling results:
##
##   ROC     Sens       Spec
##   0.8025  0.7466667  0.75
```

```
### model 2
set.seed(2021)
mod_2_roc <- train(y ~ x1 + x2 + x3 + x4 + I(x1^2)+ I(x2^2)+ I(x3^2)+ I(x4^2),
                       data = df,
                       method = "glm",
                       metric = metric_roc,
                       preProcess = c("center", "scale"),
                       trControl = ctrl_roc)
mod_2_roc
```

```
## Generalized Linear Model
##
## 155 samples
##   4 predictor
##   2 classes: 'event', 'non_event'
##
## Pre-processing: centered (8), scaled (8)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 124, 124, 124, 124, 124
## Resampling results:
##
##   ROC        Sens  Spec
##   0.9066667  0.76  0.8125
```

```
### model 3
set.seed(2021)
mod_3_roc <- train(y ~ x1 + x2 +I(x1^2)+ I(x2^2),
                       data = df,
                       method = "glm",
                       metric = metric_roc,
                       preProcess = c("center", "scale"),
                       trControl = ctrl_roc)
mod_3_roc
```

```
## Generalized Linear Model
##
## 155 samples
##   2 predictor
##   2 classes: 'event', 'non_event'
##
## Pre-processing: centered (4), scaled (4)
## Resampling: Cross-Validated (5 fold)
```

```
## Summary of sample sizes: 124, 124, 124, 124, 124
## Resampling results:
##
##   ROC   Sens       Spec
##   0.92  0.7466667  0.85
```

```
### model 4
set.seed(2021)
mod_4_roc <- train(y ~ x1 + x3 +I(x1^2)+ I(x3^2),
                   data = df,
                   method = "glm",
                   metric = metric_roc,
                   preProcess = c("center", "scale"),
                   trControl = ctrl_roc)
mod_4_roc
```

```
## Generalized Linear Model
##
## 155 samples
##   2 predictor
##   2 classes: 'event', 'non_event'
##
## Pre-processing: centered (4), scaled (4)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 124, 124, 124, 124, 124
## Resampling results:
##
##   ROC        Sens       Spec
##   0.6058333  0.6933333  0.5125
```

```
### model 5
set.seed(2021)
mod_5_roc <- train(y ~ x1 + x4 +I(x1^2)+ I(x4^2),
                   data = df,
                   method = "glm",
                   metric = metric_roc,
                   preProcess = c("center", "scale"),
                   trControl = ctrl_roc)
mod_5_roc
```

```
## Generalized Linear Model
##
## 155 samples
##   2 predictor
##   2 classes: 'event', 'non_event'
##
## Pre-processing: centered (4), scaled (4)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 124, 124, 124, 124, 124
## Resampling results:
##
##   ROC        Sens       Spec
##   0.6341667  0.6133333  0.5875
```

```
### model 6
set.seed(2021)
```

```r
mod_6_roc <- train(y ~ x2 + x3 +I(x2^2)+ I(x3^2),
                   data = df,
                   method = "glm",
                   metric = metric_roc,
                   preProcess = c("center", "scale"),
                   trControl = ctrl_roc)
mod_6_roc
```

```
## Generalized Linear Model
##
## 155 samples
##   2 predictor
##   2 classes: 'event', 'non_event'
##
## Pre-processing: centered (4), scaled (4)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 124, 124, 124, 124, 124
## Resampling results:
##
##   ROC        Sens       Spec
##   0.8458333  0.7333333  0.825
```

```r
### model 7
set.seed(2021)
mod_7_roc <- train(y ~ x2 + x4 +I(x2^2)+ I(x4^2),
                   data = df,
                   method = "glm",
                   metric = metric_roc,
                   preProcess = c("center", "scale"),
                   trControl = ctrl_roc)
mod_7_roc
```

```
## Generalized Linear Model
##
## 155 samples
##   2 predictor
##   2 classes: 'event', 'non_event'
##
## Pre-processing: centered (4), scaled (4)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 124, 124, 124, 124, 124
## Resampling results:
##
##   ROC     Sens       Spec
##   0.8475  0.7066667  0.85
```

```r
### model 8
set.seed(2021)
mod_8_roc <- train(y ~ x3 + x4 +I(x3^2)+ I(x4^2),
                   data = df,
                   method = "glm",
                   metric = metric_roc,
                   preProcess = c("center", "scale"),
                   trControl = ctrl_roc)
mod_8_roc
```

```
## Generalized Linear Model
##
## 155 samples
##   2 predictor
##   2 classes: 'event', 'non_event'
##
## Pre-processing: centered (4), scaled (4)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 124, 124, 124, 124, 124
## Resampling results:
##
##   ROC   Sens       Spec
##   0.52  0.4266667  0.6875
```

5c)

You will now compile all resample results together and compare the models based on their area under the ROC curve.

Complete the first code chunk below which assigns the models to the appropriate field within the `resamples()` function.

Then use the `summary()` function to summarize the ROC AUC across the resamples and visualize the resample averaged performance with the `dotplot()` function from `caret`. In the function calls to both `summary()` and `dotplot()`, set the `metric` argument equal to `'ROC'`.

Which model is the best based on ROC AUC? Are you confident it's the best?

HINT: The field names within the list contained in the `resamples()` call correspond to the model object you should use.

```
roc_results <- resamples(list(mod_1 = mod_1_roc,
                              mod_2 = mod_2_roc,
                              mod_3 = mod_3_roc,
                              mod_4 = mod_4_roc,
                              mod_5 = mod_5_roc,
                              mod_6 = mod_6_roc,
                              mod_7 = mod_7_roc,
                              mod_8 = mod_8_roc))
```

SOLUTION Summarize the results across the resamples.

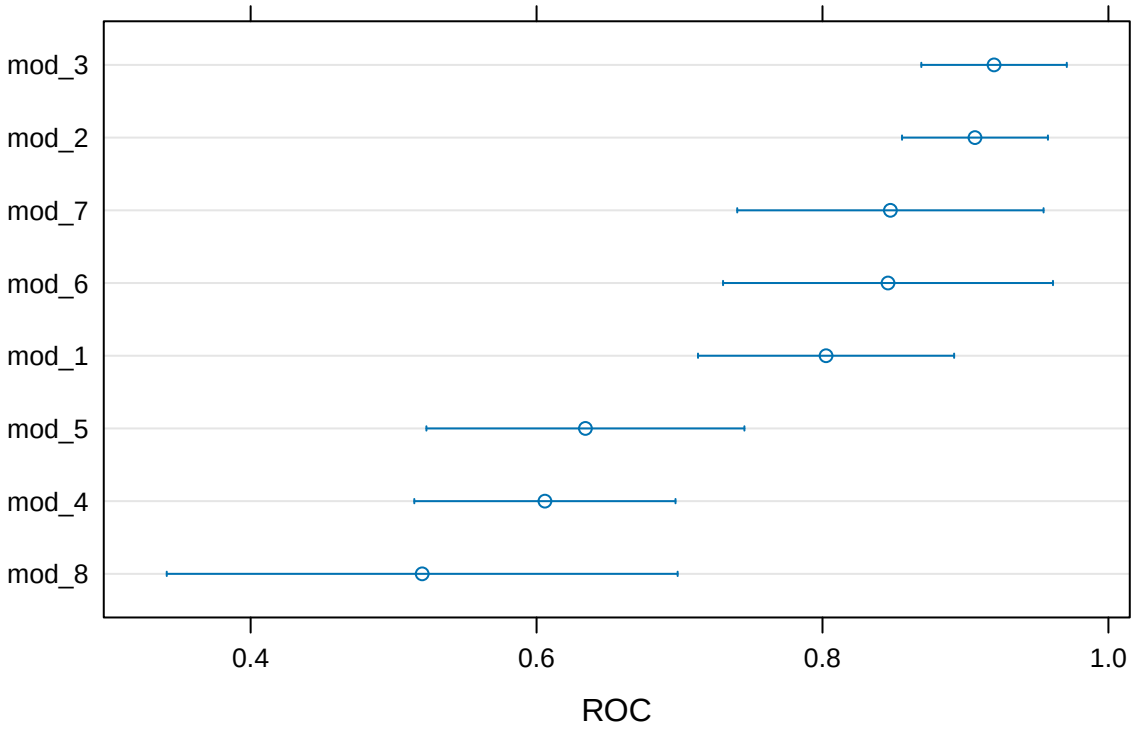
```
summary(roc_results, metric = metric_roc)
```

```
##
## Call:
## summary.resamples(object = roc_results, metric = metric_roc)
##
## Models: mod_1, mod_2, mod_3, mod_4, mod_5, mod_6, mod_7, mod_8
## Number of resamples: 5
##
## ROC
##            Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## mod_1 0.6916667 0.7958333 0.8083333 0.8025000 0.8250000 0.8916667    0
## mod_2 0.8625000 0.8750000 0.9083333 0.9066667 0.9208333 0.9666667    0
## mod_3 0.8541667 0.9083333 0.9375000 0.9200000 0.9416667 0.9583333    0
```

```
## mod_4 0.5500000 0.5583333 0.5750000 0.6058333 0.6166667 0.7291667    0
## mod_5 0.5166667 0.5750000 0.6416667 0.6341667 0.7041667 0.7333333    0
## mod_6 0.7166667 0.7833333 0.8791667 0.8458333 0.9125000 0.9375000    0
## mod_7 0.7083333 0.8291667 0.8666667 0.8475000 0.9083333 0.9250000    0
## mod_8 0.3875000 0.4333333 0.4750000 0.5200000 0.5500000 0.7541667    0
```

Visualize the resample averaged ROC AUC per model.

```
dotplot(roc_results, metric = metric_roc)
```



Which model is the best?

Based on the ROC AUC values, mod_3 has the highest mean area under the ROC curve with value of 0.9200000, making it the best-performing model.

5d)

By default, two other metrics are calculated by `caret` when we use the ROC AUC as the primary performance metric. Unlike ROC AUC, these two metrics are calculated with the default threshold. `caret` labels the the Sensitivity as the `Sens` metric and the Specificity as the `Spec` metric.

Use the `summary()` and `dotplot()` functions again, but do not specify a metric. Just provide the `roc_results` as the input argument to the functions.

Which model has the highest True-Positive Rate at the default threshold? Which model has the lowest False-Positive Rate at the default threshold?

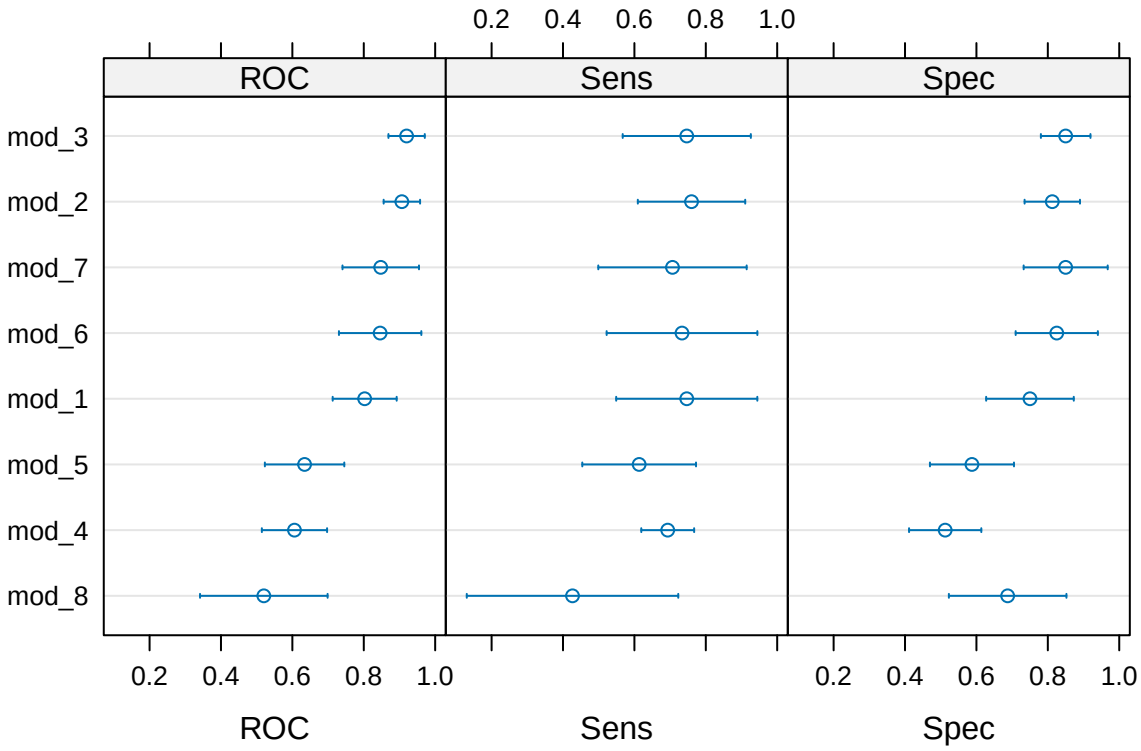
SOLUTION Add as many code chunks and as much text as you feel are necessary.

```
summary(roc_results)
```

```
##
```

```
## Call:
## summary.resamples(object = roc_results)
##
## Models: mod_1, mod_2, mod_3, mod_4, mod_5, mod_6, mod_7, mod_8
## Number of resamples: 5
##
## ROC
##            Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## mod_1 0.6916667 0.7958333 0.8083333 0.8025000 0.8250000 0.8916667    0
## mod_2 0.8625000 0.8750000 0.9083333 0.9066667 0.9208333 0.9666667    0
## mod_3 0.8541667 0.9083333 0.9375000 0.9200000 0.9416667 0.9583333    0
## mod_4 0.5500000 0.5583333 0.5750000 0.6058333 0.6166667 0.7291667    0
## mod_5 0.5166667 0.5750000 0.6416667 0.6341667 0.7041667 0.7333333    0
## mod_6 0.7166667 0.7833333 0.8791667 0.8458333 0.9125000 0.9375000    0
## mod_7 0.7083333 0.8291667 0.8666667 0.8475000 0.9083333 0.9250000    0
## mod_8 0.3875000 0.4333333 0.4750000 0.5200000 0.5500000 0.7541667    0
##
## Sens
##            Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## mod_1 0.4666667 0.8000000 0.8000000 0.7466667 0.8000000 0.8666667    0
## mod_2 0.6000000 0.7333333 0.7333333 0.7600000 0.8000000 0.9333333    0
## mod_3 0.5333333 0.7333333 0.7333333 0.7466667 0.8000000 0.9333333    0
## mod_4 0.6666667 0.6666667 0.6666667 0.6933333 0.6666667 0.8000000    0
## mod_5 0.4666667 0.5333333 0.6000000 0.6133333 0.6666667 0.8000000    0
## mod_6 0.4666667 0.7333333 0.7333333 0.7333333 0.8000000 0.9333333    0
## mod_7 0.4666667 0.6666667 0.7333333 0.7066667 0.7333333 0.9333333    0
## mod_8 0.1333333 0.3333333 0.3333333 0.4266667 0.6000000 0.7333333    0
##
## Spec
##         Min. 1st Qu. Median   Mean 3rd Qu.   Max. NA's
## mod_1 0.6250  0.6875 0.7500 0.7500  0.8125 0.8750    0
## mod_2 0.7500  0.7500 0.8125 0.8125  0.8750 0.8750    0
## mod_3 0.7500  0.8750 0.8750 0.8500  0.8750 0.8750    0
## mod_4 0.4375  0.4375 0.5000 0.5125  0.5625 0.6250    0
## mod_5 0.5000  0.5625 0.5625 0.5875  0.5625 0.7500    0
## mod_6 0.6875  0.8125 0.8125 0.8250  0.8750 0.9375    0
## mod_7 0.6875  0.8750 0.8750 0.8500  0.8750 0.9375    0
## mod_8 0.5000  0.6250 0.6875 0.6875  0.8125 0.8125    0
```

```
dotplot(roc_results)
```



mod_2 has the highest TPR with mean 0.7600000.

mod_3 has the lowest FPR with mean 1-0.8500=0.1500.

5e)

In order to visualize the ROC curve we need to understand how the resample hold-out test predictions are stored within the `caret` model objects. By default, hold-out test set predictions are not retained, in order to conserve memory. However, the `ctrl_roc` object set `savePredictions = TRUE` which overrides the default behavior and stores each resample test-set predictions.

The predictions are contained with the `$pred` field of the `caret` model object. The code chunk below displays the first few rows of the predictions for the `mod_1_roc` result for you. Note that the code chunk below is not evaluated by default. When you execute the code chunk below, you will see 7 columns. The column `obs` is the observed outcome and the column `event` is the predicted probability of the `event`. The `pred` column is the model classified outcome based on the default threshold of 50%. The `rowIndex` is the row from the original data set and serves to identify the row correctly. The `Resample` column tells us which resample fold the test point was associated with.

```
mod_1_roc$pred %>% tibble::as_tibble()
```

```
## # A tibble: 155 x 7
##    pred      obs       event non_event rowIndex parameter Resample
##    <fct>     <fct>     <dbl>     <dbl>    <int> <chr>     <chr>
##  1 non_event non_event 0.249     0.751        8 none      Fold1
##  2 non_event non_event 0.175     0.825       12 none      Fold1
##  3 non_event non_event 0.131     0.869       13 none      Fold1
##  4 event     event     0.785     0.215       21 none      Fold1
##  5 non_event event     0.285     0.715       27 none      Fold1
##  6 non_event non_event 0.442     0.558       39 none      Fold1
```

```
##  7 non_event non_event 0.435     0.565       40 none      Fold1
##  8 event     event     0.800     0.200       43 none      Fold1
##  9 event     non_event 0.698     0.302       48 none      Fold1
## 10 non_event event     0.384     0.616       52 none      Fold1
## # i 145 more rows
```

The ROC curve is calculated by comparing the model predicted probability to all possible thresholds to create many different classifications. Those different classifications are used to calculate many different confusion matrices. Thus, the columns of primary interest in the prediction object displayed above are the `obs` and `event` columns.

You manually created 3 points on the ROC curve previously in this assignment. Do NOT worry, you do NOT need to repeat those actions here! Instead you will use the `roc_curve()` function from the `yardstick` package to create the ROC curve data for you. The `roc_curve()` function has three primary arguments. The first is a data object which contains the predictions in a "tidy" format. The second is the name of the column that corresponds to the observed outcome (the truth or reference). The third is the name of the column in the data set that corresponds to the model predicted event probability.

Pipe the prediction data object for the `mod_1_roc caret` object to the `roc_curve()`. The `obs` column is the observed outcome and the `event` column is the model predicted event probability. Display the result to the screen to confirm the `roc_curve()` function worked. If it did the first few rows should correspond to very low threshold values.

Why does the `sensitivity` have values at or near 1 when the `.threshold` is so low?

```
roc_data5e <-  roc_curve(mod_1_roc$pred,obs, event)
print(roc_data5e)
```

SOLUTION

```
## # A tibble: 157 x 3
##    .threshold specificity sensitivity
##         <dbl>       <dbl>       <dbl>
##  1  -Inf           0           1
##  2     0.0153      0           1
##  3     0.0289      0           0.987
##  4     0.0364      0           0.973
##  5     0.0458      0           0.96
##  6     0.0528      0.0125      0.96
##  7     0.0559      0.0125      0.947
##  8     0.0625      0.0250      0.947
##  9     0.0765      0.0375      0.947
## 10     0.0796      0.0500      0.947
## # i 147 more rows
```

What do you think?

When the threshold is so low, the prediction is classified as "event" in each case, which makes FN=0.

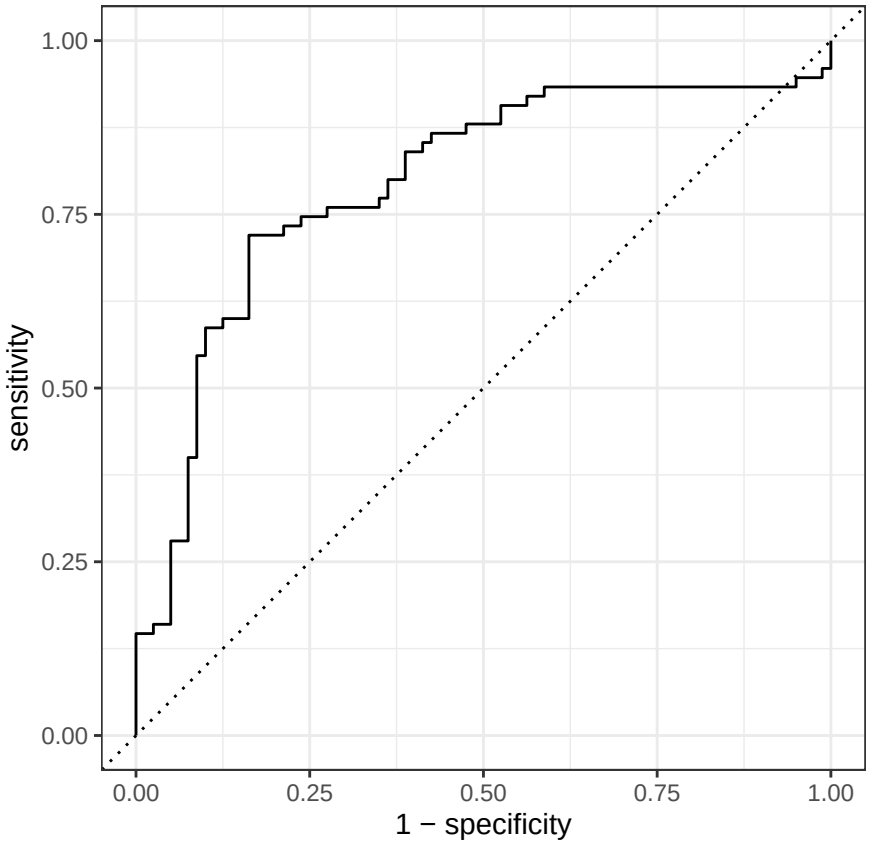
Sensitivity=TP/(TP+FN)=TP/TP=1 in the case when threshold is so low.

5f)

You will now visualize the ROC curve associated with `mod_1_roc`.

Repeat the same steps you performed in 5e) above, except pipe the result to the `autoplot()` method.

```
roc_data5f <- roc_curve(mod_1_roc$pred,obs, event)
roc_data5f %>% autoplot()
```



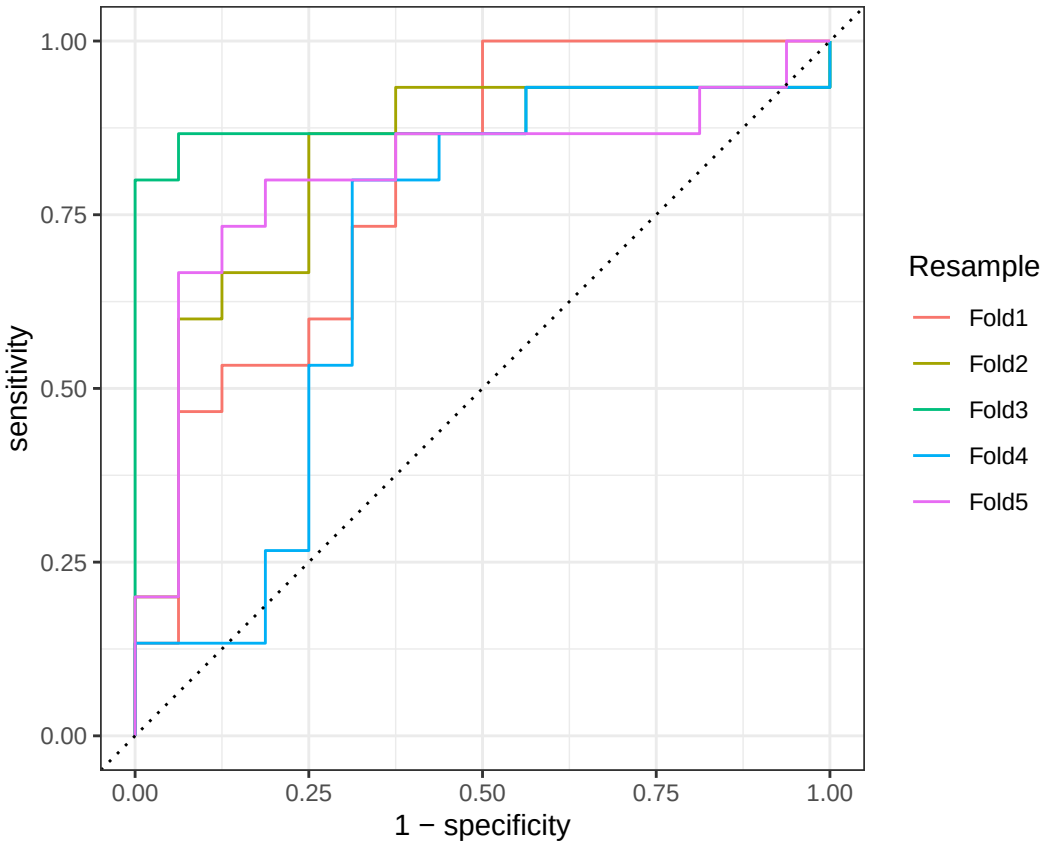
SOLUTION

5g)

The ROC curve displayed in 5f) is the resample averaged ROC curve. You can examine the individual resample hold-out test set ROC curves by specifying a grouping structure with the `group_by()` function. This can help you get a sense of the variability in the ROC curve.

Pipe the prediction object associated with `mod_1_roc` to the `group_by()` function where you specify the grouping variable to be `Resample`. Pipe the result to `roc_curve()` where you specify the same arguments as in the previous questions. Finally, pipe the result to `autoplot()`.

```
roc_data5g <- mod_1_roc$pred %>% group_by(Resample)
roc_data5gg <- roc_data5g %>%roc_curve(obs, event)
autoplot(roc_data5gg)
```



SOLUTION

5h)

A function is defined for you in the code chunk below. This function compiles all model results together to enable comparing their ROC curves.

```
compile_all_model_preds <- function(m1, m2, m3, m4, m5, m6, m7, m8)
{
  purrr::map2_dfr(list(m1, m2, m3, m4, m5, m6, m7, m8),
                  as.character(seq_along(list(m1, m2, m3, m4, m5, m6, m7, m8))),
                  function(ll, lm){
                    ll$pred %>% tibble::as_tibble() %>%
                      select(obs, event, Resample) %>%
                      mutate(model_name = lm)
                  })
}
```

The code chunk below is also completed for you. It passes the `caret` model objects with the saved predictions to the `compile_all_model_preds()` function. The result is printed for you below so you can see the column names. Notice there is a new column `model_name` which stores the name of the model associated with the resample hold-out test set predictions. By default the code chunk below is not executed.

```
all_model_preds <- compile_all_model_preds(mod_1_roc, mod_2_roc, mod_3_roc,
                                           mod_4_roc, mod_5_roc,
                                           mod_6_roc, mod_7_roc, mod_8_roc)

all_model_preds
```

```
## # A tibble: 1,240 x 4
```

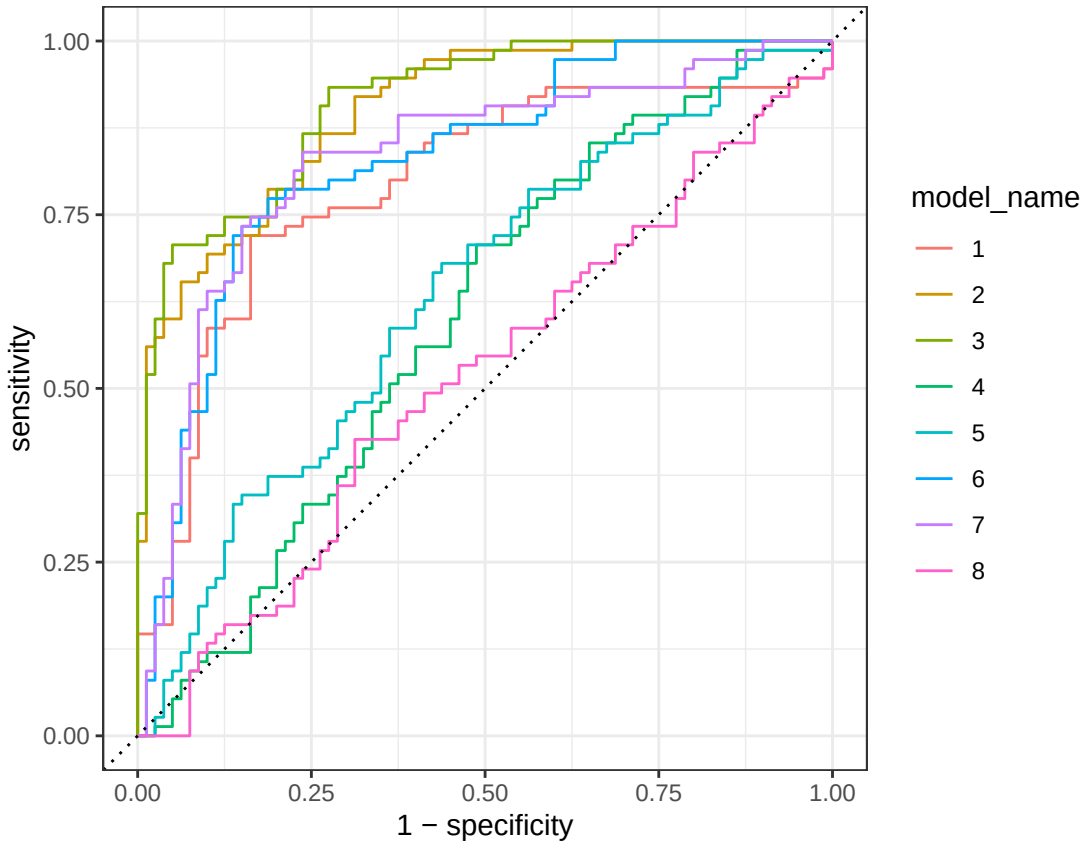
```
##    obs       event Resample model_name
##    <fct>     <dbl> <chr>    <chr>
##  1 non_event 0.249 Fold1    1
##  2 non_event 0.175 Fold1    1
##  3 non_event 0.131 Fold1    1
##  4 event     0.785 Fold1    1
##  5 event     0.285 Fold1    1
##  6 non_event 0.442 Fold1    1
##  7 non_event 0.435 Fold1    1
##  8 event     0.800 Fold1    1
##  9 non_event 0.698 Fold1    1
## 10 event     0.384 Fold1    1
## # i 1,230 more rows
```

You will now create a figure which displays the resample averaged ROC curve for each model.

Pipe the `all_model_preds` object to `group_by()` and specify the grouping variable as `model_name`. Pipe the result to `roc_curve()` and specify the arguments accordingly. Pipe the result to `autoplot()` to generate the figure.

Which model is the best? Is the result consistent with the ROC AUC summary metrics you calculated previously? Which model is closest to a "completely ineffective model" and why is that so?

```
roc_curves5h <- all_model_preds %>%
  group_by(model_name) %>%
  roc_curve(obs,event)

autoplot(roc_curves5h)
```



SOLUTION

What do you think?

1) The best model is Model 3. Because the area under the ROC curve is the highest in that case.
2) The results are consistent with ROC AUC summary metrics I calculated previously. The best models are Model 3 and Model 2, the worst Model is Model 8 in both cases.
3) Model 8 is closest to "completely ineffective model", because its ROC curve is close to the line y=x.

Problem 06

The performance metrics you have focused on up to this point are point-wise comparison metrics which evaluate the classification performance of a model. As discussed in lecture, binary classifier performance can also be measured based on the **calibration** between the predicted probability and the observed event proportion. The performance is represented graphically via the **calibration curve** which visualizes the correlation between the model predictions and the observed proportion of the event.

Regardless of your rankings in the previous questions, you will compare the performance of model 1, model 3, and model 8 with the calibration curve. Although multiple functions exist to create the calibration curve, you **must** create the calibration curve manually in this problem. You are only allowed to use functions from the `dplyr` and `ggplot2` packages. You are **not** allowed to use any third party function to calculate the calibration curve in this problem.

In the two previous problems, you used resampling to assess model performance. We could create the calibration curve based on the resample fold test sets, but we will instead start simpler and use a dedicated hold-out set that is different from the training data. The hold-out set is read in for you in the code chunk below.

```
test_data_path <- 'hw03_test_data.csv'
```

```
df_test <- readr::read_csv(test_data_path, col_names = TRUE)
```

```
## Rows: 120 Columns: 5
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (1): y
## dbl (4): x1, x2, x3, x4
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
df_test <- df_test %>%
  mutate(y = factor(y, levels = c("event", "non_event")))
```

The code chunk below shows a glimpse of the test set, which demonstrates the test set has the same column names as the training set.

```
df_test %>% glimpse()
```

```
## Rows: 120
## Columns: 5
## $ x1 <dbl> -0.226126900, 1.573924144, -1.054455037, 0.083051357, 0.424592933, ~
## $ x2 <dbl> -0.3361453, 0.5445463, 0.5837157, -0.8979020, -0.6339053, -0.641050~
## $ x3 <dbl> -1.22092445, -0.41352769, 2.45375479, -0.31070519, 0.40871874, 0.15~
## $ x4 <dbl> 0.59734091, 1.16494692, 0.41455929, 0.03209814, -0.91177742, 0.1022~
## $ y  <fct> event, non_event, non_event, non_event, non_event, non_event, non_e~
```

6a)

The first step to create the calibration curve requires making predictions. The `predict()` function for a `caret` trained object function has two main arguments. The first is the model object we will use to make predictions with and the second, `newdata`, is the data set to predict. The input columns must have the same names as the inputs to the training data that trained the model.

By default, a binary classifier will return the classifications assuming a 50% threshold. As discussed in lecture, the calibration curve does not work with classifications. Instead we need the predicted probability! We can override the default prediction behavior by setting additional arguments to the `predict()` function. Specifically, the `type` argument instructs the model to return a "type" of prediction. We want the predicted probability and so you must set `type = 'prob'` in the `predict()` function call.

Predict the hold-out test set using model 1 and assign the result to the variable `pred_test_1`. Return the predicted probability by setting the `type` argument to `'prob'`.

Print the data type of the `pred_test_1` object to the screen and use the `head()` function to display the "top" of the object.

HINT: It does not matter whether you use the model 1 assessed based on Accuracy or ROC in this problem.

```
pred_test_1 <- predict(mod_1_roc,newdata = df_test,type = 'prob')
```

```
class(pred_test_1)
```

SOLUTION

```
## [1] "data.frame"
```

```
head(pred_test_1)
```

```
##       event non_event
## 1 0.2607671 0.7392329
## 2 0.7113031 0.2886969
## 3 0.7467320 0.2532680
## 4 0.1964256 0.8035744
## 5 0.3579798 0.6420202
## 6 0.2867937 0.7132063
```

6b)

Your `pred_test_1` object should have 2 columns. The `event` column gives the predicted probability that `y == 'event'` and the `non_event` column stores the predicted probability that `y == 'non_event'`.

What is the relationship between the values in the `event` column and the `non_event` column?

SOLUTION What do you think?

In each row, the sum of the values equals 1, which is expected in binary classification.

6c)

The code chunk below binds the columns in `pred_test_1` with the `df_test` dataframe to create a new dataframe which includes the predicted probability of the event and the observed output, `y`, for the hold-out test set. **PLEASE NOTE**: the code chunk below is **NOT** evaluated by default. You must change the `eval` chunk option to make sure the code chunk is executed when you render the report.

```
test_df_1 <- df_test %>% bind_cols(pred_test_1)
```

As discussed in lecture, the calibration curve bins or lumps the predicted probability into uniformly spaced intervals. The empirical proportion of the event within each bin must be calculated. Thus, you need to convert the numeric predicted probability into a discrete or categorical variable.

A simple, yet effective, approach for categorizing a continuous variable is the `cut()` function. The cut function is demonstrated in the code chunk below. The variable `x` is a column within a tibble (a dataframe). The `x` variable consists of integers between 0 and 100. The `x` variable is "cut" or divided into bins with **break points** at values of 0, 10, 20, 30, etc. The break points are created using the `seq()` function from 0 to 100 *by* increments of 10. The tibble is piped to the `count()` function to count the number of rows associated with each unique value of the `x_bin`. Pay close attention to the displayed values of `x_bin`. The values of `x_bin` show the "cut" or "divided" intervals.

```
tibble::tibble(x = 0:100) %>%
  mutate(x_bins = cut(x,
                      breaks = seq(0, 100, by = 10),
                      include.lowest = TRUE)) %>%
  count(x_bins)
```

```
## # A tibble: 10 x 2
##    x_bins       n
##    <fct>    <int>
##  1 [0,10]      11
##  2 (10,20]     10
##  3 (20,30]     10
##  4 (30,40]     10
##  5 (40,50]     10
##  6 (50,60]     10
```

```
##  7 (60,70]     10
##  8 (70,80]     10
##  9 (80,90]     10
## 10 (90,100]    10
```

You must use the `cut()` function to bin the predicted probability associated with model 1 into 10 bins. Think carefully about how to specify the `breaks` argument to `cut()` so that the probability is divided into 10 uniform intervals.

Pipe `test_df_1` to `mutate()` and create a variable `pred_bin` by cutting the predicted probability into 10 uniform intervals. Assign the result to the `test_df_1_b` object.

HINT: You should not pipe the result to `count()` as the previous code chunk did. The `count()` function was used to show the *levels* of the created discrete variable.

```
test_df_1_b <- test_df_1 %>%
  mutate(pred_bin = cut(event, breaks = seq(0, 1, by = 0.1), include.lowest = TRUE))
```

SOLUTION

6d)

Use the `count()` function to count the number of rows associated with each unique value of `pred_bin` in the `test_df_1_b` object. Display the result to the screen. How many unique values are displayed?

```
result6d <- test_df_1_b %>%
  count(pred_bin)
result6d
```

SOLUTION

```
## # A tibble: 10 x 2
##    pred_bin      n
##    <fct>     <int>
##  1 [0,0.1]      11
##  2 (0.1,0.2]    13
##  3 (0.2,0.3]    11
##  4 (0.3,0.4]     7
##  5 (0.4,0.5]    19
##  6 (0.5,0.6]    15
##  7 (0.6,0.7]    22
##  8 (0.7,0.8]    13
##  9 (0.8,0.9]     5
## 10 (0.9,1]       4
```

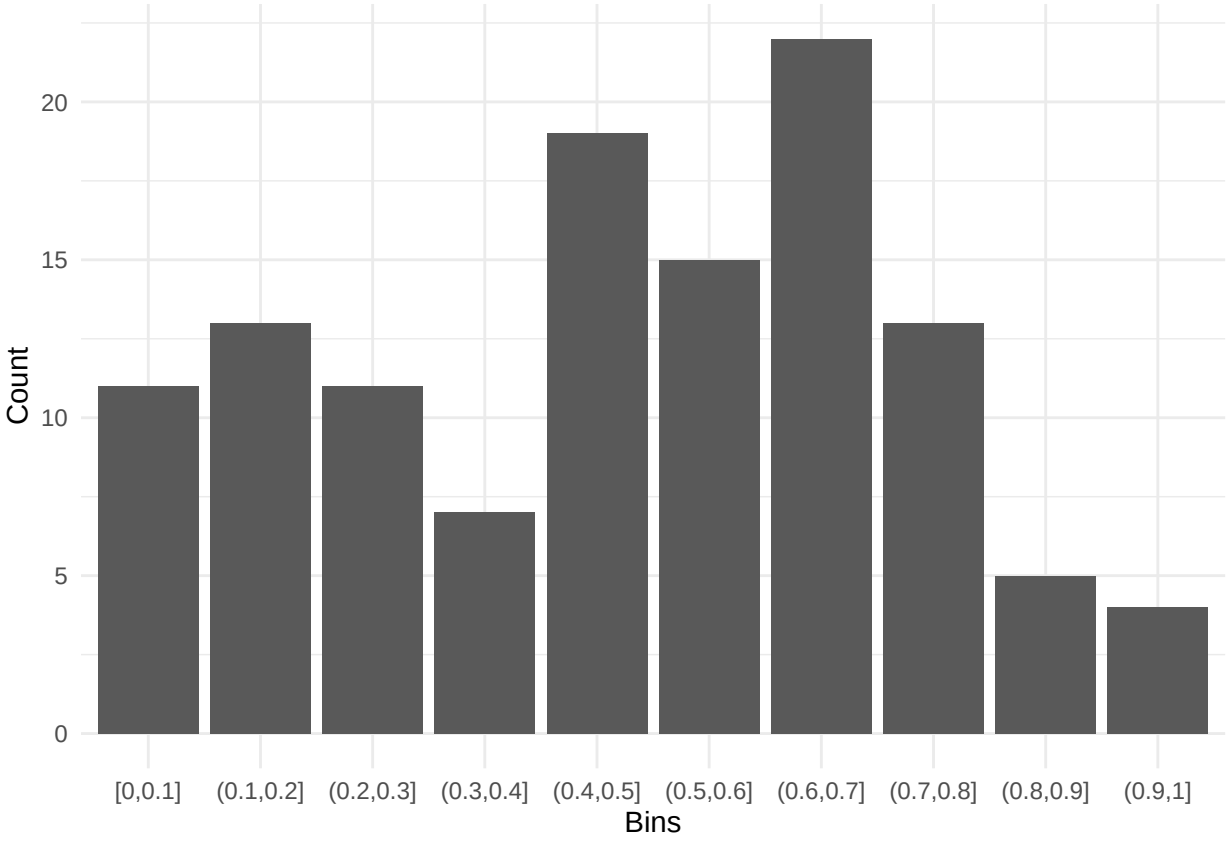
6e)

Show the counts per bin again but this time visualize the counts with a bar chart.

Create a bar chart with `ggplot2` for the counts associated with unique value of `pred_bin`.

```
ggplot(result6d, aes(x=pred_bin, y=n)) +
  geom_bar(stat = "identity") +
```

```
  labs(x ="Bins", y="Count") +
  theme_minimal()
```



SOLUTION

6f)

As shown in the lecture slides, we can create a stacked bar chart to get a rough idea about the number of events and non-events within each predicted probability bin. This is simple to do with `ggplot2` by using the `fill` aesthetic associated with the `geom_bar()` geometric object.

Create a stacked bar chart with `ggplot2` where the `fill` aesthetic is mapped to the observed binary outcome. Overrid the default fill color scheme by including the `scale_color_brewer()` function after the `geom_bar()` layer. Set the `palette` argument in `scale_color_brewer()` to `'Set1'`.

Do any bins consist of only events? Do any bins consist of only non-events?

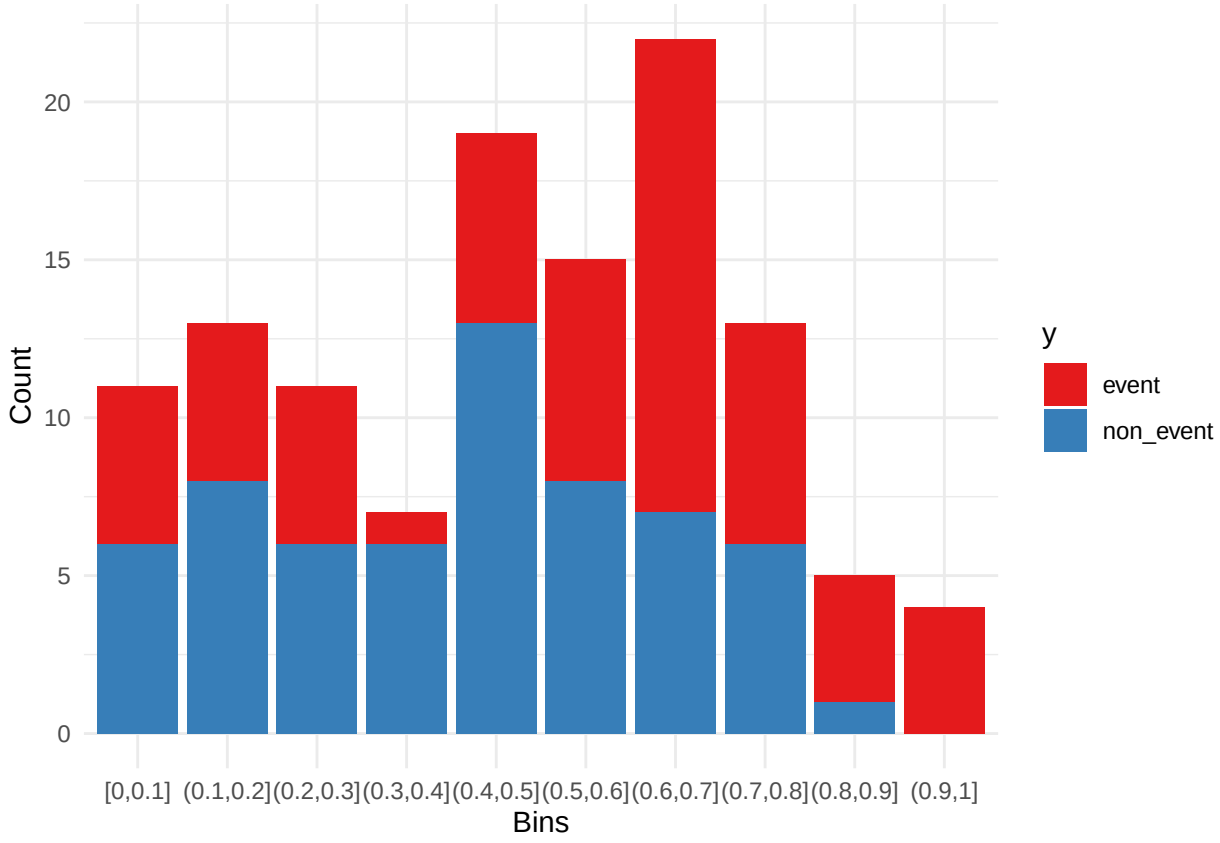
HINT: Which variable in the `test_df_1` tibble corresponds to the observed outcome?

SOLUTION What do you think?

1) "y" variable corresponds to the observed outcome.
2) The 95% bin consist of only events.
3) There are no bins consist of only non-events.

```
ggplot(test_df_1_b, aes(x=pred_bin, fill=y)) +
  geom_bar() +
  labs(x="Bins", y="Count") +
```

```
  scale_fill_brewer(palette="Set1") +
  theme_minimal()
```



6g)

Instead of showing the counts within each bin, let's change the bar chart so the maximum height is 1. The stacked bar chart will therefore show the proportion of events and non-events within each predicted probability bin.

Recreate the stacked bar chart from the previous problem, but this time set the `position` argument to `'fill'` within the `geom_bar()` layer. The `position` argument should be specified outside the `aes()` function with `geom_bar()`. You must continue to map the `fill` aesthetic to the observed outcome.

Are the empirical proptions of the event consistent with the predicted probability for model 1?

HINT: Pay close attention to the bar chart colors!

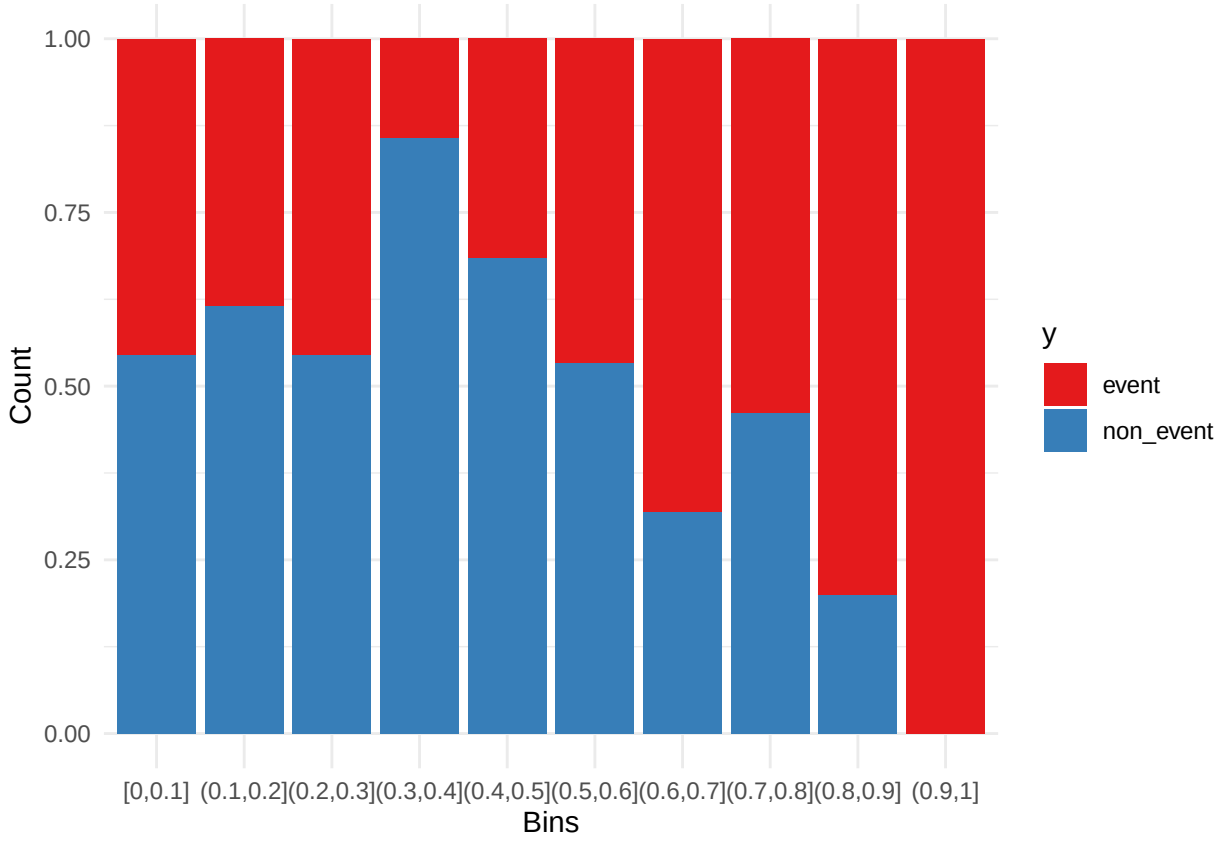
SOLUTION What do you think?

1) For these bins the empirical proptions of the event are looking consistent with the predicted probability for model 1: 95%, 85%, 65%, 55%.

2) For these bins the empirical proptions of the event are NOT looking consistent with the predicted probability for model 1: 5%, 15%, 25%, 35%, 45%, 75%.

```
ggplot(test_df_1_b, aes(x=pred_bin, fill=y)) +
  geom_bar(position='fill') +
  labs(x="Bins", y="Count") +
```

```
  scale_fill_brewer(palette="Set1") +
  theme_minimal()
```



Problem 07

The bar chart visualized in Problem 06 is the calibration curve for model 1, but that is not how calibration curves are typically displayed. As shown in lecture, calibration curves are visualized as scatter plots with lines connecting the markers. The previous problem was used to demonstrate the proportion of the event per bin. In this problem, you will calculate the proportion of the event manually in each bin.

This problem is open ended. You are free to calculate the event proportions however you want, as long as you do **NOT** use existing calibration curve functions. You are only allowed to use functions within `dplyr` and `ggplot2` in this problem.

7a)

Calculate the empirical proportion of events within each predicted probability bin associated with model 1. Although you are free to perform the calculations however you like, the results should be stored in a tibble (dataframe) named `my_calcurve_1` object. Your object should have columns for the `pred_bin`, the event proportion in the bin named `prop_event`, and the midpoint of the bin named `mid_bin`. Your `my_calcurve_1` object can have other columns, but those three columns are required.

SOLUTION Add as many code chunks as you feel are necessary.

```
### create prop_event column
bin_event_c <- numeric(10)
for (i in 1:nrow(test_df_1_b)) {
```

```
  if(test_df_1_b$y[i] == "event" && test_df_1_b$pred_bin[i]=="[0,0.1]") {bin_event_c[1]=bin_event_c[1]+
}
for (i in 1:nrow(test_df_1_b)) {
  if(test_df_1_b$y[i] == "event" && test_df_1_b$pred_bin[i]=="(0.1,0.2]") {bin_event_c[2]=bin_event_c[2]
}
for (i in 1:nrow(test_df_1_b)) {
  if(test_df_1_b$y[i] == "event" && test_df_1_b$pred_bin[i]=="(0.2,0.3]") {bin_event_c[3]=bin_event_c[3]
}
for (i in 1:nrow(test_df_1_b)) {
  if(test_df_1_b$y[i] == "event" && test_df_1_b$pred_bin[i]=="(0.3,0.4]") {bin_event_c[4]=bin_event_c[4]
}
for (i in 1:nrow(test_df_1_b)) {
  if(test_df_1_b$y[i] == "event" && test_df_1_b$pred_bin[i]=="(0.4,0.5]") {bin_event_c[5]=bin_event_c[5]
}
for (i in 1:nrow(test_df_1_b)) {
  if(test_df_1_b$y[i] == "event" && test_df_1_b$pred_bin[i]=="(0.5,0.6]") {bin_event_c[6]=bin_event_c[6]
}
for (i in 1:nrow(test_df_1_b)) {
  if(test_df_1_b$y[i] == "event" && test_df_1_b$pred_bin[i]=="(0.6,0.7]") {bin_event_c[7]=bin_event_c[7]
}
for (i in 1:nrow(test_df_1_b)) {
  if(test_df_1_b$y[i] == "event" && test_df_1_b$pred_bin[i]=="(0.7,0.8]") {bin_event_c[8]=bin_event_c[8]
}
for (i in 1:nrow(test_df_1_b)) {
  if(test_df_1_b$y[i] == "event" && test_df_1_b$pred_bin[i]=="(0.8,0.9]") {bin_event_c[9]=bin_event_c[9]
}
for (i in 1:nrow(test_df_1_b)) {
  if(test_df_1_b$y[i] == "event" && test_df_1_b$pred_bin[i]=="(0.9,1]") {bin_event_c[10]=bin_event_c[10]
}
bin_prop <- numeric(10)
for (i in 1:10) {
  bin_prop[i]=bin_event_c[i]/result6d$n[i]}

### Add prop_event column to my_calcurve_1
my_calcurve_1 <- result6d %>%
  mutate(prop_event = bin_prop)

### Create mid_bin column
bin_m <- numeric(10)
for (i in 1:10) {
  bin_m[i]=(i-0.5)/10}

### Add mid_bin column to my_calcurve_1
my_calcurve_1 <- my_calcurve_1 %>%
  mutate(mid_bin = bin_m)

### Display my_calcurve_1 in the final form
my_calcurve_1 <- my_calcurve_1 %>% select(-n)
my_calcurve_1
```

```
## # A tibble: 10 x 3
##    pred_bin  prop_event mid_bin
##    <fct>          <dbl>   <dbl>
##  1 [0,0.1]        0.455    0.05
```

```
##  2 (0.1,0.2]      0.385    0.15
##  3 (0.2,0.3]      0.455    0.25
##  4 (0.3,0.4]      0.143    0.35
##  5 (0.4,0.5]      0.316    0.45
##  6 (0.5,0.6]      0.467    0.55
##  7 (0.6,0.7]      0.682    0.65
##  8 (0.7,0.8]      0.538    0.75
##  9 (0.8,0.9]      0.8      0.85
## 10 (0.9,1]        1        0.95
```

7b)

You will now create the calibration curve associated with model 1's predictions on the hold-out test set! You must use the `my_calcurve_1` object created in Problem 7a).

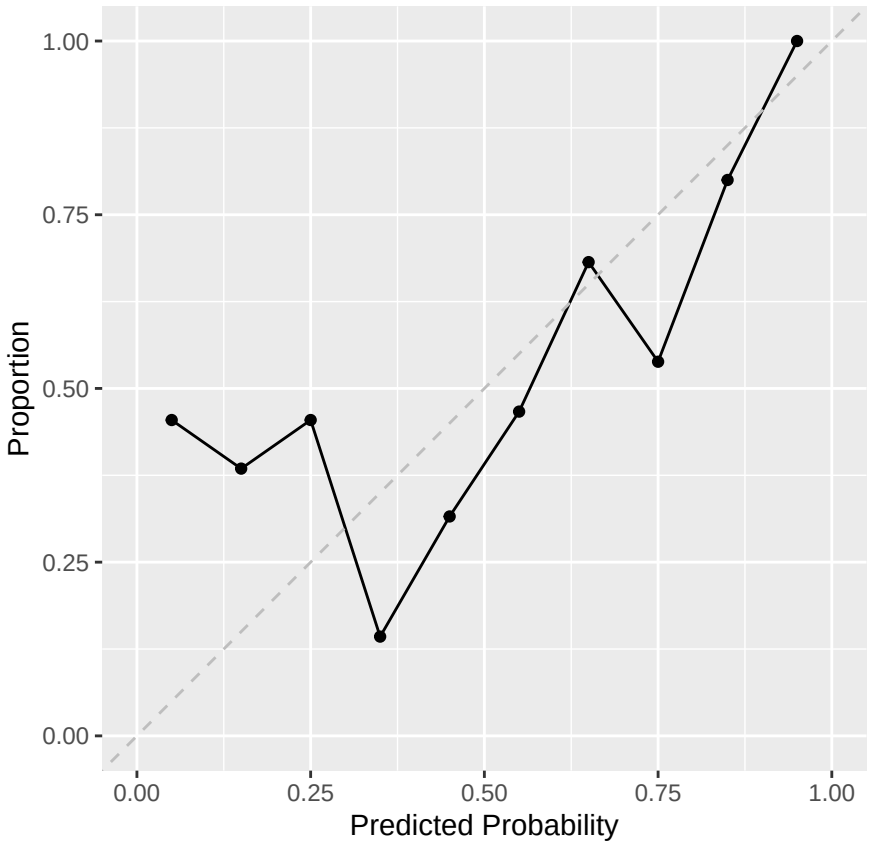
Pipe the `my_calcurve_1` object to `ggplot()` and map `mid_bin` and `prop_event` to the `x` and `y` aesthetics, respectively. Include `geom_line()` and `geom_point()` layers and a `geom_abline()` layer. Set the `geom_abline()` arguments as `slope=1`, `intercept=0`, `color='grey'`, and `linetype='dashed'`. Include `coord_equal()` with `xlim = c(0,1)` and `ylim = c(0,1)`.

Is the calibration curve consistent with your stacked filled barchart created previously? Pay close attention to the colors in the bar chart when comparing them!

SOLUTION What do you think?

Yes, the calibration curve consistent with the stacked filled barchart I created previously. For example, in bin (0.9,1] the bardchart was full of red (events), and in the calibration curve, corresponding proportion for 0.95 prediction is 1 (as expected).

```
ggplot(my_calcurve_1, aes(x=mid_bin,y=prop_event)) +
  geom_line() +
  geom_point() +
  geom_abline(intercept=0,slope=1, color = 'grey', linetype='dashed') +
  coord_equal(xlim = c(0,1), ylim = c(0,1)) +
  labs(x="Predicted Probability", y="Proportion")
```



7c)

Now it's time to create the necessary objects associated with model 3 and model 8. As we started with model 1, we must predict the hold-out test set and return the predicted event probabilities.

Predict the hold-out test set using model 3 and model 8. Assign the results to the variables `pred_test_3` and `pred_test_8` for model 3 and model 8, respectively.

```
pred_test_3 <- predict(mod_3_roc,newdata = df_test,type = 'prob')
pred_test_8 <- predict(mod_8_roc,newdata = df_test,type = 'prob')
```

SOLUTION

7d)

The code chunk below is completed for you. The model 3 and model 8 predictions are combined with the hold-out test set data so you have the predicted probabilities and observed outcome within a tibble (dataframe). **PLEASE NOTE**: the code chunk below is **NOT** evaluated by default. You must change the `eval` chunk option to make sure the code chunk is executed when you render the report.

```
test_df_3 <- df_test %>% bind_cols(pred_test_3)
test_df_8 <- df_test %>% bind_cols(pred_test_8)
```

You must create the categorical predicted probability bins for model 3 and model 8 using the `cut()` function. Assign the results to the `test_df_3_b` and `test_df_8_b` objects for model 3 and model 8, respectively.

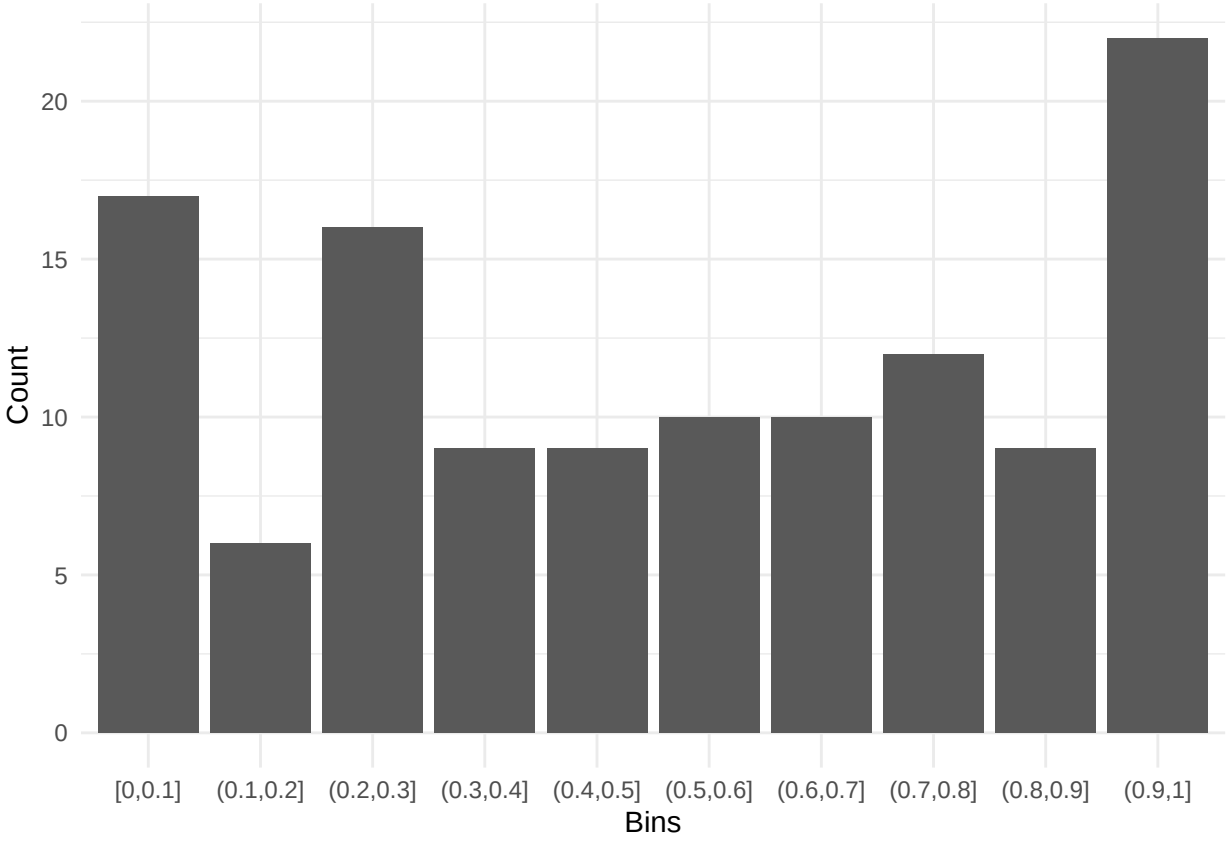
```r
test_df_3_b <- test_df_3 %>%
  mutate(pred_bin = cut(event, breaks = seq(0, 1, by = 0.1), include.lowest = TRUE))
test_df_8_b <- test_df_8 %>%
  mutate(pred_bin = cut(event, breaks = seq(0, 1, by = 0.1), include.lowest = TRUE))
```

SOLUTION

7e)

Use ggplot2 to visualize the number of observations per bin for model 3 with a bar chart.

```r
result7e <- test_df_3_b %>%
  count(pred_bin)
ggplot(result7e, aes(x=pred_bin, y=n)) +
  geom_bar(stat = "identity") +
  labs(x ="Bins", y="Count") +
  theme_minimal()
```



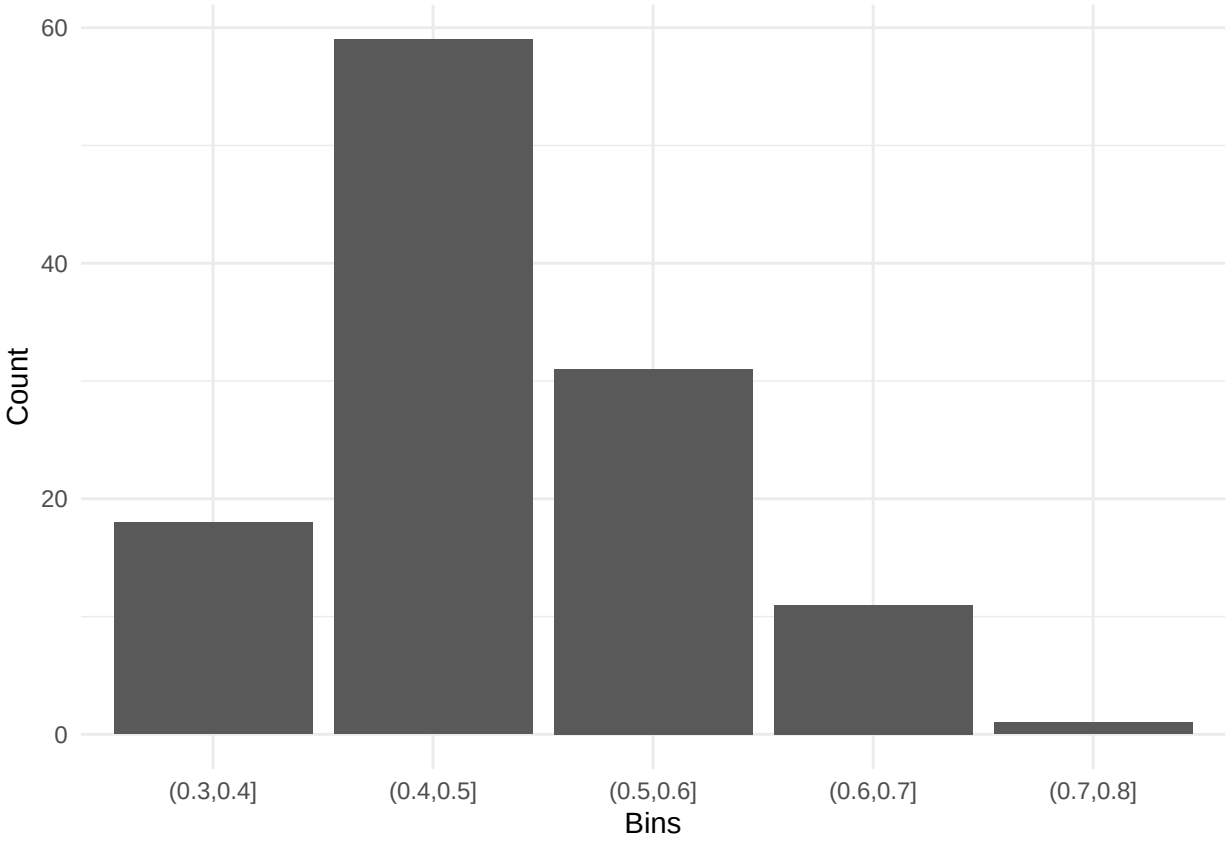
SOLUTION

7f)

Use ggplot2 to visualize the number of observations per bin for model 8 with a bar chart.

```r
result7f <- test_df_8_b %>%
  count(pred_bin)
```

```
ggplot(result7f, aes(x=pred_bin, y=n)) +
  geom_bar(stat = "identity") +
  labs(x ="Bins", y="Count") +
  theme_minimal()
```



SOLUTION

Problem 08

Rather than using the stacked filled bar charts to represent the calibration curve for models 3 and 8, let's jump straight to creating the calibration curve object for the two models.

8a)

Calculate the empirical proportion of events within each predicted probability bin associated with model 3. Although you are free to perform the calculations however you like, the results should be stored in a tibble (dataframe) named `my_calcurve_3` object. Your object should have columns for the `pred_bin`, the event proportion in the bin named `prop_event`, and the midpoint of the bin named `mid_bin`. Your `my_calcurve_3` object can have other columns, but those three columns are required.

SOLUTION Add as many code chunks as you feel are necessary.

```
### create prop_event column
bin_event_c3 <- numeric(10)
for (i in 1:nrow(test_df_3_b)) {
  if(test_df_3_b$y[i] == "event" && test_df_3_b$pred_bin[i]=="[0,0.1]") {bin_event_c3[1]=bin_event_c3[1]
}
for (i in 1:nrow(test_df_3_b)) {
```

```
  if(test_df_3_b$y[i] == "event" && test_df_3_b$pred_bin[i]=="(0.1,0.2]") {bin_event_c3[2]=bin_event_c3
}
for (i in 1:nrow(test_df_3_b)) {
  if(test_df_3_b$y[i] == "event" && test_df_3_b$pred_bin[i]=="(0.2,0.3]") {bin_event_c3[3]=bin_event_c3
}
for (i in 1:nrow(test_df_3_b)) {
  if(test_df_3_b$y[i] == "event" && test_df_3_b$pred_bin[i]=="(0.3,0.4]") {bin_event_c3[4]=bin_event_c3
}
for (i in 1:nrow(test_df_3_b)) {
  if(test_df_3_b$y[i] == "event" && test_df_3_b$pred_bin[i]=="(0.4,0.5]") {bin_event_c3[5]=bin_event_c3
}
for (i in 1:nrow(test_df_3_b)) {
  if(test_df_3_b$y[i] == "event" && test_df_3_b$pred_bin[i]=="(0.5,0.6]") {bin_event_c3[6]=bin_event_c3
}
for (i in 1:nrow(test_df_3_b)) {
  if(test_df_3_b$y[i] == "event" && test_df_3_b$pred_bin[i]=="(0.6,0.7]") {bin_event_c3[7]=bin_event_c3
}
for (i in 1:nrow(test_df_3_b)) {
  if(test_df_3_b$y[i] == "event" && test_df_3_b$pred_bin[i]=="(0.7,0.8]") {bin_event_c3[8]=bin_event_c3
}
for (i in 1:nrow(test_df_3_b)) {
  if(test_df_3_b$y[i] == "event" && test_df_3_b$pred_bin[i]=="(0.8,0.9]") {bin_event_c3[9]=bin_event_c3
}
for (i in 1:nrow(test_df_3_b)) {
  if(test_df_3_b$y[i] == "event" && test_df_3_b$pred_bin[i]=="(0.9,1]") {bin_event_c3[10]=bin_event_c3[
}
bin_prop3 <- numeric(10)
for (i in 1:10) {
  bin_prop3[i]=bin_event_c3[i]/result7e$n[i]}

### Add prop_event column to my_calcurve_3
my_calcurve_3 <- result7e %>%
  mutate(prop_event = bin_prop3)

### Create mid_bin column
bin_m3 <- numeric(10)
for (i in 1:10) {
  bin_m3[i]=(i-0.5)/10}

### Add mid_bin column to my_calcurve_3
my_calcurve_3 <- my_calcurve_3 %>%
  mutate(mid_bin = bin_m3)

### Display my_calcurve_3 in the final form
my_calcurve_3 <- my_calcurve_3 %>% select(-n)
my_calcurve_3
```

```
## # A tibble: 10 x 3
##    pred_bin  prop_event mid_bin
##    <fct>          <dbl>   <dbl>
##  1 [0,0.1]        0.118    0.05
##  2 (0.1,0.2]      0.167    0.15
##  3 (0.2,0.3]      0.312    0.25
##  4 (0.3,0.4]      0.333    0.35
```

```
##  5 (0.4,0.5]      0.111    0.45
##  6 (0.5,0.6]      0.6      0.55
##  7 (0.6,0.7]      0.6      0.65
##  8 (0.7,0.8]      0.583    0.75
##  9 (0.8,0.9]      0.778    0.85
## 10 (0.9,1]        0.955    0.95
```

8b)

Calculate the empirical propotion of events within each predicted probability bin associated with model 8. Although you are free to perform the calculations however you like, the results should be stored in a tibble (dataframe) named `my_calcurve_8` object. Your object should have columns for the `pred_bin`, the event proportion in the bin named `prop_event`, and the midpoint of the bin named `mid_bin`. Your `my_calcurve_8` object can have other columns, but those three columns are required.

SOLUTION Add as many code chunks as you feel are necessary.

```
### create prop_event column
bin_event_c8 <- numeric(10)
for (i in 1:nrow(test_df_8_b)) {
  if(test_df_8_b$y[i] == "event" && test_df_8_b$pred_bin[i]=="[0,0.1]") {bin_event_c8[1]=bin_event_c8[1]
}
for (i in 1:nrow(test_df_8_b)) {
  if(test_df_8_b$y[i] == "event" && test_df_8_b$pred_bin[i]=="(0.1,0.2]") {bin_event_c8[2]=bin_event_c8
}
for (i in 1:nrow(test_df_8_b)) {
  if(test_df_8_b$y[i] == "event" && test_df_8_b$pred_bin[i]=="(0.2,0.3]") {bin_event_c8[3]=bin_event_c8
}
for (i in 1:nrow(test_df_8_b)) {
  if(test_df_8_b$y[i] == "event" && test_df_8_b$pred_bin[i]=="(0.3,0.4]") {bin_event_c8[4]=bin_event_c8
}
for (i in 1:nrow(test_df_8_b)) {
  if(test_df_8_b$y[i] == "event" && test_df_8_b$pred_bin[i]=="(0.4,0.5]") {bin_event_c8[5]=bin_event_c8
}
for (i in 1:nrow(test_df_8_b)) {
  if(test_df_8_b$y[i] == "event" && test_df_8_b$pred_bin[i]=="(0.5,0.6]") {bin_event_c8[6]=bin_event_c8
}
for (i in 1:nrow(test_df_8_b)) {
  if(test_df_8_b$y[i] == "event" && test_df_8_b$pred_bin[i]=="(0.6,0.7]") {bin_event_c8[7]=bin_event_c8
}
for (i in 1:nrow(test_df_8_b)) {
  if(test_df_8_b$y[i] == "event" && test_df_8_b$pred_bin[i]=="(0.7,0.8]") {bin_event_c8[8]=bin_event_c8
}
for (i in 1:nrow(test_df_8_b)) {
  if(test_df_8_b$y[i] == "event" && test_df_8_b$pred_bin[i]=="(0.8,0.9]") {bin_event_c8[9]=bin_event_c8
}
for (i in 1:nrow(test_df_8_b)) {
  if(test_df_8_b$y[i] == "event" && test_df_8_b$pred_bin[i]=="(0.9,1]") {bin_event_c8[10]=bin_event_c8[
}
bin_prop8 <- numeric(5)
for (i in 1:5) {
  bin_prop8[i]=bin_event_c8[i+3]/result7f$n[i]}

### Add prop_event column to my_calcurve_8
```

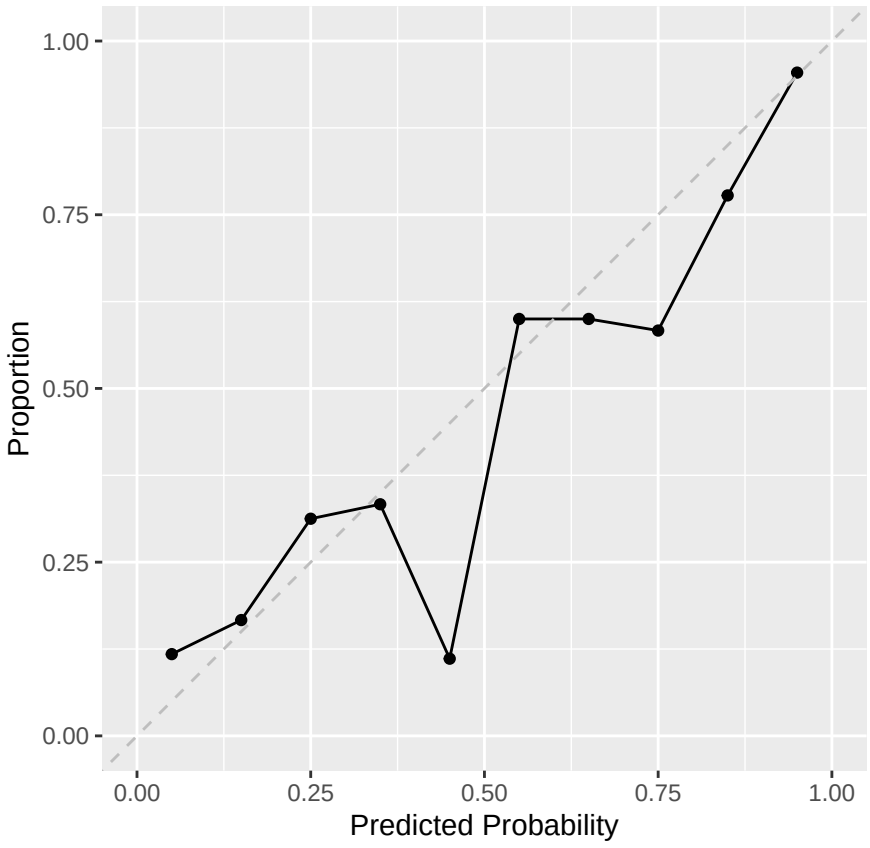
```
my_calcurve_8 <- result7f %>%
  mutate(prop_event = bin_prop8)

### Create mid_bin column
bin_m8 <- numeric(5)
for (i in 1:5) {
  bin_m8[i]=(i+3-0.5)/10}

### Add mid_bin column to my_calcurve_8
my_calcurve_8 <- my_calcurve_8 %>%
  mutate(mid_bin = bin_m8)

### Add rows to my_calcurve_8 where prop_event = 0 to display the full calibration curve
my_calcurve_8 <- my_calcurve_8 %>% select(-n)
new_row1 <- tibble(pred_bin = "(0.2,0.3]", prop_event = 0, mid_bin=0.25)
my_calcurve_8 <- add_row(my_calcurve_8, .before = 1, !!!new_row1)
new_row2 <- tibble(pred_bin = "(0.1,0.2]", prop_event = 0, mid_bin=0.15)
my_calcurve_8 <- add_row(my_calcurve_8, .before = 1, !!!new_row2)
new_row3 <- tibble(pred_bin = "[0,0.1]", prop_event = 0, mid_bin=0.05)
my_calcurve_8 <- add_row(my_calcurve_8, .before = 1, !!!new_row3)
new_row4 <- tibble(pred_bin = "(0.8,0.9]", prop_event = 0, mid_bin=0.85)
my_calcurve_8 <- add_row(my_calcurve_8, .after = 8, !!!new_row4)
new_row5 <- tibble(pred_bin = "(0.9,1]", prop_event = 0, mid_bin=0.95)
my_calcurve_8 <- add_row(my_calcurve_8, .after = 9, !!!new_row5)

### Display my_calcurve_8 in the final form
my_calcurve_8
```

```
## # A tibble: 10 x 3
##    pred_bin  prop_event mid_bin
##    <chr>          <dbl>   <dbl>
##  1 [0,0.1]        0        0.05
##  2 (0.1,0.2]      0        0.15
##  3 (0.2,0.3]      0        0.25
##  4 (0.3,0.4]      0.444    0.35
##  5 (0.4,0.5]      0.492    0.45
##  6 (0.5,0.6]      0.484    0.55
##  7 (0.6,0.7]      0.636    0.65
##  8 (0.7,0.8]      0        0.75
##  9 (0.8,0.9]      0        0.85
## 10 (0.9,1]        0        0.95
```

8c)

Visualize the calibration curve for model 3 using the approach described in Problem 7b).

```
ggplot(my_calcurve_3, aes(x=mid_bin,y=prop_event)) +
  geom_line() +
  geom_point() +
  geom_abline(intercept=0,slope=1, color = 'grey', linetype='dashed') +
  coord_equal(xlim = c(0,1), ylim = c(0,1)) +
  labs(x="Predicted Probability", y="Proportion")
```

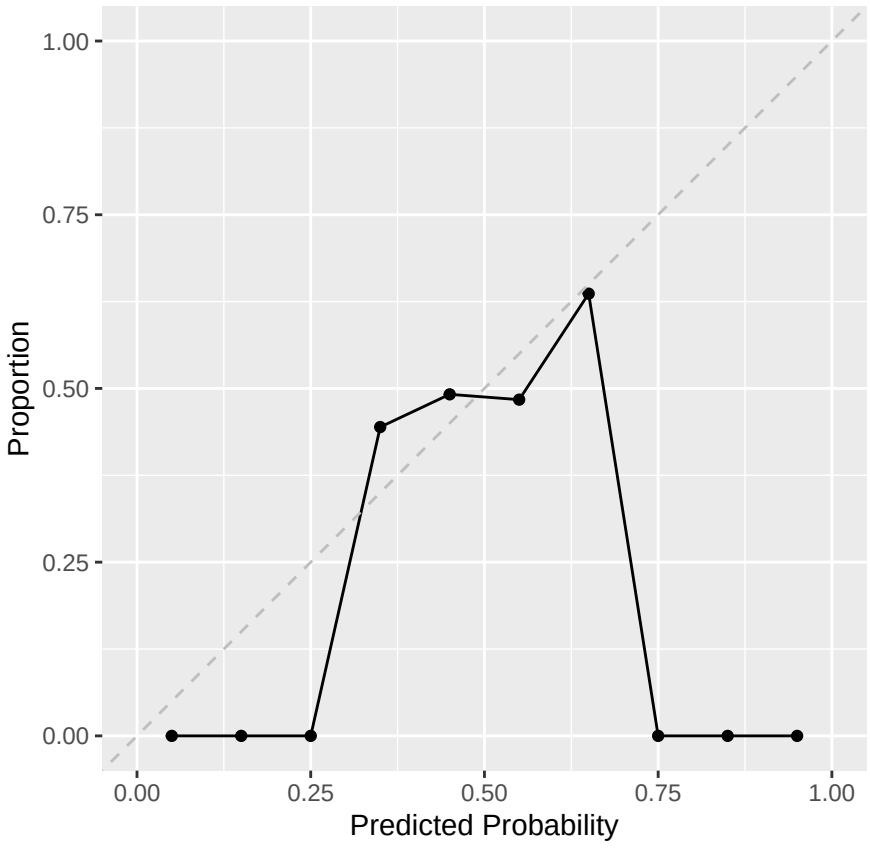


SOLUTION

8d)

Visualize the calibration curve for model 8 using the approach described in Problem 7b).

```
ggplot(my_calcurve_8, aes(x=mid_bin,y=prop_event)) +
  geom_line() +
  geom_point() +
  geom_abline(intercept=0,slope=1, color = 'grey', linetype='dashed') +
  coord_equal(xlim = c(0,1), ylim = c(0,1)) +
  labs(x="Predicted Probability", y="Proportion")
```



SOLUTION

8d)

Based on your calibration curves, which of the three models appears the most well calibrated? Is it easy obvious which model performs better using this approach? What are the most obvious aspects of performance based on your calibration curves?

SOLUTION What do you think?

1) The most well calibrated model is Model 3 because in this model the proportion values corresponding to predicted probability lie more close to y=x line compared to the other models.
2) It is easy to observe which model performs better using this approach. You just need to check which calibration curve is "close" to the line y=x.
3) Model 8 generates fewer bins. It means there are more samples per bin. Too few bins limits the ability to check model performance over wide range of conditions.

Problem 09

Calibration curves are created by executing many tedious steps. Although you had to perform those steps manually in this assignment there are existing functions which perform the necessary calculations for you. One such function is the `caret::calibration()` function. You will use that function in this problem to practice easily creating calibration curves. The `caret::calibration()` function works with resampled results, but for consistency with the previous problems, we will use with the hold-out test set predictions associated with the `df_test` data set.

The `caret::calibration()` function has 3 main arguments. The first argument is a formula, the second argument is the data set, and third is the number of bins to use. The formula follows a specific pattern:

`<binary output variable> ~ <event probability variable>`

The formula therefore gives the binary outcome variable name to the **left** of the tilde and the variable name for the event probability to the **right** of the tilde. These names **must** match the column names in the data set assigned to the `data` argument. The number of bins is specified by the `cuts` argument.

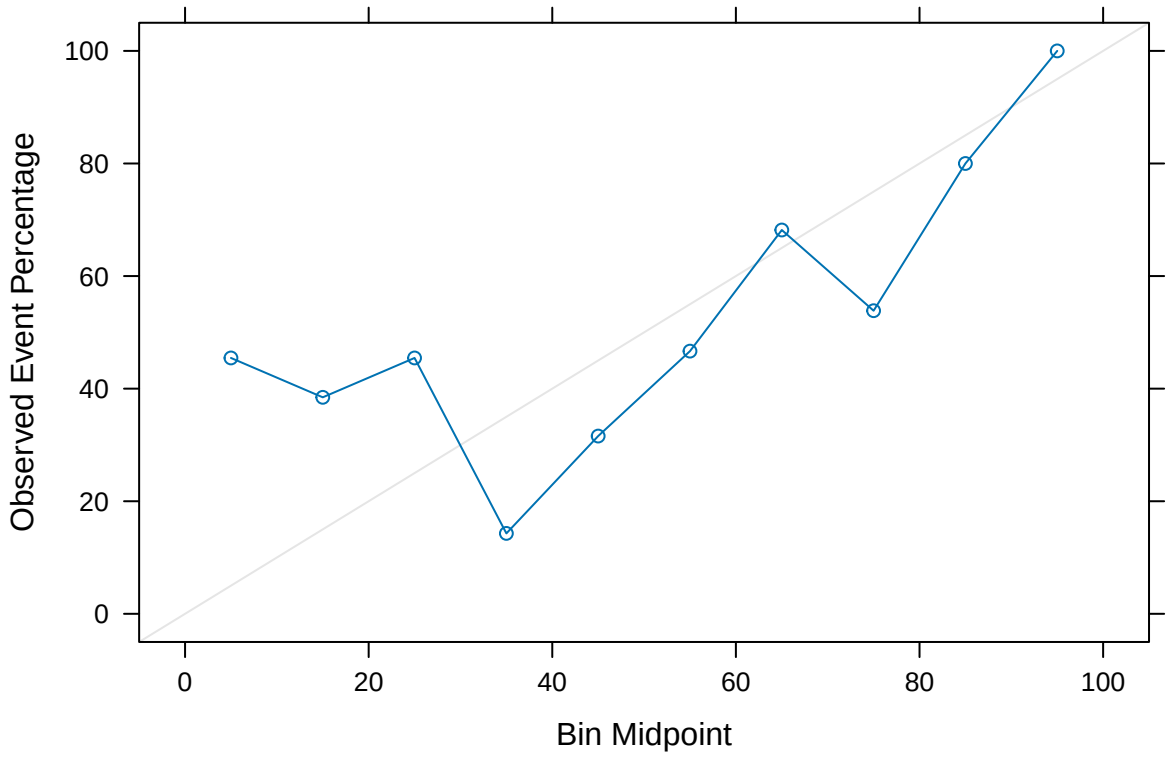
9a)

Create the calibration curve associated with model 1's hold-out test predictions using the `caret::calibration()` function. Specify the formula to be consistent with the `test_df_1` data set. Assign the `test_df_1` object to the `data` argument and assign `cuts = 10`. Pipe the result to the `xyplot()` function.

Is the created calibration curve consistent with your manually created curve from the previous problems?

SOLUTION Yes, the created calibration curve is consistent with my manually created curve from the previous problem.

```
calibration_curve1 <- calibration(y ~ event, test_df_1, cuts = 10)
xyplot(calibration_curve1)
```



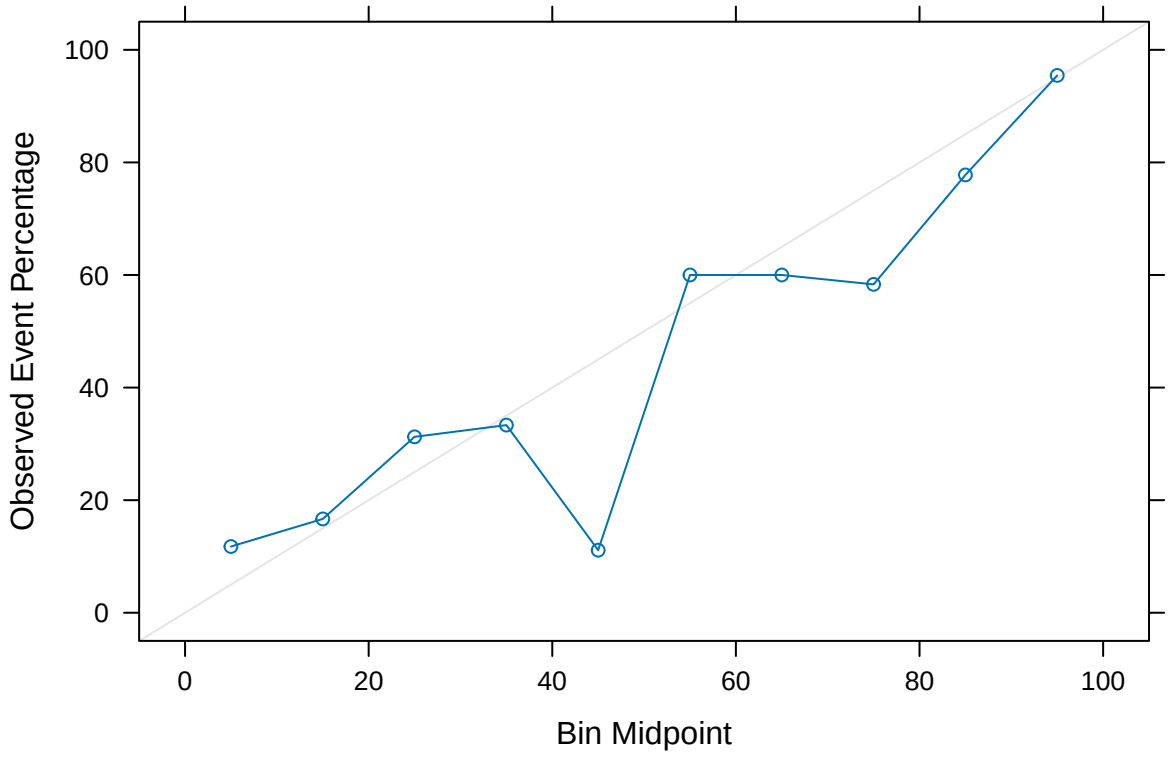
9b)

Create the calibration curve associated with model 3's hold-out test predictions using the `caret::calibration()` function. Specify the formula to be consistent with the `test_df_3` data set. Assign the `test_df_3` object to the `data` argument and assign `cuts = 10`. Pipe the result to the `xyplot()` function.

Is the created calibration curve consistent with your manually created curve from the previous problems?

SOLUTION Yes, the created calibration curve is consistent with my manually created curve from the previous problem.

```
calibration_curve3 <- calibration(y ~ event, test_df_3, cuts = 10)
xyplot(calibration_curve3)
```



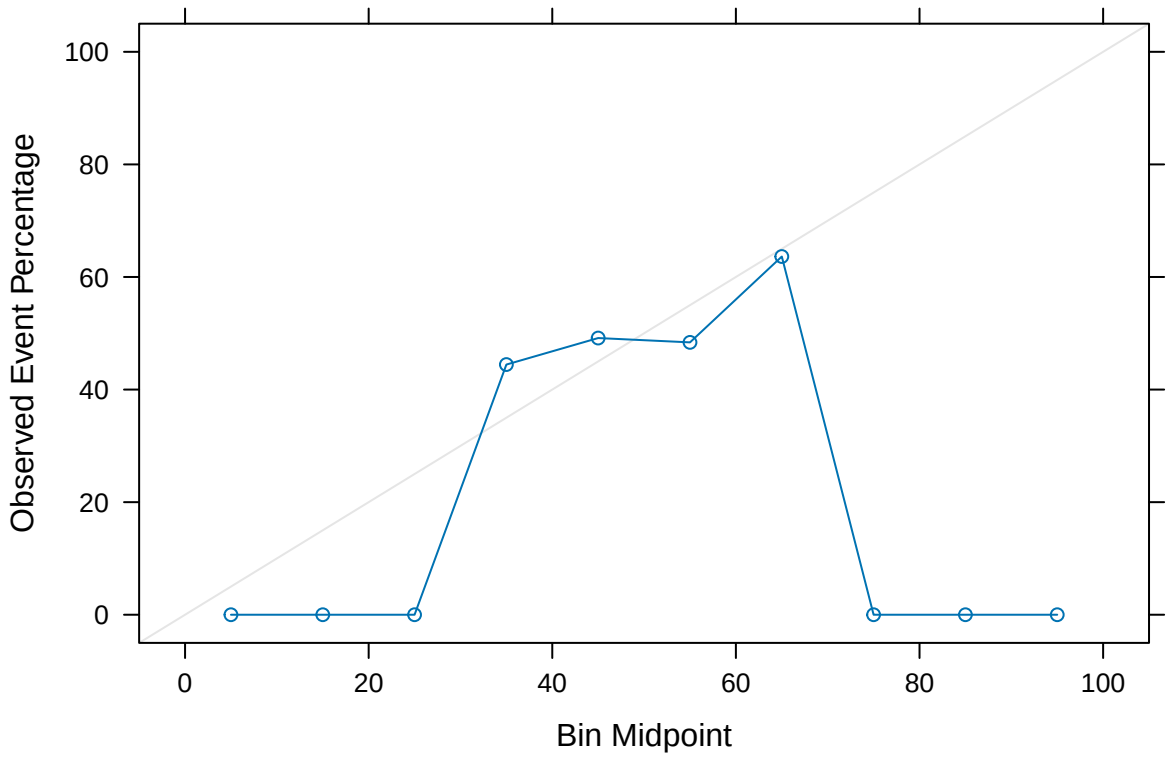
9c)

Create the calibration curve associated with model 8's hold-out test predictions using the `caret::calibration()` function. Specify the formula to be consistent with the `test_df_8` data set. Assign the `test_df_8` object to the `data` argument and assign `cuts = 10`. Pipe the result to the `xyplot()` function.

Is the created calibration curve consistent with your manually created curve from the previous problems?

SOLUTION Yes, the created calibration curve is consistent with my manually created curve from the previous problem.

```
calibration_curve8 <- calibration(y ~ event, test_df_8, cuts = 10)
xyplot(calibration_curve8)
```


Observed Event Percentage
100
80
60
40
20
0
0
20
40
60
80
100
Bin Midpoint