Overview

This assignment is focused on fitting and comparing regression models. Your task is to fit 9 polynomial models from degree 1 (linear relationship) to degree 9 (a 9th degree polynomial). You will interpret behavior through analyzing the coefficient estimates and by visualizing predictions from the models. You must select the best performing model and you will do so several different ways. You will first consider performance on the training set alone, then use a single train/test split, and finally using 5-fold cross-validation.

You will work with 3 different data sets, but all 3 data sets were generated from the same **data generating process**. These 3 different versions of the data will allow you consider the influence of sample size (low vs high) and uncontrollable variation or noise (low vs high) on the model selection process.

IMPORTANT: You are working with 3 data sets in this assignment. This does not mean you must always have multiple "versions" of a data set to complete a machine learning application. This assignment is constructed so that you will see the relationship between sample size and noise on the ability to learn and identify appropriate levels of complexity. The data in this assignment are synthetic data created specifically for this assignment.

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

IMPORTANT!!!

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

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

Load packages

The `tidyverse` is loaded in for you in the code chunk below. The visualization package, `ggplot2`, and the data manipulation package, `dplyr`, are part of the "larger" `tidyverse`.

```r
library(tidyverse)
```

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

The `modelr` package is loaded in for you in the code chunk below. You may use functions from `modelr` to calculate performance metrics for your models.

```r
library(modelr)
```

This assignment also uses functions from the `coefplot` package and uses the major functions from the `caret` package. Those packages will be loaded when necessary later in the assignment.

Problem 01

A data set is loaded for you in the code chunk below. The data are assigned to the `df_low_high` object. The naming convention of this variable represents that the data were generated with low sample size and high noise. Thus, you will begin your predictive modeling or *supervised learning* task with data that are known to be noisy.

```
path_low_high <- 'hw02_lowsize_highnoise.csv'

df_low_high <- readr::read_csv(path_low_high, col_names = TRUE)
```

```
## Rows: 40 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.
```

A glimpse of the data are provided for you below. The variable `x` is the input and the variable `y` is the response. Your 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.

```
df_low_high %>% glimpse()
```

```
## Rows: 40
## Columns: 2
## $ x <dbl> -0.11464944, 0.22502931, 0.37356406, -2.34241739, 1.15847408, 0.3443~
## $ y <dbl> 2.9964689, -0.9068521, -3.9404374, 5.0813180, -6.8506793, 2.5897728,~
```

1a)

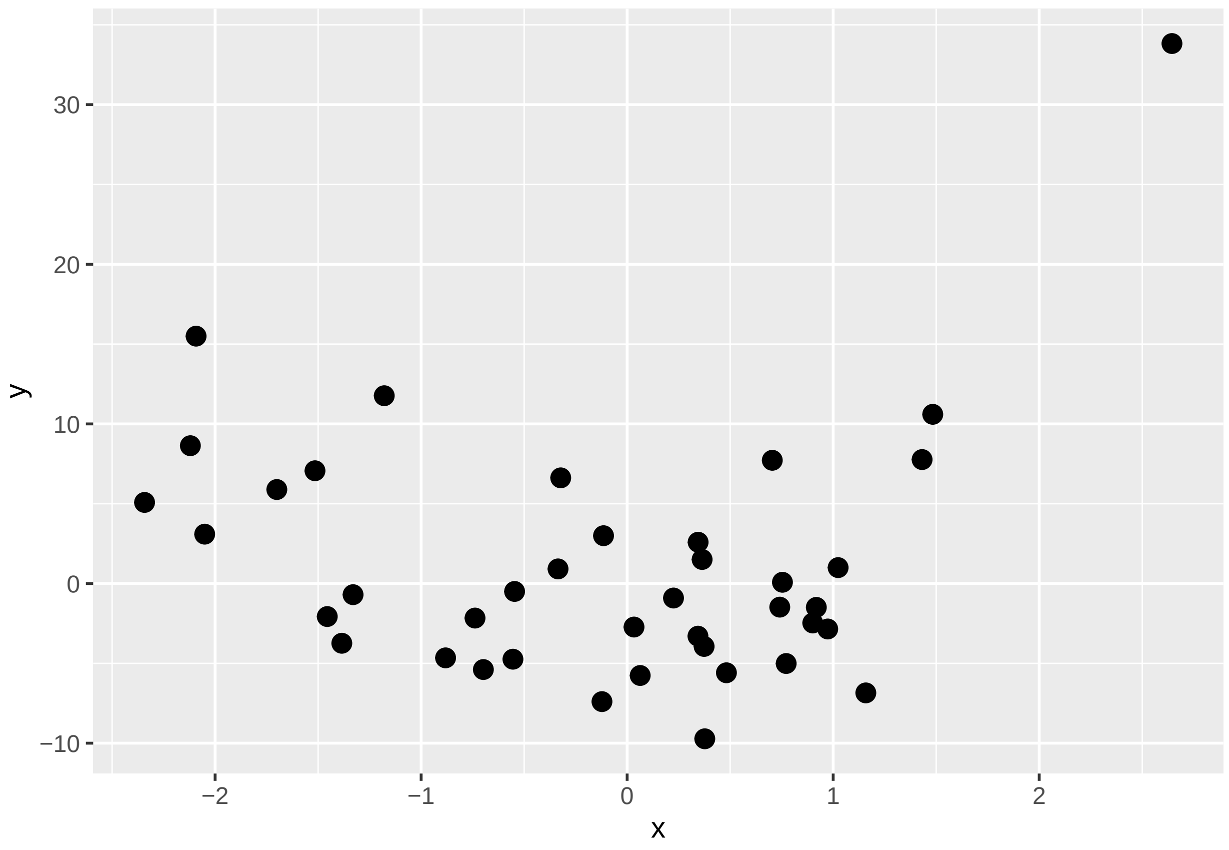
It is always best to explore the data before fitting models. For this assignment, all you are required to do is visualize the relationship between the output and input with a scatter plot.

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. Manually set the marker size to be 3 within the `geom_point()` geom.

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

```

ggplot(data = df_low_high) +
  geom_point(aes(x = x, y = y), size = 3)
```



SOLUTION

Is the relationship linear or non-linear?
The relationship is non-linear.

1b)

It's time fit the models! You must fit 9 models using the `lm()` function and the formula interface. You used both the `lm()` function and the formula interface in the previous assignment. However, this time you must specify the appropriate polynomial features. You are not allowed to use any functions to generate the polynomials. You **MUST** manually type each polynomial feature in the formulas!

Complete the code chunks below by calling the `lm()` function with the appropriate formula and correctly specifying the `data` argument. The result are assigned to variable names defined for you in each of the code chunks. The variable names and the comment within each code chunk specifies the polynomial degree you should use.

You must use the complete data for fitting the models.

```r
### linear relationship (degree 1)
fit_lm_1 <- lm(y ~ x, data =  df_low_high)
```

```r
### quadratic relationship (degree 2)
fit_lm_2 <- lm(y ~ x + I(x^2), data =  df_low_high)
```

```r
### cubic relationship (degree 3)
fit_lm_3 <- lm(y ~ x + I(x^2) + I(x^3), data =  df_low_high)
```

```r
### degree 4
fit_lm_4 <- lm(y ~ x + I(x^2) + I(x^3) + I(x^4), data =  df_low_high)
```

```r
### degree 5
fit_lm_5 <- lm(y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5), data =  df_low_high)
```

```r
### degree 6
fit_lm_6 <- lm(y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5) + I(x^6), data =  df_low_high)
```

```r
### degree 7
fit_lm_7 <- lm(y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5) + I(x^6) + I(x^7), data =  df_low_high)
```

```r
### degree 8
fit_lm_8 <- lm(y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5) + I(x^6) + I(x^7) + I(x^8), data =  df_low_hig
```

```r
### degree 9
fit_lm_9 <- lm(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=df_l
```

SOLUTION

1c)

You fit 9 regression models, it's time to pick the best one! You will initially make your selection based on the training set performance.

Calculate the training set RMSE for each model. It is up to you as to how to organize the RMSE values and display the results. Bare minimum, you must display the RMSE for each model. You are only allowed to use functions from lecture.

```r
rmse_vector <- numeric(9)
rmse_vector[1] <- modelr::rmse(fit_lm_1, df_low_high)
rmse_vector[2] <- modelr::rmse(fit_lm_2, df_low_high)
rmse_vector[3] <- modelr::rmse(fit_lm_3, df_low_high)
rmse_vector[4] <- modelr::rmse(fit_lm_4, df_low_high)
rmse_vector[5] <- modelr::rmse(fit_lm_5, df_low_high)
rmse_vector[6] <- modelr::rmse(fit_lm_6, df_low_high)
rmse_vector[7] <- modelr::rmse(fit_lm_7, df_low_high)
rmse_vector[8] <- modelr::rmse(fit_lm_8, df_low_high)
rmse_vector[9] <- modelr::rmse(fit_lm_9, df_low_high)
```

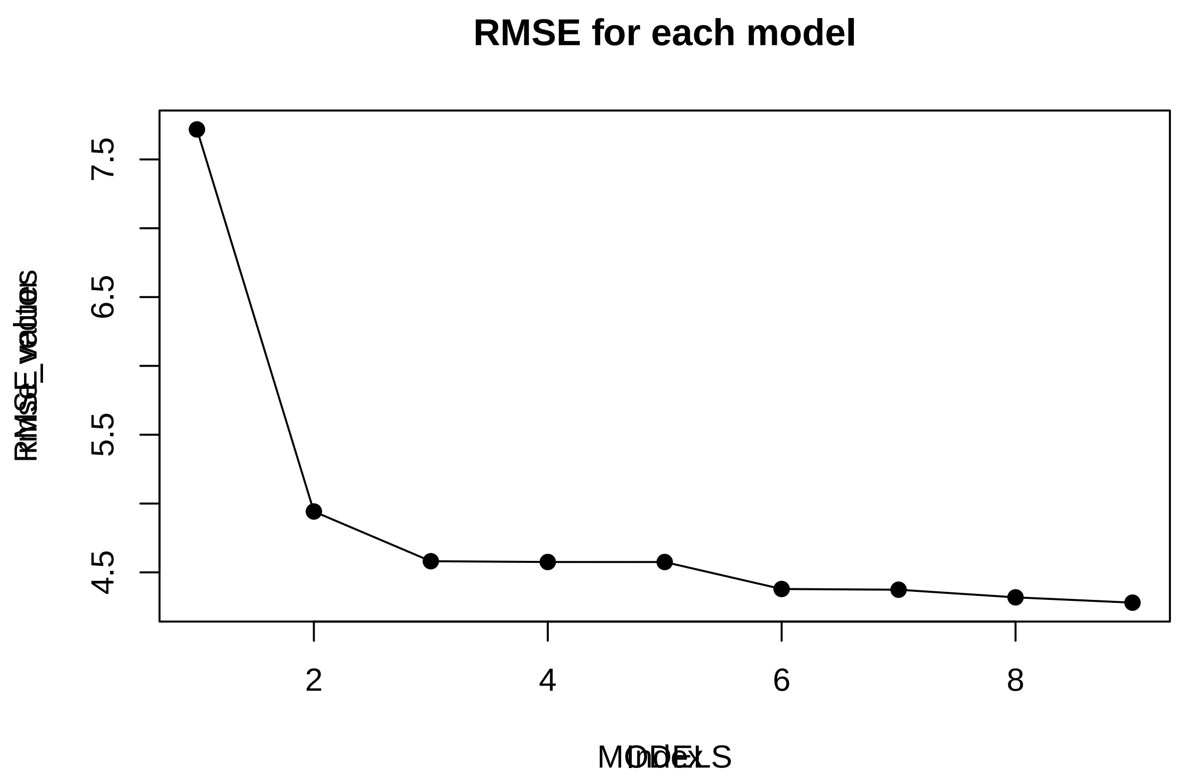
SOLUTION Here we can see the RMSE values as a vector.

```r
print(rmse_vector)
```

```
## [1] 7.717771 4.942079 4.581047 4.575364 4.575193 4.379204 4.373980 4.318560
## [9] 4.279992
```

Here we can see the RMSE values in a plot.

```r
plot(rmse_vector,type = "o",pch = 19)
title(main = "RMSE for each model", xlab = "MODELS", ylab = "RMSE values")
```



1d)

There are many different performance metrics we can use to compare models. For this problem, we will consider R-squared in addition to the RMSE.

Calculate the training set R-squared for each model. It is up to you as to how to organize the R-squared values and display the results. Bare minimum, you must display the R-squared for each model. You are only allowed to use functions from lecture.

```r
rsqrd_vector <- numeric(9)
rsqrd_vector[1] <- modelr::rsquare(fit_lm_1, df_low_high)
rsqrd_vector[2] <- modelr::rsquare(fit_lm_2, df_low_high)
rsqrd_vector[3] <- modelr::rsquare(fit_lm_3, df_low_high)
rsqrd_vector[4] <- modelr::rsquare(fit_lm_4, df_low_high)
rsqrd_vector[5] <- modelr::rsquare(fit_lm_5, df_low_high)
rsqrd_vector[6] <- modelr::rsquare(fit_lm_6, df_low_high)
rsqrd_vector[7] <- modelr::rsquare(fit_lm_7, df_low_high)
rsqrd_vector[8] <- modelr::rsquare(fit_lm_8, df_low_high)
rsqrd_vector[9] <- modelr::rsquare(fit_lm_9, df_low_high)
```

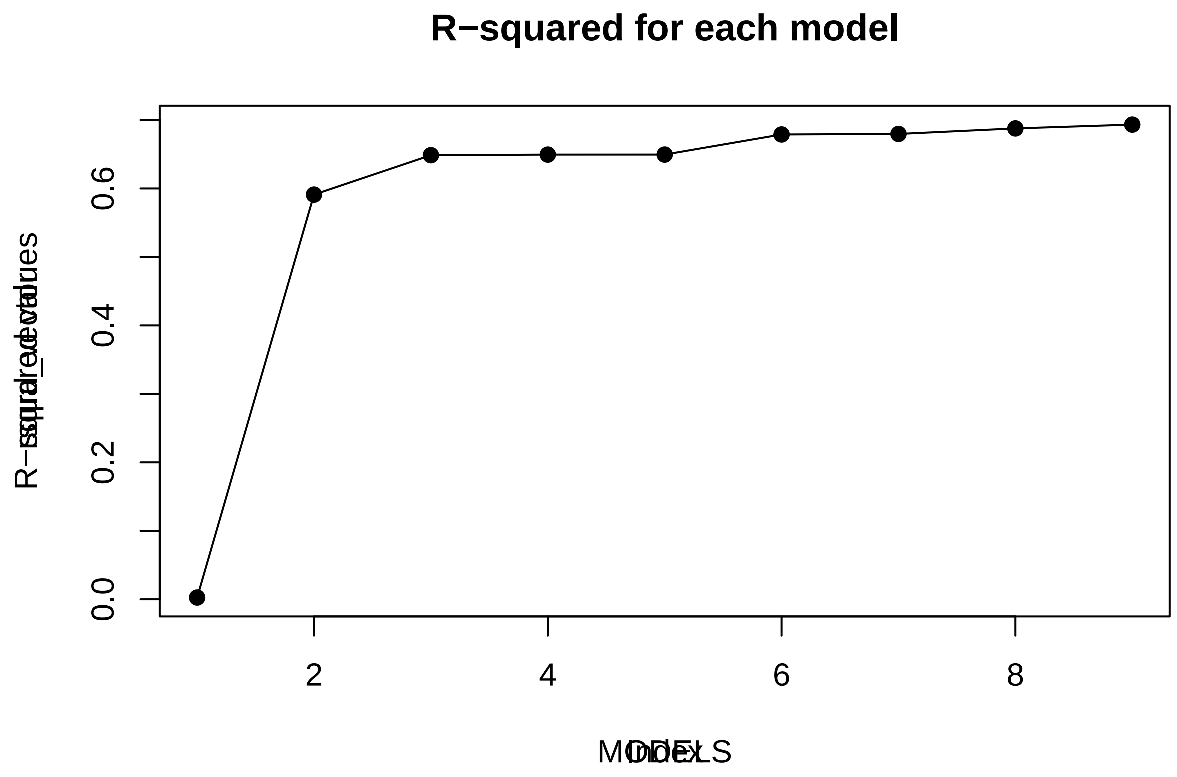
SOLUTION Here we can see the R-squared values as a vector.

```r
print(rsqrd_vector)
```

```
## [1] 0.002595816 0.591015490 0.648587647 0.649458990 0.649485284 0.678872274
## [7] 0.679638019 0.687704829 0.693257924
```

Here we can see the RMSE values in a plot.

```r
plot(rsqrd_vector,type = "o",pch = 19)
title(main = "R-squared for each model", xlab = "MODELS", ylab = "R-squared values")
```



1e)

Which model is the best according to the training set performance metrics? Why did you make the selection that you did? Do the two performance metrics agree on the best model?

SOLUTION The best model according to the training set performance metrics is the 9th model (the model which is using the 9 degree polynomial), because it has the lowest RMSE value and its R-squared value is the one that is closest to 1. (This also says that the two performance metrics agree on the best model).

1f)

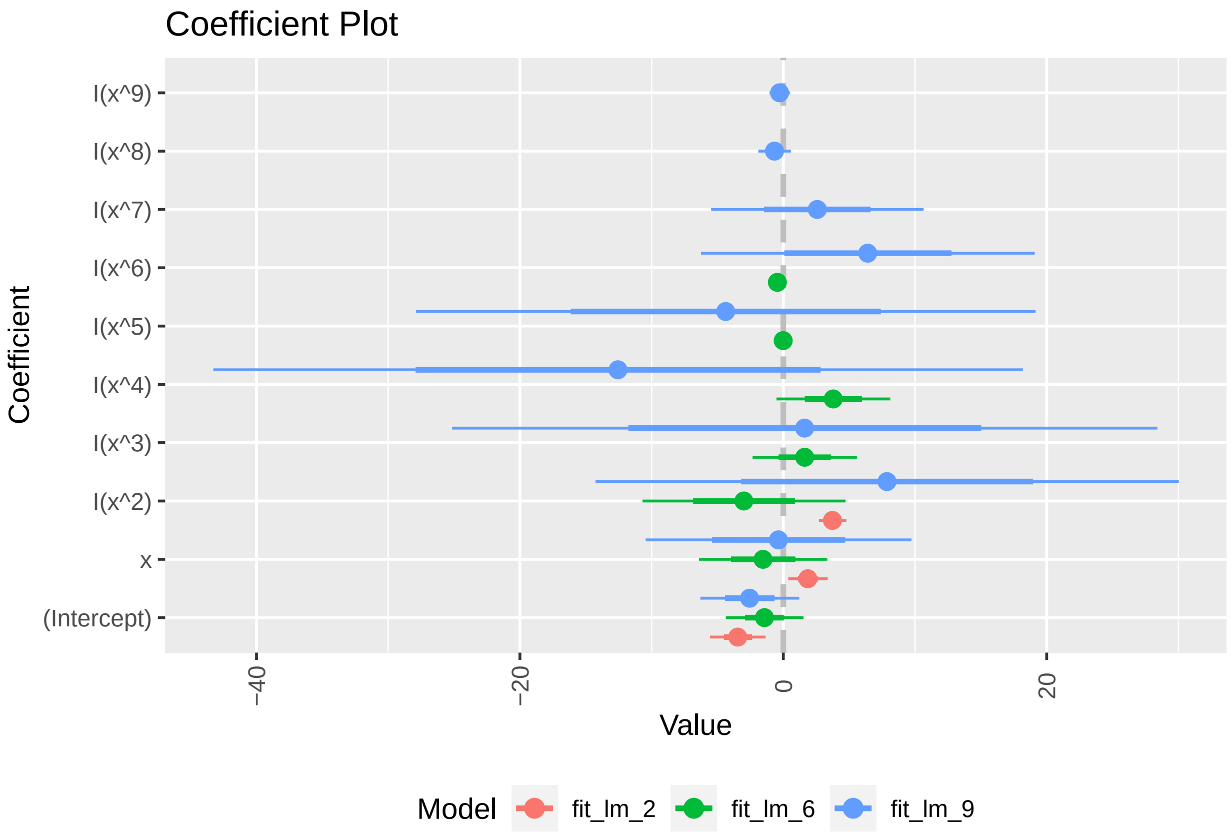
Let's dive deeper into a few of the models. You will visualize the coefficient summaries for three models to examine the number of statistically significant features and to get an idea of the uncertainty on the coefficients. Regardless of your answer to Problem 1e), you must examine the coefficients for the quadratic model (degree 2), the 6th degree model, and the 9th degree model.

You used the `coefplot()` function from the `coefplot` package to visualize model coefficient summaries in the previous assignment. You could call `coefplot::coefplot()` three different times, one for each model. However, the `coefplot` package has a useful "helper" function dedicated to visualizing the coefficients across multiple models. This function is named `coefplot::multiplot()`. The `coefplot::multiplot()` syntax is similar to `coefplot::coefplot()`. You pass in the model objects as arguments to the function. You must separate each model with a comman, `,`, within the function call. The `coefplot::multiplot()` function takes care of everything for you.

Visualize the coefficient summaries for the quadratic (degree 2), 6th degree, and 9th degree polynomial models using the `coefplot::multiplot()` function.

You do NOT need to load the `coefplot` package. The `::` operator allows accessing the function `multiplot()` from within the `coefplot` library.

```r
coefplot::multiplot(fit_lm_2,fit_lm_6,fit_lm_9)
```



SOLUTION

1g)

Some of the coefficients are present in all three models displayed in the figure from Problem 1f), while a few of the coefficients are only present in one of the models. Let's look closely at the previous figure to try and interpret the model behavior.

Are the coefficient estimates similar for those coefficients present in at least 2 of the models visualized in the previous figure?

Is the level or amount of uncertainty associated with the coefficients consistent across the three models?

SOLUTION

1) The coefficients are not similar. For examole in model-9, the coefficient of xˆ4 is close to -12, however in model-6, the coefficient of xˆ4 is close to 3. Similarly, when we look at the coefficient of x, in models 6 and 9 the values are negative, while in model-2 the value is positive.
2) The level of uncertainty with the coefficients is not consistent across the three models. For example, in model-9, the coefficient of x has a larger confidence interval compared to the confidence intervals of the coefficients of x in the other two models.

Problem 02

We discussed in lecture the importance of visualizing predictive model trends. This is an important and useful way of interpreting model behavior because it allows us to "see" what the model "thinks" happens at input values **not** present in the original training set. Creating such visualizations requires creating a new data set. Such data sets can require careful planning when there are dozens to hundreds of inputs. However, our present application has a single input and thus it is easy to define a meaningful *input grid* to support visualizing trends.

2a)

You must create the input grid to support visualization by defining a variable `x` within a data.frame (tibble) that consists of 201 evenly spaced points between the training set minimum and maximum values.

The code chunk below is started for you. The `tibble::tibble()` function defines a tibble object. The data variable `x` is assigned within that tibble. You must complete that line of code by using an appropriate function which creates a sequence of values from a lower bound to an upper bound. You must use 201 evenly spaced points.

The lower bound must equal the input training set minimum value and the upper bound must equal the input training set maximum value.

HINT: The lecture slides shows how to create this kind of object...

```
input_viz <- tibble::tibble(
  x = seq(min(df_low_high$x),max(df_low_high$x),length.out=201)
)
```

SOLUTION If you created `input_viz` correctly, it should have 201 rows. The code chunk below checks the number of rows for you.

```
input_viz %>% nrow()
```

```
## [1] 201
```

2b)

You must now make predictions for each model! In lecture, we discussed the importance including two types of uncertainty when we make predictions. However, since we are at the beginning of the semester, you are only responsible for making predictions of the **trend** or **average response**. Your returned predictions will therefore consist of numeric vectors.

Complete the code chunks below by calling the appropriate function for making predictions for each model. The returned predictions are assigned must be assigned to the variables named in the code chunk below. The comments and the variable names specify which model should be used.

HINT: Pay close attention to which data set you use when making the predictions...

```
### linear relationship (degree 1) predictions
viz_trend_1 <- predict(fit_lm_1,input_viz)
```

```
### quadratic relationship (degree 2) predictions
viz_trend_2 <- predict(fit_lm_2,input_viz)
```

```
### cubic relationship (degree 3) predictions
viz_trend_3 <- predict(fit_lm_3,input_viz)
```

```r
### degree 4
viz_trend_4 <- predict(fit_lm_4,input_viz)
```

```r
### degree 5
viz_trend_5 <- predict(fit_lm_5,input_viz)
```

```r
### degree 6
viz_trend_6 <- predict(fit_lm_6,input_viz)
```

```r
### degree 7
viz_trend_7 <- predict(fit_lm_7,input_viz)
```

```r
### degree 8
viz_trend_8 <- predict(fit_lm_8,input_viz)
```

```r
### degree 9
viz_trend_9 <- predict(fit_lm_9,input_viz)
```

SOLUTION

2c)

Let's manipulate the data before visualizing the predictions. This way we can organize the data into a "tidy format" to support proper visualization strategies.

The code chunk below is started for you. The `input_viz` object is piped into the `mutate()` function where two data variables are assigned, `pred_trend` and `poly_degree`. The predictions made in the previous question will be assigned to the `pred_trend` data variable. You will manually assign the `poly_degree` variable for the model which made the predictions. Please note that you only need to provide a scalar value for `poly_degree`. The `mutate()` function will make sure to broadcast that scalar value to all rows in the tibble.

Let's start by compiling the `df_viz_1` object which stores the prediction input and predicted trend for the linear relationship model.

Complete the code chunk below by assigning the linear relationship model's predicted trend to the `pred_trend` data variable and assign the value `1` to the `poly_degree` data variable. The `df_viz_1` object is assigned for you and a glimpse is displayed to the screen. If you created the `df_viz_1` object correctly it should consist of 3 variable (columns) and 201 rows.

```r
df_viz_1 <- input_viz %>%
  mutate(pred_trend = viz_trend_1,
         poly_degree = 1)

df_viz_1 %>% glimpse()
```

SOLUTION

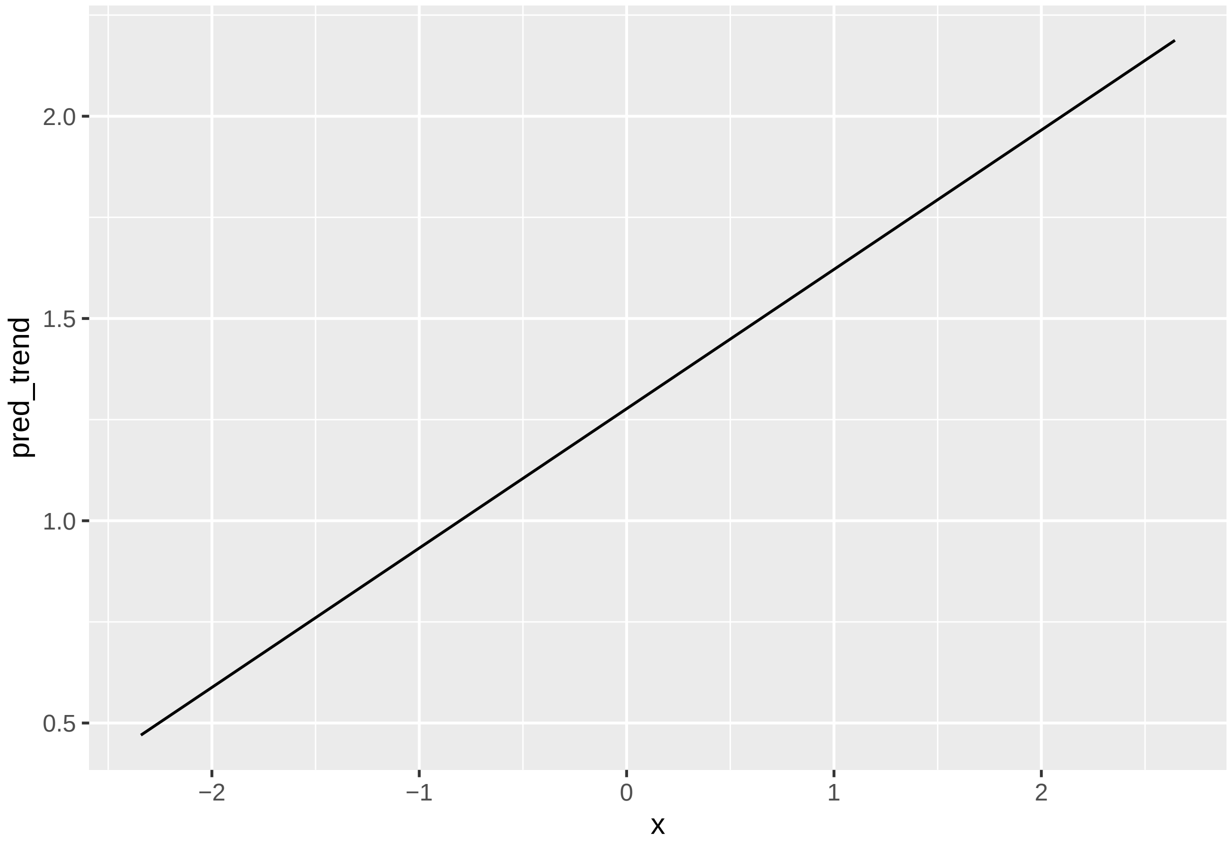
```
## Rows: 201
## Columns: 3
## $ x           <dbl> -2.342417, -2.317484, -2.292550, -2.267616, -2.242682, -2.~
## $ pred_trend  <dbl> 0.4700900, 0.4786779, 0.4872658, 0.4958537, 0.5044415, 0.5~
## $ poly_degree <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
```

2d)

You can now visualize the predicted trend for the linear relationship! You will use a line to visualize the trend so that way the smooth sequential trend is visualized.

Pass the `df_viz_1` tibble into `ggplot()`. Map the `x` aesthetic to the `x` variable within the "parent" `ggplot()` call. Add the `geom_line()` layer and map the `y` aesthetic to the `pred_trend` variable within the `geom_line()` geom. Thus, you should NOT map both `x` and `y` aesthetics in the "parent" `ggplot()` call.

```
ggplot(data = df_viz_1, aes(x=x))+
  geom_line(aes(y = pred_trend))
```



SOLUTION

2e)

Let's add further context to the predictive trend visualization by overlaying the original training set scatter plot. We thus need to create a graphic that uses two different data sets. This is ok! Each `geom_*` function has a `data` argument. By default, the `geom_*` function, (such as `geom_point()` or `geom_line()`) inherits the `data` provided to the "parent" `ggplot()` call. However, we can override this behavior by manually specifying a new data set to the `data` argument within the `geom_*` layer! This effectively allows us to show behavior coming from multiple data sets.

Visualize the predicted trend associated with the linear relationship again. Pass the `df_viz_1` tibble into `ggplot()`. Map the `x` aesthetic to the `x` variable within the "parent" `ggplot()` call. However, before adding `geom_line()` add `geom_point()` where you specify the `data` argument in `geom_point()` to be `df_low_high`. Map the `y` aesthetic within `geom_point()` to the `y` variable and

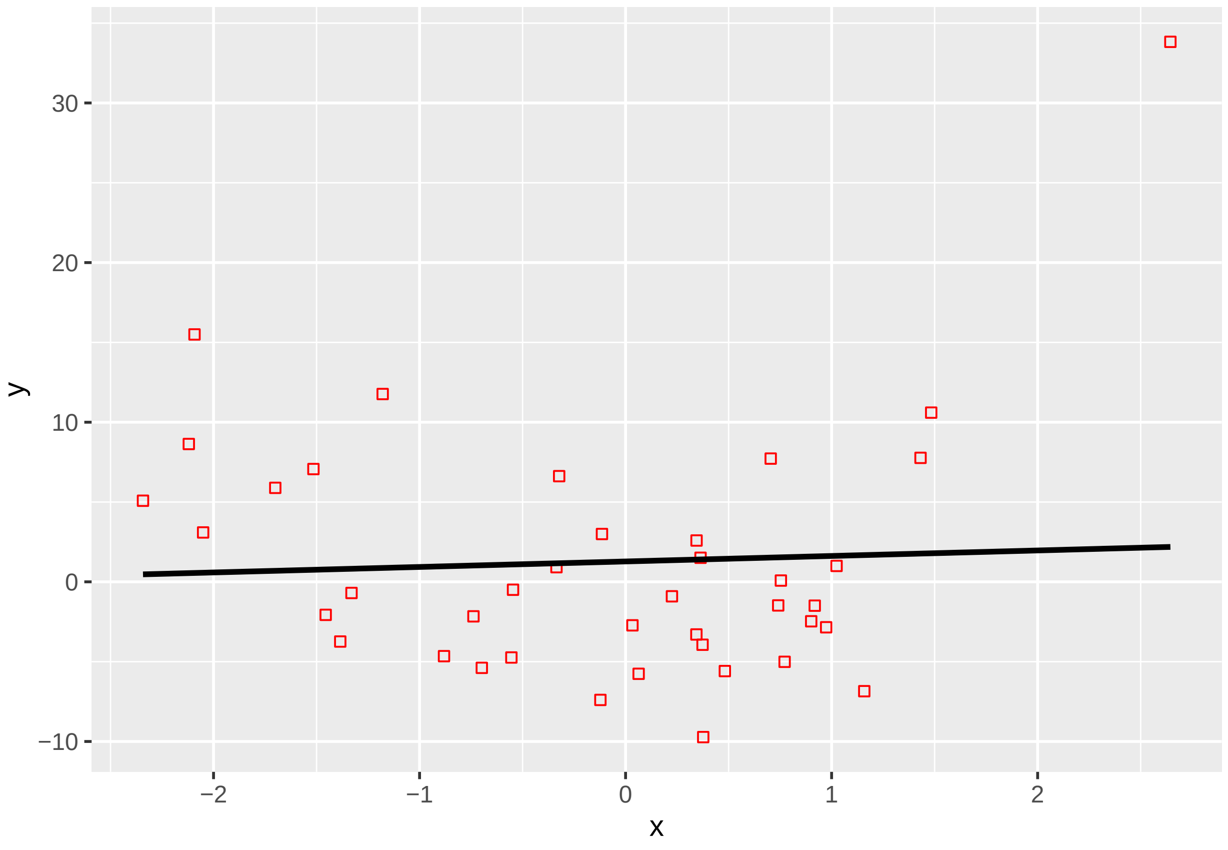
manually assign the marker `shape` to 0 and the marker `color` to `'red'`. Add the `geom_line()` layer where you map the `y` aesthetic to the `pred_trend` variable and manually assign the line `size` to 1.

Your completed graphic should display the original training set as red open squares and the predicted trend as a black line.

```r
ggplot(data = df_viz_1, aes(x=x))+
  geom_point(data = df_low_high, aes(y=y),color = 'red',shape=0) +
  geom_line(aes(y = pred_trend), size = 1)
```

SOLUTION

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



2f)

You visualized the predicted trend associated with 1 model! You must now repeat these steps for the remaining 8 models. You will start by properly organizing the predictive trends from the other 8 models into their own tibbles.

Complete the code chunks below. You must assign the predictive trends to the `pred_trend` data variable and assign the correct value for the `poly_degree` data variable. The comments and variable names state the models associated with each code chunk.

NOTE: This is NOT the most efficient way to properly organize the predictions. More streamlined approaches make use of **functional programming** techniques to create one properly organized and "tidy" data set in just a few lines of code. We will learn how to do that later in the semester. For now though, we will go through the tedious way!

```r
### quadratic relationship (degree 2)
df_viz_2 <- input_viz %>%
  mutate(pred_trend = viz_trend_2,
         poly_degree = 2)
```

```r
### cubic relationship (degree 3)
df_viz_3 <- input_viz %>%
  mutate(pred_trend = viz_trend_3,
         poly_degree = 3)
```

```r
### degree 4
df_viz_4 <- input_viz %>%
  mutate(pred_trend = viz_trend_4,
         poly_degree = 4)
```

```r
### degree 5
df_viz_5 <- input_viz %>%
  mutate(pred_trend = viz_trend_5,
         poly_degree = 5)
```

```r
### degree 6
df_viz_6 <- input_viz %>%
  mutate(pred_trend = viz_trend_6,
         poly_degree = 6)
```

```r
### degree 7
df_viz_7 <- input_viz %>%
  mutate(pred_trend = viz_trend_7,
         poly_degree = 7)
```

```r
### degree 8
df_viz_8 <- input_viz %>%
  mutate(pred_trend = viz_trend_8,
         poly_degree = 8)
```

```r
### degree 9
df_viz_9 <- input_viz %>%
  mutate(pred_trend = viz_trend_9,
         poly_degree = 9)
```

SOLUTION

2g)

Do not worry, you will not create separate plots for each model's predictions! Instead, you will concatenate the 9 tibbles together into a single tibble! You will "stack" the tibbles vertically by *binding the rows* together. This operation works because all 9 `df_viz_*` objects have the same columns! A little planning goes a long way!

You can vertically concatenate or bind rows via the `bind_rows()` function from `dplyr`. You must provide all data.frames (tibbles) you wish to bind together as arguments to `bind_rows()`. Each data set (argument to the function) must be separated by a comma, `,`. You do **not** need to worry about the `.id` argument to `bind_rows()` for this current operation.

The code chunk below creates a variable `df_viz_all`. You must use the `bind_rows()` function to bind or stack together all 9 visualization tibbles.

HINT: If you created the `df_viz_all` object correctly, it should consist of 1809 rows and 3 columns.

```r
df_viz_all <- bind_rows(df_viz_1, df_viz_2, df_viz_3, df_viz_4, df_viz_5, df_viz_6, df_viz_7, df_viz_8,
df_viz_all %>% glimpse()
```

SOLUTION

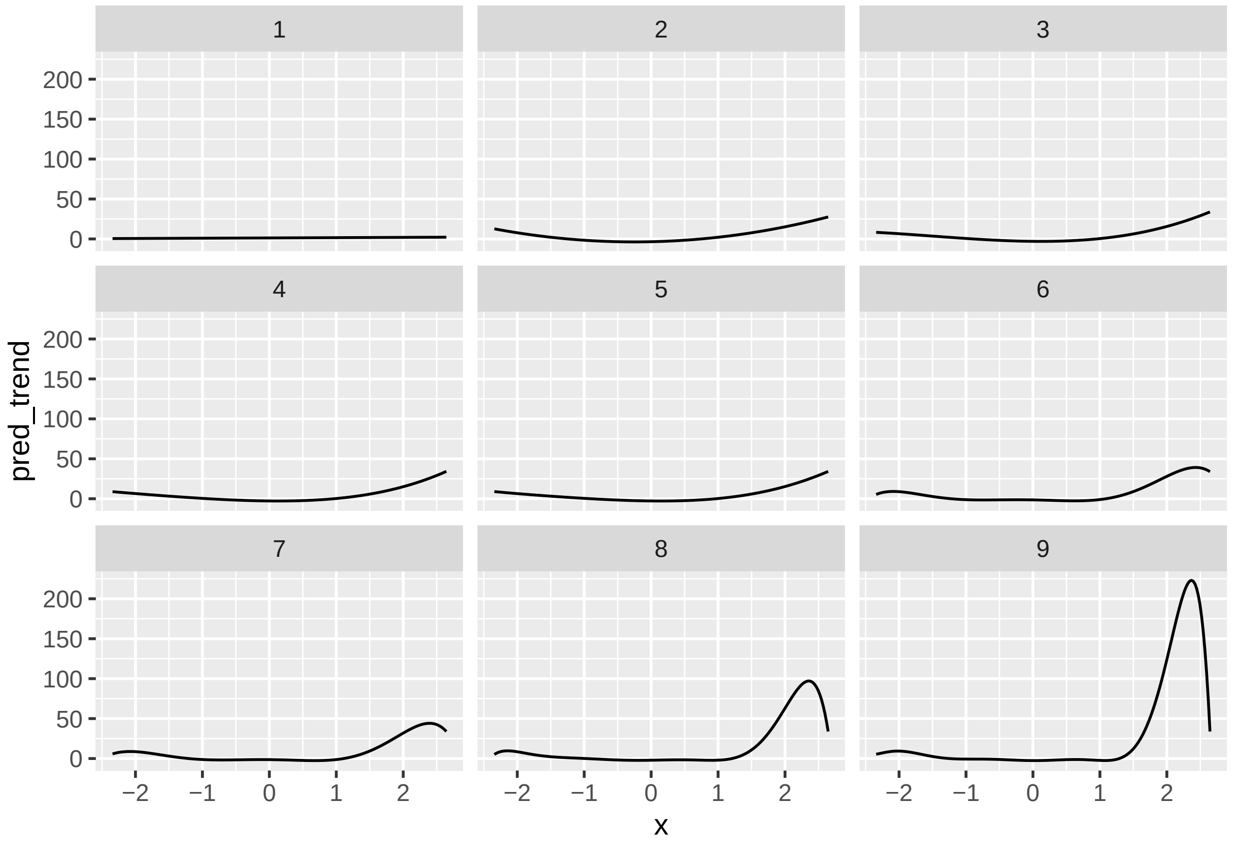
```
## Rows: 1,809
## Columns: 3
## $ x           <dbl> -2.342417, -2.317484, -2.292550, -2.267616, -2.242682, -2.~
## $ pred_trend  <dbl> 0.4700900, 0.4786779, 0.4872658, 0.4958537, 0.5044415, 0.5~
## $ poly_degree <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
```

2h)

The `df_viz_all` is tidy. A column (variable) exists for each attribute associated with the data. A variable exists for the input, `x`, a variable exists for the predicted trend, `pred_trend`, and an identifier exists which identifies which model made the predictions, `poly_degree`. You will now use all variables to create a faceted figure showing the predictions associated with each model.

Pipe the `df_viz_all` object to `ggplot()` and map the `x` aesthetic to the `x` variable within the "parent" `ggplot()` call. Add the `geom_line()` layer and map the `y` aesthetic to the `pred_trend` variable within `geom_line()`. Add the facets via the `facet_wrap()` function. The facets must be a function of the `poly_degree` variable.

```r
df_viz_all %>%
  ggplot(aes(x=x)) +
  geom_line(aes(y=pred_trend)) +
  facet_wrap(~poly_degree)
```



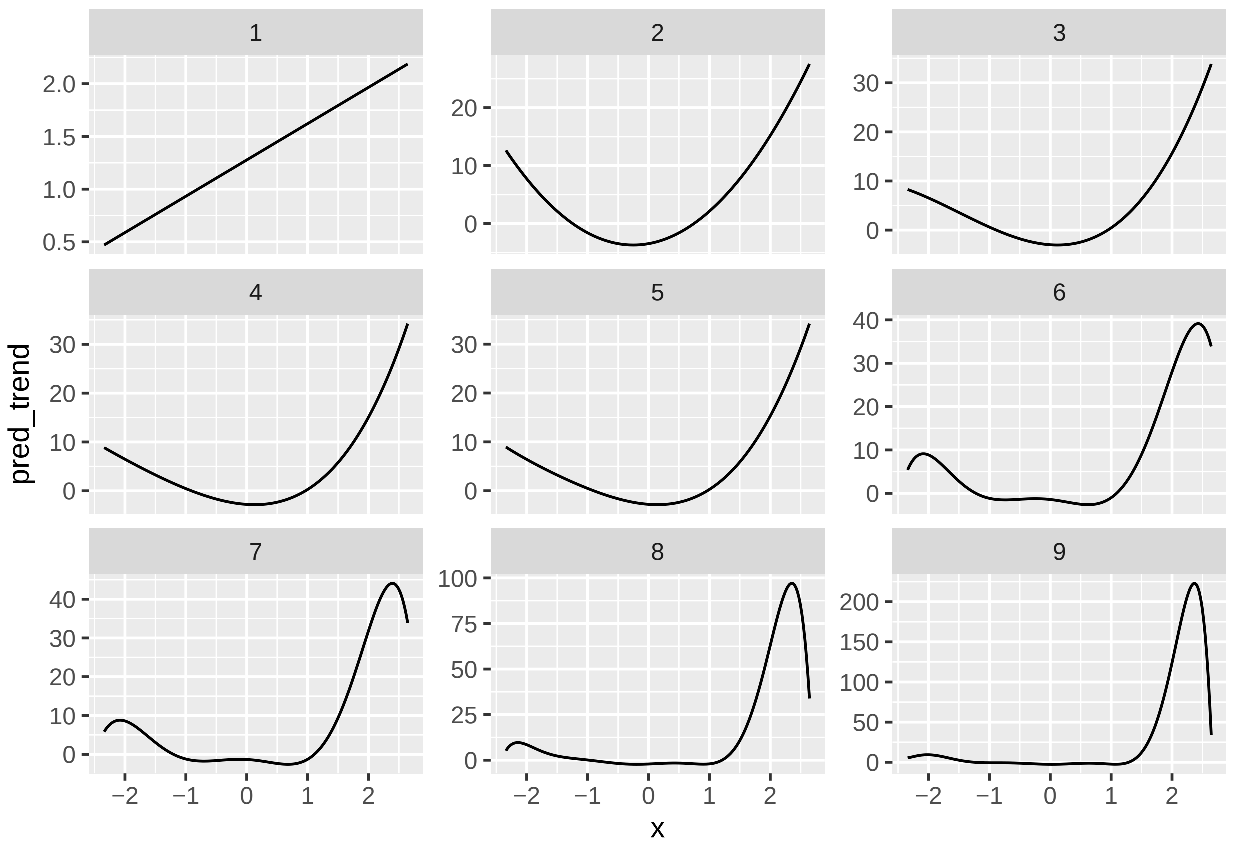
SOLUTION

2i)

It might be difficult to see what's going on within each facet. Let's "zoom" the y-axis within each facet to give a better picture of the trend associated with each model.

Repeat the visualization from the previous problem, but this time set `scales = 'free_y'` within the `facet_wrap()` function.

```r
df_viz_all %>%
  ggplot(aes(x=x)) +
  geom_line(aes(y=pred_trend)) +
  facet_wrap(~poly_degree,scales = 'free_y')
```



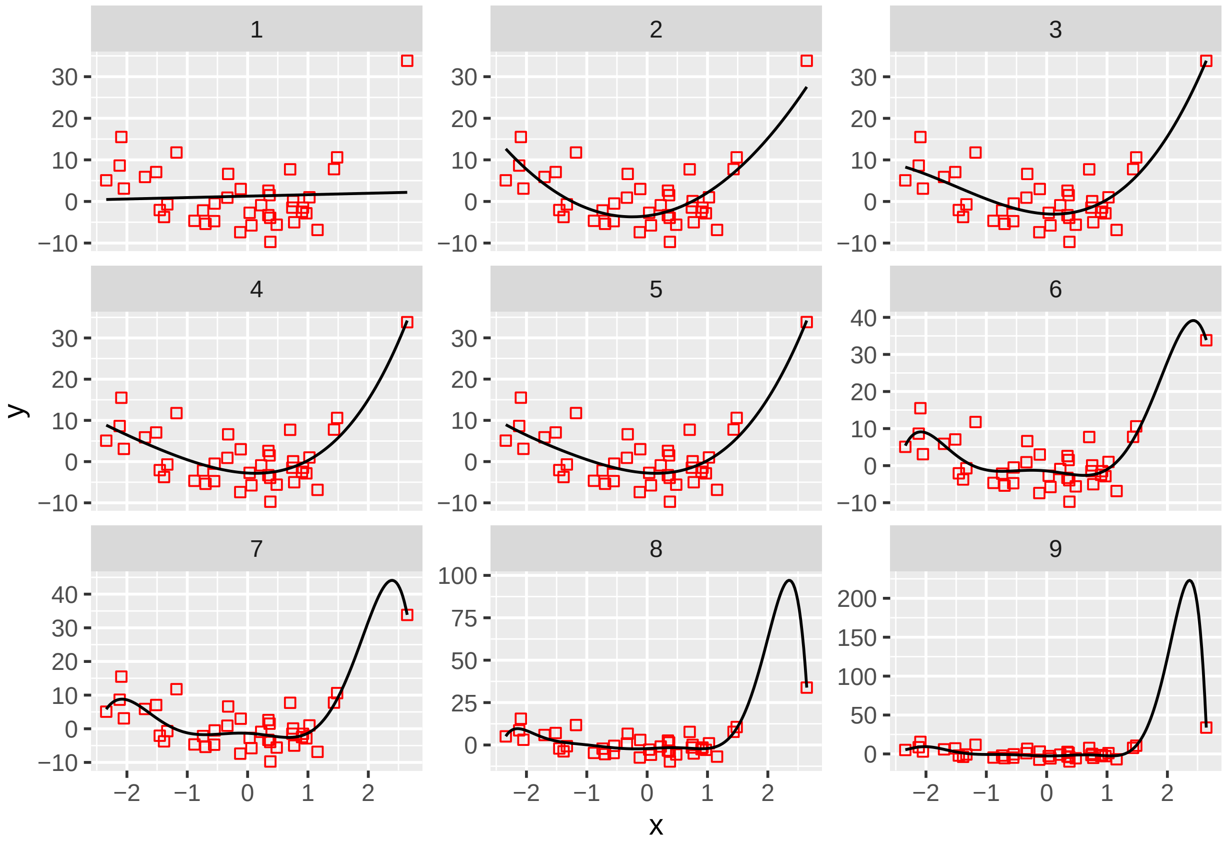
SOLUTION

2j)

Lastly, let's overlay the training set for additional context within each facet. To do so, you must override the `data` argument within the additional `geom_point()` layer, just as you did in Problem 2e).

Repeat the visualization from the previous problem, however this time include `geom_point()` in between the "parent" `ggplot()` call and the `geom_line()` layer. You must override the `data` argument within this `geom_point()` layer by assigning `data` to `df_low_high`. You must map the `y` aesthetic directly within `geom_point()` to the `y` variable. Manually specify the marker `shape` to be `0` and the marker `color` to be `'red'` within `geom_point()`.

```r
df_viz_all %>%
  ggplot(aes(x=x)) +
  geom_point(data = df_low_high, aes(y=y), shape = 0, color = 'red') +
  geom_line(aes(y=pred_trend)) +
  facet_wrap(~poly_degree,scales = 'free_y')
```



SOLUTION

2k)

You previously used two performance metrics to identify the best model. How would you describe the predictive trends associated with the best performing model? Do the trends seem consistent with the training set?

SOLUTION

1) The best model from the two performance metrics was the 9th model. Its values are looking close to the training set at the input values of the training set, however, when the training set values are sparse or jump, the predictive trends become bumpy, which is not an inducation for a good performance.

2) As the degree of the polynomial increases, trends become more consistent with the training test.

Problem 03

You previously selected the best model based on the **training set** alone. Your selection therefore did not consider how well the model will perform on new data! You will now try to identify the best model in hopes of finding one that *generalizes* to new data.

You will begin by creating a single training/test split. Sometimes this procedure is referred to as a training/validation split, but the motivation and goals are the same. You will randomly partition the data into a dedicated training set and dedicated hold-out set. You will train the models on the training set, and assess their performance on the hold-out set.

3a)

You will use an 80/20 train/test split. Thus, 80% of the data will be used for training the models and 20% of the data will be used to test the models. You will use the base R `sample()` function to create the vector of row indices that will serve as the training set.

The random seed is specified for you with the `set.seed()` function in the code chunk below. This ensures that the data splitting is **reproducible**. You will therefore get the same split every time you run the code chunk below. Please note, you must run the entire code chunk to ensure reproducibility.

Complete the code chunk below. Use the `sample()` function to randomly sample 80% of the rows from the `df_low_high` data set. The `sample()` function randomly selects elements from a provided vector. You must therefore provide a vector that starts at 1 and ends at the number of rows in `df_low_high`. The result of `sample()` must be assigned to the `id_train` variable.

You must then use `id_train` to slice the training set from `df_low_high` and slice all except the `id_train` indices to create the hold-out test set. The training set is `df_train` and the hold-out test set is `df_holdout` in the code chunk below.

```r
set.seed(202214)
id_train <- sample(1:nrow(df_low_high),0.8 * nrow(df_low_high))
df_train <- df_low_high %>% slice(id_train)
df_holdout <- df_low_high %>% slice(-id_train)
```

SOLUTION

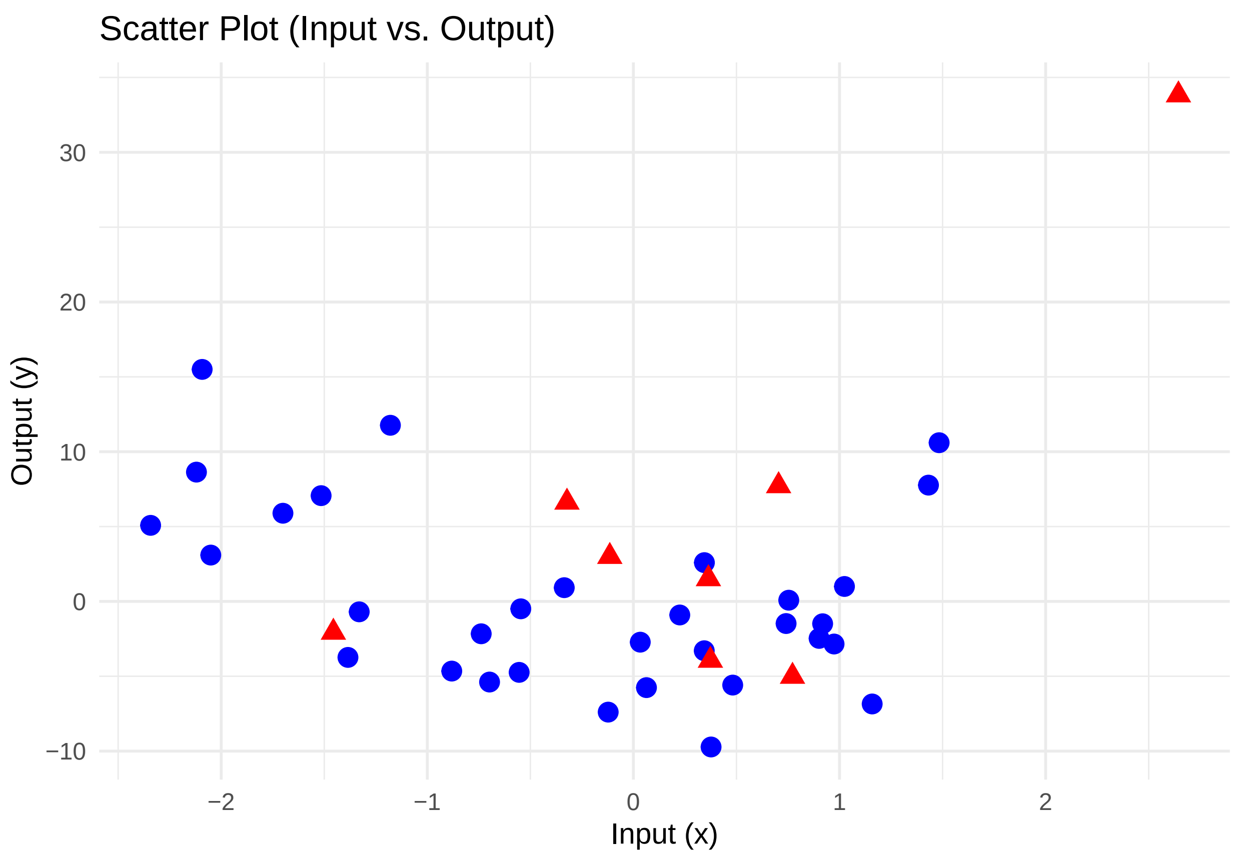
3b)

Let's visualize the data split so we know which observations were selected for training and which observations were selected for testing.

Create a scatter plot using `ggplot2` which visualizes the relationship between the output `y` and the input `x`. You must use the marker color and marker shape to denote the training set from the hold-out test set.

There are multiple ways this figure can be created. It is up to you to make it, but you must create the figure using `ggplot2`. Use this as an opportunity to test out your understanding of the elements of the `ggplot2` philosophy for creating statistical visualizations.

```r
ggplot() +
  geom_point(data = df_train, aes(x = x, y = y),color = "blue", shape = 19, size = 3) +
  geom_point(data = df_holdout, aes(x = x, y = y),color = "red", shape = 17, size = 3) +
  labs(x = "Input (x)", y = "Output (y)", title = "Scatter Plot (Input vs. Output)") +
  theme_minimal()
```



SOLUTION

3c)

You will now fit the 9 polynomials using the training split. You **must NOT** use the complete original data set. You must fit the models using the randomly generated training split.

You must use the `lm()` function, type in the correct formula, and assign the correct data set to the `data` argument of the `lm()` function. Again, this is tedious for now, but it is a learning experience!

The comments and variable names state which model should be specified in each code chunk.

```r
### linear relationship (degree 1)
fit_train_1 <- lm(y ~ x, data =  df_train)
```

```r
### quadratic relationship (degree 2)
fit_train_2 <- lm(y ~ x + I(x^2), data =  df_train)
```

```r
### cubic relationship (degree 3)
fit_train_3 <- lm(y ~ x + I(x^2) + I(x^3), data =  df_train)
```

```r
### degree 4
fit_train_4 <- lm(y ~ x + I(x^2) + I(x^3) + I(x^4), data =  df_train)
```

```r
### degree 5
```

```r
fit_train_5 <- lm(y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5), data =  df_train)
```

```r
### degree 6
fit_train_6 <- lm(y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5) + I(x^6), data =  df_train)
```

```r
### degree 7
fit_train_7 <- lm(y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5) + I(x^6) + I(x^7), data =  df_train)
```

```r
### degree 8
fit_train_8 <- lm(y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5) + I(x^6) + I(x^7) + I(x^8), data =  df_trai
```

```r
### degree 9
fit_train_9 <- lm(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 =
```

SOLUTION

3d)

You fit 9 models on the training split, it is now time to assess their performance on the dedicated **hold-out** test split! You are therefore evaluating the models on data **not** used for estimating their coefficients!

Calculate the hold-out set RMSE for each model. It is up to you as to how to organize the RMSE values and display the results. Bare minimum, you must display the RMSE for each model. You are only allowed to use functions from lecture.

```r
rmse_vector2 <- numeric(9)
rmse_vector2[1] <- modelr::rmse(fit_train_1, df_holdout)
rmse_vector2[2] <- modelr::rmse(fit_train_2, df_holdout)
rmse_vector2[3] <- modelr::rmse(fit_train_3, df_holdout)
rmse_vector2[4] <- modelr::rmse(fit_train_4, df_holdout)
rmse_vector2[5] <- modelr::rmse(fit_train_5, df_holdout)
rmse_vector2[6] <- modelr::rmse(fit_train_6, df_holdout)
rmse_vector2[7] <- modelr::rmse(fit_train_7, df_holdout)
rmse_vector2[8] <- modelr::rmse(fit_train_8, df_holdout)
rmse_vector2[9] <- modelr::rmse(fit_train_9, df_holdout)
```

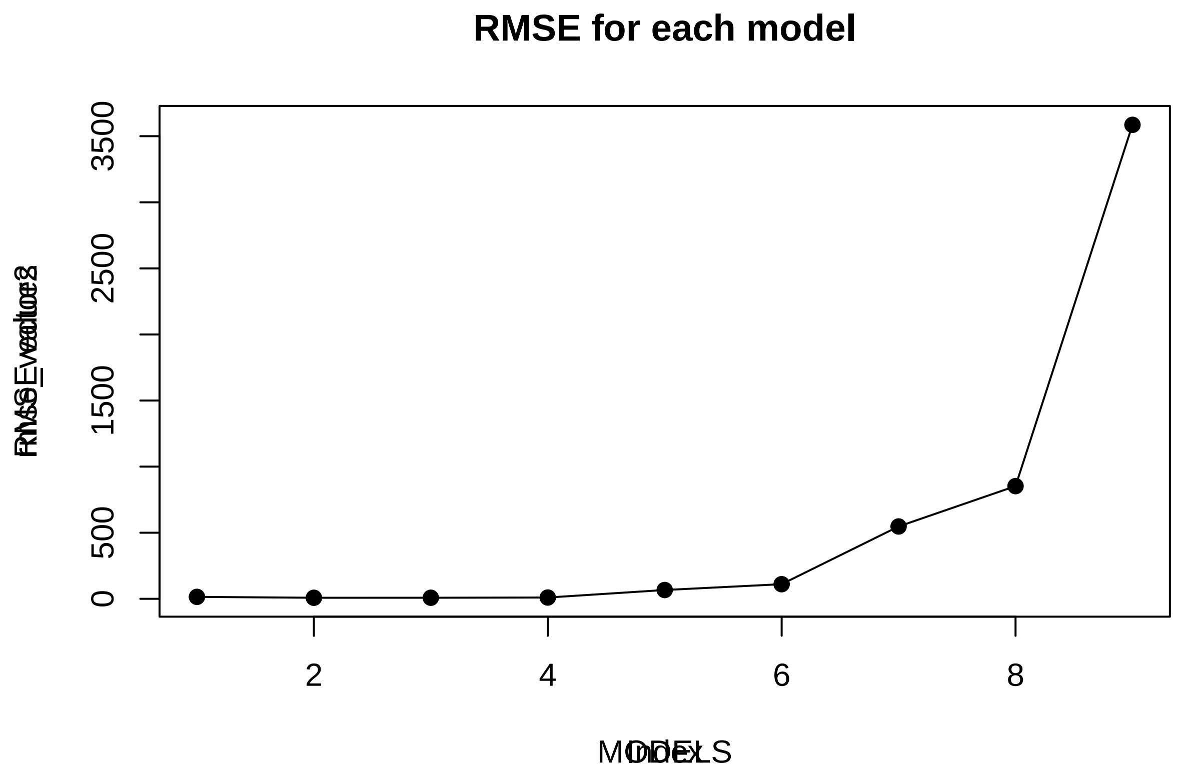
SOLUTION Here we can see the RMSE values as a vector.

```r
print(rmse_vector2)
```

```
## [1]   14.649529    7.877605    7.994202    9.623962   66.528608  110.592780
## [7]  547.302900  852.592341 3585.752884
```

Here we can see the RMSE values in a plot.

```r
plot(rmse_vector2,type = "o",pch = 19)
title(main = "RMSE for each model", xlab = "MODELS", ylab = "RMSE values")
```



3e)

Which model is the best according to the hold-out evaluation? Is it the same model as identified from the training set performance? How does the hold-out set rank the model identified as the best according to the training set?

SOLUTION The best model is identified as the 2nd degree polynomial, because its rmse value is the lowest. The worst model is 9th degree polynomial which was the best model in the original training set according to rmse values.

3f)

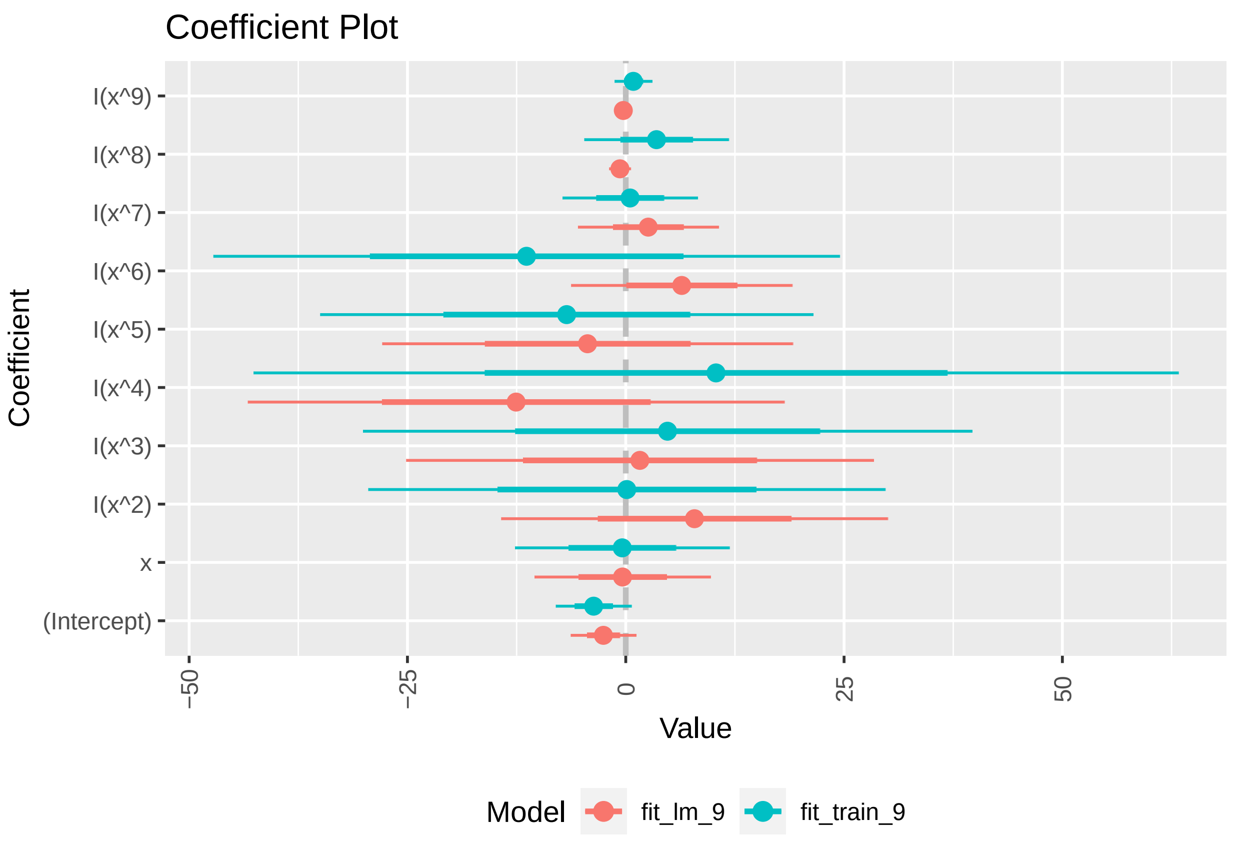
The models named `fit_lm_*` were trained on the entire data set, while the models named `fit_train_*` were trained on the random training split. Although the training sizes are different, let's try to get some idea on the sensitivity of the coefficients to the data.

You must use the `coefplot::multiplot()` function to compare the coefficients based on the original data and based on the training split data. You will compare the same model, but the data used for estimating the coefficients are different.

Compare the coefficient summaries using `coefplot::multiplot()` for the model that was considered to be the best based on the training set.

How do the coefficients compare between the two slightly different training sets?

```r
coefplot::multiplot(fit_lm_9,fit_train_9)
```



SOLUTION

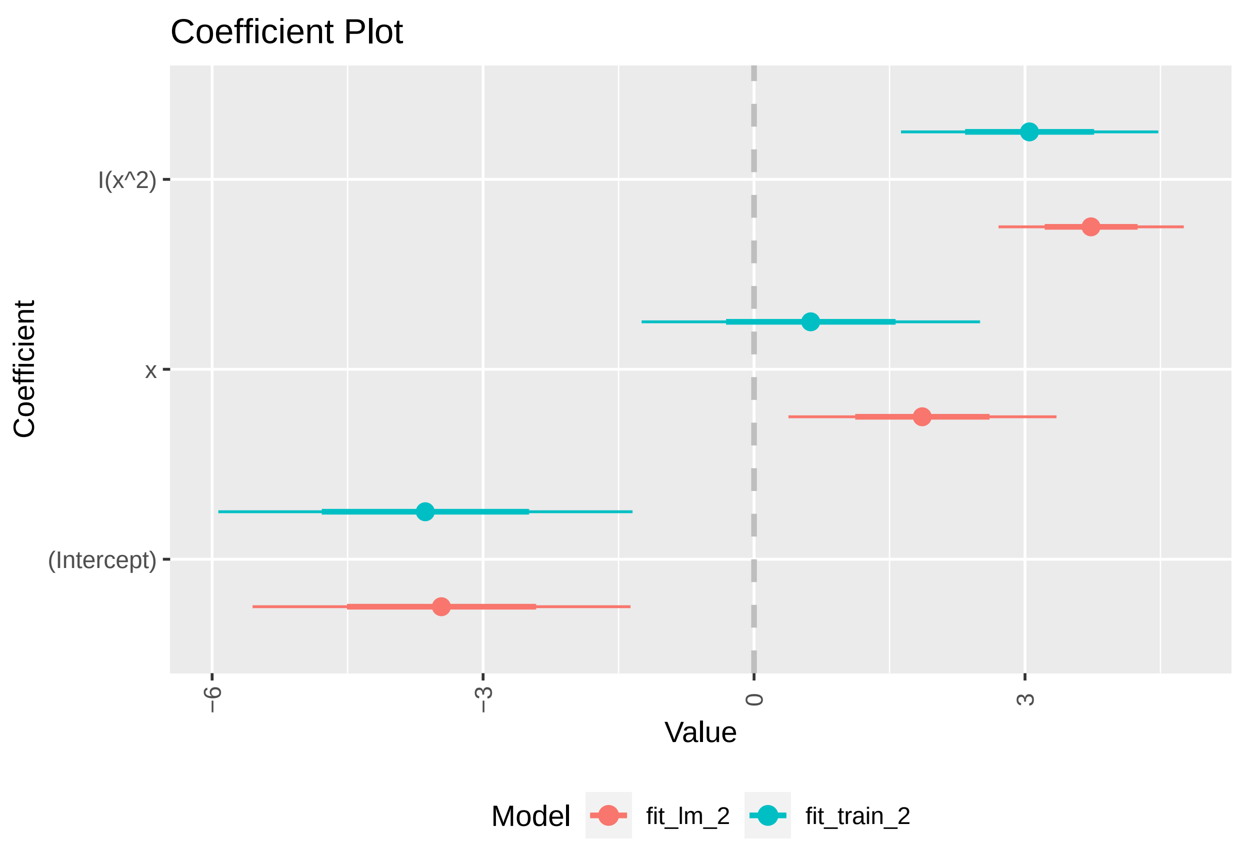
9th model was the best one according to the rmse values in the original data set. When we compare the coefficients, we can observe that the confidence intervals in the model "fit_train_9" are larger. Also, the coefficients in the model "fit_train_9" are bigger in absolute value, which shows it has "uncertain coefficients".

3g)

Compare the coefficient summaries using `coefplot::multiplot()` for the model that was considered to be the best based on the train/test split evaluation.

How do the coefficients compare between the two slightly different training sets?

```
coefplot::multiplot(fit_lm_2,fit_train_2)
```



SOLUTION

2nd model is the best one according to the rmse values on the random training split. When we compare the coefficients, we can observe that the confidence intervals and values of the coefficients in the two models "fit_lm_2" and "fit_train_2" are close to each other. Also coefficients are not large, which indicates that they are not in the class of "uncertain coefficients".

Problem 04

You have fit the 9 polynomials and assessed their performance multiple ways. It's time to now use **resampling** to understand the reliability of your performance assessments. You will specifically use the `caret` package to manage all aspects of data splitting, training, and evaluating the models. If you do not have the `caret` package downloaded and installed please do so before proceeding. You only need to download and install `caret` once.

4a)

Load the `caret` library into the environment with the `library()` function.

```r
library(caret)
```

SOLUTION

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

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

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

4b)

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 `method` argument must equal `'cv'` and the number of folds is controlled by the `number` argument. We could instruct `caret` to use repeated cross-validation by specifying `method` to be `'repeatedcv'` and including the number of repeats via the `repeates` argument. However, we will follow the process consistent with lecture and use 5-fold cross-validation for this assignment.

Specify the resampling scheme by completing the code chunk below. Assign the result of the `trainControl()` function to the `my_ctrl` variable.

How many times will a model be trained and tested using the desired resampling scheme?

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

SOLUTION How many times will an individual model be trained and tested?

An individual model will be trained nrow(df_low_high)/5 = 8 times.

4c)

The `caret` package requires that we specify a primary performance metric of interest, even though it will calculate several performance metrics for us. Please remember that 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!

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

Why did you make your choice?

It is common to consider Root Mean Squared Error (RMSE) as a performance metric. It is in the same units as the response.

4d)

You will now go through training and evaluating the 9 polynomial 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 just as you did before. The difference is that you are not calling the `lm()` function directly. Instead, the `caret` package will manage the fitting process for you.

Please note that you will use the entire data set for training and evaluating with resampling. The `caret` package will manage the data splitting for you.

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 4c). You must specify the `trControl` argument to be `my_ctrl` that you specified in Problem 4b). Don't forget to set the `data` argument to be `df_low_high`!

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. The `caret` object is displayed to the screen for you within each code chunk. This will make it obvious when each model completes the resampling process.

```r
### linear relationship (degree 1)
set.seed(2001)
mod_lowhigh_1 <- train(y ~ x,
                       data = df_low_high,
                       method = "lm",
                       metric = my_metric,
                       preProcess = c("center", "scale"),
                       trControl = my_ctrl)

mod_lowhigh_1
```

SOLUTION

```
## Linear Regression
##
## 40 samples
##  1 predictor
##
## Pre-processing: centered (1), scaled (1)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 32, 32, 32, 32, 32
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   7.619589  0.1623255  5.77079
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```r
### quadratic relationship (degree 2)
set.seed(2001)
mod_lowhigh_2 <- train(y ~ x + I(x^2),
                       data = df_low_high,
                       method = "lm",
                       metric = my_metric,
                       preProcess = c("center", "scale"),
                       trControl = my_ctrl)

mod_lowhigh_2
```

```
## Linear Regression
##
## 40 samples
##  1 predictor
##
## Pre-processing: centered (2), scaled (2)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 32, 32, 32, 32, 32
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   6.116789  0.4668578  4.847323
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```r
### cubic relationship (degree 3)
set.seed(2001)
mod_lowhigh_3 <- train(y ~ x + I(x^2) + I(x^3),
                       data = df_low_high,
                       method = "lm",
                       metric = my_metric,
                       preProcess = c("center", "scale"),
                       trControl = my_ctrl)

mod_lowhigh_3
```

```
## Linear Regression
##
## 40 samples
##  1 predictor
##
## Pre-processing: centered (3), scaled (3)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 32, 32, 32, 32, 32
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   5.068396  0.4698768  4.231458
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```r
### degree 4
set.seed(2001)
mod_lowhigh_4 <- train(y ~ x + I(x^2) + I(x^3) + I(x^4),
                       data = df_low_high,
                       method = "lm",
                       metric = my_metric,
                       preProcess = c("center", "scale"),
                       trControl = my_ctrl)

mod_lowhigh_4
```

```
## Linear Regression
##
## 40 samples
##  1 predictor
##
## Pre-processing: centered (4), scaled (4)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 32, 32, 32, 32, 32
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   7.938712  0.4259179  5.533135
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```r
### degree 5
set.seed(2001)
mod_lowhigh_5 <- train(y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5),
                       data = df_low_high,
                       method = "lm",
                       metric = my_metric,
                       preProcess = c("center", "scale"),
                       trControl = my_ctrl)

mod_lowhigh_5
```

```
## Linear Regression
##
## 40 samples
##  1 predictor
##
## Pre-processing: centered (5), scaled (5)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 32, 32, 32, 32, 32
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   16.89356  0.3772241  8.776187
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```r
### degree 6
set.seed(2001)
mod_lowhigh_6 <- train(y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5) + I(x^6),
                       data = df_low_high,
                       method = "lm",
                       metric = my_metric,
                       preProcess = c("center", "scale"),
                       trControl = my_ctrl)

mod_lowhigh_6
```

```
## Linear Regression
##
## 40 samples
##  1 predictor
##
## Pre-processing: centered (6), scaled (6)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 32, 32, 32, 32, 32
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   23.08368  0.4712291  10.75952
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```r
### degree 7
set.seed(2001)
mod_lowhigh_7 <- train(y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5) + I(x^6) + I(x^7),
                       data = df_low_high,
                       method = "lm",
                       metric = my_metric,
                       preProcess = c("center", "scale"),
                       trControl = my_ctrl)

mod_lowhigh_7
```

```
## Linear Regression
##
## 40 samples
##  1 predictor
##
## Pre-processing: centered (7), scaled (7)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 32, 32, 32, 32, 32
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   91.71999  0.4049495  35.4399
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```
### degree 8
set.seed(2001)
mod_lowhigh_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 = df_low_high,
                       method = "lm",
                       metric = my_metric,
                       preProcess = c("center", "scale"),
                       trControl = my_ctrl)

mod_lowhigh_8
```

```
## Linear Regression
##
## 40 samples
##  1 predictor
##
## Pre-processing: centered (8), scaled (8)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 32, 32, 32, 32, 32
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   353.6662  0.4381417  128.0087
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```
### degree 9
set.seed(2001)
mod_lowhigh_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 = df_low_high,
                       method = "lm",
                       metric = my_metric,
                       preProcess = c("center", "scale"),
                       trControl = my_ctrl)
```

```
mod_lowhigh_9
```

```
## Linear Regression
##
## 40 samples
##  1 predictor
##
## Pre-processing: centered (9), scaled (9)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 32, 32, 32, 32, 32
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   813.8003  0.4214028  291.0134
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

4e)

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

```
lowhigh_results = resamples(list(fit_01 = mod_lowhigh_1,
                                 fit_02 = mod_lowhigh_2,
                                 fit_03 = mod_lowhigh_3,
                                 fit_04 = mod_lowhigh_4,
                                 fit_05 = mod_lowhigh_5,
                                 fit_06 = mod_lowhigh_6,
                                 fit_07 = mod_lowhigh_7,
                                 fit_08 = mod_lowhigh_8,
                                 fit_09 = mod_lowhigh_9))
```

The `caret` package provides default summary and plot methods which rank the models based on their resampling hold-out results. The `summary()` function prints a table like object which summarizes the resampling results. The `dotplot()` function creates a dot plot with confidence intervals on the resampling performance metrics.

You must display the summary table using the `summary()` function for the resampling results and visualize the resampling results using the `dotplot()` function. In both the `summary()` and `dotplot()` functions, you must specify the `metric` argument to be primary performance metric that you specified previously.

Which model is the best according to the resampling results?

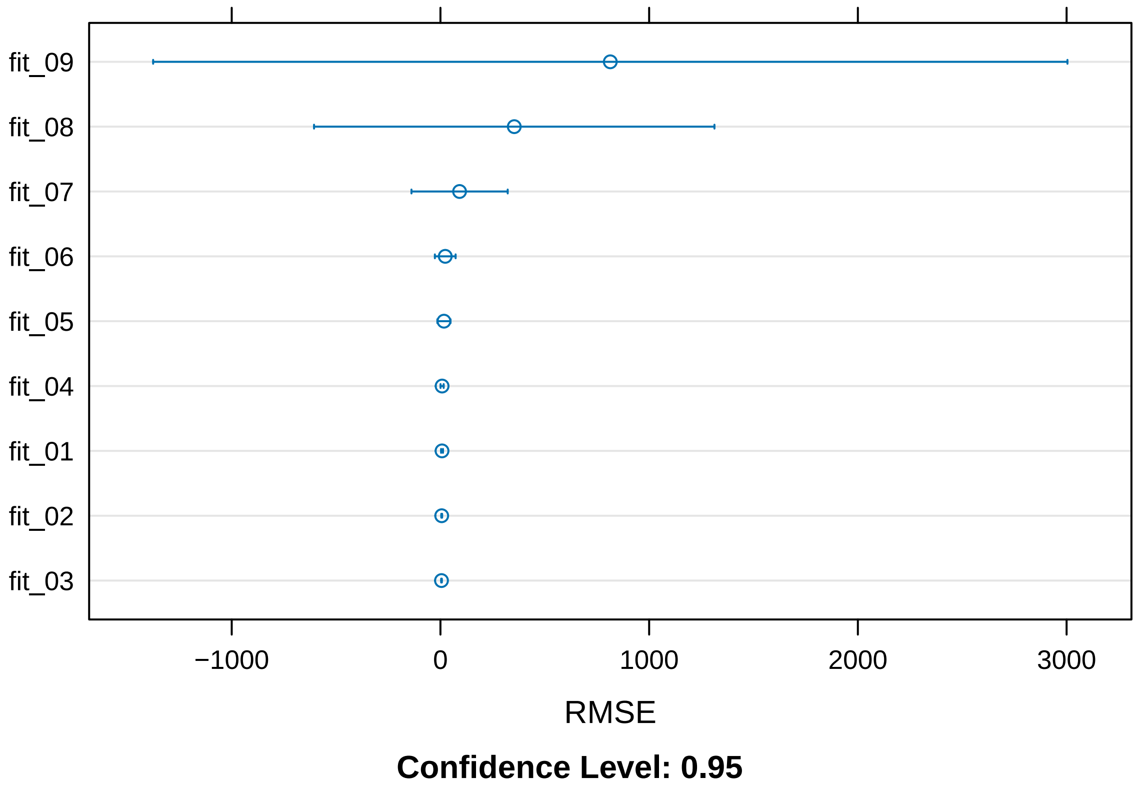
```
summary(lowhigh_results, metric = my_metric)
```

SOLUTION

```
##
## Call:
## summary.resamples(object = lowhigh_results, metric = my_metric)
##
## Models: fit_01, fit_02, fit_03, fit_04, fit_05, fit_06, fit_07, fit_08, fit_09
## Number of resamples: 5
##
```

```
## RMSE
##              Min.  1st Qu.   Median       Mean   3rd Qu.        Max. NA's
## fit_01 5.617485 5.930271 6.009046   7.619589  6.399527   14.141618    0
## fit_02 3.775855 5.580535 6.308326   6.116789  7.075319    7.843909    0
## fit_03 3.267682 4.436099 5.026415   5.068396  6.298289    6.313496    0
## fit_04 3.721641 4.658383 6.298265   7.938712  6.372285   18.642988    0
## fit_05 4.351063 5.143578 6.343898  16.893558  6.412467   62.216785    0
## fit_06 3.612705 4.241104 6.235668  23.083680  6.699303   94.629620    0
## fit_07 4.395510 6.764401 7.166665  91.719986 16.555141  423.718216    0
## fit_08 4.099486 7.222621 7.284775 353.666192 14.100723 1735.623356    0
## fit_09 4.740879 7.222207 7.404804 813.800310 80.792443 3968.841215    0
```

```r
dotplot(lowhigh_results, metric = my_metric)
```



Which model is the best?

Based on the Mean RMSE values, 3rd model (the one with 3rd degree polynomial) has the lowest RMSE of 5.068396, making it the best-performing model.

4f)

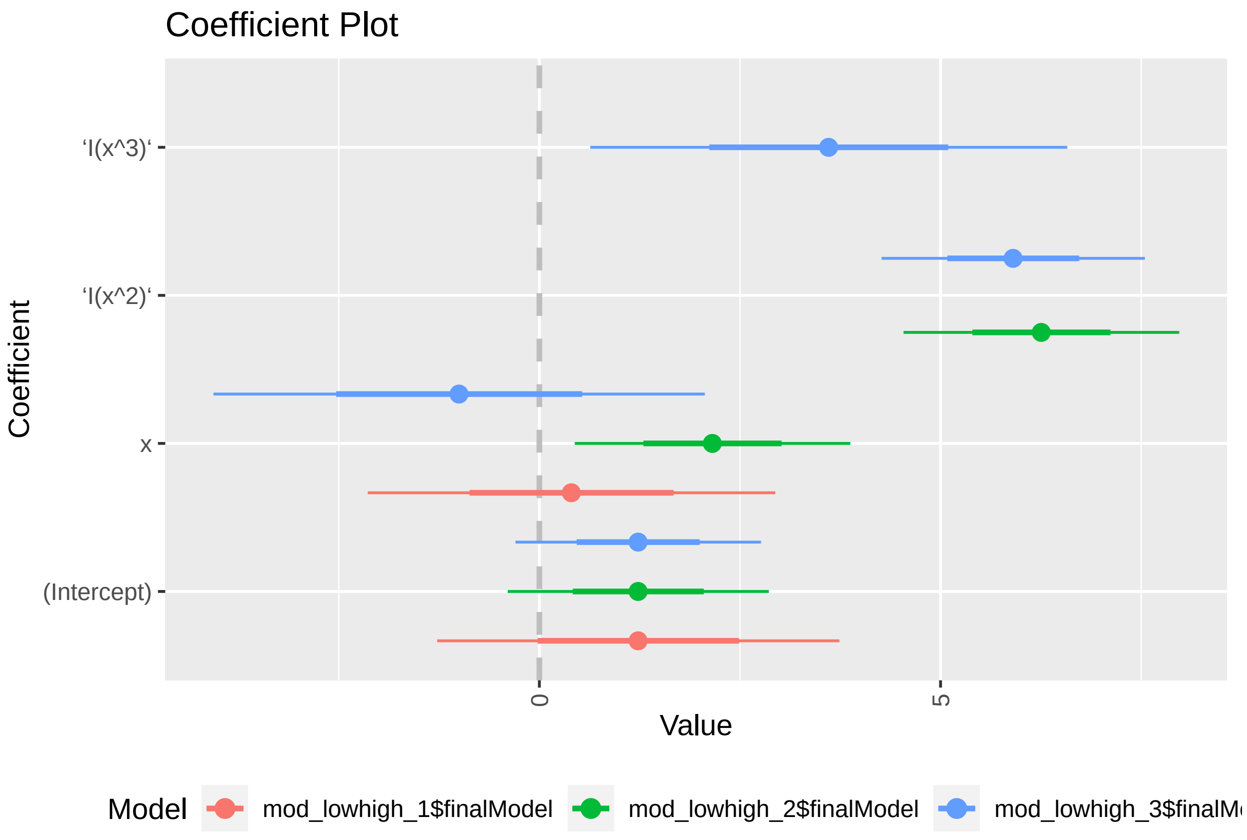
The `mod_lowhigh_*` variables are `caret` model objects and are essentially lists. Each object, `mod_lowhigh_1` through `mod_lowhigh_9` consists of numerous fields which can be accessed via the dollar sign operator, `$`. One such field, `$finalModel`, provides access to the underlying `lm()` model object associated with the **final fit**. Thus, even though `mod_lowhigh_1` is a `caret` object the field `mod_lowhigh_1$finalModel` is an `lm()` object just like the `fit_lm_1` object you fit at the beginning of the assignment. We can apply functions to help interpret the model behavior just as we did previously.

You must compare the coefficients between the top 3 models identified by the resampling procedure using the `coefplot::multiplot()` function.

Are the coefficient estimates similar for those coefficients present in at least 2 of the models visualized in the previous figure?

Is the level or amount of uncertainty associated with the coefficients consistent across the three models?

```r
coefplot::multiplot(mod_lowhigh_1$finalModel,mod_lowhigh_2$finalModel,mod_lowhigh_3$finalModel)
```



SOLUTION

The top performing 3 models are 1st, 2nd and 3rd models. When we see the figure above, we can observe that,

1) the coefficients of xˆ2 in the 2nd and 3rd model are close to each other. Also, their confidence intervals are similar,
2) the constants of all models are close to each other and the confidence intervals are similar,
3) the coefficients of x in the 1st and 2nd model have the same sign, but the values are not very close among the 3 models.

Problem 05

You have completed an entire regression problem! You correctly used resampling to train and compare simple to complex models in order to identify the most appropriate model that generalizes to new data!

However, let's try and gain additional insight by repeating the resampling procedure on another data set. The data used in the previous problems were generated using a "high" level of noise. You will now train and compare the same 9 polynomial models, but with data coming from a "low" level of noise. The underlying **data generating process** is the same. 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 levels.

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 before.

```
path_low_low <- 'hw02_lowsize_lownoise.csv'

df_low_low <- readr::read_csv(path_low_low, col_names = TRUE)
```

```
## Rows: 40 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.
```

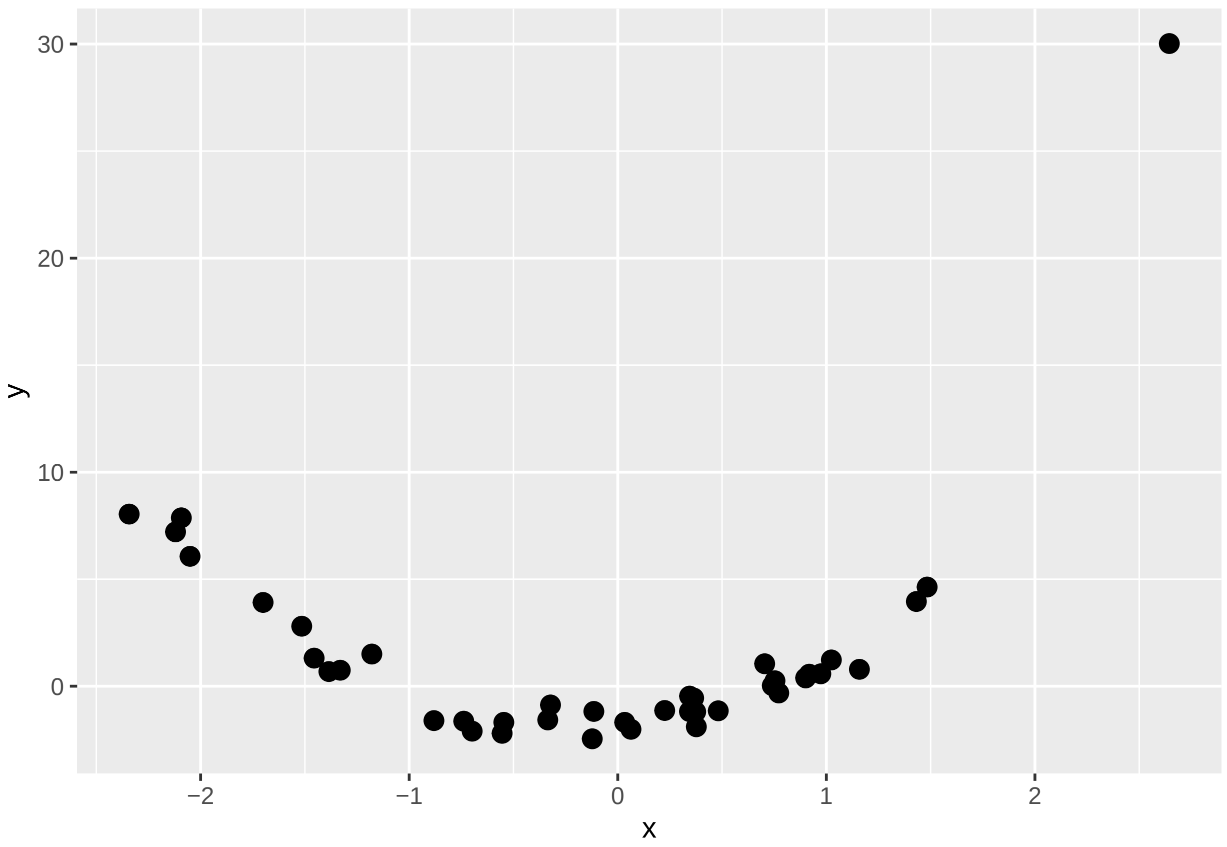
```
df_low_low %>% glimpse()
```

```
## Rows: 40
## Columns: 2
## $ x <dbl> -0.11464944, 0.22502931, 0.37356406, -2.34241739, 1.15847408, 0.3443~
## $ y <dbl> -1.1796816, -1.1355929, -1.1975801, 8.0392787, 0.7875834, -0.4632007~
```

5a)

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)?

```
ggplot(data = df_low_low) +
  geom_point(aes(x = x, y = y), size = 3)
```



SOLUTION

How does it compare?

The values are looking more close to each other compared to the previous data set. It looks like they a "parabola".

5b)

You will not repeat all of the previous exercises. Instead, you will jump straight to training and evaluating models with resampling. You will use the same resampling scheme and primary performance metric that you used in Problem 04 to train the 9 polynomial models. This time however you will use the low noise level data.

Train the 9 polynomial models using the required formula interface, `method`, `trControl`, and `metric` arguments that you used in Problem 04. However, pay close attention and set the `data` argument to be `df_low_low`.

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.

```
### linear relationship (degree 1)
set.seed(2001)
mod_lowlow_1 <- train(y ~ x,
                      data = df_low_low,
                      method = "lm",
                      metric = my_metric,
                      preProcess = c("center", "scale"),
                      trControl = my_ctrl)
```

```r
mod_lowlow_1
```

SOLUTION

```
## Linear Regression
##
## 40 samples
##  1 predictor
##
## Pre-processing: centered (1), scaled (1)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 32, 32, 32, 32, 32
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   5.144982  0.2415764  3.543544
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```r
### quadratic relationship (degree 2)
set.seed(2001)
mod_lowlow_2 <- train(y ~ x + I(x^2),
                       data = df_low_low,
                       method = "lm",
                       metric = my_metric,
                       preProcess = c("center", "scale"),
                       trControl = my_ctrl)

mod_lowlow_2
```

```
## Linear Regression
##
## 40 samples
##  1 predictor
##
## Pre-processing: centered (2), scaled (2)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 32, 32, 32, 32, 32
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   2.163325  0.8636634  1.366768
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```r
### cubic relationship (degree 3)
set.seed(2001)
mod_lowlow_3 <- train(y ~ x + I(x^2) + I(x^3),
                       data = df_low_low,
                       method = "lm",
                       metric = my_metric,
                       preProcess = c("center", "scale"),
                       trControl = my_ctrl)

mod_lowlow_3
```

```
## Linear Regression
##
## 40 samples
##  1 predictor
##
## Pre-processing: centered (3), scaled (3)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 32, 32, 32, 32, 32
## Resampling results:
##
##   RMSE      Rsquared  MAE
##   1.890006  0.90314   1.128333
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```
### degree 4
set.seed(2001)
mod_lowlow_4 <- train(y ~ x + I(x^2) + I(x^3) + I(x^4),
                      data = df_low_low,
                      method = "lm",
                      metric = my_metric,
                      preProcess = c("center", "scale"),
                      trControl = my_ctrl)

mod_lowlow_4
```

```
## Linear Regression
##
## 40 samples
##  1 predictor
##
## Pre-processing: centered (4), scaled (4)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 32, 32, 32, 32, 32
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   1.785649  0.9122532  0.9938186
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```
### degree 5
set.seed(2001)
mod_lowlow_5 <- train(y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5),
                      data = df_low_low,
                      method = "lm",
                      metric = my_metric,
                      preProcess = c("center", "scale"),
                      trControl = my_ctrl)

mod_lowlow_5
```

```
## Linear Regression
##
## 40 samples
##  1 predictor
```

```
##
## Pre-processing: centered (5), scaled (5)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 32, 32, 32, 32, 32
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   2.727245  0.9310017  1.327683
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```
### degree 6
set.seed(2001)
mod_lowlow_6 <- train(y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5) + I(x^6),
                      data = df_low_low,
                      method = "lm",
                      metric = my_metric,
                      preProcess = c("center", "scale"),
                      trControl = my_ctrl)

mod_lowlow_6
```

```
## Linear Regression
##
## 40 samples
##  1 predictor
##
## Pre-processing: centered (6), scaled (6)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 32, 32, 32, 32, 32
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   1.054518  0.9286889  0.6944083
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```
### degree 7
set.seed(2001)
mod_lowlow_7 <- train(y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5) + I(x^6) + I(x^7),
                      data = df_low_low,
                      method = "lm",
                      metric = my_metric,
                      preProcess = c("center", "scale"),
                      trControl = my_ctrl)

mod_lowlow_7
```

```
## Linear Regression
##
## 40 samples
##  1 predictor
##
## Pre-processing: centered (7), scaled (7)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 32, 32, 32, 32, 32
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   18.33918  0.9102542  6.864169
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```r
### degree 8
set.seed(2001)
mod_lowlow_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 = df_low_low,
                      method = "lm",
                      metric = my_metric,
                      preProcess = c("center", "scale"),
                      trControl = my_ctrl)

mod_lowlow_8
```

```
## Linear Regression
##
## 40 samples
##  1 predictor
##
## Pre-processing: centered (8), scaled (8)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 32, 32, 32, 32, 32
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   47.47409  0.9119027  17.16845
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```r
### degree 9
set.seed(2001)
mod_lowlow_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 = df_low_low,
                      method = "lm",
                      metric = my_metric,
                      preProcess = c("center", "scale"),
                      trControl = my_ctrl)

mod_lowlow_9
```

```
## Linear Regression
##
## 40 samples
##  1 predictor
##
## Pre-processing: centered (9), scaled (9)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 32, 32, 32, 32, 32
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   71.31048  0.8973848  25.59357
```

```
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

5c)

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

```
lowlow_results = resamples(list(fit_01 = mod_lowlow_1,
                                fit_02 = mod_lowlow_2,
                                fit_03 = mod_lowlow_3,
                                fit_04 = mod_lowlow_4,
                                fit_05 = mod_lowlow_5,
                                fit_06 = mod_lowlow_6,
                                fit_07 = mod_lowlow_7,
                                fit_08 = mod_lowlow_8,
                                fit_09 = mod_lowlow_9))
```

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

You must display the summary table using the `summary()` function for the resampling results and visualize the resampling results using the `dotplot()` function. In both the `summary()` and `dotplot()` functions, you must specify the `metric` argument to be primary performance metric that you specified previously.

Which model is the best according to the resampling results?

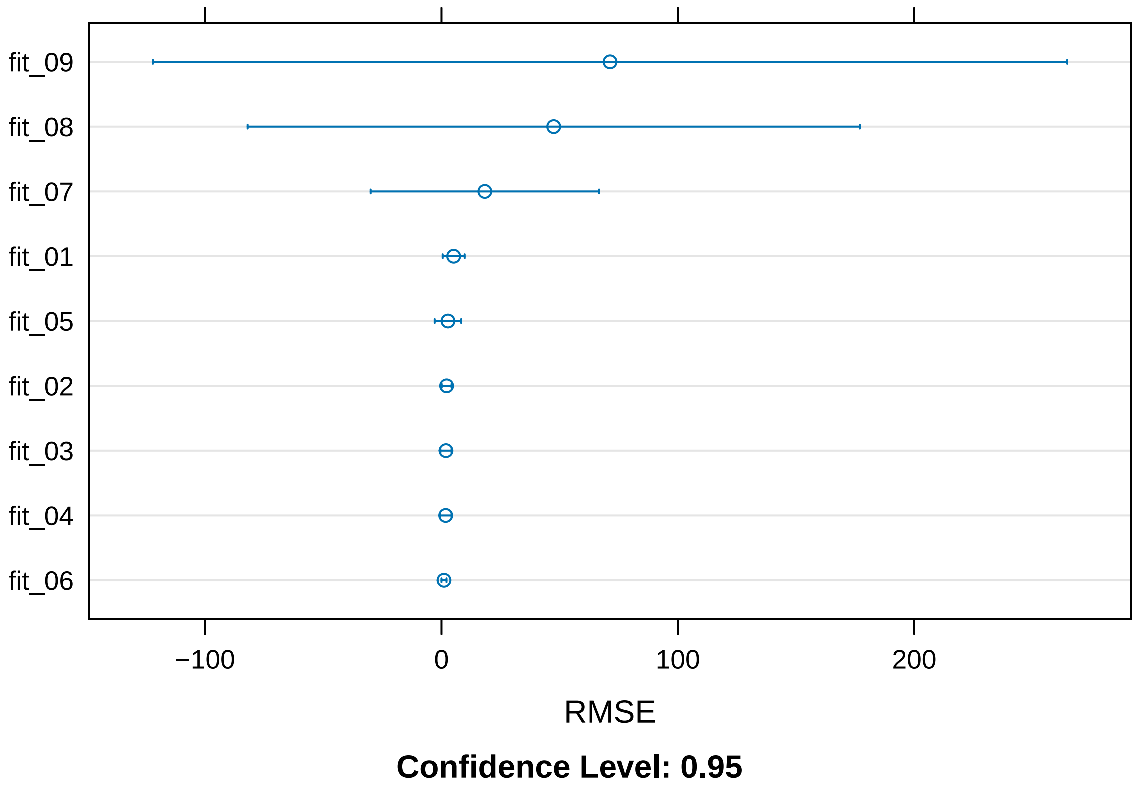
```

summary(lowlow_results, metric = my_metric)
```

SOLUTION

```
##
## Call:
## summary.resamples(object = lowlow_results, metric = my_metric)
##
## Models: fit_01, fit_02, fit_03, fit_04, fit_05, fit_06, fit_07, fit_08, fit_09
## Number of resamples: 5
##
## RMSE
##             Min.   1st Qu.    Median      Mean   3rd Qu.       Max. NA's
## fit_01 2.4338576 3.6894089 3.7831721  5.144982 4.0641653  11.754307    0
## fit_02 0.7867695 1.2841627 1.7803386  2.163325 1.8969124   5.068442    0
## fit_03 0.8176783 1.0461808 1.1316759  1.890006 1.1782465   5.276249    0
## fit_04 0.7013583 0.8282032 0.9413818  1.785649 1.1272395   5.330062    0
## fit_05 0.5905692 0.5969589 0.7099658  2.727245 0.9455988  10.793134    0
## fit_06 0.5181681 0.6009772 0.6456859  1.054518 0.9478657   2.559893    0
## fit_07 0.5442659 0.6108852 0.9748669 18.339176 1.6140096  87.951850    0
## fit_08 0.5316738 0.6041379 0.9768202 47.474093 1.2175328 234.040298    0
## fit_09 0.5388171 0.5996942 0.9616194 71.310477 4.5202109 349.932044    0
```

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



Which model is the best?

Based on the Mean RMSE values, 6th model (the one with 6th degree polynomial) has the lowest RMSE of 1.054518, making it the best-performing model.

5d)

The `df_low_high` and `df_low_low` data have the same sample size, but the data were generated under different uncontrollable noise levels. This exercise is analogous to recording measurements with worn out low fidelity sensors (`df_low_high`) compared to new modern high resolution sensors (`df_low_low`). The low level of noise gives a "better" glimpse of the real underlying **data generating process**.

Did the resampling scheme identify the same model as the best model for the two noise cases? Did the models identified to be poor for the high noise case remain poor for the low noise case?

SOLUTION What do you think?

1) In the "high noise" case, the best model was identified as the 3rd model. However, in the "low noise" case, the best performing model is the 6th model.
2) 7th, 8th, and 9th models are identified to be poor in both "high noise" and "low noise" cases.

Problem 06

You will repeat the resampling exercise with one more data set. This version of the data were generated under the high noise level, as with the first data set you worked with. However, this version of the data has more observations. Thus, you are still working with noisy data, but you now have a higher sample size to work with. This problem therefore focuses on the influence of sample size on the modeling effort.

The high sample size high noise level data are loaded in the code chunk below. The glimpse shows that the variable names are the same as those in the previous two cases.

```r
path_high_high <- 'hw02_highsize_highnoise.csv'

df_high_high <- readr::read_csv(path_high_high, col_names = TRUE)
```

```
## Rows: 200 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.
```

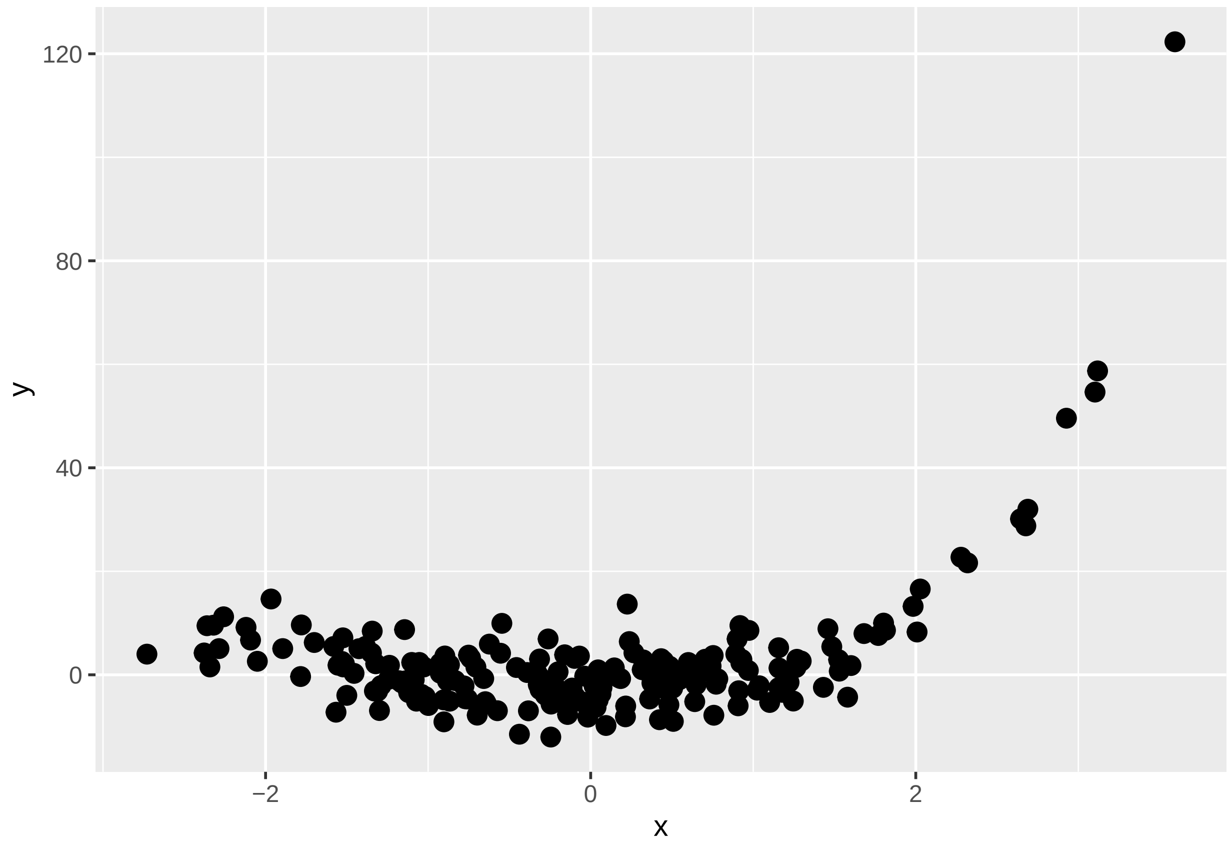
```r
df_high_high %>% glimpse()
```

```
## Rows: 200
## Columns: 2
## $ x <dbl> -0.11464944, 0.22502931, 0.37356406, -2.34241739, 1.15847408, 0.3443~
## $ y <dbl> -2.56496089, 13.65865394, 0.09488574, 1.51679017, 1.30866543, 1.4072~
```

6a)

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)?

```r
ggplot(data = df_high_high) +
  geom_point(aes(x = x, y = y), size = 3)
```



SOLUTION

How does it compare?

The scaling in y-axis is different compared to the scatter plot in Problem 5a). The output values in this case are not close to each other, however we have a larger sample size in this case.

6b)

You must execute resampling on the 9 polynomial models using the high sample size and high noise level data.

Train the 9 polynomial models using the required formula interface, `method`, `trControl`, and `metric` arguments that you used in Problem 04. However, pay close attention and set the `data` argument to be `df_high_high`.

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.

```r
### linear relationship (degree 1)
set.seed(2001)
mod_highhigh_1 <- train(y ~ x,
                        data = df_high_high,
                        method = "lm",
                        metric = my_metric,
                        preProcess = c("center", "scale"),
                        trControl = my_ctrl)

mod_highhigh_1
```

SOLUTION

```
## Linear Regression
##
## 200 samples
##   1 predictor
##
## Pre-processing: centered (1), scaled (1)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 160, 160, 160, 160, 160
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   10.81521  0.1491076  7.121727
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```r
### quadratic relationship (degree 2)
set.seed(2001)
mod_highhigh_2 <- train(y ~ x + I(x^2),
                        data = df_high_high,
                        method = "lm",
                        metric = my_metric,
                        preProcess = c("center", "scale"),
                        trControl = my_ctrl)
```

```r
mod_highhigh_2
```

```
## Linear Regression
##
## 200 samples
##   1 predictor
##
## Pre-processing: centered (2), scaled (2)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 160, 160, 160, 160, 160
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   7.400325  0.6209405  4.934927
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```r
### cubic relationship (degree 3)
set.seed(2001)
mod_highhigh_3 <- train(y ~ x + I(x^2) + I(x^3),
                        data = df_high_high,
                        method = "lm",
                        metric = my_metric,
                        preProcess = c("center", "scale"),
                        trControl = my_ctrl)

mod_highhigh_3
```

```
## Linear Regression
##
## 200 samples
##   1 predictor
##
## Pre-processing: centered (3), scaled (3)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 160, 160, 160, 160, 160
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   5.811265  0.6668802  4.134066
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```r
### degree 4
set.seed(2001)
mod_highhigh_4 <- train(y ~ x + I(x^2) + I(x^3) + I(x^4),
                        data = df_high_high,
                        method = "lm",
                        metric = my_metric,
                        preProcess = c("center", "scale"),
                        trControl = my_ctrl)

mod_highhigh_4
```

```
## Linear Regression
```

```
##
## 200 samples
##   1 predictor
##
## Pre-processing: centered (4), scaled (4)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 160, 160, 160, 160, 160
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   5.037926  0.6814885  3.769584
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```
### degree 5
set.seed(2001)
mod_highhigh_5 <- train(y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5),
                        data = df_high_high,
                        method = "lm",
                        metric = my_metric,
                        preProcess = c("center", "scale"),
                        trControl = my_ctrl)

mod_highhigh_5
```

```
## Linear Regression
##
## 200 samples
##   1 predictor
##
## Pre-processing: centered (5), scaled (5)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 160, 160, 160, 160, 160
## Resampling results:
##
##   RMSE     Rsquared   MAE
##   4.33134  0.7031167  3.473011
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```
### degree 6
set.seed(2001)
mod_highhigh_6 <- train(y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5) + I(x^6),
                        data = df_high_high,
                        method = "lm",
                        metric = my_metric,
                        preProcess = c("center", "scale"),
                        trControl = my_ctrl)

mod_highhigh_6
```

```
## Linear Regression
##
## 200 samples
##   1 predictor
```

```
##
## Pre-processing: centered (6), scaled (6)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 160, 160, 160, 160, 160
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   5.028911  0.6920768  3.637877
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```r
### degree 7
set.seed(2001)
mod_highhigh_7 <- train(y ~ x + I(x^2) + I(x^3) + I(x^4) + I(x^5) + I(x^6) + I(x^7),
                       data = df_high_high,
                       method = "lm",
                       metric = my_metric,
                       preProcess = c("center", "scale"),
                       trControl = my_ctrl)

mod_highhigh_7
```

```
## Linear Regression
##
## 200 samples
##   1 predictor
##
## Pre-processing: centered (7), scaled (7)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 160, 160, 160, 160, 160
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   5.198282  0.6710965  3.686402
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```r
### degree 8
set.seed(2001)
mod_highhigh_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 = df_high_high,
                       method = "lm",
                       metric = my_metric,
                       preProcess = c("center", "scale"),
                       trControl = my_ctrl)

mod_highhigh_8
```

```
## Linear Regression
##
## 200 samples
##   1 predictor
##
## Pre-processing: centered (8), scaled (8)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 160, 160, 160, 160, 160
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   5.986668  0.6932605  3.890623
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```r
### degree 9
set.seed(2001)
mod_highhigh_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 = df_high_high,
                       method = "lm",
                       metric = my_metric,
                       preProcess = c("center", "scale"),
                       trControl = my_ctrl)

mod_highhigh_9
```

```
## Linear Regression
##
## 200 samples
##   1 predictor
##
## Pre-processing: centered (9), scaled (9)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 160, 160, 160, 160, 160
## Resampling results:
##
##   RMSE      Rsquared  MAE
##   11.74026  0.611759  4.976636
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

6c)

The code chunk below compiles all of the resampling results together for the models associated with the high sample size and high noise level data.

```r
highhigh_results = resamples(list(fit_01 = mod_highhigh_1,
                                  fit_02 = mod_highhigh_2,
                                  fit_03 = mod_highhigh_3,
                                  fit_04 = mod_highhigh_4,
                                  fit_05 = mod_highhigh_5,
                                  fit_06 = mod_highhigh_6,
                                  fit_07 = mod_highhigh_7,
                                  fit_08 = mod_highhigh_8,
                                  fit_09 = mod_highhigh_9))
```

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

You must display the summary table using the `summary()` function for the resampling results and visualize the resampling results using the `dotplot()` function. In both the `summary()` and `dotplot()` functions, you must specify the `metric` argument to be primary performance metric that you specified previously.

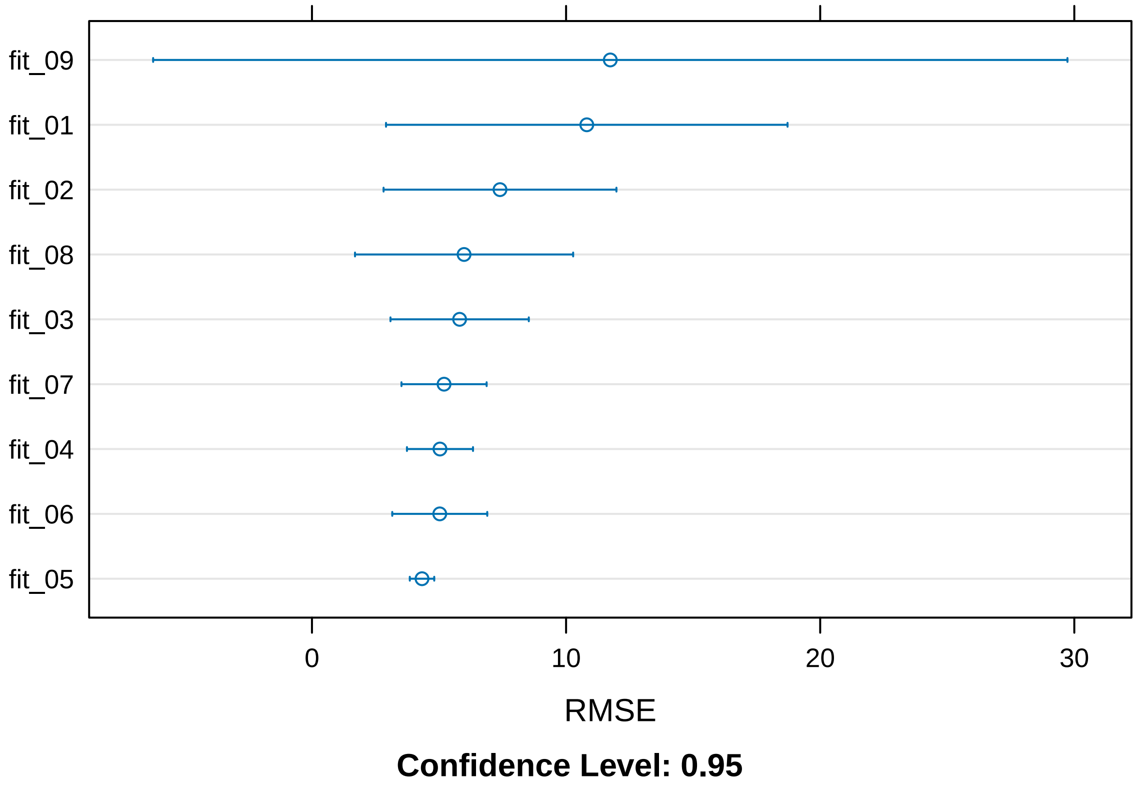
Which model is the best according to the resampling results?

```
summary(highhigh_results, metric = my_metric)
```

SOLUTION

```
##
## Call:
## summary.resamples(object = highhigh_results, metric = my_metric)
##
## Models: fit_01, fit_02, fit_03, fit_04, fit_05, fit_06, fit_07, fit_08, fit_09
## Number of resamples: 5
##
## RMSE
##            Min.  1st Qu.   Median      Mean  3rd Qu.      Max. NA's
## fit_01 7.144399 7.709821 8.132484 10.815210 8.953933 22.135414    0
## fit_02 5.190688 5.522222 5.733782  7.400325 6.622219 13.932716    0
## fit_03 4.417490 4.613093 5.121780  5.811265 5.218235  9.685728    0
## fit_04 4.122581 4.595870 4.710784  5.037926 4.925714  6.834681    0
## fit_05 3.930696 4.003262 4.282074  4.331340 4.604970  4.835701    0
## fit_06 3.930049 4.108618 4.618845  5.028911 4.848635  7.638410    0
## fit_07 3.908117 4.535044 4.800166  5.198282 5.311533  7.436552    0
## fit_08 3.897065 4.456369 4.711269  5.986668 4.727407 12.141231    0
## fit_09 3.893277 4.732182 4.751114 11.740260 7.803348 37.521379    0
```

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



Which model is the best?

Based on the Mean RMSE values, 5th model (the one with 5th degree polynomial) has the lowest RMSE of

4.331340, making it the best-performing model.

6d)

The `df_low_high` and `df_high_high` data have the same noise level, but different sample sizes.

Did the resampling scheme identify the same model as the best model for the two sample size cases? Did the models identified to be poor for the low sample size case remain poor for the high sample size case?

SOLUTION What do you think?

1) In the `df_low_high` case, 3rd model was the best, while in the `df_high_high` case 5th model is the best.
2) In the `df_low_high` case, 7th, 8th and 9th models did not perform well as their rmse values were large. However, in the `df_high_high` case, 7th, 8th and 9th models have small rmse values.

6e)

You have executed a model selection task under varying levels of noise and sample sizes. Your goal was to select the best model from a candidate set of varying levels of complexity.

What do you think the impact is between sample size and noise level on the ability to identify complex behavior?

SOLUTION What do you think?

1) Larger sample sizes improve the ability to detect complex patterns because they provide more data points.
2) High noise levels in data can obscure complex behavior. For example, the `df_low_high` case has poor results in 7th, 8th, and 9th models.
3) A larger sample size can help reduce the impact of noise as we see in 7th, 8th, and 9th models with `df_low_high` and `df_high_high` cases.
4) Balancing sample size and noise is essential for meaningful complex behavior analysis.