Overview

This homework is dedicated to working with non-Bayesian regularization. You will fit your own RIDGE and LASSO penalized regression models in order to gain experience with interpreting *elastic net* model results. You will also practice *tuning* a penalized regression model by finding the *best* regularization strength, λ, value using a train/test split. You will then use the `glmnet` package and `caret` to tune elastic net models in order to identify the most important features in a model by turning off unimportant features. You will see that the programming associated with non-Bayesian models is VERY similar to the programming requried for Bayesian models! You will continue to use fundamentally important matrix operations for making predictions, while gaining a deeper understanding of how regularization prevents extreme coefficient values.

This assignment will provide practice working with realistic modeling situations such as large numbers of inputs, examining the influence of correlated features, working with interactions, and working with categorical inputs.

If you do not have `glmnet` or `caret` installed please download both before starting the assignment.

You will also work with the `corrplot` package in this assignment. You must download and install `corrplot` before starting the assignment.

IMPORTANT: The RMarkdown assumes you have downloaded the data sets (CSV files) to the same directory you saved the template Rmarkdown file. If you do not have the 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

This assignment will use packages from the `tidyverse` suite.

```
library(tidyverse)
```

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

This assignment also uses the `broom` package. The `broom` package is part of `tidyverse` and so you have it installed already. The `broom` package will be used via the `::` operator later in the assignment and so you do not need to load it directly.

The `glmnet` package will be loaded later in the assignment. As stated previously, please download and install `glmnet` if you do not currently have it.

Problem 01

You will gain experience with non-Bayesian Ridge and Lasso regression by working with a large number of inputs in this problem. The training data are loaded for you below.

```
dfA_train <- readr::read_csv('hw09_probA_train.csv', col_names = TRUE)
```

```
## Rows: 96 Columns: 76
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (76): x01, x02, x03, x04, x05, x06, x07, x08, x09, x10, x11, x12, x13, x...
##
## 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.
```

All input variables have names that start with `x` and the output is named `y`. The glimpse provided below shows there are many input columns!

```
dfA_train %>% glimpse()
```

```
## Rows: 96
## Columns: 76
## $ x01 <dbl> -2.06943319, -1.29642012, 0.91302407, 0.53847200, 1.58258670, -2.3~
## $ x02 <dbl> -0.69220535, -0.74646173, 0.50415529, -2.24983365, -0.06869533, -1~
## $ x03 <dbl> -1.01314640, 0.46711663, 0.35073648, -0.26764563, -0.06524989, 0.8~
## $ x04 <dbl> 1.3310201, -1.1901216, 0.4567449, 1.9486926, 1.1343209, -0.5150867~
## $ x05 <dbl> 0.2662435, -0.2801269, 0.5267086, -1.2564140, 1.5565706, -0.237865~
## $ x06 <dbl> -0.72471702, -0.09158292, -0.70865959, 1.07960684, 0.72399642, -0.~
## $ x07 <dbl> 0.119003559, 1.794130638, -0.131307071, -0.634631730, 0.791042823,~
## $ x08 <dbl> -0.21240839, 1.43690253, 0.37810170, -0.95218224, 0.02586663, -0.9~
## $ x09 <dbl> 0.86530394, -0.13721049, -0.87982548, -1.81113831, -0.60337380, -1~
## $ x10 <dbl> -0.67479792, 0.49138297, -0.40283973, 1.21654838, 1.37487525, -2.9~
## $ x11 <dbl> 0.4768842, 1.3187331, -0.2375626, -0.8417482, -0.2555588, -0.73944~
## $ x12 <dbl> -0.55028261, -0.57696349, 0.34973240, -0.31482804, -1.32927922, 1.~
## $ x13 <dbl> 1.14678993, -0.03428652, -0.36594800, -0.19693086, 2.64028389, -0.~
## $ x14 <dbl> 0.62387485, -0.25243935, -0.23542049, -1.60157449, 1.17348036, -1.~
## $ x15 <dbl> -0.11340307, 0.62118450, -0.07521731, 0.22553162, -0.03979362, -0.~
## $ x16 <dbl> -1.04538727, -1.65054917, 1.35264110, -0.74933493, -0.20964781, -0~
## $ x17 <dbl> 0.47243145, 0.05101400, -0.61833554, -0.03849856, -0.47878135, 0.8~
## $ x18 <dbl> -0.590890777, 1.169765794, -0.059145998, -0.309537577, 0.842366000~
## $ x19 <dbl> 0.53364049, 2.33425890, -0.47852405, 0.54132470, 0.60261169, -2.02~
## $ x20 <dbl> -0.05844756, -0.74187288, 0.95175354, -0.20784247, 1.13489628, -2.~
## $ x21 <dbl> 0.65525513, 1.68884588, -0.57038117, -1.80555409, 2.08148634, 0.53~
## $ x22 <dbl> -1.20221003, 1.74128942, -0.74659747, 0.87619987, -0.10779368, -0.~
## $ x23 <dbl> 0.87133442, -0.14821616, 2.14691210, -0.19088522, -0.29948310, 0.3~
## $ x24 <dbl> 0.87836580, 1.12460649, 0.22236027, 0.20390661, 0.72825984, -0.163~
## $ x25 <dbl> -0.64288824, -0.06167637, -0.34527940, 0.62755689, -0.29965224, -0~
## $ x26 <dbl> -0.43031692, -0.72294116, -0.02082838, -0.19986085, 0.94931216, -1~
## $ x27 <dbl> -0.549731535, 0.104640571, -0.005052764, -1.555809942, 0.800257477~
## $ x28 <dbl> -0.37636467, 1.05012853, 1.09066971, -1.73283373, -0.52552393, -0.~
## $ x29 <dbl> 1.29310734, 1.13690150, -0.32233373, -0.31890823, 0.40454183, -0.0~
## $ x30 <dbl> 1.60799472, -0.34339053, -1.94283061, -0.61647080, 0.74117929, 1.0~
## $ x31 <dbl> -1.8940459, -1.3444175, 0.4917738, 1.2478144, 0.7653535, 0.4159829~
## $ x32 <dbl> 1.69469353, 0.17893691, 0.08081505, -1.63310270, 0.75915432, 0.541~
## $ x33 <dbl> -1.55078952, -0.15141527, -0.44858391, 0.21312202, 0.63037564, 0.7~
## $ x34 <dbl> 2.07109783, 0.95276628, 1.32599060, 2.27539009, -0.12636968, 0.247~
## $ x35 <dbl> 1.82290334, -0.31907075, -1.16353613, 0.85407328, -1.28814754, 0.6~
## $ x36 <dbl> -0.7394679, -1.7322869, 1.0537616, 0.1366273, -0.4719353, 0.413739~
## $ x37 <dbl> 0.93508239, -0.79537446, -1.87254115, 0.11414416, 0.05009938, -0.1~
## $ x38 <dbl> -0.507117532, -0.001069243, -0.636333478, 0.419641050, -0.47657448~
```

```
## $ x39 <dbl> 0.22888128, 1.31815563, 0.42261732, 0.15486481, -0.94886750, 0.802~
## $ x40 <dbl> -1.53550785, 0.08945169, 0.10561738, 1.40392237, 3.16451628, -1.59~
## $ x41 <dbl> 0.54995962, -1.49673970, 0.45267944, 0.07628427, -0.55596341, -0.7~
## $ x42 <dbl> -1.8288926, 1.7451062, 0.8158462, -0.1390173, -0.8047000, -0.11506~
## $ x43 <dbl> 1.90648214, 0.67113809, -0.05362633, 0.39563373, 2.67730180, 0.922~
## $ x44 <dbl> 1.29447870, -1.68009812, -0.32558008, -1.59679394, -0.49488477, -0~
## $ x45 <dbl> 0.95338061, -0.99034220, -0.98995377, 0.26891339, 0.12166661, -0.9~
## $ x46 <dbl> -0.13483662, -0.50696860, -0.51738788, -2.06234042, -0.10680057, -~
## $ x47 <dbl> 2.43187944, 0.37847403, -0.06741761, 0.59418109, -0.22940658, -0.7~
## $ x48 <dbl> -0.9565450, 0.4803113, -0.3920700, -0.4585714, -1.9929885, 0.88366~
## $ x49 <dbl> 0.94853247, 0.26362181, -0.16464646, 0.11935923, -0.32447324, 0.04~
## $ x50 <dbl> 0.19018260, 1.13505566, -0.16972687, -1.50584636, -0.07841405, -1.~
## $ x51 <dbl> -0.6831468, 0.1100079, 0.7239210, 0.2311925, -0.7642252, 0.1093128~
## $ x52 <dbl> 0.65307722, -0.80046181, 0.86927368, 1.59861934, -0.47053886, -0.0~
## $ x53 <dbl> -0.95918072, 0.09582336, 0.05805066, -1.65059194, 0.52212944, -0.8~
## $ x54 <dbl> -0.41506279, 0.53415776, -1.08420393, 0.31151571, 1.11321863, 2.20~
## $ x55 <dbl> -0.9469963, -0.9124609, -1.4035349, -0.3556957, 0.7224917, -0.1489~
## $ x56 <dbl> 0.54883719, -1.91727346, 0.18654617, -0.75041670, -1.04649763, 0.3~
## $ x57 <dbl> -0.72422453, 1.43064413, -1.47964130, 0.57516981, 0.49719144, 0.58~
## $ x58 <dbl> -2.20950208, -0.23497601, 1.07557427, 0.24929374, 0.56432730, -0.2~
## $ x59 <dbl> -0.8485211, -0.5308077, 1.5559665, -1.6331253, -0.2573712, -0.2842~
## $ x60 <dbl> -2.52402189, 1.29349906, -0.20218647, 1.87527348, -0.87476342, -0.~
## $ x61 <dbl> -0.66892027, -1.72557732, 0.79925161, 1.80655230, 0.22016194, -2.3~
## $ x62 <dbl> -0.92960688, 0.04079371, 0.33876203, 0.72753901, -1.08300120, 0.71~
## $ x63 <dbl> -0.42021184, 2.03151546, -0.21235170, -0.54181759, 1.27431677, 0.1~
## $ x64 <dbl> -0.68218725, -0.13237431, 0.66637214, 0.63351491, 0.70331183, -1.2~
## $ x65 <dbl> -1.853949165, -0.006625952, -1.239688043, 1.724372575, -0.53278079~
## $ x66 <dbl> 0.758622374, -0.247157562, -1.170425464, 1.188988247, 0.423596811,~
## $ x67 <dbl> 1.506210427, 0.085497740, -1.698973185, 1.076071882, -0.459228447,~
## $ x68 <dbl> 0.32879336, 0.80760027, -2.87189688, 0.15125671, 0.24641242, 0.857~
## $ x69 <dbl> -0.22660315, -0.86527369, -0.37419920, 1.28663788, -0.17221500, -1~
## $ x70 <dbl> 0.15206087, 0.50449311, 1.24957901, -3.66360118, 0.46417993, -1.09~
## $ x71 <dbl> 1.10750748, -1.51821411, -1.04662914, 0.91478065, 0.31102705, -1.2~
## $ x72 <dbl> -0.46295847, 0.75343551, -1.12696912, -0.84281597, -1.22013393, 0.~
## $ x73 <dbl> -0.74889700, -1.29222023, -2.58930724, -1.74360767, -1.09158189, 0~
## $ x74 <dbl> 1.48771931, -0.56530125, -1.56187174, -1.10624410, -1.65851229, -0~
## $ x75 <dbl> -1.22958786, 0.57706123, 0.38631089, 1.37850476, -1.21528450, -2.7~
## $ y   <dbl> -5.033324, -8.874837, 1.720462, 16.535303, 22.683417, -3.642452, -~
```

The data are reshaped for you into *long-format*. The output `y` is preserved and all input columns are "stacked" on top of each other. The `name` column provides the input's name and the `value` column is the value of the input. The `input_id` column is created for you. It stores the input ID as an integer instead of a string. The `stringr` package is part of `tidyverse`.

```
lfA_train <- dfA_train %>%
  tibble::rowid_to_column() %>%
  pivot_longer(starts_with('x')) %>%
  mutate(input_id = as.numeric(stringr::str_extract(name, '\\d+')))
lfA_train
```

```
## # A tibble: 7,200 x 5
##    rowid     y name   value input_id
##    <int> <dbl> <chr>  <dbl>    <dbl>
##  1     1 -5.03 x01   -2.07         1
##  2     1 -5.03 x02   -0.692        2
```

```
##  3     1 -5.03 x03   -1.01         3
##  4     1 -5.03 x04    1.33         4
##  5     1 -5.03 x05    0.266        5
##  6     1 -5.03 x06   -0.725        6
##  7     1 -5.03 x07    0.119        7
##  8     1 -5.03 x08   -0.212        8
##  9     1 -5.03 x09    0.865        9
## 10     1 -5.03 x10   -0.675       10
## # i 7,190 more rows
```

1a)

It is always important to explore data before modeling. You will use **regularized regression** models in this assignment. Regularized regression requires that all features have roughly the same scale in order for the **regularization strength** to have the same influence.

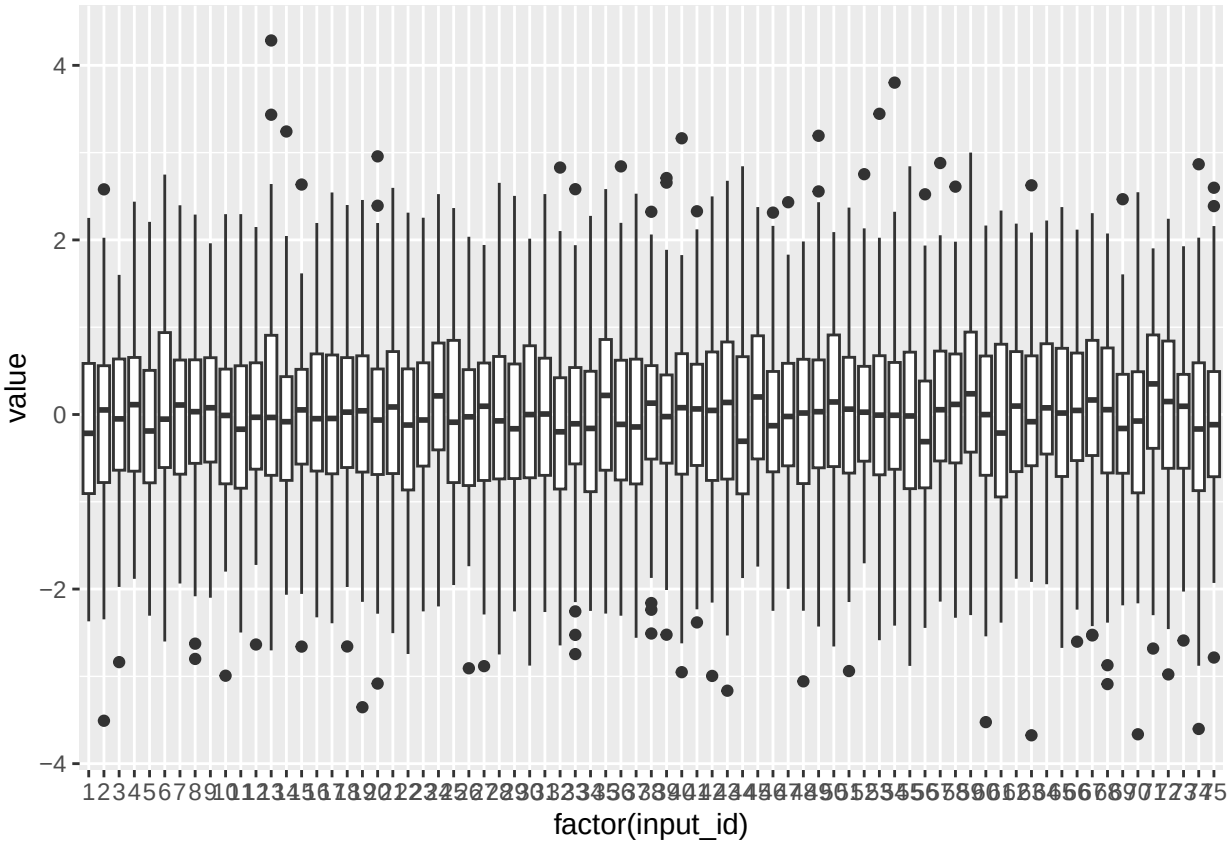
Use a boxplot to check if all inputs have roughly the same scale. Pipe the long-format data, `lfA_train` to `ggplot()` and map the `input_id` to the `x` aesthetic and the `value` to the `y` aesthetic. Add the `geom_boxplot()` layer and map the `input_id` to the `group` aesthetic.

How do the scales compare across the inputs?

HINT: Remember the importance of the `aes()` function!

SOLUTION Yes, in the plot below, it looks like all inputs have roughly the same scale.

```
ggplot(lfA_train, aes(x = factor(input_id), y = value)) +
  geom_boxplot(aes(group = factor(input_id)))
```



1b)

Let's now visualize the relationship between the output, `y`, and each input. Each input will be associated with a facet, but you will still need to show several figures because there are so many inputs in this problem.

Create a scatter plot to show the relationship between the output and each input. Use facets for each input.

In the first code chunk, pipe the long-format data to `filter()` and keep all rows where `input_id` is less than 26. Pipe the result to `ggplot()` and map `value` to the `x` aesthetic and map `y` to the `y` aesthetic. Add the appropriate geom to make the scatter plot and add a `facet_wrap()` layer with the facets "as a function" of `name`.

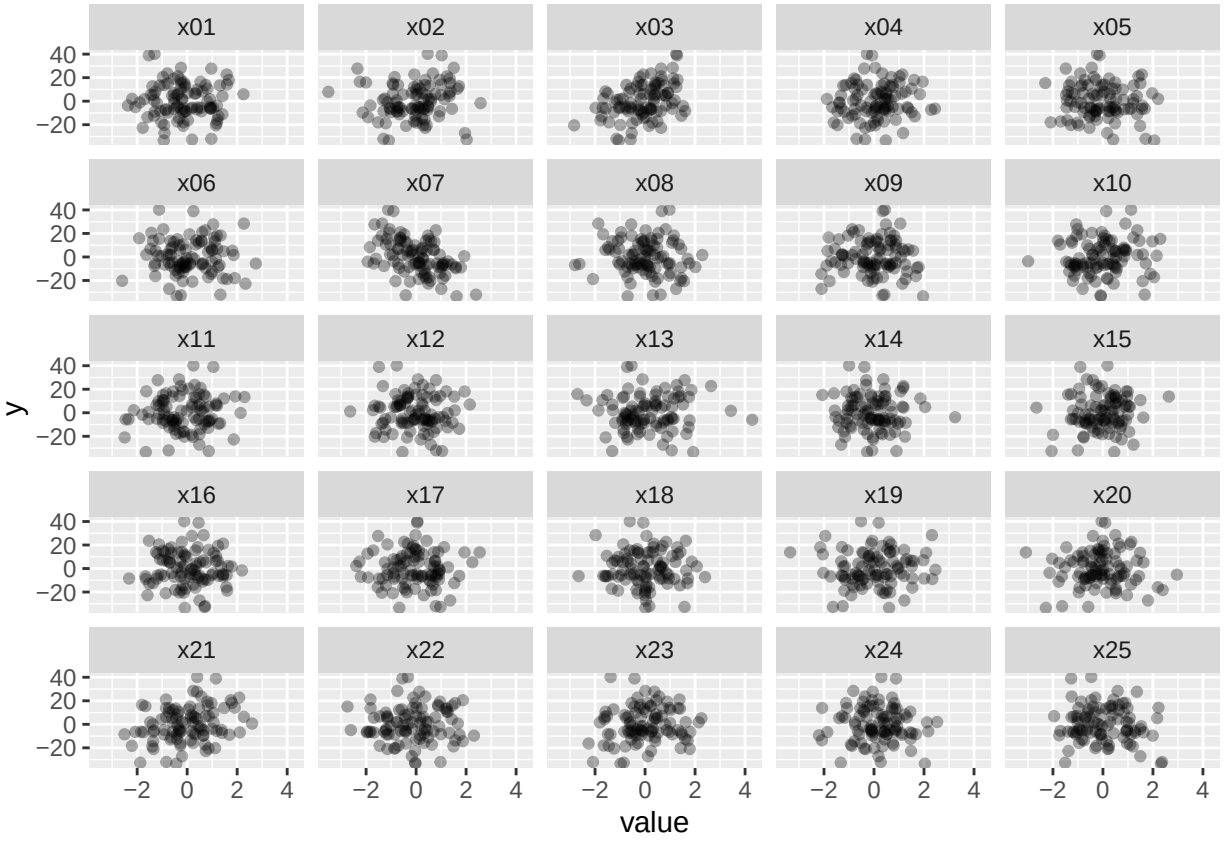
In the second code chunk, pipe the long-format data to `filter()` and keep all rows where `input_id` is between 26 and 50. Use the same aesthetics and facet structure as the first code chunk.

In the third code chunk, pipe the long-format data to `filter()` and keep all rows where `input_id` is greater than 50. Use the same aesthetics and facet structure as the first code chunk.

Set the `alpha` to 0.33 in all three scatter plots.

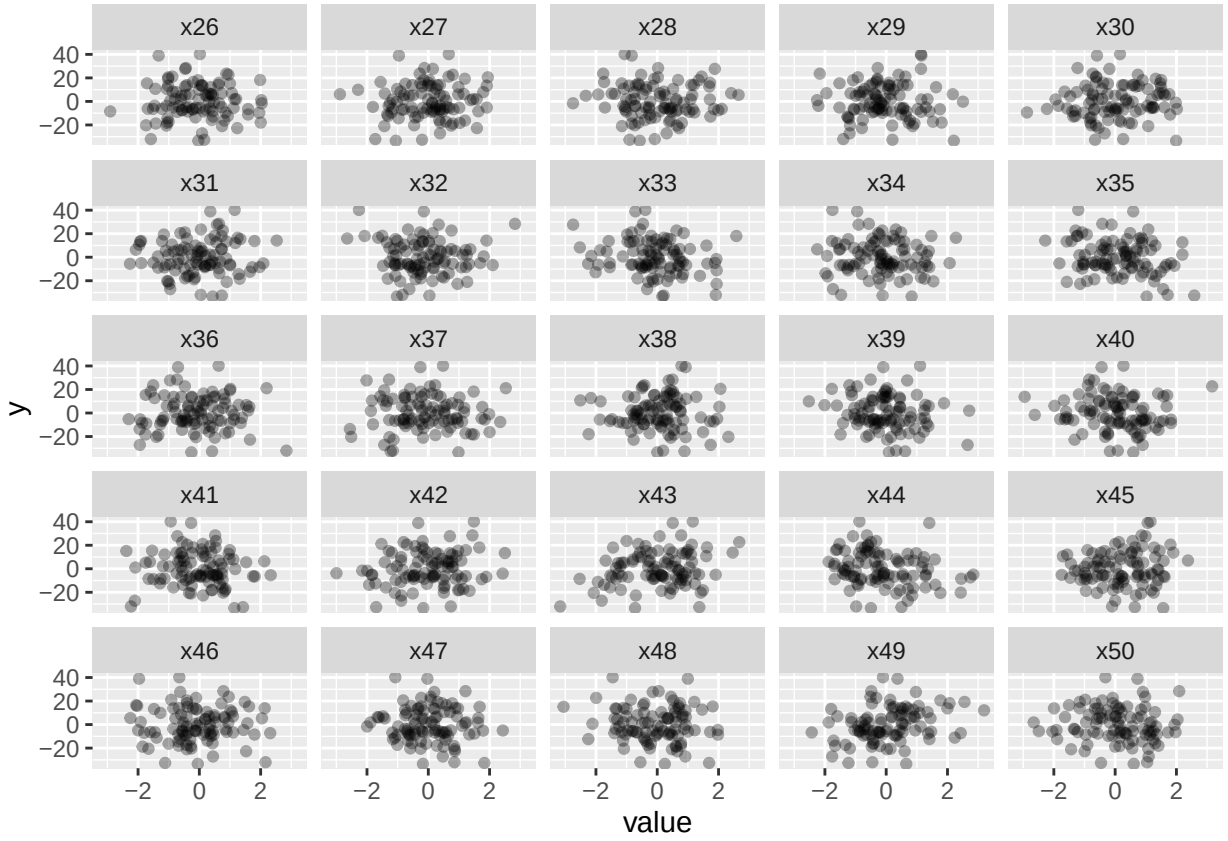
HINT: The `between()` function can help you here...

```
lfA_train %>%
  filter(input_id < 26) %>%
  ggplot(aes(x = value, y = y)) +
  geom_point(alpha= 0.33) +
  facet_wrap(~name)
```

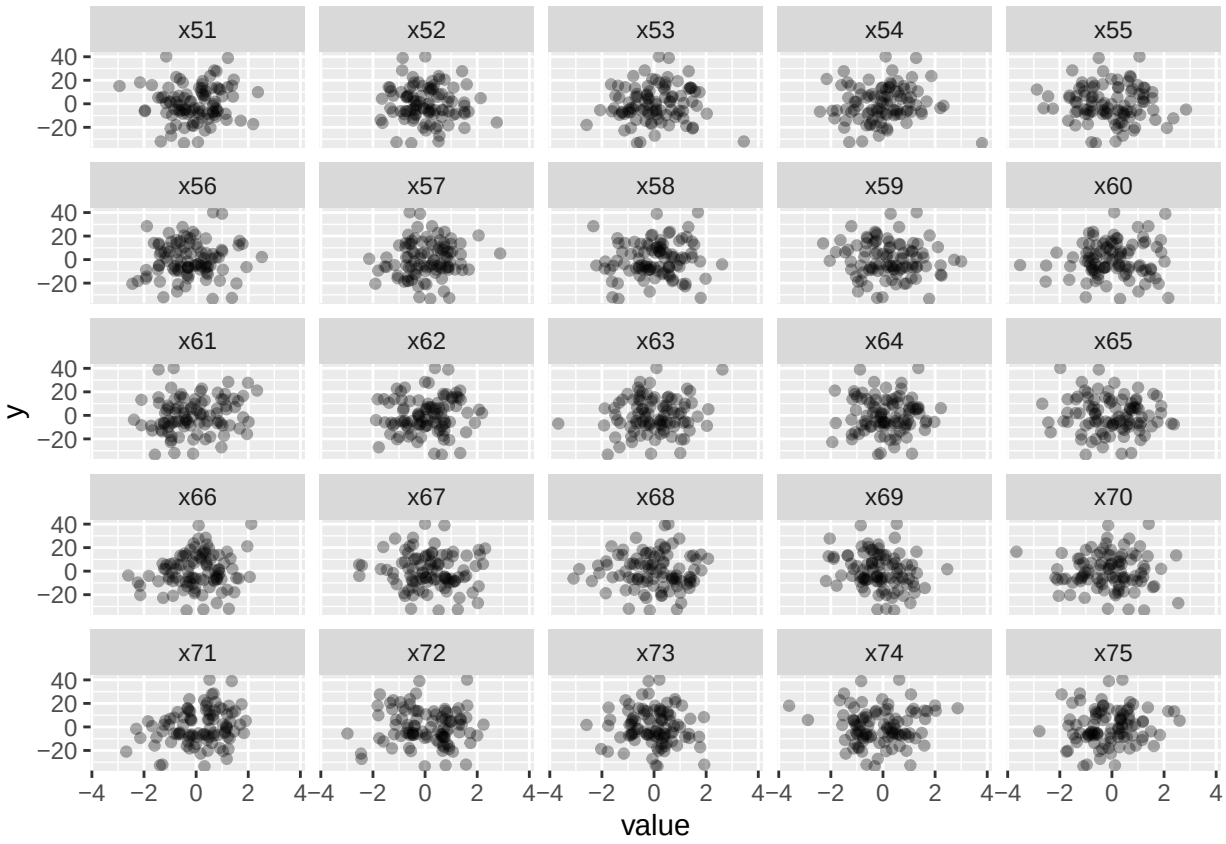


SOLUTION

```
lfA_train %>%
  filter(input_id >= 26, input_id <= 50) %>%
  ggplot(aes(x = value, y = y)) +
  geom_point(alpha= 0.33) +
  facet_wrap(~name)
```




```
lfA_train %>%
  filter(input_id > 50) %>%
  ggplot(aes(x = value, y = y)) +
  geom_point(alpha= 0.33) +
  facet_wrap(~name)
```



1c)

Based your previous figures, which inputs seem related to the output?

SOLUTION All 75 figures look similar above and there do not seem to be a direct relationship (e.g. linear or quadratic) between inputs and output y.

1d)

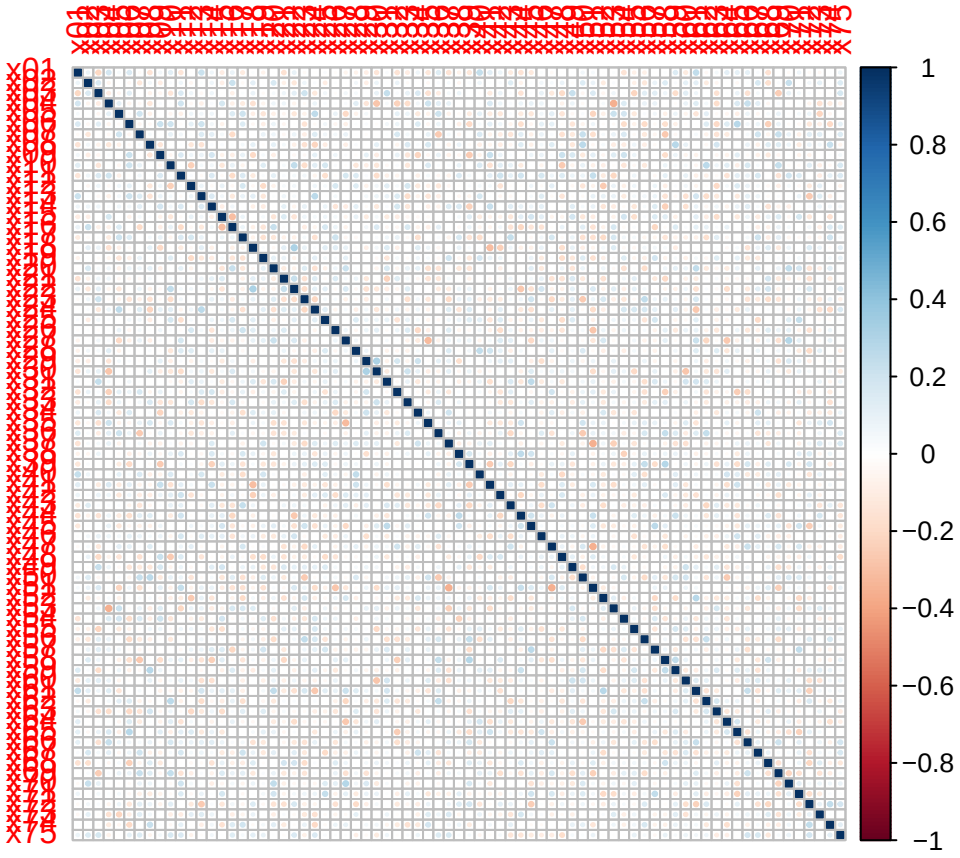
You visualized the output to input relationships, but you must also examine the relationship between the inputs! The `cor()` function can be used to calculate the correlation matrix between all columns in a data frame.

Correlation matrices are created from wide-format data and so you will use the original wide-format data, `dfA_train`, for this question instead of the long-format data, `lfA_train`. However, you already know there are many inputs to this problem. Thus, you will **not** display the correlation matrix between all inputs. Instead, you will visualize the correlation matrix as a heat map. This type of plot is typically referred to as the **correlation plot**. Let's create a correlation plot to visualize the correlation coefficient between each pair of inputs. The `corrplot` package provides the `corrplot()` function to easily create clean and simple correlation plots. You do not have to load or import the `corrplot` package, instead you will call the `corrplot()` function from `corrplot` using the `::` operator. Thus, you will call the function as `corrplot::corrplot()`.

The first argument to `corrplot::corrplot()` is a correlation matrix. You must therefore calculate the correlation matrix associated with a data frame and pass that matrix into `corrplot::corrplot()`. You will create the default correlation plot from `corrplot()`. You will gain more experience with correlation plots later in this assignment.

Pipe `dfA_train` into `select()` and select all columns EXCEPT the response `y`. Pipe the result to the `cor()` function and then pipe the result to the `corrplot::corrplot()` function.

```
dfA_train %>%
  select(-y) %>%
  cor() %>%
  corrplot::corrplot()
```



SOLUTION

1e)

Based on your correlation plot in 1d), do you think the inputs are highly correlated?

SOLUTION I think the inputs are not highly correlated because the plot in Problem 1d) shows mostly light-colored blocks.

1f)

Let's fit a non-Bayesian linear model by maximizing the likelihood. You do not have program the model from scratch. Instead, you are allowed to use `lm()` to fit the model.

Fit a linear model by maximizing the likelihood using linear additive features for all inputs.

You must use the original wide-format data, `dfA_train`.

Assign the model to the `modA` object.

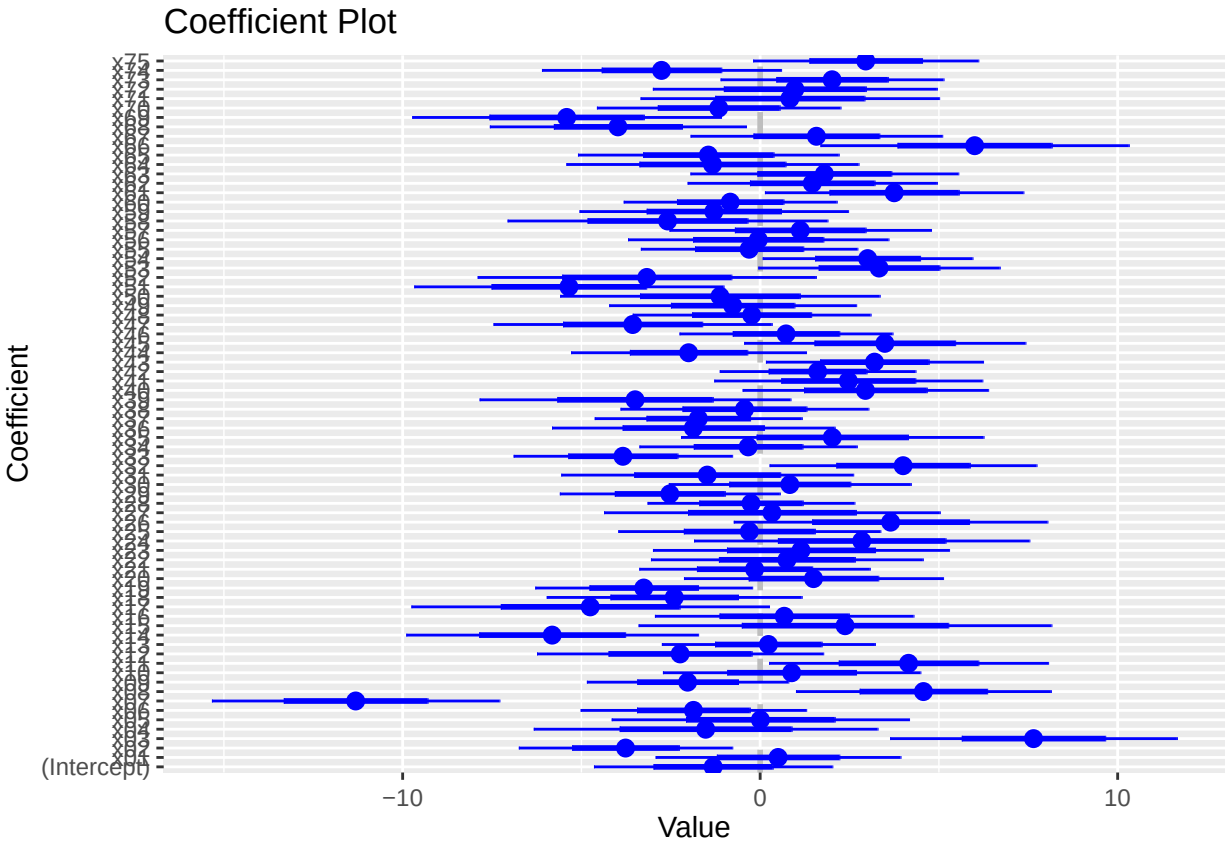
```
modA <- lm(y ~ ., data = dfA_train)
```

SOLUTION

1g)

Use the `coefplot::coefplot()` function to visualize the coefficient summaries for your linear additive features model.

```
coefplot::coefplot(modA)
```



SOLUTION

1h)

There are other ways to examine the coefficient summaries besides the coefficient plot visualization. The `broom` package has multiple functions which can help interpret model behavior. The `broom` package organizes model results into easy to manipulate TIDY dataframes. You will use the `broom::tidy()` function to examine the coefficient summaries. Applying the `broom::tidy()` function to an `lm()` model object produces a dataframe. One row is one coefficient. The `term` column is the name of feature the coefficient is applied to and the other columns provide different statistics about the coefficient. You can use any other `tidyverse` function, such as `select()`, `filter()`, `count()`, and others to manipulate the `broom::tidy()` returned dataframe!

Non-Bayesians focus on the p-value (`p.value` column from `broom::tidy()`) to identify if the feature is statistically significant.

Use `broom::tidy()` to identify which inputs non-Bayesians will identify as being statistically significant.

HINT: The common convention for statistical significance is 0.05...

SOLUTION As shown in the table below, 14 of the inputs are statistically significant.

```
tidy_modA <- broom::tidy(modA)

significant_inputs_modA <- tidy_modA %>% filter(p.value <0.05)

significant_inputs_modA
```

```
## # A tibble: 14 x 5
##    term  estimate std.error statistic   p.value
##    <chr>    <dbl>     <dbl>     <dbl>     <dbl>
##  1 x02      -3.76      1.49     -2.52 0.0203
##  2 x03       7.65      2.01      3.81 0.00109
##  3 x07     -11.3       2.01     -5.63 0.0000165
##  4 x08       4.57      1.78      2.56 0.0186
##  5 x11       4.15      1.95      2.13 0.0460
##  6 x14      -5.82      2.04     -2.85 0.00989
##  7 x19      -3.26      1.52     -2.15 0.0443
##  8 x32       4.00      1.87      2.14 0.0447
##  9 x33      -3.84      1.53     -2.51 0.0207
## 10 x43       3.20      1.52      2.11 0.0480
## 11 x51      -5.35      2.16     -2.47 0.0225
## 12 x66       6.00      2.16      2.78 0.0115
## 13 x68      -3.98      1.79     -2.22 0.0381
## 14 x69      -5.41      2.16     -2.51 0.0209
```

Problem 02

You have explored the data and fit the first un-regularized non-Bayesian model. You have examined the statistically significant features, but now you must use regularized regression to identify the important features. Regularized regression is an important tool for "turning off" unimportant features in models. Regularized regression is especially useful when there are many features, as with this application. Consider for example that you are working at a startup company that produces components for medical devices. Your company makes the components with Additive Manufacturing (3D printing) and are trying to maximize the longevity of the printed part. Additive Manufacturing involves many different features that control the performance of a component. The company has tracked variables associated with the supplied material, the operation of the printer, and measured features associated with the printed objects. The company identified 75 inputs they think impact a Key Performance Indicator (KPI). It is your job to identify the most important inputs that influence the output!

This may seem like a daunting task, but that is where regularized regression can help! In lecture we introduced non-Bayesian regularized regression by stating the regression coefficients, the $\boldsymbol{\beta}'s$, are estimated by minimizing the following loss function:

$$SSE + \lambda \times penalty$$

The *penalty* formulation depends on if we are using Ridge or Lasso regression. However, in this assignment we will work with a slightly different loss function. We will work with the loss function consistent with the `glmnet` package:

$$\frac{1}{2}MSE + \lambda \times penalty$$

This does not change the concepts discussed in lecture. It simply changes the magnitude of the **regularization strength** parameter, λ. As an aside, this demonstrates why I prefer the Bayesian interpretation of regularization. The regularization is more interpretable in the Bayesian setting. The regularization is the

prior standard deviation which controls the allowable range on the coefficients. But that is a philosophical debate for another time!

You must program the regularized loss function yourself, rather than using `glmnet` to fit the regularized regression model! This will highlight how the regularization strength is applied and how it relates to the SSE (or MSE). This problem deals with programming the **RIDGE** penalty and comparing the resulting **RIDGE** estimates to the coefficient MLEs.

2a)

You will program the Ridge loss function in a manner consistent with the log-posterior functions from previous assignments. You must create an initial list of required information before you program the function. This will allow you to test your function to make sure you programmed it correctly.

Complete the code chunk below by filling in the fields to the list of required information. The `$yobs` field is the "regular" R vector for the output. The `$design_matrix` is the design matrix for the problem. The `$lambda` field is the regularization strength, λ.

You must use the original wide format `dfA_train` to specify the output and create the design matrix. The design matrix must be made consistent with how we have organized design matrices in lecture and thus must include the "intercept column of 1s". Your design matrix should use linear additive features for all inputs. Lastly, set the regularization strength to 1.

```
check_info_A <- list(
  yobs = dfA_train$y,
  design_matrix = model.matrix(y ~ ., data = dfA_train),
  lambda = 1
)
```

SOLUTION

2b)

The function, `loss_ridge()`, is started for you in the code chunk below. The `loss_ridge()` function consists of two arguments. The first, `betas`, is the "regular" R vector of the regression coefficients and the second, `my_info`, is the list of required information. You must assume that `my_info` contains fields you specified in the `check_info_A` list in the previous problem.

Complete the code chunk below by calculating the mean trend, the model's mean squared error (MSE) on the training set, the RIDGE penalty, lastly return the effective Ridge loss. The comments and variable names below state what you should calculate at each line.

To check if you programmed `loss_ridge()` correctly, test the function with a guess of -1.2 for all regression coefficients. If you programmed `loss_ridge()` correctly you should get a value of 287.3114. As another test, use 0.2 for all regression coefficients. If you programmed `loss_ridge()` correctly you should get a value of 111.2219.

```
loss_ridge <- function(betas, my_info)
{
  # extract the design matrix
  X <- my_info$design_matrix

  # calculate linear predictor
  mu <- X %*% betas
```

```
  # calculate MSE
  MSE <- mean((my_info$yobs - mu)^2)

  # calculate ridge penalty
  penalty <- sum(betas^2)

  # return effective total loss
  total_loss <- (1/2)*MSE + my_info$lambda*penalty

  return(total_loss)
}
```

SOLUTION Test `loss_ridge()` with -1.2 for all regression coefficients.

Test `loss_ridge()` with 0.2 for all regression coefficients.

The following code chunks test `loss_ridge()` function and the desired values are acquired.

```
betas_test_1 <- rep(-1.2, ncol(check_info_A$design_matrix))
betas_test_2 <- rep(0.2, ncol(check_info_A$design_matrix))

loss_1 <- loss_ridge(betas_test_1, check_info_A)
print(loss_1)
```

```
## [1] 287.3114
```

```
loss_2 <- loss_ridge(betas_test_2, check_info_A)
print(loss_2)
```

```
## [1] 111.2219
```

2c)

You will use the `optim()` function to manage the optimization and thus fitting of the Ridge regression model. In truth, Ridge regression has a closed form analytic expression for the β estimates! The formula has a lot in common with the Bayesian posterior mean formula assuming a known σ! However, we will focus on executing and interpreting the results rather than the mathematical derivation in this assignment.

You must define a function, `fit_regularized_regression()`, to manage the optimization for you. This function will be general and not specific to Ridge regression. This way you can use this same function later in the assignment to fit the Lasso regression model. The `fit_regularized_regression()` function is started for you in the code chunk below and has three arguments. The first, `lambda_use`, is the assumed regularization strength parameter value. The second, `loss_func`, is a function handle for the loss function you are using to estimate the coefficients, and the third argument, `my_info`, is the list of required information.

The `my_info` argument is different from `check_info_A` you defined previously. The `my_info` argument in `fit_regularized_regression()` does NOT include `$lambda`. Instead, you must assign `lambda_use` to the `$lambda` field. The purpose of `fit_regularized_regression()` is to allow iterating over many different values for λ.

The bulk of `fit_regularized_regression()` is calling `optim()` to execute the optimization. You must specify the arguments correctly to the `optim()` function. Please note that you are **minimizing** the loss and so you should **not** specify `fnscale` in the `control` argument to `optim()`.

The last portion of `fit_regularized_regression()` is a book keeping step which organizes the coefficient estimates in a manner consistent with the `broom::tidy()` function.

Complete the code chunk below by specifying the initial guess as 0 for all regression coefficients, assigning `lambda_use` to the `$lambda` field, complete the arguments to the `optim()` call, and

complete the book keeping correctly. The `estimate` column in the returned `tibble` is the optimized estimate for the coefficients. The `term` column in the returned `tibble` is the coefficient name. You do NOT need to return the Hessian matrix from `optim()` and you should specify the `method` is `'BFGS"` and the maximum number of iterations is 5001.

HINT: How can you identify the feature names associated with each coefficient?

HINT: Which of the returned elements from `optim()` correspond to the optimized estimates?

```
fit_regularized_regression <- function(lambda_use, loss_func, my_info)
{
  # create the initial guess of all zeros
  start_guess <- rep(0, ncol(my_info$design_matrix))

  # add the regularization strength to the list of required information
  my_info$lambda <- lambda_use

  fit <- optim(
    par = start_guess,
    fn = loss_func,
    my_info = my_info,
    method = 'BFGS',
    control = list(maxit = 5001))

  feature_names <- colnames(my_info$design_matrix)[-1]

  # return the regularized beta estimates
  tibble::tibble(
    term = c("Intercept", feature_names),
    estimate = fit$par
  ) %>%
    mutate(lambda = lambda_use)
}
```

SOLUTION

2d)

You need to specify a new list of required information that only contains the output and design matrix in order to test the `fit_regularized_regression()` function.

Complete the first code chunk below by filling in the fields to the list of required information. The `$yobs` field is the "regular" R vector for the output. The `$design_matrix` is the design matrix for the problem. You must use the original wide format `dfA_train` to specify the output and create the design matrix and you use should linear additive features for all inputs.

Complete the second code chunk below by fitting the Ridge regularized model with a low regularization strength value of 0.0001. Assign the result to the `check_ridge_fit_lowpenalty` object. Display the `glimpse()` of `check_ridge_fit_lowpenalty` to the screen.

```
reg_info <- list(
  yobs = dfA_train$y,
  design_matrix = model.matrix(y ~ ., data = dfA_train)
)
```

SOLUTION Fit the Ridge model with a regularization strength of 0.0001.

```
check_ridge_fit_lowpenalty <- fit_regularized_regression(0.0001, loss_ridge, reg_info)

glimpse(check_ridge_fit_lowpenalty)
```

```
## Rows: 76
## Columns: 3
## $ term     <chr> "Intercept", "x01", "x02", "x03", "x04", "x05", "x06", "x07",~
## $ estimate <dbl> -1.31028135, 0.49421026, -3.74499404, 7.62406592, -1.46362043~
## $ lambda   <dbl> 1e-04, 1e-04, 1e-04, 1e-04, 1e-04, 1e-04, 1e-04, 1e-04, 1e-04~
```

2e)

The code chunk below is completed for you. It defines a function, `viz_compare_to_mles()`, which compares the regularized coefficient estimates to the MLEs. The first argument is a dataframe (tibble) returned from the `fit_regularized_regression()` function. The second argument is an `lm()` model object.

```
### create function to visualize the coefficient estimates
viz_compare_to_mles <- function(regularized_estimates, lm_mod)
{
  regularized_estimates %>%
    left_join(broom::tidy(lm_mod) %>%
                select(term, mle = estimate),
              by = 'term') %>%
    pivot_longer(c("estimate", "mle")) %>%
    filter(term != '(Intercept)') %>%
    ggplot(mapping = aes(y = value, x = term)) +
    geom_hline(yintercept = 0, color = 'grey30',
               size = 1.2, linetype = 'dashed') +
    stat_summary(geom = 'linerange',
                 fun.max = 'max', fun.min = 'min',
                 mapping = aes(group = term),
                 color = 'black', size = 1) +
    geom_point(mapping = aes(color = name,
                             shape = name)) +
    coord_flip() +
    scale_shape_discrete('', solid = FALSE) +
    scale_color_brewer('', palette = 'Set1') +
    labs(y = 'coefficient estimate', x = 'coefficient') +
    theme_bw()
}
```

Use the `viz_compare_to_mles()` function to compare the Ridge regularized coefficients with the low λ value to the coefficient MLEs.

How do they compare?

```
viz_compare_to_mles(check_ridge_fit_lowpenalty, modA)
```

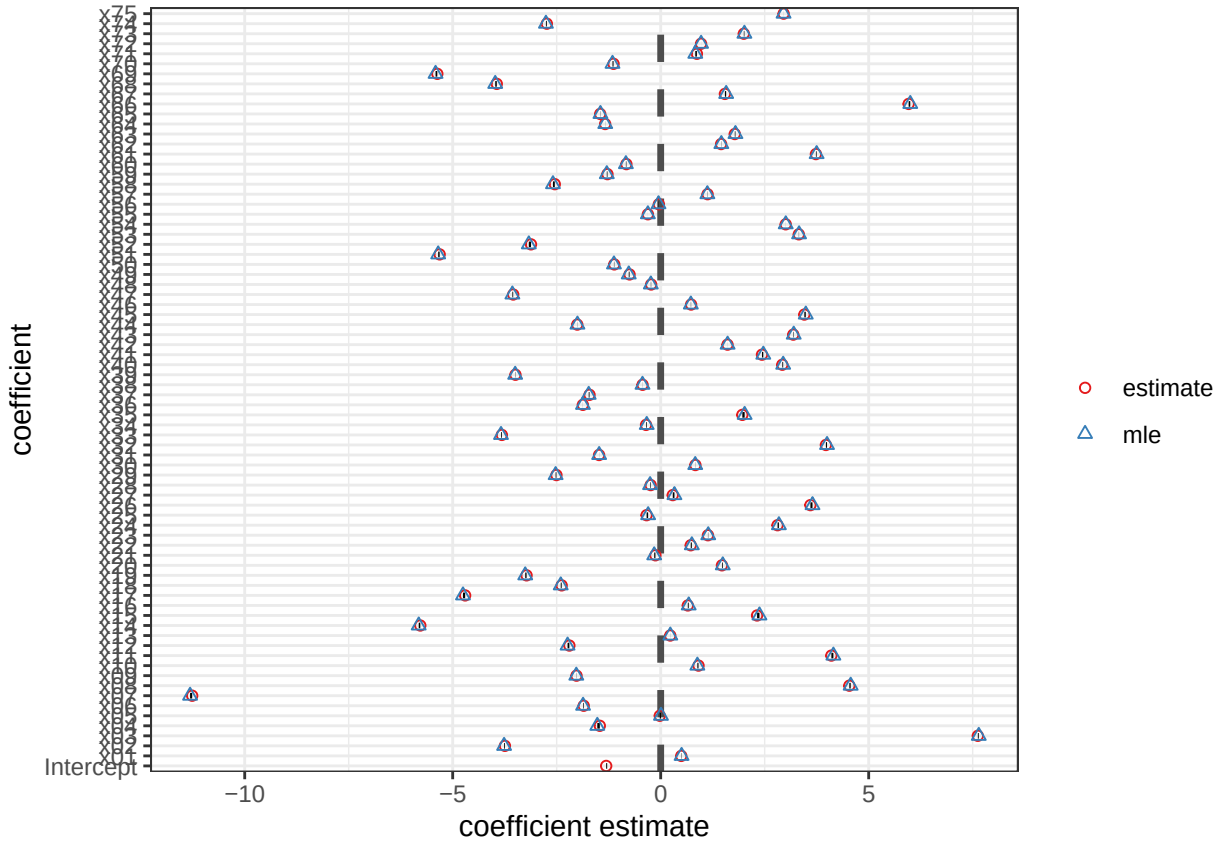
SOLUTION

```
## 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.
```

```
## Warning: Removed 1 rows containing non-finite values (`stat_summary()`).
```

```
## Warning: Removed 1 rows containing missing values (`geom_point()`).
```



How do the coefficients compare?

As it can be seen in the plot above, the coefficient values are very close to each other because a low lambda value in Ridge regression does not enforce much regularization.

2f)

Let's now try a high regularization strength of 10000 and see the impact on the coefficient estimates.

Use the `fit_regularized_regression()` to fit the Ridge regression model with a λ value of 10000. Assign the result to the `check_ridge_fit_highpenalty` object. Use the `viz_compare_to_mles()` function to compare the high penalized Ridge coefficient estimates to the MLEs.

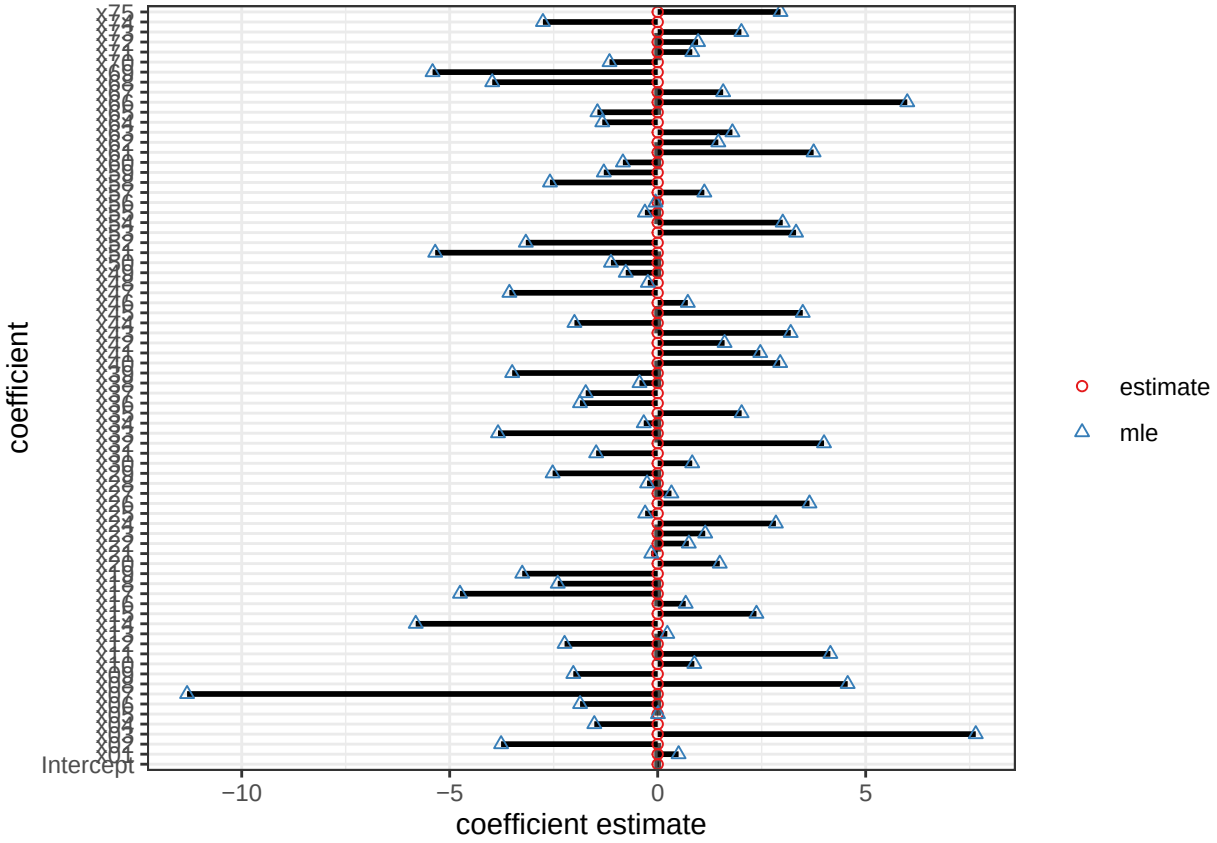
How do the high penalized estimates compare the MLEs?

```
check_ridge_fit_highpenalty <- fit_regularized_regression(10000, loss_ridge, reg_info)
viz_compare_to_mles(check_ridge_fit_highpenalty, modA)
```

SOLUTION

```
## Warning: Removed 1 rows containing non-finite values (`stat_summary()`).
```

```
## Warning: Removed 1 rows containing missing values (`geom_point()`).
```



How do they coefficients compare to the MLEs?

As it can be seen in the plot above, the coefficient values are not close to each other. This is because with a high value of lambda, the Ridge regression coefficients should be considerably shrunken towards zero compared to the MLEs, as it is observed in the plot.

2g)

The first argument to the `fit_regularized_regression()` function was intentionally set to `lambda_use` to allow iterating over a sequence of λ values. This allows us to try out dozens to hundreds of candidate λ values and see the regularization effect on the coefficient estimates.

However, you must first create a sequence of candidate λ values to iterate over!

Define a sequence of lambda values and assign the result to the `lambda_grid` variable in the first code chunk below. The grid should be 101 evenly spaced points in the natural log scale. The values in `lambda_grid` however must be in the original λ scale and thus NOT in the log-scale. The lower bound should correspond to $\lambda = 0.0001$ and the upper bound should correspond to $\lambda = 10000$.

The second code chunk executes the iteration for you with the `purrr::map_dfr()` function. The `map_dfr()` function applies the `fit_regularized_regression()` function to each unique value in the `lambda_grid` vector.

```
log_lambdamin <- log(0.0001)
log_lambdamax <- log(10000)
log_lambdagrid <- seq(log_lambdamin, log_lambdamax, length.out = 101)
lambda_grid <- exp(log_lambdagrid)
```

SOLUTION The `eval` flag is set to FALSE in the code chunk arguments below. Set `eval=TRUE` once you have properly defined `lambda_grid`.

Please note: The code chunk below may take a few minutes to complete.

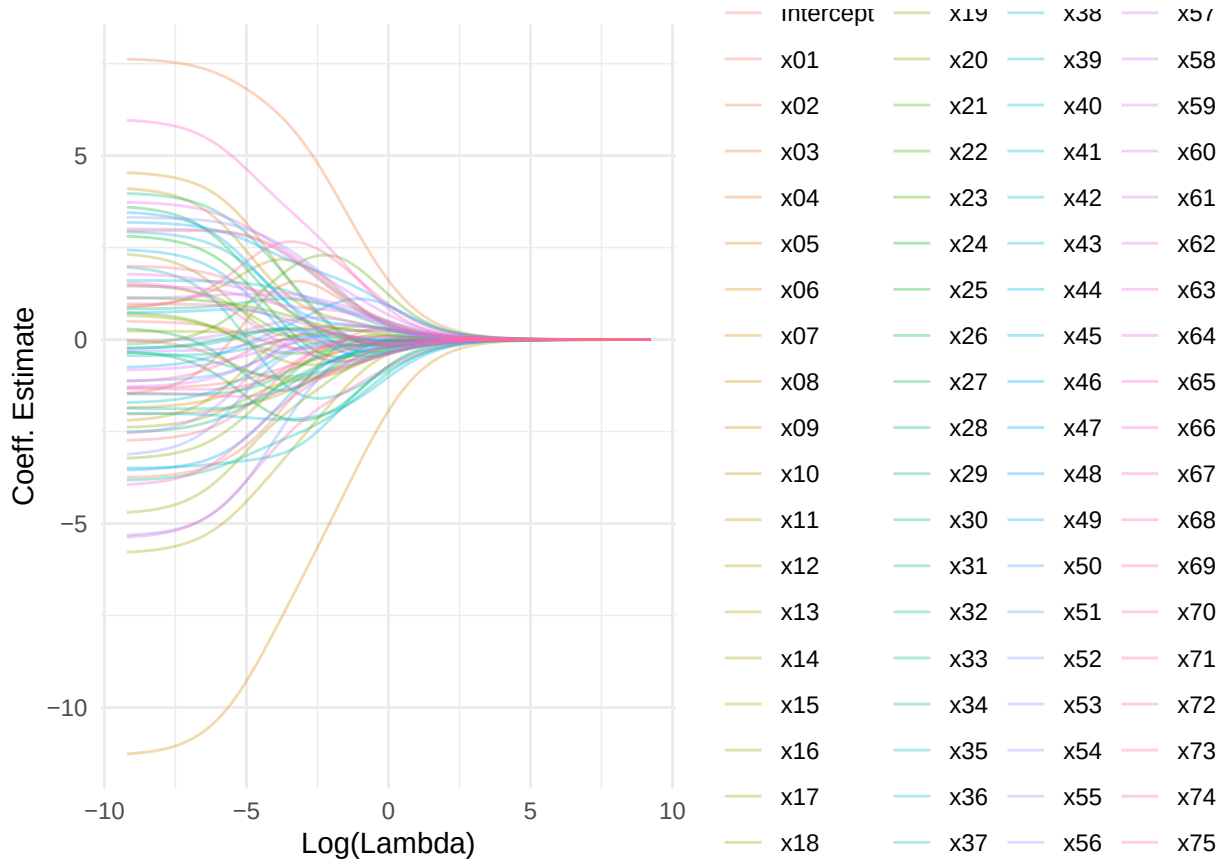
```
ridge_path_results <- purrr::map_dfr(lambda_grid,
                                     fit_regularized_regression,
                                     loss_func = loss_ridge,
                                     my_info = reg_info)
```

2h)

It's now time to visualize the **coefficient path** for the Ridge regression model! The code chunk below is started for you. The Intercept is removed to be consistent with the coefficient path visualizations created by `glmnet` package.

Complete the code chunk below by piping the dataframe to `ggplot()`. Map the `log(lambda)` to the `x` aesthetic and the `estimate` to the `y` aesthetic. Include a `geom_line()` layer and map the `term` variable to the `group` aesthetic within `geom_line()`. Set the `alpha` to 0.33.

```
ridge_path_results %>%
  filter(term != '(Intercept)') %>%
  ggplot(aes(x = log(lambda), y = estimate, group=term)) +
  geom_line(aes(color = term), alpha = 0.33) +
  labs(x = "Log(Lambda)", y = "Coeff. Estimate") +
  theme_minimal()
```



SOLUTION

2i)

Describe the behavior of the coefficient paths as the regularization strength increases in the plot shown in 2h).

SOLUTION

1) The absolute values of the coefficients tend to decrease. This is due to the penalty term in the Ridge regression which increases with lambda.
2) The coefficient paths tend to move smoothly towards zero rather than taking abrupt steps. This is a characteristic feature of Ridge regression where coefficients may become very small (but not exact zeros).
3) As lambda approaches a very large value, all coefficients will converge to zero. The regularization effect is so strong that the model becomes equivalent to a null model.

Problem 03

As discussed in lecture the Ridge penalty is analogous to applying independent Gaussian priors in Bayesian linear models! However, the regularization penalty can take other forms besides the Ridge penalty. The **LASSO** penalty behaves differently than Ridge. This problem is focused on applying the Lasso penalty and comparing it to the behavior of the Ridge penalty.

You will begin by defining a loss function similar to the `loss_ridge()` function you defined earlier. You will then use this new function, `loss_lasso()`, to fit the Lasso regression model.

3a)

The function, `loss_lasso()`, is started for you in the code chunk below. The `loss_lasso()` function has the same structure as the `loss_ridge()` function and thus has 2 arguments. The first, `betas`, is the "regular" R vector of the regression coefficients and the second, `my_info`, is the list of required information. You must assume that `my_info` contains the same fields as the `my_info` list used within `loss_ridge()`.

Complete the code chunk below by calculating the mean trend, the model's mean squared error (MSE) on the training set, the LASSO penalty, lastly return the effective Lasso loss. The comments and variable names below state what you should calculate at each line.

To check if you programmed `loss_lasso()` correctly, test the function with a guess of -1.2 for all regression coefficients and use the `check_info_A` list defined earlier. If you programmed `loss_lasso()` correctly you should get a value of 269.0714. As another test, use 0.2 for all regression coefficients and use the `check_info_A` list defined earlier. If you programmed `loss_lasso()` correctly you should get a value of 123.3819.

```
loss_lasso <- function(betas, my_info)
{
  # extract the design matrix
  X <- my_info$design_matrix

  # calculate linear predictor
  mu <- X %*% betas

  # calculate MSE
  MSE <- mean((my_info$yobs - mu)^2)

  # calculate LASSO penalty
  penalty <- sum(abs(betas))
```

```
  # return effective total loss
  total_loss <- (1/2)*MSE + my_info$lambda*penalty
  return(total_loss)
}
```

SOLUTION Test `loss_lasso()` with -1.2 for all regression coefficients.

Test `loss_lasso()` with 0.2 for all regression coefficients.

The following code chunks test `loss_ridge()` function and the desired values are acquired.

```
betas_test_3 <- rep(-1.2, ncol(check_info_A$design_matrix))
betas_test_4 <- rep(0.2, ncol(check_info_A$design_matrix))

loss_3 <- loss_lasso(betas_test_3, check_info_A)
print(loss_3)
```

```
## [1] 269.0714
```

```
loss_4 <- loss_lasso(betas_test_4, check_info_A)
print(loss_4)
```

```
## [1] 123.3819
```

3b)

You now have everything ready to fit the Lasso regression model! The `fit_regularized_regression()` function was defined to be general. You should use `fit_regularized_regression()` to fit the Lasso model by assigning `loss_lasso` to the `loss_func` argument. You must use the `reg_info` list as the `my_info` argument. The user supplied `lambda_use` value will then be used as the regularization strength applied to the Lasso penalty.

Fit the Lasso regression model with a low penalty of 0.0001 and assign the result to the `check_lasso_fit_lowpenalty` object.

After fitting, use the `viz_compare_to_mles()` function to compare the Lasso estimates to the MLEs.

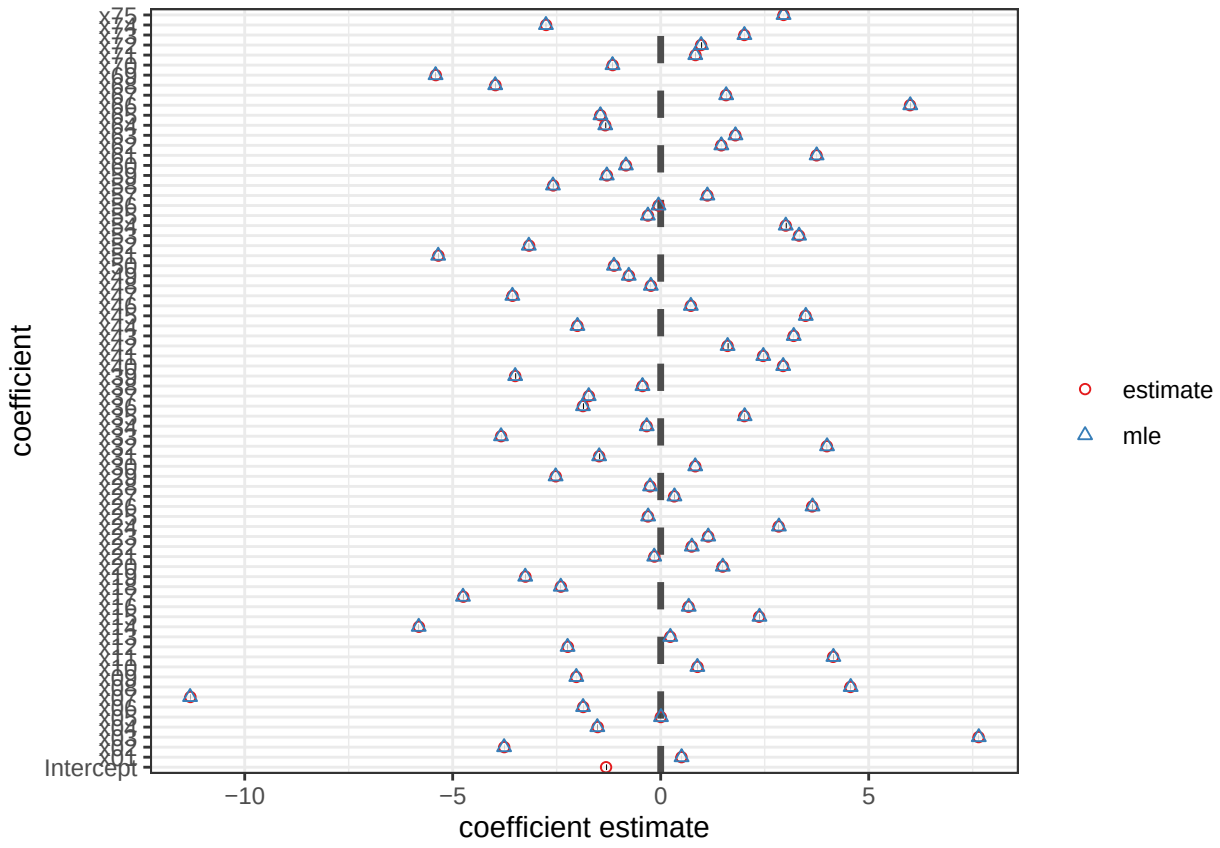
How do the Lasso estimates compare to the MLEs?

```
check_lasso_fit_lowpenalty <- fit_regularized_regression(0.0001, loss_lasso, reg_info)
viz_compare_to_mles(check_lasso_fit_lowpenalty, modA)
```

SOLUTION

```
## Warning: Removed 1 rows containing non-finite values (`stat_summary()`).
```

```
## Warning: Removed 1 rows containing missing values (`geom_point()`).
```



How do they compare?

As it can be seen in the plot above, the coefficient values are very close to each other because with a low penalty parameter, the Lasso estimates should be quite close to the MLEs since the regularization effect is minimal.

3c)

Next, let's examine how a strong Lasso penalty impacts the coefficients.

Fit the Lasso regression model again but this time with a high penalty of 10000 and assign the result to the `check_lasso_fit_highpenalty` object.

After fitting, use the `viz_compare_to_mles()` function to compare the Lasso estimates to the MLEs.

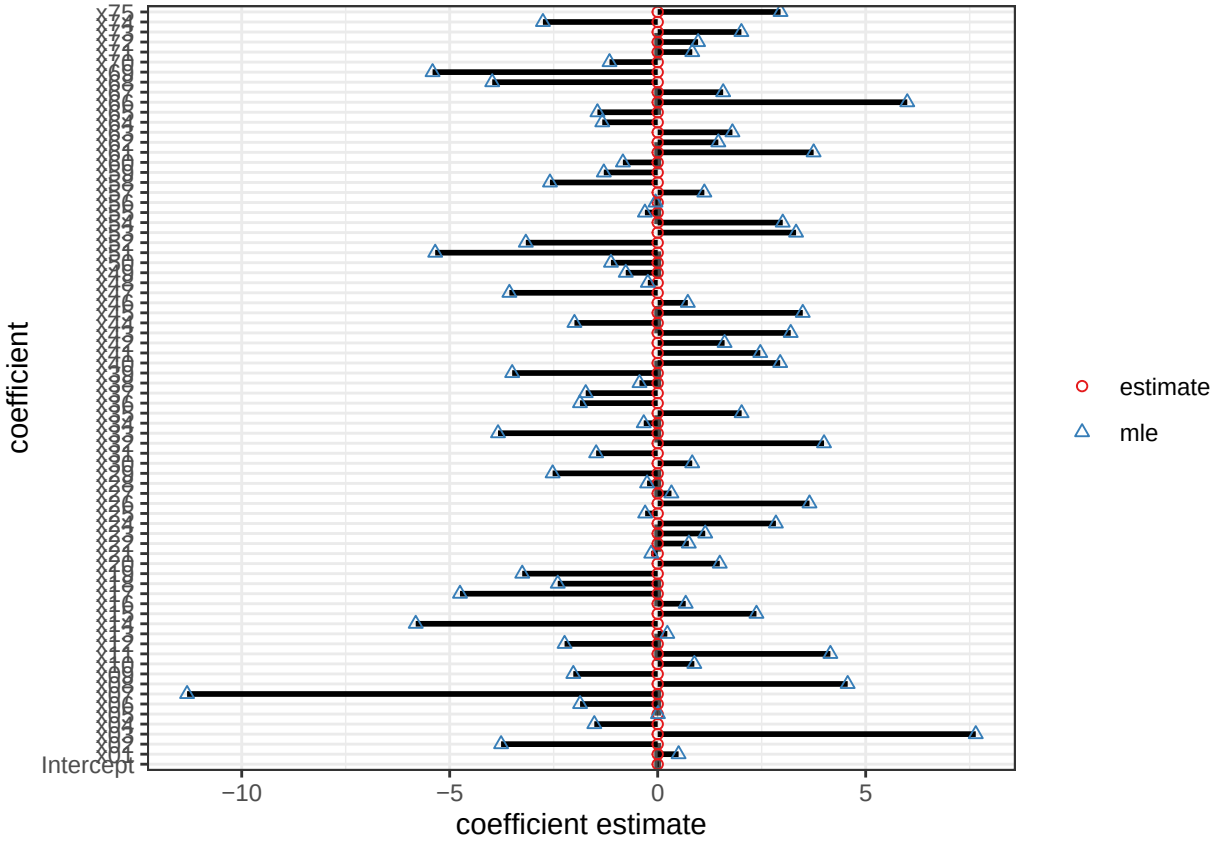
How do the Lasso estimates compare to the MLEs?

```
check_lasso_fit_highpenalty <- fit_regularized_regression(10000, loss_lasso, reg_info)
viz_compare_to_mles(check_lasso_fit_highpenalty, modA)
```

SOLUTION

```
## Warning: Removed 1 rows containing non-finite values (`stat_summary()`).
```

```
## Warning: Removed 1 rows containing missing values (`geom_point()`).
```



As it can be seen in the plot above, the coefficient values are not close to each other. This is with a high penalty parameter, we expect the Lasso to shrink many of the coefficients towards zero, potentially setting some to exactly zero if the penalty is large enough.

3d)

Previously, you defined a sequence of λ values and assigned those values to the `lambda_grid` object. This sequence serves the role as a candidate **search grid**. We do not know the best λ to use so we try out many values! The code chunk below is completed for you. The `eval` flag is set to FALSE and so you must set `eval=TRUE` in the code chunk options. The Lasso regression model is fit for each λ value in the search grid and the coefficient estimates are returned within a dataframe (tibble). The `lasso_path_results` dataframe includes the `lambda` value, just as the `ridge_path_results` object included the `lambda` value previously.

Please note: The code chunk below may take a few minutes to complete.

```
lasso_path_results <- purrr::map_dfr(lambda_grid,
                                     fit_regularized_regression,
                                     loss_func = loss_lasso,
                                     my_info = reg_info)
```

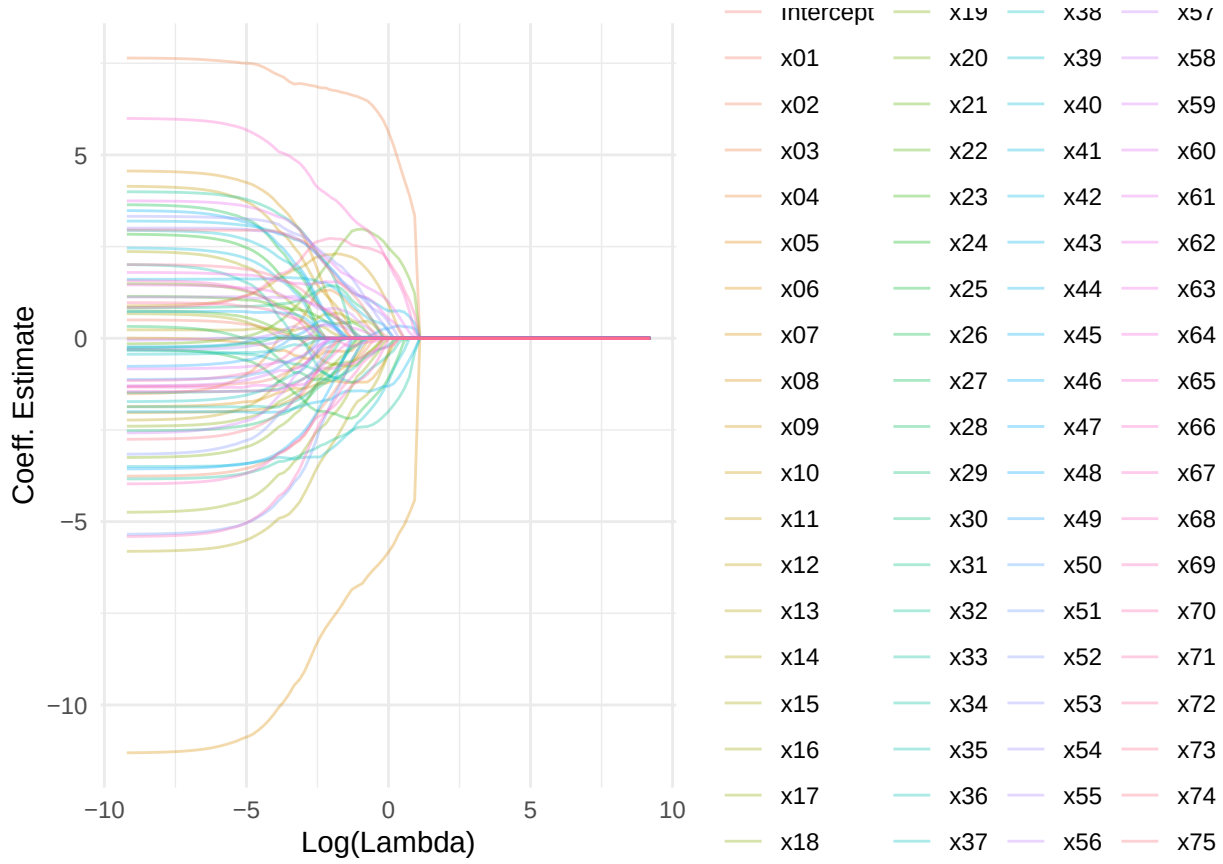
You will use the `lasso_path_results` to visualize the Lasso **coefficient path**. As with the Ridge coefficient path from earlier in the assignment, the Intercept is removed for you to be consistent with the `glmnet` path visualizations.

Complete the code chunk below by piping the dataframe to `ggplot()`. Map the `log(lambda)` to the `x` aesthetic and the `estimate` to the `y` aesthetic. Include a `geom_line()` layer and map the `term` variable to the `group` aesthetic within `geom_line()`. Set the `alpha` to 0.33.

How does the figure below compare to the figure created in Problem 2h)? How are the Lasso

paths different from the Ridge paths?

```
lasso_path_results %>%
  filter(term != '(Intercept)') %>%
  ggplot(aes(x = log(lambda), y = estimate, group=term)) +
  geom_line(aes(color = term), alpha = 0.33) +
  labs(x = "Log(Lambda)", y = "Coeff. Estimate") +
  theme_minimal()
```



SOLUTION

1) The Lasso paths typically show coefficients going to exactly zero for some value of log(lambda), due to the nature of L1 regularization which can shrink coefficients to zero.

2) Lasso paths to show increases and decreases in coefficient values, however Ridge regression coefficients tend to monotonically approach zero.

3) Lasso paths have sharper angles compared to Ridge paths, which tend to be smoother. This is because L1 penalty affects the coefficients in a piecewise linear fashion, while L2 penalty (Ridge) affects the coefficients quadratically.

Problem 04

Now that you have studied the behavior of the Ridge and Lasso penalties in greater depth, it's time to tune the regularization strength. You will tune the regularization strength just for the Lasso model. The steps are the same for the Ridge model, but we will focus on Lasso in this question.

The code chunk below reads in another data set for you. This data set is a random test split from the same data the training set was created from. You will thus use this data set as the hold-out test set to assess or evaluate the Lasso model for each regularization strength in the search grid.

```
dfA_test <- readr::read_csv('hw09_probA_test.csv', col_names = TRUE)
```

```
## Rows: 24 Columns: 76
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (76): x01, x02, x03, x04, x05, x06, x07, x08, x09, x10, x11, x12, x13, x...
##
## 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 is provided below to show the variable names are the same as those in `dfA_train`.

```
dfA_test %>% glimpse()
```

```
## Rows: 24
## Columns: 76
## $ x01 <dbl> 1.14313808, 0.01816419, 0.49591261, -1.49232633, -0.62059649, -0.0~
## $ x02 <dbl> -0.6464637, -0.4662545, -0.7916580, -0.7770189, 1.7378213, -0.4956~
## $ x03 <dbl> -1.89562312, -1.83545340, 0.58783552, -0.56476710, -0.15024320, 0.~
## $ x04 <dbl> -1.815775806, 0.281280533, -0.106889840, 1.066147596, 0.484055779,~
## $ x05 <dbl> -0.7128627, 1.4877917, -1.3365942, 2.0733856, -1.3475529, 0.331148~
## $ x06 <dbl> 1.49499607, -0.58088658, -2.78972454, -0.57166274, 0.03457907, 1.7~
## $ x07 <dbl> -1.17828630, -0.07111160, -0.64334020, -0.24136133, 0.55998293, 0.~
## $ x08 <dbl> -0.319216142, -0.314247113, -0.006775336, 0.472946659, -0.99712812~
## $ x09 <dbl> -0.92278608, -0.78961832, 0.16124184, -0.37300982, 0.31941510, -0.~
## $ x10 <dbl> 0.43104894, 0.06647602, -0.38361750, -0.98903565, -0.84237893, -0.~
## $ x11 <dbl> -0.75607015, -0.83118449, -1.30590588, 0.65576000, -0.32057164, 0.~
## $ x12 <dbl> -1.61445673, 2.23094618, 0.00335797, 0.63447707, -0.83379618, -2.1~
## $ x13 <dbl> -1.04732333, 0.94678115, -1.17904809, 0.01401656, 1.10230968, 2.84~
## $ x14 <dbl> 1.24779778, 2.21320896, -0.35418771, -2.29432424, 0.32813203, -0.3~
## $ x15 <dbl> 1.152395009, 0.871427100, -0.809043870, 1.068301785, -0.319201329,~
## $ x16 <dbl> -0.68759127, 0.22346479, 0.27785233, 0.63106951, -1.20893271, -0.9~
## $ x17 <dbl> -1.32498321, 0.99321252, 0.54829226, -1.98481617, -0.68417370, 2.1~
## $ x18 <dbl> 0.57914293, 1.04769622, -1.77207000, 0.82656793, 0.06793236, -0.26~
## $ x19 <dbl> -2.7324805, 0.8359603, 0.2268546, 0.1402093, -0.7464301, -0.554660~
## $ x20 <dbl> -0.13225186, -0.11301496, 1.45685079, 1.08894467, 0.52618785, 0.48~
## $ x21 <dbl> -0.05511845, 0.67435934, -1.23584780, -0.52775922, -2.39672927, -0~
## $ x22 <dbl> 0.376887793, 1.143383468, 0.347587585, 0.366396727, 1.204298906, 1~
## $ x23 <dbl> -0.02315765, -0.17996268, -0.06702020, -0.96745663, 1.05621951, 0.~
## $ x24 <dbl> -0.79364619, -0.75141178, -1.44564412, -0.26952336, -0.07881057, -~
## $ x25 <dbl> -1.1762732, 0.2858181, -1.9007820, 1.7190158, -2.1020462, 0.521933~
## $ x26 <dbl> 0.03440645, -0.16521310, 1.37251771, 0.15838025, 0.36806501, 0.481~
## $ x27 <dbl> 0.3555787, 0.7234963, -0.0119092, 0.5693598, 0.6075264, 0.9278386,~
## $ x28 <dbl> 0.46563934, 0.63759021, 0.69499926, -1.22618311, 0.45442219, 0.330~
## $ x29 <dbl> -0.30017643, -1.03137258, -0.76110963, 1.01467089, 1.30270574, -0.~
## $ x30 <dbl> 0.45063846, -0.27610195, 0.02663231, 0.23689472, -0.49330443, 0.29~
## $ x31 <dbl> 0.37249149, 0.99151848, -1.45214015, -1.20993730, 0.95817968, 0.47~
## $ x32 <dbl> 0.160626608, 0.298804786, 1.392771898, 2.397480316, -0.350553982, ~
## $ x33 <dbl> 0.6959056, -0.5876913, 1.0865048, -0.2956193, 1.1790686, 0.2194538~
## $ x34 <dbl> -1.286318824, -0.573059103, 0.709131820, 0.603993658, -0.009135101~
## $ x35 <dbl> -0.029824439, 0.120974022, -0.048754275, 1.998250597, -0.285683304~
## $ x36 <dbl> 1.74619464, -1.15938972, 0.52561374, -1.41721149, 0.45313713, -0.5~
## $ x37 <dbl> -1.6102714, 1.2100254, -0.8188211, 0.4097319, 0.3564670, 1.4065122~
## $ x38 <dbl> -0.52626336, -0.13884224, -0.97867868, 1.97088576, 0.99121364, 0.9~
## $ x39 <dbl> 0.47424567, 0.16380496, 0.09808709, -0.15455438, 0.38395056, 0.215~
```

```
## $ x40 <dbl> 1.04905686, -1.12755696, 0.81950481, -0.62597926, 0.18891435, 0.27~
## $ x41 <dbl> 0.9802753, 1.0727297, -0.1034688, -0.7738019, 1.4521328, 0.5824250~
## $ x42 <dbl> 0.8026953, 0.3912334, 0.2311717, 0.4416862, 0.6058927, 1.3619968, ~
## $ x43 <dbl> 1.497014705, 0.199461083, 0.877037014, 1.275111836, -1.463546089, ~
## $ x44 <dbl> 0.88361734, -0.43124514, 0.76909353, 0.34181375, 0.84916573, 0.527~
## $ x45 <dbl> -0.08048870, -0.87731052, -1.23497116, 1.45209109, 0.17710336, -0.~
## $ x46 <dbl> -0.22229459, 1.39743346, 1.46384916, 0.03334550, 0.73187037, 0.790~
## $ x47 <dbl> -1.7476152825, -0.2879971901, 1.7775635629, -0.8658681201, -0.1762~
## $ x48 <dbl> 1.07512535, -1.42892804, 1.14830168, -1.15034090, 1.14859354, -0.8~
## $ x49 <dbl> 0.11297505, -0.33298760, 2.62739250, -0.06144636, -0.62081207, -0.~
## $ x50 <dbl> -1.760714e+00, 1.546895e-01, -1.336308e+00, -5.341918e-01, -1.3781~
## $ x51 <dbl> 0.355802760, 1.261605721, -1.007393498, -0.229887771, -0.494385689~
## $ x52 <dbl> 1.12843542, 0.18742887, -0.15268669, -1.69652901, 1.00424500, -0.3~
## $ x53 <dbl> 0.2612449, 0.4780664, -0.2110005, -0.5120048, -1.3017326, 0.713045~
## $ x54 <dbl> -0.38075761, 0.84135339, 0.19037086, -0.80044132, -0.02384702, 0.1~
## $ x55 <dbl> 2.491695857, 0.463015446, 0.592801784, -0.513619781, 1.192908697, ~
## $ x56 <dbl> 0.6072242, -0.6014201, -0.8298660, 1.6731066, -1.1881461, 0.800793~
## $ x57 <dbl> 0.562299802, 1.513023999, -0.919529890, -0.637432106, -1.079789149~
## $ x58 <dbl> -1.90164605, 1.32513196, -1.32907538, -0.06284810, 0.57858234, -0.~
## $ x59 <dbl> -0.678017763, 0.671531331, 0.373627202, -1.360163880, -0.798847302~
## $ x60 <dbl> -1.01817726, 0.05251363, -0.29394473, 0.89162065, 0.84220139, -0.7~
## $ x61 <dbl> -0.309767250, -1.295571666, 0.476181835, 0.987615256, -0.421788553~
## $ x62 <dbl> 0.94130874, 1.49237811, 1.48027176, 1.49906080, -1.18024281, 0.061~
## $ x63 <dbl> 1.117138820, 1.783084869, -1.758350099, 0.321090650, 0.156595164, ~
## $ x64 <dbl> 0.01336110, -2.61963800, 0.23684126, -1.75412865, 1.56489203, -1.1~
## $ x65 <dbl> 1.16813558, 1.27909451, -0.23801311, -1.55946227, 0.47303683, 0.84~
## $ x66 <dbl> -0.68179048, -0.62455405, -1.04955883, 0.08481691, 0.13387617, 2.8~
## $ x67 <dbl> -0.107278267, 0.014839010, 0.584834981, -2.277171121, 2.358414125,~
## $ x68 <dbl> -0.34835071, -0.01205553, 0.15787720, -1.72791685, -0.23807589, -0~
## $ x69 <dbl> 0.891925590, 1.195514481, 0.339961334, -0.246669111, -1.672261892,~
## $ x70 <dbl> 0.81289142, 0.74375772, -0.62796234, -2.20482715, -1.21972263, 1.4~
## $ x71 <dbl> 2.27572777, 1.19452221, -0.40839006, -0.60256595, -0.83067145, 1.3~
## $ x72 <dbl> -0.69807238, 0.31327920, -0.15541045, -2.06316223, -1.24371309, 0.~
## $ x73 <dbl> 0.17105096, -0.24375675, -1.56184416, -0.09647939, 0.38517769, 0.8~
## $ x74 <dbl> 0.40569893, -0.26919348, -1.22765740, -0.78351810, -0.51345731, -1~
## $ x75 <dbl> -0.15896868, -0.73370485, -0.40814114, 0.41869379, 0.52360174, 0.6~
## $ y   <dbl> -19.9840444, -21.5075154, 11.6157884, -4.8924499, -17.1137287, 11.~
```

You will execute fitting the Lasso model on the training set and then testing the Lasso model on the hold-out test set. You will "score" the model by calculating the hold-out test MSE for each λ value in the search grid. You already fit the Lasso model on the training set, `dfA_train`, for each value in `lambda_grid`. However, you will create a function, `train_and_test_regularized()`, which executes both the training and testing actions. Thus, your function, `train_and_test_regularized()`, will do everything required to evaluate the model performance. Your function is therefore analogous to the `cv.glmnet()` function, the `caret::train()` function, and the training and testing functions within `tidymodels`!

4a)

The `train_and_test_regularized()` function is started for you in the code chunk below. It consists of 4 arguments. The first, `lambda_use`, is the user specified regularization strength. The second, `train_data`, is the training data set. The third, `test_data`, is the hold-out test data set. The last argument, `loss_func`, is the loss function used to fit the model. Notice that you do not provide the lists of required information to `train_and_test_regularized()`. Instead, the data are provided and `train_and_test_regularized()` defines the list of required information!

It is important to note that the we need to standardize the training data before fitting regularized models. The testing data must then be standardized **based on** the training data. The `dfA_test` data provided to you has already been appropriately pre-processed and so you do not need to worry about that here. This lets you focus on training and assessing the model performance.

Complete the code chunk below to define the `train_and_test_regularized()` function. The comments and variable names state the actions you should perform within each line of code.

Remember that we are working with linear additive features for all inputs.

HINT: Remember that you defined a function to allow fitting the model for a given λ value previously...

```
train_and_test_regularized <- function(lambda_use, train_data, test_data, loss_func)
{
  # make design matrix on TRAINING set
  Xtrain <- model.matrix(~ ., data = train_data)

  # make list of required information for the TRAINING set
  info_train <- list(
    yobs = train_data$y,
    design_matrix = Xtrain
  )

  # train the model
  train_results <- fit_regularized_regression(lambda_use, loss_func, info_train)

  # extract the training set coefficient estimates (completed for you)
  beta_estimates <- train_results %>% pull(estimate)

  # make the design matrix on the TEST set
  Xtest <- model.matrix(~ ., data = test_data)

  # predict the trend on the TEST data
  mu_test <- Xtest %*% beta_estimates

  # calculate the MSE on the TEST set
  MSE_test <- mean((test_data$y - mu_test)^2)

  # book keeping (completed for you)
  list(lambda = lambda_use,
       MSE_test = MSE_test)
}
```

SOLUTION

4b)

The code chunk below is completed for you. It applies the `train_and_test_regularized()` function to each λ value in the `lambda_grid` search grid. The `loss_lasso` function is assigned to the `loss_func` argument and thus the Lasso model is trained and tested for each regularization strength candidate value. The `eval` flag is set to FALSE and so you must set `eval=TRUE` in the code chunk options.

Please note: The code chunk below may take a few minutes to complete.

```
train_test_lasso_results <- purrr::map_dfr(lambda_grid,
                                           train_and_test_regularized,
```

```
                                                train_data = dfA_train,
                                                test_data = dfA_test,
                                                loss_func = loss_lasso)
```

A glimpse of the train/test assessment results is shown to the screen below. The result is a dataframe (tibble) with two columns. The `lambda` column is the regularization strength and `MSE_test` is the MSE on the test set.

```
train_test_lasso_results %>% glimpse()
```

```
## Rows: 101
## Columns: 2
## $ lambda   <dbl> 0.0001000000, 0.0001202264, 0.0001445440, 0.0001737801, 0.000~
## $ MSE_test <dbl> 1.114354e-09, 1.450289e-09, 1.571757e-09, 1.957315e-09, 2.248~
```

You must visualize the test set performance by plotting the hold-out test MSE with respect to the log of the regularization strength.

Create a line plot in `ggplot2` to visualize the hold-out test MSE vs the log of the regularization strength.

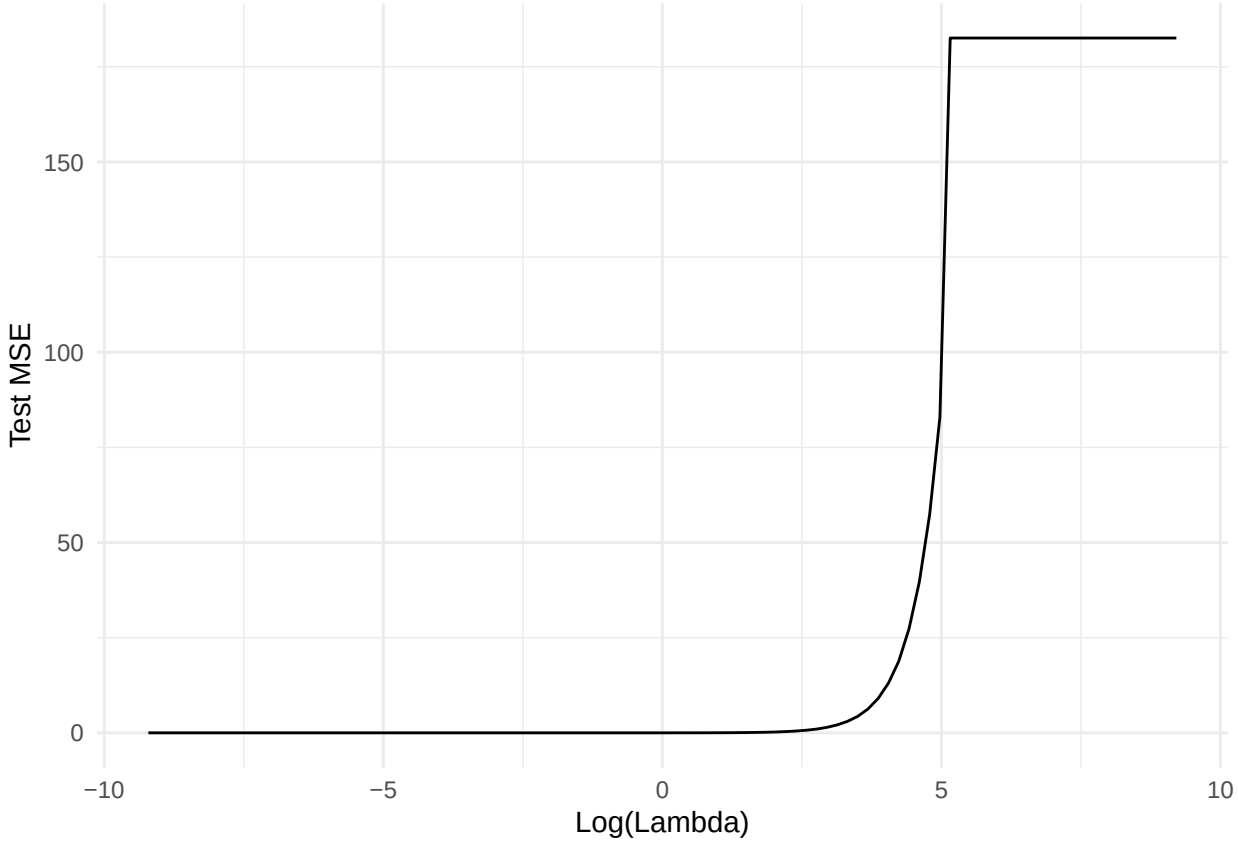
Which λ value yields the best test set performance?

SOLUTION Add your code chunks here.

Which λ is the best?

As we can see in the plot below, the lowest value of lambda, which is lambda = 0.0001, gives the best performance, because it corresponds to the lowest MSE value.

```
ggplot(train_test_lasso_results, aes(x = log(lambda), y = MSE_test)) +
  geom_line() +
  labs(x = "Log(Lambda)", y = "Test MSE") +
  theme_minimal()
```



4c)

You must now compare the Lasso estimates associated with the best λ value to the MLEs.

HINT: Remember that you already calculated the Lasso estimates for all λ values within the `lasso_path_results` object. You just need to filter that object for the appropriate `lambda` value to identify the best Lasso estimates.

```
best_lambda <- train_test_lasso_results %>%
  filter(MSE_test == min(MSE_test)) %>%
  pull(lambda)

best_lasso_estimates <- lasso_path_results %>%
  filter(lambda == best_lambda)

best_lasso_estimates
```

SOLUTION

```
## # A tibble: 76 x 3
##    term       estimate   lambda
##    <chr>         <dbl>    <dbl>
##  1 Intercept  -1.31    0.000100
##  2 x01         0.500   0.000100
##  3 x02        -3.76    0.000100
##  4 x03         7.64    0.000100
##  5 x04        -1.51    0.000100
```

```
##  6 x05         0.00292 0.000100
##  7 x06        -1.86    0.000100
##  8 x07       -11.3     0.000100
##  9 x08         4.56    0.000100
## 10 x09        -2.03    0.000100
## # i 66 more rows
```

4d)

You just used a train/test split to validate and tune a machine learning model! However, single train/test splits do not allow us to estimate the **reliability** of the performance. We are not able to state how **confident** we are in the finding the best λ value in this case. Resampling methods, such as cross-validation, provide more robust and stable results compared to single train/test splits.

However, you will not execute the Resampling manually. You will use the `cv.glmnet()` function from the `glmnet` library to manage cross-validation and "scoring" for you.

The `glmnet` library is loaded for you in the code chunk below.

```
library(glmnet)
```

```
## Loading required package: Matrix
```

```
##
## Attaching package: 'Matrix'
```

```
## The following objects are masked from 'package:tidyr':
##
##     expand, pack, unpack
```

```
## Loaded glmnet 4.1-8
```

The `dfA_train` and `dfA_test` splits were randomly created by splitting a larger data set with an 80/20 train/test split. The code chunk below binds the rows together, recreating the original data set. This allows `cv.glmnet()` to manage splitting the original 120 observations rather than the smaller randomly created training set.

```
dfA <- dfA_train %>% bind_rows(dfA_test)
```

Let's start out by fitting the Lasso model directly with `glmnet` before executing the cross-validation. This way we can compare our previous Lasso path to the `glmnet` path!

You need to create the design matrix consistent with the `glmnet` requirements before fitting the `glmnet` model. You must therefore create a new design matrix for the linear additive features for all inputs but **REMOVE** the intercept column from the design matrix.

Complete the code chunk below by defining the `glmnet` required design matrix, extracting the response vector, and then fitting the Lasso model with `glmnet`.

Assign the `lambda_grid` object you created previously to the `lambda` argument within `glmnet()`. You are therefore using a custom `lambda` grid rather than the default set of `lambda` values.

```
Xenet <- model.matrix(y ~ . - 1, data = dfA)

yenet <- dfA$y

lasso_glmnet <- glmnet(x = Xenet, y = yenet, alpha = 1, lambda = lambda_grid)
```

SOLUTION

4e)

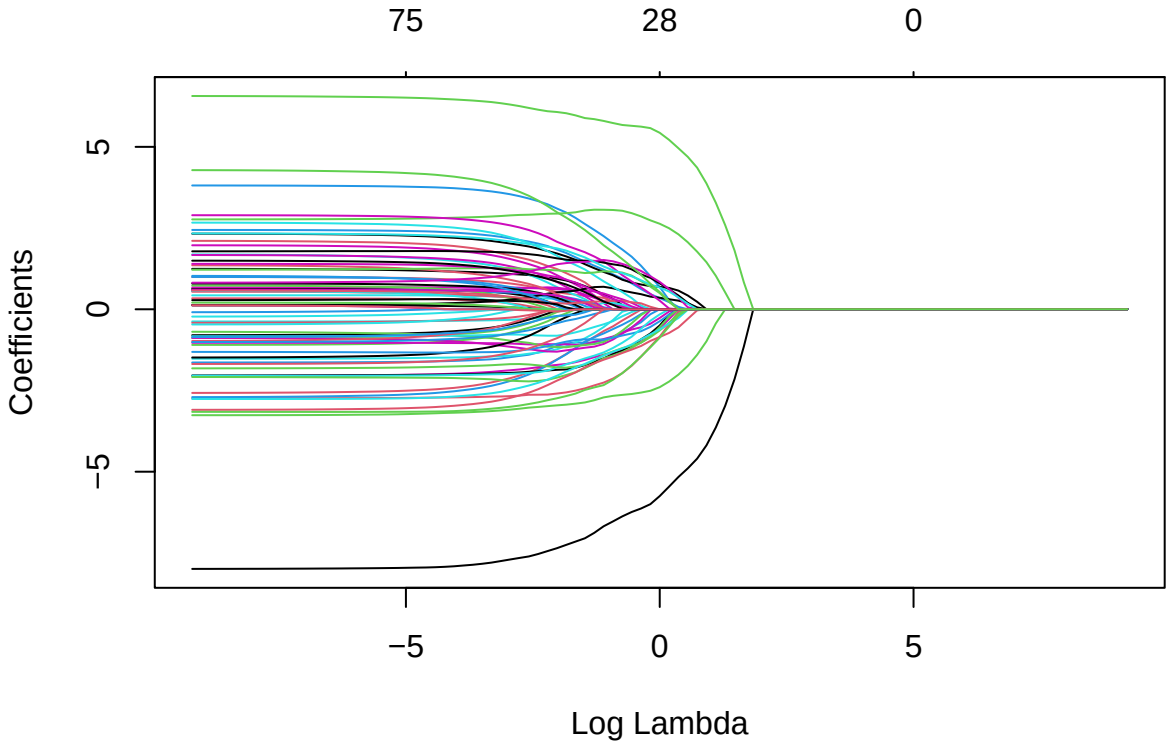
Use the `glmnet` default plot method to visualize the coefficient path for the `glmnet` trained Lasso model. You must assign the `xvar` argument in the plot function call to `'lambda'` to show the path consistent with your previous figures.

How does the `glmnet` created coefficient path compare to Lasso coefficient path you created in Problem 3d)?

SOLUTION How do they compare?

As it can be seen in the figure below, the `glmnet` created coefficient path and Lasso coefficient path we created in Problem 3d) look similar.

```
plot(lasso_glmnet, xvar = 'lambda')
```



4f)

Now it's time to tune the Lasso model with cross-validation. You will specifically use 5 fold cross-validation and you must use the same λ search grid defined previously within `lambda_grid`.

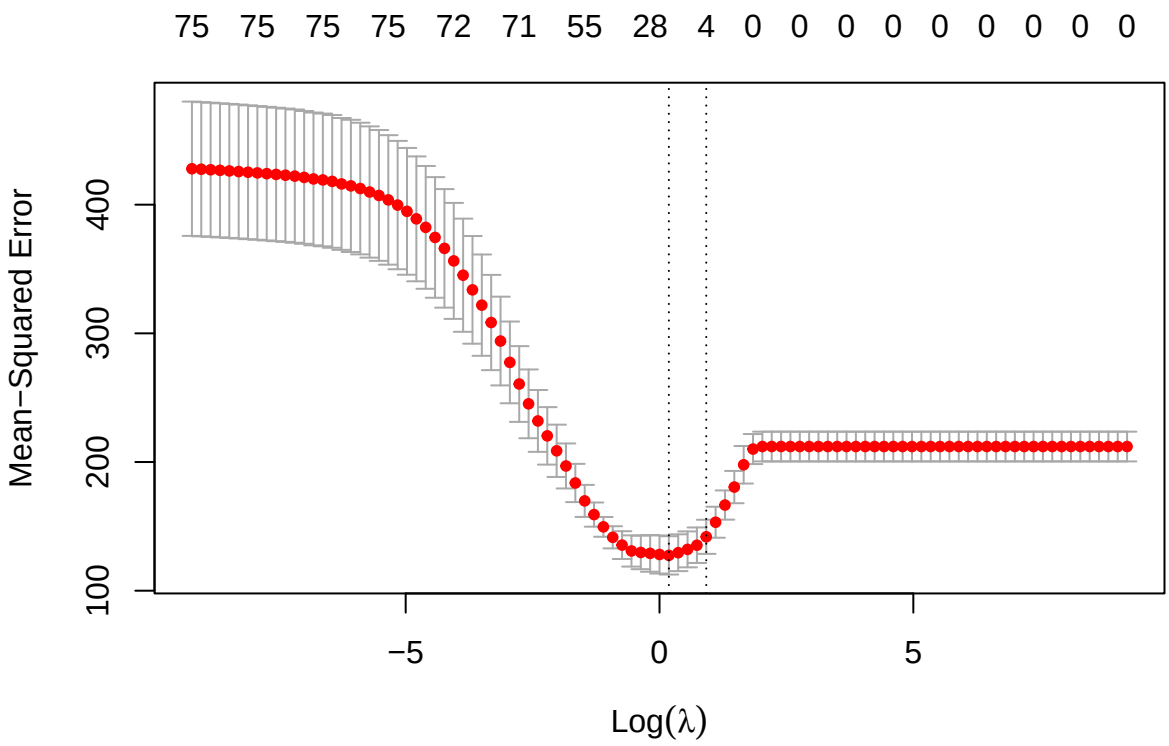
Use the `cv.glmnet()` function to tune the Lasso model with 5-fold cross-validation and the defined search grid. Assign the `lambda_grid` object to the `lambda` argument within `cv.glmnet()`. Assign the result to the `lasso_cv` object.

Plot the cross-validation results to the screen.

How many non-zero features does the best Lasso model have? How many non-zero features does the Lasso model recommended by the 1-standard error rule have?

NOTE: The random seed is set for you below to force the splits to be the same every time you run the code chunk.

```
set.seed(71231)

lasso_cv <- cv.glmnet(x = Xenet, y = yenet, alpha = 1, lambda = lambda_grid, nfolds = 5)

plot(lasso_cv)
```



SOLUTION

```
best_lambda <- lasso_cv$lambda.min
best_model <- glmnet(x = Xenet, y = yenet, alpha=1, lambda = best_lambda)
non_zero_best_model <- sum(coef(best_model) !=0)

lambda_1se <- lasso_cv$lambda.1se
model_1se <- glmnet(x = Xenet, y = yenet, alpha=1, lambda = lambda_1se)
non_zero_model_1se <- sum(coef(model_1se) !=0)

non_zero_best_model
```

```
## [1] 25
```

```
non_zero_model_1se
```

```
## [1] 5
```

As the code chunks indicate above the best Lasso model has 25 non-zero features and the Lasso model recommended by the 1-standard error rule has 5 non-zero features.

4g)

Which inputs matter based on the tuned Lasso model?

SOLUTION The inputs opposite nonzero values on the table below matter based on the tuned Lasso model.

```r
best_lasso_coefficients <- coef(best_model, s = best_lambda)
best_lasso_coefficients
```

```
## 76 x 1 sparse Matrix of class "dgCMatrix"
##                       s1
## (Intercept) -0.007610626
## x01          .
## x02          .
## x03          5.216255153
## x04          0.220576924
## x05          .
## x06         -0.300722520
## x07         -5.459712455
## x08          .
## x09          .
## x10          0.284642374
## x11          .
## x12          .
## x13          .
## x14         -0.461052734
## x15          .
## x16          .
## x17          .
## x18         -0.067750493
## x19          .
## x20          .
## x21          2.495063947
## x22          .
## x23          .
## x24          .
## x25         -0.166337884
## x26          .
## x27          .
## x28          .
## x29          .
## x30          .
## x31          .
## x32          .
## x33         -2.225553862
## x34         -0.163943925
## x35          .
## x36          .
## x37          .
## x38          .
## x39         -0.225285163
## x40          .
## x41          .
## x42          .
## x43          0.656595355
## x44         -0.728169427
## x45          .
## x46          .
## x47          .
## x48          .
```

```
## x49          0.308812614
## x50          .
## x51         -0.414909821
## x52          .
## x53          .
## x54          0.023000903
## x55          .
## x56          .
## x57          0.227933389
## x58          .
## x59          .
## x60          .
## x61          0.615281240
## x62          .
## x63          .
## x64          .
## x65         -0.442463783
## x66          0.622592021
## x67          .
## x68          .
## x69         -0.530860210
## x70          .
## x71          0.547002751
## x72          .
## x73          .
## x74          .
## x75          0.256918325
```

Problem 05

By default `glmnet` trains LASSO penalized models. You can could repeat the exercise for RIDGE penalized models to identify if the RIDGE penalty outperforms the LASSO penalty. However, you do **not** need to do that! The ELASTIC NET penalty **blends** or **mixes** the RIDGE and LASSO penalties! ELASTIC NET accomplishes this by including an additional tuning parameter, the **mixing fraction**, which controls the blending between the two. a MIXING FRACTION of 1 corresponds to LASSO while a MIXING FRACTION of 0 corresponds to RIDGE. The `glmnet` package refers to the MIXING FRACTION as the `alpha` parameter. Tuning the ELASTIC NET penalty therefore tunes the MIXING FRACTION, `alpha`, **and** the regularization strength, `lambda`. Tuning the ELASTIC NET penalty therefore selects whether you should use one penalty **or** if you need a mixture of the two!

Tuning ELASTIC NET is rather tedious with `glmnet` directly. You will therefore use the `caret` package to manage the training, validation, and tuning of the ELASTIC NET model.

The `caret` library is loaded for you in the code chunk below.

```
library(caret)
```

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

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

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

5a)

You worked with `caret` in earlier assignments. You will use it again but now you are using resampling to **tune** two parameters, `alpha` and `lambda`, that dictate **how** the regression coefficients are estimated. Just as in earlier assignments, you must begin by specifying the resampling scheme or "how the training is *controlled*" by `caret`.

Specify the resampling scheme to be 5 fold with 5 repeats. Assign the result of the trainControl() function to the `my_ctrl_A` object. Specify the primary performance metric to be 'RMSE' and assign that to the `my_metric_A` object.

```
my_ctrl_A <- trainControl(method = "repeatedcv",
                          number = 5,
                          repeats = 5,
                          search = "random")

my_metric_A <- "RMSE"
```

SOLUTION

5b)

The ELASTIC NET model consists of 2 tuning parameters. By default. `caret` specifies a GRID or candidate set of tuning parameters to try. You will use this DEFAULT grid to gain experience TUNING the ELASTIC NET model. Programming examples provided in Canvas demonstrate how to create your own custom tuning grids if you are interested.

You must train, assess, and tune an ELASTIC NET model using the default `caret` tuning grid. In the `caret::train()` function you must use the formula interface to specify a model with linear additive features derived from all inputs to predict the response `y`. You must use the entire data set, `dfA`. Assign the method argument to `'glmnet'` and set the metric argument to `my_metric`. You **MUST** also instruct `caret` to standardize the features by setting the `preProcess` argument equal to `c('center', 'scale')`. Assign the `trControl` argument to the `my_ctrl` object.

IMPORTANT: The `caret::train()` function works with the formula interface. Even though you are using `glmnet` to fit the model, `caret` does not require you to organize the design matrix as required by `glmnet`! Thus, you do **NOT** need to remove the intercept when defining the formula to `caret::train()`!

Train, assess, and tune the `glmnet` elastic net model consisting of linear additive features using all inputs with the defined resampling scheme. Assign the result to the `enet_A` object and display the results to the screen.

SOLUTION The random seed is set for you for reproducibility.

```
set.seed(1234)

enet_A <- caret::train(
  y ~ .,
  data = dfA,
  method = "glmnet",
  metric = my_metric_A,
  preProcess = c("center", "scale"),
  trControl = my_ctrl_A
)

print(enet_A)
```

```
## glmnet
##
## 120 samples
##  75 predictor
##
## Pre-processing: centered (75), scaled (75)
## Resampling: Cross-Validated (5 fold, repeated 5 times)
## Summary of sample sizes: 96, 96, 96, 96, 96, 96, ...
## Resampling results across tuning parameters:
##
##   alpha      lambda       RMSE      Rsquared   MAE
##   0.6092747  0.312957329  12.66073  0.3391784  10.25090
##   0.6233794  0.001063802  17.89483  0.2448207  14.36988
##   0.8609154  0.007938910  17.46009  0.2509654  14.02991
##
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were alpha = 0.6092747 and lambda
##  = 0.3129573.
```

5c)

Which tuning parameter combinations are considered to be the best?

Is the best set of tuning parameters more consistent with Lasso or Ridge regression?

SOLUTION As we see on the table above the best set of tuning parameters are alpha = 0.6092747 and lambda = 0.3129573. Since the alpha value is close to 1, then it suggests that the model is more consistent with Lasso regression.

5d)

Display the coefficients to the screen for the tuned ELASTIC NET model. Which coefficients are non-zero?

SOLUTION The inputs opposite nonzero values on the table below are the non-zero coefficients.

```
best_lambda <- enet_A$bestTune$lambda
coef_enet <- coef(enet_A$finalModel, s = best_lambda)

coef_enet
```

```
## 76 x 1 sparse Matrix of class "dgCMatrix"
##                         s1
## (Intercept)  0.1657353750
## x01          .
## x02         -1.1840089095
## x03          5.3460075480
## x04          1.4815177803
## x05         -0.1673429088
## x06         -1.7154373609
## x07         -6.2683694211
## x08          0.8735715703
## x09          .
## x10          2.6811603009
## x11          0.2811980381
## x12          0.6666126637
```

```
## x13          0.8695482442
## x14         -2.2940399304
## x15          .
## x16         -0.9516058966
## x17         -0.9641450918
## x18         -1.0450777597
## x19          .
## x20          0.4567586450
## x21          3.1702146829
## x22         -1.2343576412
## x23          0.3998348881
## x24         -1.1154952104
## x25         -1.8573788185
## x26          .
## x27         -1.1003123693
## x28          0.1933558658
## x29         -0.6959879257
## x30          0.5610639560
## x31         -0.1145385619
## x32          .
## x33         -2.8405309109
## x34         -0.9144342072
## x35         -1.5997557261
## x36          .
## x37          .
## x38          .
## x39         -1.7370165921
## x40          0.3241106246
## x41          .
## x42          0.6145820299
## x43          1.6213726663
## x44         -2.0276852614
## x45          .
## x46          0.0654482641
## x47          .
## x48          0.5005709435
## x49          0.6388622649
## x50          0.2607799505
## x51         -1.6070759919
## x52          .
## x53          1.3684714918
## x54          1.8304716518
## x55          0.0965921544
## x56          .
## x57          1.0863627884
## x58         -0.1467038054
## x59          .
## x60          0.3058330347
## x61          1.5294422328
## x62          0.2322539292
## x63          .
## x64          .
## x65         -1.8645776726
## x66          1.4627485659
```

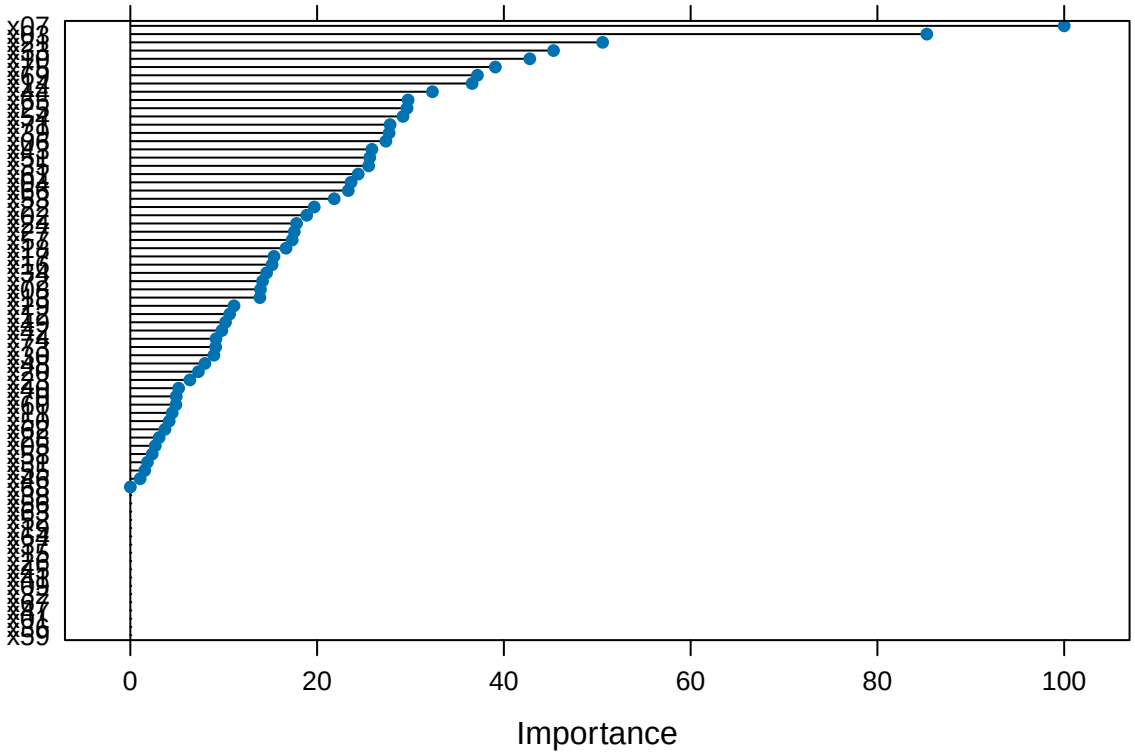
```
## x67           .
## x68          -0.0001539579
## x69          -2.3298845509
## x70          -0.3082105012
## x71           1.7434363471
## x72           0.8873188182
## x73           0.5727542755
## x74          -0.5751158001
## x75           2.4501885340
```

5e)

`caret` provides several useful helper functions for ranking the features based on their influence on the response. This is known as ranking the variable importances and the `varImp()` function will extract the variable importances from a model for you. Wrapping the `plot()` function around `varImp()` creates a figure showing the variable importances. By default, the displayed importance values are in a relative scale with 100 corresponding to the most important variable.

Plot the variable importances for the `caret` tuned ELASTIC NET model.

```
importances <- varImp(enet_A, scale=TRUE)

plot(importances)
```



SOLUTION

5f)

What is the MOST IMPORTANT input?

Are the variable importance rankings consistent with the coefficient magnitudes you printed to the screen in Problem 5d)?

SOLUTION What do you think?

As it can be seen in the plot in 5e) above, the most important input is "x07". Also, the variable importance rankings are consistent with the coefficient magnitudes we printed to the screen in Problem 5d). Foe examle in the table there the coefficient of "x07" is "-6.2683694211", which has the largest magnitude.

Problem 06

Correlated features cause problems for many models. One approach to try and identify if the features correlation matters is through tuning the ELASTIC NET model. The elastic net blends RIDGE and LASSO penalties and involves two tuning parameters as described previously. Tuning the mixing fraction will help us identify if the feature correlation impacts the model behavior.

The code chunk below reads in a data set that will let you practice working with correlated features.

```
dfB <- readr::read_csv('hw09_probB.csv', col_names = TRUE)
```

```
## Rows: 110 Columns: 10
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (10): x1, x2, x3, x4, x5, x6, x7, x8, x9, 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.
```

The glimpse provided below shows there are 9 continuous inputs and 1 continuous response.

```
dfB %>% glimpse()
```

```
## Rows: 110
## Columns: 10
## $ x1 <dbl> 1.381146373, 0.362491477, 1.381504700, -1.411010806, -0.806235745, ~
## $ x2 <dbl> -1.01410438, -0.04659940, -1.02810580, 1.18063292, 0.77387392, 1.99~
## $ x3 <dbl> 0.521999324, -0.628264234, 1.261490720, -1.977614406, -0.719286220,~
## $ x4 <dbl> -0.922953260, -0.244410231, -1.734594915, 1.350547617, 0.761285604,~
## $ x5 <dbl> 2.01713417, 0.20193624, 0.94821339, -0.73355318, -1.42037948, -1.34~
## $ x6 <dbl> -1.59855276, -0.40714750, -1.48595293, 0.53425585, 1.20138087, 1.10~
## $ x7 <dbl> 1.38336030, 1.28670587, 1.76647219, -1.78858187, -0.43824475, -2.04~
## $ x8 <dbl> -0.6015361, 0.4261699, -1.1451876, 0.5443889, 0.2917105, 0.7774429,~
## $ x9 <dbl> 1.32899009, 0.98205124, -0.03446554, -1.81200384, -0.72647487, -0.5~
## $ y  <dbl> 6.269098, 3.996943, 5.335141, -6.362414, -1.528389, -5.859906, 4.22~
```

The reshaped long-format data set is created for you in the code chunk below. The response `y` is not stacked and thus the long-format includes all inputs, `x1` through `x9` stacked together.

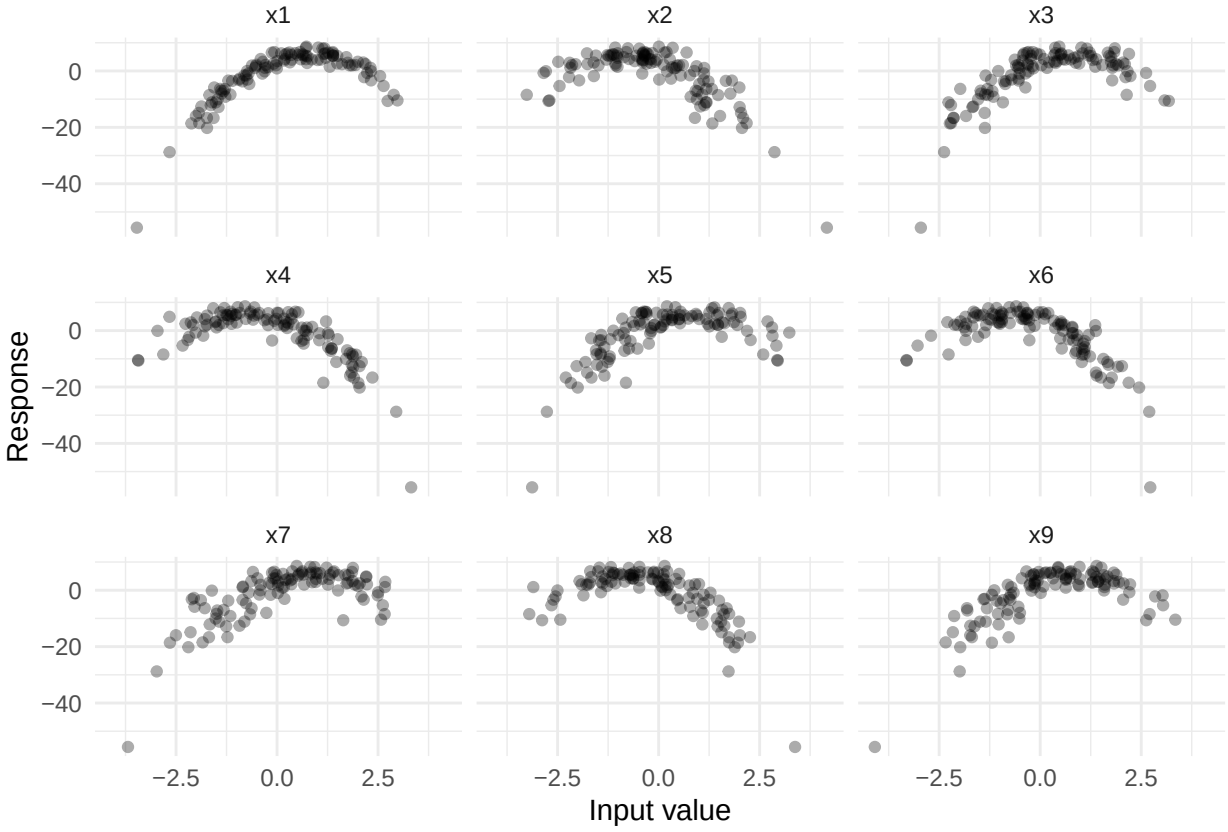
```
lfB <- dfB %>%
  tibble::rowid_to_column() %>%
  pivot_longer(starts_with('x'))

lfB %>% glimpse()
```

```
## Rows: 990
## Columns: 4
## $ rowid <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3~
## $ y     <dbl> 6.269098, 6.269098, 6.269098, 6.269098, 6.269098, 6.269098, 6.26~
## $ name  <chr> "x1", "x2", "x3", "x4", "x5", "x6", "x7", "x8", "x9", "x1", "x2"~
## $ value <dbl> 1.3811464, -1.0141044, 0.5219993, -0.9229533, 2.0171342, -1.5985~
```

6a)

Create a scatter plot to show the relationship between the output and each input. Use the reshaped long-format data set to create facets for each input.

```
ggplot(lfB, aes(x = value, y = y)) +
  geom_point(alpha = 0.33) +
  facet_wrap(~ name) +
  labs(x = 'Input value', y = 'Response') +
  theme_minimal()
```



SOLUTION

6b)

You visualized the output to input relationships, but you must also examine the relationship between the inputs! The `cor()` function can be used to calculate the correlation matrix between all columns in a data frame.

Correlation matrices are created from wide-format data and so you will use the original wide-format data, `dfB`, for this question instead of the long-format data, `lfB`.

Pipe `dfB` into `select()` and select all columns except the response `y`. Pipe the result to the `cor()` function and display the correlation matrix to screen.

```
dfB %>%
  select(-y) %>%
  cor()
```

SOLUTION

```
##            x1         x2         x3         x4         x5         x6         x7
## x1  1.0000000 -0.9389955  0.9536235 -0.9547008  0.9470124 -0.9493356  0.9396088
## x2 -0.9389955  1.0000000 -0.8735053  0.8811235 -0.8848975  0.8846558 -0.8773211
## x3  0.9536235 -0.8735053  1.0000000 -0.9097993  0.9138529 -0.9209883  0.8819940
## x4 -0.9547008  0.8811235 -0.9097993  1.0000000 -0.9073242  0.9280242 -0.8925269
## x5  0.9470124 -0.8848975  0.9138529 -0.9073242  1.0000000 -0.8882862  0.8853547
## x6 -0.9493356  0.8846558 -0.9209883  0.9280242 -0.8882862  1.0000000 -0.9012759
## x7  0.9396088 -0.8773211  0.8819940 -0.8925269  0.8853547 -0.9012759  1.0000000
## x8 -0.9448926  0.8953953 -0.9088091  0.9173552 -0.8966782  0.8982929 -0.8686375
## x9  0.9353346 -0.8938712  0.8810448 -0.8783161  0.8661697 -0.8949083  0.8719199
##            x8         x9
## x1 -0.9448926  0.9353346
## x2  0.8953953 -0.8938712
## x3 -0.9088091  0.8810448
## x4  0.9173552 -0.8783161
## x5 -0.8966782  0.8661697
## x6  0.8982929 -0.8949083
## x7 -0.8686375  0.8719199
## x8  1.0000000 -0.8800298
## x9 -0.8800298  1.0000000
```

6c)

Rather than displaying the numbers of the correlation matrix, let's create a correlation plot to visualize the correlation coefficient between each pair of inputs. The `corrplot` package provides the `corrplot()` function to easily create clean and simple correlation plots. You do not have to load the `corrplot` package, instead you will call the `corrplot()` function from `corrplot` using the `::` operator. Thus, you will call the function as `corrplot::corrplot()`.

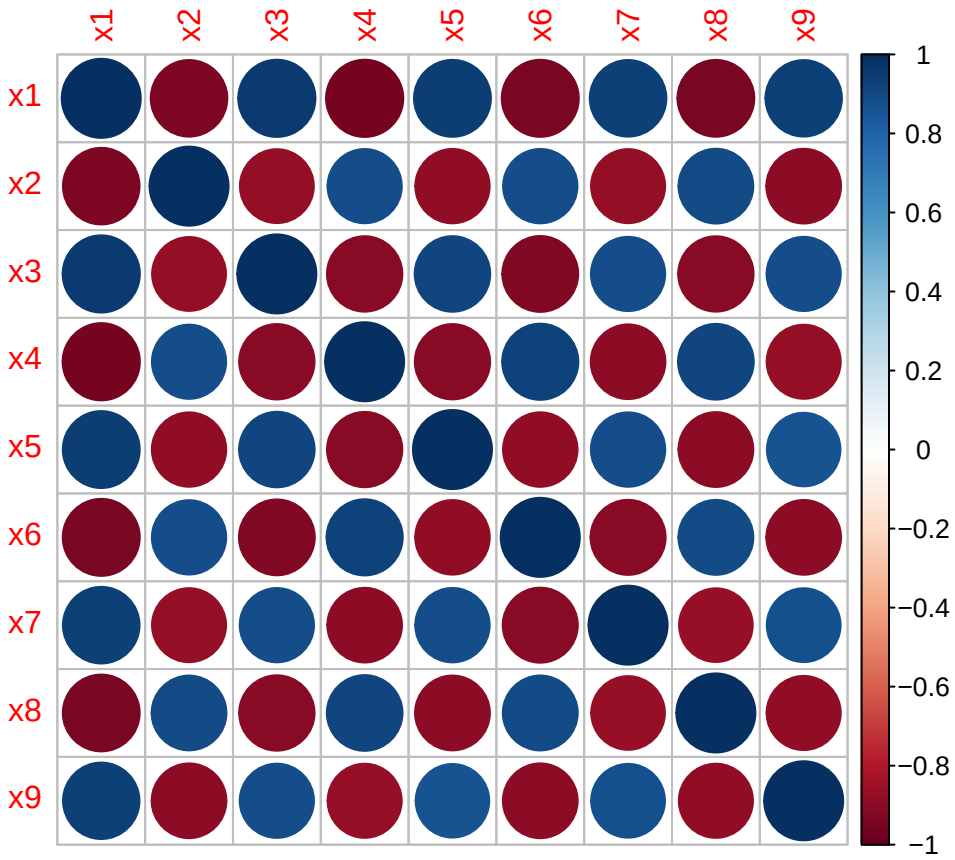
The first argument to `corrplot::corrplot()` is a correlation matrix. You must therefore calculate the correlation matrix associated with a data frame and pass that matrix into `corrplot::corrplot()`.

You will create three correlation plots. For the first, use the default input arguments to `corrplot::corrplot()`. For the second, assign the `type` argument to `'upper'` to visualize the correlation plot as an upper-triangular matrix. For the third, assign the `type` argument to `'upper'` and assign the `method` argument to `'color'`.

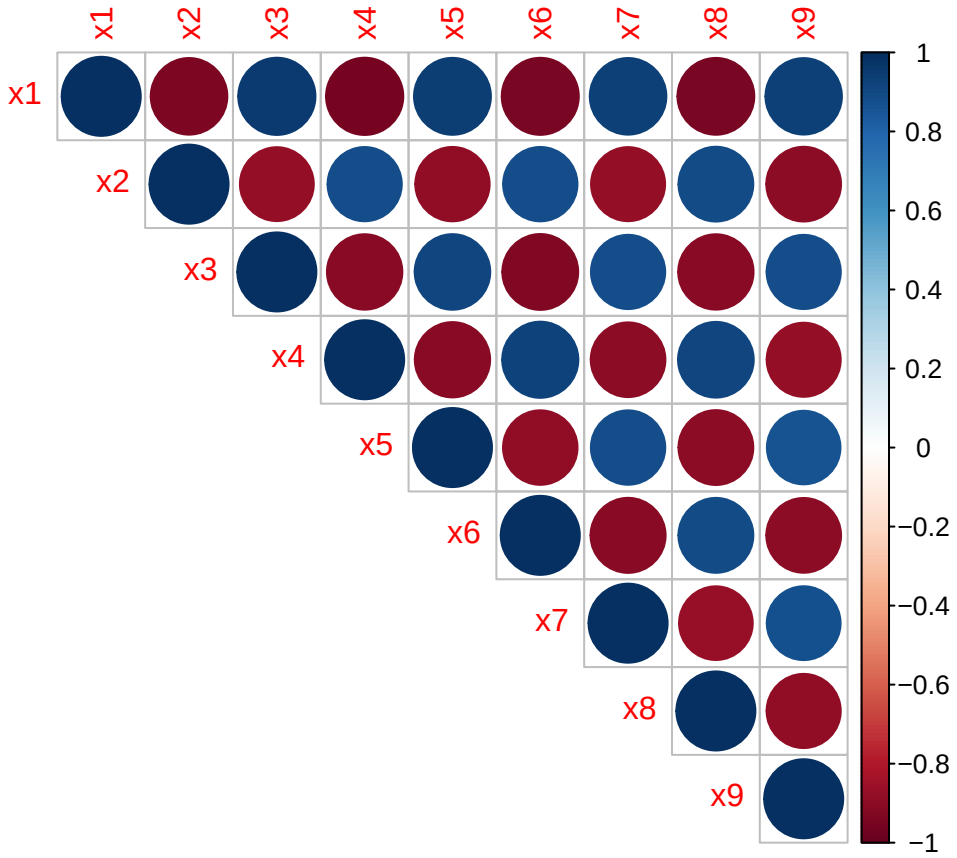
Based on your visualizations, are the inputs correlated?

SOLUTION Yes, based on the visualizations on the figures below, the inputs are highly correlated.

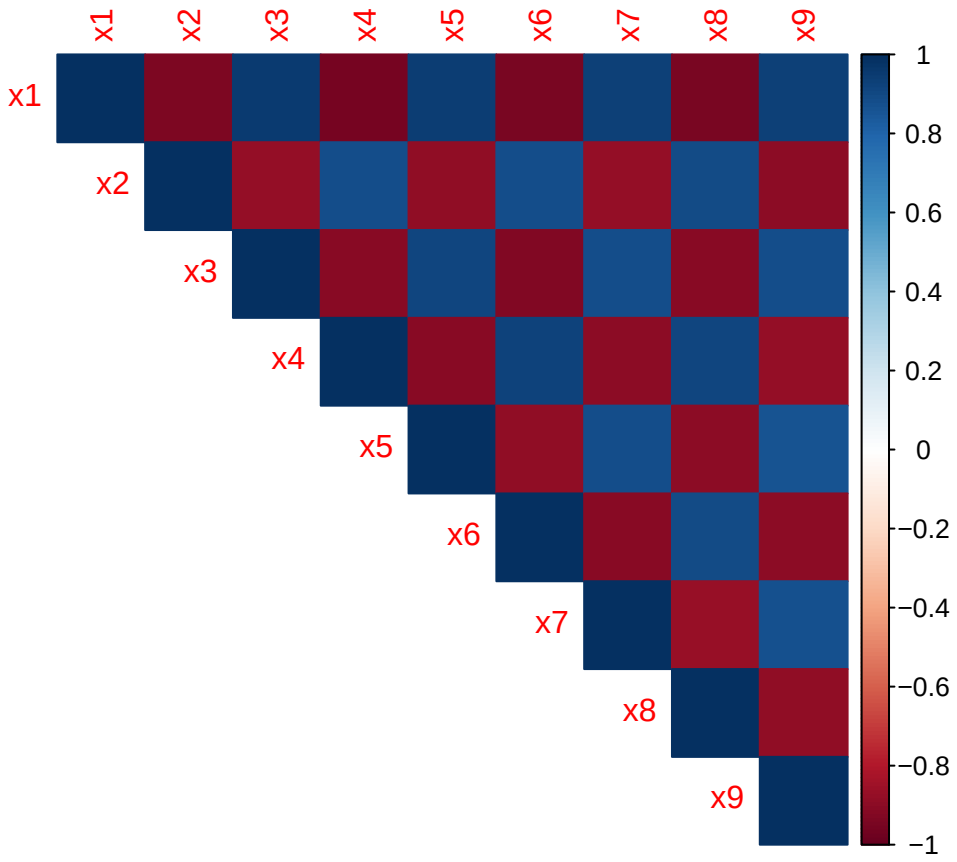
```
cor_matrix <- dfB %>%
  select(-y) %>%
  cor()

corrplot::corrplot(cor_matrix)
```




```
corrplot::corrplot(cor_matrix, type="upper")
```




```
corrplot::corrplot(cor_matrix, type="upper", method ="color")
```



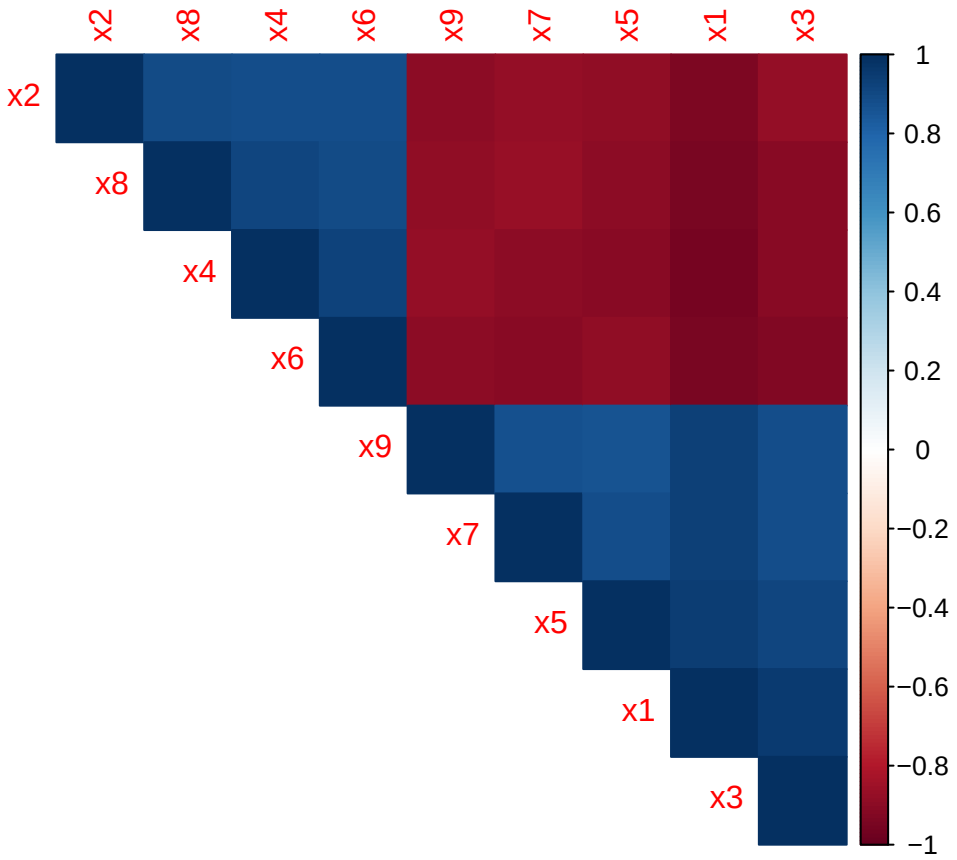
6d)

The `corrplot::corrplot()` function can reorder correlation plots to group correlated variables in CLUSTERS. This can help us "see" correlation structure easier compared to keeping the variables in their original order.

Create the third correlation plot from 6c) again. However, you must set the `order` argument to `'hclust'` and the `hclust.method` argument to `'ward.D2'`.

HINT: Remember that the correlation matrix must be calculated first before calling `corrplot::corrplot()`!

```
corrplot::corrplot(cor_matrix, type = "upper", method = "color", order = "hclust", hclust.method = "ward
```



SOLUTION

6e)

In lecture we discussed that we can easily create more features than inputs by considering interactions between the inputs! Thus, when exploring if feature correlation could impact model results, we should not simply examine the correlation between the inputs. We should also examine the correlation between the features derived from the inputs!

For this problem, let's work with all pair wise interactions between the inputs. You should therefore create a model that has all main effects and all pair wise products between the inputs.

Define the design matrix that includes all main effects and pair wise produces between the inputs associated with `dfB`. Assign the result to the `XBpairs` objects.

We are focused on exploring the relationship between the features and so you must remove the intercept column of ones from this design matrix.

How many columns are in the design matrix?

SOLUTION There are 45 colums in the design matrix as expressed in the code chunk below.

```
XBpairs <- model.matrix(y ~ (.)^2 -1, data = dfB)
ncol(XBpairs)
```

```
## [1] 45
```

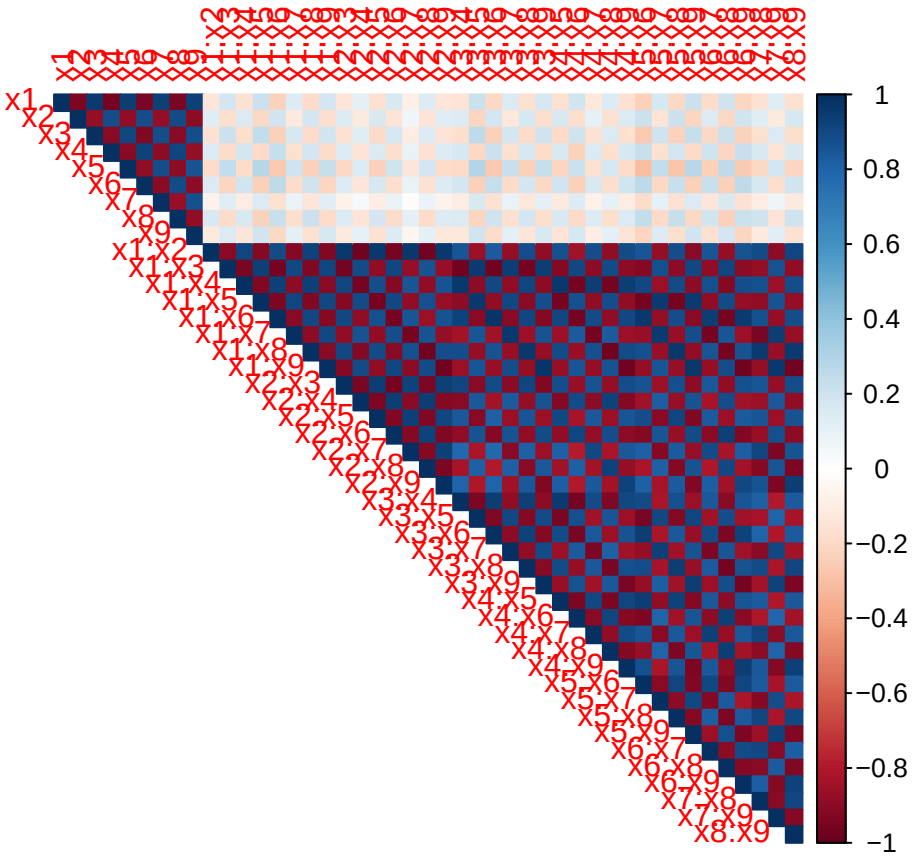
6f)

Create the correlation plot between all features in the `XBpairs` design matrix. Assign the `type` argument to `'upper'` and set the `method` argument to `'color'` in `corrplot::corrplot()`

to visualize the correlation coefficients as colored square "tiles" within an upper triangular matrix.

HINT: Remember that the correlation matrix must be calculated first before calling `corrplot::corrplot()`!

```
cor_matrix <- cor(XBpairs)

corrplot::corrplot(cor_matrix, type = "upper", method = "color")
```



SOLUTION

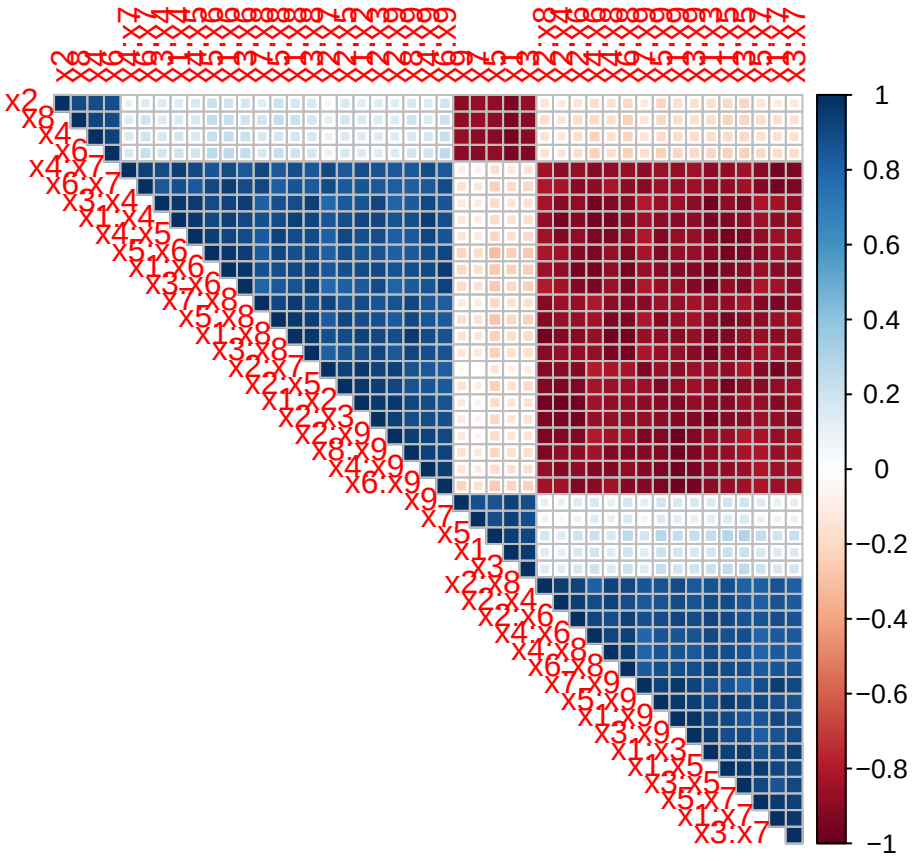
6g)

The `corrplot::corrplot()` function can reorder correlation plots to group correlated variables in clusters. This can help us "see" correlation structure easier compared to keeping the variables in their original order.

Create the correlation plot between all features in the `XBpairs` design matrix again. However, you should set the `order` argument to `'hclust'` and the `hclust.method` argument to `'ward.D2'`. Continue to set `method` to `'square'` and `type` to `'upper'`.

HINT: Remember that the correlation matrix must be calculated first before calling `corrplot::corrplot()`!

```
corrplot::corrplot(cor_matrix, order = "hclust", hclust.method = "ward.D2", method = "square", type = "
```



SOLUTION

6h)

Your goal is to find out if the feature correlation impacts model behavior. Thus, you will NOT train a LASSO or RIDGE model. Instead, you will train, assess, and tune ELASTIC NET which blends LASSO and RIDGE for you! You will now use `caret` to tune the mixing fraction and regularization strength of the elastic net model.

You must specify the resampling scheme that caret will use to train, assess, and tune a model.

Specify the resampling scheme to be 5 fold with 5 repeats. Assign the result of the `trainControl()` function to the `my_ctrl` object. Specify the primary performance metric to be `'RMSE'` and assign that to the `my_metric` object.

```
my_ctrl <- trainControl(method = "repeatedcv",
                        number = 5,
                        repeats = 5,
                        search = "random")
my_metric <- "RMSE"
```

SOLUTION

6i)

You must train, assess, and tune an elastic model using the default `caret` tuning grid. In the `caret::train()` function you must use the formula interface to specify a model with all pair wise interactions to predict the response `y`. Assign the method argument to `'glmnet'` and set the metric argument to `my_metric`. You **must**

also instruct `caret` to standardize the features by setting the `preProcess` argument equal to `c('center', 'scale')`. Assign the `trControl` argument to the `my_ctrl` object.

Important: The `caret::train()` function works with the formula interface. Thus, even though you are using `glmnet` to fit the model, `caret` does **NOT** require you to organize the design matrix as required by `glmnet`! Thus, you do **NOT** need to remove the intercept when defining the formula to `caret::train()`!

Train, assess, and tune the `glmnet` elastic net model consisting of main effects and all pair wise interactions with the defined resampling scheme. Assign the result to the `enet_default` object and display the result to the screen.

Which tuning parameter combinations are considered to be the best?

Is the best set of tuning parameters more consistent with Lasso or Ridge regression?

Does the feature correlation seem to be impacting model behavior in this application?

SOLUTION

1) As we see on the table below the best set of tuning parameters are alpha = 0.8609154 and lambda = 0.00793891.
2) Since the alpha value is close to 1, then it suggests that the model is more consistent with Lasso regression.
3) Yes, the feature correlation seem to be impacting model behavior in this application. It can be observed from the non-zero coefficients in the table in Problem 6j) below.

```
set.seed(1234)

enet_default <- caret::train(
  y ~ .^2,
  data = dfB,
  method = 'glmnet',
  metric = my_metric,
  preProcess = c('center', 'scale'),
  trControl = my_ctrl
)

print(enet_default)
```

```
## glmnet
##
## 110 samples
##   9 predictor
##
## Pre-processing: centered (45), scaled (45)
## Resampling: Cross-Validated (5 fold, repeated 5 times)
## Summary of sample sizes: 87, 88, 87, 89, 89, 88, ...
## Resampling results across tuning parameters:
##
##   alpha      lambda       RMSE      Rsquared   MAE
##   0.6092747  0.312957329  1.838162  0.9539104  1.466386
##   0.6233794  0.001063802  1.824601  0.9500836  1.513468
##   0.8609154  0.007938910  1.782623  0.9518993  1.485877
##
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were alpha = 0.8609154 and lambda
##  = 0.00793891.
```

6j)

Display the coefficients to the screen for the tuned elastic net model. Which coefficients are non-zero?

SOLUTION The inputs opposite nonzero values on the table below are the non-zero coefficients.

```
best_lambda2 <- enet_default$bestTune$lambda
coef_enet2 <- coef(enet_default$finalModel, s = best_lambda2)

coef_enet2
```

```
## 46 x 1 sparse Matrix of class "dgCMatrix"
##                          s1
## (Intercept) -1.2303635803
## x1           0.3428393923
## x2          -0.9294767119
## x3           0.7154656657
## x4          -0.7490925552
## x5           0.5134941442
## x6          -1.2760329303
## x7           0.9956078287
## x8          -1.1792934336
## x9           0.4176541029
## x1:x2        .
## x1:x3       -0.0005462067
## x1:x4        0.9035083157
## x1:x5       -0.0472261628
## x1:x6        0.0020348748
## x1:x7       -0.0011528836
## x1:x8        1.1161726588
## x1:x9       -0.2433732449
## x2:x3        1.1056414378
## x2:x4        .
## x2:x5        .
## x2:x6        .
## x2:x7        0.4665136113
## x2:x8       -0.1437844983
## x2:x9        .
## x3:x4        .
## x3:x5        .
## x3:x6        0.0181363878
## x3:x7        .
## x3:x8        0.5273270087
## x3:x9        .
## x4:x5        .
## x4:x6       -1.3957406220
## x4:x7        .
## x4:x8        .
## x4:x9        .
## x5:x6        .
## x5:x7        .
## x5:x8        .
## x5:x9        .
## x6:x7        .
```

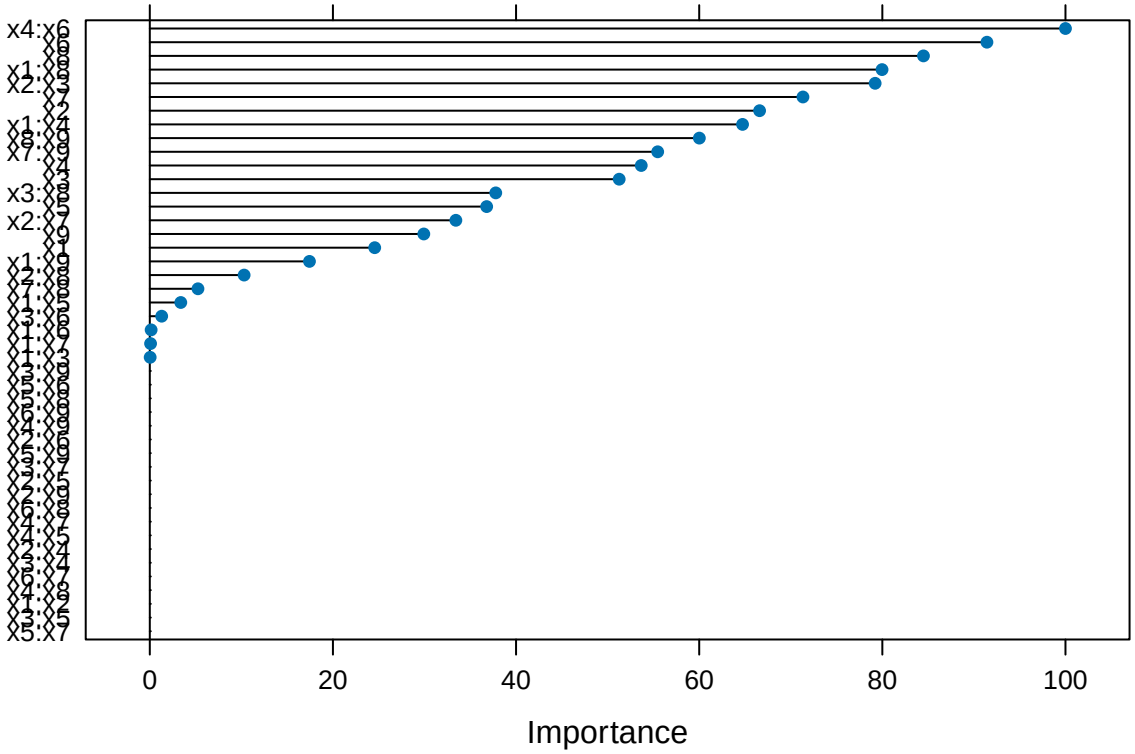
```
## x6:x8         .
## x6:x9         .
## x7:x8         0.0734115941
## x7:x9        -0.7741828805
## x8:x9         0.8376833514
```

6k)

`caret` provides several useful helper functions for ranking the features based on their influence on the response. This is known as ranking the variable importances and the `varImp()` function will extract the variable importances from a model for you. Wrapping the `varImp()` result with `plot()` will create a figure showing the variable importances. By default, the displayed importance values are in a relative scale with 100 corresponding to the most important variable.

Plot the variable importances for the `caret` tuned elastic net model. Are the rankings consistent with the magnitude of the coefficients you printed to the screen in Problem 6j)?

```
importances2 <- varImp(enet_default, scale=TRUE)

plot(importances2)
```



SOLUTION

As it can be seen in the plot above, the most important input is "x4:x6". Also, the rankings are consistent with the coefficient magnitudes we printed to the screen in Problem 6j). For examle in the table there the coefficient of "x4:x6" is "-1.3957406220", which has the largest magnitude.

6l)

You ranked the feature importance and identified the most important input. However, what must you keep in mind when INTERPRETING the feature rankings?

SOLUTION

1) When predictors are correlated, their coefficients in the model may not accurately reflect their individual importance.
2) Variable importance does not reflect non-linear relationships or interactions unless specifically modeled.
3) Feature importance is model-dependent. Different models can give different importance rankings. Thus, conclusions about feature importance are conditional on the chosen model.

Problem 07

We have focused heavily on working with continuous inputs this semester. However, linear models may also use **categorical inputs** as features. This problem introduces working with categorical inputs by fitting non-Bayesian linear models with `lm()`.

The data you will work with is loaded for you in the code chunk below.

```
dfC <- readr::read_csv('hw09_probC.csv', col_names = TRUE)
```

```
## Rows: 75 Columns: 3
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (1): v
## 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.
```

The glimpse shows there are two inputs, `x` and `v`, and one continuous response `y`. The `v` input is non-numeric.

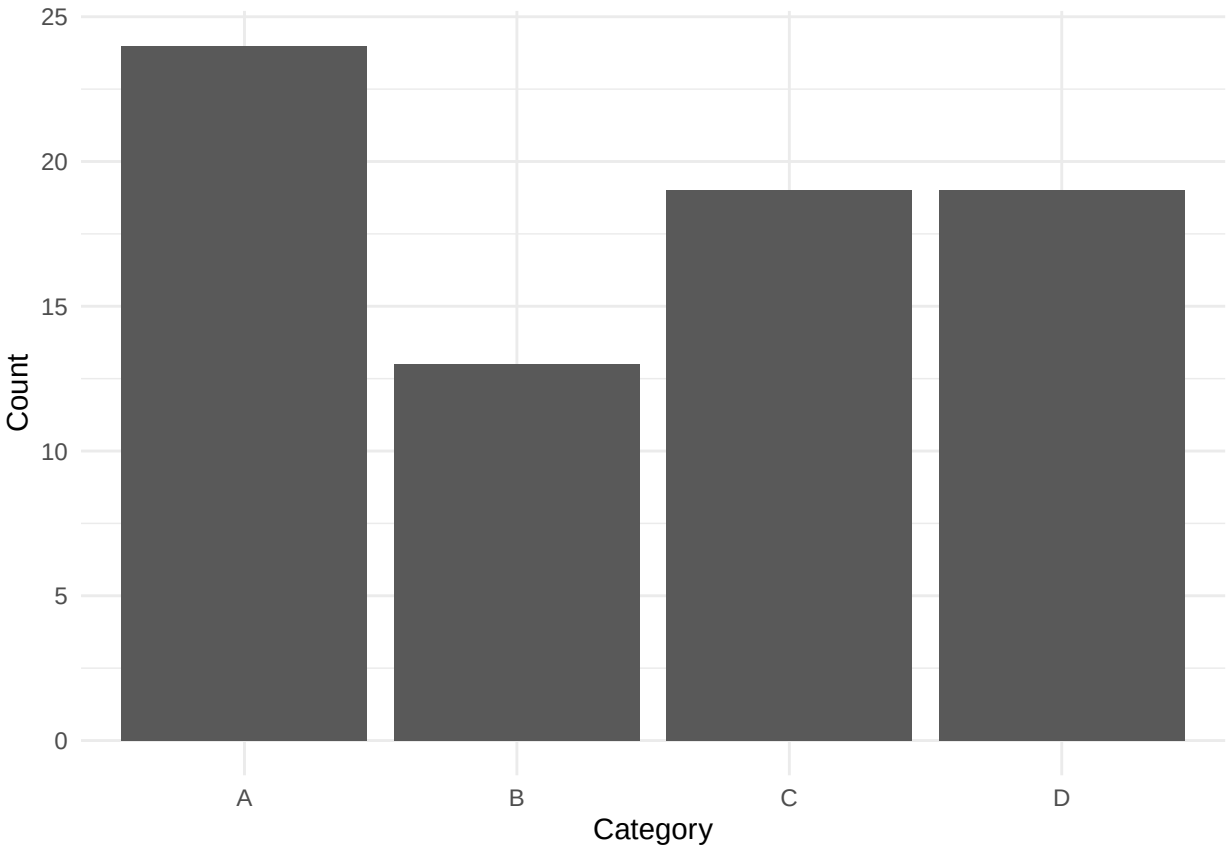
```
dfC %>% glimpse()
```

```
## Rows: 75
## Columns: 3
## $ x <dbl> -1.65522340, -0.40027559, -0.80346451, 0.76686259, 0.93237497, 0.010~
## $ v <chr> "A", "A", "A", "B", "D", "D", "D", "A", "C", "D", "C", "A", "B", "A"~
## $ y <dbl> -1.0995699, -2.0743885, -2.2994641, 1.5491779, -0.1327855, -0.381488~
```

7a)

Create a bar chart in `ggplot2` to show the counts associated with each unique value of the categorical variable `v`.

```
ggplot(dfC, aes(x = v)) +
  geom_bar() +
  labs(x = "Category", y = "Count") +
  theme_minimal()
```



SOLUTION

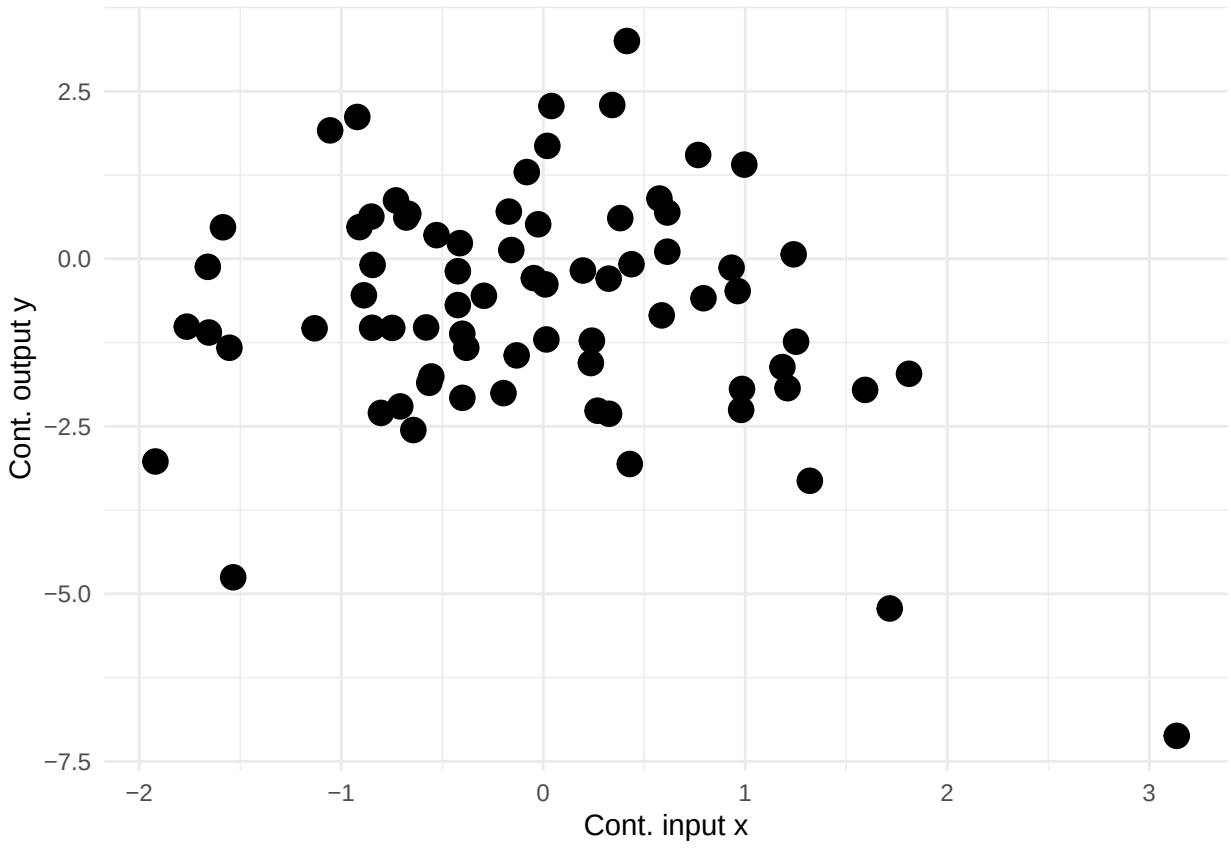
7b)

Next, let's focus on the relationship between the continuous output `y` and the continuous input `x`.

Create a scatter plot in `ggplot2` to the show relationship between `y` and `x`.

Manually assign the marker `size` to 4.

```
ggplot(dfC, aes(x = x, y = y)) +
  geom_point(size = 4) +
  labs(x = "Cont. input x", y = "Cont. output y") +
  theme_minimal()
```



SOLUTION

7c)

It is important to consider if the output to input relationship depends on the categorical variable When working with **mixed** categorical and continuous inputs. A simple way to explore such an effect is by coloring the markers in a scatter plot by the categorical variable.

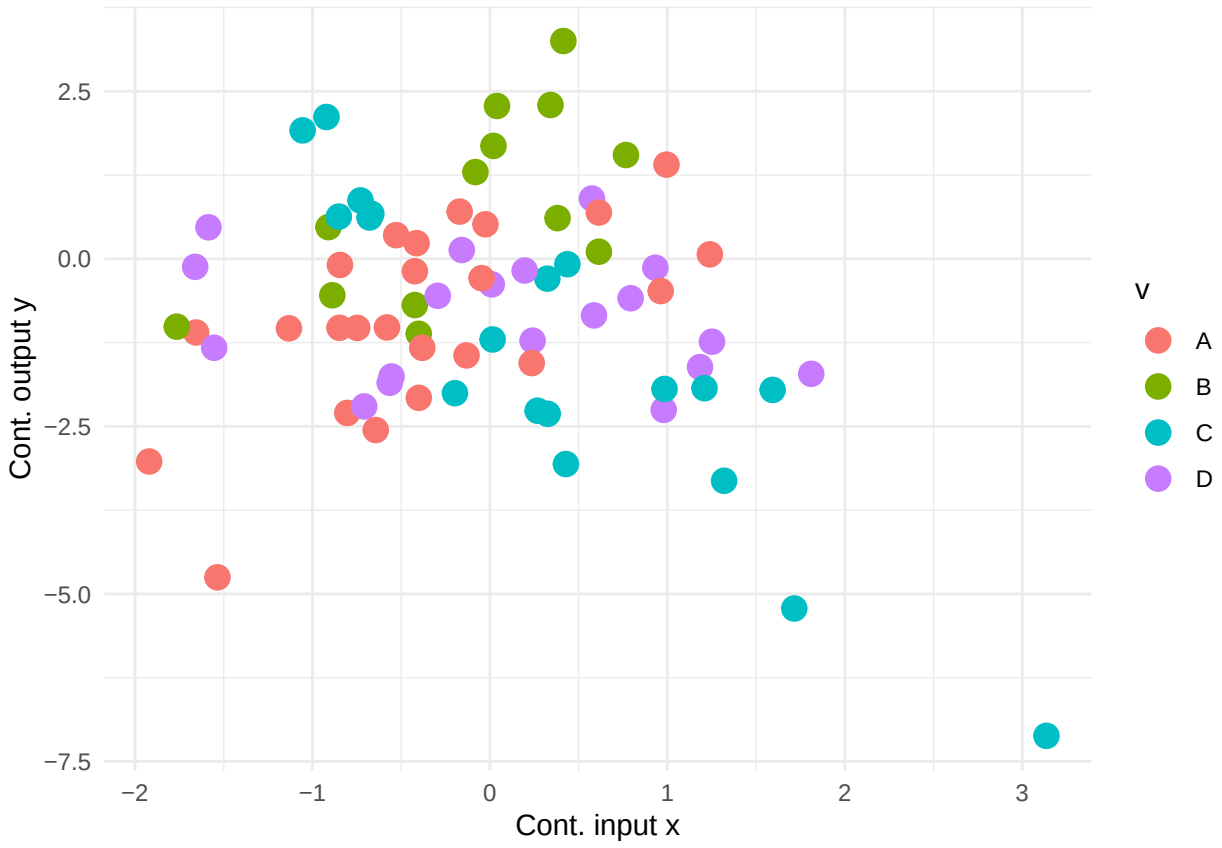
Repeat the same scatter plot between `y` and `x` that you created in 7b) but this time map the `color` aesthetic to the `v` categorical variable.

Does the output to input relationship appear to vary across the levels (categories) of `v`?

HINT: Remember the importance of the the `aes()` function!

SOLUTION Yes, the output to input relationship appear to vary across the levels (categories) of `v` as it can be seen on the plot below.

```
ggplot(dfC, aes(x = x, y = y, color = v)) +
  geom_point(size = 4) +
  labs(x = "Cont. input x", y = "Cont. output y") +
  theme_minimal()
```



7d)

Sometimes we need to include **smoothers** in our plots to help guide our interpretation of the trends. The `geom_smooth()` function allows adding a **smoothing** layer on top of a scatter plot. You used this function earlier in the semester and will use it now to help your exploration.

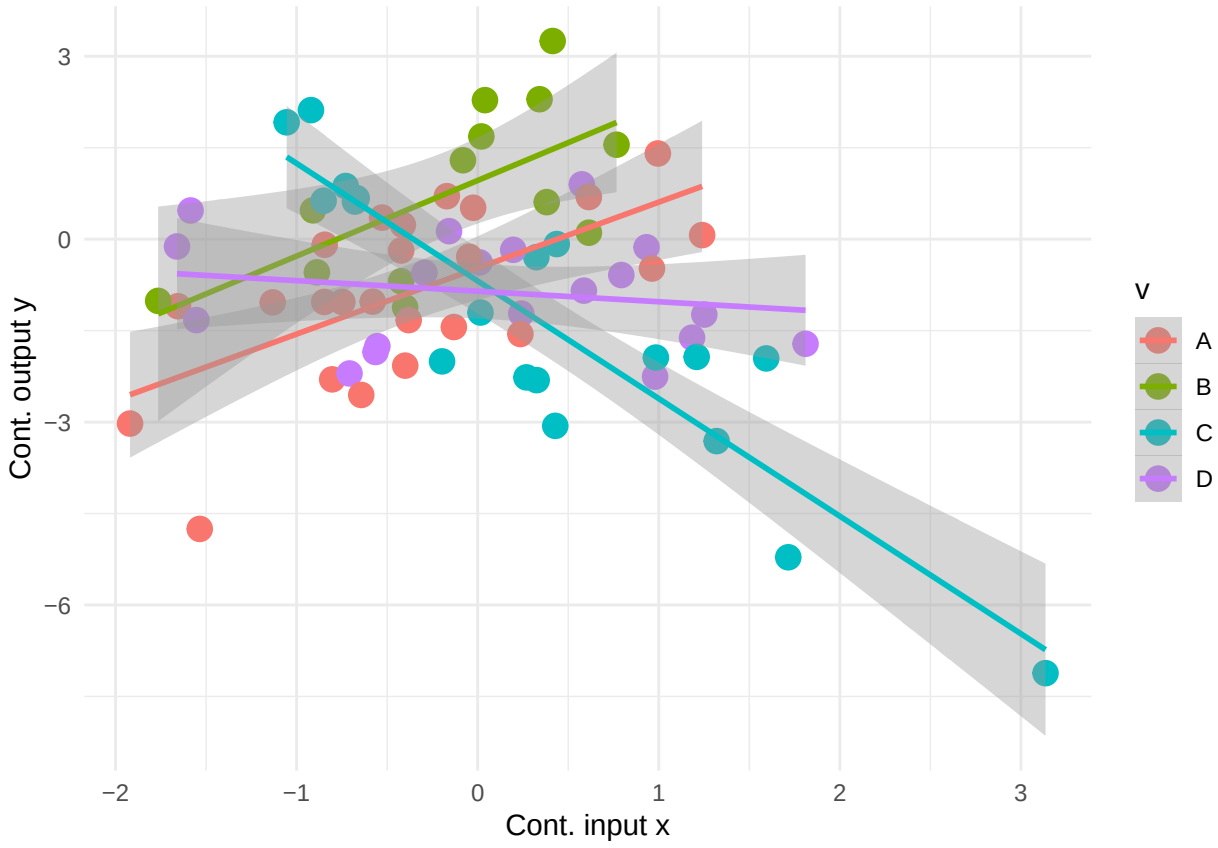
Repeat the same scatter plot between `y` and `x` that you created in 7c). You should thus continue to map `color` to the `v` variable. This time however, add in a `geom_smooth()` layer. Assign the `formula` argument to `y ~ x` and assign the `method` argument to `lm` (you do not need quotes). You must also map the `color` aesthetic to `v` within the `geom_smooth()` layer.

Does the output to input relationship appear to vary across the levels (categories) of `v`?

HINT: Remember the importance of the `aes()` function!

SOLUTION Yes, the output to input relationship appear to vary across the levels (categories) of `v` as it can be seen on the plot below.

```
ggplot(dfC, aes(x = x, y = y, color = v)) +
  geom_point(size = 4) +
  geom_smooth(aes(color = v), formula = y ~ x, method = lm) +
  labs(x = "Cont. input x", y = "Cont. output y") +
  theme_minimal()
```



7e)

Let's fit two simple models for this application. The first will use linear additive features. Thus, you should add the effect of the continuous input `x` to the categorical input `v`.

Fit a non-Bayesian linear model to predict the response `y` based on linear additive features between the two inputs. Assign the model object to the `modC_add` object and display the model `summary()` to the screen.

How many coefficients are estimated?

What does the Intercept correspond to?

SOLUTION

1) As it can be seen on the table below, 5 coefficients are estimated in total.

2) The intercept equals the average response associated with the v=A level!

```
modC_add <- lm(y ~ x + v, data = dfC)
summary(modC_add)
```

```
##
## Call:
## lm(formula = y ~ x + v, data = dfC)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.7640 -0.9464  0.0123  1.0740  3.0260
##
```

```
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -1.0249     0.3327  -3.081  0.00295 **
## x            -0.3562     0.2025  -1.759  0.08294 .
## vB            1.7558     0.5479   3.204  0.00204 **
## vC           -0.2124     0.5088  -0.417  0.67769
## vD            0.1857     0.4956   0.375  0.70897
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.585 on 70 degrees of freedom
## Multiple R-squared:  0.2075, Adjusted R-squared:  0.1622
## F-statistic: 4.583 on 4 and 70 DF,  p-value: 0.002408
```

7f)

Next, let's interact the categorical and continuous inputs!

Fit a non-Bayesian linear model to predict the response `y` based on interacting the categorical input `v` and the continuous input `x`. Your model should include all main effects and interaction features. Assign the model object to the `modC_int` object and display the model `summary()` to the screen.

How many coefficients are estimated?

What does the intercept correspond to?

SOLUTION

1) As it can be seen on the table below, 8 coefficients are estimated in total.

2) The intercept corresponds to the expected value of y when x is zero with the v=A level!

```
modC_int <- lm(y ~ x * v, data = dfC)
summary(modC_int)
```

```
##
## Call:
## lm(formula = y ~ x * v, data = dfC)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.6171 -0.8263  0.1105  0.7751  1.8525
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -0.4748     0.2394  -1.984 0.051404 .
## x             1.0819     0.2740   3.948 0.000191 ***
## vB            1.4377     0.3829   3.755 0.000365 ***
## vC           -0.2113     0.3493  -0.605 0.547224
## vD           -0.3786     0.3409  -1.111 0.270633
## x:vB          0.1585     0.5043   0.314 0.754209
## x:vC         -3.0104     0.3545  -8.492 3.13e-12 ***
## x:vD         -1.2546     0.3686  -3.403 0.001127 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

```
## Residual standard error: 1.054 on 67 degrees of freedom
## Multiple R-squared:  0.6643, Adjusted R-squared:  0.6293
## F-statistic: 18.94 on 7 and 67 DF,  p-value: 1.085e-13
```

7g)

Which of the two models, `modC_add` or `modC_int`, are better?

Is your result consistent with the figure you created in 6d)?

SOLUTION

1) A higher R-squared value indicates that the model explains a greater proportion of variance in the response variable. However, for models with a different number of predictors, it's better to look at the adjusted R-squared.
2) Unlike R-squared, the adjusted R-squared accounts for the number of predictors in the model, which can provide a more accurate comparison for models with different numbers of predictors.
3) Adjusted R-squared value in the model in Problem 7f) is 0.6293.
4) Adjusted R-squared value in the model in Problem 7e) is 0.1622.
5) Typically, the model with the higher adjusted R-squared is considered better. Thus we can conclude that the model `modC_int` is better.
6) We concluded that the interaction model is better. This is consistent with the plot in Problem 7d), which shows different slopes for different levels of the categorical variable v.