CS 1675 Spring 2022 Homework: 03

Assigned January 26, 2022; Due: February 2, 2022

Sameera Boppana

Submission time: February 2, 2022 at 11:00PM EST

Overview

In the previous assignment, you calculated important binary classification performance metrics by hand. You will now use existing functions from the `caret` and `yardstick` packages to train and assess the performance of multiple binary classifiers. The resampling and model evaluation will be managed by the `caret` package. You will consider several performance metrics and study what those metrics tell you about the model behavior. You will use functions from the `yardstick` package to help visualize the ROC curve. Please download and install `yardstick` before starting the assignment. You can do so directly or by downloading and installing `tidymodels` .

After evaluating performance based on Accuracy and the ROC curve you will also compare performance based on the **calibration curve**. You will first create the calibration curve manually before using existing functions from `caret` to generate the calibration curve automatically.

IMPORTANT: 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.

Load packages

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

```
library(tidyverse)
```

```
## ── Attaching packages ─────────────────────────────────────── tidyverse 1.3.1 ──
```

```
## ✓ ggplot2 3.3.5     ✓ purrr   0.3.4
## ✓ tibble  3.1.6     ✓ dplyr   1.0.7
## ✓ tidyr   1.1.4     ✓ stringr 1.4.0
## ✓ readr   2.1.1     ✓ forcats 0.5.1
```

```
## ── Conflicts ─────────────────────────────────────────── tidyverse_conflicts() ──
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Problem 01

The code chunk below reads in the data you will work with in the first two problems. 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`.

```
hw03_url <- 'https://raw.githubusercontent.com/jyurko/CS_1675_Spring_2022/main/HW/03/hw03_train_data.csv'

df <- readr::read_csv(hw03_url, 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,…
```

1a)

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 of the binary outcome 'y' is balanced. The response of "event" occurs just about 48%.

1b)

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.

```
# loading caret package
library(caret)
```

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

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

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

```
# loading yardstick package
library(yardstick)
```

```
## For binary classification, the first factor level is assumed to be the event.
## Use the argument `event_level = "second"` to alter this as needed.
```

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

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

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

1c)

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 = 10
)

metric_acc <- 'Accuracy'
```

1d)

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 approach is trying to identify the best possible set of inputs to use to model the binary outcome in a step-wise like fashion.

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.

SOLUTION

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

```
### 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, met
hod= "glm",family = "binomial", trControl = ctrl_acc, metric=metric_acc)
```

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

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

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

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

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

```
### model 8
set.seed(2021)
mod_8_acc <- train(y~x3+x4+I(x3^2)+ I(x4^2), data = df, method= "glm",family = "binom
ial", trControl = ctrl_acc, metric=metric_acc)
```

1e)

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.

SOLUTION

```
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))
```

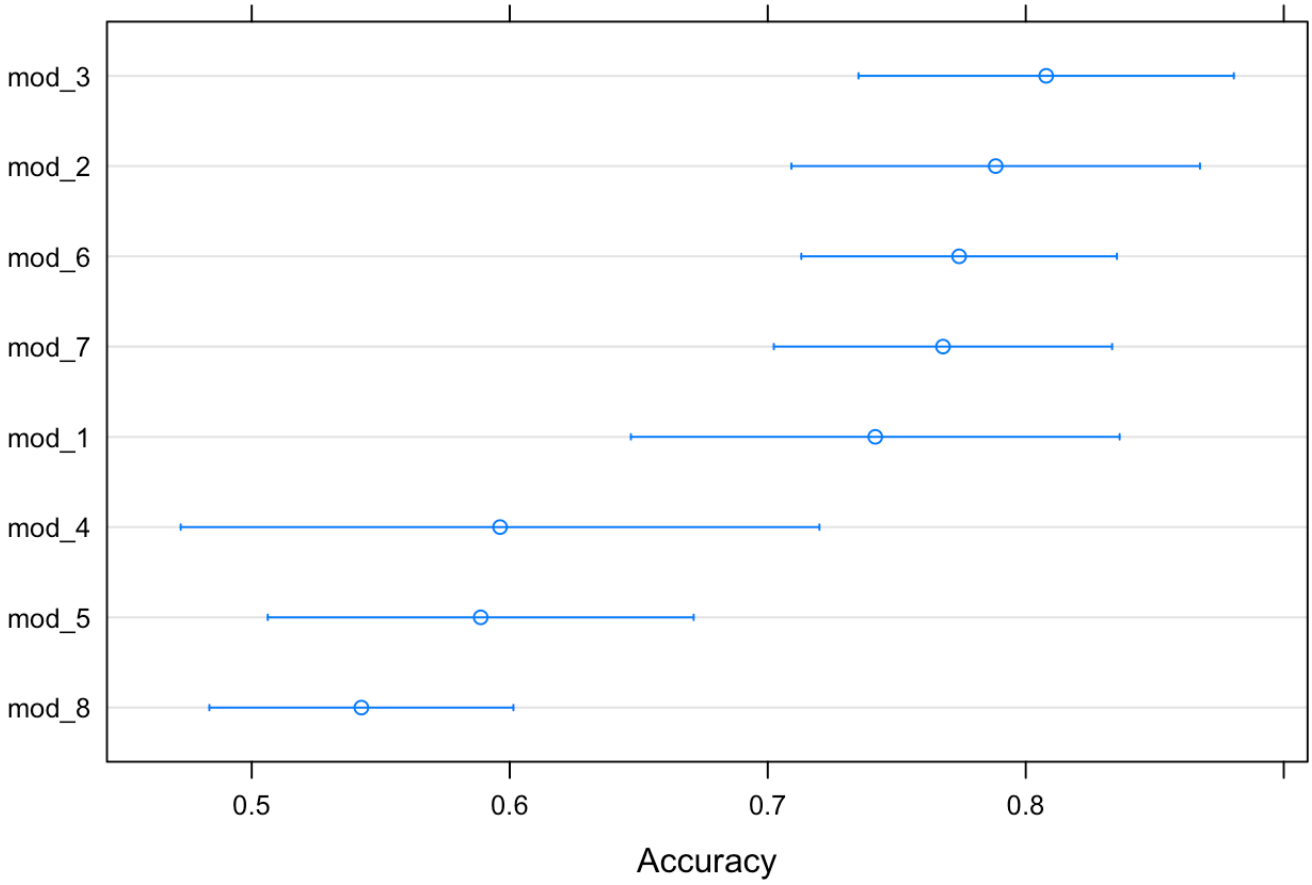
Summarize the results across the resamples.

```
### add your code here
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: 10
##
## Accuracy
##         Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## mod_1 0.6000 0.6666667 0.6875000 0.7416667 0.7875000 1.0000000    0
## mod_2 0.5625 0.7500000 0.8062500 0.7883333 0.8666667 0.9333333    0
## mod_3 0.6250 0.7500000 0.8125000 0.8079167 0.8729167 0.9333333    0
## mod_4 0.2500 0.5406250 0.5625000 0.5962500 0.7562500 0.8000000    0
## mod_5 0.4375 0.4906250 0.5812500 0.5887500 0.6822917 0.7333333    0
## mod_6 0.6250 0.7333333 0.7500000 0.7741667 0.8093750 0.9375000    0
## mod_7 0.6250 0.7333333 0.7500000 0.7679167 0.8093750 0.9333333    0
## mod_8 0.4375 0.4750000 0.5479167 0.5425000 0.6000000 0.6875000    0
```

Visualize the resample averaged Accuracy per model.

```
### add your code here
dotplot(acc_results, metric = metric_acc)
```



Confidence Level: 0.95

Which model is the best?
Model 3 is the best as it has the highest accuracy rate. ### 1f)

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 in 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.

```
#confusion matrix for top model
confusionMatrix(mod_3_acc)
```

```
## Cross-Validated (10 fold) Confusion Matrix
##
## (entries are percentual average cell counts across resamples)
##
##            Reference
## Prediction  event non_event
##   event      36.8       7.7
##   non_event  11.6      43.9
##
##  Accuracy (average) : 0.8065
```

```
#confusion matrix for second best model
confusionMatrix(mod_2_acc)
```

```
## Cross-Validated (10 fold) Confusion Matrix
##
## (entries are percentual average cell counts across resamples)
##
##            Reference
## Prediction  event non_event
##   event      36.1       9.0
##   non_event  12.3      42.6
##
##  Accuracy (average) : 0.7871
```

```
#confusion matrix for second worst model
confusionMatrix(mod_5_acc)
```

```
## Cross-Validated (10 fold) Confusion Matrix
##
## (entries are percentual average cell counts across resamples)
##
##            Reference
## Prediction  event non_event
##   event      29.0      21.9
##   non_event  19.4      29.7
##
##  Accuracy (average) : 0.5871
```

```
#confusion matrix for worst model
confusionMatrix(mod_8_acc)
```

```
## Cross-Validated (10 fold) Confusion Matrix
##
## (entries are percentual average cell counts across resamples)
##
##            Reference
## Prediction  event non_event
##   event      20.0      17.4
##   non_event  28.4      34.2
##
##  Accuracy (average) : 0.5419
```

Problem 02

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.

2a)

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 1c). 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.

SOLUTION

```
ctrl_roc <- trainControl( method = 'cv', number = 10,
                          summaryFunction = twoClassSummary,
                          classProbs = TRUE,
                          savePredictions = TRUE)

metric_roc <- "ROC"
```

2b)

You will retrain the same set of 8 models that you trained in Problem 1d), 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.

SOLUTION

```
### model 1
set.seed(2021)
mod_1_roc <- train(y~x1+x2+x3+x4, data = df, method= "glm",family = "binomial", trCon
trol = ctrl_roc, metric=metric_roc)
```

```
### 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, met
hod= "glm",family = "binomial", trControl = ctrl_roc, metric=metric_roc)
```

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

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

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

```
### model 6
set.seed(2021)
mod_6_roc <- train(y~x2+x3+ I(x2^2)+ I(x3^2), data = df, method= "glm",family = "bino
mial", trControl = ctrl_roc, metric=metric_roc)
```

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

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

2c)

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.

SOLUTION

```
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))
```

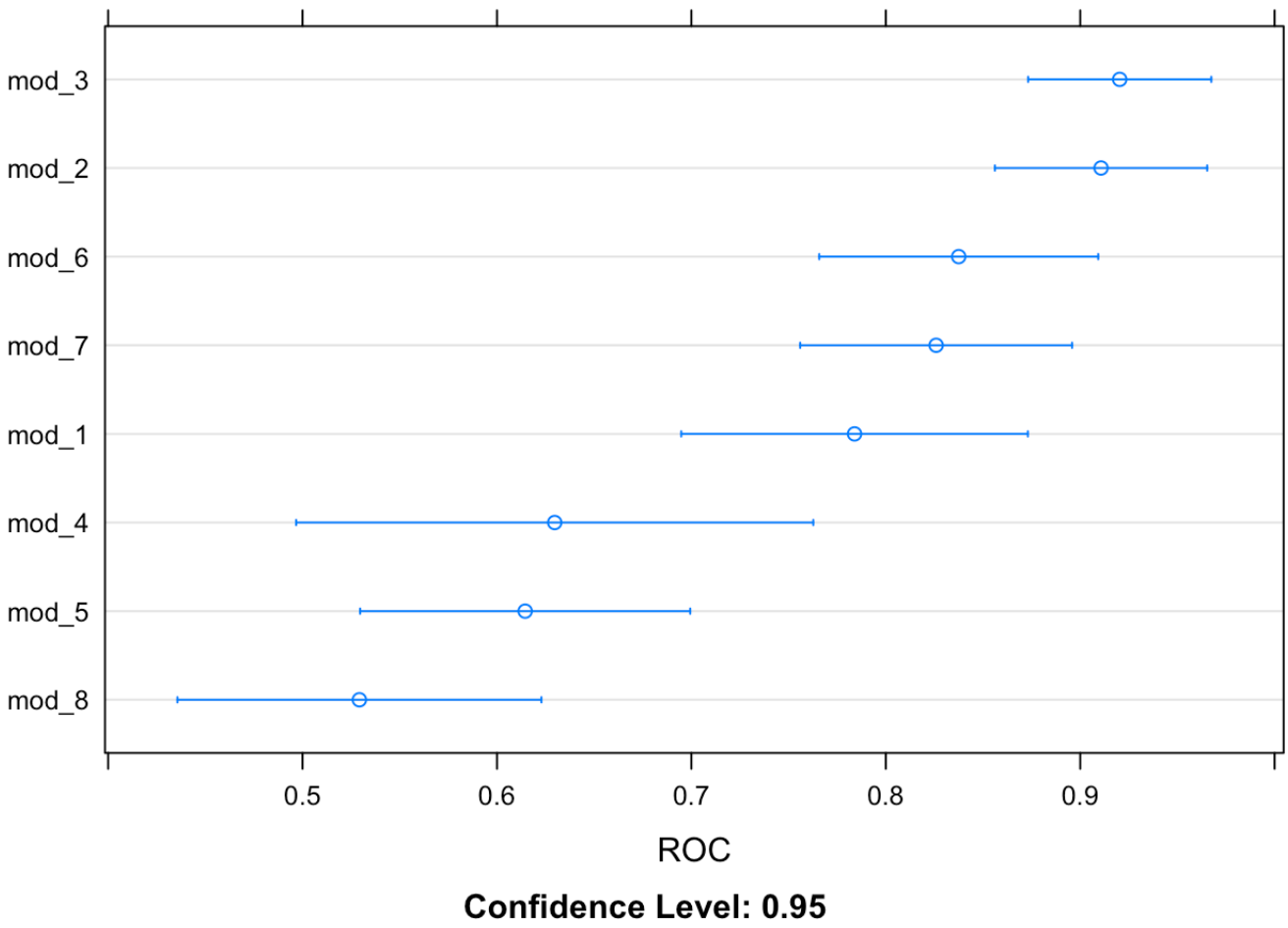
Summarize the results across the resamples.

```
### add your code here
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: 10
##
## ROC
##            Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## mod_1 0.5535714 0.7220982 0.7812500 0.7839286 0.8337054 1.0000000    0
## mod_2 0.7500000 0.8750000 0.9084821 0.9107143 0.9776786 1.0000000    0
## mod_3 0.7812500 0.8962054 0.9252232 0.9203125 0.9732143 1.0000000    0
## mod_4 0.2031250 0.5664062 0.6484375 0.6296875 0.7410714 0.8571429    0
## mod_5 0.3437500 0.5619420 0.6484375 0.6145089 0.7036830 0.7321429    0
## mod_6 0.7031250 0.7678571 0.8002232 0.8375000 0.9324777 0.9843750    0
## mod_7 0.6718750 0.7633929 0.8325893 0.8258929 0.8978795 0.9843750    0
## mod_8 0.2187500 0.4810268 0.5591518 0.5292411 0.6250000 0.6607143    0
```

Visualize the resample averaged ROC AUC per model.

```
### add your code here
dotplot(roc_results, metric = metric_roc)
```



Which model is the best?
Model 3 is the best model using ROC.

2d)

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?

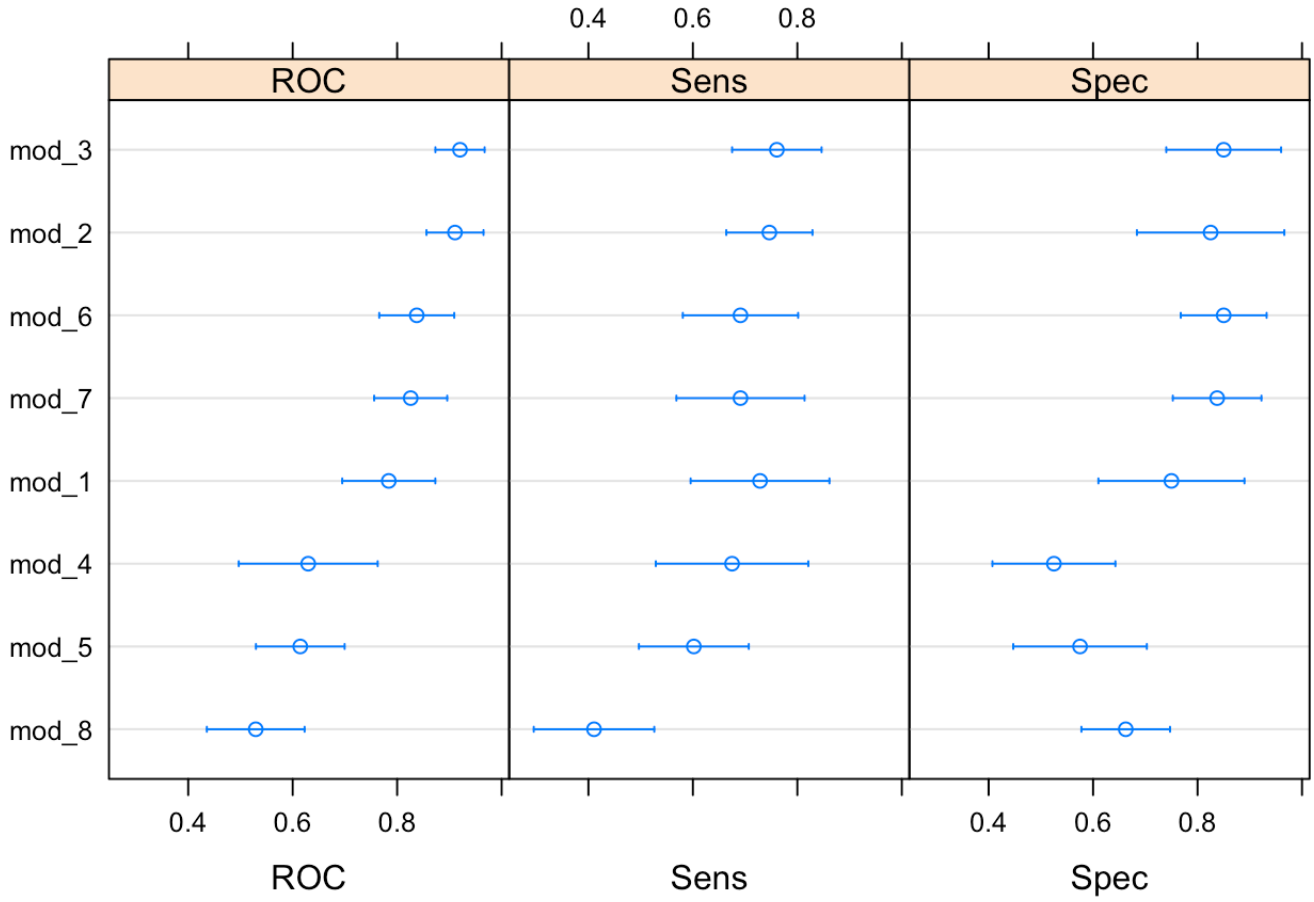
SOLUTION

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

```
#summary
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: 10
##
## ROC
##            Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## mod_1 0.5535714 0.7220982 0.7812500 0.7839286 0.8337054 1.0000000    0
## mod_2 0.7500000 0.8750000 0.9084821 0.9107143 0.9776786 1.0000000    0
## mod_3 0.7812500 0.8962054 0.9252232 0.9203125 0.9732143 1.0000000    0
## mod_4 0.2031250 0.5664062 0.6484375 0.6296875 0.7410714 0.8571429    0
## mod_5 0.3437500 0.5619420 0.6484375 0.6145089 0.7036830 0.7321429    0
## mod_6 0.7031250 0.7678571 0.8002232 0.8375000 0.9324777 0.9843750    0
## mod_7 0.6718750 0.7633929 0.8325893 0.8258929 0.8978795 0.9843750    0
## mod_8 0.2187500 0.4810268 0.5591518 0.5292411 0.6250000 0.6607143    0
##
## Sens
##            Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## mod_1 0.4285714 0.5848214 0.7410714 0.7285714 0.8705357 1.0000000    0
## mod_2 0.5714286 0.6473214 0.7321429 0.7464286 0.8571429 0.8750000    0
## mod_3 0.5714286 0.6473214 0.8035714 0.7607143 0.8571429 0.8750000    0
## mod_4 0.2500000 0.6250000 0.6250000 0.6750000 0.8214286 1.0000000    0
## mod_5 0.3750000 0.5178571 0.5982143 0.6017857 0.6919643 0.8571429    0
## mod_6 0.5000000 0.5714286 0.6696429 0.6910714 0.7500000 1.0000000    0
## mod_7 0.4285714 0.5714286 0.6875000 0.6910714 0.8571429 0.8750000    0
## mod_8 0.1428571 0.3080357 0.4017857 0.4107143 0.5357143 0.6250000    0
##
## Spec
##        Min. 1st Qu. Median   Mean 3rd Qu.  Max. NA's
## mod_1 0.500 0.62500  0.750 0.7500  0.9375 1.000    0
## mod_2 0.500 0.75000  0.875 0.8250  1.0000 1.000    0
## mod_3 0.625 0.75000  0.875 0.8500  1.0000 1.000    0
## mod_4 0.250 0.40625  0.500 0.5250  0.6250 0.750    0
## mod_5 0.250 0.50000  0.625 0.5750  0.6250 0.875    0
## mod_6 0.625 0.78125  0.875 0.8500  0.8750 1.000    0
## mod_7 0.625 0.75000  0.875 0.8375  0.8750 1.000    0
## mod_8 0.500 0.62500  0.625 0.6625  0.7500 0.875    0
```

```
#visualizing roc_results
dotplot(roc_results)
```



Confidence Level: 0.95

The model with the highest true positive rate at the default threshold is model 3. The model with the lowest false positive rate at the default threshold is model 3. Model 3 has the highest specificity, therefore has the lowest FPR as FPR = 1 - Specificity.

2e)

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 × 7
##    pred      obs        event non_event rowIndex parameter Resample
##    <fct>     <fct>      <dbl>     <dbl>    <int> <chr>     <chr>
##  1 non_event non_event 0.352      0.648        6 none      Fold01
##  2 event     event     0.540      0.460       16 none      Fold01
##  3 non_event non_event 0.286      0.714       34 none      Fold01
##  4 event     event     0.756      0.244       37 none      Fold01
##  5 non_event non_event 0.411      0.589       45 none      Fold01
##  6 event     event     0.735      0.265       53 none      Fold01
##  7 non_event event     0.0231     0.977       88 none      Fold01
##  8 non_event non_event 0.211      0.789       97 none      Fold01
##  9 non_event event     0.0808     0.919      107 none      Fold01
## 10 non_event event     0.483      0.517      117 none      Fold01
## # … with 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 do not need to create the ROC curve manually in this assignment. Instead you will use the `roc_curve()` function from the `yardstick` package. 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?

SOLUTION

```
### add your code here
mod_1_roc$pred %>%
  roc_curve(obs, event)
```

```
## # A tibble: 157 × 3
##    .threshold specificity sensitivity
##         <dbl>       <dbl>       <dbl>
##  1 -Inf            0           1
##  2    0.00354      0           1
##  3    0.0199       0           0.987
##  4    0.0231       0           0.973
##  5    0.0255       0           0.96
##  6    0.0305       0.0125      0.96
##  7    0.0416       0.0125      0.947
##  8    0.0550       0.025       0.947
##  9    0.0701       0.0375      0.947
## 10    0.0808       0.05        0.947
## # … with 147 more rows
```

What do you think?
The sensitivity has values at or near 1 when the threshold is so low because when the threshold is so low, it makes increases the sensitivity, making it easier to predict be an event, increasing the sensitivity.

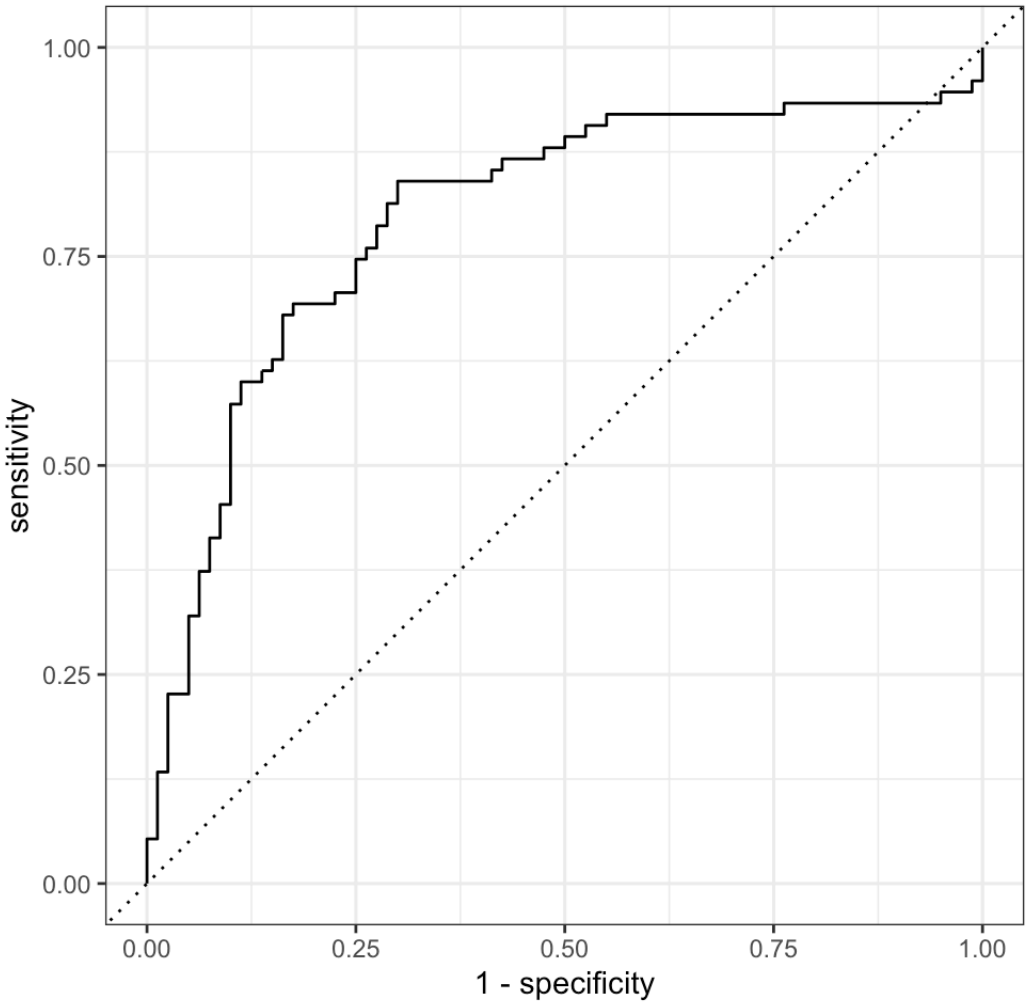
2f)

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

Repeat the same steps you performed in 2e) above, except pipe the result to the `autoplot()` method.

SOLUTION

```
### add your code here
mod_1_roc$pred %>%
  roc_curve(obs, event) %>%
  autoplot()
```



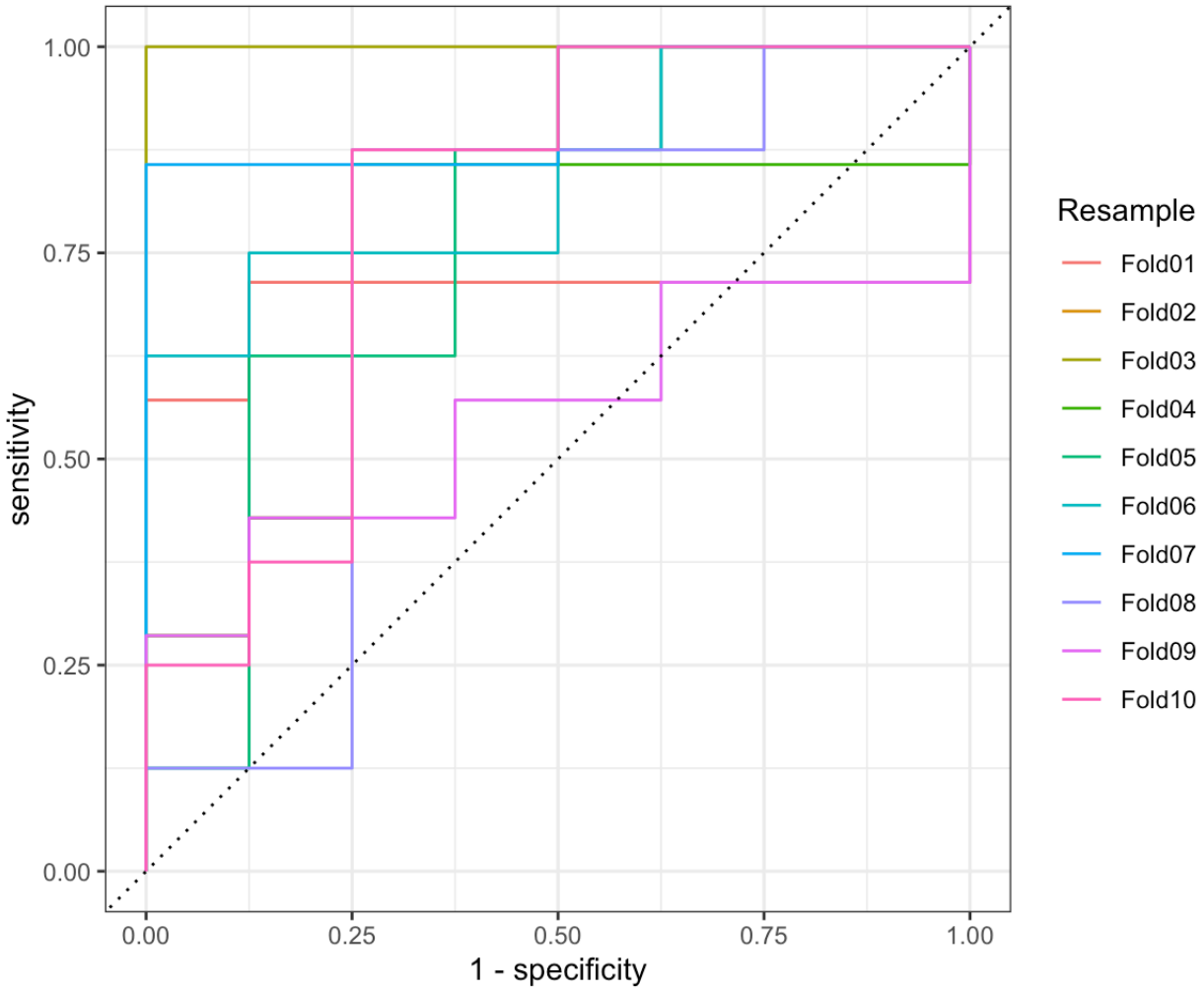
2g)

The ROC curve displayed in 2f) 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()`.

SOLUTION

```
mod_1_roc$pred %>%
  group_by(Resample) %>%
  roc_curve(obs, event) %>%
  autoplot()
```



2h)

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 × 4
##    obs        event Resample model_name
##    <fct>      <dbl> <chr>    <chr>
##  1 non_event 0.352  Fold01   1
##  2 event     0.540  Fold01   1
##  3 non_event 0.286  Fold01   1
##  4 event     0.756  Fold01   1
##  5 non_event 0.411  Fold01   1
##  6 event     0.735  Fold01   1
##  7 event     0.0231 Fold01   1
##  8 non_event 0.211  Fold01   1
##  9 event     0.0808 Fold01   1
## 10 event     0.483  Fold01   1
## # … with 1,230 more rows
```

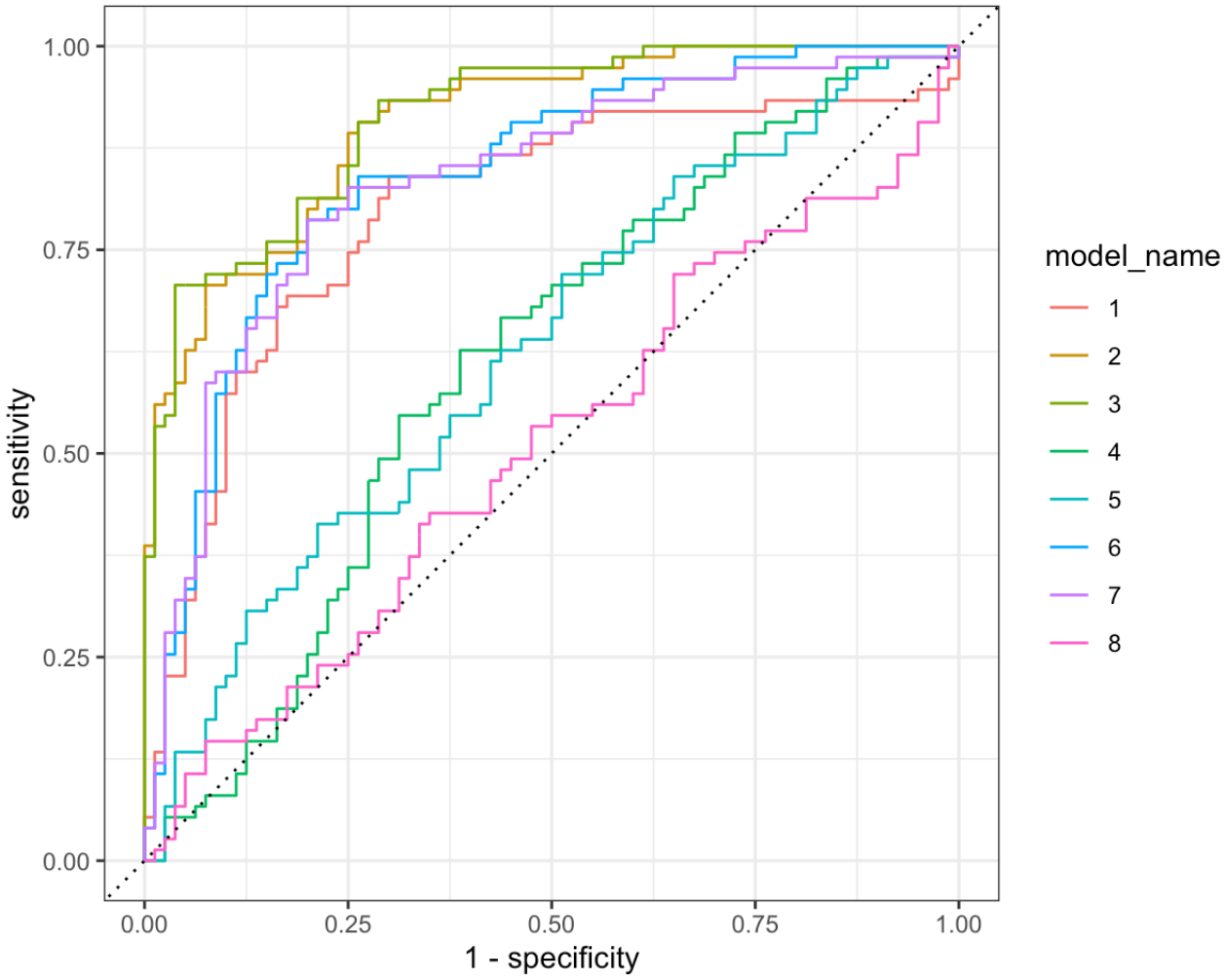
You will now create a figure which displays the resample averaged ROC curve for each model.

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?

SOLUTION

```
all_model_preds %>%
  group_by(model_name) %>%
  roc_curve(obs, event) %>%
  autoplot()
```



What do you think?
Model 3 appears to be the best because it has the closest curve to the ideal ROC curve which would be at the top left corner of the graph. This ideal ROC curve would correspond to an AUC value of 1, meaning that the TPR rate is the highest, with a FPR of zero. Both models 2 and 3 follow this trend very closely, but since the two models perform very similarly Model 3 might be a better choice as it less complex than model 2. This result is consistent with the ROC AUC summary metrics previously calculated. The model that is closest to completely ineffective model is model 1. Model 1 closely follows the reference 1, meaning that as the TPR increase, so does the FPR. Therefore, as the threshold changes, the errors are being switched, making the model not very effective.

Problem 03

In the previous assignment, you manually calculated the Accuracy, confusion matrix, and a highly simplifed ROC curve. You built upon that work in this assignment by using existing functions to calculate the performance metrics for you. 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.

```
hw03_test_url <- 'https://raw.githubusercontent.com/jyurko/CS_1675_Spring_2022/main/HW/03/hw03_test_data.csv'

df_test <- readr::read_csv(hw03_test_url, 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…
```

3a)

The first step to create the calibration curve requires making predictions. The `predict()` 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 predictioin 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.

SOLUTION

```
pred_test_1 <- predict(mod_1_acc, newdata = df_test, type = 'prob')
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
```

3b)

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?
The relationship between the values in the event column and the non_event column is that for each row, the sum is equal to 1. That is, the (probability of event) + (probability of non_event) = 1.

3c)

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 × 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.

SOLUTION

```
test_df_1_b <- test_df_1 %>%
  mutate(
    pred_bin = cut(test_df_1$event,
                   breaks = seq(0,1, by = 0.10),
                   include.lowest = TRUE)
  )
```

3d)

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?

SOLUTION

```
### your code here
test_df_1_b %>%
  count(pred_bin)
```

```
## # A tibble: 10 × 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
```

There are 10 unique values of pred_bin.

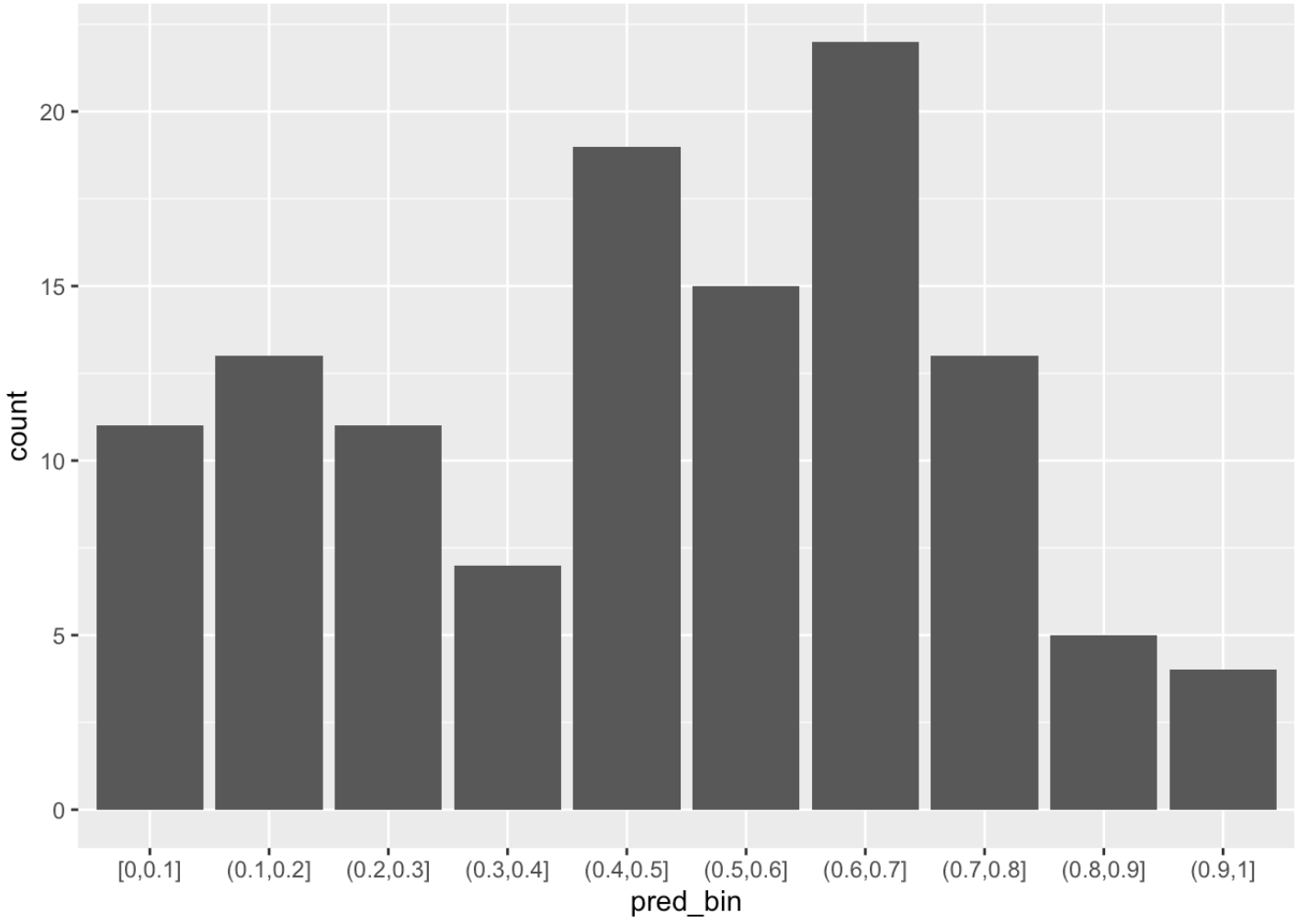
3e)

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`.

SOLUTION

```
### your code here
test_df_1_b %>%
    ggplot(mapping = aes(x = pred_bin)) + geom_bar()
```



3f)

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'`.

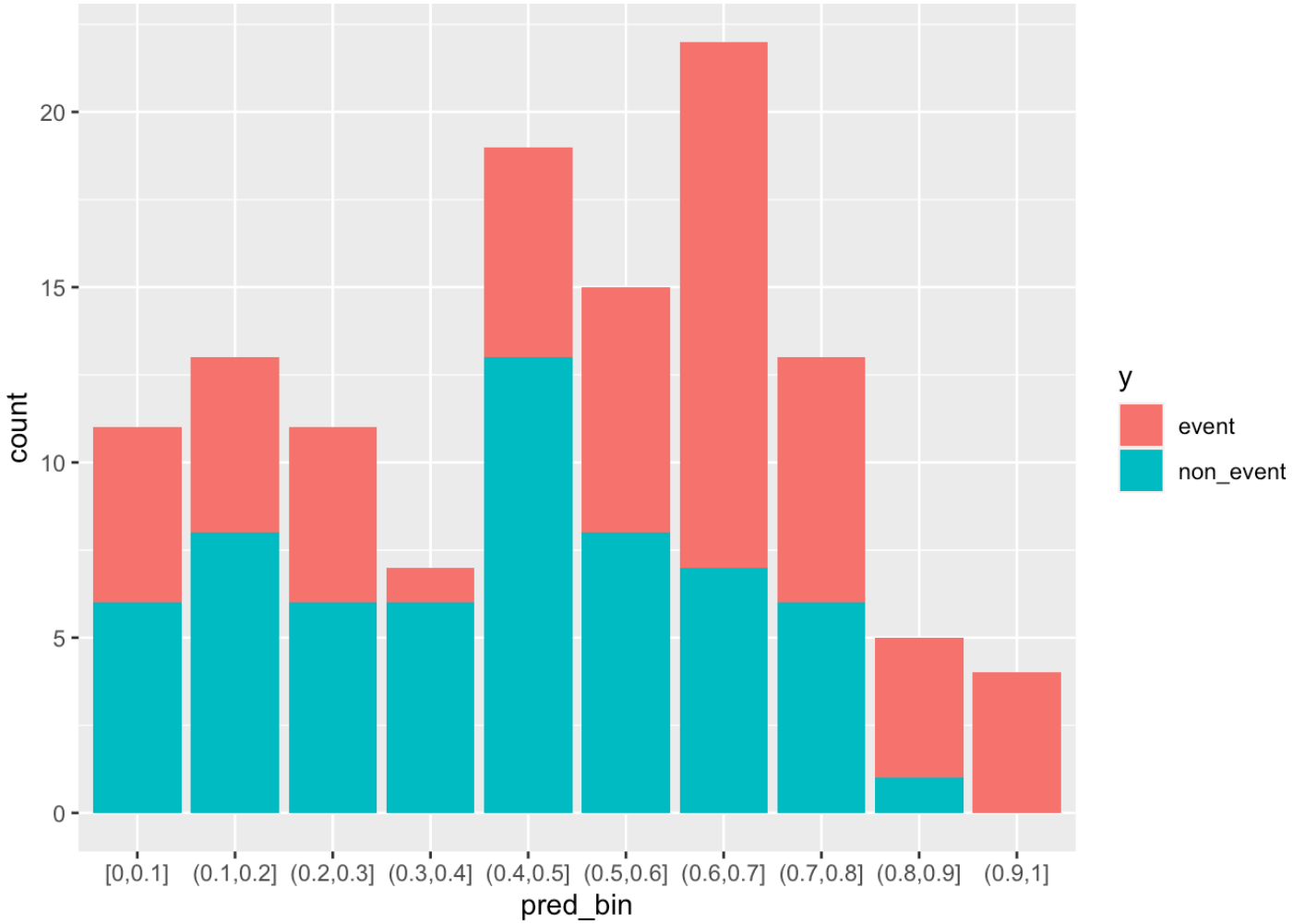
Do any bins consist of only events? Do any bins consist of only non-events?

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

SOLUTION

What do you think?

```
test_df_1_b %>%
    ggplot(mapping = aes(x = pred_bin)) + geom_bar(mapping = aes(fill = y)) + scale_c
olor_brewer(palette = "Set1")
```



The (0.9,1] bin consists of only events and there are no bins that only contain non_events.

3g)

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.

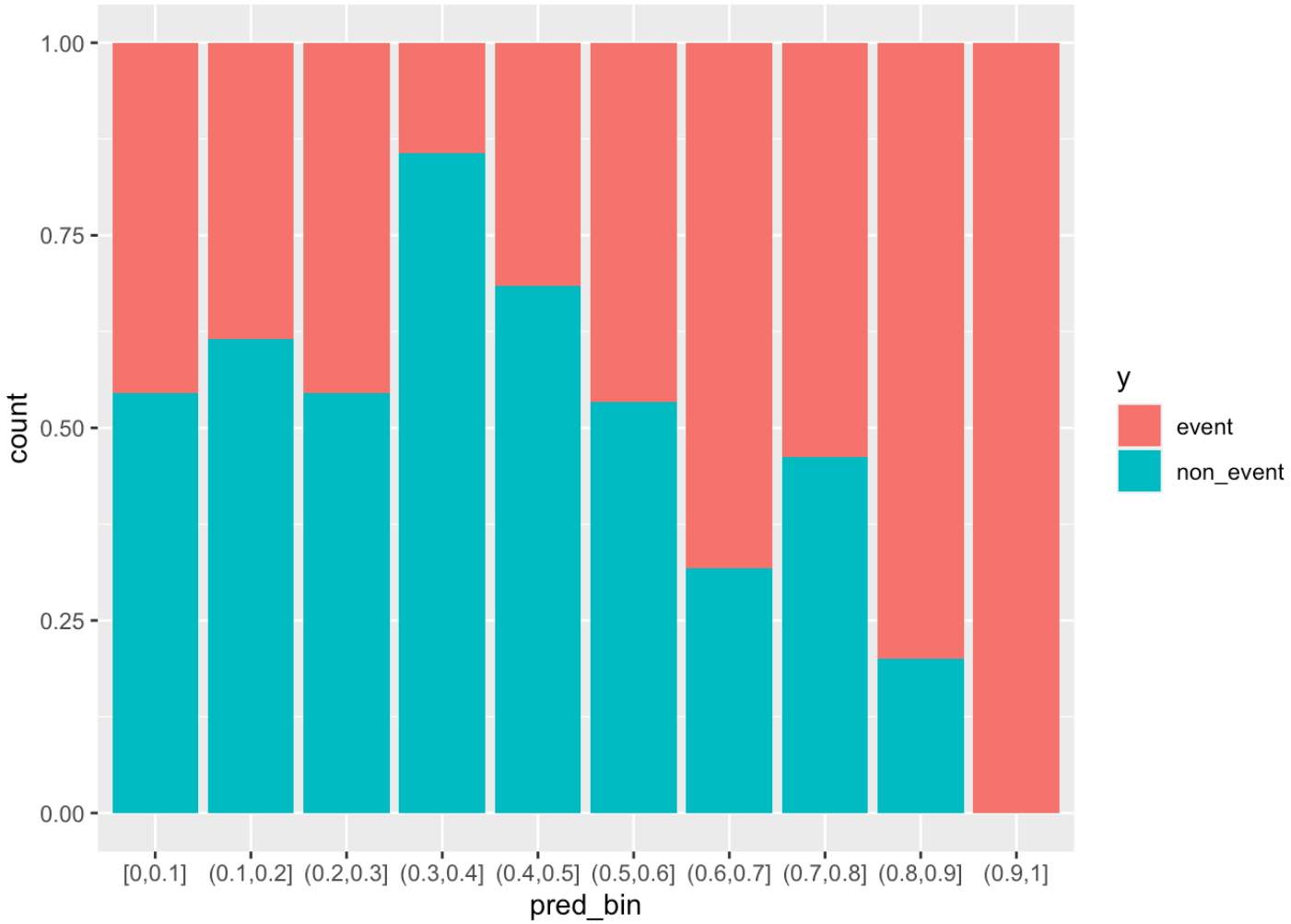
Are the empirical proptions of the event consistent with the predicted probability for model 1?

HINT: Pay close attention to the bar chart colors!

SOLUTION

What do you think?
I think that the empirical proportions of the event are pretty consistent with the predicted probability of model 1. When the predicted probability is less than 0.5 most of the time a non_event is more likely than an event. And in the bins with a threshold is greater than 0.5, an event was predicted more times often than a non_event in all of the bins.

```
test_df_1_b %>%
    ggplot(mapping = aes(x = pred_bin)) + geom_bar(mapping = aes(fill = y), position
= "fill") +
    scale_color_brewer(palette = "Set1")
```



Problem 04

The bar chart visualized in Problem 03 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.

4a)

Calculate the empirical propotion 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.

```
cal_curve <- data.frame("y" = test_df_1_b$y, "pred_bin" = test_df_1_b$pred_bin)
cal_curve <- cal_curve %>%
  group_by(pred_bin) %>%
  mutate(
    "prop_event" = mean(y == "event")
  )
cal_curve <- data.frame("pred_bin" =cal_curve$pred_bin,
                        "prop_event" = cal_curve$prop_event)
mid_bin <- c(0.05,.15,.25,.35,.45,.55,.65,.75,.85,.95)
my_calcurve_1 <- cbind(cal_curve, mid_bin)


my_calcurve_1 <- my_calcurve_1 %>%
  group_by(pred_bin) %>%
  summarise(prop_event = mean(prop_event)) %>%
  cbind(mid_bin)

my_calcurve_1
```

```
##     pred_bin prop_event mid_bin
## 1    [0,0.1]  0.4545455    0.05
## 2  (0.1,0.2]  0.3846154    0.15
## 3  (0.2,0.3]  0.4545455    0.25
## 4  (0.3,0.4]  0.1428571    0.35
## 5  (0.4,0.5]  0.3157895    0.45
## 6  (0.5,0.6]  0.4666667    0.55
## 7  (0.6,0.7]  0.6818182    0.65
## 8  (0.7,0.8]  0.5384615    0.75
## 9  (0.8,0.9]  0.8000000    0.85
## 10   (0.9,1]  1.0000000    0.95
```

4b)

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 4a).

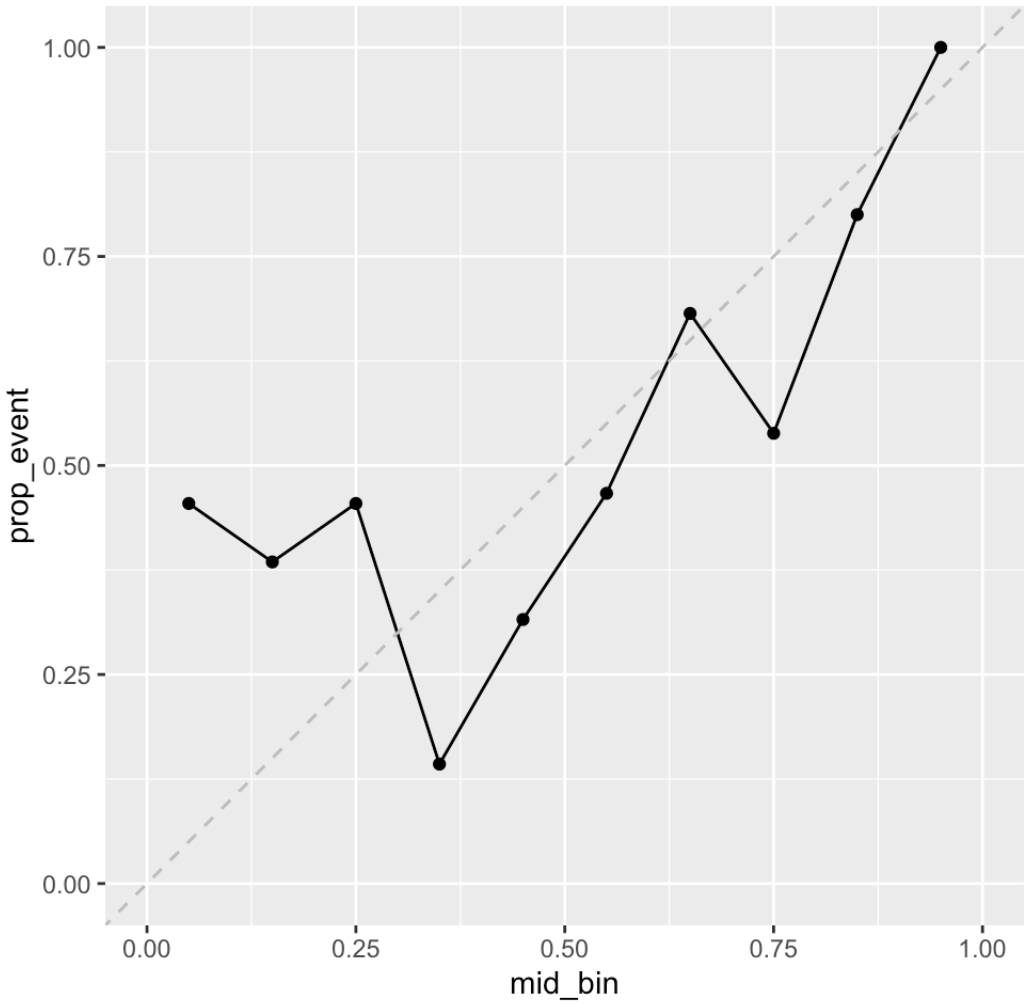
Pipe the `my_calcurve_1` object to `ggplot()` and map `mid_bin` and `prop_event` to the `x` and `y` aesthetics, respectively. Include `geom_line()` and `geom_point()` layers and a `geom_abline()` layer. Set the `geom_abline()` arguments as `slope=1`, `intercept=0`, `color='grey'`, and `linetype='dashed'`.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

```
my_calcurve_1 %>%
  ggplot(mapping = aes(x = mid_bin, y = prop_event)) + geom_line() + geom_point() +ge
om_abline(slope = 1, intercept= 0,  color = 'grey', linetype = 'dashed') + coord_equa
l(xlim = c(0,1), ylim= c(0,1))
```



The calibration plot appears to follow the trends of the stacked barplot pretty cloesly. When the calibration plot decreases from bin to bin, the proportion of the event is decreasing. In the corresponding bars in the stacked bar plot, this is reflected with an decrease in event percentage. The same is true for the opposite (an increase in calibration plot corresponds to more of a bar being filled as event).

4c)

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.

SOLUTION

```
pred_test_3 <- predict(mod_3_acc, newdata = df_test, type = 'prob')
pred_test_8 <- predict(mod_8_acc, newdata = df_test, type = 'prob')
```

4d)

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.

SOLUTION

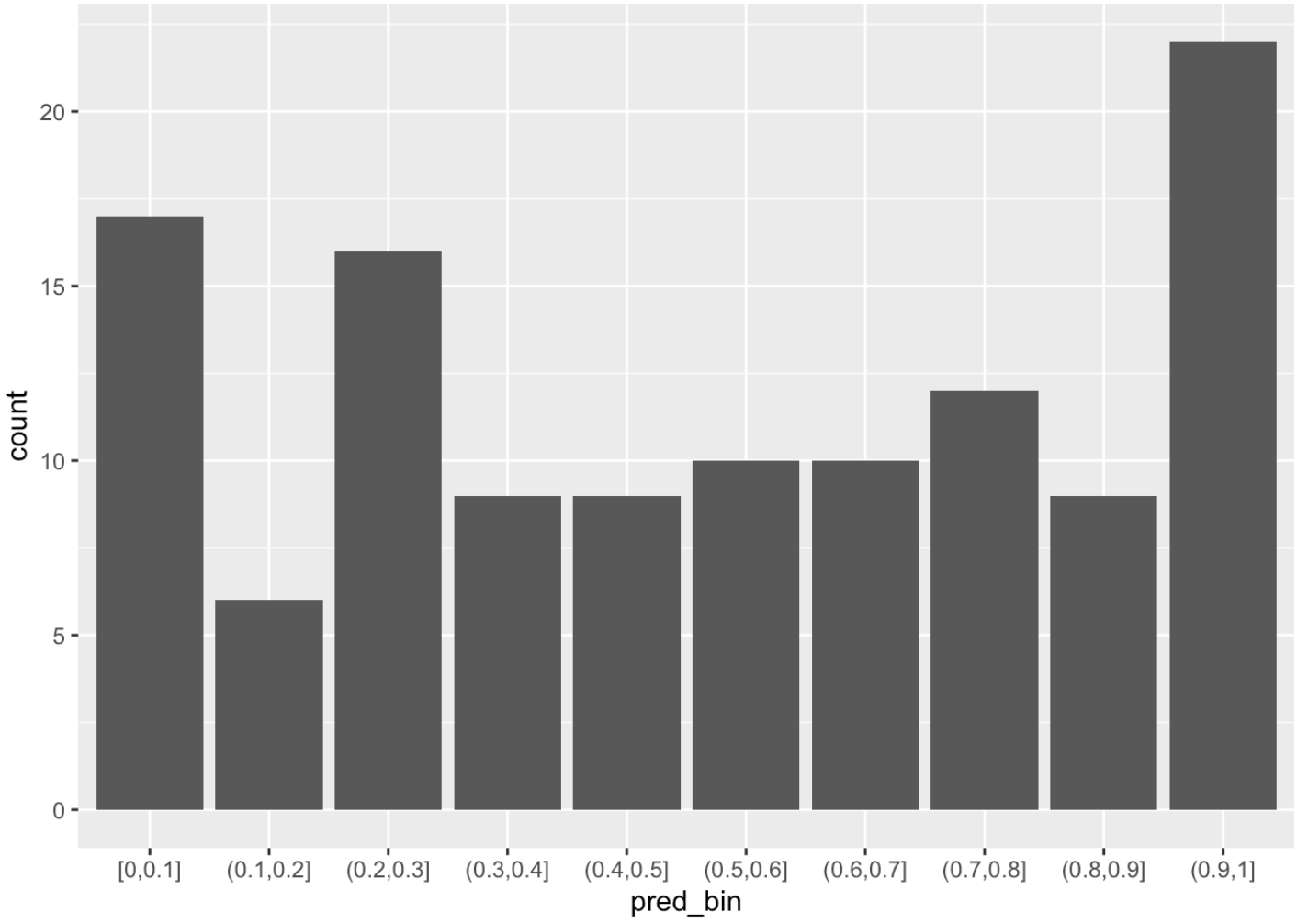
```
test_df_3_b <- test_df_3 %>%
  mutate(
    pred_bin = cut(test_df_3$event,
                   breaks = seq(0,1, by = 0.10),
                   include.lowest = TRUE)
  )

test_df_8_b <- test_df_8 %>%
  mutate(
    pred_bin = cut(test_df_8$event,
                   breaks = seq(0,1, by = 0.10),
                   include.lowest = TRUE)
  )
```

4e)

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

SOLUTION

```
test_df_3_b %>%
    ggplot(mapping = aes(x = pred_bin)) + geom_bar()
```

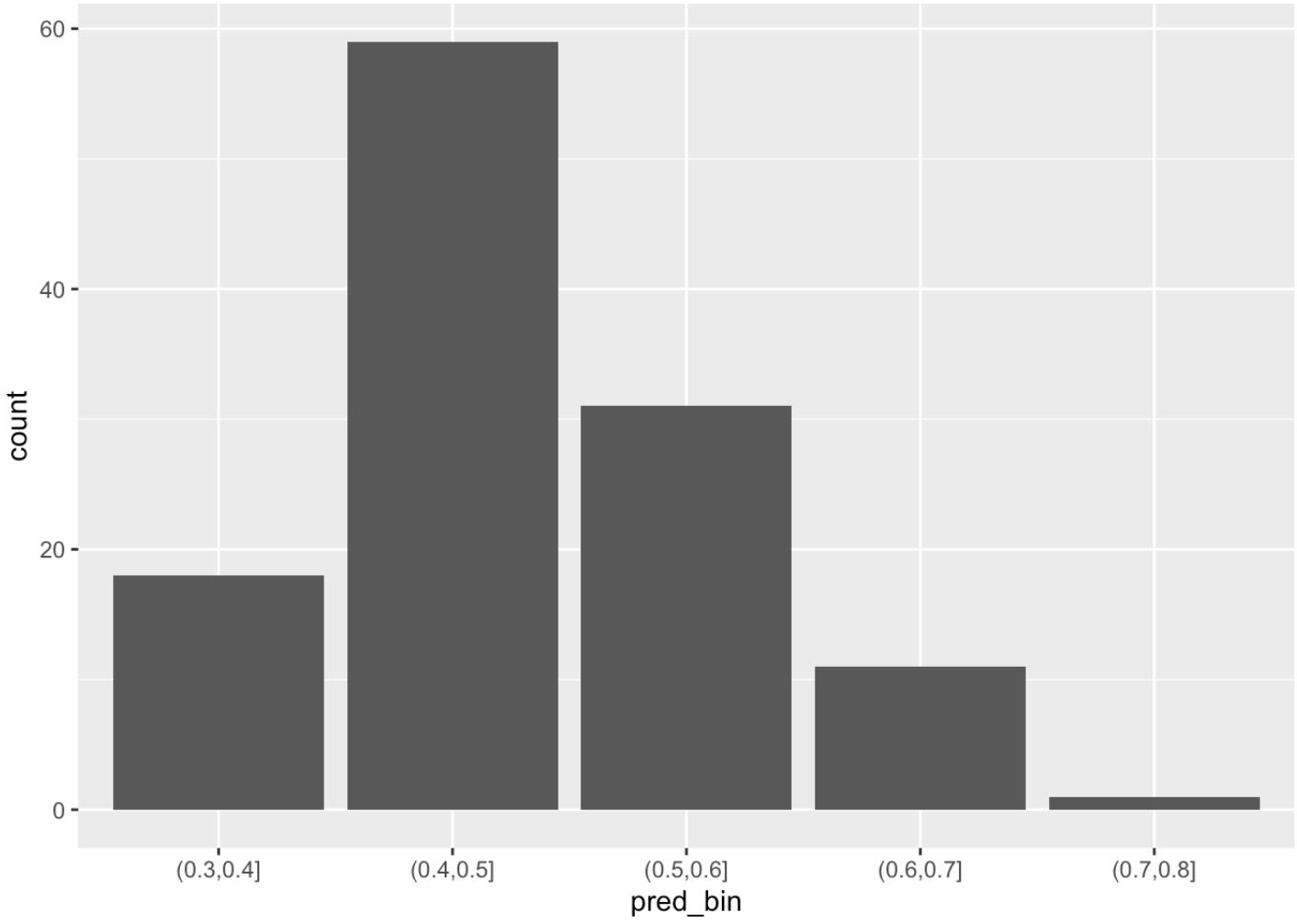


4f)

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

SOLUTION

```
test_df_8_b %>%
    ggplot(mapping = aes(x = pred_bin)) + geom_bar()
```



Problem 05

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.

5a)

Calculate the empirical propotion 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.

```
cal_curve3 <- data.frame("y" = test_df_3_b$y, "pred_bin" = test_df_3_b$pred_bin)
cal_curve3 <- cal_curve3 %>%
  group_by(pred_bin) %>%
  mutate(
    "prop_event" = mean(y == "event")
  )
cal_curve3 <- data.frame("pred_bin" =cal_curve3$pred_bin,
                      "prop_event" = cal_curve3$prop_event)
mid_bin <- c(0.05,.15,.25,.35,.45,.55,.65,.75,.85,.95)
my_calcurve_3 <- cbind(cal_curve3, mid_bin)


my_calcurve_3 <- my_calcurve_3 %>%
  group_by(pred_bin) %>%
  summarise(prop_event = mean(prop_event)) %>%
  cbind(mid_bin)

my_calcurve_3
```

```
##     pred_bin prop_event mid_bin
## 1    [0,0.1]  0.1176471    0.05
## 2  (0.1,0.2]  0.1666667    0.15
## 3  (0.2,0.3]  0.3125000    0.25
## 4  (0.3,0.4]  0.3333333    0.35
## 5  (0.4,0.5]  0.1111111    0.45
## 6  (0.5,0.6]  0.6000000    0.55
## 7  (0.6,0.7]  0.6000000    0.65
## 8  (0.7,0.8]  0.5833333    0.75
## 9  (0.8,0.9]  0.7777778    0.85
## 10   (0.9,1]  0.9545455    0.95
```

5b)

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.

```
cal_curve8 <- data.frame("y" = test_df_8_b$y, "pred_bin" = test_df_8_b$pred_bin)
cal_curve8 <- cal_curve8 %>%
  group_by(pred_bin) %>%
  mutate(
    "prop_event" = mean(y == "event")
  )
cal_curve8 <- data.frame("pred_bin" =cal_curve8$pred_bin,
                      "prop_event" = cal_curve8$prop_event)
mid_bin8 <- c(.35,.45,.55,.65,.75)
my_calcurve_8 <- cbind(cal_curve8, mid_bin8)


my_calcurve_8 <- my_calcurve_8 %>%
  group_by(pred_bin) %>%
  summarise(prop_event = mean(prop_event)) %>%
  cbind(mid_bin8)

my_calcurve_8
```

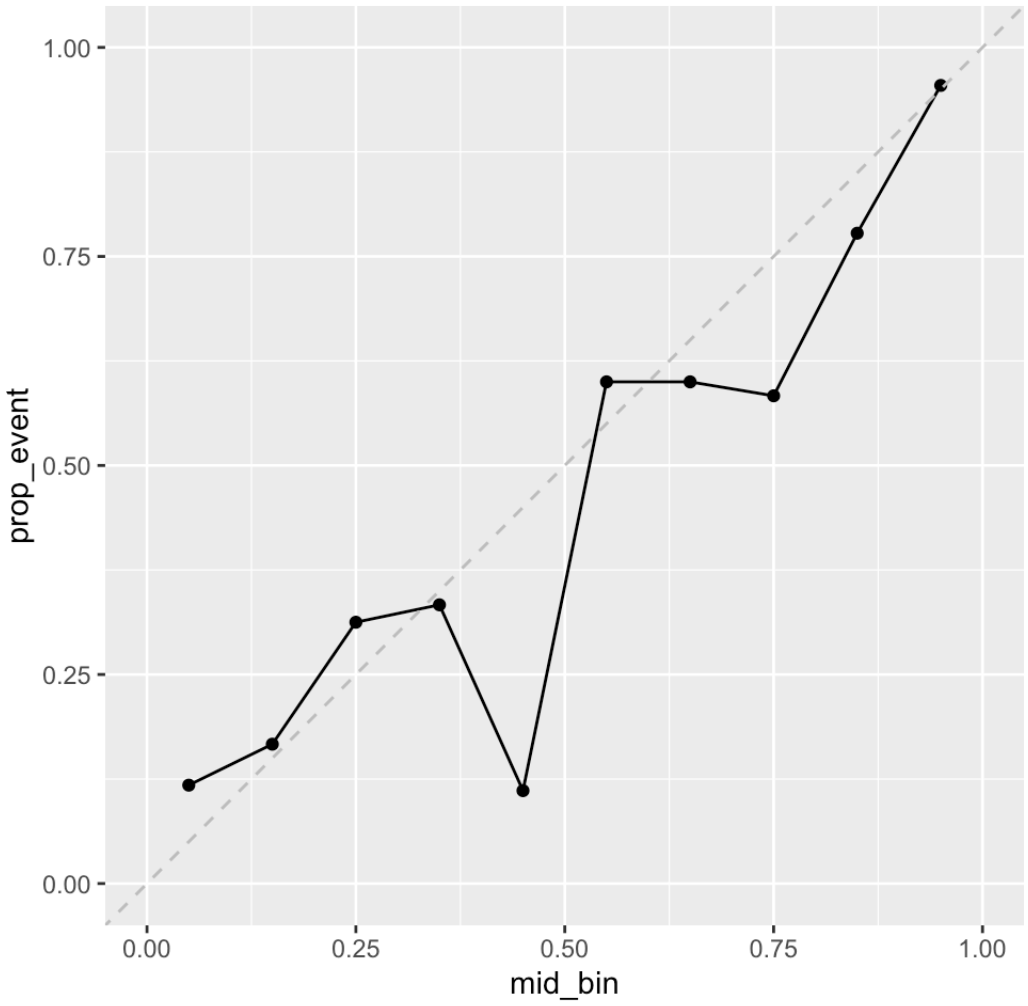
```
##    pred_bin prop_event mid_bin8
## 1 (0.3,0.4]  0.4444444     0.35
## 2 (0.4,0.5]  0.4915254     0.45
## 3 (0.5,0.6]  0.4838710     0.55
## 4 (0.6,0.7]  0.6363636     0.65
## 5 (0.7,0.8]  0.0000000     0.75
```

5c)

Visualize the calibration curve for model 3 using the approach described in Problem 4b).

SOLUTION

```
my_calcurve_3 %>%
  ggplot(mapping = aes(x = mid_bin, y = prop_event)) + geom_line() + geom_point() +ge
om_abline(slope = 1, intercept= 0,  color = 'grey', linetype = 'dashed') + coord_equa
l(xlim = c(0,1), ylim= c(0,1))
```

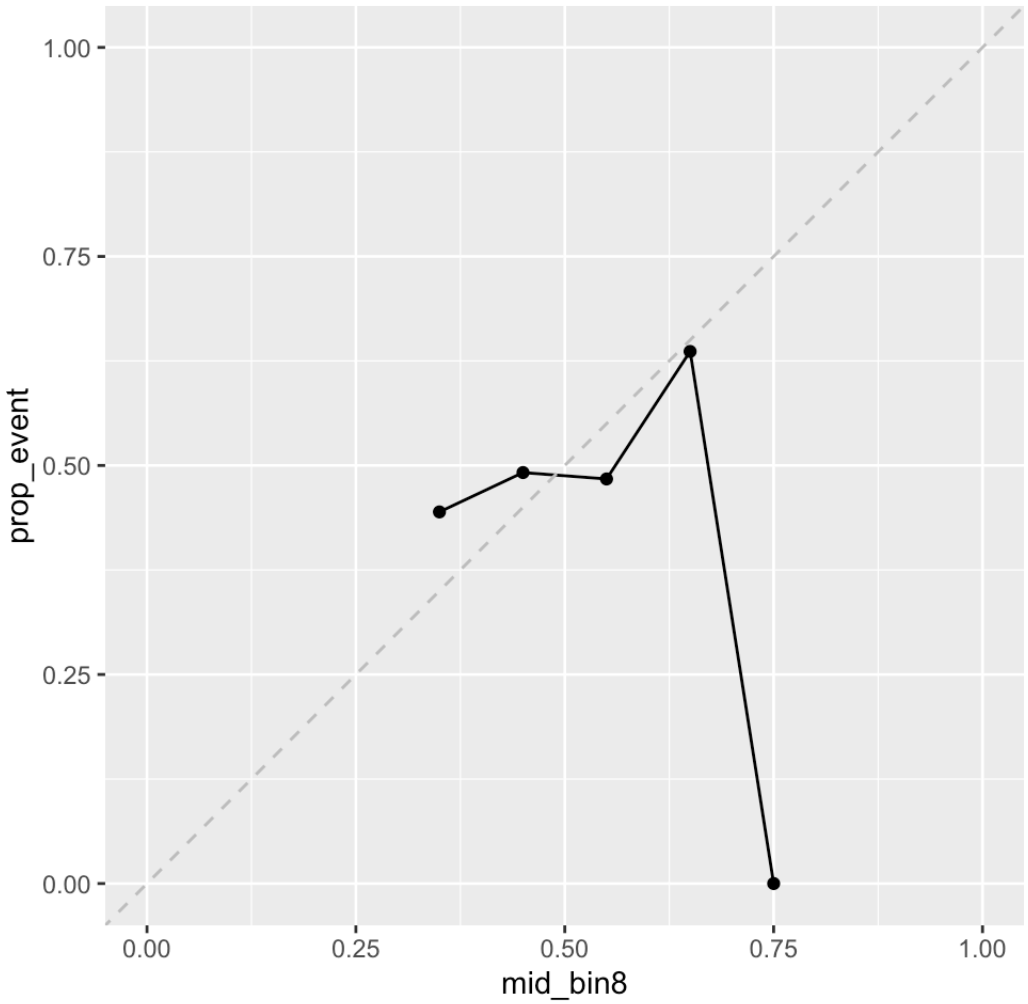


5d)

Visualize the calibration curve for model 8 using the approach described in Problem 4b).

SOLUTION

```
my_calcurve_8 %>%
  ggplot(mapping = aes(x = mid_bin8, y = prop_event)) + geom_line() + geom_point() +g
eom_abline(slope = 1, intercept= 0,  color = 'grey', linetype = 'dashed') + coord_equ
al(xlim = c(0,1), ylim= c(0,1))
```



5d)

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?
Model 3 appears to the most well calibrated. It is a little difficult to see which models performs better by simply using the calibration curve because with the different models the different curves may perform differently in each bin, therefore it is harder to understand the overall performance of a model. The most obvious aspect of performance based on the calibration curve is whether the midpoint of each bin falls within their respective side of the 45 degree line. For bins greater than the threshold, the proportion of events should be greater than 50%, meaning that when the bin is over the threshold, the event should be more likely to be predicted.

Problem 06

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.

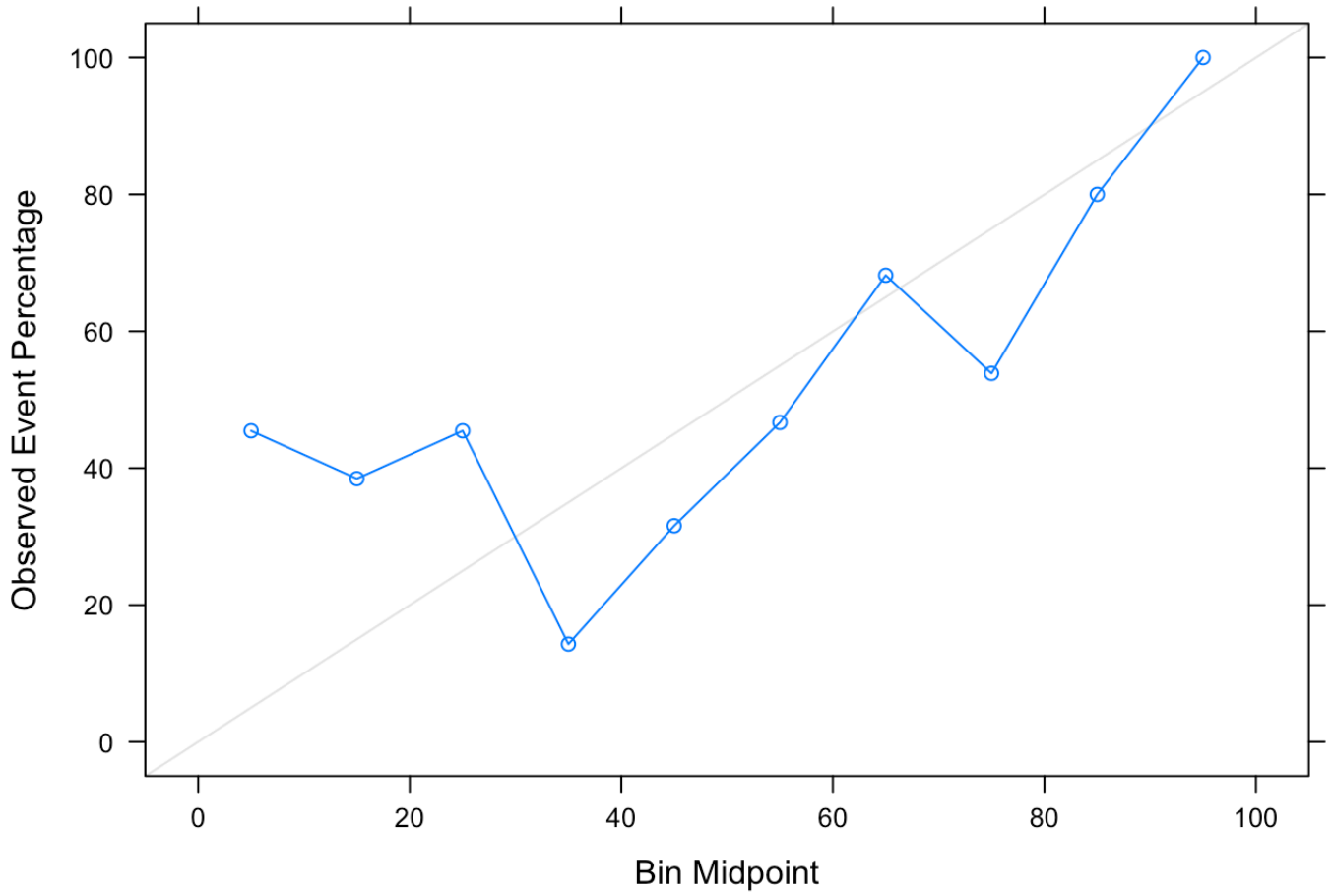
6a)

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

```
caret::calibration(y ~ event, data = test_df_1, cuts = 10) %>%
  xyplot()
```



The created calibration curbe is pretty consistent with my manually created curve for model 1.

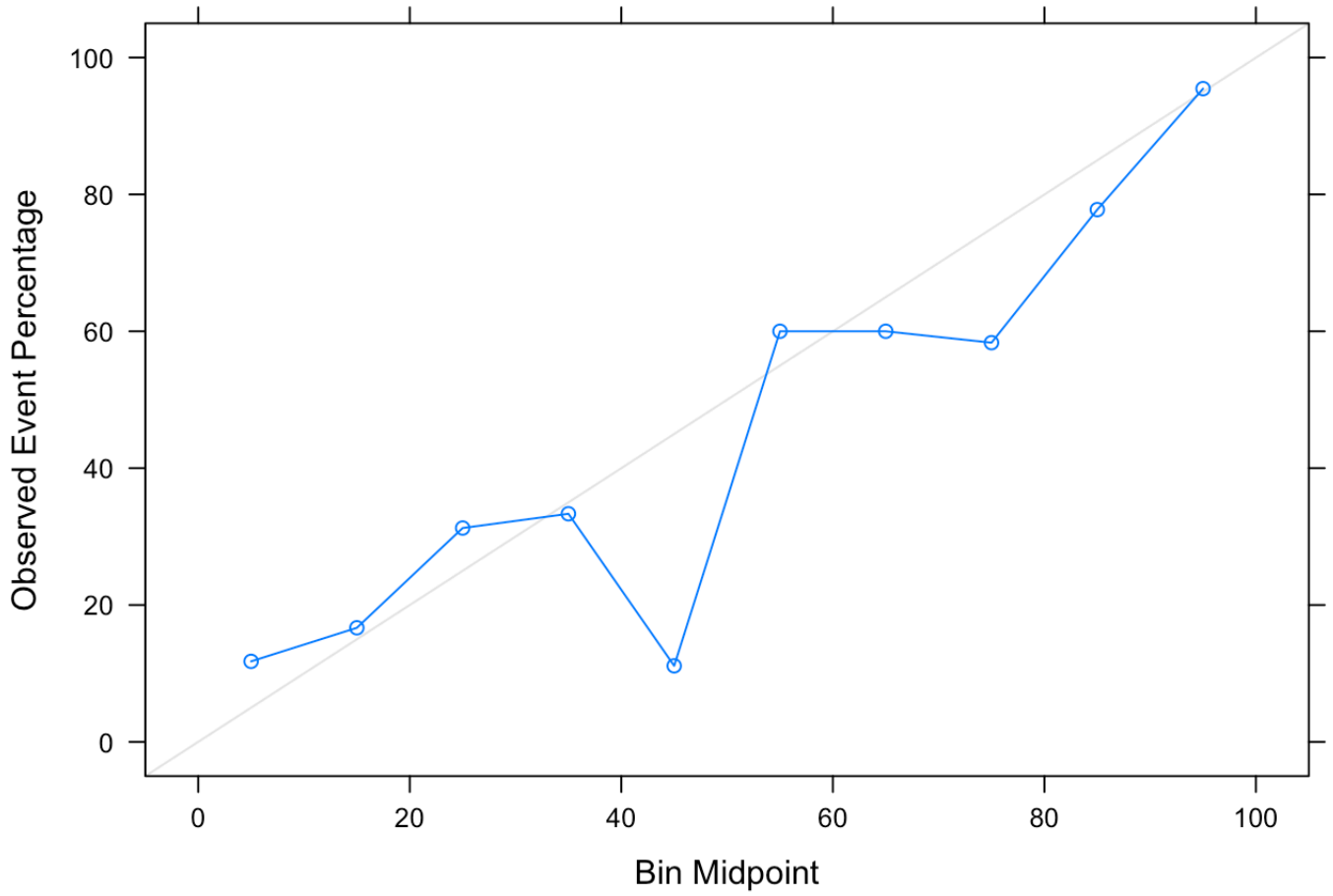
6b)

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

```
caret::calibration(y ~ event, data = test_df_3, cuts = 10) %>%
  xyplot()
```



The created curve is consistent with the manual curve for model 3.

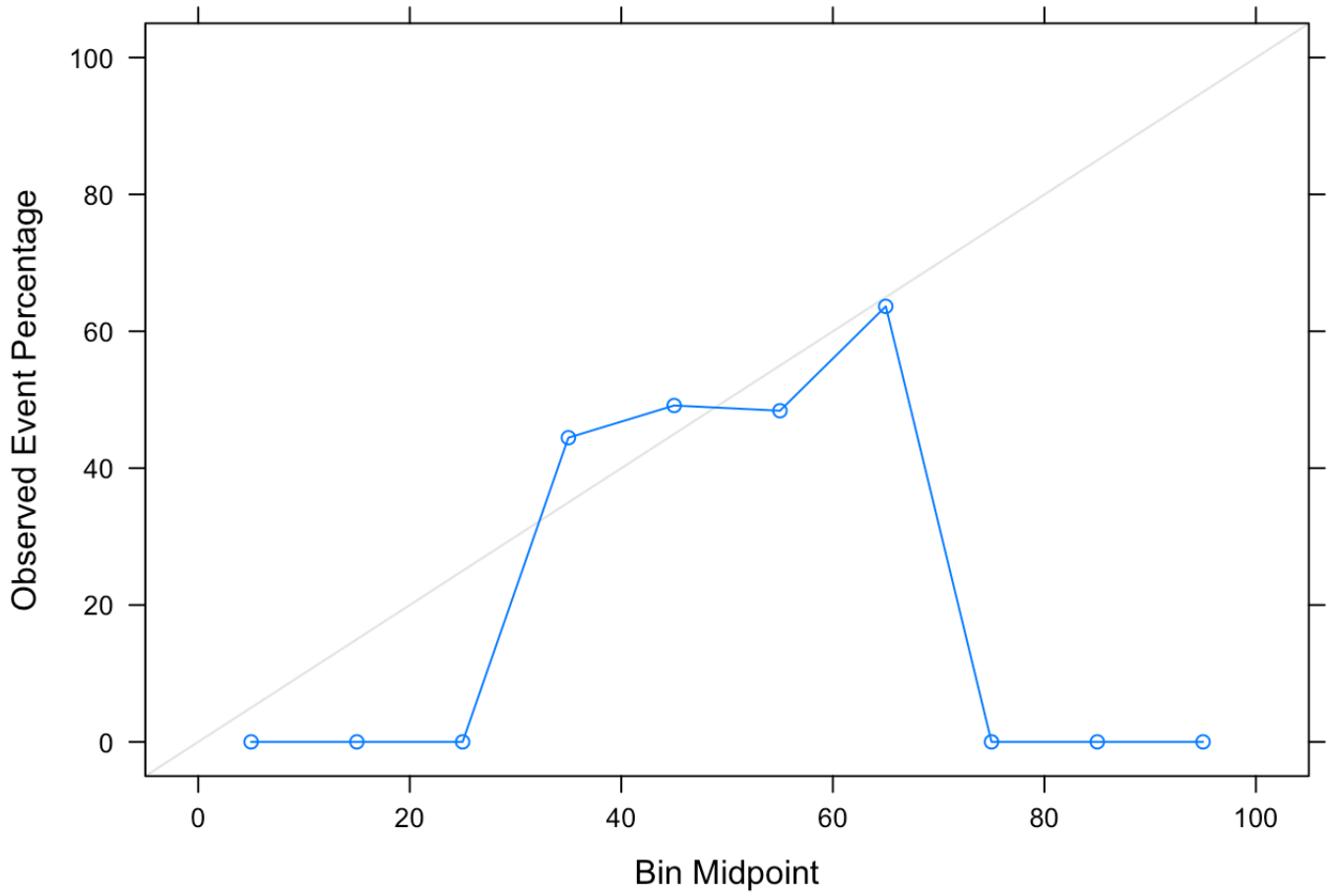
6c)

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

```
caret::calibration(y ~ event, data = test_df_8, cuts = 10) %>%
  xyplot()
```



Yes, the created calibration curve is consistent with the manual curve for model 8. In the manual curve that have a zero event percentage are not displayed but the results are consistent.