CS 1675 Spring 2022 Homework: 02

Assigned January 19, 2022; Due: January 26, 2022

Sameera BoppanaE

Submission time: January 26, 2022 at 11:00PM EST

Collaobrators

Include the names of your collaborators here.

Overview

This assignment introduces training and comparing models with the `caret` package. You will demonstrate a majority of the steps in predictive modeling applications. You will apply those actions to a simplified regression example. You will then work with a binary classification problem, focusing on calculating confusion matrix metrics from given model predictions.

If you need help with understanding the R syntax please see the R4DS book (https://r4ds.had.co.nz/) and/or the R tutorial videos and demos available on the Canvas site for the course.

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

Load in the packages to be used in the first two problems.

```
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

Last assignment, you were introduced to `R'` s formula interface. You will continue to use the formula interface in this assignment. You will train and resample models using the `caret` package instead of calling the `lm()` directly.

You will use the `caret` package to train 9 polynomial models of various levels of complexity in Problem 01 and 02. You will specify a resampling scheme and a primary performance metric. The 9 models will be assessed through the resampled hold-out sets. The `caret` package will take care of the "book keeping" for you. That said, you will work through typing in the tedious formula interface for the different models. This is not the most efficient way to perform these actions. It is to get more practice with syntax, and to give a sense that complexity is related to the number of features in a model.

You will work with a synthetic data set. In Problem 01, the data are considered "noisy" while the data in Problem 02 has considerably less noise. You will try and interpret the influence noise has the model training process.

The code chunk below reads in a data set for you. The code chunk below that displays a glimpse of the dataset which shows there are just 2 variables. The variable `x` is the input and the variable `y` is the response. You task is to predict `y` as a function of `x` using various polynomial models. As stated before, the data in Problem 01 are the "noisy" data.

```
high_noise_github_file <- "https://raw.githubusercontent.com/jyurko/CS_1675_Spring_2022/main/HW/02/prob_high_noise_dataset.csv"

prob_high_noise <- readr::read_csv(high_noise_github_file,
                                   col_names = TRUE)
```

```
## Rows: 75 Columns: 2
```

```
## ── Column specification ────────────────────────────────────────────────────
## Delimiter: ","
## dbl (2): x, y
```

```
##
## 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.
```

```
prob_high_noise %>% glimpse()
```

```
## Rows: 75
## Columns: 2
## $ x <dbl> 0.01771688, -0.50620378, -0.71966520, 0.53919075, 1.27406702, -0.960…
## $ y <dbl> -3.9699220, -0.3447584, -3.1207953, 6.4064465, 10.1139354, 1.5896728…
```

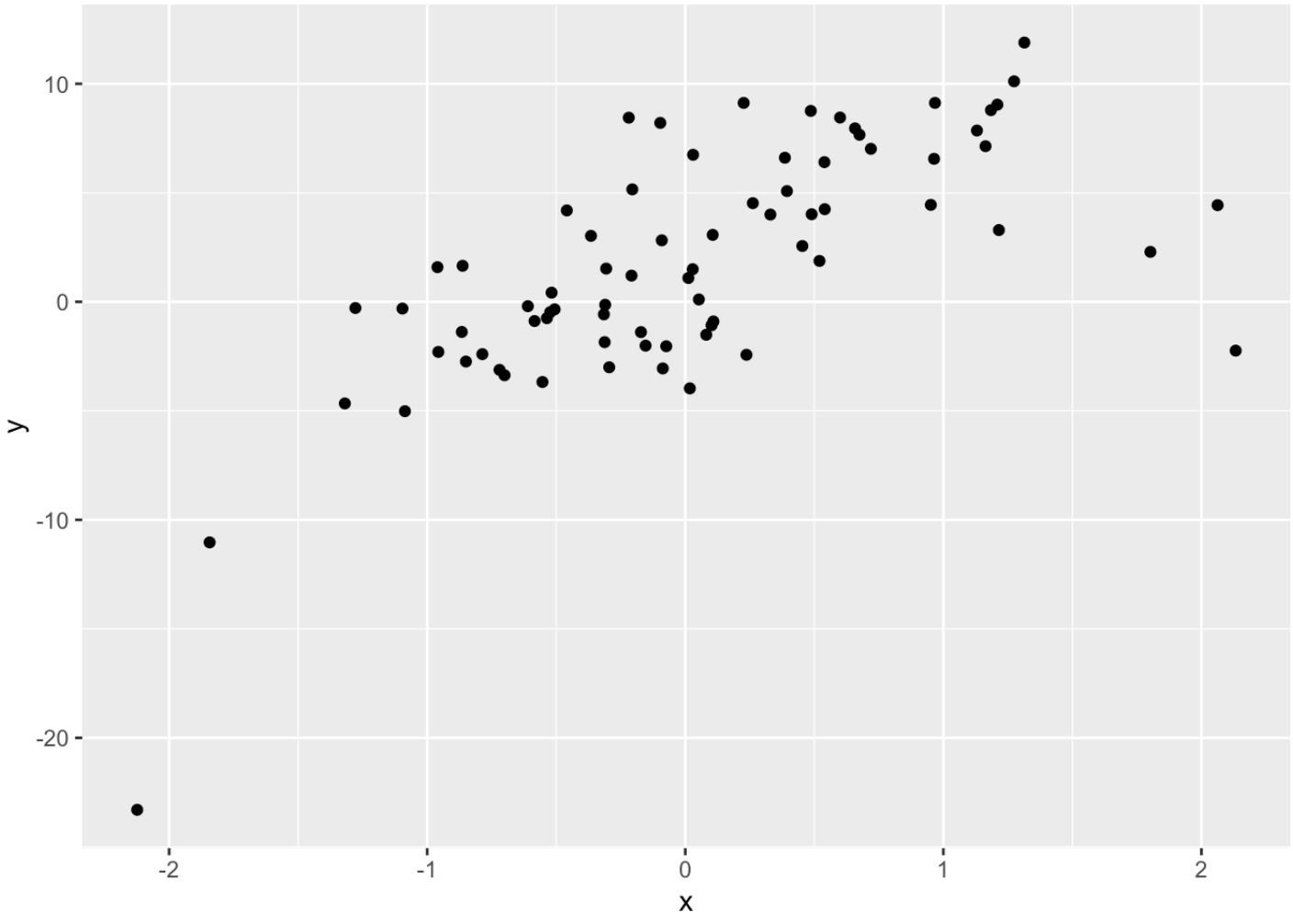
1a)

Use ggplot2 to create a scatter plot between the input, `x`, and the response, `y`. The scatter plot is created with the `geom_point()` geom.

Does the relationship between the input and response appear to be linear or non-linear?

SOLUTION

```
prob_high_noise %>%
  ggplot(mapping=aes(x,y)) + geom_point()
```



Is the relationship linear or non-linear?
The relationship between the input and response appears to be non-linear.

1b)

If you have not downloaded and installed `caret` please go ahead and do so now.

Load in the `caret` package with the `library()` function in the code chunk below.

SOLUTION

```
library(caret)
```

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

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

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

1c)

The resampling scheme is specified by the `trainControl()` function in `caret`. The type of scheme is controlled by the `method` argument. For k-fold cross-validation, the number of folds is controlled by the `number` argument. You can decide to use repeated cross-validation by specifying the `repeats` argument to be greater than 1, as long as you specify the `method` argument to be `"repeatedcv"` instead of just `"cv"`.

Decide the type of resampling scheme you would like to use. Assign the result of the `trainControl()` function to the `my_ctrl` variable. Based on your desired number of folds and repeats, how many times will a model be trained and tested?

SOLUTION

```
my_ctrl <- trainControl(
  method = 'cv',
  number = 10
)
```

How many times will an individual model be trained and tested? Each model will be trained 9 times and be held out 1 time as the test set.

1d)

The response is a continuous variable.

You must select a primary performance metric to use to compare the models. Specify an appropriate metric to use for this modeling task. Choices must be written as a string and assigned to the `my_metric` variable. Possible choices are "Accuracy", "RMSE", "Kappa", "Rsquared", "MAE", "ROC". Why did you make the the choice that you did?

NOTE: Not all of the listed performance metrics above are relevant to regression problems!

```
my_metric <- "RMSE"
```

Why did you make your choice?
I chose RMSE because the units of the RMSE will be the same as the response and it will be sensitive to outliers and accurately assess the overall fit of the models. Since we know that there is high noise, it is better to be more sensitive to outliers.

1e)

You will now go through fitting 9 different models with the `train()` function from `caret`. You will use the formula interface to specify the model relationship. You must fit a linear (first order polynomial), quadratic (second order polynomial), cubic (third order polynomial), and so on up to and including a 9th order polynomial.

You must specify the `method` argument in the `train()` function to be `"lm"`. You must specify the `metric` argument to be `my_metric` that you selected in Problem 1d). You must specify the `trControl` argument to be `my_ctrl` that you specified in Problem 1c). Don't forget to set the `data` argument to be `prob_high_noise`.

The variable names below and comments are used to tell you which polynomial order you should assign to which object.

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

SOLUTION

```
### linear relationship
set.seed(2001)
mod_high_1 <- train(y~x, data=prob_high_noise, trControl=my_ctrl, metric=my_metric)
```

```
### quadratic relationship
set.seed(2001)
mod_high_2 <- train(y~x + I(x^2), data=prob_high_noise, trControl=my_ctrl, metric=my_
metric)
```

```
## note: only 1 unique complexity parameters in default grid. Truncating the grid to
1 .
```

```
### cubic relationship
set.seed(2001)
mod_high_3 <- train(y~x+I(x^2)+I(x^3),data=prob_high_noise, trControl=my_ctrl, metric
=my_metric)
```

```
## note: only 2 unique complexity parameters in default grid. Truncating the grid to
2 .
```

```
### 4th order
set.seed(2001)
mod_high_4 <- train(y~x+I(x^2)+I(x^3)+I(x^4), data=prob_high_noise, trControl=my_ctrl
, metric=my_metric)
```

```
### 5th order
set.seed(2001)
mod_high_5 <- train(y~x+I(x^2)+I(x^3)+I(x^4)+I(x^5), data=prob_high_noise, trControl=
my_ctrl, metric=my_metric)
```

```
### 6th order
set.seed(2001)
mod_high_6 <- train(y~x+I(x^2)+I(x^3)+I(x^4)+I(x^5)+I(x^6), data=prob_high_noise, trC
ontrol=my_ctrl, metric=my_metric)
```

```
### 7th order
set.seed(2001)
mod_high_7 <- train(y~x+I(x^2)+I(x^3)+I(x^4)+I(x^5)+I(x^6)+I(x^7), data=prob_high_noi
se, trControl=my_ctrl, metric=my_metric)
```

```
### 8th order
set.seed(2001)
mod_high_8 <- train(y~x+I(x^2)+I(x^3)+I(x^4)+I(x^5)+I(x^6)+I(x^7)+I(x^8), data=prob_h
igh_noise, trControl=my_ctrl, metric=my_metric)
```

```
### 9th order
set.seed(2001)
mod_high_9 <- train(y~x+I(x^2)+I(x^3)+I(x^4)+I(x^5)+I(x^6)+I(x^7)+I(x^8)+I(x^9), data
=prob_high_noise, trControl=my_ctrl, metric=my_metric)
```

1f)

The code chunk below compiles all of the model training results for you. The `high_noise_results` object can be used to compare the models through tables and visualizations.

```
high_noise_results = resamples(list(fit_01 = mod_high_1,
                                    fit_02 = mod_high_2,
                                    fit_03 = mod_high_3,
                                    fit_04 = mod_high_4,
                                    fit_05 = mod_high_5,
                                    fit_06 = mod_high_6,
                                    fit_07 = mod_high_7,
                                    fit_08 = mod_high_8,
                                    fit_09 = mod_high_9))
```

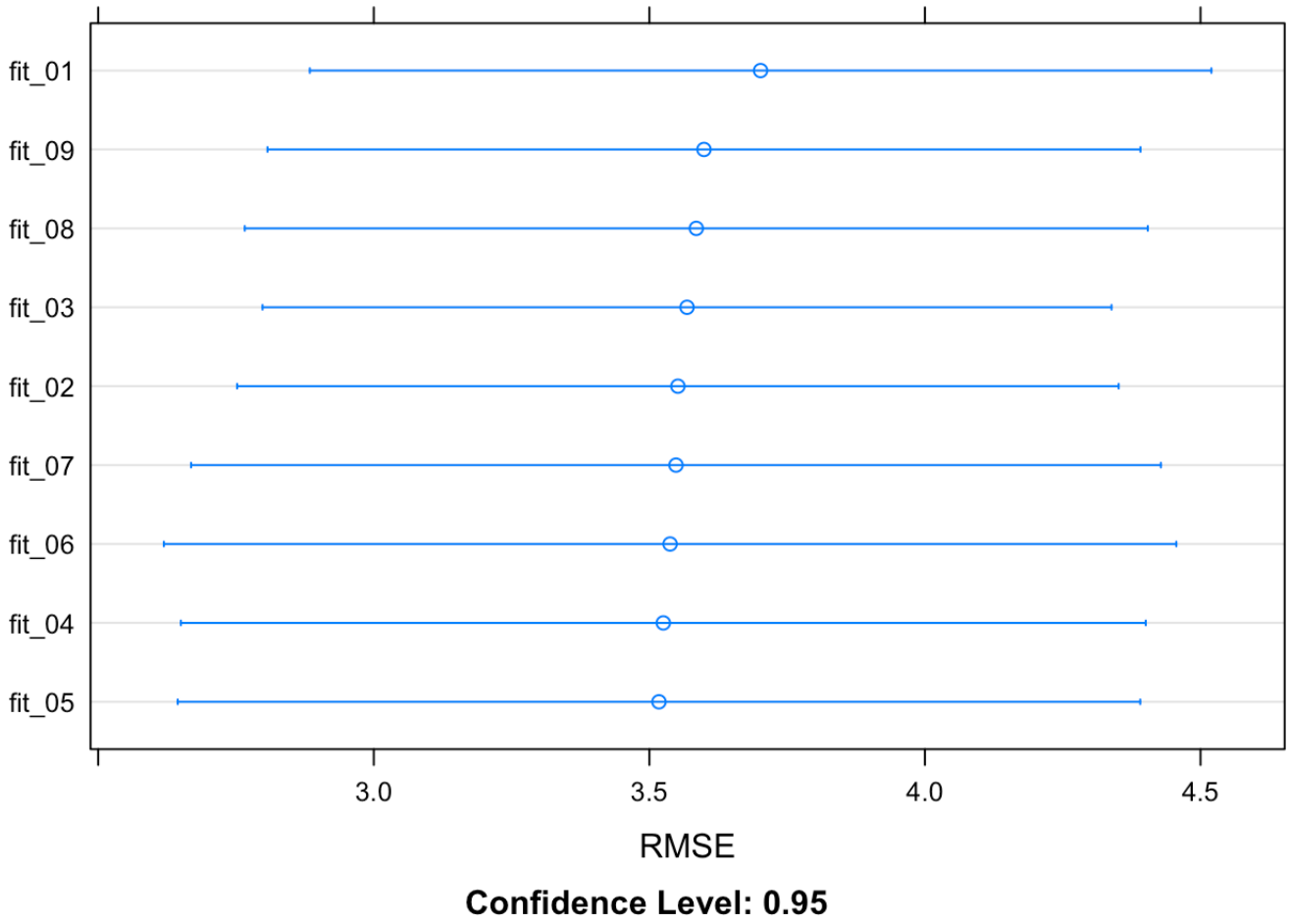
The `caret` package provides default visuals which rank the models based on their resampling hold-out results. Use the `dotplot()` function to create a dot plot with confidence intervals on the hold-out set performance metrics.

You must create two plots. One for the metric you specified in `my_metric` and another with a second performance metric appropriate for a regression problem. You specify the metric to show in `dotplot()` with the `metric` argument.

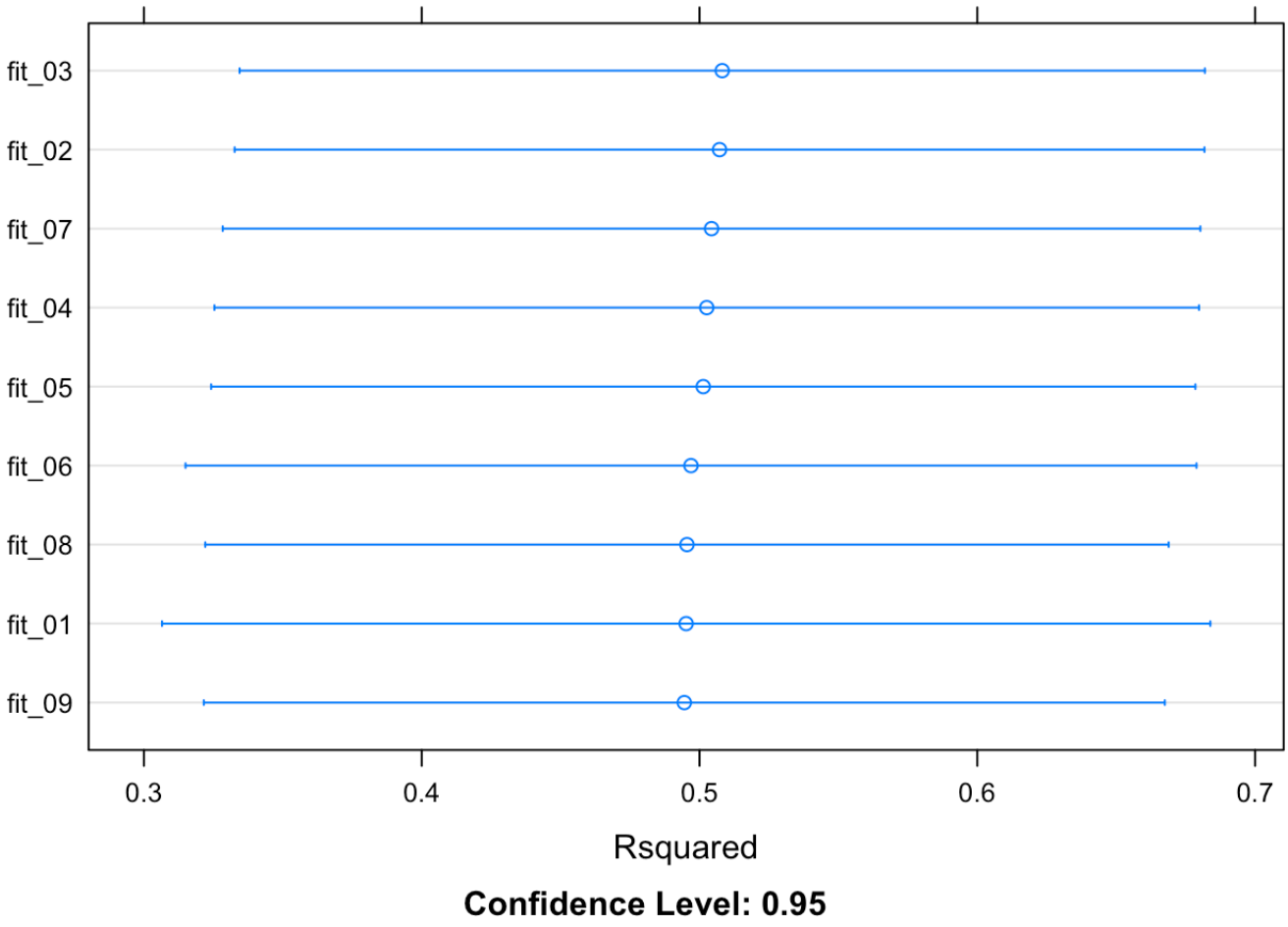
Based on your two figures, is there a clear best performing model? If so, which model is the best? If not, what are the top three models?

SOLUTION

```
### dotplot for your specified primary performance metric
dotplot(high_noise_results,  metric=my_metric)
```




```
### dotplot for another performance metric
metric2 = "Rsquared"
dotplot(high_noise_results,  metric=metric2)
```



Is there a best model?

There does not appear to be a clear best model. However, from the two metrics, Fit 5,4,6 may perform slightly better as they have the lowest RMSE values. All the models are slightly below or equal to 0.5 for R-squared.

1g)

The variable `mod_high_2` is a `caret` model object. However, you able to access the "underlying" model with `mod_high_2$finalModel` and use that object just like if we used the `lm()` function directly to fit the model. Therefore, regardless of your answer in Problem 1f), you will compare the coefficients of the top three models using the `coefplot::multiplot()` function. If you have not installed `coefplot` yet, please go ahead and do so.

Use `coefplot::multiplot()` to visualize the coefficients of your top three models by passing in three $finalModel objects into `coefplot::multiplot()`.

Does anything stand out to you in the figure?

SOLUTION

```
### your code here
mod_high_5$finalModel
```

```
##
## Call:
##  randomForest(x = x, y = y, mtry = min(param$mtry, ncol(x)))
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 2
##
##           Mean of squared residuals: 14.9482
##                     % Var explained: 48.08
```

Does anything stand out?

Problem 02

The data used in Problem 01 were generated using a “high” level of noise. You will now train and compare the same 9 models, but with data coming from a “low” level of noise. The underlying **data generating process** is the same between Problem 01 and Problem 02. All that changed is the noise level. We will learn what that means in more detail throughout this semester. For now, the main purpose is to compare what happens when you can train models under different noise level assumptions.

The low noise level data are loaded in the code chunk below. The glimpse shows that the variable names are the same as those in the high noise level data from Problem 01.

```
low_noise_github_file <- "https://raw.githubusercontent.com/jyurko/CS_1675_Spring_2022/main/HW/02/prob_low_noise_dataset.csv"

prob_low_noise <- readr::read_csv(low_noise_github_file, col_names = TRUE)
```

```
## Rows: 75 Columns: 2
```

```
## ── Column specification ────────────────────────────────────────────────────────
## Delimiter: ","
## dbl (2): x, y
```

```
##
## 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.
```

```
prob_low_noise %>% glimpse()
```

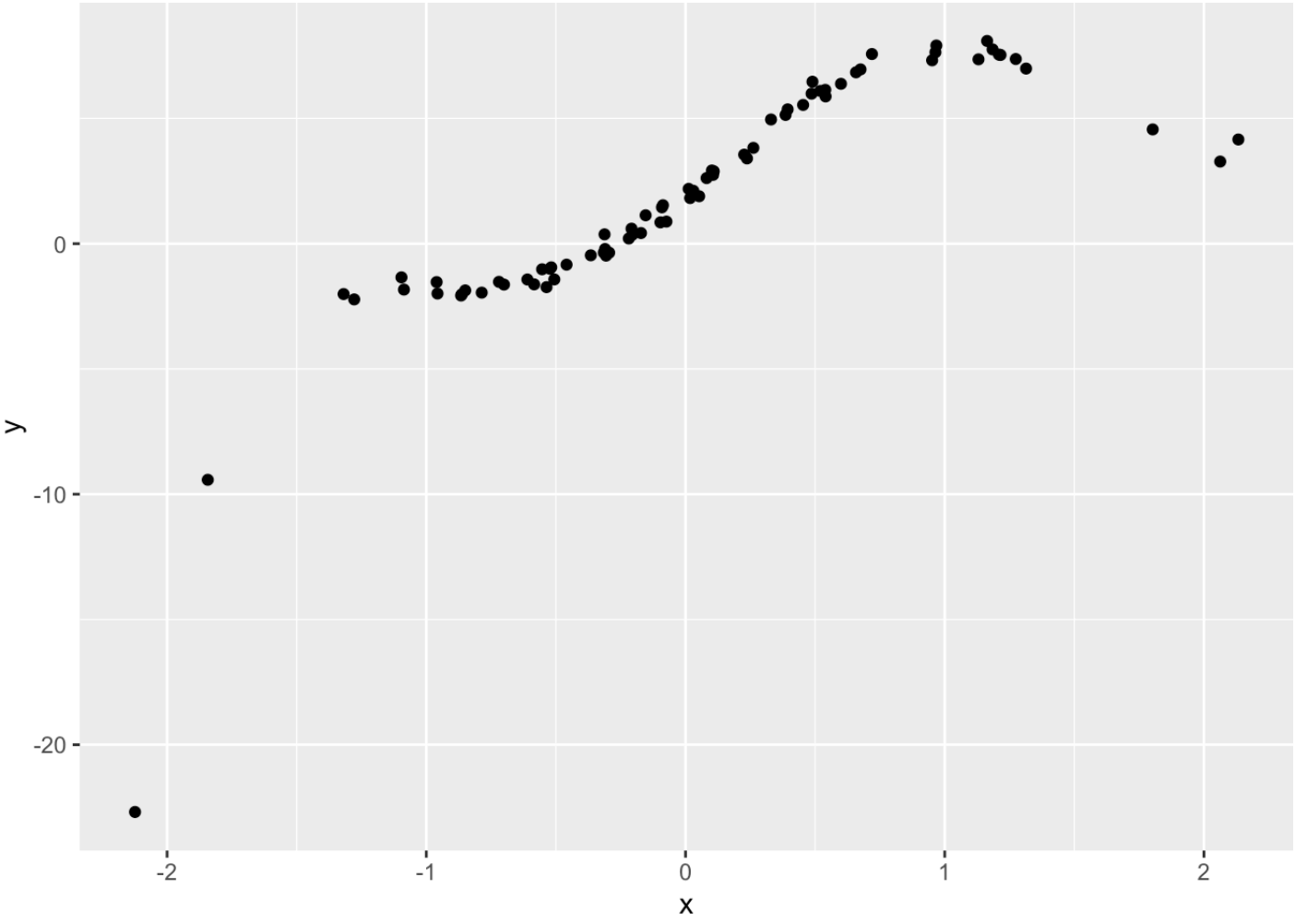
```
## Rows: 75
## Columns: 2
## $ x <dbl> 0.01771688, -0.50620378, -0.71966520, 0.53919075, 1.27406702, -0.960…
## $ y <dbl> 1.8214295, -1.4259504, -1.5224624, 6.1403486, 7.3708454, -1.5320704,…
```

2a)

Create a scatter plot between the input, `x`, and the response, `y`, for the low noise level data. How does this figure compare to the one you made in Problem 1a)?

SOLUTION

```
prob_low_noise %>%
    ggplot(mapping=aes(x,y)) + geom_point()
```



How does it compare?
The fit produced in 2a has a much more defined shape than the one produced in 1a. Since there is less noise in the dataset in 2a, a more clear relationship between the input and output can be defined.

2b)

You will use the same resampling scheme and primary performance metric that you used in Problem 01 to train 9 different polynomial models. This time however you will use the low noise level data.

Train the 9 different polynomail models using the required formula interface, `method` , `trControl` , and `metric` arguments that you used in Problem 01. However, pay close attention and set the `data` argument to be `prob_low_noise` .

The variable names and comments specify which polynomial to use.

NOTE: again this is VERY tedious…we will see more efficient ways of going through such a process later in the semester.

SOLUTION

```
### linear relationship
set.seed(2001)
mod_low_1 <- train(y~x, data=prob_low_noise, trControl=my_ctrl, metric=my_metric)
```

```
### quadratic relationship
set.seed(2001)
mod_low_2 <- train(y~x+I(x^2), data=prob_low_noise, trControl=my_ctrl, metric=my_metr
ic)
```

```
## note: only 1 unique complexity parameters in default grid. Truncating the grid to
1 .
```

```
### cubic relationship
set.seed(2001)
mod_low_3 <- train(y~x+I(x^2)+I(x^3), data=prob_low_noise, trControl=my_ctrl, metric=
my_metric)
```

```
## note: only 2 unique complexity parameters in default grid. Truncating the grid to
2 .
```

```
### 4th order
set.seed(2001)
mod_low_4 <- train(y~x+I(x^2)+I(x^3)+I(x^4), data=prob_low_noise, trControl=my_ctrl,
metric=my_metric)
```

```
### 5th order
set.seed(2001)
mod_low_5 <- train(y~x+I(x^2)+I(x^3)+I(x^4)+I(x^5), data=prob_low_noise, trControl=my
_ctrl, metric=my_metric)
```

```
### 6th order
set.seed(2001)
mod_low_6 <- train(y~x+I(x^2)+I(x^3)+I(x^4)+I(x^5)+I(x^6), data=prob_low_noise, trCon
trol=my_ctrl, metric=my_metric)
```

```
### 7th order
set.seed(2001)
mod_low_7 <- train(y~x+I(x^2)+I(x^3)+I(x^4)+I(x^5)+I(x^6)+I(x^7), data=prob_low_noise
, trControl=my_ctrl, metric=my_metric)
```

```
### 8th order
set.seed(2001)
mod_low_8 <- train(y~x+I(x^2)+I(x^3)+I(x^4)+I(x^5)+I(x^6)+I(x^7)+I(x^8), data=prob_lo
w_noise, trControl=my_ctrl, metric=my_metric)
```

```
### 9th order
set.seed(2001)
mod_low_9 <- train(y~x+I(x^2)+I(x^3)+I(x^4)+I(x^5)+I(x^6)+I(x^7)+I(x^8)+I(x^9), data=
prob_low_noise, trControl=my_ctrl, metric=my_metric)
```

2c)

The code chunk below compiles all of the resampling results together for the models associated with the low noise level data.

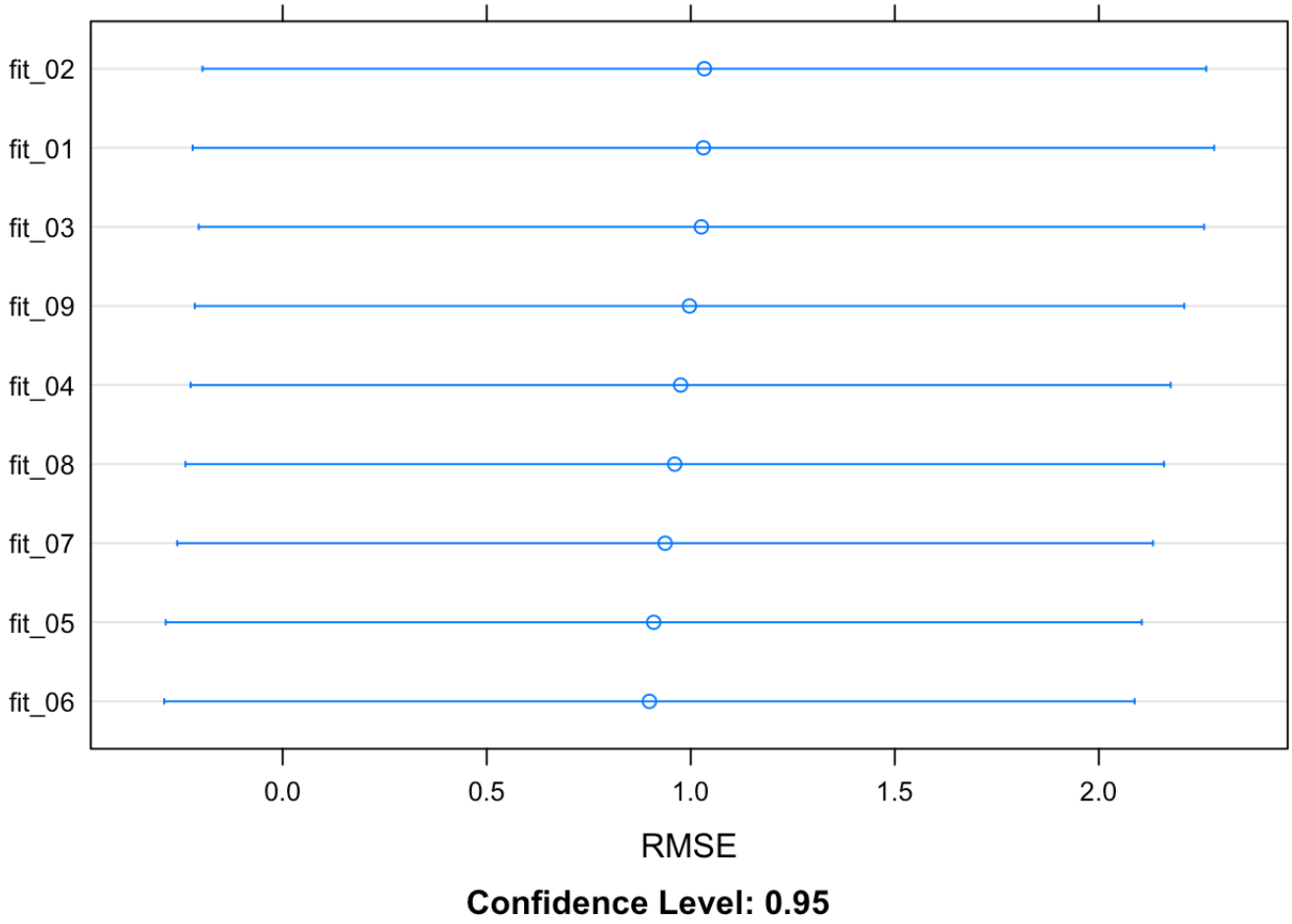
```
low_noise_results = resamples(list(fit_01 = mod_low_1,
                                   fit_02 = mod_low_2,
                                   fit_03 = mod_low_3,
                                   fit_04 = mod_low_4,
                                   fit_05 = mod_low_5,
                                   fit_06 = mod_low_6,
                                   fit_07 = mod_low_7,
                                   fit_08 = mod_low_8,
                                   fit_09 = mod_low_9))
```

As with the `high_noise_results` object, you can now visualize summary statistics associated with the resampling results and identify the best performing models.

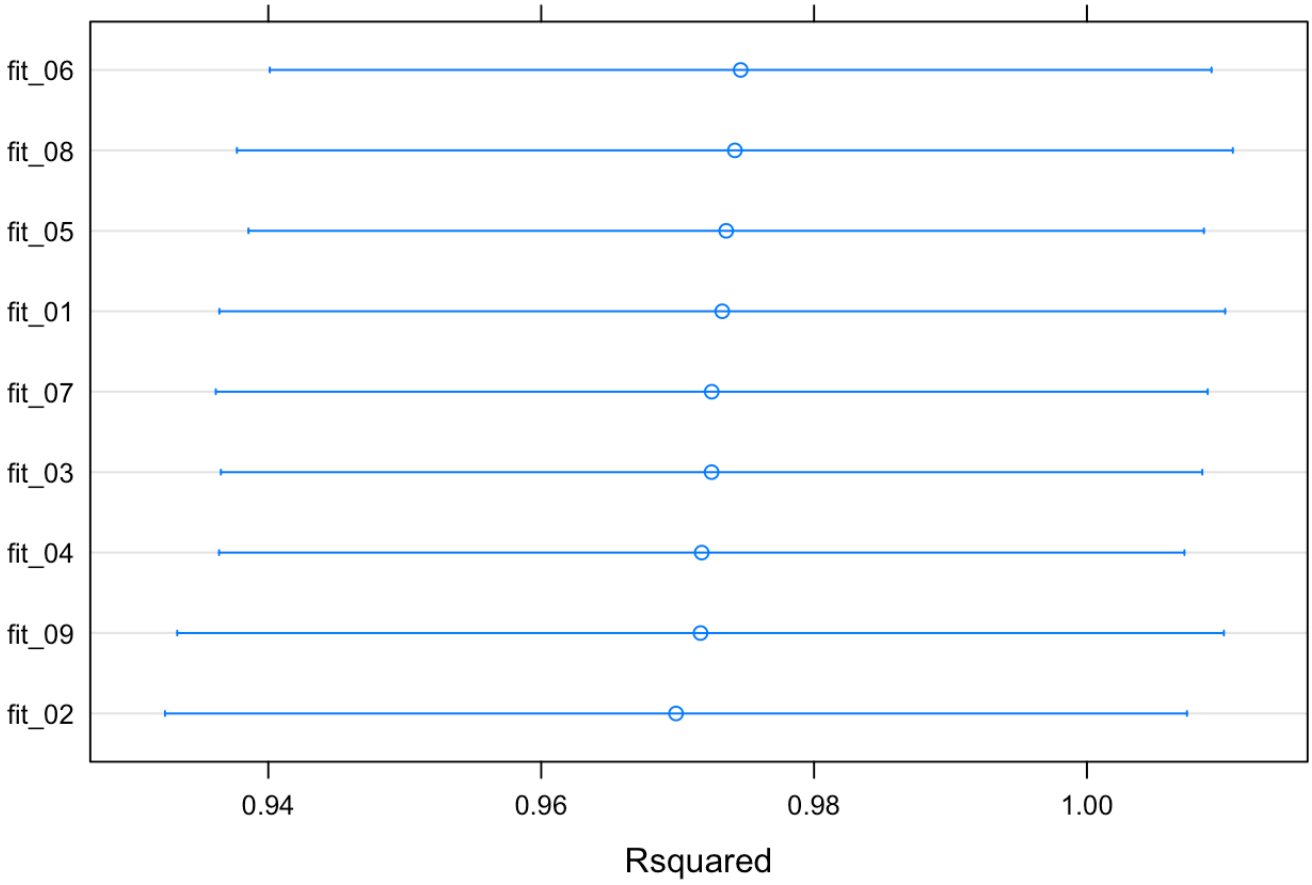
Create two dotplots again, using the same same two performance metrics you selected in Problem 1f). Is there a clear best model now? Did the order of the model performance change compared to what you saw with the high noise level data results?

SOLUTION

```
dotplot(low_noise_results,  metric=my_metric)
```




```
dotplot(low_noise_results,  metric=metric2)
```



Confidence Level: 0.95

Is there a best model? Are the results different from the high level noise results?
There does not appear to be a clear best model, however, there is a difference between the high level noise and low level noise results. All the models with low noise have a much higher R-squared and lower RMSE values. This indicates that there is a better fit of the models with low noise within the data, With fit 6,8,5 having a slighlty higher Rsquared value, which is different than the top 3 models of the high noise data.

2d)

You have now trained and assessed simple to complex models under low and high noise level conditions. As stated earlier, both datasets were generated from the same underlying **data generating process**.

Based on the results in Problem 01 and Problem 02, what impact do you think NOISE has on model training and the relationship to complexity?

SOLUTION

What are your thoughts?
The impact that noise has on a training model is that it introduces a more uncertainty between the relationship of the predictors and the response. It makes the model more complex and also more difficult to determine the true relationship.

Problem 03

The code chunk below reads in a data set that you will work with in Problems 3, 4 and 5. 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`. You will work with this data set for the remainder of the assignment to get experience with binary classification performance metrics.

```
binary_class_data_url <- "https://raw.githubusercontent.com/jyurko/CS_1675_Spring_2022/main/HW/02/hw02_binary_class.csv"

model_pred_df <- readr::read_csv(binary_class_data_url, 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…
```

3a)

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

SOLUTION

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

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

3b)

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 of the observations that correspond to the event of interest. Is the data set a balanced data set?

SOLUTION

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

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

Is the data set balanced?
Yes, the data set is roughly balanced with a slight skew towards an observation being an event.

3c)

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 into 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.

SOLUTION

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

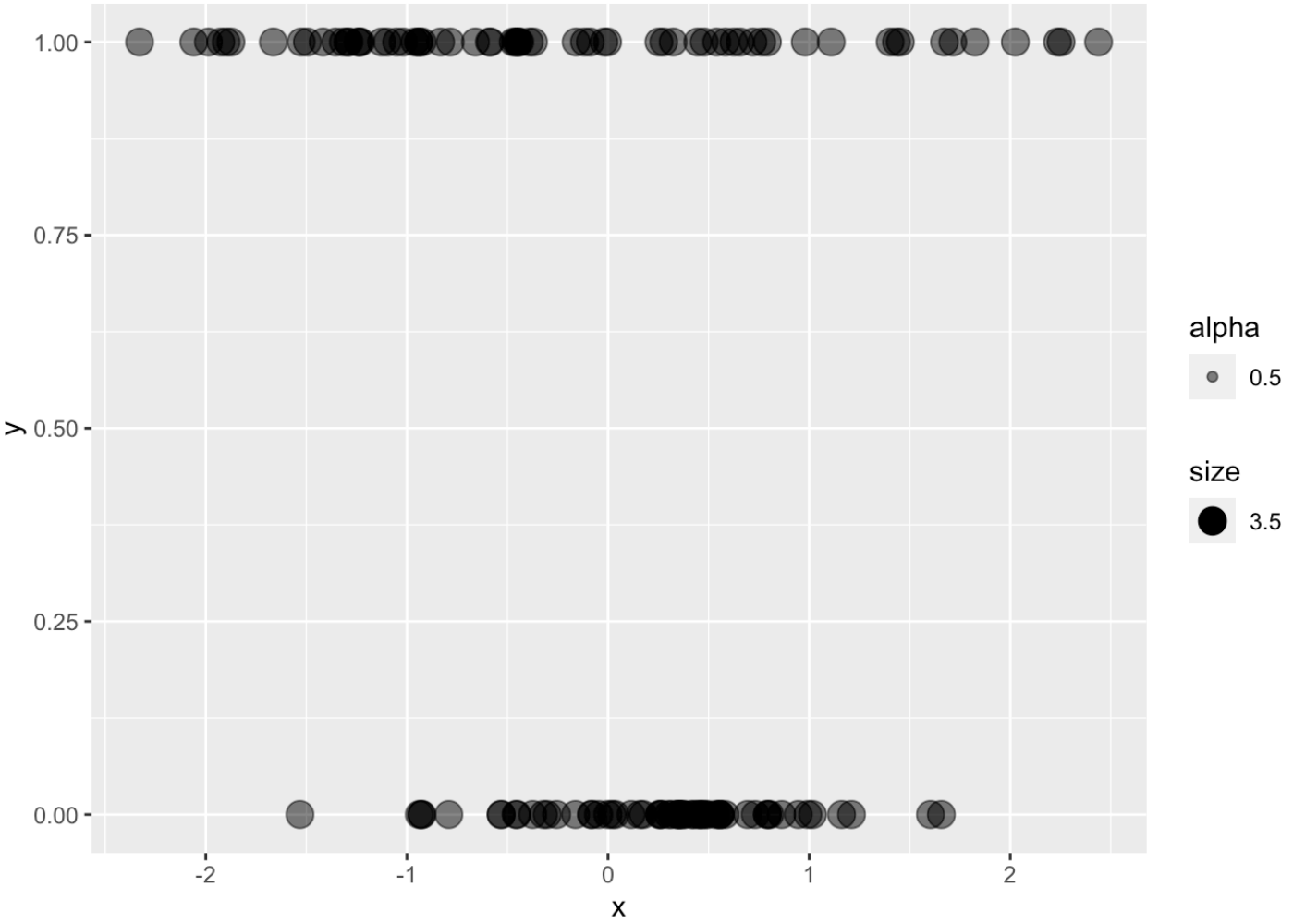
3d)

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.

SOLUTION

```
model_pred_df %>%
  ggplot(mapping = aes(x,y)) + geom_point(aes(size=3.5, alpha=0.5))
```



3e)

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.

SOLUTION

```
model_pred_df %>%
  summary(pred_prob)
```

```
##        x                 pred_prob       obs_class                y          
##  Min.   :-2.329177   Min.   :0.2227   Length:125         Min.   :0.000  
##  1st Qu.:-0.784695   1st Qu.:0.2323   Class :character   1st Qu.:0.000  
##  Median : 0.114752   Median :0.3218   Mode  :character   Median :1.000  
##  Mean   :-0.003672   Mean   :0.4547                      Mean   :0.528  
##  3rd Qu.: 0.582418   3rd Qu.:0.6859                      3rd Qu.:1.000  
##  Max.   : 2.438859   Max.   :0.9988                      Max.   :1.000
```

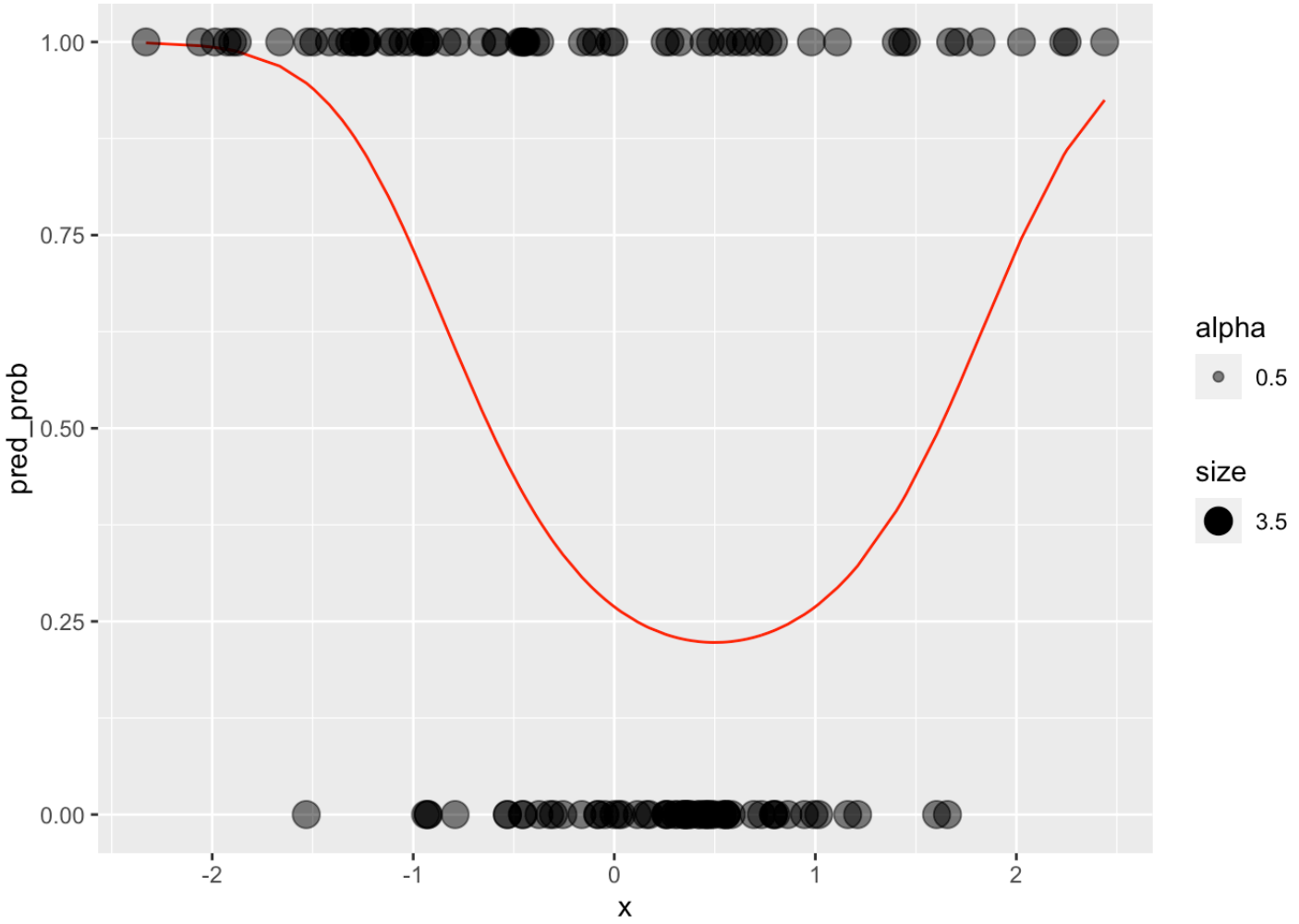
3f)

With the binary outcome encoded as 0/1 with 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 resonse `y` as a scatter plot with `geom_point()`. Use the same marker size and transparency that you used for Problem 3d).

SOLUTION

```
 ggplot(data = model_pred_df) + geom_line(mapping=aes(y = pred_prob, x = x), color = "red") + geom_point(mapping=aes(x,y, size =3.5, alpha = 0.5))
```



3g)

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

SOLUTION

What do you think?
Yes, the binary response does follow the model predicted probability. When the predicted probaiblity is very high, there are many more responses of 1, but when the predicted probability is around 50%, there are responses in both categories of event or non-event. Finally, when the predicted probability is much less than 50%, there appears to be more repsones classified as a non-event.

Problem 04

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

4a)

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`.

SOLUTION

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

4b)

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.

SOLUTION

```
num_correct <- sum(model_class_0.5$obs_class == model_class_0.5$pred_class)
accuracy <- num_correct / length(model_class_0.5$obs_class)
accuracy
```

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

4c)

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?

SOLUTION

```
model_class_0.5 %>%
  count(pred_class, obs_class)
```

```
## # A tibble: 4 × 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.
False-Positives = 6 (event, non_event) True-Negative = 53 (non_event, non_event)

4d)

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.

SOLUTION

```
true_negative_0.5 <- sum(model_class_0.5$obs_class == "non_event" & model_class_0.5$p
red_class == "non_event")
false_positive_0.5 <- sum(model_class_0.5$obs_class == "non_event" & model_class_0.5$
pred_class == "event")
specficity_0.5 <- true_negative_0.5 /(true_negative_0.5 + false_positive_0.5)
fpr_0.5 <- 1- specficity_0.5
fpr_0.5
```

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

```
true_positive <- sum(model_class_0.5$obs_class == "event" & model_class_0.5$pred_clas
s == "event")
false_negative <- sum(model_class_0.5$obs_class == "event" & model_class_0.5$pred_cla
ss == "non_event")
sensitivity_0.5 <- true_positive / (true_positive + false_negative)
sensitivity_0.5
```

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

4e)

We also discussed the ROC curve in addition to the confusion matrix. You will not have to calculate the ROC curve for many threshold values in this assignment. You will go through several calculations in order to get 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`.

SOLUTION

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

4f)

Perform the same action as in Problem 4e), but this time for a threshold value of 0.3. Assign the result to the object `model_class_0.3`.

SOLUTION

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

Problem 5

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

5a)

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.

SOLUTION

```
true_positive_0.7 <- sum(model_class_0.7$obs_class == "event" & model_class_0.7$pred_
class == "event")
accuracy_.07 <- true_positive_0.7 / length(model_class_0.7$obs_class)
accuracy_.07
```

```
## [1] 0.208
```

5b)

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.

SOLUTION

```
false_negative_0.7 <- sum(model_class_0.7$obs_class == "event" & model_class_0.7$pred
_class == "non_event")
sensitivity_0.7 <- true_positive_0.7 / (true_positive_0.7 + false_negative_0.7)
sensitivity_0.7
```

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

```
true_negative_0.7 <- sum(model_class_0.7$obs_class == "non_event" & model_class_0.7$p
red_class == "non_event")
false_positive_0.7 <- sum(model_class_0.7$obs_class == "non_event" & model_class_0.7$
pred_class == "event")
specficity_0.7 <- true_negative_0.7 /(true_negative_0.7 + false_positive_0.7)
fpr_0.7 <- 1 - specficity_0.7
fpr_0.7
```

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

5c)

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

SOLUTION

```
true_positive_0.3 <- sum(model_class_0.3$obs_class == "event" & model_class_0.3$pred_
class == "event")
accuracy_.03 <- true_positive_0.3 / length(model_class_0.3$obs_class)
accuracy_.03
```

```
## [1] 0.384
```

5d)

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.

SOLUTION

```
false_negative_0.3 <- sum(model_class_0.3$obs_class == "event" & model_class_0.3$pred
_class == "non_event")
sensitivity_0.3 <- true_positive_0.3 / (true_positive_0.3 + false_negative_0.3)
sensitivity_0.3
```

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

```
true_negative_0.3 <- sum(model_class_0.3$obs_class == "non_event" & model_class_0.3$p
red_class == "non_event")
false_positive_0.3 <- sum(model_class_0.3$obs_class == "non_event" & model_class_0.3$
pred_class == "event")
specificity_0.3 <- true_negative_0.3 /(true_negative_0.3 + false_positive_0.3)
fpr_0.3 <- 1 - specificity_0.3
fpr_0.3
```

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

5e)

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.

SOLUTION

```
sensitivity_fpr <- data.frame("sensitivity" = c(sensitivity_0.3, sensitivity_0.5, sen
sitivity_0.7),
                              "fpr" = c( fpr_0.3, fpr_0.5, fpr_0.7))
rownames(sensitivity_fpr) <- c("0.3", "0.5", "0.7")

roc <- ggplot(sensitivity_fpr) +geom_abline(slope = 1, intercept = 0) + geom_point(ma
pping = aes(y = sensitivity, x =fpr)) + coord_equal(xlim=c(0,1), ylim=c(0,1))
roc
```