Project 2: Modeling and Evaluation

CSE6242 - Data and Visual Analytics - Fall 2017

Due: Sunday, November 26, 2017 at 11:59 PM UTC-12:00 on T-Square

Vincent La (Georgia Tech ID - vla6)

Data

We will use the same dataset as Project 1: `movies_merged`.

Objective

Your goal in this project is to build a linear regression model that can predict the `Gross` revenue earned by a movie based on other variables. You may use R packages to fit and evaluate a regression model (no need to implement regression yourself). Please stick to linear regression, however.

Instructions

You should be familiar with using an RMarkdown Notebook by now. Remember that you have to open it in RStudio, and you can run code chunks by pressing *Cmd+Shift+Enter*.

Please complete the tasks below and submit this R Markdown file (as **pr2.Rmd**) containing all completed code chunks and written responses, and a PDF export of it (as **pr2.pdf**) which should include the outputs and plots as well.

*Note that **Setup** and **Data Preprocessing** steps do not carry any points, however, they need to be completed as instructed in order to get meaningful results.*

Setup

Same as Project 1, load the dataset into memory:

```
setwd("~/git/GeorgiaTech/cse6242/pr2")
load('movies_merged')
```

This creates an object of the same name (`movies_merged`). For convenience, you can copy it to `df` and start using it:

```
df = movies_merged
cat("Dataset has", dim(df)[1], "rows and", dim(df)[2], "columns", end="\n", file="")
```

```
## Dataset has 40789 rows and 39 columns
```

```
colnames(df)
```

```
##  [1] "Title"             "Year"              "Rated"
##  [4] "Released"          "Runtime"           "Genre"
##  [7] "Director"          "Writer"            "Actors"
## [10] "Plot"              "Language"          "Country"
```

```
## [13] "Awards"            "Poster"            "Metascore"
## [16] "imdbRating"        "imdbVotes"         "imdbID"
## [19] "Type"              "tomatoMeter"       "tomatoImage"
## [22] "tomatoRating"      "tomatoReviews"     "tomatoFresh"
## [25] "tomatoRotten"      "tomatoConsensus"   "tomatoUserMeter"
## [28] "tomatoUserRating"  "tomatoUserReviews" "tomatoURL"
## [31] "DVD"               "BoxOffice"         "Production"
## [34] "Website"           "Response"          "Budget"
## [37] "Domestic_Gross"    "Gross"             "Date"
```

Load R packages

Load any R packages that you will need to use. You can come back to this chunk, edit it and re-run to load any additional packages later.

```
library(GGally)  # Used for ggpairs
library(ggplot2)
library(lubridate)  # Used to extract year from Released
```

```
##
## Attaching package: 'lubridate'
```

```
## The following object is masked from 'package:base':
##
##     date
```

```
library(qdapTools)  # Used for mtabulate (to one-hot encode a column)
library(reshape)  # Used for melt function
```

```
##
## Attaching package: 'reshape'
```

```
## The following object is masked from 'package:lubridate':
##
##     stamp
```

```
library(stringi)  # Used for processing Awards
```

If you are using any non-standard packages (ones that have not been discussed in class or explicitly allowed for this project), please mention them below. Include any special instructions if they cannot be installed using the regular `install.packages('<pkg name>')` command.

Non-standard packages used: None

Data Preprocessing

Before we start building models, we should clean up the dataset and perform any preprocessing steps that may be necessary. Some of these steps can be copied in from your Project 1 solution. It may be helpful to print the dimensions of the resulting dataframe at each step.

1. Remove non-movie rows

```
# TODO: Remove all rows from df that do not correspond to movies
df <- df[df$Type == "movie",]
```

2. Drop rows with missing `Gross` value

Since our goal is to model `Gross` revenue against other variables, rows that have missing `Gross` values are not useful to us.

```
# TODO: Remove rows with missing Gross value
df = df[which(!is.na(df$Gross)), ]
```

3. Exclude movies released prior to 2000

Inflation and other global financial factors may affect the revenue earned by movies during certain periods of time. Taking that into account is out of scope for this project, so let's exclude all movies that were released prior to the year 2000 (you may use `Released`, `Date` or `Year` for this purpose).

```
# TODO: Exclude movies released prior to 2000
df$released_year = year(df$Released)
df = df[which(df$released_year >= 2000), ]
```

4. Eliminate mismatched rows

Note: You may compare the ***`Released`*** *column (string representation of release date) with either* ***`Year`*** *or* ***`Date`*** *(numeric representation of the year) to find mismatches. The goal is to avoid removing more than 10% of the rows.*

```
# TODO: Remove mismatched rows
# This is just taken from my work in pr1; only drop if the difference in the years is 2 or greater.
df$not_matched = (abs(df$Year != df$released_year) >= 2)

# Number of rows with non-null Gross Value
print(nrow(df))
```

```
## [1] 3325
```

```
print(nrow(df[which(!is.na(df$Gross)), ]))
```

```
## [1] 3325
```

```
# Remove mismatched rows
df = df[which(!(df$not_matched == TRUE)), ]
print(nrow(df))
```

```
## [1] 3325
```

5. Drop `Domestic_Gross` column

`Domestic_Gross` is basically the amount of revenue a movie earned within the US. Understandably, it is very highly correlated with `Gross` and is in fact equal to it for movies that were not released globally. Hence, it should be removed for modeling purposes.

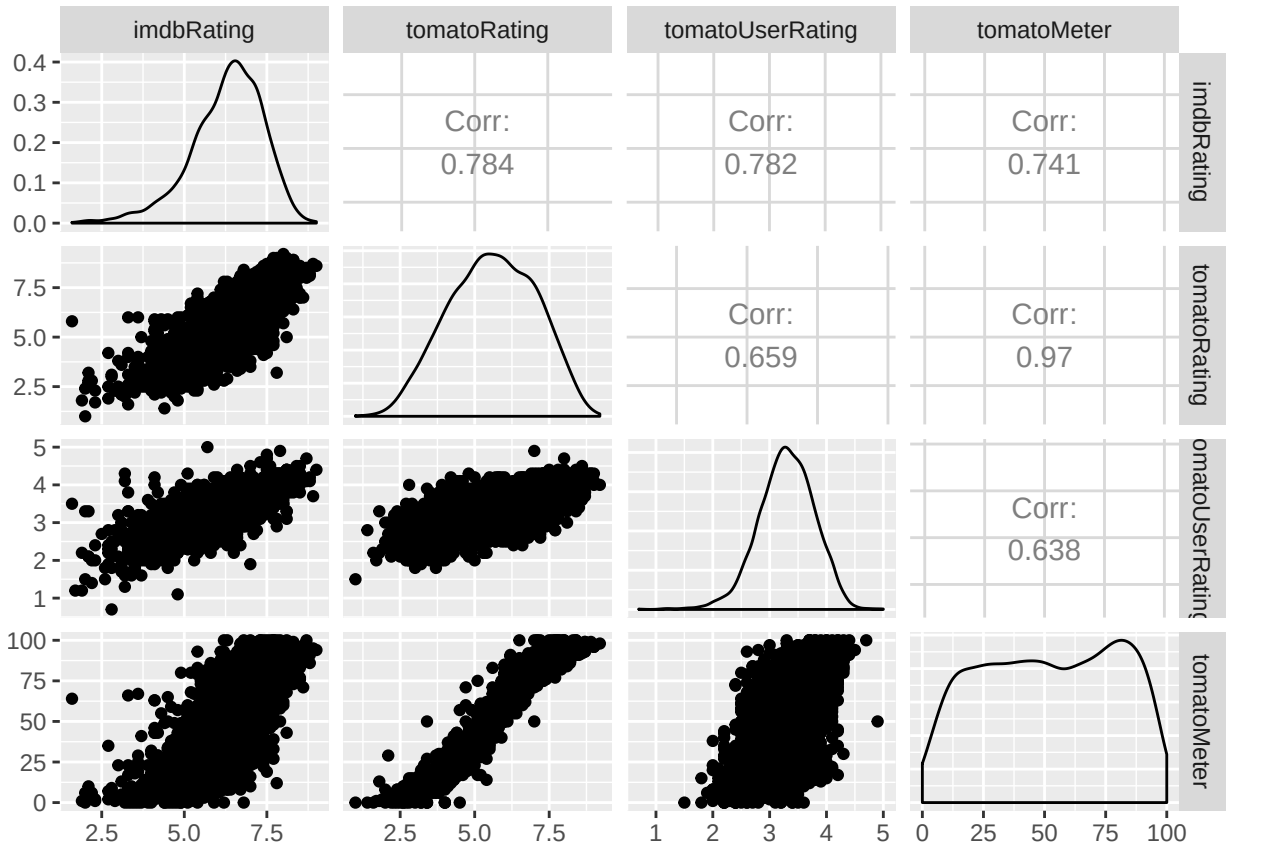
```
# TODO: Exclude the `Domestic_Gross` column
df$Domestic_Gross = NULL
```

6. Process `Runtime` column

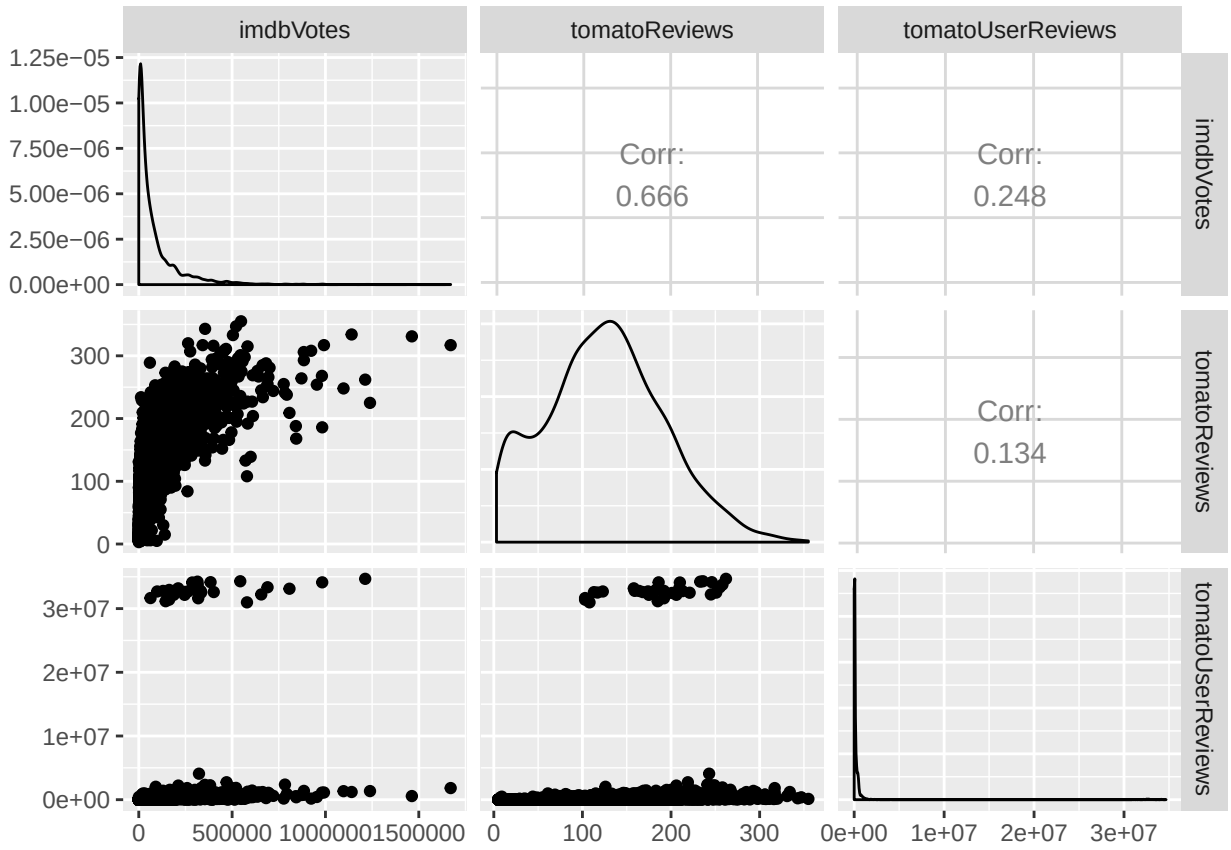
```
# TODO: Replace df$Runtime with a numeric column containing the runtime in minutes
# Taking work from pr1
extract_runtime = function(r){
    times = unlist(r)
    minutes = 0
    for (i in 1:length(times) - 1){
        if (times[i + 1] == 'h'){
            minutes = minutes + as.numeric(times[i]) * 60
        } else if (times[i + 1] == 'min'){
            minutes = minutes + as.numeric(times[i])
        }
    }
    if (minutes == 0){
        return(NA)
    } else{
        return(minutes)
    }
}

y=strsplit(df$Runtime,' ')
new_runtimes = unlist(lapply(y, extract_runtime))
df$Runtime = new_runtimes
```

Perform any additional preprocessing steps that you find necessary, such as dealing with missing values or highly correlated columns (feel free to add more code chunks, markdown blocks and plots here as necessary).

```
# TODO(optional): Additional preprocessing

# Plotting Correlation Plots of the different Rating Columns:
ggpairs(df[c('imdbRating', 'tomatoRating', 'tomatoUserRating', 'tomatoMeter')])
```




```
# Plotting Correlation Plots of the different Review number Columns:
ggpairs(df[c('imdbVotes', 'tomatoReviews', 'tomatoUserReviews')])
```




```
# Remove tomato-like Columns to avoid high variance in models.
df$tomatoMeter = NULL
df$tomatoRating = NULL
df$tomatoReviews = NULL
df$tomatoFresh = NULL
df$tomatoRotten = NULL
df$tomatoUserMeter = NULL
df$tomatoUserRating = NULL
df$tomatoUserReviews = NULL

# Remove Year since we will use released_year column
df$Year = NULL

# Also creating categorical value for runtime, just as we did in Project 1 to explore relationships
df$runtime_categorical = ifelse(df$Runtime <= 68, 'short', ifelse(df$Runtime <= 101, 'medium', 'long'))
```

From the above, notice that `imdbRating` is very highly correlated with `tomatoRating`, `tomatoUserRating` and `tomatoMeter`. Thus, to avoid high variance in the models, it makes more sense to just include `imdbRating` as a proxy for the rating of the movie.

When plotting a similar plot for user reviews, it again, it looks like `imdbVotes` is highly correlated with `tomatoReviews`. Furthermore, while `imdbVotes` is not as correlated with `tomatoUserReviews`, there does seem to be still sufficient linear correlation. Based on these two plots, in this assignment, we're going to remove all the 'tomato' columns to avoid unnecessary potentially high variance in models (ala bias-variance tradeoff). It seems that both `imdbRating` and `imdbVotes` capture enough information along these dimensions.

We will also remove movies released in 2017. Notice that if we look at the distribution of movies by year:

```r
table(df$released_year)
```

```
##
## 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014
##  172  173  186  184  183  189  223  205  220  197  221  226  209  205  224
## 2015 2016 2017
##  207   97    4
```

notice that only 4 movies were released in 2017 in our data set. This is probably just a consequence of when this data was pulled. It probably is not representative to include this data, since this is not actually representative of movies in 2017. Thus, we will remove these 4 observations

```r
df = df[which(df$released_year < 2017), ]
```

***Note**: Do NOT convert categorical variables (like `Genre`) into binary columns yet. You will do that later as part of a model improvement task.*

Final preprocessed dataset

Report the dimensions of the preprocessed dataset you will be using for modeling and evaluation, and print all the final column names. (Again, `Domestic_Gross` should not be in this list!)

```r
# TODO: Print the dimensions of the final preprocessed dataset and column names
cat("Dataset has", dim(df)[1], "rows and", dim(df)[2], "columns", end="\n", file="")
```

```
## Dataset has 3321 rows and 32 columns
```

```r
colnames(df)
```

```
##  [1] "Title"              "Rated"              "Released"
##  [4] "Runtime"            "Genre"              "Director"
##  [7] "Writer"             "Actors"             "Plot"
## [10] "Language"           "Country"            "Awards"
## [13] "Poster"             "Metascore"          "imdbRating"
## [16] "imdbVotes"          "imdbID"             "Type"
## [19] "tomatoImage"        "tomatoConsensus"    "tomatoURL"
## [22] "DVD"                "BoxOffice"          "Production"
## [25] "Website"            "Response"           "Budget"
## [28] "Gross"              "Date"               "released_year"
## [31] "not_matched"        "runtime_categorical"
```

Evaluation Strategy

In each of the tasks described in the next section, you will build a regression model. In order to compare their performance, you will compute the training and test Root Mean Squared Error (RMSE) at different training set sizes.

First, randomly sample 10-20% of the preprocessed dataset and keep that aside as the **test set**. Do not use these rows for training! The remainder of the preprocessed dataset is your **training data**.

Now use the following evaluation procedure for each model:

- Choose a suitable sequence of training set sizes, e.g. 10%, 20%, 30%, ..., 100% (10-20 different sizes should suffice). For each size, sample that many inputs from the training data, train your model, and compute the resulting training and test RMSE.

- Repeat your training and evaluation at least 10 times at each training set size, and average the RMSE results for stability.
- Generate a graph of the averaged train and test RMSE values as a function of the train set size (%), with optional error bars.

You can define a helper function that applies this procedure to a given set of features and reuse it.

```
# Setting up Training and Test DF
split_train_test_df = function(df, train_split=0.8){
  # Given a DataFrame, randomly split to training and test DataFrame
  # Now Selecting 80% of data as sample from total 'n' rows of the data
  set.seed(100)
  sample <- sample.int(n = nrow(df), size = floor(.8*nrow(df)), replace = F)
  train_df <- df[sample, ]
  test_df  <- df[-sample, ]
  return(list('train_df'=train_df,
              'test_df'=test_df,
              'train_idx'=sample))
}

split_df = split_train_test_df(df, train_split=0.8)
train_df = split_df$train_df
test_df = split_df$test_df
train_idx = split_df$train_idx

# Create training set size vector
train_sizes = c(0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1)
```

```
# Helper Functions
getrmse = function(mod, test_df){
    # Helper function to return both the training RMSE and test RMSE given model and test DataFrame
    # From https://stackoverflow.com/questions/43123462/how-to-obtain-rmse-out-of-lm-result
    train_rss = c(crossprod(mod$residuals))
    train_mse = train_rss / length(mod$residuals)
    train_rmse = sqrt(train_mse)

    test_rmse = sqrt(mean((test_df$Gross - predict.lm(mod, test_df)) ^ 2, na.rm=TRUE))

    return(list('train_rmse'=train_rmse,
                'test_rmse'=test_rmse))
}
```

```
# Helper function to build and evaluate model according to the instructions provided
eval_model = function(train_df, test_df, model_exp, train_sizes, num_repeat=10){
  # Keyword Args:
  #  train_df: Training DataFrame
  #  test_df: Test DataFrame
  #  model_exp: Expression of linear regression model as a string
  #  num_repeat: Number of times to repeat evaluating the model (since there might be high variance)
  # Returns: df_eval which is a DataFrame with test and train rmse
  train_rmse_all = c()
  test_rmse_all = c()
  for(train_size in train_sizes){
    train_rmse_one_size = c()
    test_rmse_one_size = c()
    for(r in 1:num_repeat){
```

```
      sample <- sample.int(n = nrow(train_df), size = floor(train_size*nrow(train_df)), replace = F)
      train_df_subset = train_df[sample, ]

      # Training and evaluating model
      mod1 = lm(model_exp, data=train_df_subset)
      model_eval = getrmse(mod1, test_df)
      train_rmse = model_eval$train_rmse
      test_rmse = model_eval$test_rmse

      train_rmse_one_size = c(train_rmse_one_size, train_rmse)
      test_rmse_one_size = c(test_rmse_one_size, test_rmse)
    }
    train_rmse_all = c(train_rmse_all, mean(train_rmse_one_size))
    test_rmse_all = c(test_rmse_all, mean(test_rmse_one_size))
  }

  rmse_type = c(rep('train', length(train_sizes)), rep('test', length(train_sizes)))
  rmses = c(train_rmse_all, test_rmse_all)

  df_eval = data.frame(train_size=train_sizes, rmse_type=rmse_type, rmses=rmses)
  best_test_rmse = min(df_eval[df_eval$rmse_type=='test', 'rmses'])
  best_train_size = train_sizes[which.min(df_eval[df_eval$rmse_type=='test', 'rmses'])]

  return(list('df_eval'=df_eval,
              'best_test_rmse'=best_test_rmse,
              'best_train_size'=best_train_size))
}
```

Tasks

Each of the following tasks is worth 20 points, for a total of 100 points for this project. Remember to build each model as specified, evaluate it using the strategy outlined above, and plot the training and test errors by training set size (%).

1. Numeric variables

Use Linear Regression to predict `Gross` based on available *numeric* variables. You can choose to include all or a subset of them.

```
# TODO: Build & evaluate model 1 (numeric variables only)
numeric_cols = colnames(df[, sapply(df, is.numeric)])
print(numeric_cols)
```

```
## [1] "Runtime"       "imdbRating"    "imdbVotes"     "Budget"
## [5] "Gross"         "Date"          "released_year"
```

```
print(summary(df[, numeric_cols]))
```

```
##     Runtime        imdbRating      imdbVotes           Budget
##  Min.   :  1.0   Min.   :1.600   Min.   :      5   Min.   :     1100
##  1st Qu.: 93.0   1st Qu.:5.700   1st Qu.:   8962   1st Qu.:  5250000
##  Median :102.0   Median :6.400   Median :  36268   Median : 20000000
```

```
##  Mean   :104.9   Mean   :6.294   Mean   :  84748   Mean   : 34625509
##  3rd Qu.:115.0   3rd Qu.:7.100   3rd Qu.: 101076   3rd Qu.: 45000000
##  Max.   :219.0   Max.   :9.000   Max.   :1670736   Max.   :425000000
##  NA's   :21      NA's   :22      NA's   :22
##      Gross                Date      released_year
##  Min.   :0.000e+00   Min.   :1999   Min.   :2000
##  1st Qu.:2.924e+06   1st Qu.:2004   1st Qu.:2004
##  Median :3.036e+07   Median :2008   Median :2008
##  Mean   :9.460e+07   Mean   :2008   Mean   :2008
##  3rd Qu.:1.021e+08   3rd Qu.:2012   3rd Qu.:2012
##  Max.   :2.784e+09   Max.   :2016   Max.   :2016
##
```

```
# Number of times to repeat training for stability
num_repeat = 10
model1_cols = c('released_year', 'Runtime', 'imdbRating', 'imdbVotes', 'Budget')
model1_exp = paste('Gross~', paste(model1_cols, collapse='+'))

eval_model1 = eval_model(train_df, test_df, model1_exp, train_sizes, num_repeat=10)
df_eval = eval_model1$df_eval
```

Q: List the numeric variables you used.

A: The numeric variables that we used were Runtime, imdbRating, imdbVotes, Budget, Gross, Date, released_year. (Recall that we stripped out the tomato-like columns in our preprocessing steps above.)

Q: What is the best mean test RMSE value you observed, and at what training set size?

A:

```
# Plotting Training and Test RMSE
ggplot(data=df_eval, aes(x=train_size, y=rmses)) +
  geom_line(aes(color=rmse_type, group=rmse_type)) +
  ggtitle('Model 1: Training and Test RMSE')
```


```
# Best test RMSE
best_testrmse_mod1 = eval_model1$best_test_rmse
best_train_size_mod1 = eval_model1$best_train_size
```

The best mean test RMSE value I observed was 8.7450682×10^7 and I observed that at the training set size 80 percent. Notice from the graph above as well that we see that Test RMSE generally decreases with higher training set. This makes sense because with very low training set size, the model will overfit the more limited data.

2. Feature transformations

Try to improve the prediction quality from **Task 1** as much as possible by adding feature transformations of the numeric variables. Explore both numeric transformations such as power transforms and non-numeric transformations of the numeric variables like binning (e.g. `is_budget_greater_than_3M`).

```
# Helper function to plot relative proportions
plot_relative_proportions = function(df, cols){
  # Keyword Args:
  #  df: DataFrame with columns you want to plot relative proportions of
  #  cols: The columns you want to plot relative proportions of
  # Returns:
  #  plot_proportions: ggplot of proportions
  #  top_cols: Vector of Column Counts
  df.subset = df[c('Title', cols)]
  df.long = melt(df.subset, id.vars=c('Title'))

  # Removing rows where value is 0 because when value is 0, that means movie is not that genre.
```

```
  df.long = df.long[apply(df.long['value'], 1, function(z) !any(z==0)), ]

  # Finally plot relative proportions
  # https://sebastiansauer.github.io/percentage_plot_ggplot2_V2/
  plot_proportions = ggplot(df.long, aes(x=variable)) +
    geom_bar(aes(y = (..count..)/sum(..count..))) +
    ylab('Relative Proportion')

  top_cols = sort(colSums(df[cols]), decreasing=TRUE)
  return(list('plot_proportions'=plot_proportions,
              'top_cols'=top_cols))
}
```

```
# From Lesson 8: Preprocessing Data (22. Skewness and Power Transformations)
# Also from Lessong 10: Logistic Regression (19. Increasing Data Dimensionality)
# The Transforms we are going to use are
# 1. x^2
# 2. log(x) (in theory we could choose difference values of lambda to choose different types of power t

# Helper function to create power transforms
create_power_transforms = function(df, column){
  # Keyword Args:
  #  df: DataFrame with data that you want to add power transformations to
  #  column: Column within df that you want to do power transformations to
  # Returns:
  #  df: DataFrame with new columns as transformations

  # Power Transforms
  # For now, let's only consider power transformations where lambda = 2 or -1. In theory
  # we could try infinite values of lambda with some kind of cross validation approach to tune this
  # but for simplicity let's just use inverse and quadratic.
  df[[paste0(column, '2')]] = (df[[column]] ^ 2 - 1) / 2  # lambda = 2
  df[[paste0(column, 'neg1')]] = -1 * (df[[column]] ^ (-1) - 1) / (-1)  # lambda = -1
  df[[paste0('log', column)]] = log(df[[column]])
  return(df)
}

create_binning_transforms = function(x, threshold){
  # Keyword Args:
  #  x: Vector of data you want to transfor
  #  threshold: Threshold over which transformed vector equals 1, else 0
  # Returns:
  #  binned: New binned transformed vector

  # Power Transforms
  binned = as.numeric(x >= threshold)
  return(binned)
}
```

```
# TODO: Build & evaluate model 2 (transformed numeric variables only)

# Creating new variables
df$is_budget_greater_than_60m = as.numeric(df$Budget >= 60000000)
df$imdbRating_greater_than_75 = as.numeric(df$imdbRating >= 7.5)
```

```
df = create_power_transforms(df, 'Budget')
df = create_power_transforms(df, 'Runtime')
df = create_power_transforms(df, 'imdbRating')
df = create_power_transforms(df, 'imdbVotes')
df = create_power_transforms(df, 'released_year')

# Adding bins by years
df$early_2000s = as.numeric(df$released_year <= 2004)
df$mid_2000s = as.numeric(df$released_year >= 2005 & df$released_year <= 2009)
df$post_2010 = as.numeric(df$released_year >= 2010)

# Adding bins by Runtime
df$RuntimeUnder75 = as.numeric(df$Runtime <= 75)
df$RuntimeBetween75_125 = as.numeric(df$Runtime > 75 & df$Runtime <= 125)
df$RuntimeGreater125 = as.numeric(df$Runtime > 125)

# "Recreate" the training and test df with the same indices to include the new columns
train_df <- df[train_idx, ]
test_df  <- df[-train_idx, ]

model2_cols = c(model1_cols, c('imdbRating_greater_than_75',
                               'imdbRating2',
                               'is_budget_greater_than_60m',
                               'RuntimeUnder75',
                               'RuntimeBetween75_125',
                               'RuntimeGreater125',
                               'Runtime2',
                               'early_2000s',
                               'mid_2000s',
                               'post_2010'
                              ))
model2_exp = paste('Gross~', paste(model2_cols, collapse='+'))

eval_model2 = eval_model(train_df, test_df, model2_exp, train_sizes, num_repeat=10)
df_eval_mod2 = eval_model2$df_eval
```

Q: Explain which transformations you used and why you chose them.

A: In addition to the features from Part 1, I created the following new features

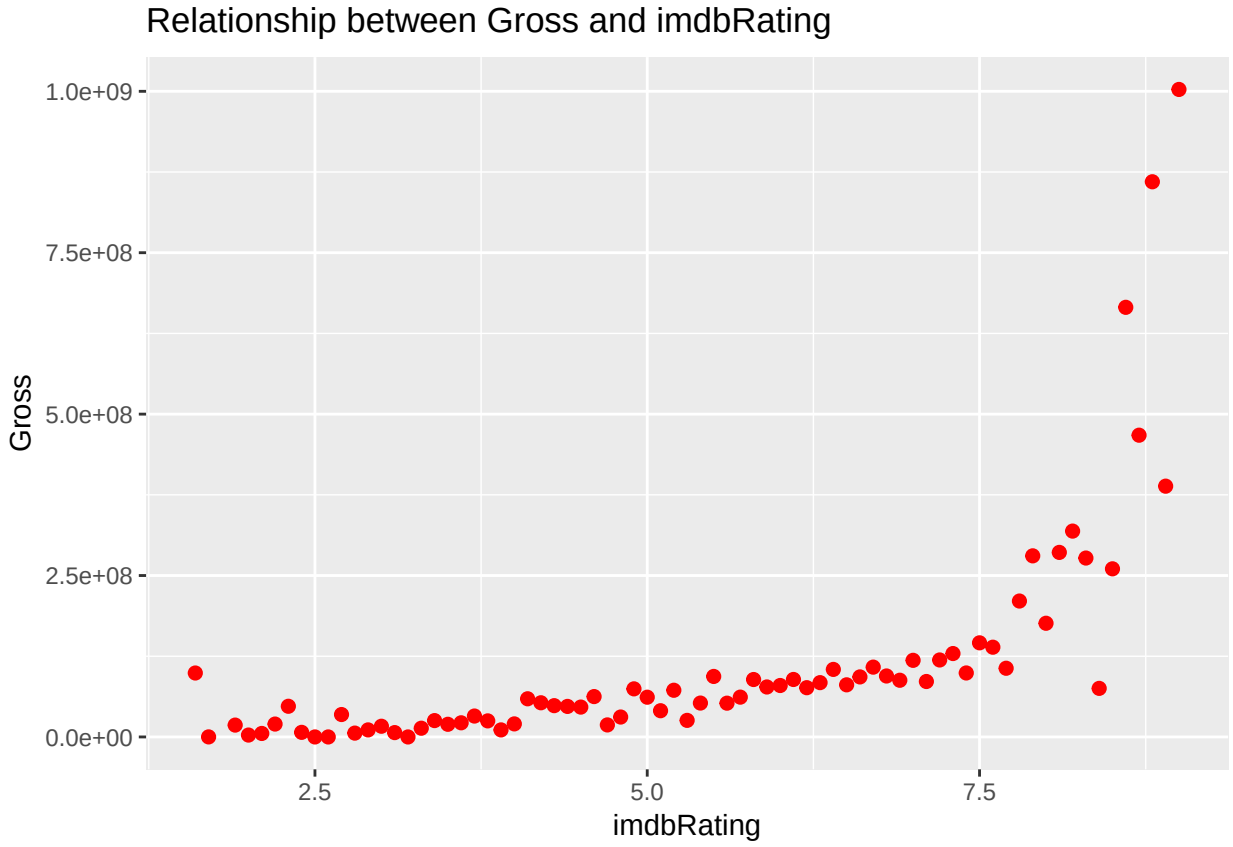
1. Binned imdbRating: Created an indicator variable for if the imdbRating was greater than 7.5
2. Power Transformation for imdbRating: lambda=2
3. Binned Budget: Created an indicator variable for if budget greater than 60 million
4. Binned Runtime (under 75, between 75 and 125, and over 125)
5. Power Transformation for Runtime: lambda=2
6. Binned Released Year (between 2000 and 2004; 2005 and 2009; 2010 and after)

First, I created a binned variable for imdbRating at 7.5. (Note, later on in part 5 as we explore more interesting relationships, I interact this binned variable with imdbRating to capture a different relationship after 7.5 – more on this below).

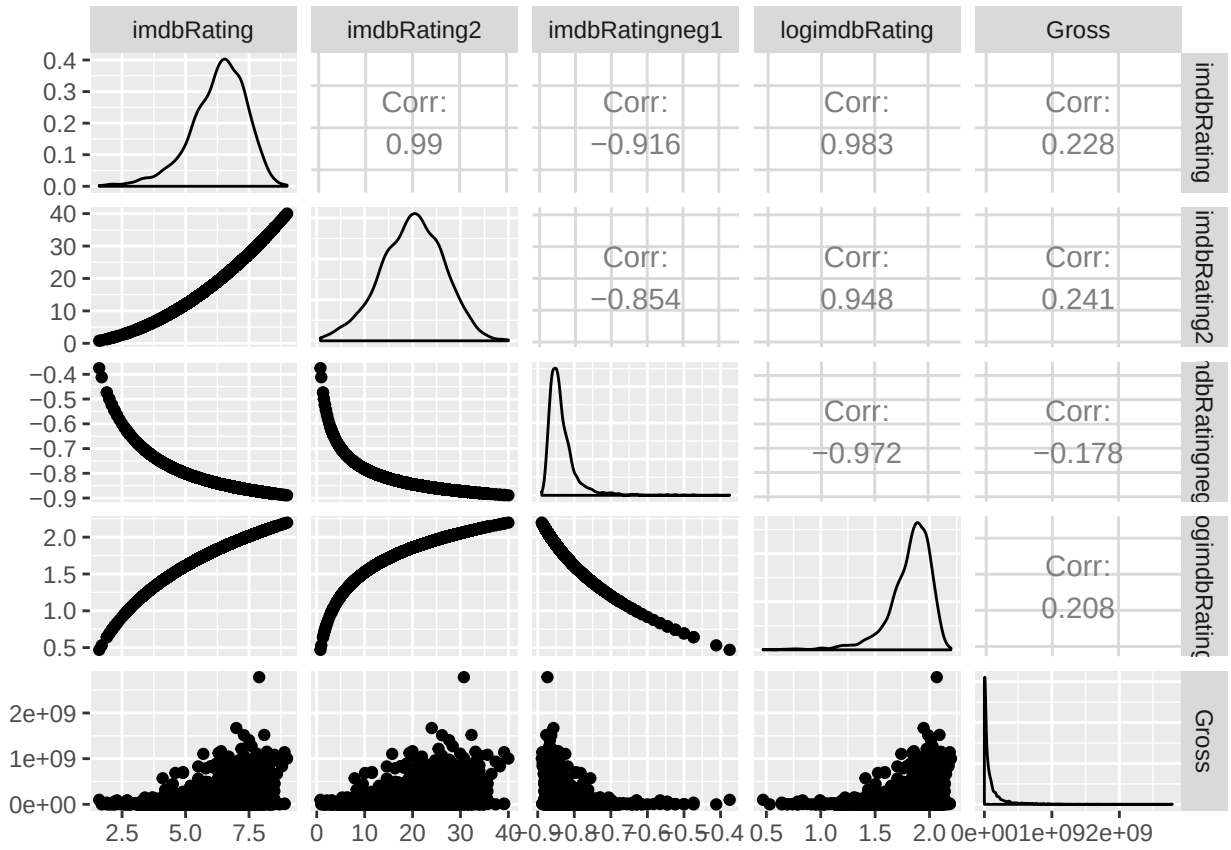
I chose to look at imdbRating because I noticed an interesting relationship with imdbRating and Gross. If you look at the plot below, we plot the average Gross given imdbRating, and notice that the relationship seems to be linear until about 7.5. Then the average gross increases a lot. This is an interesting relationship because maybe movies below 7.5 rating are mostly not that great and so the relationship is linear. But if a movie is really really good, when the relationship between Gross and rating changes, that is a really

really good movie grosses much much more. Thus, I chose to create a binned variable at `imdbRating = 7.5`. This will allow for the model to potentially capture a difference in the effect of rating on `Gross` for movies with ratings greater than 7.5. However, from the graph, notice that the slope of the relationship is not constant before and after 7.5. Thus, in part 5, we will also create an interaction variable to try to capture a different relationship after `imdbRating = 7.5` with `imdbRating2` where `imdbRating2` is the squared term of imdbRating (since the relationship after imdbRating=7.5 looks exponential. Furthermore, looking at the `GGPairs` graph, it looks like `imdbRating2` has a higher linear correlation with `Gross` than `imdbRating`. The other power transformations did not seem to increase linear correlation.)

```
ggplot(df, aes(imdbRating, Gross)) +
  stat_summary(fun.y = "mean", colour = "red", size = 2, geom = "point") +
  ggtitle('Relationship between Gross and imdbRating')
```

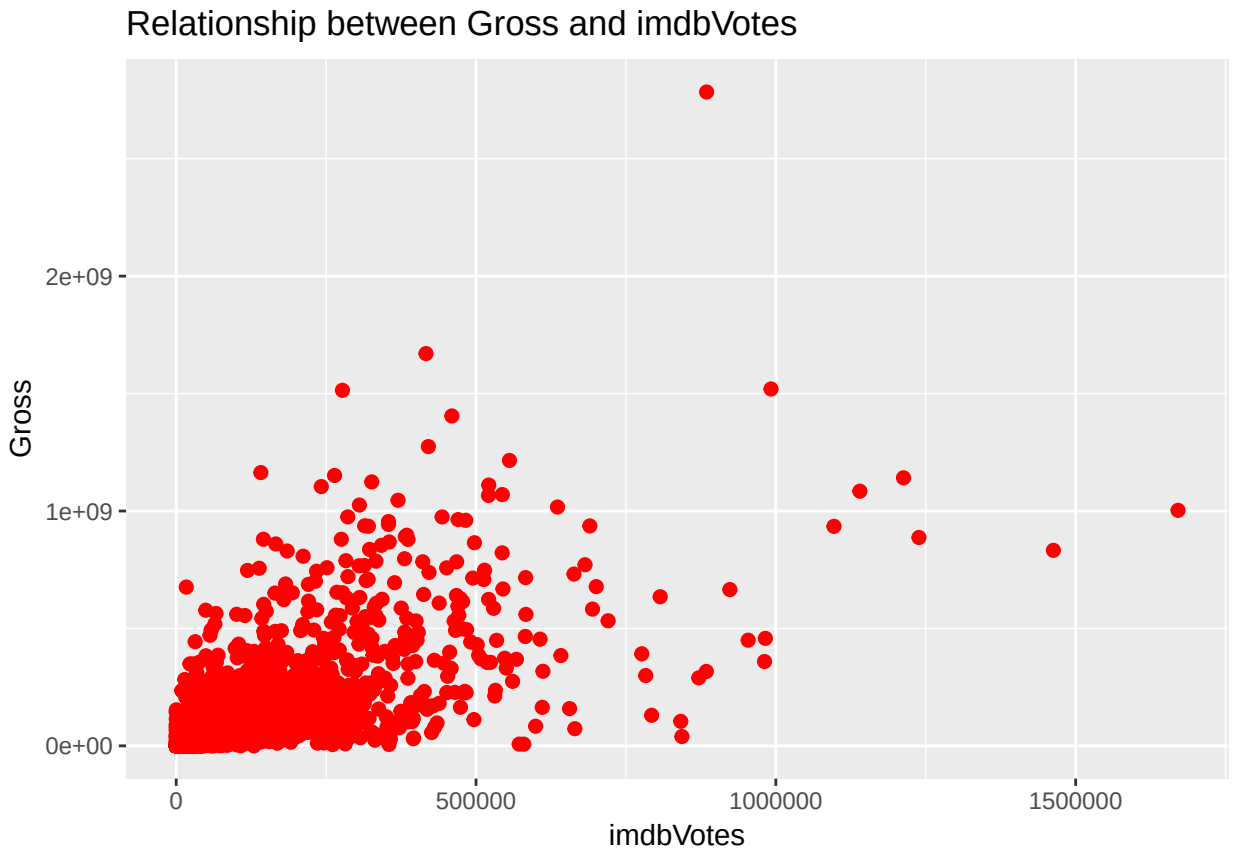



```
ggpairs(df[c('imdbRating', 'imdbRating2', 'imdbRatingneg1', 'logimdbRating', 'Gross')])
```

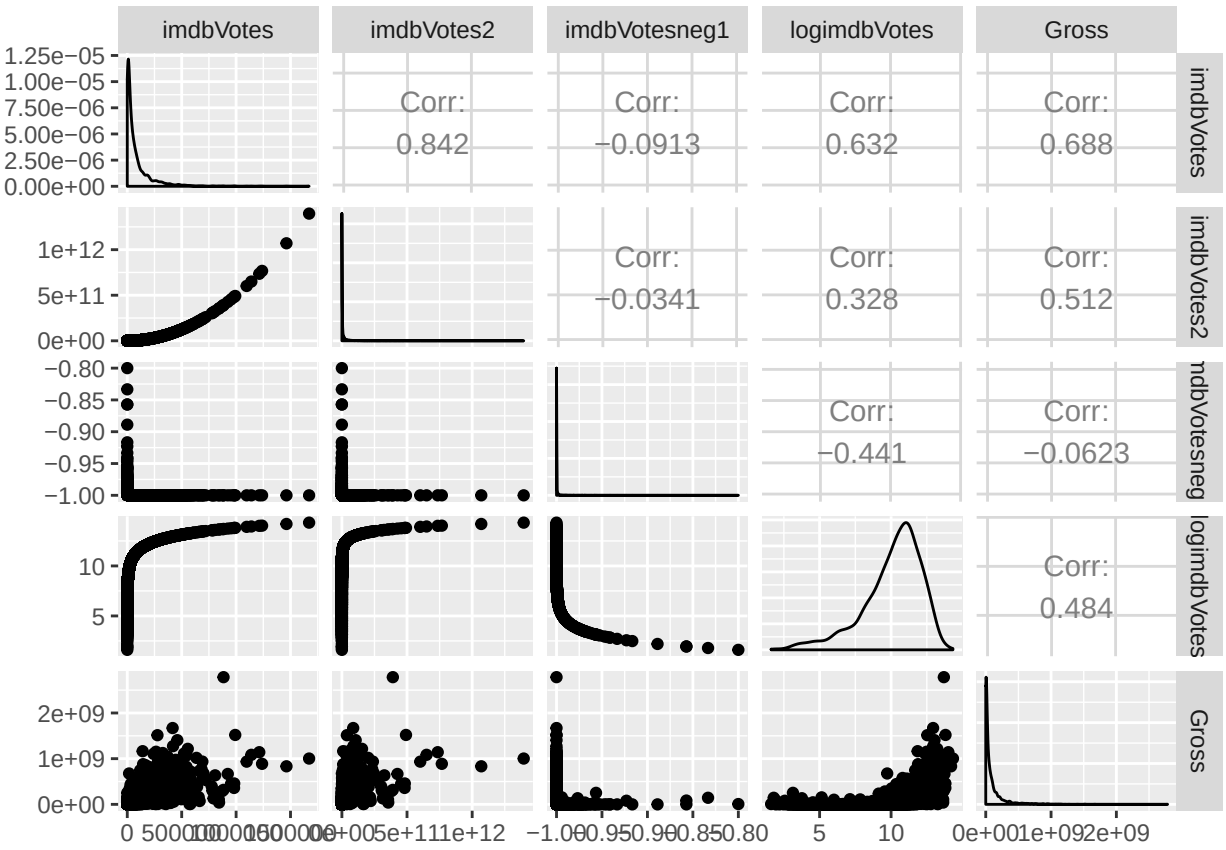


Next, I looked at the relationship between `Gross` and `imdbVotes`. From the graphs below, it doesn't look like there are meaningful bins that would add value. Furthermore, looking at the `GGPairs` plot, none of the power transformations add higher linear correlation with Gross, so it does not make sense to add additional transformations with `imdbVotes` into our model.

```
ggplot(df, aes(imdbVotes, Gross)) +
  stat_summary(fun.y = "mean", colour = "red", size = 2, geom = "point") +
  ggtitle('Relationship between Gross and imdbVotes')
```

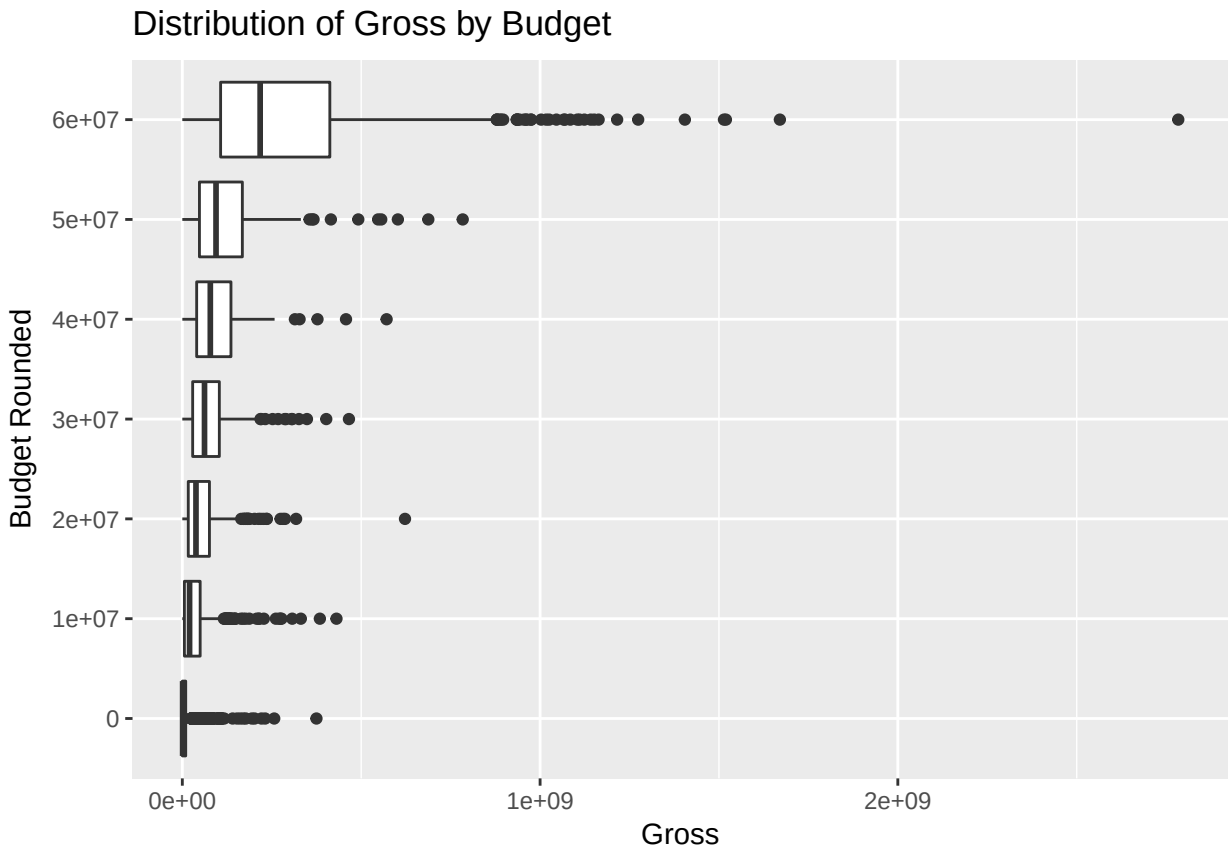



```
ggpairs(df[c('imdbVotes', 'imdbVotes2', 'imdbVotesneg1', 'logimdbVotes', 'Gross')])
```

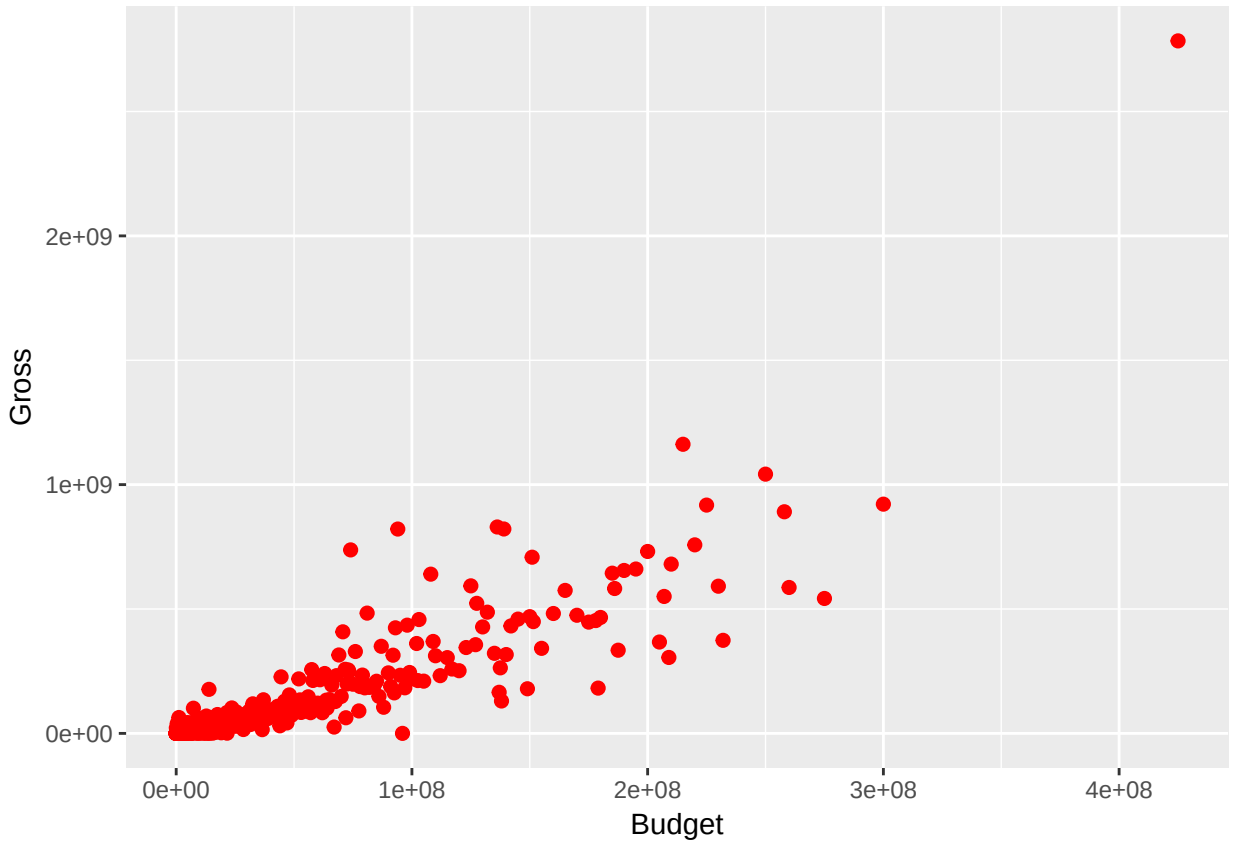


Next, I look at the relationship between Budget and Gross. This is because if we remember back to Project 1, higher budget films tended to gross much higher. Looking at the graphs below, it seems like $60M would be an interesting bin. However, we also plot the GGPairs plot of `Budget2` (power transformation with lambda = 2), `Budgetneg1` (power transformation with lambda = -1), `logBudget` (power transformation with lambda = 0) and `Budget` against `Gross` and you can see at least with the linear correlation, none of the power transformations offer better linear correlation with `Gross` than `Budget`. Thus, we choose not to add any of the power transformations of `Budget` into this model.

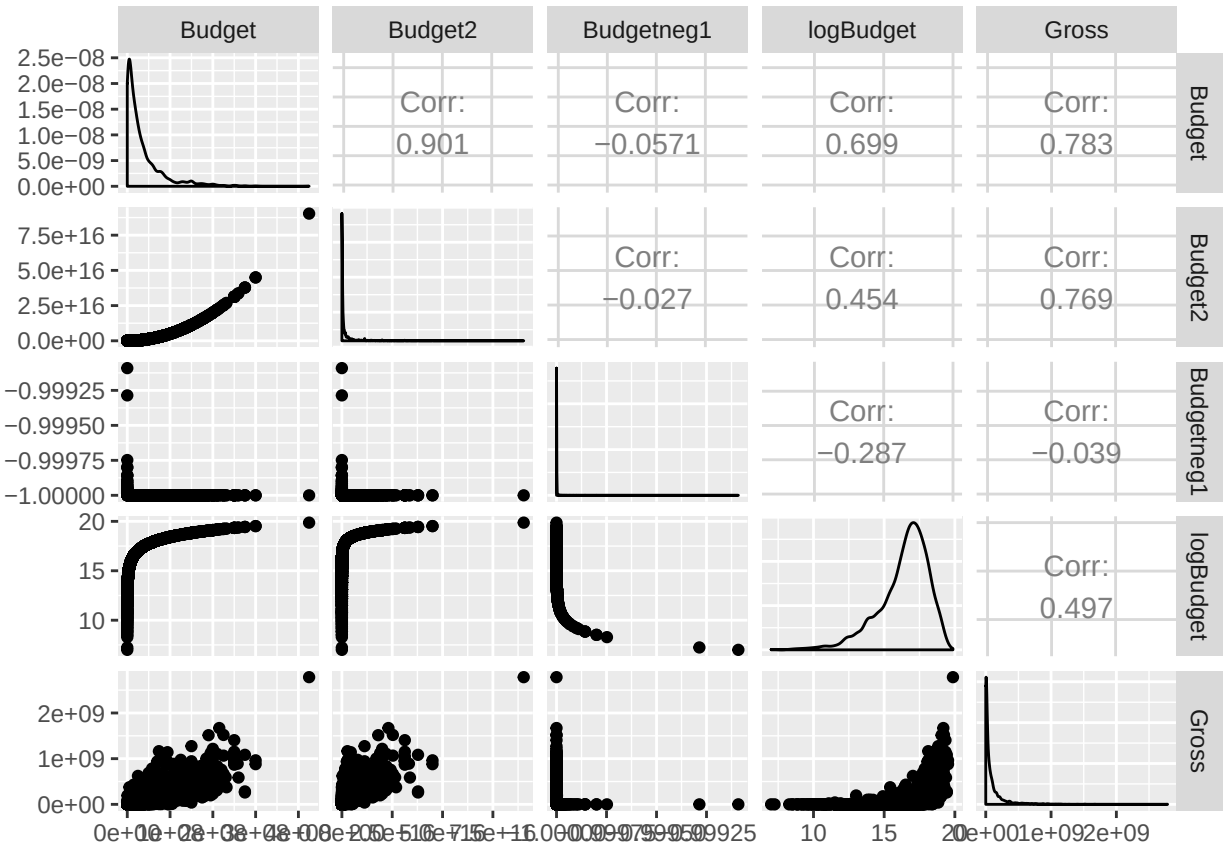
```
# Distribution of Gross by Budget
# Now we need to show distribution of runtime by Budget. Do a boxplot grouped by budget
df$budget_rounded = round_any(df$Budget, 10000000, f = floor)
df$budget_rounded[df$Budget >= 60000000] = 60000000
df$budget_rounded = as.character(df$budget_rounded)
df$budget_rounded[df$budget_rounded == '600000000'] = 'Over 60M'

ggplot(df, aes(as.factor(budget_rounded), Gross)) +
  geom_boxplot() +
  coord_flip() +
  scale_x_discrete("Budget Rounded") +
  ggtitle('Distribution of Gross by Budget')
```




```
ggplot(df, aes(Budget, Gross)) +
  stat_summary(fun.y = "mean", colour = "red", size = 2, geom = "point")
```

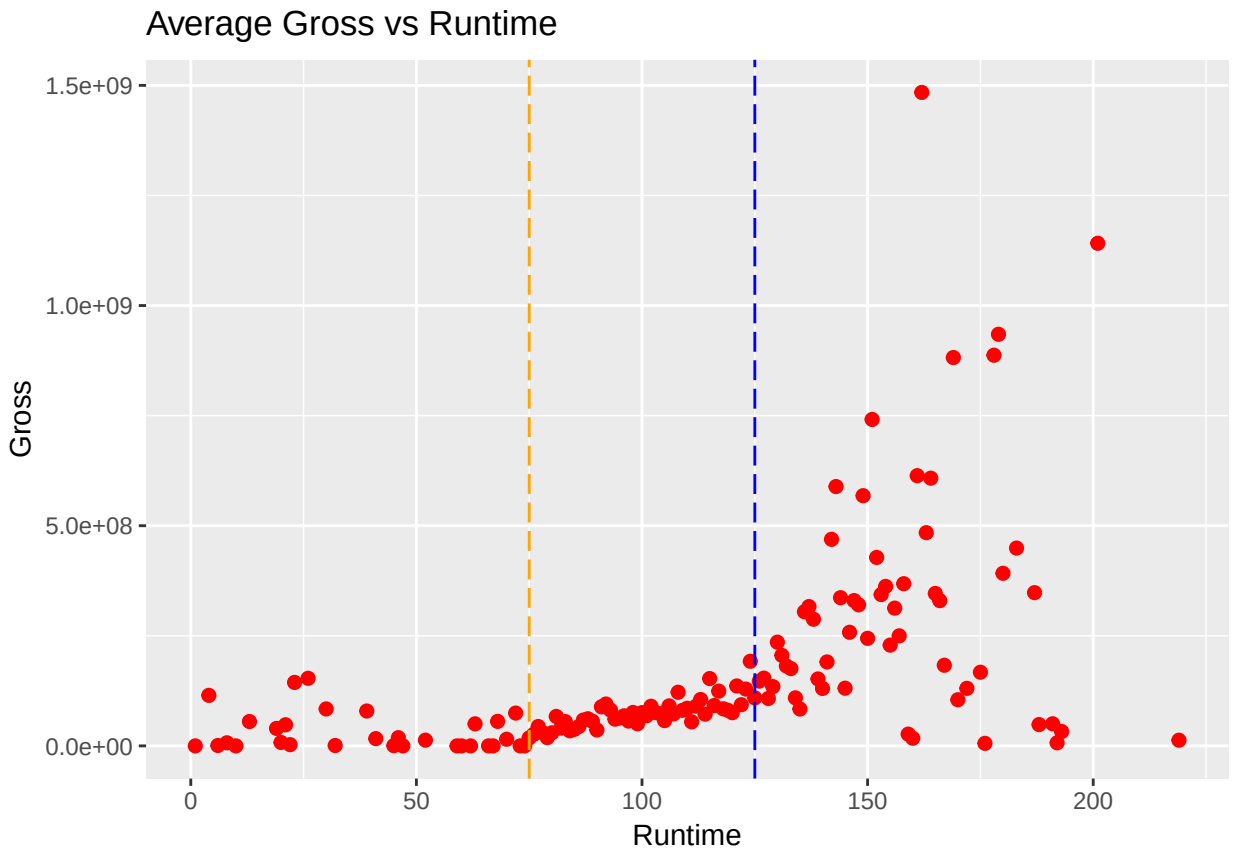



```
ggpairs(df[c('Budget', 'Budget2', 'Budgetneg1', 'logBudget', 'Gross')])
```

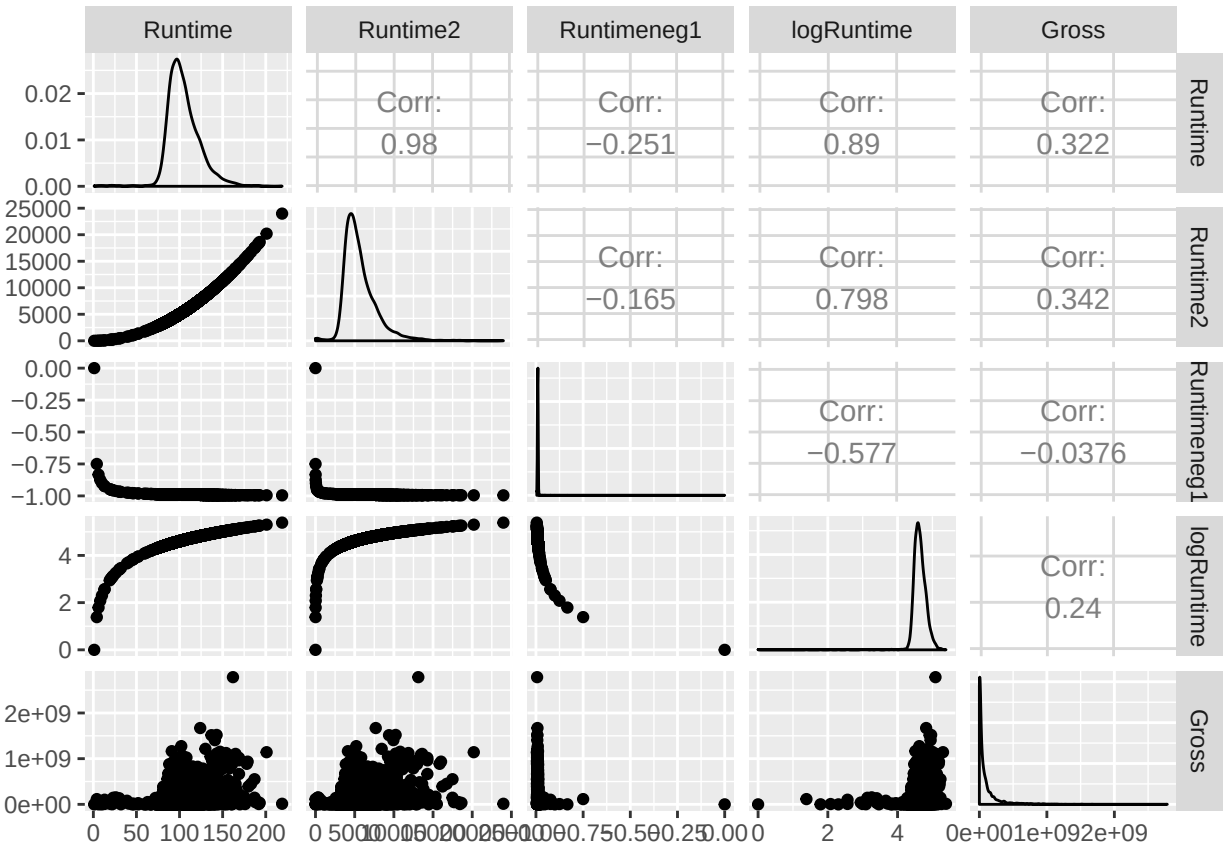


Next, we look at the relationship between Runtime and Gross. There are a few things that are interesting here. Notice first that if we look at the relationship between `Gross` and `Runtime` there seem to be a few interesting "cut points". For `Runtime` between 0 and 75 (up until the orange dotted line), the relationship seems to be flat. For `Runtime` between 75 and 125, the relationship is linearly positively sloped. Finally, for `Runtime` more than 125 (over 2 hours), the relationship with Gross seems to be very highly variable. Furthermore, the overall relationship seems to be slightly quadratic; notice that in the `GGPairs` plot, the correlation between `Runtime2` (power transformation with lambda = 2) and `Gross` is higher than the correlation between `Runtime` and `Gross`. Thus, we choose to add bins for `Runtime` up to 75, between 75 and 125, and over 125. Furthermore, we choose to add power transformation of `Runtime2`. In Question 5, we will also explore adding interactions between the binned variable and the power transformation, which seems appropriate since the relationship between `Runtime` and `Gross` does not seem to be the same within these points.

```
ggplot(df, aes(Runtime, Gross)) +
  stat_summary(fun.y = "mean", colour = "red", size = 2, geom = "point") +
  geom_vline(xintercept=75, color='orange', linetype='longdash') +
  geom_vline(xintercept=125, color='blue', linetype='longdash') +
  ggtitle('Average Gross vs Runtime')
```



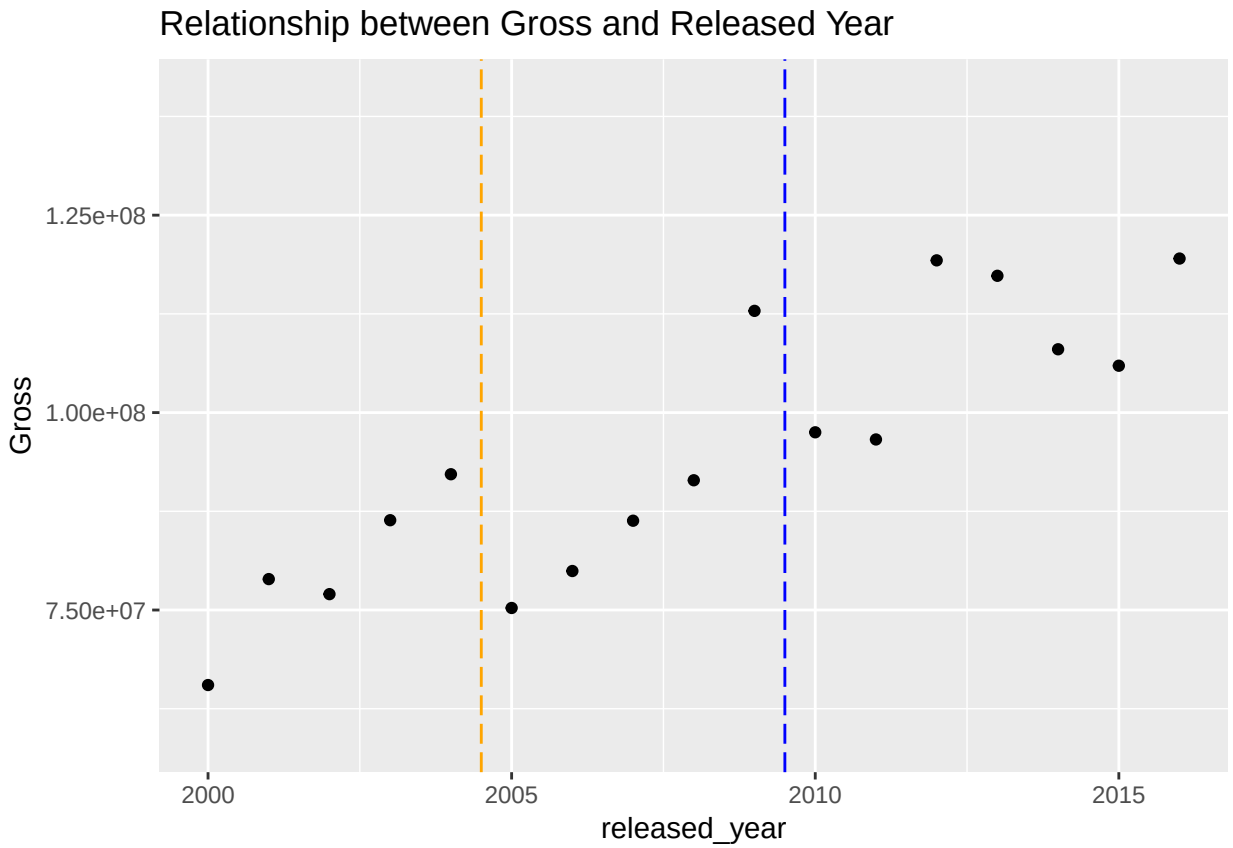

```
ggpairs(df[c('Runtime', 'Runtime2', 'Runtimeneg1', 'logRuntime', 'Gross')])
```



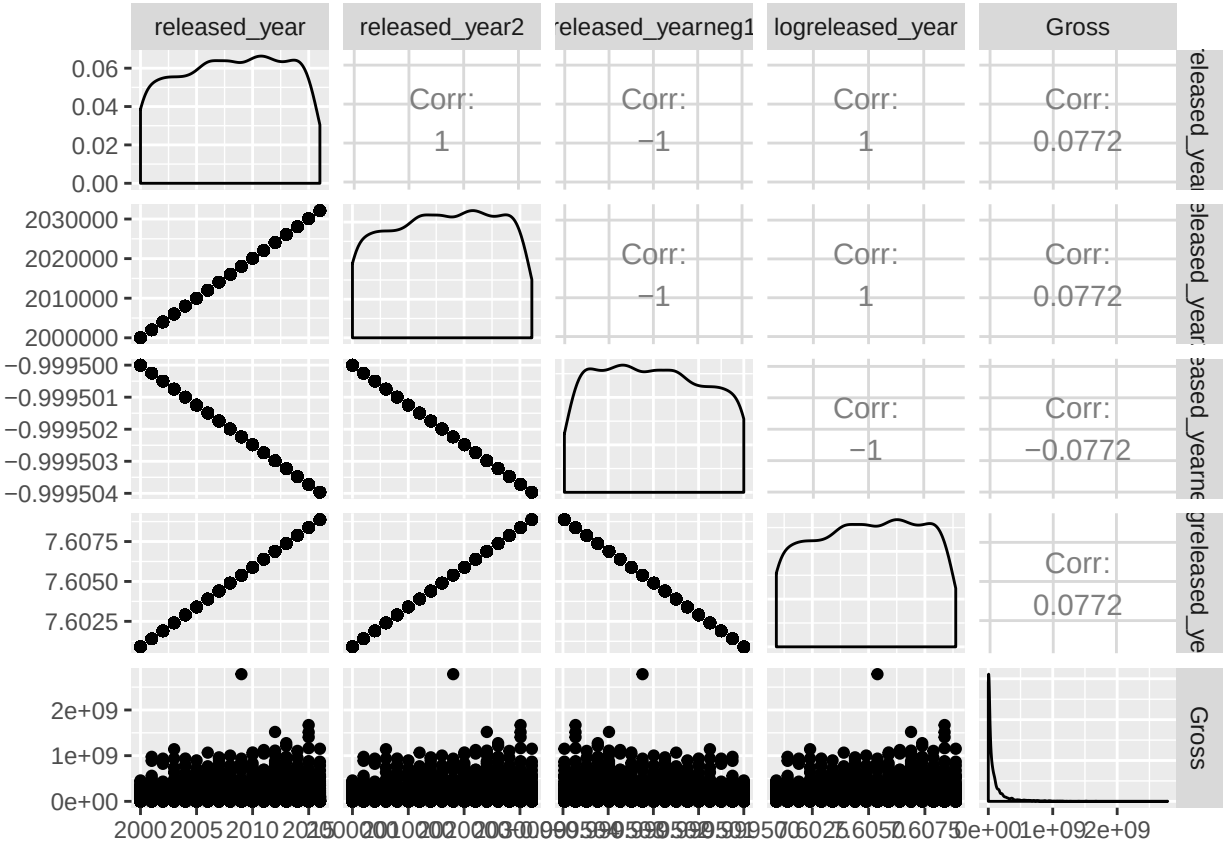
Finally, the last numerical column is `released_year`. If we plot the average gross amount by `released_year` we see there are actually "cycles" that appear to be forming.

```
ggplot(df, aes(released_year, Gross)) +
  geom_point(stat='summary') +
  geom_vline(xintercept=2004.5, color='orange', linetype='longdash') +
  geom_vline(xintercept=2009.5, color='blue', linetype='longdash') +
  ggtitle('Relationship between Gross and Released Year')
```

```
## No summary function supplied, defaulting to `mean_se()
```




```
ggpairs(df[c('released_year', 'released_year2', 'released_yearneg1', 'logreleased_year', 'Gross')])
```



For example, in the graph above, we see that gross amounts are generally increasing between 2000 to 2004. Then drops in 2005 and generally rises again between 2005 and 2009. Then, average gross amounts drops again in 2010. This is likely reflective of the economy/business cycles. Especially the drop between 2009 to 2010 could be a reflection of the Great Recession where people spent less money on entertainment. Thus, I wanted to, instead of just using `released_year` as a continuous variable, create three bins of years: early 2000's, mid 2000's, and post 2010.

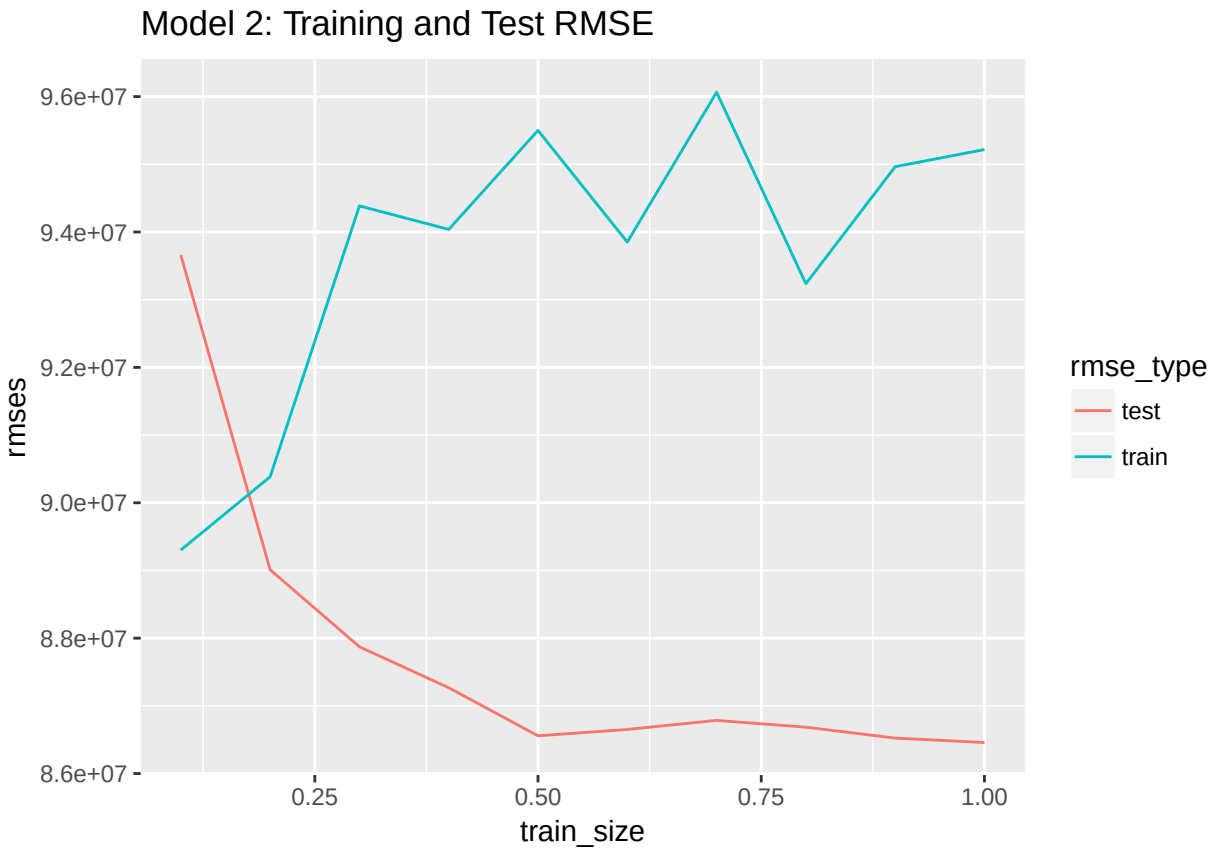
However, notice that in the `GGPairs` plot, none of the power transformations seemed to offer any additional linear correlation; thus I chose not to add any power transformations of `released_year` into the model.

Q: How did the RMSE change compared to Task 1?

A:

Below, we plot the Training and Test RMSE for Model 2. Furthermore, we can directly compare the best test RMSE below:

```
ggplot(data=df_eval_mod2, aes(x=train_size, y=rmses)) +
  geom_line(aes(color=rmse_type, group=rmse_type)) +
  ggtitle('Model 2: Training and Test RMSE')
```




```
print(eval_model1$best_test_rmse)
```

```
## [1] 87450682
```

```
print(eval_model2$best_test_rmse)
```

```
## [1] 86458175
```

```
improvement_in_rmse_model2 = eval_model2$best_test_rmse - eval_model1$best_test_rmse
percentage_improvement_in_rmse_model2 = 1.0 * abs(eval_model2$best_test_rmse - eval_model1$best_test_rms
print('Improvement in RMSE')
```

```
## [1] "Improvement in RMSE"
```

```
print(improvement_in_rmse_model2)
```

```
## [1] -992507.5
```

```
# As a percentage
print(percentage_improvement_in_rmse_model2)
```

```
## [1] 0.01134934
```

Notice that the improvement in model2 is -9.9250752×10^5 which actually represents a 1.134934 percentage gain! While this might seem modest; remember that we did note some meaningful interactions that will be further explored in Part 5!

Also, we can look at where the best test MSE occured, as a function of training size:

```
# Best test RMSE
best_testrmse_mod2 = eval_model2$best_test_rmse
best_train_size_mod2 = eval_model2$best_train_size
```

The best mean test RMSE value I observed was 8.6458175×10^7 and I observed that at the training set size 100 percent. Notice from the graph above as well that we see that Test RMSE generally decreases with higher training set. This makes sense because with very low training set size, the model will overfit the more limited data.

3. Non-numeric variables

Write code that converts genre, actors, directors, and other categorical variables to columns that can be used for regression (e.g. binary columns as you did in Project 1). Also process variables such as awards into more useful columns (again, like you did in Project 1). Now use these converted columns only to build your next model.

```
# Create a helper function to help one-hot encode certain text columns
# For example, Genre, Actors, Directors
# By default we will keep the top ten of each
create_one_hot_encoded = function(df, column, numTopCols=10){
  # Keyword Args:
  #  df: DataFrame with data and column you want to one-hot-encode
  #  column: Column within the df that you want to one-hot encode
  # Returns:
  #  df: Original DataFrame with newly appended columns of numTopCols highest occuring value of Column
  #  top_cols: The top_cols that were returned

  # Example of how to one-hot encode: https://stackoverflow.com/questions/39778387/r-dataframe-one-hot-
  df = cbind(df, mtabulate(strsplit(df[[column]], ",")))
  cols = unique(unlist(strsplit(df[[column]], ',')))
  top_cols = sort(colSums(df[cols]), decreasing=TRUE)
  top_cols_names = names(top_cols[1:numTopCols])
  non_top_cols_names = setdiff(cols, top_cols_names)

  # Remove the non_top_cols
  df[non_top_cols_names] = list(NULL)
  return(list('df'=df,
              'top_cols'=top_cols_names))
}
```

```
# Processing Genre Column -- Work taken from PR1
# Note: Change 'Sci-Fi' to 'SciFi' because dashes in column names are really annoying in R
# Also changing N/A in Genre to na since / are also really annoying
df$Genre <- gsub('Sci-Fi', 'SciFi', df$Genre)

# Processing Rating Column -- Work taken from PR1
df$rated_cleaned = ifelse(df$Rated %in% c('G', 'PG-13', 'PG', 'R', 'NC-17'), df$Rated, 'Other')
df$rated_cleaned <- gsub('PG-13', 'PG13', df$rated_cleaned)
df$rated_cleaned <- gsub('NC-17', 'NC17', df$rated_cleaned)

# Preprocessing Categorical Variables to remove N/A
df$Actors <- gsub('N/A', 'na', df$Actors)
df$Country <- gsub('N/A', 'na', df$Country)
df$Director <- gsub('N/A', 'na', df$Director)
df$Genre <- gsub('N/A', 'na', df$Genre)
df$Writer <- gsub('N/A', 'na', df$Writer)
df$rated_cleaned <- gsub('N/A', 'na', df$rated_cleaned)
```

```
# Preprocessing Categorical Variables to remove spaces
df$Actors <- gsub(' ', '', df$Actors)
df$Country <- gsub(' ', '', df$Country)
df$Director <- gsub(' ', '', df$Director)
df$Genre <- gsub(' ', '', df$Genre)
df$Writer <- gsub(' ', '', df$Writer)
df$rated_cleaned <- gsub(' ', '', df$rated_cleaned)

# Preprocessing to remove parantheses
df$Writer <- gsub('\\(', '', df$Writer)
df$Writer <- gsub('\\)', '', df$Writer)

# Creating one-hot encoded Categorical columns
genre_one_hot_encoded = create_one_hot_encoded(df, 'Genre', 10)
df = genre_one_hot_encoded$df
top_genres = genre_one_hot_encoded$top_cols

actor_one_hot_encoded = create_one_hot_encoded(df, 'Actors', 20)
df = actor_one_hot_encoded$df
top_actors = actor_one_hot_encoded$top_cols

director_one_hot_encoded = create_one_hot_encoded(df, 'Director', 20)
df = director_one_hot_encoded$df
top_directors = director_one_hot_encoded$top_cols

country_one_hot_encoded = create_one_hot_encoded(df, 'Country', 10)
df = country_one_hot_encoded$df
top_countries = country_one_hot_encoded$top_cols

writer_one_hot_encoded = create_one_hot_encoded(df, 'Writer', 20)
df = writer_one_hot_encoded$df
top_writers = writer_one_hot_encoded$top_cols

rating_one_hot_encoded = create_one_hot_encoded(df, 'rated_cleaned', 5)
df = rating_one_hot_encoded$df
top_ratings = rating_one_hot_encoded$top_cols

# Remove columns from DataFrame
df$Genre = NULL
df$Director = NULL
df$Actors = NULL
df$Country = NULL
df$Rating = NULL
```

```
# Processing Awards column
# Work taken from PR1
cols = c('Title', 'Awards', 'Gross')
df6 = df[cols]

awards = stri_extract_all(df6$Awards, regex="\\d+")
award_labels = stri_extract_all(tolower(df6$Awards), regex="win|won|wins|nomin", ignore.case=TRUE)

wins_list = c()
nominations_list = c()
```

```
for (i in 1:length(awards)){
    wins = 0
    nominations = 0
    for(j in 1:length(awards[[i]])){
        if(!is.na(award_labels[[i]][j]) & (award_labels[[i]][j] == 'wins' | award_labels[[i]][j] == 'wi
            wins = wins + as.numeric(awards[[i]][j])
        } else if (!is.na(award_labels[[i]][j]) & award_labels[[i]][j] == 'nomin'){
            nominations = nominations + as.numeric(awards[[i]][j])
        }
    }
    wins_list = c(wins_list, wins)
    nominations_list = c(nominations_list, nominations)
}

df$wins = wins_list
df$nominations = nominations_list
```

```
# TODO: Build & evaluate model 3 (converted non-numeric variables only)
# "Recreate" the training and test df with the same indices to include the new columns
train_df <- df[train_idx, ]
test_df  <- df[-train_idx, ]

# Turning a vector of columns in a model expression: https://stackoverflow.com/questions/7986805/how-do
model3_cols = c(top_genres, top_directors, top_actors, top_countries, top_writers, top_ratings, c('wins
model3_exp = paste('Gross~', paste(model3_cols, collapse='+'))

eval_model3 = eval_model(train_df, test_df, model3_exp, train_sizes, num_repeat=10)
df_eval_mod3 = eval_model3$df_eval
```

Q: Explain which categorical variables you used, and how you encoded them into features.

A:

The categorical variables that I used were

1. Genres (top 10)
2. Directors (top 20)
3. Actors (top 20)
4. Countries (top 10)
5. Writers (top 20)
6. Awards (processed into wins and nominations)
7. Ratings (e.g. R, PG13, PG, G; top 5)

The reason for these categorical variables why I only use the top 10/20 is because of curse of dimensionality. If I included all as features, I would have too much variation (ala bias-variance tradeoff). To encode them into features, for each entity (e.g. genre, director, actor) I created a new one-hot encoded binary variable. Each variable would equal 1 if the movie contained that entity, else 0. Please see the helper function I defined `create_one_hot_encoded` for the code that I used.

Q: What is the best mean test RMSE value you observed, and at what training set size? How does this compare with Task 2?

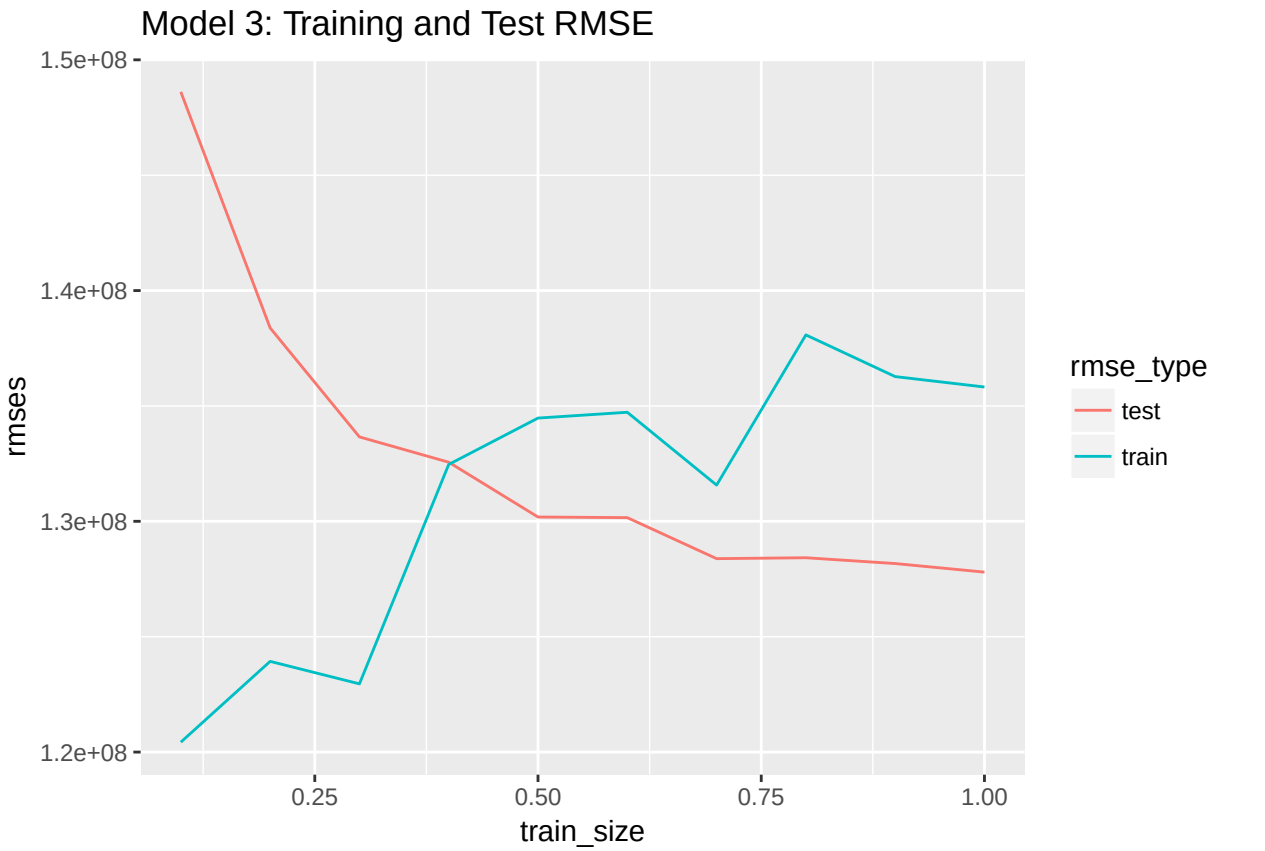
A:

```
# Best test RMSE
best_testrmse_mod3 = eval_model3$best_test_rmse
best_train_size_mod3 = eval_model3$best_train_size
```

The best mean test RMSE value I observed was 1.2779824×10^8 and I observed that at the training set size 100 percent. Notice from the graph above as well for model 3 that we see that Test RMSE generally decreases with higher training set. Again, this makes sense because with very low training set size, the model will overfit the more limited data.

We can also compare the best test RMSE directly between Models 1, 2, and 3

```
ggplot(data=df_eval_mod3, aes(x=train_size, y=rmses)) +
  geom_line(aes(color=rmse_type, group=rmse_type)) +
  ggtitle('Model 3: Training and Test RMSE')
```




```
print(eval_model1$best_test_rmse)
```

```
## [1] 87450682
```

```
print(eval_model2$best_test_rmse)
```

```
## [1] 86458175
```

```
print(eval_model3$best_test_rmse)
```

```
## [1] 127798239
```

Notice that compared to Model 2, Model 3 performed much worse. This is because we only used categorical variables for Model 3 and we have a pretty sparse data set just using categorical variables. This also potentially suggests that just using the categorical variables is not sufficient, we need to combine numerical and even interact numerical and categorical variables to come up with an even better model.

4. Numeric and categorical variables

Try to improve the prediction quality as much as possible by using both numeric and non-numeric variables from **Tasks 2 & 3**.

```
# TODO: Build & evaluate model 4 (numeric & converted non-numeric variables)
model4_cols = unique(c(model2_cols, model3_cols))
model4_exp = paste('Gross~', paste(model4_cols, collapse='+'))

eval_model4 = eval_model(train_df, test_df, model4_exp, train_sizes, num_repeat=10)
df_eval_mod4 = eval_model4$df_eval
```

Q: Compare the observed RMSE with Tasks 2 & 3.

A: For Model 4, I simply combined (but no additional interactions) the terms from Task 2 and 3 together. The Training vs Test RMSE for Model 4 is shown above. In addition, we can directly compare the best test RMSEs:

```
ggplot(data=df_eval_mod4, aes(x=train_size, y=rmses)) +
  geom_line(aes(color=rmse_type, group=rmse_type)) +
  ggtitle('Model 4: Training and Test RMSE')
```


```
print(eval_model1$best_test_rmse)
```

```
## [1] 87450682
```

```
print(eval_model2$best_test_rmse)
```

```
## [1] 86458175
```

```r
print(eval_model3$best_test_rmse)
```

```
## [1] 127798239
```

```r
print(eval_model4$best_test_rmse)
```

```
## [1] 84986467
```

The improvement of Model 4 compared to Model 2 was:

```r
improvement_in_rmse_model4 = eval_model4$best_test_rmse - eval_model2$best_test_rmse
percentage_improvement_in_rmse_model4 = 1.0 * abs(eval_model4$best_test_rmse - eval_model2$best_test_rms
print('Improvement in RMSE')
```

```
## [1] "Improvement in RMSE"
```

```r
print(improvement_in_rmse_model4)
```

```
## [1] -1471708
```

```r
# As a percentage
print(percentage_improvement_in_rmse_model4)
```

```
## [1] 0.0170222
```

Notice that the improvement in Model 4 was -1.4717082×10^6 which represents a further 1.7022198 percent improvement over Model 2! While this improvement might seem modest, again, in Part 5, we explore how to improve this even further with meaningful interactions. Furthermore, note that there is a massive improvement over Part 3 where we just had categorical variables.

```r
# Best test RMSE
best_testrmse_mod4 = eval_model4$best_test_rmse
best_train_size_mod4 = eval_model4$best_train_size
```

The best mean test RMSE value I observed for Model 4 was 8.4986467×10^7 and I observed that at the training set size 100 percent. Notice again, we see a similar pattern as other models. From the graph above, notice that for Model 4 that we see that Test RMSE generally decreases with higher training set. Again, this makes sense because with very low training set size, the model will overfit the more limited data.

5. Additional features

Now try creating additional features such as interactions (e.g. `is_genre_comedy` x `is_budget_greater_than_3M`) or deeper analysis of complex variables (e.g. text analysis of full-text columns like `Plot`).

```r
# TODO: Build & evaluate model 5 (numeric, non-numeric and additional features)

# Creating interaction variables between variables, as explored and discussed in Part 2
df$imdbRating_greater_than_75_imdbRating = df$imdbRating_greater_than_75 * df$imdbRating
df$imdbRating_greater_than_75_imdbRating2 = df$imdbRating_greater_than_75 * df$imdbRating2
df$is_budget_greater_than_60m_budget = df$is_budget_greater_than_60m * df$Budget
df$Runtime_RuntimeUnder75 = df$Runtime * df$RuntimeUnder75
df$Runtime_RuntimeBetween75_125 = df$Runtime * df$RuntimeBetween75_125
df$Runtime_RuntimeGreater125 = df$Runtime * df$RuntimeGreater125
df$released_year_early_2000s = df$released_year * df$early_2000s
df$released_year_mid_2000s = df$released_year * df$mid_2000s
df$released_year_post_2010 = df$released_year * df$post_2010

# Creating Family Friendly variable that is G, PG, or PG13. Also adding interactions.
```

```r
df$family_friendly = as.numeric(df$G + df$PG + df$PG13 >= 1)
df$family_friendly_wins = df$family_friendly * df$wins
df$family_friendly_nominations = df$family_friendly * df$nominations

# Made to DVD
df$made_to_dvd = as.numeric(!is.na(df$DVD))

# Number of Languages
df$number_of_languages = sapply(strsplit(df[['Language']], ","), length)

# "Recreate" the training and test df with the same indices to include the new columns
train_df <- df[train_idx, ]
test_df  <- df[-train_idx, ]

additional_model5_cols = c('imdbRating_greater_than_75_imdbRating',
                           'imdbRating_greater_than_75_imdbRating2',
                           'is_budget_greater_than_60m_budget',
                           'Runtime_RuntimeUnder75',
                           'Runtime_RuntimeBetween75_125',
                           'Runtime_RuntimeGreater125',
                           'released_year_early_2000s',
                           'released_year_mid_2000s',
                           'released_year_post_2010',
                           'family_friendly',
                           'family_friendly_wins',
                           'family_friendly_nominations',
                           'made_to_dvd',
                           'number_of_languages')
model5_cols = unique(c(model4_cols, additional_model5_cols))
model5_exp = paste('Gross~', paste(model5_cols, collapse='+'))

eval_model5 = eval_model(train_df, test_df, model5_exp, train_sizes, num_repeat=10)
df_eval_mod5 = eval_model5$df_eval
```

Q: Explain what new features you designed and why you chose them.

A: In addition to the features from the previous parts, I created the following new interaction features. Note that the reasoning behind these interactions were already partly explored in Part 2, and I will elaborate further here

1. Interaction of `imdbRating` and `imdbRating_greater_than_75`
2. Interaction of `imdbRating2` and `imdbRating_greater_than_75`
3. Interaction of `Budget` and `is_budget_greater_than_60m`
4. Interaction of `Runtime` and `RuntimeUnder75`, `RuntimeBetween75_125`, and `RuntimeGreater125`
5. Interaction of `Runtime2` and `RuntimeUnder75`, `RuntimeBetween75_125`, and `RuntimeGreater125`
6. Interaction of `released_year` and `early_2000s`, `mid_2000s`, and `post_2010`

Furthermore, I explore other relationships that were entirely previously undiscussed such as

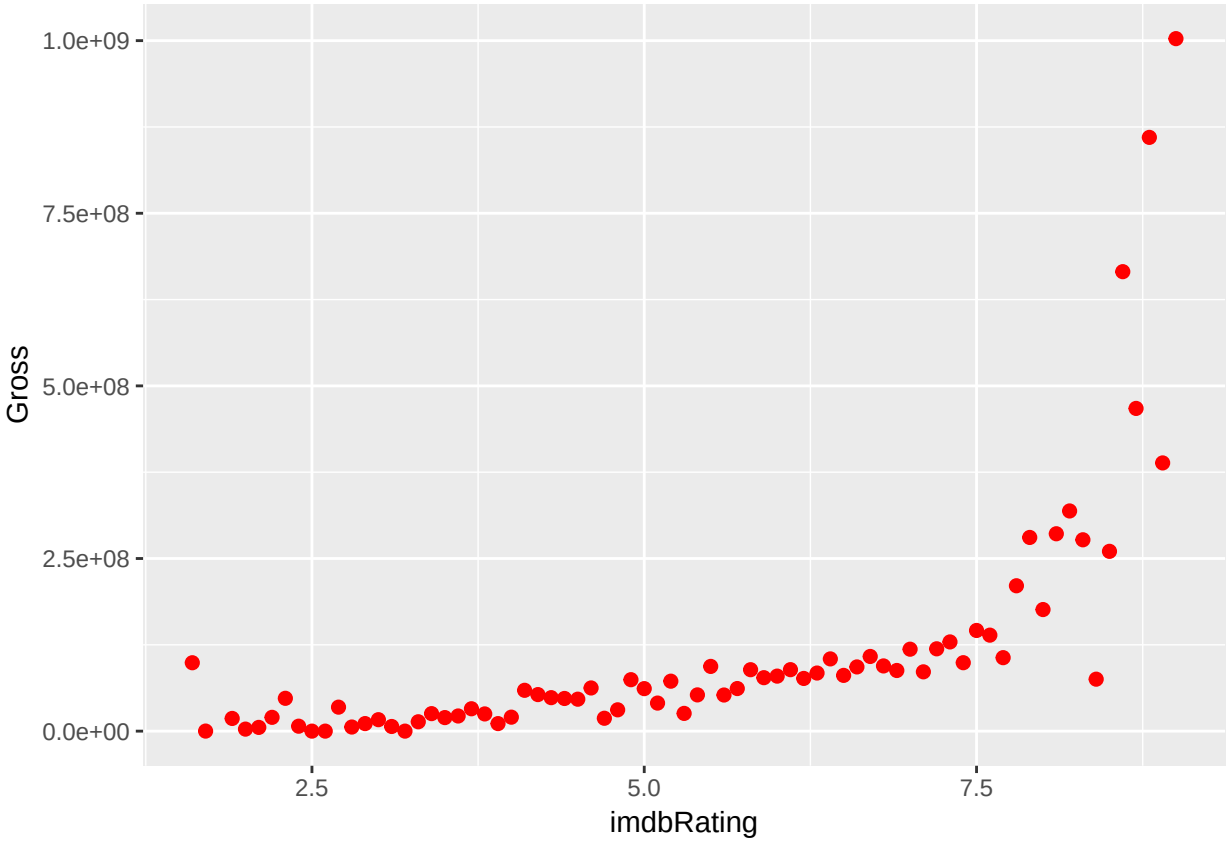
7. `family_friendly`: Defined as MPAA rating of G, PG, or PG-13
8. `family_friendly_wins`: Interaction of `family_friendly` and `wins`
9. `family_friendly_nominations`: Interaction of `family_friendly` and `nominations`.
10. `made_to_dvd`: Indicator variable for if movie was made to DVD, intuition being if made to DVD perhaps it was a successful movie and Grossed more.
11. `number_of_languages`: Parsed the `Language` column and count number of languages movie is made. Intuition being if multiple languages, it is an international release/success and thus Grossed more.

For the first 6 interactions, we explored these a bit in Part 2, but I'll elaborate furhter here. Recall that in Part 2, I created a binned variable for imdbRating at 7.5. Now, I will interact this binned variable with `imdbRating` and `imdbRating2` to capture a different relationship after 7.5 (more on this below).

If we re-look at the relationship between `imdbRating` and `Gross`, we plot the average Gross given imdbRating, and notice that the relationship seems to be linear until about 7.5. Then the average gross increases a lot. This is an interesting relationship because maybe movies below 7.5 rating are mostly not that great and so the relationship is linear. But if a movie is really really good, when the relationship between Gross and rating changes, that is a really really good movie grosses much much more. Thus, I chose to create a binned variable at `imdbRating = 7.5` and interacted it with `imdbRating` and `imdbRating2` to try to capture a different relationship after `imdbRating = 7.5`, where `imdbRating2` is the squared term of imdbRating (since the relationship after imdbRating=7.5 looks exponential).

```
ggplot(df, aes(imdbRating, Gross)) +
  stat_summary(fun.y = "mean", colour = "red", size = 2, geom = "point")
```

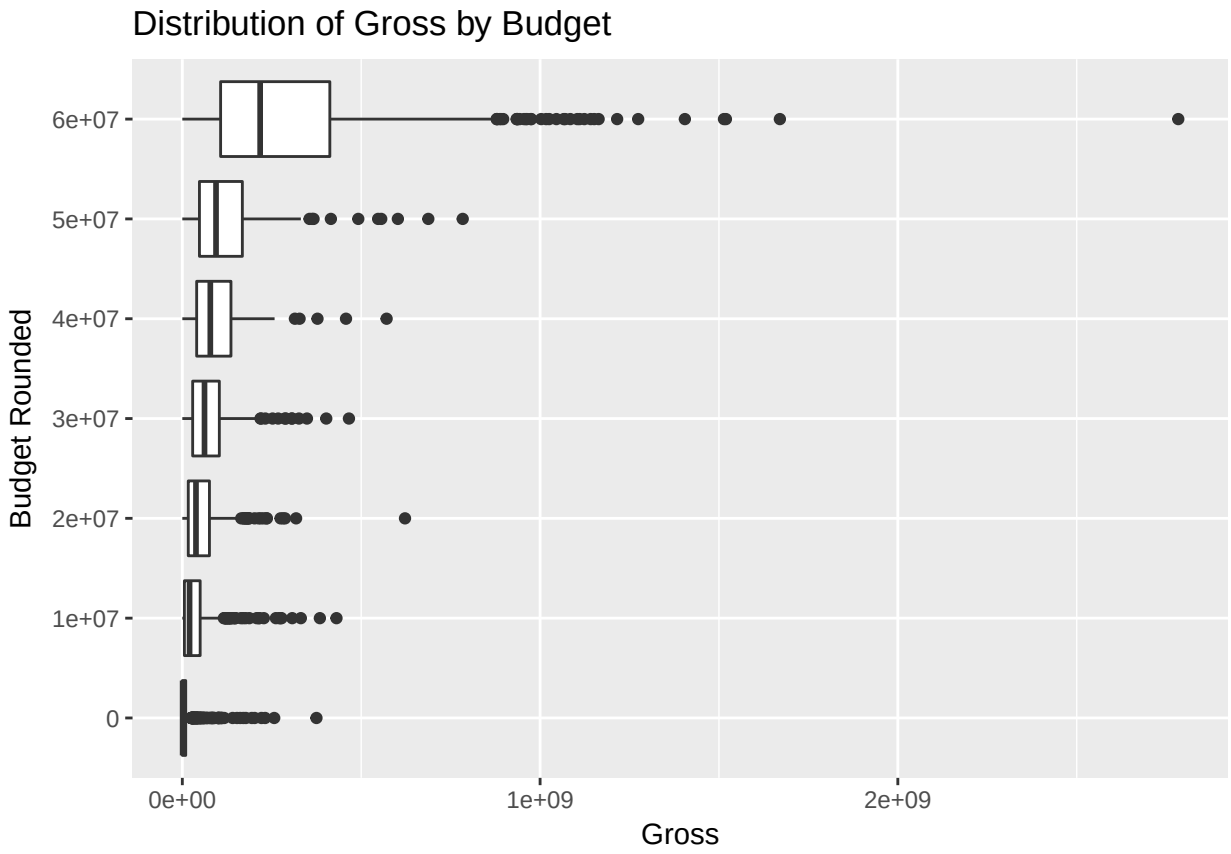
```
## Warning: Removed 22 rows containing non-finite values (stat_summary).
```



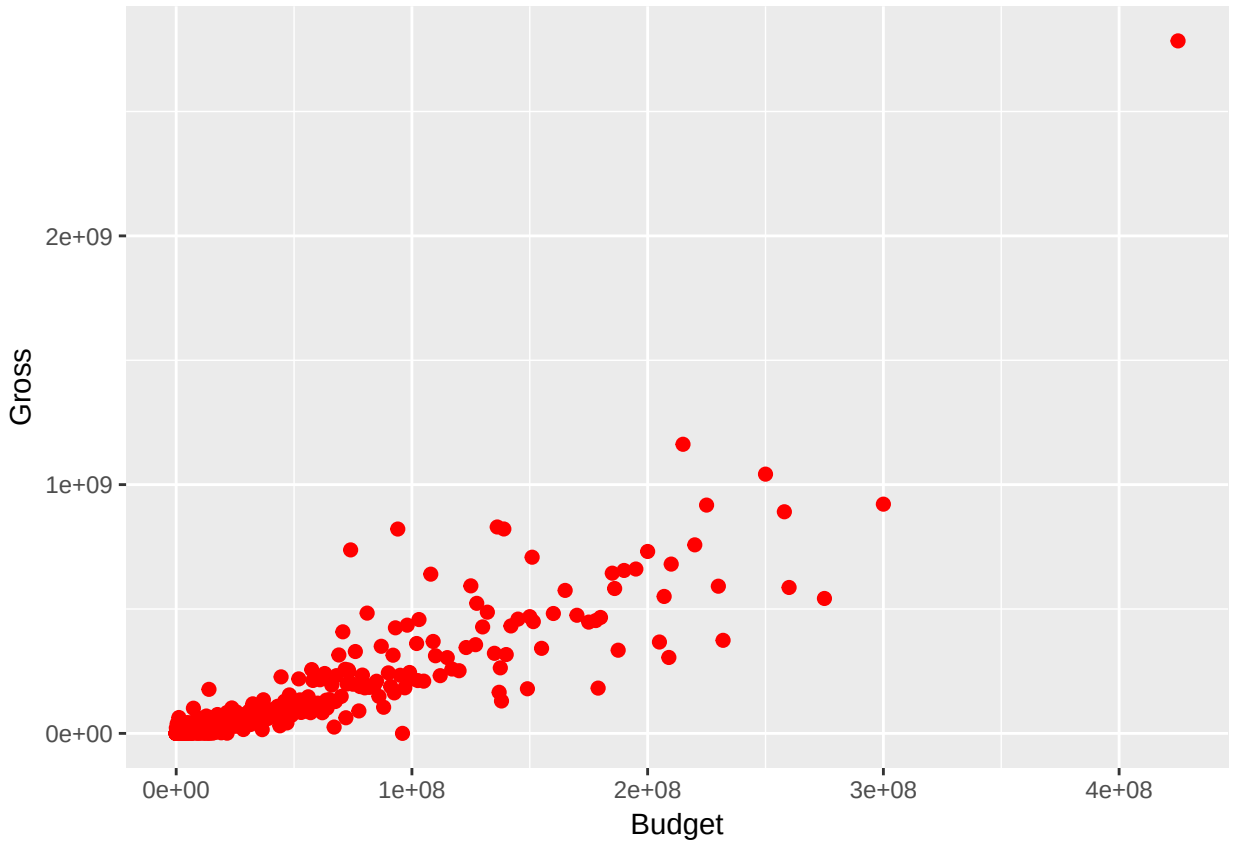
Second, I look at the relationship between Budget and Gross. This is because if we remember back to Project 1, higher budget films tended to gross much higher. Looking at the graphs below, it seems like $60M would be an interesting bin, and I wanted to capture the interaction of the binned variable with `Budget` to allow for a difference in slope after $60M.

```
# Distribution of Gross by Budget
# Now we need to show distribution of runtime by Budget. Do a boxplot grouped by budget
df$budget_rounded = round_any(df$Budget, 10000000, f = floor)
df$budget_rounded[df$Budget >= 60000000] = 60000000
df$budget_rounded = as.character(df$budget_rounded)
df$budget_rounded[df$budget_rounded == '600000000'] = 'Over 60M'
```

```
ggplot(df, aes(as.factor(budget_rounded), Gross)) +
  geom_boxplot() +
  coord_flip() +
  scale_x_discrete("Budget Rounded") +
  ggtitle('Distribution of Gross by Budget')
```

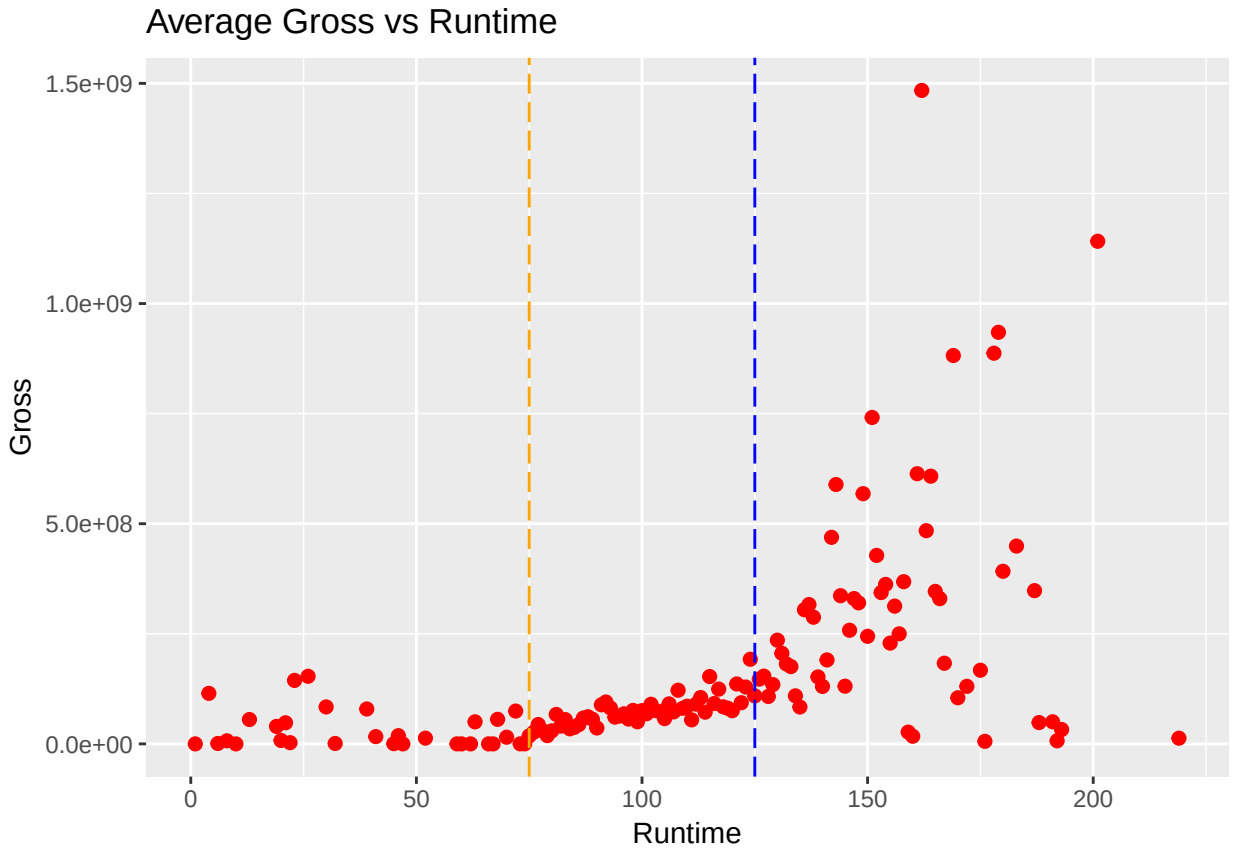



```
ggplot(df, aes(Budget, Gross)) +
  stat_summary(fun.y = "mean", colour = "red", size = 2, geom = "point")
```



Next, with regards to `Runtime` recall from Part 2 we saw an interesting relationship between the bins of pre-75, between 75 and 125, and after 125 (reproduced visual below). To capture this interaction, we interacted `Runtime` and `Runtime2` with the binned variables.

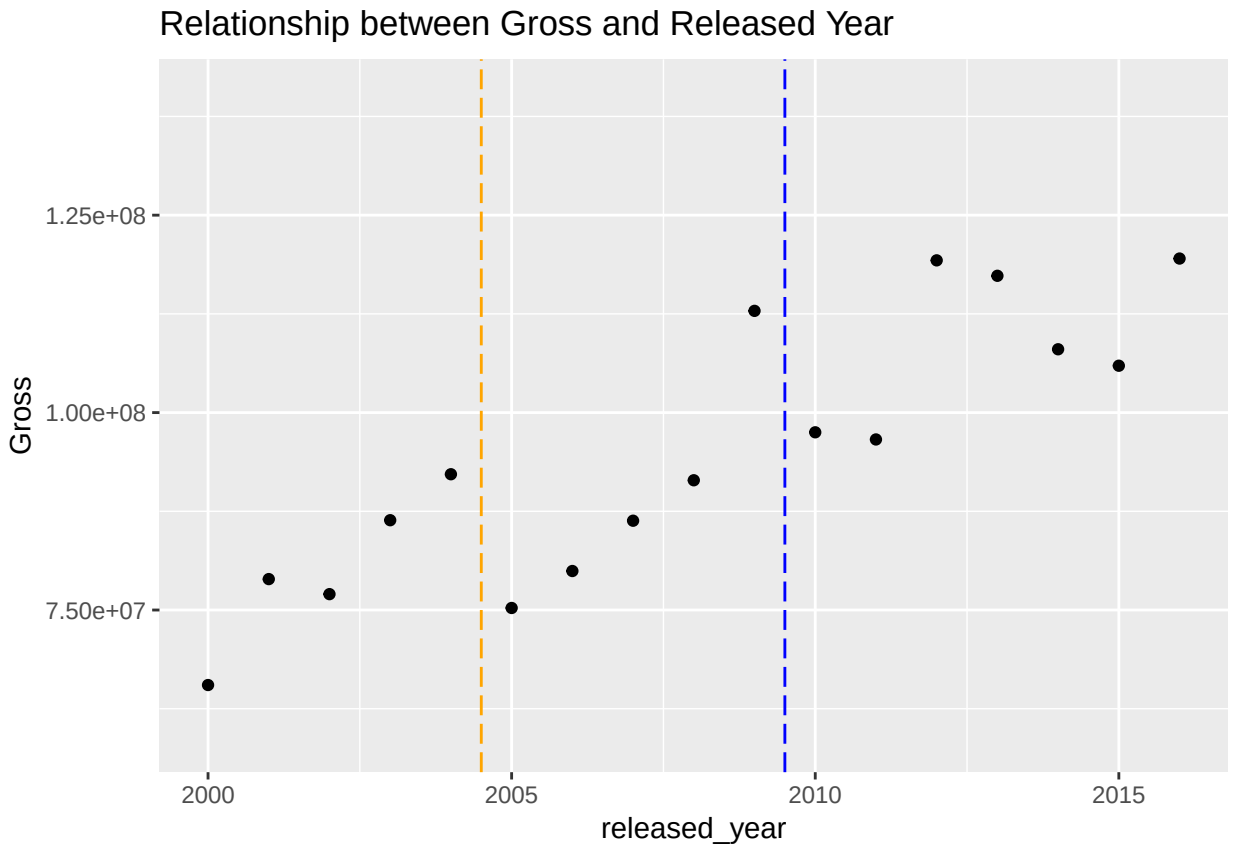
```
ggplot(df, aes(Runtime, Gross)) +
  stat_summary(fun.y = "mean", colour = "red", size = 2, geom = "point") +
  geom_vline(xintercept=75, color='orange', linetype='longdash') +
  geom_vline(xintercept=125, color='blue', linetype='longdash') +
  ggtitle('Average Gross vs Runtime')
```



Next, with regards to `released_year` recall from Part 2 we saw an interesting relationship, potentially explained by Economic business cycles. Below, we reproduce the visualization, and to allow for the model to capture this interesting effect, we interact `released_year` with the binned variables.

```
ggplot(df, aes(released_year, Gross)) +
  geom_point(stat='summary') +
  geom_vline(xintercept=2004.5, color='orange', linetype='longdash') +
  geom_vline(xintercept=2009.5, color='blue', linetype='longdash') +
  ggtitle('Relationship between Gross and Released Year')
```

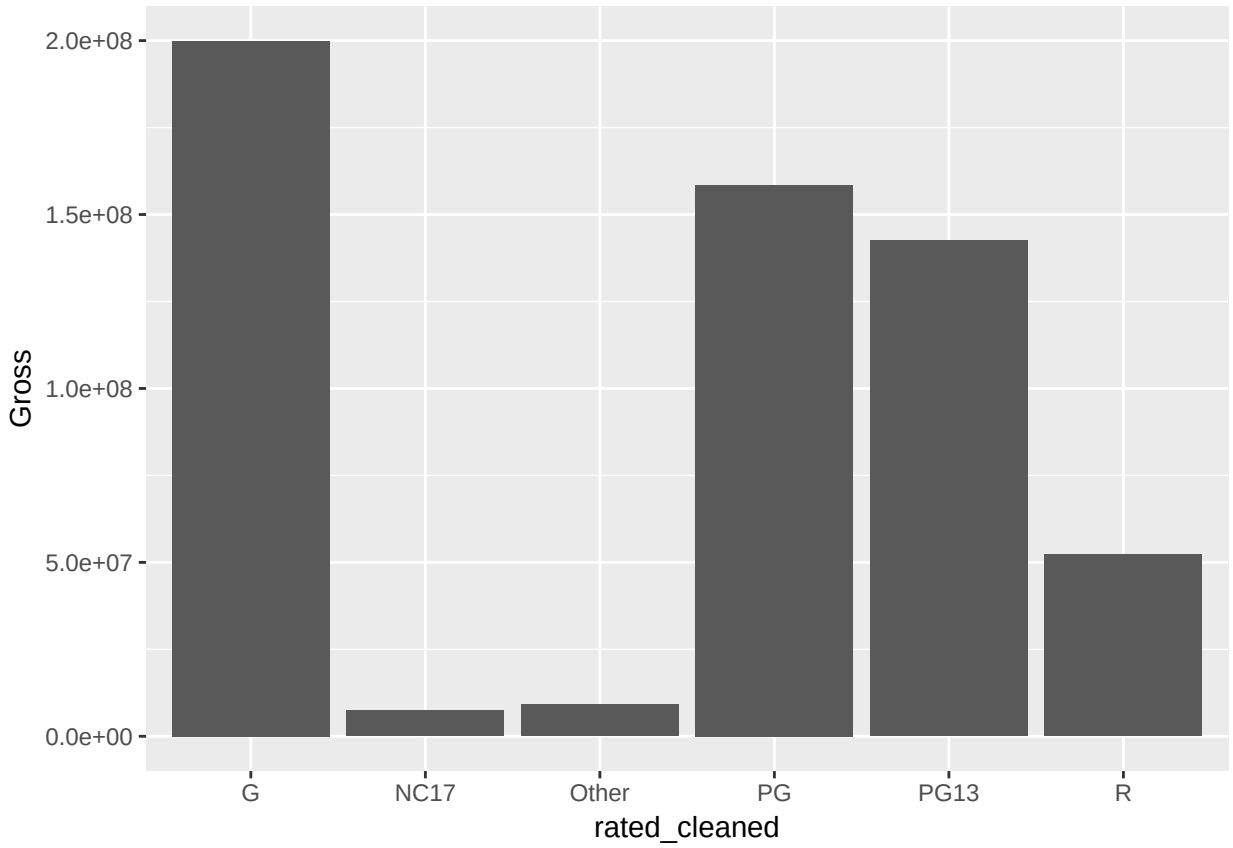
```
## No summary function supplied, defaulting to `mean_se()
```



Next, I wanted to explore MPAA Ratings more because I noticed a pretty significant difference in Gross by different MPAA Ratings. In particular, we can see from the graph below that:

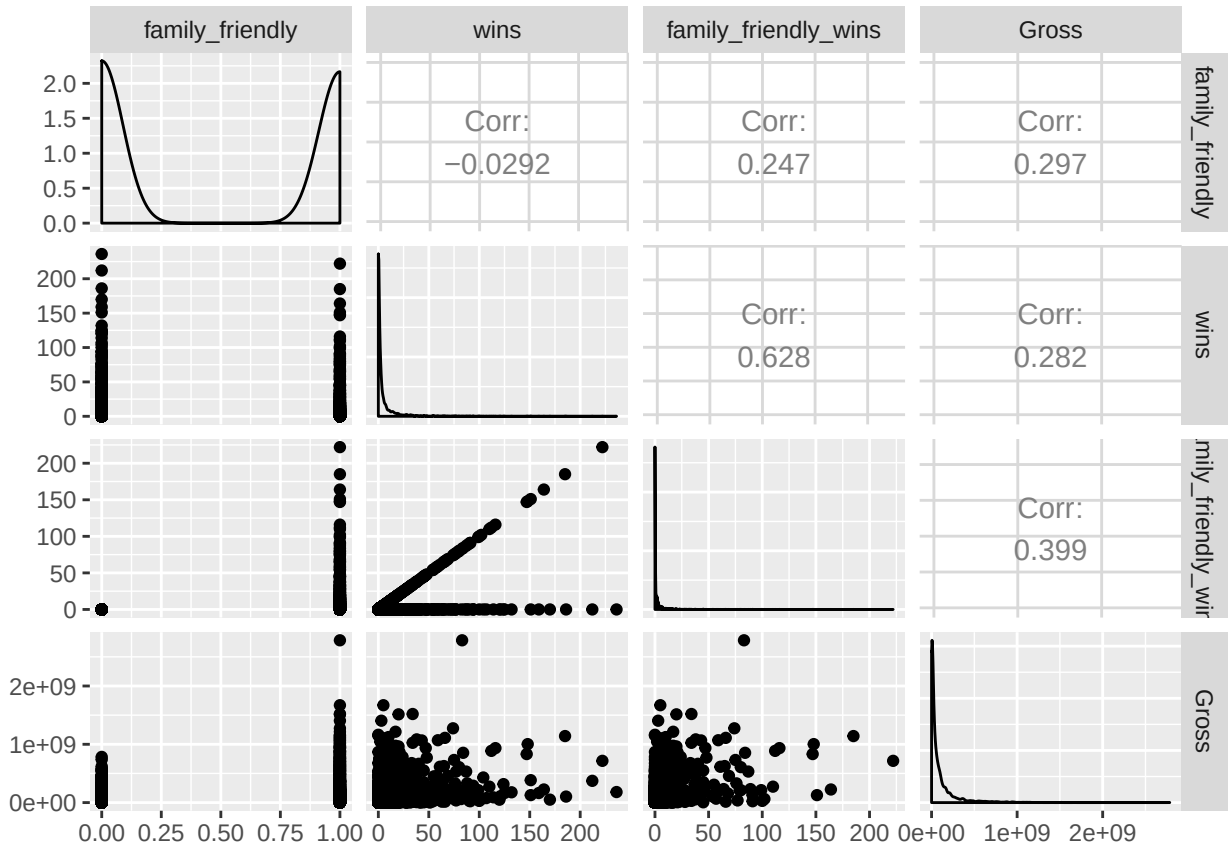
```
ggplot(df, aes(rated_cleaned, Gross)) +
  geom_bar(stat='summary')
```

```
## No summary function supplied, defaulting to `mean_se()
```

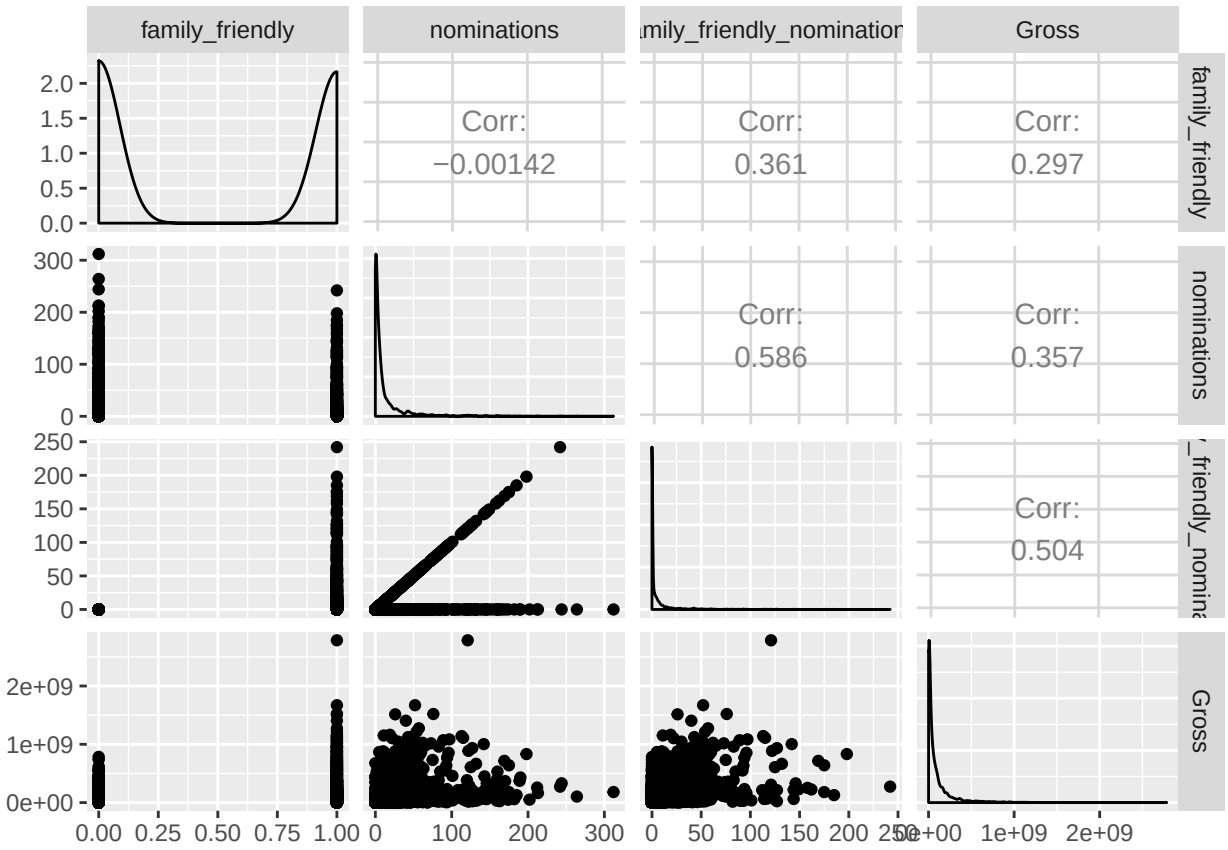


It looks like G, PG, and PG13 rated movies gross much higher on average than other MPAA rated movies. This makes sense as these types of movies are more "family friendly" and can relate to a much bigger audience. Thus, I created a binary variable `family_friendly` and interacted it with `wins` and `nominations` to see if there was any meaningful interactions. If we look at the ggpairs, it looks promising.

```
# Plotting Correlation Plots of the different Review number Columns:
ggpairs(df[c('family_friendly', 'wins', 'family_friendly_wins', 'Gross')])
```




```
ggpairs(df[c('family_friendly', 'nominations', 'family_friendly_nominations', 'Gross')])
```



Among other things from the `ggpairs` graph above, you can see that the correlation between `Gross` vs. `wins` and `nominations` is fairly high (approximately 0.3 for each), but interacted with `family_friendly`, the correlation goes up to almost 0.5! Thus, it seems like there is a meaningful interaction between `wins`, `nominations` and `family_friendly`. This essentially means that for family friendly movies, winning and/or being nominated for an award has functionally different relationship with Gross than non-family friendly movies.

Q: Comment on the final RMSE values you obtained, and what you learned through the course of this project.

A: For Model 5, we can directly compare the best test RMSEs:

```
ggplot(data=df_eval_mod5, aes(x=train_size, y=rmses)) +
  geom_line(aes(color=rmse_type, group=rmse_type)) +
  ggtitle('Model 5: Training and Test RMSE')
```


```
print(eval_model1$best_test_rmse)
```

```
## [1] 87450682
```

```
print(eval_model2$best_test_rmse)
```

```
## [1] 86458175
```

```
print(eval_model3$best_test_rmse)
```

```
## [1] 127798239
```

```
print(eval_model4$best_test_rmse)
```

```
## [1] 84986467
```

```
print(eval_model5$best_test_rmse)
```

```
## [1] 83328591
```

The improvement of Model 5 compared to Model 4 was:

```
improvement_in_rmse_model5 = eval_model5$best_test_rmse - eval_model4$best_test_rmse
percentage_improvement_in_rmse_model5 = 1.0 * abs(eval_model5$best_test_rmse - eval_model4$best_test_rms
print('Improvement in RMSE')
```

```
## [1] "Improvement in RMSE"
```

```
print(improvement_in_rmse_model5)
```

```
## [1] -1657875
```

```r
# As a percentage
print(percentage_improvement_in_rmse_model5)
```

```
## [1] 0.01950752
```

Notice that the improvement in Model 5 was -1.4717082×10^6 which represents a further 1.9507523 percent improvement over Model 4! In addition, notice that the all-in improvement of Model 5 compared to Model 1 is -4.1220911×10^6 which represents a 4.713618 percentage improvement!

```r
# Best test RMSE
best_testrmse_mod5 = eval_model5$best_test_rmse
best_train_size_mod5 = eval_model5$best_train_size
```

The best mean test RMSE value I observed for Model 5 was 8.3328591×10^7 and I observed that at the training set size 100 percent. Notice again, we see a similar pattern as other models. From the graph above, notice that for Model 5 that we see that Test RMSE generally decreases with higher training set. Again, this makes sense because with very low training set size, the model will overfit the more limited data.

This project was immensely useful in getting experience into digging deeper about building models. Even with this relatively limited data set; on the order of 1000s of observations and 100s of features, there is practically infinite ways we can engineer new features and thus do feature selection as to what we want to put into our models.

One thing I definitely learned was that feature engineering is useful. Especially with linear regression, the relationship between an outcome variable and potential features is very rarely truly linear. Exploring which transformations and interactions improve the model is important. However, this is not riskless!!! Because of curse of dimensionality and bias/variance tradeoff, you cannot just put whatever you want into the model. Thus, the hold-out test set is important.

Another thing that I learned is the importance of training/test split, and I found the relationship with size of training set interesting. The test error seemed to generally decrease and generally the lowest test error was found when trained on 100% of training set. This is a key lesson since if you train on small sample size, the model will overfit the small training set (even with a relatively inflexible method like linear regression!) Thus, always be careful of overfitting and never trust the training error.

If I wanted to take this a step forward, I would definitely want to include some kind of validation set/procedure (e.g. K-fold validation). However, given the scope of this project assignment and this Piazza post: https://piazza.com/class/j6gt7ycx6nk145?cid=1123, I did not do so here. However, this would be one tool to help us figure out the best features to include in our model. For example, this would be a good tool to help us figure out what power transformations to use.