# CS 1675 Spring 2020: Homework 07

Assigned April 11, 2020; Due: April 19, 2020

Your name here

Submission time: April 19, 2020 at 5:00PM EST

**Collaborators** Include the names of your collaborators here.

## Overview

This assignment focuses on applying multiple models to a regression problem and a binary classification problem. You will train and tune several models ranging from simple to complex. You will then compare those models with 5-fold cross-validation to identify the model that appears to generalize the best. You will also get a little extra practice with visualizations, and you will work with categorical inputs.

**IMPORTANT:** Please pay attention to the `eval` flag within the code chunk options. Code chunks with `eval=FALSE` will **not** be evaluated (executed) when you Knit the document. You **must** change the `eval` flag to be `eval=TRUE`. This was done so that you can Knit (and thus render) the document as you work on the assignment, without worrying about errors crashing the code in questions you have not started. Code chunks which require you to enter all of the required code do not set the `eval` flag. Thus, those specific code chunks use the default option of `eval=TRUE`.

## Load packages

You will use `caret` to manage the resampling, tuning, and fitting of the various models. The code chunk below loads in `dplyr`, and `caret`, and is usually done, as well as `caret`. You will also need the `coefplot`, `corrplot`, and `mlbench` packages to complete this assignment. If you do not have either `coefplot`, `corrplot`, `mlbench` please download and install them. You may use the RStudio Install Packages GUI to do so. The `caret` package will prompt you to install the required packages for each of the methods you will consider in this assignment.

The second bonus questions requires the `plotROC` package. If you do not have the the `plotROC` package please download it.

```
library(dplyr)
library(ggplot2)

library(caret)
```

## Problem 01

The data set of interest is loaded for you in the code chunk below. The response `y` is continuous and there are six inputs. The first two, `x1` and `x2`, are categorical or discrete inputs, while the remaining four inputs are continuous. Before the `glimpse()` function is called, to give a quick view of the data, the categorical inputs are converted from `"character"` classes to `"factor"` for you.

```
prob_01_df <- readr::read_csv("https://raw.githubusercontent.com/jyurko/CS_1675_Spring_2020/master/hw_da
```

```
## Parsed with column specification:
## cols(
##   x1 = col_character(),
##   x2 = col_character(),
##   x3 = col_double(),
##   x4 = col_double(),
##   x5 = col_double(),
##   x6 = col_double(),
##   y = col_double()
## )
```

```
prob_01_df <- prob_01_df %>%
  mutate_at(c("x1", "x2"), as.factor)

prob_01_df %>% glimpse()
```

```
## Observations: 350
## Variables: 7
## $ x1 <fct> C, B, B, C, B, B, B, B, A, B, C, B, C, B, C, B, B, C, A, B, A, C...
## $ x2 <fct> b, b, a, b, b, a, b, a, b, b, b, a, a, a, b, a, b, b, b, a, a, a...
## $ x3 <dbl> -4.4305418, 2.4612167, -1.3995810, -0.1730205, 1.6056858, -2.059...
## $ x4 <dbl> -1.91019553, 3.84940449, 2.35145439, -4.39345253, 1.13180505, -0...
## $ x5 <dbl> -0.09181755, 1.22868033, 0.44622505, 1.01273857, 1.04781499, -0....
## $ x6 <dbl> -0.7518888, 0.9029199, 1.6786596, -1.1314424, 1.2358925, 0.33449...
## $ y  <dbl> -0.2350620, -2.3769196, 0.8064492, -2.9197783, -4.2733737, -2.82...
```

**1a)**

We should always start out any modeling task by exploring the data set. Since there are two categorical variables, `x1` and `x2`, let's get an idea about the number of observations associated with the combinations of the two.

**PROBLEM Pipe the `prob_01_df` data set into `ggplot()`. Set the `x` aesthetic equal to `x1` and then add a bar graph layer with the `geom_bar()` function. Within `geom_bar()` set the `fill` aesthetic equal to `x2`. Within `geom_bar()` set the `position` argument equal to `"dodge"`. Be careful about what needs to go inside the `aes()` function within the call to `geom_bar()`!**

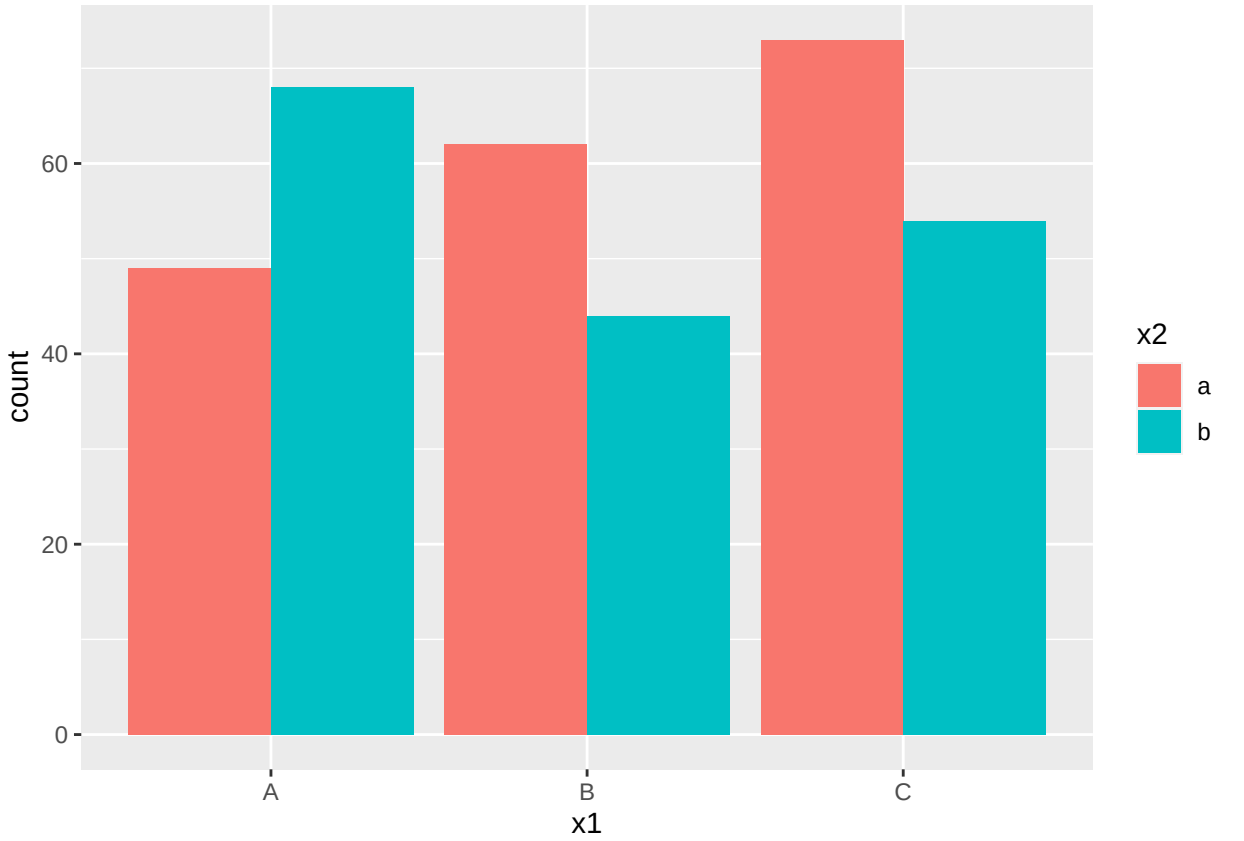
**Your result should show a bar giving the number of observations for each level of `x1` filled by the levels of `x2`. Is there a particular combination of the two variables that dominates the counts in the data set? Or is the data set fairly uniform between the two discrete variables?**

```
prob_01_df
```

**SOLUTION**

```
## # A tibble: 350 x 7
##    x1    x2        x3     x4      x5     x6      y
##    <fct> <fct>  <dbl>  <dbl>   <dbl>  <dbl>  <dbl>
##  1 C     b     -4.43  -1.91  -0.0918 -0.752 -0.235
##  2 B     b      2.46   3.85   1.23    0.903 -2.38
##  3 B     a     -1.40   2.35   0.446   1.68   0.806
##  4 C     b     -0.173 -4.39   1.01   -1.13  -2.92
##  5 B     b      1.61   1.13   1.05    1.24  -4.27
##  6 B     a     -2.06  -0.690 -0.694   0.334 -2.83
##  7 B     b      0.835  0.150  0.264   0.858 -6.18
##  8 B     a     -0.555 -2.19  -0.424  -2.15   1.16
##  9 A     b      2.06  -0.265  1.22    1.87  -1.70
## 10 B     b      0.608  2.91  -2.22   -1.10  -2.08
## # ... with 340 more rows
```

```
prob_01_df %>% ggplot(mapping = aes(x = x1)) +
  geom_bar(mapping = aes(fill = x2), position  = 'dodge')
```




when X1 = C, the combination counts are greater than the other combinations but not necessarily dominates the data set.

**1b)**

Let's now take a look at the summary statistics of the continuous inputs for each combination of the discrete inputs. The code chunk below is completed for you. The `prob_01_df` data set is converted into a long-format data set and assigned to the `lf_01` object. As shown by the print out of the first few rows, the continuous

variables have been gathered into a new column, `input_name` with the value contained in the `input_value` column. The categorical inputs and the response were not gathered together, and so are still stored in separate columns.

```
lf_01 <- prob_01_df %>%
  tibble::rowid_to_column("obs_id") %>%
  tidyr::gather(key = "input_name", value = "input_value",
                -obs_id, -x1, -x2, -y)

lf_01 %>% head()
```

```
## # A tibble: 6 x 6
##   obs_id x1    x2         y input_name input_value
##    <int> <fct> <fct>  <dbl> <chr>            <dbl>
## 1      1 C     b     -0.235 x3              -4.43
## 2      2 B     b     -2.38  x3               2.46
## 3      3 B     a      0.806 x3              -1.40
## 4      4 C     b     -2.92  x3              -0.173
## 5      5 B     b     -4.27  x3               1.61
## 6      6 B     a     -2.83  x3              -2.06
```

To confirm the continuous variables are contained in the `input_name` column, the code chunk below uses the `count()` function to count all rows associated with each unique value of the `input_name` column in `lf_01`. As you should see below, the number of rows associated with each unique `input_name` level corresponds to the number of rows in the `prob_01_df` data set, `nrow(prob_01_df)`.

```
lf_01 %>% count(input_name)
```

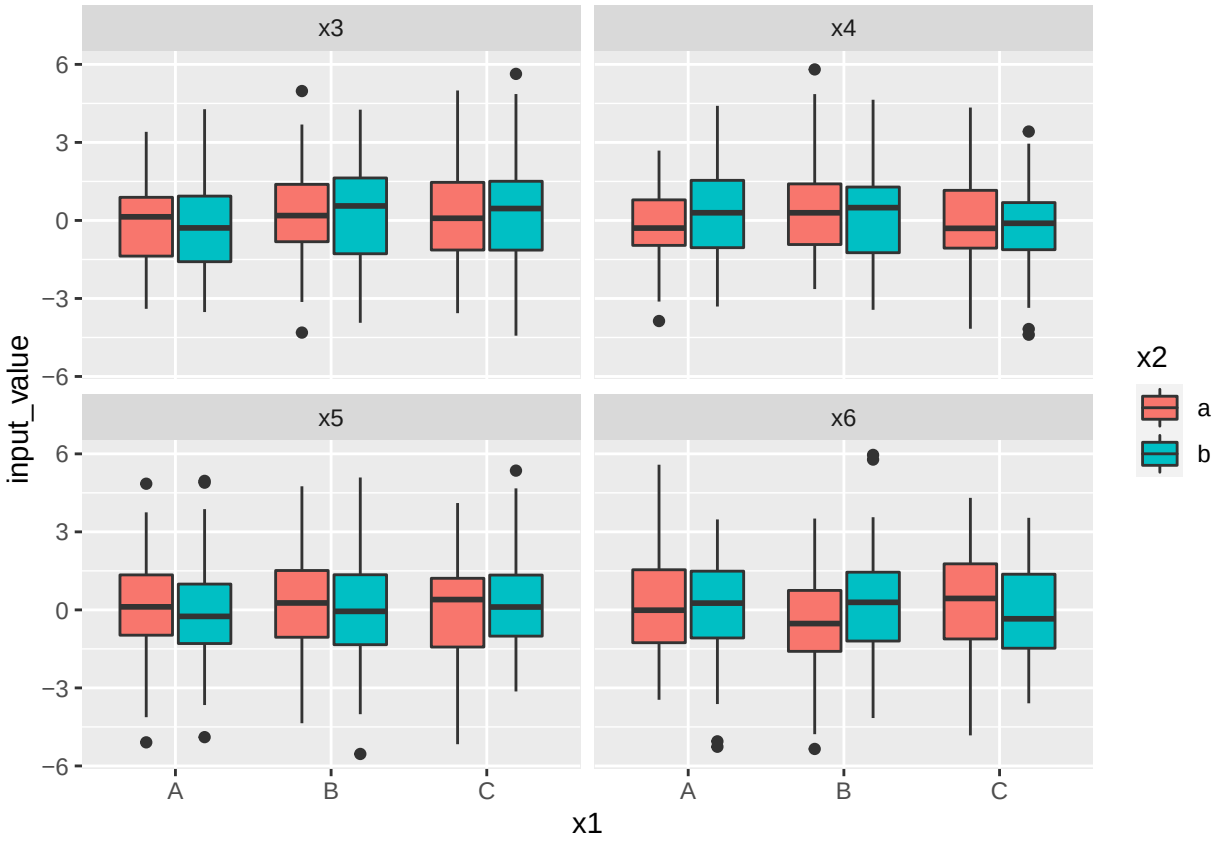
```
## # A tibble: 4 x 2
##   input_name     n
##   <chr>      <int>
## 1 x3           350
## 2 x4           350
## 3 x5           350
## 4 x6           350
```

You will now use this long-format data set to study the behavior of the continuous input variables.

**PROBLEM Summarize the distributions of the continuous inputs with boxplots for each combination of the discrete variables. To do so, pipe the `lf_01` data set into `ggplot()`. Set the `x` aesthetic to `x1` and the `y` aesthetic to `input_value`. Add the boxplot layer to the graphic with the `geom_boxplot()` function. Within `geom_boxplot()`, map the `x2` variable to the fill aesthetic. Lastly, use `facet_wrap()` to create a separate facet (subplot) for each level of `input_name`.**

**Do any of the continuous input summaries change based on the discrete inputs?**

```
lf_01 %>% ggplot(mapping = aes(x = x1, y = input_value)) +
  geom_boxplot(mapping = aes(fill = x2)) +
  facet_wrap(vars(input_name))
```




**SOLUTION**
Yeah all four continuous inputs change in ranges and median as the discrete input x2 changes.

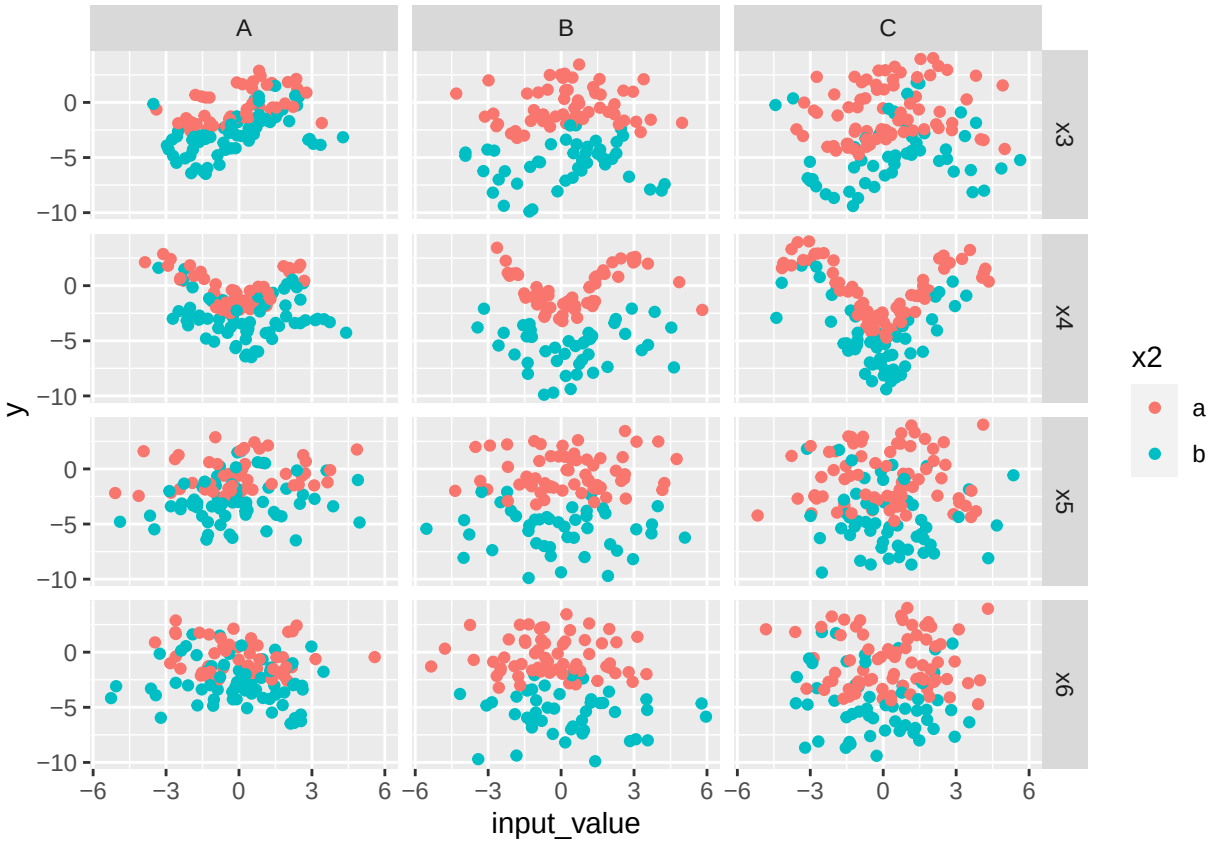
**1c)**

Let's now study the behavior of the response `y` with respect to each of the continuous inputs. Because of how the long-format object, `lf_01`, is structured we can also consider the impact of the discrete inputs. You will visualize the relationships between the response and each input with scatter plots below.

**PROBLEM Pipe the `lf_01` data set into `ggplot()`. Set the `x` aesthetic equal to `input_value` and the `y` aesthetic equal to `y`. Add a scatter plot with the `geom_point()` function, and set the color aesthetic equal to the `x2` variable. Create separate facets with the `facet_grid()` function by faceting horizontally by the `input_name` and vertically by `x1`.**

**Describe the kind of relationships you observe. Do the trends of the response with the continuous inputs seem to be depend on the discrete inputs?**

```
lf_01 %>% ggplot(mapping = aes(x = input_value, y = y)) +
  geom_point(mapping = aes(color = x2)) +
  facet_grid(rows = vars(input_name), cols = vars(x1))
```




**SOLUTION**

Yes, all continuous inputs depend on the discrete inputs. If we compare the subplots on the same row we can see that the function trend changes drastically in response to different discrete values of X1. They also change with different x2 values as well.

**1d)**

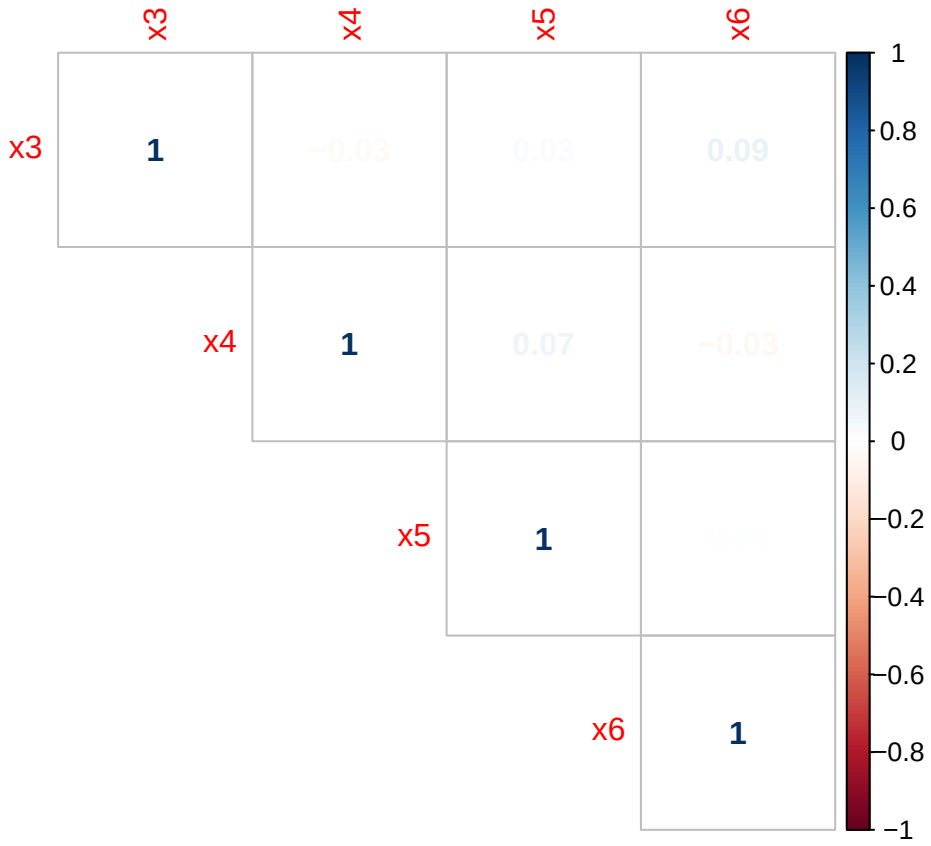
Lastly, let's study the correlation structure between the continuous inputs. We could break this up by each combination of the discrete inputs, but we will keep it simple for now and consider all observations together.

**PROBLEM Pipe the original data set, `prob_01_df`, into `dplyr::select()` and remove the discrete inputs and the response. You can remove a column (a variable) by typing - within the `select()` call. For example, to remove the y variable you type `dplyr::select(-y)`. Pipe to the `cor()` function and then pipe to `corrplot::corrplot()`. Set the `method` argument to be `"number"` and the `type` argument to be `"upper"`.**

**Are continuous inputs correlated?**

*HINT*: The concrete and ionosphere example markdowns from the applied machine learning portion of the course (weeks 04 and 05) may help with required syntax to create this figure.

```
prob_01_df %>%
  dplyr::select(-x1, -x2, -y) %>%
  cor %>%
  corrplot::corrplot('number', 'upper')
```




**SOLUTION**
I think the continuous inputs are all minutely correlated to each other as their correlation plots all have very small but non-zero values.

## Problem 02

With the basic EDA completed, it's time to build predictive models. You will use `caret` to train the models, rather than using the function calls directly. Thus, instead of using `lm()` to fit a linear model, you use the `train()` function from `caret` and set the `method` argument equal to `"lm"`.

### 2a)

The first step in training models with `caret` is to specify the resampling scheme. You will use 5-fold cross-validation to assess the performance on the models. You must also specify a primary performance metric in order for `caret` to select the best tuning parameters (when tuning parameters are present). Since this is a regression problem, we will consider minimizing the root mean square error (RMSE) as our primary performance metric.

**PROBLEM Use the `trainControl()` function to specify using 5-fold cross-validation. The result is assigned to the `ctrl_k05` variable for you. Also, assign the character string `"RMSE"` to the variable `metric_01`.**

```
ctrl_k05 <- trainControl(method='cv', number=5)

metric_01 <- "RMSE"
```

**SOLUTION**

**2b)**

You will now fit a linear model and assess its performance with 5-fold cross-validation. With this data set, you do not really need to standardize the continuous inputs, but you will do so to get practice using the `preProc` argument to `train`. The linear model will be a linear additive relationship between the response and the inputs. You could type out the complete formula by hand to do this. Instead though, you can use the shortcut `y ~ .` to represent that the response, `y`, is a function of "everything else in the data set". By default for linear models, "everything else" corresponds to creating linear additive relationships (no interactions and no non-linear basis functions).

**PROBLEM Fit a linear model between the response `y` and all inputs. Set the `data` argument to be `prob_01_df`. To use a linear model, set the `method` argument equal to `"lm"`. Set the `metric` argument to be `metric_01`. Tell `caret` to standardize the continuous inputs by setting the `preProc` argument to be `c("center", "scale")`. Lastly, set the `trControl` argument to be `ctrl_k05`.**

**The result is printed to screen for you. Approximately, how many observations are used to test the model each fold? What is the cross-validation averaged R-squared value?**

```
set.seed(3321)
fit_01_lm <- train(y ~ ., data=prob_01_df, method='lm', preProcess=c('center', 'scale'), metric=metric_

fit_01_lm
```

**SOLUTION**

```
## Linear Regression
##
## 350 samples
##   6 predictor
##
## Pre-processing: centered (7), scaled (7)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 281, 280, 279, 280, 280
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   2.191728  0.4391936  1.771226
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

70 observatins are used to test each fold and the cross-validation averaged R-squared is 0.4391936

**2c)**

Once you have trained a model with `caret` you can access the model object directly as `$finalModel`. Thus, we can perform all actions on `fit_01_lm$finalModel` that we would do if we had fit the model with `lm()`. The code chunk below demonstrates this by printing the OLS model coefficients (the $\beta$ parameters) for you.

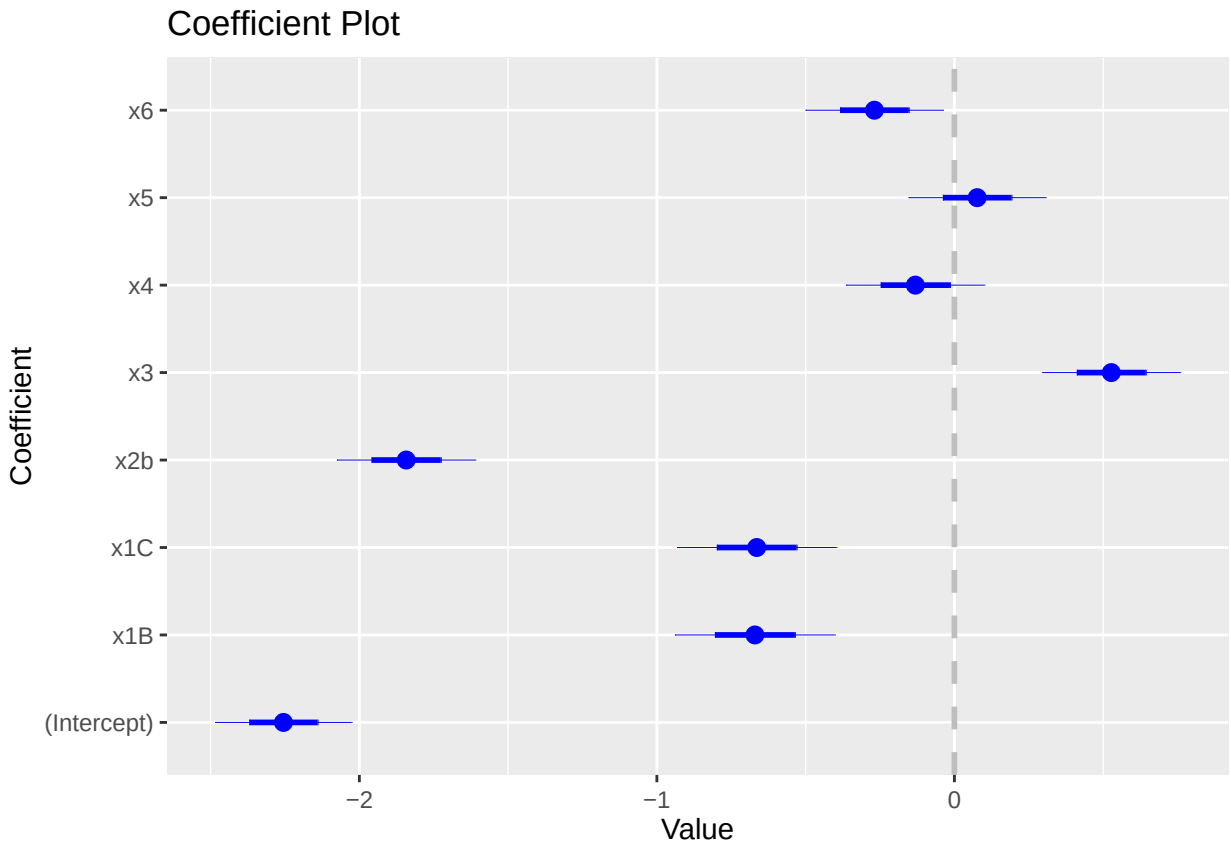
```
coef(fit_01_lm$finalModel)
```

```
## (Intercept)          x1B          x1C          x2b           x3           x4
## -2.25521407 -0.67051619 -0.66437930 -1.84264385  0.52733358 -0.13131150
##           x5           x6
##  0.07651872 -0.26894796
```

However, rather than looking at print outs or summaries for the coefficients, lets visualize the OLS estimates and their confidence intervals. You will use the convenient `coefplot::coefplot()` function to create a visualization showing all of the parameter estimates and 95% confidence intervals. The syntax is simple, you just need to set the model object as the first argument to `coefplot::coefplot()`. For example, if the `caret` model object was `my_model` we can create the necessary figure with `coefplot::coefplot(my_model$finalModel)`.

**PROBLEM Create the coefficient plot for the linear model you fit in the Problem 2b). Do any of the coefficients have confidence intervals that contain zero? What does that mean if the confidence intervals contain zero?**

```
coefplot::coefplot(fit_01_lm$finalModel)
```




**SOLUTION**
Yes, x4 and x5 coefficients interval contain 0. That means those coefficients might be inessential to the response.

**2d)**

The `prob_01_df` data set contains `ncol(prob_01_df) - 1` inputs and 1 response. Your coefficient plot figure in Problem 2c) however, should show 8 coefficients, including the intercept. If we simply used linear additive terms for our model, why does the number of coefficients not equal the number of inputs plus 1?

**PROBLEM** **There are $D = 6$ inputs. Why are there are $D + 1 = 7$ coefficients (including the intercept)?**

**SOLUTION** Since the input X1 has 3 possible values A, B and C, and X2 has only 2, we have to derive two variables for X1 but only one for X2 for the sake of their discrete values. In total, we have 8 coefficients (including the response).

**2e)**

Let's now consider fitting a linear model with all pair-wise interactions. Typing out all of the terms using the formula interface would be quite tedious. So we will use a short cut to help. The short cut in the formula interface to denote all pair-wise combinations of "everything else" is `(.)^2`. Thus, with the response `y` the formula for all pair-wise combinations is `y ~ (.)^2`.

**PROBLEM** **Use the `model.matrix()` function to check how many columns and thus how many parameters we would have in a model with all pair-wise interactions of the inputs.**

**SOLUTION** How many columns are in the design matrix?

```
#(model.matrix(y ~ (.)^2, data=prob_01_df)) %>% head
ncol(model.matrix(y ~ (.)^2, data=prob_01_df))
```

```
## [1] 28
```

**2f)**

Let's try fitting a linear model with all pair-wise interactions. Use `caret` to perform the training, and thus assess the model performance with 5-fold cross-validation. You will use the same settings as you did when you trained the `fit_01_lm` model. Thus, the only change that must be made is to the formula.

**PROBLEM** **Train a linear model with all pair-wise interactions. Use the same specifications as you did in Problem 2b), but change the formula such that you are considering all pair-wise interactions.**

**The average cross-validation performance is printed to the screen for you. Does this model appear toperform better, as assessed by 5-fold cross-validation, compared to the linear model with linear additive terms?**

```
set.seed(3321)
fit_01_lm_pairs <- train(y ~ (.)^2, data=prob_01_df, method='lm', preProcess=c('center', 'scale'), metr

fit_01_lm_pairs
```

**SOLUTION**

```
## Linear Regression
##
## 350 samples
##   6 predictor
##
## Pre-processing: centered (27), scaled (27)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 281, 280, 279, 280, 280
## Resampling results:
##
##   RMSE      Rsquared  MAE
##   2.219731  0.429152  1.792783
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

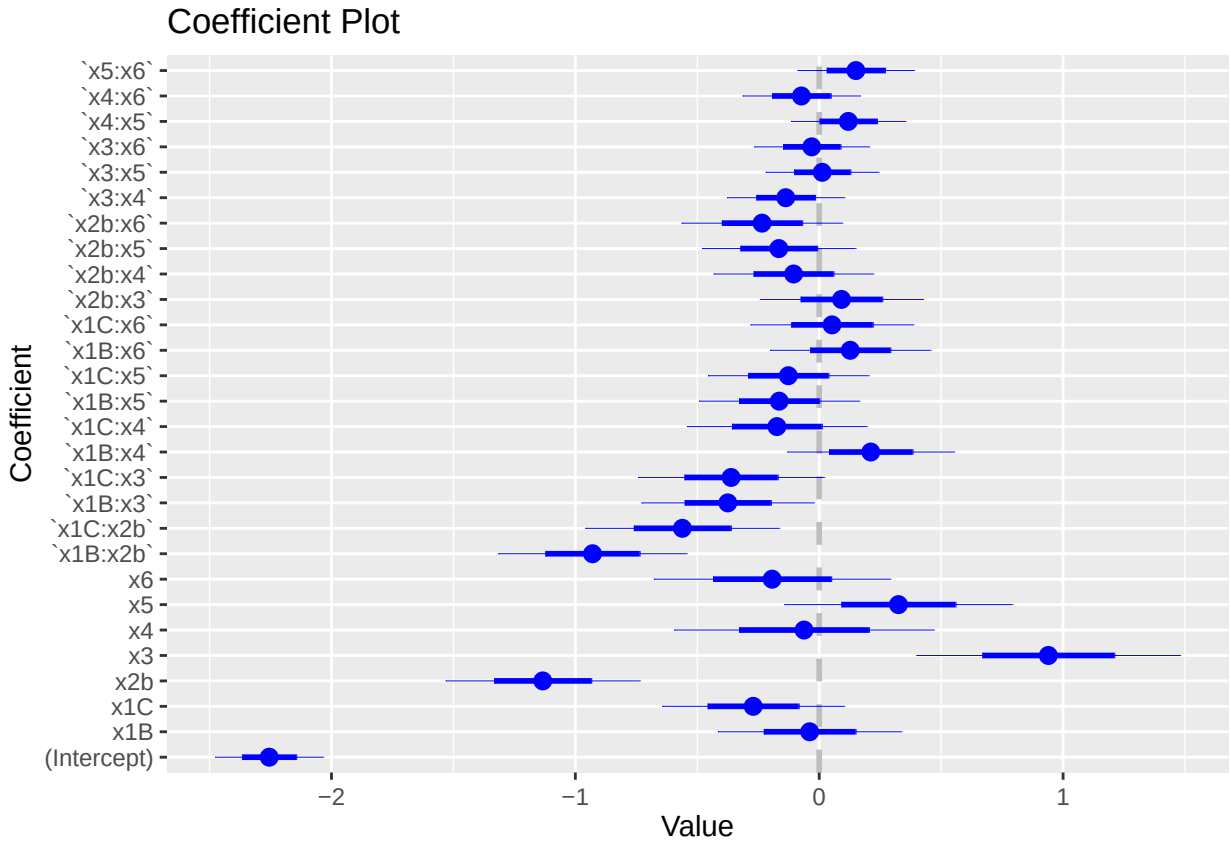
This model doesn't perform better since our metric, 'RMSE', shows that this model has a higher loss.

**2g)**

Visualize the coefficients on the model with all pair-wise interactions using `coefplot::coefplot()`.

**PROBLEM Visualize the coefficient plot for the model with all pair-wise interactions. Are any of the interaction terms significant?**

```
coefplot::coefplot(fit_01_lm_pairs$finalModel)
```




**SOLUTION**
the interation terms x1B:x2b and x1C:x2b seems to indicate a strong correlation of their constituent variables.

**2h)**

Let's now see what happens if we allow for all triplet interactions. Thus we will have a model with terms such as `x3:x4:x5`. Let's apply regularization with elastic net to see if we can effectively turn off terms that do not matter, since the number of parameters will now really start to grow.

You will use the `glmnet` package to fit the elastic net model. `caret` treats the penalty factor `lambda` and the mixing weight `alpha` as tuning parameters. By default `caret` uses a simple grid search to optimize these tuning parameters. We can supply a custom grid to find the optimal tuning parameters, but `caret` provides a simple default grid of tuning parameters that we can use. You will just use the default grid to keep things simple for now.

**PROBLEM Train an elastic net model which considers all triplet interactions between the inputs. You can use the same settings that you used in Problem 2f), EXCEPT you must change the formula slightly and you must set the `method` argument to `"glmnet"`.**

**Based the result from the default grid, is your trained elastic net mode considered to be more like LASSO or RIDGE regression?**

*HINT*: If `y ~ (.)^2` allows you to try out all pair-wise interactions, what do you think you should do to try out all triplet or three-way interactions?

```
set.seed(3321)
fit_01_glmnet_trips <- train(y ~ (.)^3, data=prob_01_df, method='glmnet', preProcess=c('center', 'scale

fit_01_glmnet_trips
```

**SOLUTION**

```
## glmnet
##
## 350 samples
##   6 predictor
##
## Pre-processing: centered (57), scaled (57)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 281, 280, 279, 280, 280
## Resampling results across tuning parameters:
##
##   alpha  lambda       RMSE      Rsquared   MAE
##   0.10   0.003509359  2.474299  0.3496508  1.941549
##   0.10   0.035093591  2.430573  0.3596107  1.913115
##   0.10   0.350935911  2.281892  0.3982671  1.827165
##   0.55   0.003509359  2.463810  0.3522683  1.934794
##   0.55   0.035093591  2.354247  0.3824714  1.868387
##   0.55   0.350935911  2.195470  0.4443639  1.770055
##   1.00   0.003509359  2.453181  0.3551276  1.927654
##   1.00   0.035093591  2.312431  0.3948185  1.845991
##   1.00   0.350935911  2.225719  0.4469942  1.795221
##
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were alpha = 0.55 and lambda = 0.3509359.
```

Since RMSE is used to select the optimal model, the optimal model has alpha $= 0.55$ and lambda $= 0.3509$. Since alpha of this elastic net is 0.55, we could say that the trained elastic net mode is slightly more like LASSO.

## Problem 03

You will try out non-linear methods to see if you can improve upon the results from the linear models.

**3a)**

You will fit a neural network model with `nnet`. To tell `caret` to use `nnet` you must set the `method` argument equal to `"nnet"`. Non-linear models will attempt to find non-linear relationships between the inputs and the response even if the simple formula `y ~ .` is used. This provides a simple and convenient approach to tell a complex model like a neural network to learn the relationships between the response and all inputs. You must just be careful to make sure that the data set you provide only contains the response and the inputs, and no other variables (such as index or identifier variables).

**PROBLEM Train a neural net model by setting `method` equal to `"nnet"`. You may continue to use the default tuning grid and you should also standardize the inputs with the `preProc` argument, as you have done in previous problems.**

**The code chunk below specifies the `trace` argument to be FALSE for you. Otherwise, all of the iterations results will be printed to the screen.**

**How do the RMSE and R-squared compare with the linear models? What are the tuning parmaters for the best performing neural network model, using the default search grid?**

```
set.seed(3321)
fit_01_nnet <- train(y ~ .,
                     data=prob_01_df,
                     method='nnet',
                     preProcess=c('center', 'scale'),
                     metric=metric_01,
                     trControl=ctrl_k05,
                     trace=FALSE,
                     linout=TRUE)

fit_01_nnet
```

**SOLUTION**

```
## Neural Network
##
## 350 samples
##   6 predictor
##
## Pre-processing: centered (7), scaled (7)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 281, 280, 279, 280, 280
## Resampling results across tuning parameters:
##
##   size  decay  RMSE      Rsquared   MAE
##   1     0e+00  2.202831  0.4413478  1.7937456
##   1     1e-04  2.211762  0.4344381  1.7925839
##   1     1e-01  2.202833  0.4384675  1.7816349
##   3     0e+00  2.060454  0.5078552  1.6159497
##   3     1e-04  1.947883  0.5458880  1.5809284
##   3     1e-01  1.863911  0.5836970  1.5147935
##   5     0e+00  1.188181  0.8248368  0.8696638
```

```
##   5     1e-04  1.494006  0.7298137  1.1905098
##   5     1e-01  1.416642  0.7540679  1.0639695
##
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were size = 5 and decay = 0.
```

RMSE and Rsquared have quite distinct evaluation for the models. RMSE indicates the model with the best tuning parameters as size = 5 and decay = 0, while Rsquared the model with size as 1 and 1e-04 decay.

**3b)**

Let's now try a random forest model. To train a random forest model, you must specify `method` equal to `"rf"`. Up to this point you have performed standardization on the inputs. Do you need to consider standardization for a random forest?

**PROBLEM Train a random forest model. Note that in the code chunk the `importance` argument has been set to TRUE for you. Does it matter if you apply preprocessing to the continuous inputs of a random forest model?**

**Which `mtry` value is considered the best, and what does the `mtry` parameter correspond to in the model?**

```
set.seed(3321)
fit_01_rf <- train(y ~ .,
                   data=prob_01_df,
                   method='rf',
                   preProcess=c('center', 'scale'),
                   metric=metric_01,
                   trControl=ctrl_k05,
                   importance=TRUE)

fit_01_rf
```

**SOLUTION**

```
## Random Forest
##
## 350 samples
##   6 predictor
##
## Pre-processing: centered (7), scaled (7)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 281, 280, 279, 280, 280
## Resampling results across tuning parameters:
##
##   mtry  RMSE      Rsquared   MAE
##   2     1.378894  0.8370174  1.0604865
##   4     1.082219  0.8759997  0.7821625
##   7     1.056450  0.8676836  0.7613596
##
```

```
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was mtry = 7.
```

Preprocessing doesn't matter since the scale or how normalized the observations are doesn't matter since decision tree-based methods only concern with classification. the mtry value 7 is considered the best. It indicates the number of variables available for splitting at each tree node.

**3c)**

Now try training a boosted tree model with `xgboost`. You must set the `method` argument to `xgbTree` in order to tell `caret` to use the XGBoost implementation of the boosted tree algorithm. By default, many different tuning parameters are considered, so instead of printing out the results, the the best tuning parameters are printed out for you. Then, the performance results are plotted for you.

**PROBLEM Train the boosted tree model with `xgbTree`. In XGBoost, the number of trees is denoted as the `nrounds` tuning parameter. How many trees gave the best results? Are decision stumps considered the best type of tree to use as the "base" or "weak" learner that gets sequentially improved?**

*NOTE*: This may take a minute or two to complete.

**SOLUTION** The boosted tree model is fit below.

```
set.seed(3321)
fit_01_xgb <- train(y ~ .,
                    data=prob_01_df,
                    method='xgbTree',
                    preProcess=c('center', 'scale'),
                    metric=metric_01,
                    trControl=ctrl_k05,
                    importance = TRUE)
```

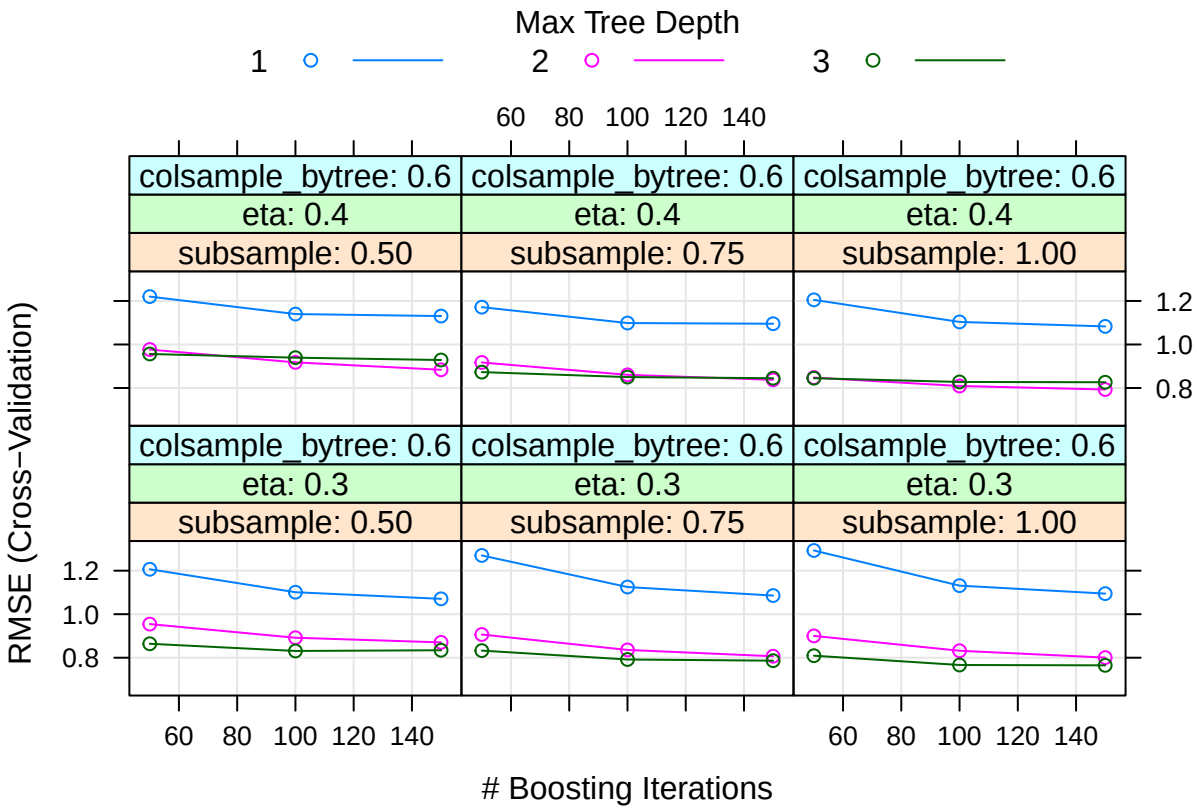
Print out the best tuning parameters.

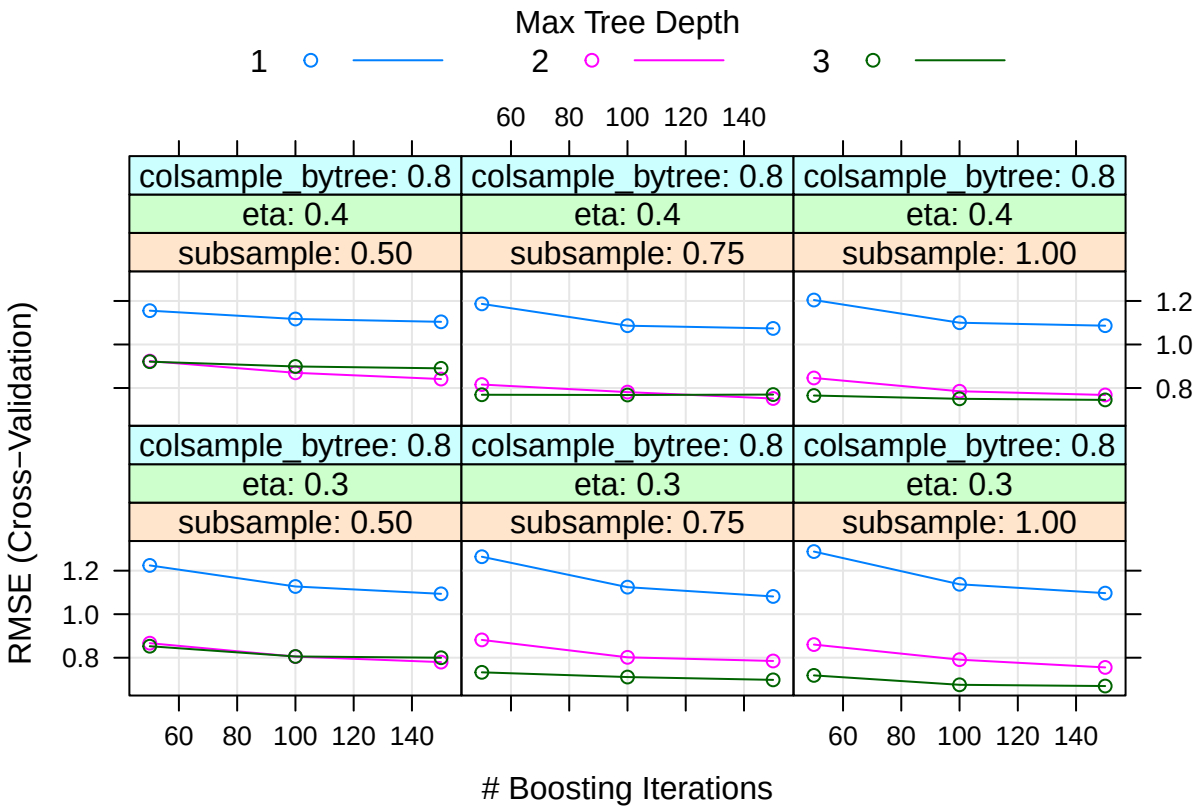
```
fit_01_xgb$bestTune
```

```
##    nrounds max_depth eta gamma colsample_bytree min_child_weight subsample
## 54     150         3 0.3     0              0.8                1         1
```

Plot the cross-validation results for each of the tuning parameter combinations.

```
plot(fit_01_xgb)
```


Max Tree Depth
1
2
3
colsample_bytree: 0.6
eta: 0.4
subsample: 0.50
colsample_bytree: 0.6
eta: 0.4
subsample: 0.75
colsample_bytree: 0.6
eta: 0.4
subsample: 1.00
colsample_bytree: 0.6
eta: 0.3
subsample: 0.50
colsample_bytree: 0.6
eta: 0.3
subsample: 0.75
colsample_bytree: 0.6
eta: 0.3
subsample: 1.00
RMSE (Cross−Validation)
# Boosting Iterations
60
80
100
120
140
0.8
1.0
1.2




The boosted model with 150 trees gives the best result. Decisions stumps are considered as the best base or weak learner since it results in the highest RMSE, very suitable for boosted tree.

**3d)**

Let's now compare all of the methods we have trained. The code chunk below is started for you. You must complete it by assigning each of the `caret` objects to their corresponding names in the list below.

**PROBLEM Complete the list below, by assigning the `caret` model objects to each name in the list. For example, you must assign `fit_01_lm` to the `LM` variable in the list.**

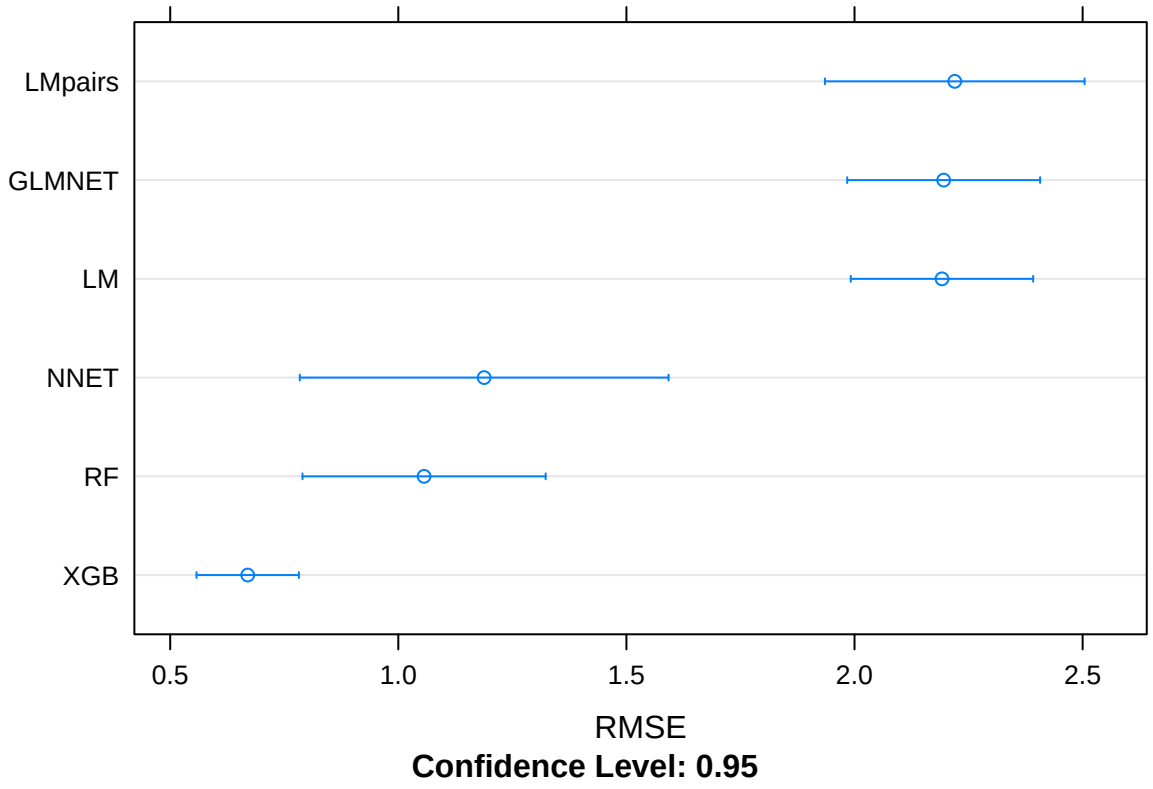
**The `reg_mod_compare` object can then be used to compare the resampling results across the models. The cross-validation averaged performance metrics and confidence intervals are plotted for you below using the `dotplot()` function. Which model is considered the best?**

**SOLUTION** Compile all of the model cross-validation results together.

```
reg_mod_compare <- resamples(list(LM = fit_01_lm,
                                  LMpairs = fit_01_lm_pairs,
                                  GLMNET = fit_01_glmnet_trips,
                                  NNET = fit_01_nnet,
                                  RF = fit_01_rf,
                                  XGB = fit_01_xgb))
```

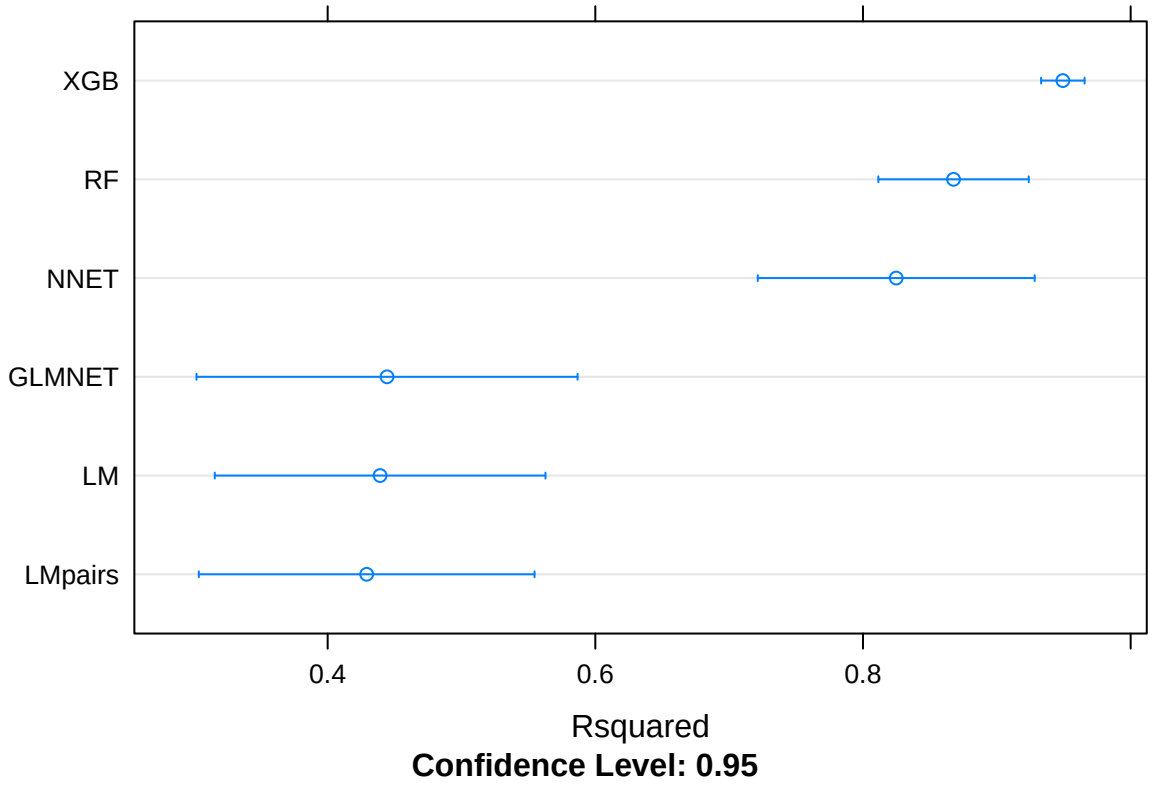
The cross-validation averaged root mean squared error (RMSE) with corresponding confidence intervals are shown in the plot below.

```
dotplot(reg_mod_compare, metric = "RMSE")
```




The cross-validation averaged R-squared with corresponding confidence intervals are shown below.

```
dotplot(reg_mod_compare, metric = "Rsquared")
```




XGB model is the best as it has the least RMSE.

**3e)**

Let's now make predictions with our trained models. The code chunk below defines an input grid for you. The `expand.grid()` function is used to create a grid to allow visualizing the predicted trends with respect to `x3` for several unique values of `x4`, and all combinations of the discrete inputs, `x1` and `x2`. The `x5` and `x6` inputs were set equal to their median values from the training set.

```
test_input_grid_01 <- expand.grid(x1 = unique(prob_01_df$x1),
                                  x2 = unique(prob_01_df$x2),
                                  x3 = seq(min(prob_01_df$x3), max(prob_01_df$x3), length.out = 101),
                                  x4 = seq(min(prob_01_df$x4), max(prob_01_df$x4), length.out = 5),
                                  x5 = median(prob_01_df$x5),
                                  x6 = median(prob_01_df$x6),
                                  KEEP.OUT.ATTRS = FALSE,
                                  stringsAsFactors = FALSE) %>%
  as.data.frame() %>% tbl_df()
```

Predictions with `caret` model objects are simple to make. The basic syntax is:

`predict(<model object>, <new data>)`

Thus, you simply assign the `caret` model object as the first argument to the `predict()` function, and you provide the new data set you wish to make predictions with as the second argument. You do not have to worry about applying the preprocessing actions to the input variables in order to make the predictions. The `caret` model object will do that for you!

Regardless of the which model you identified to be the best, you will make predictions with the boosted tree and the elastic net model. This way you can compare a linear model with a non-linear method.

**PROBLEM** **Make predictions with the boosted tree model and the elastic net model. Assign the boosted tree predictions to the `pred_test_01_xbg` variable and the elastic net predictions to the `pred_test_01_glmnet` variable.**

```
pred_test_01_xbg <- predict(fit_01_xgb, test_input_grid_01)

pred_test_01_glmnet <- predict(fit_01_glmnet_trips, test_input_grid_01)
```

**SOLUTION**

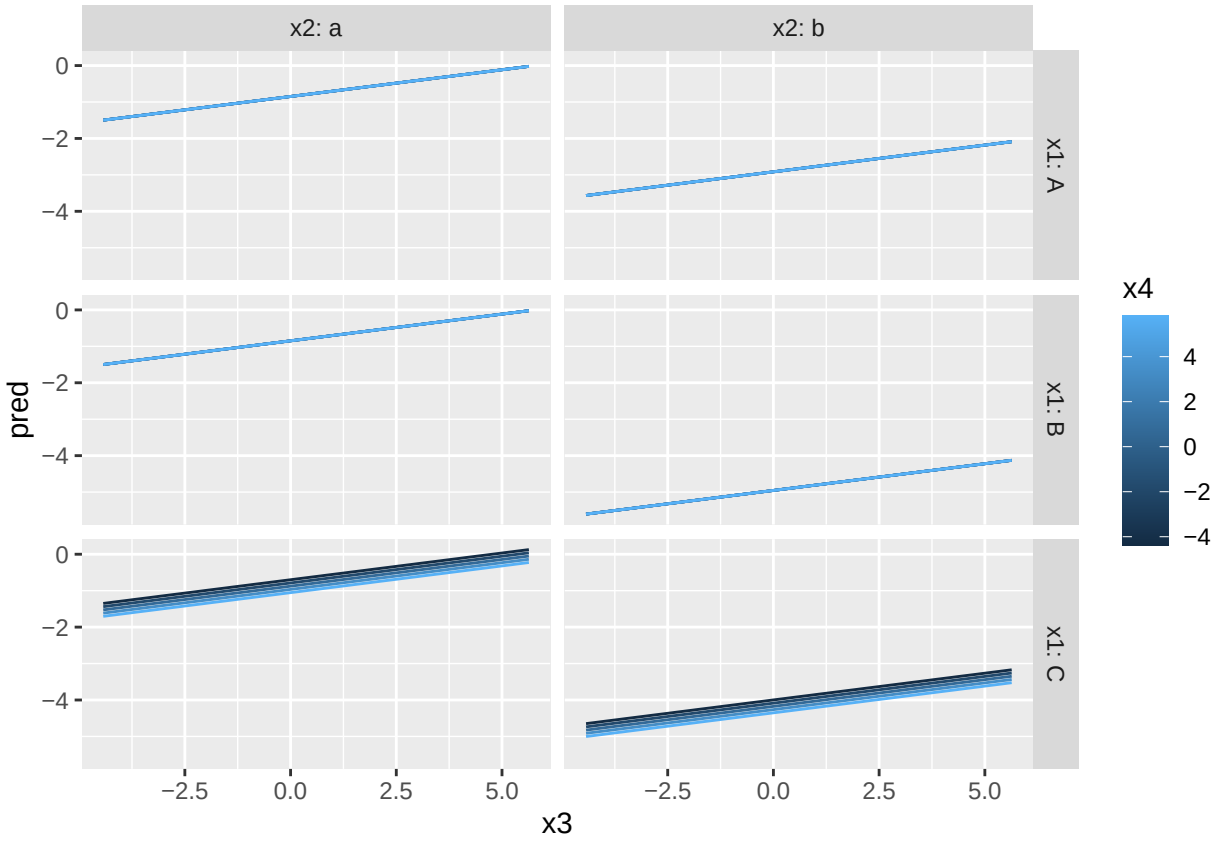
**3f)**

Let's now visualize the predictions. The two code chunks are started for you below. The `test_input_grid_01` object is piped into `mutate()` where the new variable `pred` is assigned. In the first code chunk, the elastic net predictions are assigned, and the boosted tree predictions are assigned in the second code chunk. You must complete the code chunks to create the figures.

**PROBLEM** **Pipe the result into `ggplot()` and set the `x` aesthetic equal to `x3` and the `y` aesthetic equal to `pred`. Plot the predictions with the `geom_line()` function, and set the `color` aesthetic equal to `x4` and the `group` aesthetic equal to `interaction(x1, x2, x4, x5, x6)`. Create a separate facet for each combination of the discrete inputs using the `facet_grid()` function. Have the horizontal facets correspond to `x1` and the vertical subplots correspond to `x2`. Set the `labeller` argumen within `facet_grid()` to be `"label_both"`.**

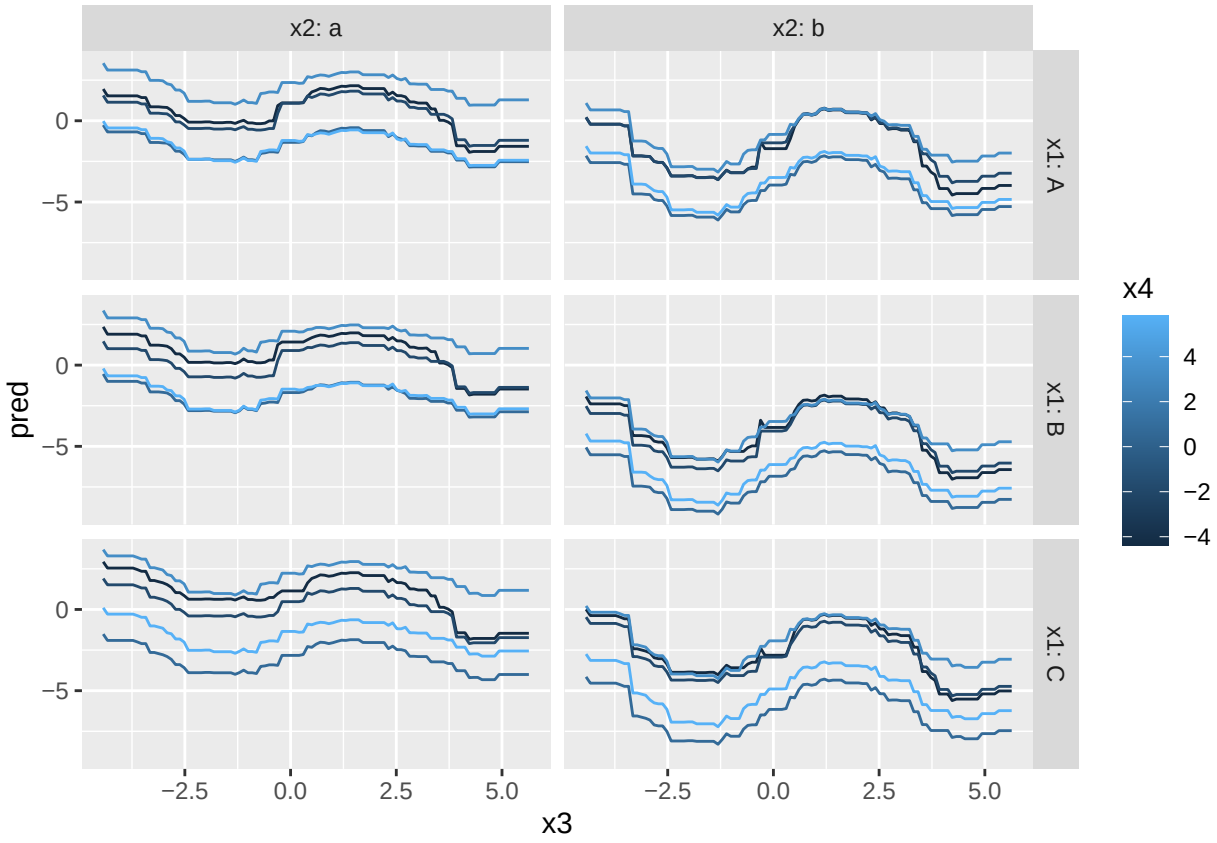
**SOLUTION** Plot the predictions from the elastic net model below.

```
test_input_grid_01 %>%
  mutate(pred = pred_test_01_glmnet) %>%
  ggplot(mapping = aes(x = x3, y = pred)) +
  geom_line(mapping = aes(color = x4, group = interaction(x1, x2, x4, x5, x6))) +
  facet_grid(rows = vars(x1), cols = vars(x2), labeller = label_both)
```




Plot the predictions from the boosted tree model below.

```
test_input_grid_01 %>%
  mutate(pred = pred_test_01_xbg) %>%
  ggplot(mapping = aes(x = x3, y = pred)) +
  geom_line(mapping = aes(color = x4, group = interaction(x1, x2, x4, x5, x6))) +
  facet_grid(rows = vars(x1), cols = vars(x2), labeller = label_both)
```



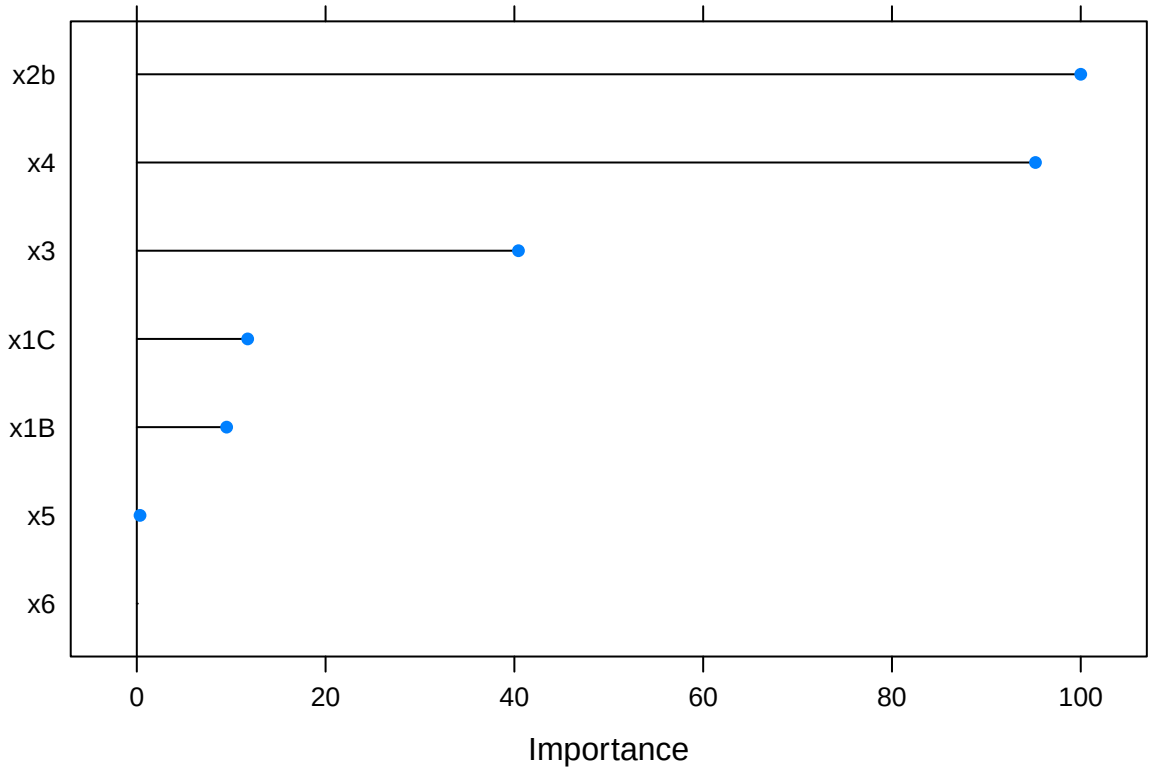

**3g)**

The `importance` argument was set equal to TRUE in each of the two tree based methods. This instructs the methods to keep track of the variables that are selected at each split, and provides a metric for ranking variable importances. Regardless of which model you found to be the best, you will rank the inputs using the boosted tree algorithm.

The variable importances are accessed by calling the `varImp()` function and can be plotted by simply wrapping `plot()` around the `varImp()` call. Thus, to plot the variable importances, simply use `plot(varImp(<caret model>))`.

**PROBLEM Plot the variable importance rankings for the boosted tree model. Which is the most important variable? Which variable is considered the least importance?**

```
plot(varImp(fit_01_xgb))
```




**SOLUTION**
x2b is the most important variable while x6 is the least important, x6 is the least important.

## Problem 04

You will work with the Sonar data set contained in the `mlbench` package as a binary classification data set. As long as you have the `mlbench` package downloaded and installed, the code chunk below loads in the Sonar data set for you.

```
data("Sonar", package = "mlbench")
```

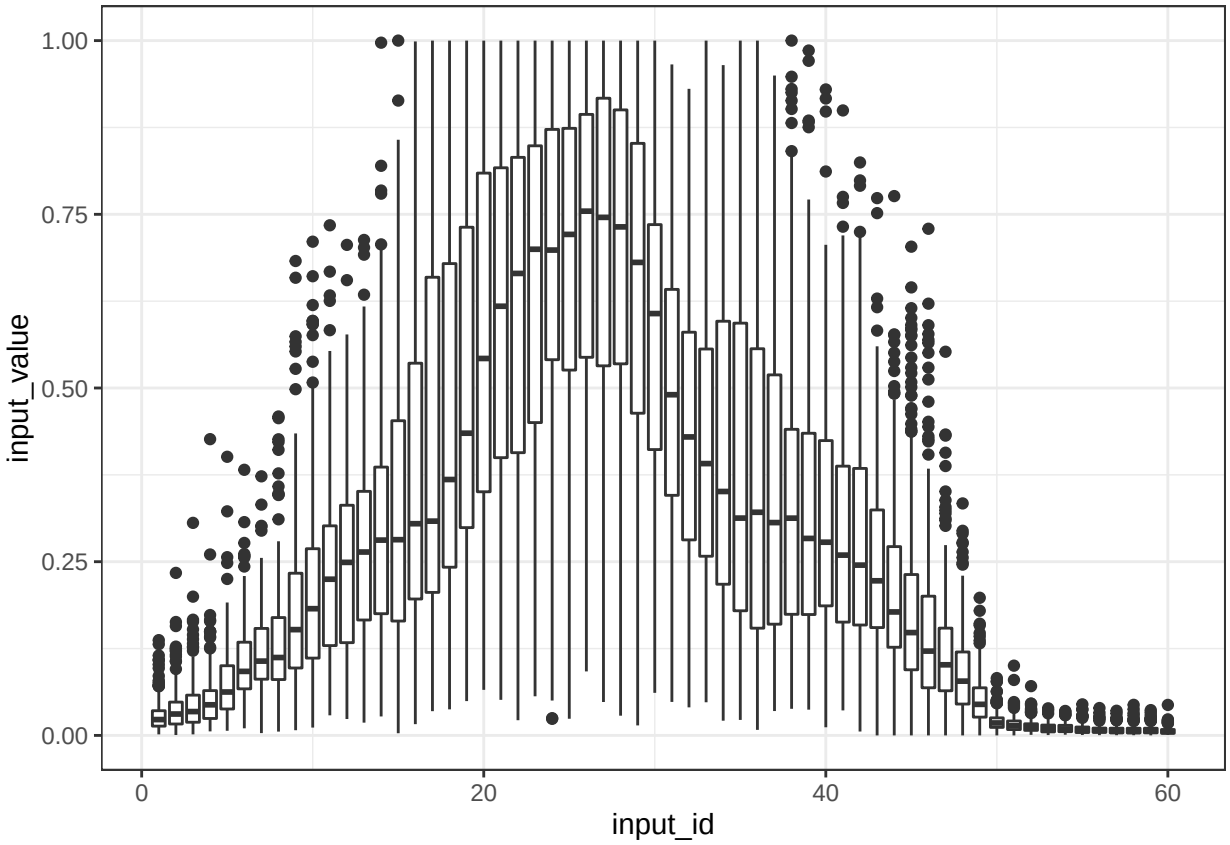
The input features of the Sonar data set correspond to sonar signal (as the name suggests). Your task is to classify the binary outcome, `Class`, as either `"M"` for metal or `"R"` for rock. The Sonar data set has relatively few observations compared with with the number of input features.

### 4a)

We will perform a short exploration of the features in this data set. First, let's look at summary statistics for each of the input variables. The code chunk below creates a boxplot for each input feature. All of the input features are signals which are between 0 and 1, so all inputs can be summarized with the same vertical axis.

```
Sonar %>%
  tibble::rowid_to_column("obs_id") %>%
  tidyr::gather(key = "input_name",
                value = "input_value",
```

```
              -obs_id, -Class) %>%
mutate(input_id = stringr::str_extract(input_name, "\\d+")) %>%
mutate_at("input_id", as.numeric) %>%
tbl_df() %>%
ggplot(mapping = aes(x = input_id,
                     y = input_value)) +
geom_boxplot(mapping = aes(group = input_id)) +
theme_bw()
```




**PROBLEM** **Even though all of the inputs are between 0 and 1, do you think you should still pre-process these inputs? If so, why?**

**SOLUTION** I think yes. Like why we did preprocess for our previous dataset, the 0-1 range is actually pretty large compared to the individual ranges of the inputs. The middle 90 percentiles of many inputs are completely separate from each other.

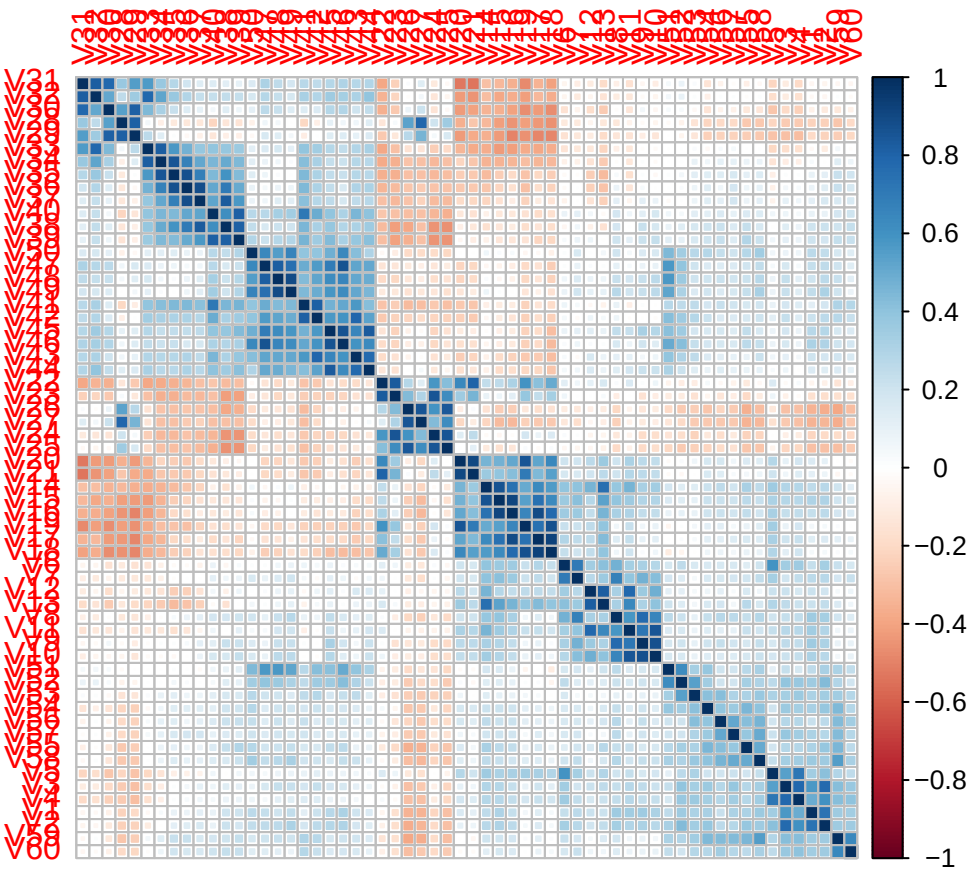
**4b)**

Let's now consider the correlation structure of the inputs. Use the `corrplot::corrplot()` function to create the correlation plot matrix associated with all 60 input features. However, rather than using the default ordering, instruct `corrplot()` to reorder inputs such that all highly correlated inputs are grouped together. `corrplot()` will use a hierarchical clustering method and group all inputs for you.

**PROBLEM** Pipe the `Sonar` data into the `dplyr::select()` function and remove the `Class` variable. Pipe the result into `cor()` and pipe the result into `corrplot::corrplot()`. Set the `method` argument equal to `"square"` and the `order` argument equal to `"hclust"`.

**Is there a correlation structure between the inputs?**

```
Sonar %>%
  dplyr::select(-Class) %>% cor %>% corrplot::corrplot(method = 'square', order = 'hclust')
```




**SOLUTION**
Yes there is.

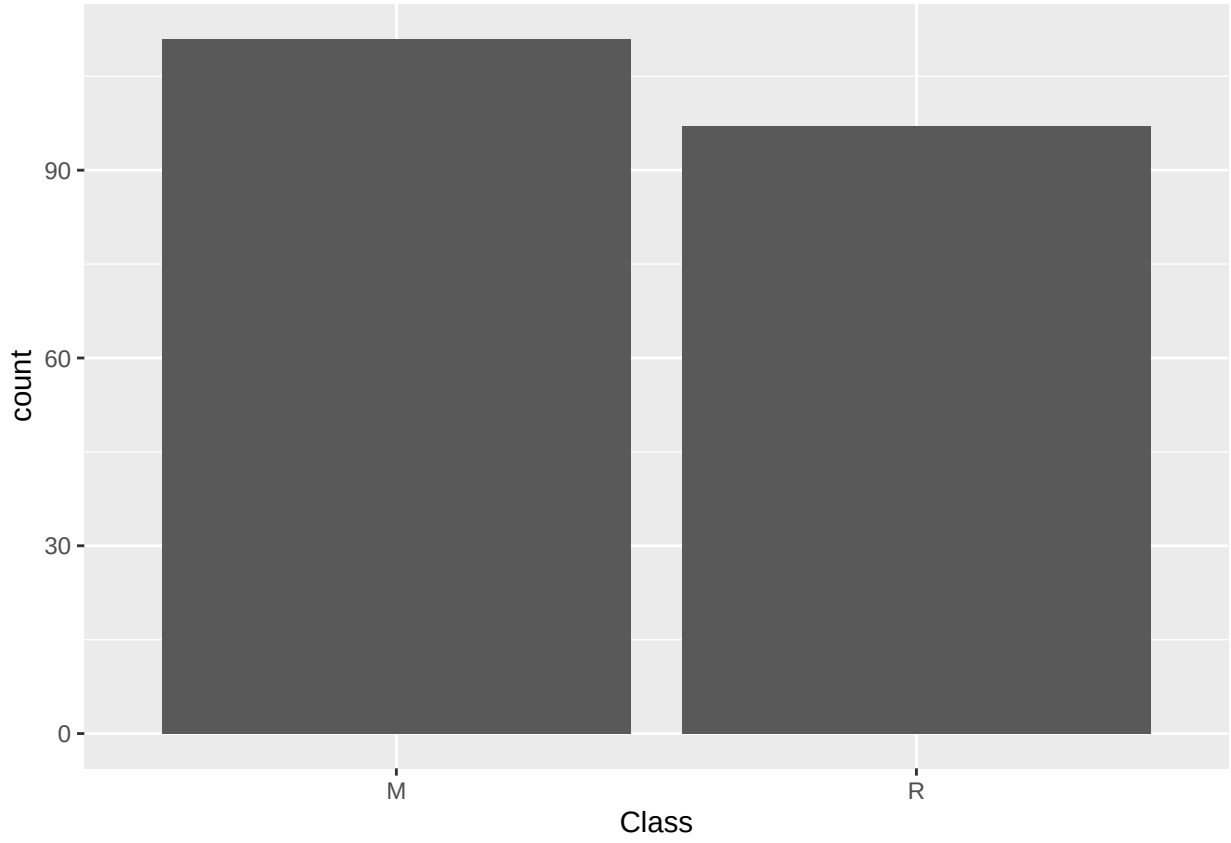
**4c)**

The last visualization you will make as part of a quick exploratry data analysis (EDA) is to count the number of observations per level of the binary outcome `Class`. This is important to do before training any binary classification model, in order to check if there is a severe imbalance between either of the two classes.

**PROBLEM** Pipe the `Sonar` data set into `ggplot()` and set the `x` aesthetic equal to `Class`. Use the `goem_bar()` function to show a bar chart giving the number of observations per level of `Class`.

**Do you think we should be concerned about an imbalance between the `"M"` and `"R"` levels?**

```
Sonar %>% ggplot(mapping = aes(x = Class))+ geom_bar()
```




**SOLUTION**
there is not an severe imbalance between the two levels. We don't need to.

## Problem 05

You will train several models to predict the binary outcome `Class`. You will try logistic regression and more complex non-linear methods.

### 5a)

You must start, as you did with the regression problem, by specifying the resampling scheme and the primary performance metric. You will continue to use 5-fold cross-validation, but in this problem you will compare models by maximizing the Area Under the ROC Curve (AUC). You must specify the metric to be `"ROC"` in order to tell `caret` to maximize the AUC (the naming convention is a little confusing). The code chunk below is started for you, and provides some arguments to the `trainControl()` function which are required in order to use `"ROC"` as the primary performance metric.

**PROBLEM Complete the code chunk below. Finish the `trainControl()` call such that you will use 5-fold cross-validation. Assign `"ROC"` to the `metric_sonar` variable.**

```
ctrl_k05_roc <- trainControl(method = 'cv',
                             number = 5,
                             summaryFunction = twoClassSummary,
                             classProbs = TRUE,
                             savePredictions = TRUE)

metric_sonar <- 'ROC'
```

**SOLUTION**

**5b)**

You will start with a logistic regression model using linear additive terms for all input features. Remember that the short cut operator `.` denotes using "everything in the data set". So you do not have to type out all 60 input variable names in the formula interface. The formula requires the name of the response variable, and so remember that the outcome is the `Class` variable, not `y` as in Problem 1.

You will still use the `train()` function to train the logistic regression model. You must specify the formula interface, and specify the `data` as you did in the regression problem before. However, you must change the arguments to correspond to the values for the binary classification task. To use the base `R` logistic regression method in `glm()`, you must set the `method` argument equal to `"glm"`. You must also specify the `metric` argument to be the `metric_sonar` variable you assigned Problem 4d) in order for `caret` to identify the best model by maximizing the area under the ROC curve. You must also specify the `trControl` argument to be the `ctrl_k05_roc`.

You must set the `preProc` argument based on your answer to Problem 4a). If you do not feel you need to pre-process the inputs then you do not need to include the `preProc` argument. If you feel you should, then you should set the `preProc` argument to your desired pre-processing operation.

**PROBLEM Specify the arguments to the `train()` function in order to train a logistic regression model for the `Sonar` data set with 5-fold cross-validation and calculate the area under the ROC curve.**

**The area under the ROC curve is referred to by `caret` as `"ROC"`. The cross-validation averaged performance merics are printed to the screen for you. What is the area under the ROC curve for your logistic regression model?**

*HINT*: You can ignore warnings displayed during the training of the logistic regression model.

**SOLUTION** Train the logistic regression model in the code chunk below.

```
set.seed(4321)
fit_glm_sonar <- train(Class ~ ., data = Sonar, method = 'glm', metric = metric_sonar, trControl = ctrl
```

```
## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## Warning: glm.fit: algorithm did not converge

## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
fit_glm_sonar
```

```
## Generalized Linear Model
##
## 208 samples
##  60 predictor
##   2 classes: 'M', 'R'
##
## Pre-processing: scaled (60), centered (60)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 166, 167, 165, 167, 167
## Resampling results:
##
##   ROC        Sens       Spec
##   0.7690207  0.7747036  0.7036842
```

The ROC is 0.7690207

**5c)**

Since there are relatively few observations based on the number of inputs in `Sonar`, let's apply regularization to the linear additive features. The `glmnet` package will fit a logistic regression model with the elastic net penalty term. The syntax is identical to how you fit the elastic net model in Problem 2h) for the regression problem. You simply need to set `method` to `"glmnet"`. However, for this problem you will use linear additive terms just as you did in Problem 4e), instead of using all triplet interactions.

**PROBLEM Train an elastic net model and tune the hyperparameters with 5-fold cross-validation to maximize the area under the ROC curve. You can use the default tuning grid from `caret` and so you do not need to set the `tuneGrid` argument.**

**Based on the training results, does the elastic net model favor LASSO or RIDGE more?**

**SOLUTION** Train the elastic net model below.

```r
set.seed(4321)
fit_glmnet_sonar <- train(Class ~ ., data = Sonar, method = 'glmnet', preProcess=c('center', 'scale'), m

fit_glmnet_sonar
```

```
## glmnet
##
## 208 samples
##  60 predictor
##   2 classes: 'M', 'R'
##
## Pre-processing: centered (60), scaled (60)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 166, 167, 165, 167, 167
## Resampling results across tuning parameters:
##
##   alpha  lambda        ROC        Sens       Spec
##   0.10   0.0004318733  0.8381527  0.8019763  0.7221053
##   0.10   0.0043187332  0.8567839  0.8118577  0.7526316
##   0.10   0.0431873324  0.8528271  0.7762846  0.7531579
##   0.55   0.0004318733  0.8345278  0.7928854  0.7026316
##   0.55   0.0043187332  0.8543437  0.8118577  0.7426316
##   0.55   0.0431873324  0.8367568  0.8027668  0.7226316
##   1.00   0.0004318733  0.8250947  0.7837945  0.6931579
##   1.00   0.0043187332  0.8511889  0.7940711  0.7321053
##   1.00   0.0431873324  0.8319097  0.8027668  0.7226316
##
## ROC was used to select the optimal model using the largest value.
## The final values used for the model were alpha = 0.1 and lambda = 0.004318733.
```

since alpha of the best model is 0.1, the elastic net model favors RIDGE more.

**5d)**

Let's now try a neural network model with the `nnet` package. You must specify the `method` argument equal to `"nnet"`, just as you did in Problem 3a). However, the rest of the arguments to the `train()` function must be consistent with the binary classification arguments of Problem 5. You can use the default tuning grid, and so you do not need to specify the `tuneGrid` argument to `train()`.

**PROBLEM Train a neural network binary classifier using the `"nnet"` package with `caret`. Does the neural network model achieve a higher area under the ROC curve compared to the elastic net model?**

**SOLUTION** Train the neural network with `"nnet"` below.

```r
set.seed(4321)
fit_nnet_sonar <- train(Class ~ ., data = Sonar, method = 'nnet', preProcess=c('center', 'scale'), metri

fit_nnet_sonar
```

```
## Neural Network
##
## 208 samples
##  60 predictor
##   2 classes: 'M', 'R'
##
## Pre-processing: centered (60), scaled (60)
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 166, 167, 165, 167, 167
## Resampling results across tuning parameters:
##
##   size  decay  ROC        Sens       Spec
##   1     0e+00  0.7778292  0.7316206  0.7547368
##   1     1e-04  0.8127554  0.7403162  0.7547368
##   1     1e-01  0.8423237  0.7675889  0.7226316
##   3     0e+00  0.7896229  0.7675889  0.7542105
##   3     1e-04  0.8533061  0.8031621  0.7526316
##   3     1e-01  0.8659455  0.7948617  0.7115789
##   5     0e+00  0.8013990  0.7494071  0.7631579
##   5     1e-04  0.8471281  0.7766798  0.7415789
##   5     1e-01  0.8859871  0.8122530  0.7736842
##
## ROC was used to select the optimal model using the largest value.
## The final values used for the model were size = 5 and decay = 0.1.
```

No the nnet method does achieve a slightly higher ROC.

**5e)**

Now use a random forest model. You can use the default tuning grid to the `mtry` tuning parameter.

**PROBLEM Train a random forest binary classifer by setting the `method` argument equal to `"rf"`. The code chunk below includes the `importance=TRUE` argument for you.**

**What value of `mtry` was selected as the best, based on the cross-validation results? Why do you think that value was selected?**

**SOLUTION** Train the random forest below.

```
set.seed(4321)
fit_rf_sonar <- train(Class ~ ., data = Sonar, method = 'rf', preProcess=c('center', 'scale'), metric=m

fit_rf_sonar
```

```
## Random Forest
##
## 208 samples
##  60 predictor
##   2 classes: 'M', 'R'
##
## Pre-processing: centered (60), scaled (60)
## Resampling: Cross-Validated (5 fold)
```

```
## Summary of sample sizes: 166, 167, 165, 167, 167
## Resampling results across tuning parameters:
##
##   mtry  ROC        Sens       Spec
##    2    0.9436322  0.9371542  0.7847368
##   31    0.9074516  0.8648221  0.7326316
##   60    0.8921838  0.8466403  0.6910526
##
## ROC was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 2.
```

the best mtry value selected is 2, since the random forest with tree # equals to 2 produces the highest ROC.

**5f)**

Lastly, train a boosted tree model with XGBoost. You must specify the `method` argument to be `"xgbTree"`. You can use the default tuning grid to train the model. Since a lot of tuning parameter combinations are attempted, the results are plotted for you, rather than printing the results to the screen. Also the best tuning parameter values are displayed for you. Note that the `importance=TRUE` flag is set for you.

**PROBLEM Train the XGBoost model with 5-fold cross-validation to maximize the area under the ROC curve.**

*NOTE*: The following code may take a few minutes to complete.

**SOLUTION** Train the boosted tree model below.

```
set.seed(4321)
fit_xgb_sonar <- train(Class ~ ., data = Sonar, method = 'xgbTree', preProcess=c('center', 'scale'), me
```

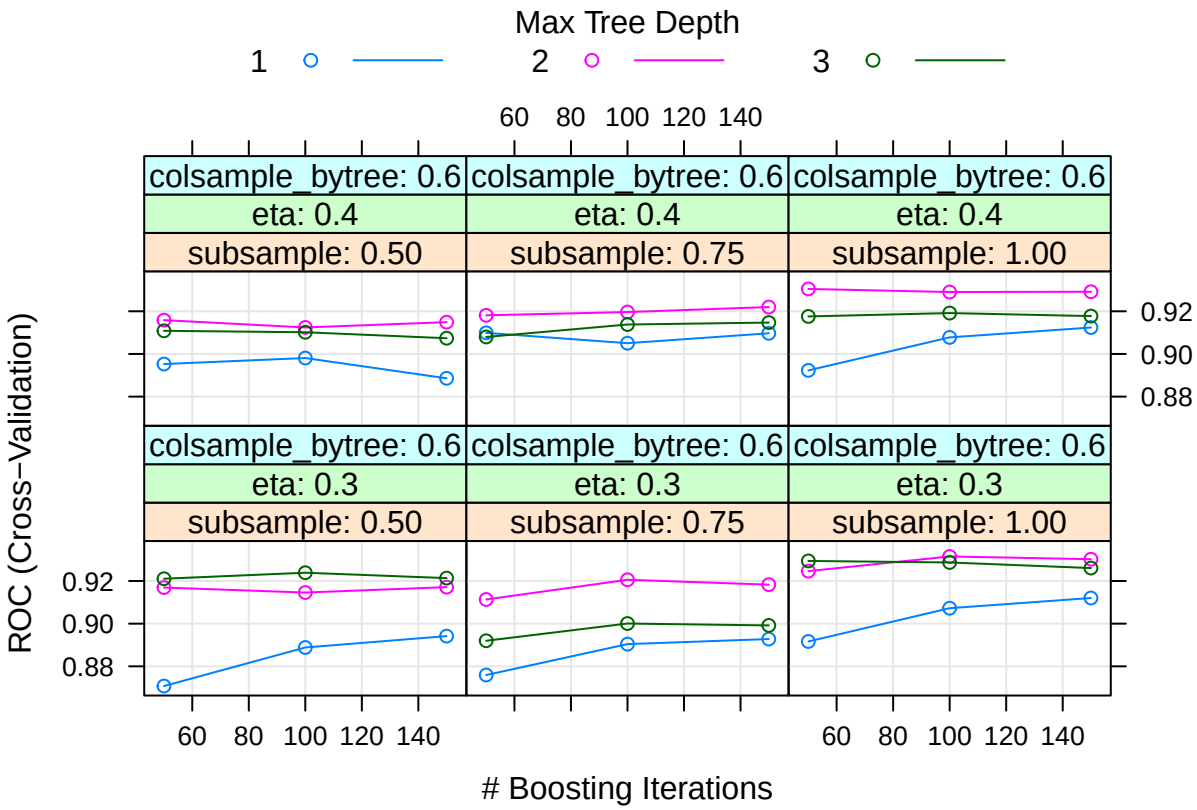
The best identifed tuning parameters are given below.

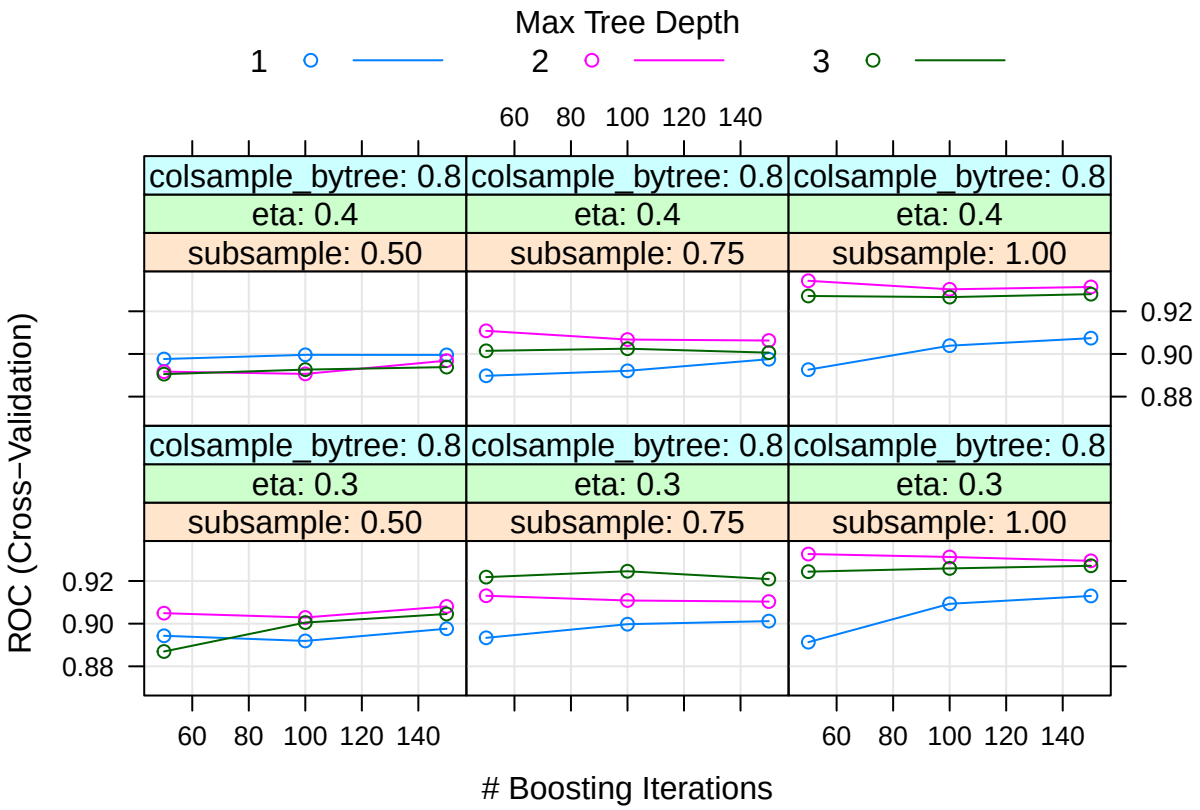
```
fit_xgb_sonar$bestTune
```

```
##    nrounds max_depth eta gamma colsample_bytree min_child_weight subsample
## 88      50         2 0.4     0              0.8                1         1
```

The cross-validation results are printed to the screen for you below.

```
plot(fit_xgb_sonar)
```


Max Tree Depth
1
2
3
ROC (Cross−Validation)
# Boosting Iterations
colsample_bytree: 0.6
eta: 0.4
subsample: 0.50
subsample: 0.75
subsample: 1.00
eta: 0.3
60 80 100 120 140
0.92
0.90
0.88




**5g)**

The resampling results are compiled together using the `resamples()` function. You must complete the first code chunk below by assigning the model object to the corresponding name in the list. For example, you must set the `fit_glm_sonar` object to the `GLM` variable in the list.

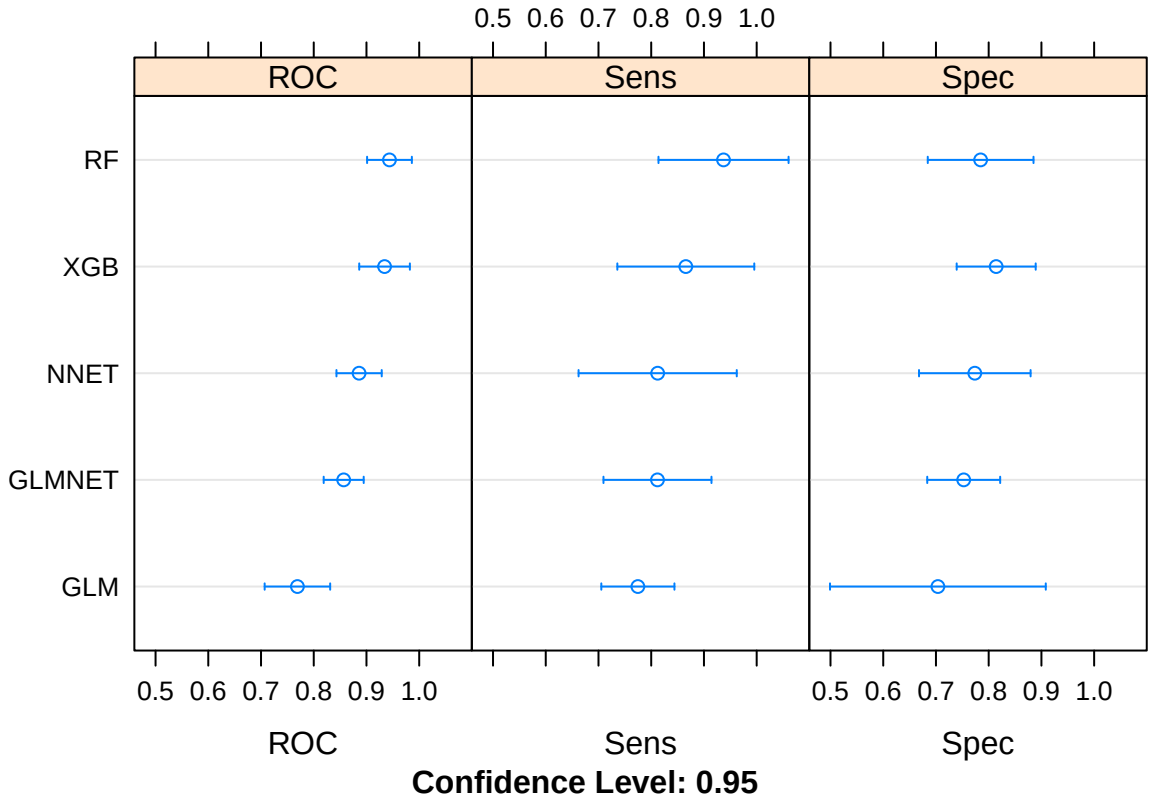
**PROBLEM Complete the first code chunk below, by assigning the `caret` trained model objects to their appropriate variables in the list.**

**The results are then plotted for you using `dotplot()` which model is the best?**

**SOLUTION** Complete the list below. Discuss which model is the best.

```
sonar_roc_compare <- resamples(list(GLM = fit_glm_sonar ,
                                    GLMNET = fit_glmnet_sonar,
                                    NNET = fit_nnet_sonar,
                                    RF = fit_rf_sonar,
                                    XGB = fit_xgb_sonar))

dotplot(sonar_roc_compare)
```




Since our metric is the area under ROC, we will consider the random forest method as the best.

## Problem 06

It can be a little challenging to visualize prediction trends of the binary outcome with the Sonar data set because there are just so many inputs. However, we can use the variable importance rankings to consider which inputs to focus on. The `importance` argument was set to TRUE for you in the tree based methods for you. Regardless of which model you found to be the best, plot the variable importance rankings for the random forest model. After inspecting the variable importance rankings you will make predictions on a test set.

### 6a)

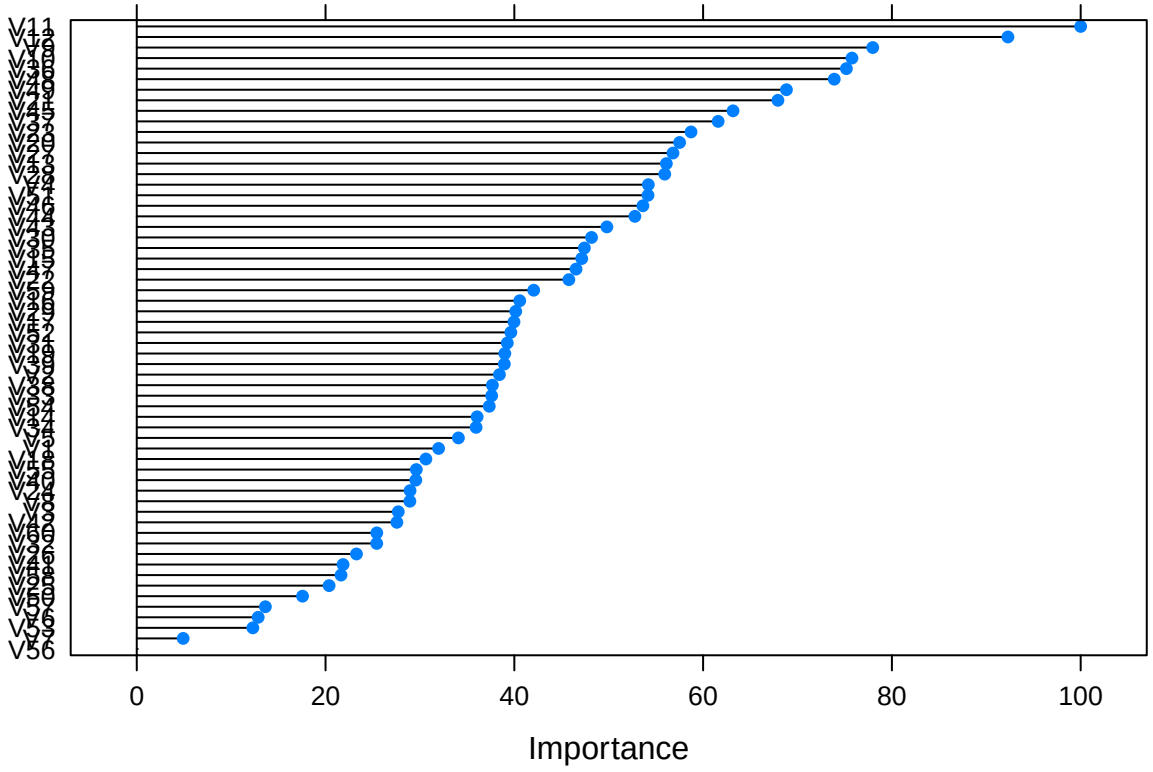
There are a lot of inputs to the Sonar data set. Thus, the variable importance rankings plot might be very cluttered and difficult to read. After plotting it in the first code chunk below, plot the variable importance rankings again, but this time set the `top` argument to 20 to focus just on the top 20 ranked inputs. The syntax is `plot(varImp(<caret model>), top = <number of show>)`.

**PROBLEM Plot the variable importance rankings for all inputs based on the random forest model in the first code chunk below. Then plot the variable importance rankings for just the top 20 inputs in the second code chunk.**

**How many inputs are within 70% of the importance of the top ranked input?**

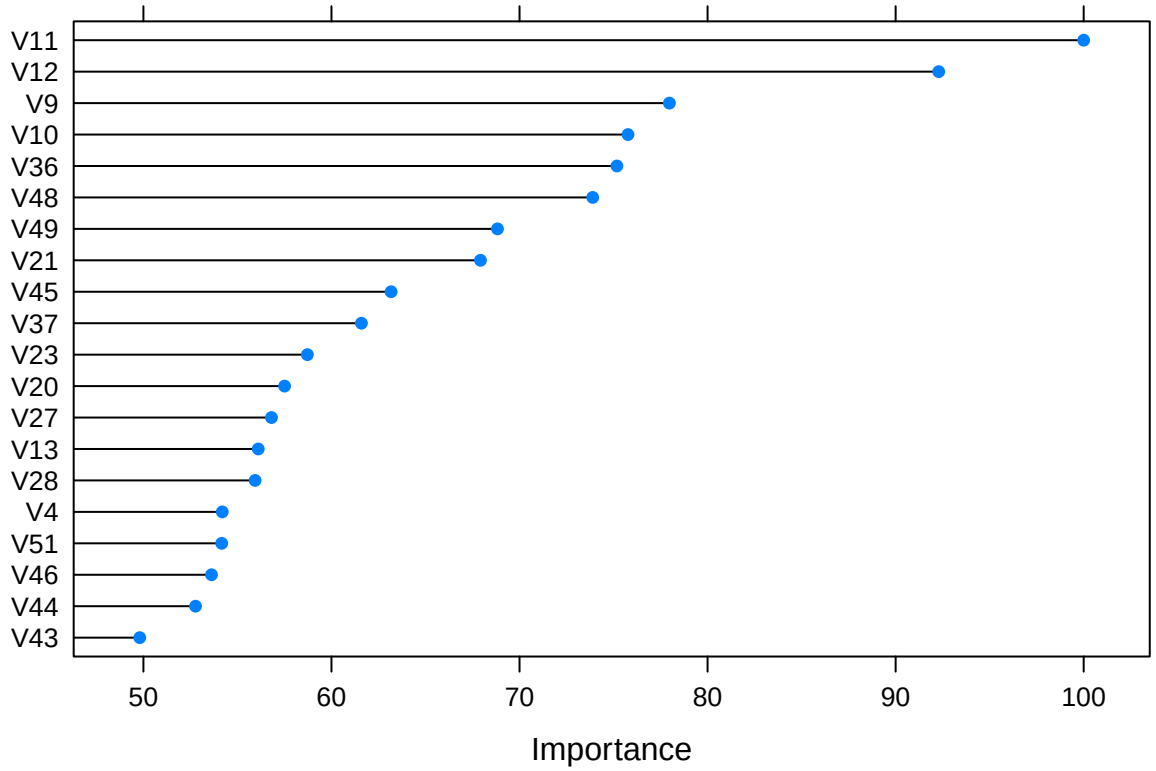
**SOLUTION** Plot the variable importance rankings for all inputs below.

```
plot(varImp(fit_rf_sonar))
```




Plot the top 20 ranked inputs below.

```
plot(varImp(fit_rf_sonar), 20)
```




6 inputs have importance higher than 70% of the top-ranked input's importance.

**6b)**

We will now make predictions over a test grid. The grid is created for you in the code chunk below. Regardless of the top ranked variables you identified in Problem 6a), you will make predictions to study the behavior with respect to `V11`, `V12`, `V9`, and `V10`.

The code chunk creates the test grid in a few steps. The `make_sonar_input_grid()` function checks to see if an input is one of the 4 inputs we wish to visualize trends over. If it is one of the first 2 listed variables in the `viz_input_names` argument, then 25 evenly spaced values over bounds of the Sonar data set are created. If the variable is one of the last two listed names in `viz_input_names`, then 5 quantiles of that variable from the Sonar data set are calculated. Otherwise, the input is set to a single value equal to the median value from the Sonar data set.

The `make_sonar_input_grid()` is applied to all inputs in the Sonar data set by iterating with the `purrr::map()` function. The result is a list with elements containing specific values to use for each input. That list is then passed into the `expand.grid()` function in order to create the grid of input values to make predictions with.

```
### make a test input grid to visualize trends
make_sonar_input_grid <- function(var_name, viz_input_names, all_data)
{
  xvar <- all_data %>% select(var_name) %>% pull()

  if (var_name %in% viz_input_names[1:2]){
    # use 25 unique values
```

```
    xgrid <- seq(min(xvar), max(xvar), length.out = 25)

  } else if (var_name %in% viz_input_names[3:4]){
    # use specific quantiles
    xgrid <- quantile(xvar,
                      probs = c(0.05, 0.25, 0.5, 0.75, 0.95),
                      na.rm = TRUE)
    xgrid <- as.vector(xgrid)
  } else {
    # set to median
    xgrid <- median(xvar, na.rm = TRUE)
  }

  return(xgrid)
}

all_input_names <- Sonar %>%
  select(-Class) %>%
  names()

### inputs to visualize
viz_sonar_names <- c("V11", "V12", "V9", "V10")

### create the list of the inputs
test_sonar_list <- purrr::map(all_input_names,
                              make_sonar_input_grid,
                              viz_input_names = viz_sonar_names,
                              all_data = Sonar)
```

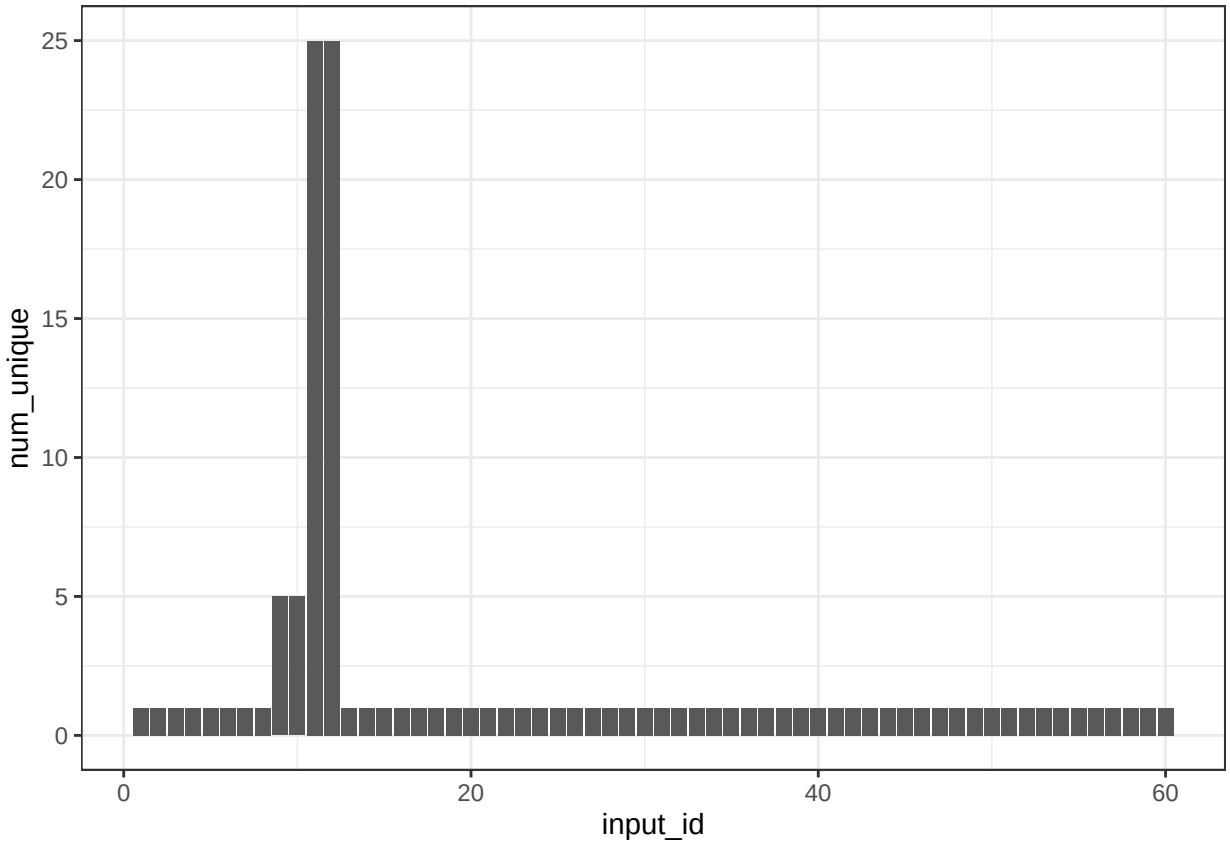
```
## Note: Using an external vector in selections is ambiguous.
## i Use `all_of(var_name)` instead of `var_name` to silence this message.
## i See <https://tidyselect.r-lib.org/reference/faq-external-vector.html>.
## This message is displayed once per session.
```

```
### use the list of inputs to create the prediction grid
test_sonar_grid <- expand.grid(test_sonar_list,
                               KEEP.OUT.ATTRS = FALSE,
                               stringsAsFactors = FALSE) %>%
  purrr::set_names(all_input_names)
```

To help understand what happened in the code chunk above, the number of unique values per input is displayed for each input in the `test_sonar_grid` object with the bar chart below. As you should see below, the 11th and 12th inputs have 25 unique values. The 9th and 10th inputs have 5 unique values. All other inputs have just a single unique value. You will therefore study the trends associated with the 25 by 25 grid of points in `V11` and `V12` for 25 different combinations of `V9` and `V10`, with all other inputs set to their median value from the Sonar data set.

```
test_sonar_grid %>%
  tibble::rowid_to_column("pred_id") %>%
  tidyr::gather(key = "key", value = "value", -pred_id) %>%
  group_by(key) %>%
  summarise(num_rows = n(),
```

```
          num_unique = n_distinct(value)) %>%
ungroup() %>%
mutate(input_id = as.numeric(stringr::str_extract(key, "\\d+"))) %>%
ggplot(mapping = aes(x = input_id,
                     y = num_unique)) +
geom_bar(stat = "identity",
         mapping = aes(group = input_id)) +
theme_bw()
```




Predictions from binary classification models can be performed in one of two ways. The default prediction method is to return the predicted classification based on a threshold value of 0.5. The syntax is identical to making predictions with the regression models. You specify the first argument to the `predict()` function to be the `caret` model object, and the second argument as the prediction or test data set. The result will be a factor (categorical variable) with levels equal to the levels of the discrete outcome from the model.

Make predictions with the random forest and elastic net models. The `head()` function is used to print out the first few predictions so you can see what the predicted classifications look like. The `class()` of the variables are also shown to you.

**PROBLEM Make predictions with the random forest and elastic net models on the provided test data set `test_sonar_grid`. Assign the predictions for the elastic net model to the `pred_test_sonar_glmnet` variable and the predictions for the random forest model to the `pred_test_sonar_rf` variable.**

```
pred_test_sonar_glmnet <- predict(fit_glmnet_sonar, test_sonar_grid)

pred_test_sonar_rf <- predict(fit_rf_sonar, test_sonar_grid)
```

**SOLUTION** The class (data type) for the predictions.

```
class(pred_test_sonar_rf)
```

```
## [1] "factor"
```

The first few predictions from the random forest model are shown below.

```
pred_test_sonar_rf %>% head()
```

```
## [1] R R R R R R
## Levels: M R
```

**6c)**

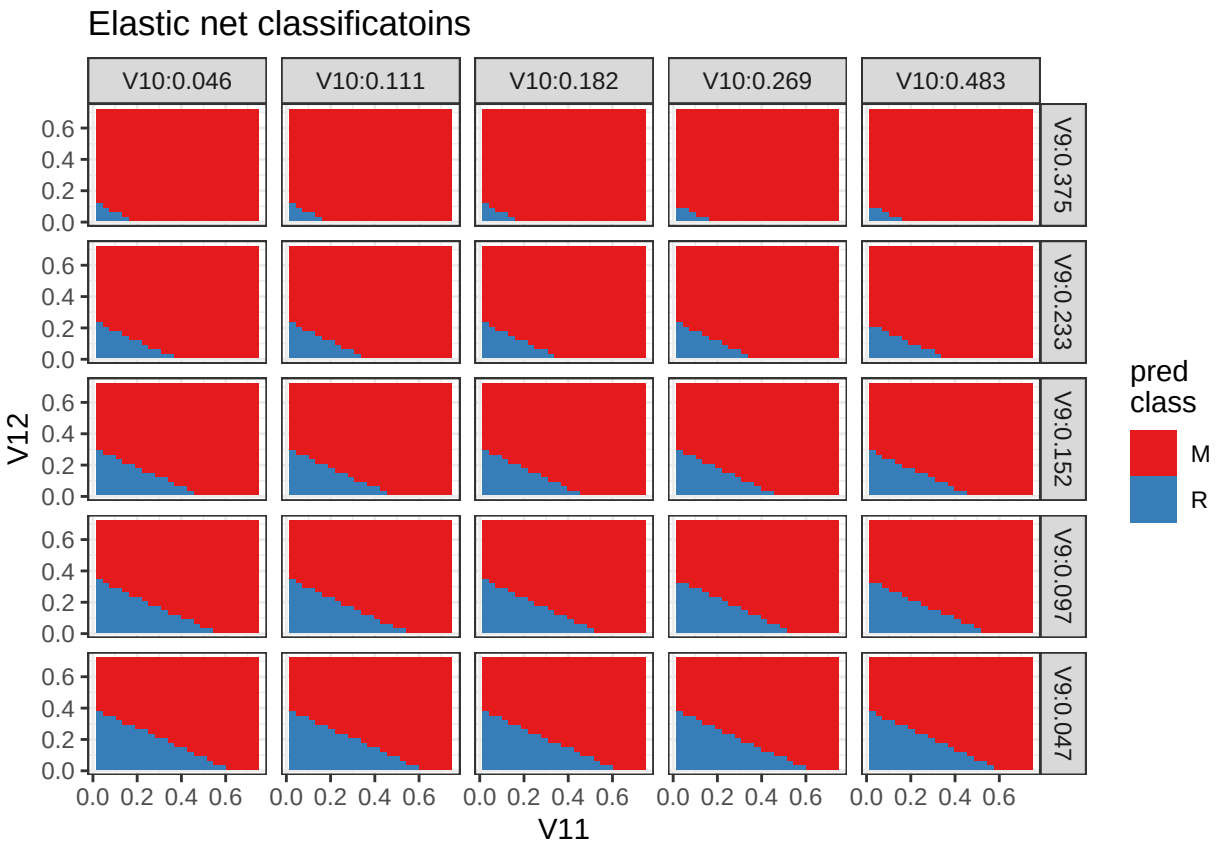
Let's now visualize the predicted classifications over the test grid. Visualizing the predicted classification *surface* takes a little extra effort than visualizing the predicted regression trends. Thus, the two code chunks are completed for you below. The `geom_raster()` function is used to visualize the surface of classifications between the two possible levels, `"M"` and `"R"`. Red areas correspond to predicted classifications of metal and blue areas correspond to predicted classifications of rock. Each suface corresponds to the 25 by 25 grid of points between `V11` and `V12` and each facet (subplot) is a combination of `V9` and `V10`. There are a few extra steps in the code chunks to help make the labels easy to read for the subplots.

The first code chunk visualizes the predicted classifications from the elastic net model and the second code chunk visualizes the predicted classifications from the random forest model.

**PROBLEM Discuss the behavior in the classifications between the two models. Essentially why do elastic net predicted surfaces look the way they do compared to the random forest classification surfaces? Are there any similarities between the two predictions?**

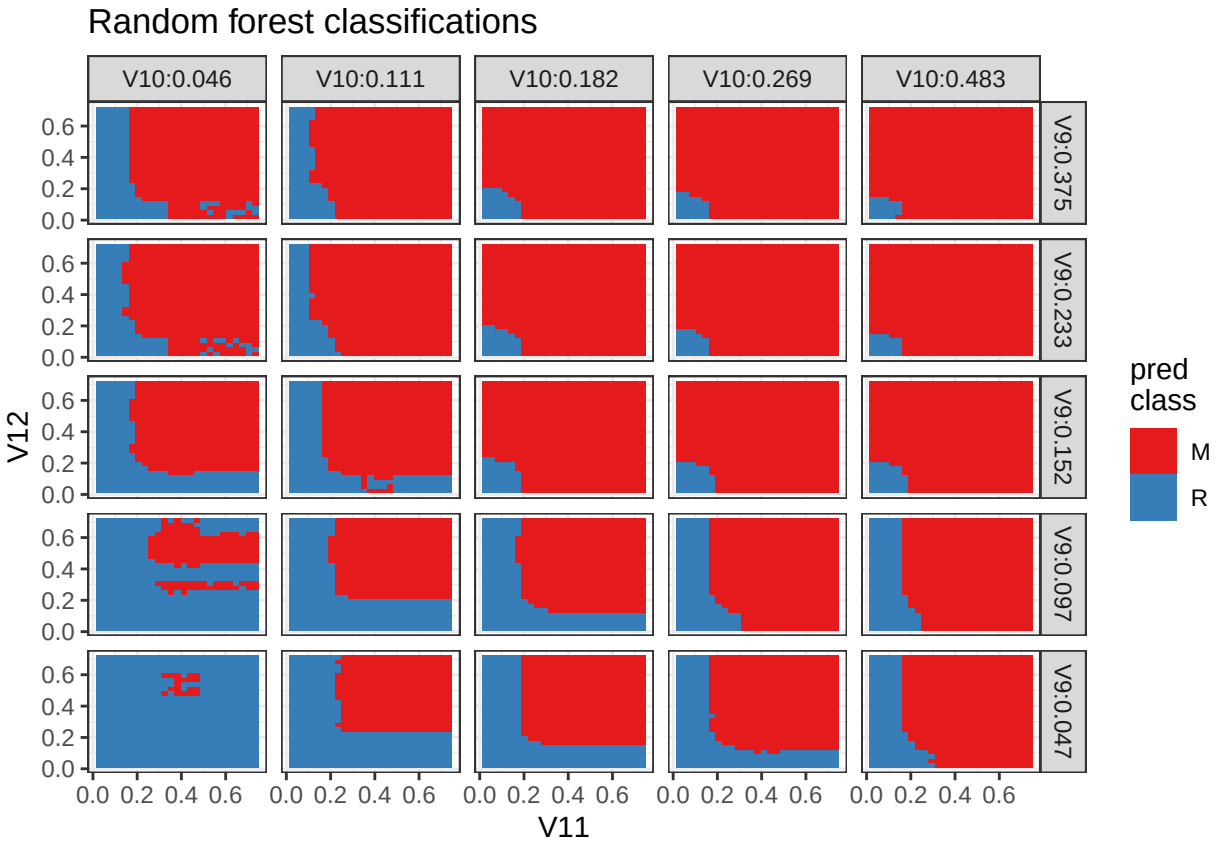
**SOLUTION** Elastic net predicted classifications are shown below.

```
test_sonar_grid %>%
  mutate(pred_class = pred_test_sonar_glmnet) %>%
  mutate(view_V9 = forcats::fct_rev(sprintf("V9:%1.3f", V9))) %>%
  ggplot(mapping = aes(x = V11, y = V12)) +
  geom_raster(mapping = aes(fill = pred_class)) +
  facet_grid(view_V9 ~ V10,
             labeller = label_bquote(rows = .(view_V9),
                                     cols = .(sprintf("V10:%1.3f", V10)))) +
  scale_fill_brewer("pred\nclass", palette = "Set1") +
  labs(title = "Elastic net classificatoins") +
  theme_bw()
```




Random forest predicted classifications are shown below.

```
test_sonar_grid %>%
  mutate(pred_class = pred_test_sonar_rf) %>%
  mutate(view_V9 = forcats::fct_rev(sprintf("V9:%1.3f", V9))) %>%
  ggplot(mapping = aes(x = V11, y = V12)) +
  geom_raster(mapping = aes(fill = pred_class)) +
  facet_grid(view_V9 ~ V10,
             labeller = label_bquote(rows = .(view_V9),
                                     cols = .(sprintf("V10:%1.3f", V10)))) +
  scale_fill_brewer("pred\nclass", palette = "Set1") +
  labs(title = "Random forest classifications") +
  theme_bw()
```




Since the elastic net method has certain cross-validation between two coefficients, the function of the two coefficients should look relatively more linear. Random forest on the other hand, is a tree-based method so it is more vulnerable to overfitting. And thus its curve looks more arbitrary. However, there is a noticeable similarity between the two methods as a lot the subplots suggest that their functions denote the same trend of the relation between the coefficients, which is that V12 is generally inversely proportional to V11.

**6d)**

We might be interested in the predicted probability per level of `Class`, in addition to just a classification. Probability predictions are still made the with `predict()` function, but we must include a third argument which tells the model to return the predicted probability per `Class` level. The syntax is:

```
predict(<model object>, <new data>, type = "prob")
```

The result will no longer be a "regular" vetor, but will be a data.frame. Each column in the data.frame is named for the `levels` of the outcome `Class`. The data type for the random forest based predictions and the first few rows of the predictions are printed to the screen, to show you the structure of the objects.

**PROBLEM Make predictions with the random forest and elastic net models on the provided test data set `test_sonar_grid` and return the probabilities rather than the classifications. Assign the predictions for the elastic net model to the `pred_test_prob_sonar_glmnet` variable and the predictions for the random forest model to the `pred_test_prob_sonar_rf` variable.**

**SOLUTION** Make predictions for the class probabilities for the elastic net and random forest models.

```
pred_test_prob_sonar_glmnet <- predict(fit_glmnet_sonar, test_sonar_grid, type = 'prob')

pred_test_prob_sonar_rf <- predict(fit_rf_sonar, test_sonar_grid, type = 'prob')
```

The random forest predicted probabilities are shown in the code chunks below.

```
pred_test_prob_sonar_rf %>% class()
```

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

```
pred_test_prob_sonar_rf %>% head()
```

```
##       M     R
## 1 0.338 0.662
## 2 0.344 0.656
## 3 0.394 0.606
## 4 0.412 0.588
## 5 0.412 0.588
## 6 0.346 0.654
```

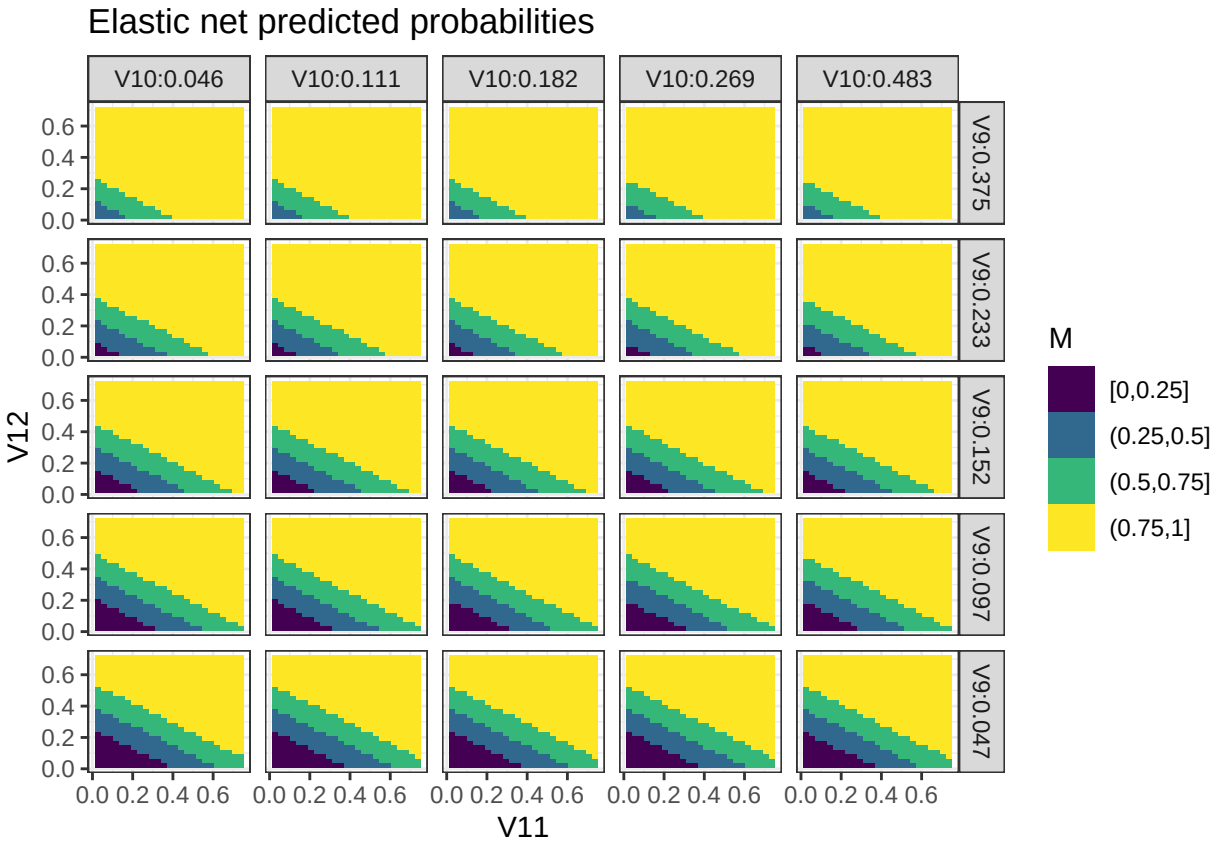
**6e)**

The predicted class probabilities can now be visualized in a surface, similar to the classification surface. The predicted probabilities of the `"M"` level are shown discretized into interval of 0.25 to help make easier to focus on low, medium, and high predicted probabilities. The surfaces are created for you for both sets of predictions.

**PROBLEM Discuss how the predicted probabilities for the `"M"` level are consistent with the classification surfaces visualized in Problem 6c).**

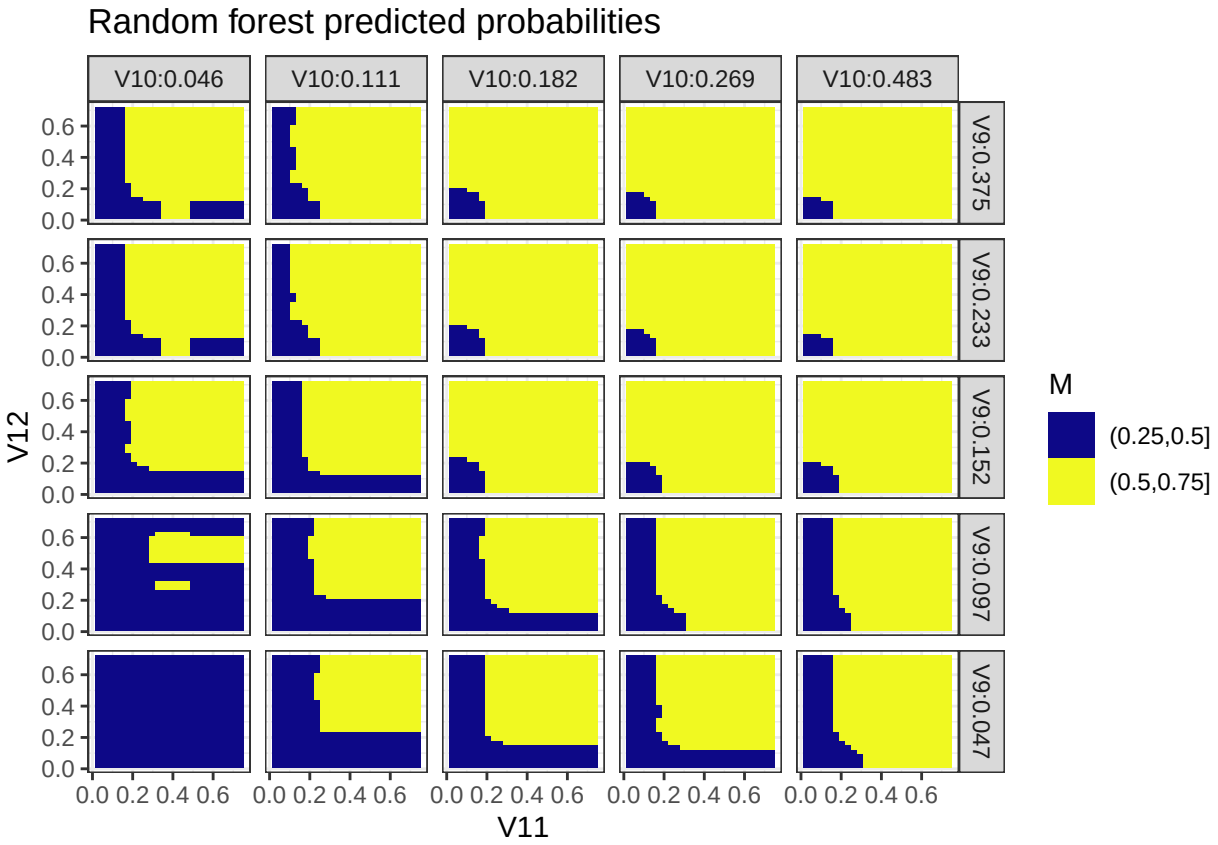
**SOLUTION** Elastic net predicted probabilities are shown below.

```
test_sonar_grid %>%
  bind_cols(pred_test_prob_sonar_glmnet) %>%
  mutate(view_V9 = forcats::fct_rev(sprintf("V9:%1.3f", V9))) %>%
  ggplot(mapping = aes(x = V11, y = V12)) +
  geom_raster(mapping = aes(fill = cut(M,
                                       breaks = seq(0, 1, by = 0.25),
                                       include.lowest = TRUE))) +
  facet_grid(view_V9 ~ V10,
             labeller = label_bquote(rows = .(view_V9),
                                     cols = .(sprintf("V10:%1.3f", V10)))) +
  scale_fill_viridis_d("M") +
  labs(title = "Elastic net predicted probabilities") +
  theme_bw()
```




The random forest predicted probabilities are shown below.

```
test_sonar_grid %>%
  bind_cols(pred_test_prob_sonar_rf) %>%
  mutate(view_V9 = forcats::fct_rev(sprintf("V9:%1.3f", V9))) %>%
  ggplot(mapping = aes(x = V11, y = V12)) +
  geom_raster(mapping = aes(fill = cut(M,
                                       breaks = seq(0, 1, by = 0.25),
                                       include.lowest = TRUE))) +
  facet_grid(view_V9 ~ V10,
             labeller = label_bquote(rows = .(view_V9),
                                     cols = .(sprintf("V10:%1.3f", V10)))) +
  scale_fill_viridis_d("M", option = "plasma") +
  labs(title = "Random forest predicted probabilities") +
  theme_bw()
```




Take the elastic net probabilities surface for example, the green and yellow areas of the surface should denote a high possibility for M, thus we should classify the green and yellow to M. And if we confirm this theory of with the classification graph we can see the the yellow and green areas in the elastic net graph are assigned to M label and the rest of area is assigned to R.

## BONUS

You have two bonus questions you can try to earn extra credit. These questions are not required to be completed.

### BONUS - 1 (10 points)

You trained all of the `caret` models with the default tuning grids. However, we have seen in lecture how to use the `tuneGrid` argument to try out custom search grids, in an attempt to improve model performance. Try to tune the regression models to get better performance than what you had in Problem 3.

You may add as many code chunks as you want. When you tune a model, create a new variable for that trained object. For example, if you would try and tune the elastic net model with all triplet interactions, name the tuned model `fit_01_glmnet_trips_tune` to make sure it's a different variable than the default result.

### BONUS - 2 (15 points)

You trained models to maximize the area under the ROC curve. But we did not visualize the ROC curves for the models. Example code provided in lecture and earlier in the semester demonstrated how to use the

`plotROC` package to create ROC curves, which average over the cross-validation results. Compare 2 of the classification models by comparing their ROC curves averaged over the cross-validation results. Then show the 2 models as separate subplots to allow showing the ROC curve associated with each fold.

You will need to access the saved hold-out set predictions to create the ROC curve. The predictions are contained in the `$pred` field of the `caret` object. Thus, to the elastic net hold-out set predictions are in `fit_glmnet_sonar$pred`. Remember that the hold-out set predictions include ALL tuning parameter combinations. So you will need to filter to the best tuned parameter values.