Predicting the Number of Song Streams on Spotify

Lydia Kim

2023-03-19

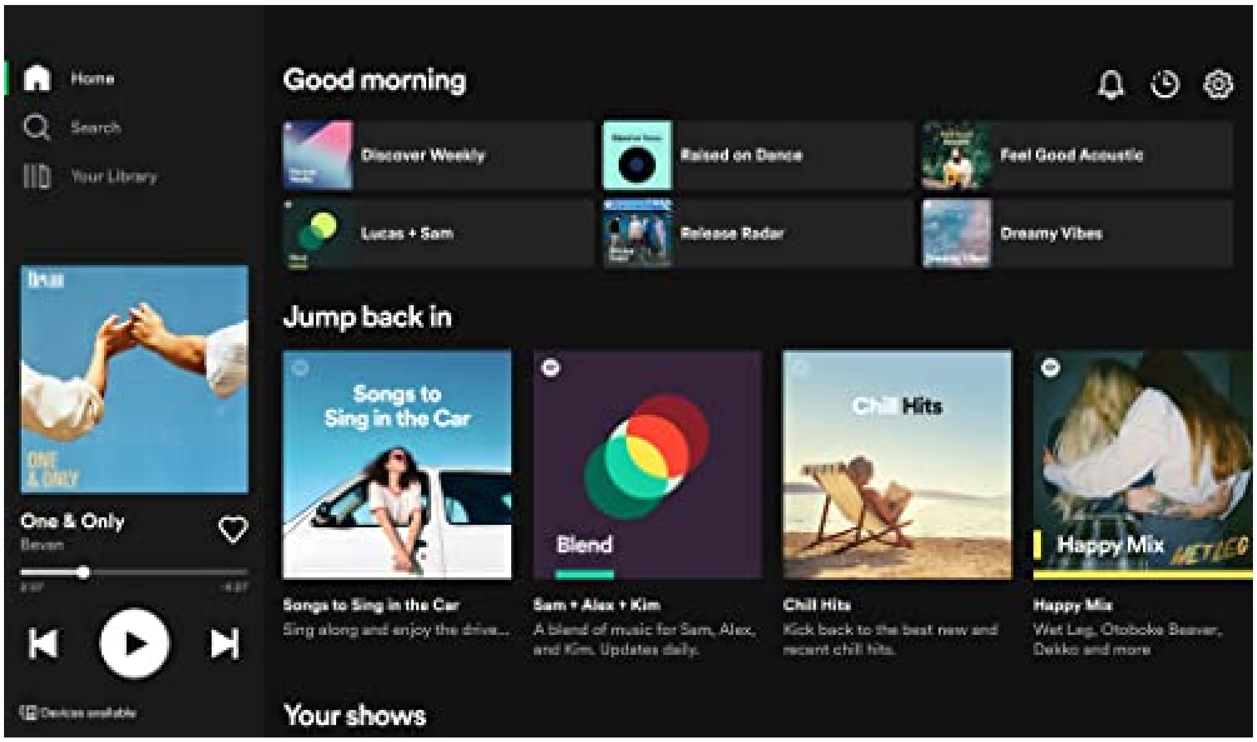
Contents

Introduction

The goal of this project is to build a model that predicts the number of streams (in billions) of different songs on Spotify.

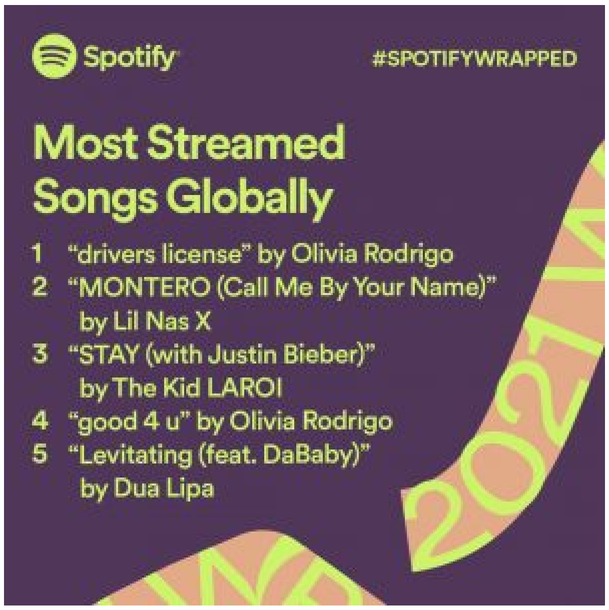
What is spotify?

If you already did not know, Spotify is a digital music service that gives you access to unlimited songs from all over the world. There's an option to subscribe for an ad-free experience, or you could simply use it for free with the occasional advertisement. People of all ages use Spotify due to the wide variety of music that it offers.



Our Goal

Within a platform that contains millions of different songs, there is bound to be some more popular than others. Our goal is to predict the number of streams a song uploaded on Spotify would get depending on certain variables such as energy, danceability, valence, and more. To do this, we will start by creating our visual EDA to get an idea of the variables relationships. Then we will create a recipe and begin building our models! Our hope is by the end, we find the model best suited for our data.


Spotify
#SPOTIFYWRAPPED
Most Streamed Songs Globally
1 "drivers license" by Olivia Rodrigo
2 "MONTERO (Call Me By Your Name)" by Lil Nas X
3 "STAY (with Justin Bieber)" by The Kid LAROI
4 "good 4 u" by Olivia Rodrigo
5 "Levitating (feat. DaBaby)" by Dua Lipa
2021

Who Could Benefit From This Data?

The most obvious answer to who could benefit from our analysis is anyone who uploads music on Spotify. Using these statistics, they could alter their music to match the songs that are currently topping the charts. However, any regular Spotify user could also benefit from this data. Let's say you discover a new upcoming artist. By looking into the predictor variables I mentioned before, they could have an idea of whether or not this new artist would blow up.

With that being said, let's load our packages and get started!

Loading Packages and Raw Data

```r
# packages
library(tidyverse)
library(dplyr)
library(tidymodels)
library(readr)
library(kknn)
library(janitor)
library(ISLR)
library(discrim)
library(poissonreg)
library(glmnet)
library(corrr)
library(corrplot)
library(randomForest)
library(xgboost)
library(rpart.plot)
library(vip)
library(ranger)
library(tidytext)
library(ggplot2)
theme_set(theme_bw())

# assigning the data to a variable
streams <- read_csv("data/unprocessed/Streams.csv") # predictor - # of streams
features <- read_csv("data/unprocessed/Features.csv") # features of each song

# renaming the 'song name' column in the streams data set to 'name' to match the features data set
streams <- streams %>%
  rename(
    name = Song
    )

# merging data sets into one
raw_songs <- merge(streams,features,by='name')
write.csv(raw_songs, "/Users/lydiakim/Documents/Pstat 131/Final project/data/unprocessed/raw_songs.csv"

# making variable names easier to access
raw_songs <- as_tibble(raw_songs) %>%
  clean_names()
```

```r
# setting seed for consistency
set.seed(0541)
```

```r
# exhibiting the first 6 rows of the data
head(raw_songs)
```

```
## # A tibble: 6 x 17
##   name   artist strea~1 relea~2 id    durat~3 energy   key loudn~4  mode speec~5
##   <chr>  <chr>    <dbl> <chr>   <chr>   <dbl>  <dbl> <dbl>   <dbl> <dbl>   <dbl>
## 1 'Till~ Emine~    1.58 26-May~ 4xkO~    4.96  0.847     1   -3.24     1  0.186
## 2 7 Rin~ Arian~    1.92 18-Jan~ 6ocb~    2.98  0.317     1  -10.7      0  0.334
## 3 7 Yea~ Lukas~    1.53 16-Jun~ 5kqI~    3.96  0.473    10   -5.83     1  0.0514
## 4 All o~ John ~    1.97 12-Aug~ 3U4i~    4.49  0.264     8   -7.06     1  0.0322
## 5 As It~ Harry~    2.01 01-Apr~ 4LRP~    2.79  0.731     6   -5.34     0  0.0557
## 6 Bad G~ Billi~    2.22 29-Mar~ 2Fxm~    3.23  0.425     7  -11.0      1  0.375
## # ... with 6 more variables: acousticness <dbl>, instrumentalness <dbl>,
## #   liveness <dbl>, valence <dbl>, tempo <dbl>, danceability <dbl>, and
## #   abbreviated variable names 1: streams_billions, 2: release_date,
## #   3: duration, 4: loudness, 5: speechiness
```

This data set, Most Streamed Songs (All Time), was downloaded from Kaggle and uploaded by user Amaan Ansari. Link: https://www.kaggle.com/datasets/amaanansari09/most-streamed-songs-all-time?select=Streams.csv Our raw data came in two different files, so we merged them using the names column for consistent data!

Tidying Raw Data

Variable Selection

Since we are given all variables in the raw data set, let's explore them to see which ones would be significant in building our model!

```
# displaying the number of rows and columns
dim(raw_songs)
```

```
## [1] 100  17
```

We have 100 rows and 17 columns, meaning there's 100 songs and 17 variables. That's not a long of songs we have to work with, so we will keep in mind that the data we are working with is the top 100 songs. We have 16 potential predictor variables. Since name, artist, release_date, and id are character variables that don't have as big of an impact of the number of streams, we will drop them from our final data. We are left with 11 predictor variables and our outcome, streams in billions.

```
# selecting the variables we have chosen
songs <- raw_songs %>%
  select(c("streams_billions", "energy", "key",  "loudness", "mode", "speechiness", "acousticness", "ins
  clean_names()

write.csv(songs, "/Users/lydiakim/Documents/Pstat 131/Final project/data/processed/cleaned_songs.csv")

# displaying the first 6 rows of our new data set
head(songs) # 11 predictor variables, response: streams
```

```
## # A tibble: 6 x 12
##   streams_b~1 energy   key loudn~2  mode speec~3 acous~4 instr~5 liven~6 valence
##         <dbl>  <dbl> <dbl>   <dbl> <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1        1.58  0.847     1   -3.24     1  0.186   0.0622 0        0.0816   0.1
## 2        1.92  0.317     1  -10.7      0  0.334   0.592  0        0.0881   0.327
## 3        1.53  0.473    10   -5.83     1  0.0514  0.287  0        0.391    0.34
## 4        1.97  0.264     8   -7.06     1  0.0322  0.922  0        0.132    0.331
## 5        2.01  0.731     6   -5.34     0  0.0557  0.342  0.00101  0.311    0.662
## 6        2.22  0.425     7  -11.0      1  0.375   0.328  0.13     0.1      0.562
## # ... with 2 more variables: tempo <dbl>, danceability <dbl>, and abbreviated
## #   variable names 1: streams_billions, 2: loudness, 3: speechiness,
## #   4: acousticness, 5: instrumentalness, 6: liveness
```

Codebook

Since our data set didn't come with a codebook file, I created my own using the descriptions on Kaggle:

- energy: Energy is a measure from 0.0 to 1.0 and represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy. For example, death metal has high energy, while a Bach prelude scores low on the scale. Perceptual features contributing to this attribute include dynamic range, perceived loudness, timbre, onset rate, and general entropy.
- key: The key the track is in. Integers map to pitches using standard Pitch Class notation.
- loudness: The overall loudness of a track in decibels. Loudness values are averaged across the entire track and are useful for comparing relative loudness of tracks. Loudness is the quality of a sound that is the primary psychological correlate of physical strength (amplitude). Values typically range between -60 and 0 db.

- mode: Mode indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived. Major is represented by 1 and minor is 0.
- speechiness: Speechiness detects the presence of spoken words in a track. The more exclusively speech-like the recording (e.g. talk show, audio book, poetry), the closer to 1.0 the attribute value. Values above 0.66 describe tracks that are probably made entirely of spoken words. Values between 0.33 and 0.66 describe tracks that may contain both music and speech, either in sections or layered, including such cases as rap music. Values below 0.33 most likely represent music and other non-speech-like tracks.
- acousticness: A confidence measure from 0.0 to 1.0 of whether the track is acoustic. 1.0 represents high confidence the track is acoustic.
- instrumentalness: Predicts whether a track contains no vocals. "Ooh" and "aah" sounds are treated as instrumental in this context. Rap or spoken word tracks are clearly "vocal". The closer the instrumentalness value is to 1.0, the greater likelihood the track contains no vocal content. Values above 0.5 are intended to represent instrumental tracks, but confidence is higher as the value approaches 1.0.
- liveness: Refers directly to reverberation time. A live room has a long reverberation time and a dead room a short reverberation time.
- valence: Describes the musical positivity conveyed by a piece of music. Songs with high valence sound more positive (e.g. happy, cheerful), while pieces with low valence sound more negative (e.g. sad, angry).
- tempo: The speed or pace of a given piece.
- danceability: Determines the ease with which a person could dance to a song over the course of the whole song.

With our selected variables defined, let's move on!

Missing Data

We must check for any missing data because they could cause issues in building our model.

```r
songs %>%
  summary()
```

```
##  streams_billions     energy            key           loudness
##  Min.   :1.521    Min.   :0.1850   Min.   : 0.00   Min.   :-12.205
##  1st Qu.:1.647    1st Qu.:0.5238   1st Qu.: 2.00   1st Qu.: -7.104
##  Median :1.770    Median :0.6400   Median : 6.00   Median : -5.971
##  Mean   :1.896    Mean   :0.6255   Mean   : 5.34   Mean   : -6.176
##  3rd Qu.:2.046    3rd Qu.:0.7410   3rd Qu.: 8.25   3rd Qu.: -4.941
##  Max.   :3.449    Max.   :0.9120   Max.   :11.00   Max.   : -2.810
##       mode       speechiness       acousticness       instrumentalness
##  Min.   :0.00   Min.   :0.02810   Min.   :0.0000255   Min.   :0.000000
##  1st Qu.:0.00   1st Qu.:0.04120   1st Qu.:0.0261500   1st Qu.:0.000000
##  Median :1.00   Median :0.05940   Median :0.1625000   Median :0.000000
##  Mean   :0.65   Mean   :0.09645   Mean   :0.2609545   Mean   :0.007848
##  3rd Qu.:1.00   3rd Qu.:0.10600   3rd Qu.:0.4315000   3rd Qu.:0.000086
##  Max.   :1.00   Max.   :0.43800   Max.   :0.9450000   Max.   :0.459000
##     liveness         valence           tempo         danceability
##  Min.   :0.0344   Min.   :0.0612   Min.   : 74.90   Min.   :0.3400
##  1st Qu.:0.0943   1st Qu.:0.3407   1st Qu.: 98.02   1st Qu.:0.5480
##  Median :0.1140   Median :0.4920   Median :116.86   Median :0.6715
##  Mean   :0.1657   Mean   :0.5020   Mean   :121.25   Mean   :0.6496
##  3rd Qu.:0.2170   3rd Qu.:0.6630   3rd Qu.:142.44   3rd Qu.:0.7592
##  Max.   :0.7900   Max.   :0.9690   Max.   :186.00   Max.   :0.9210
```

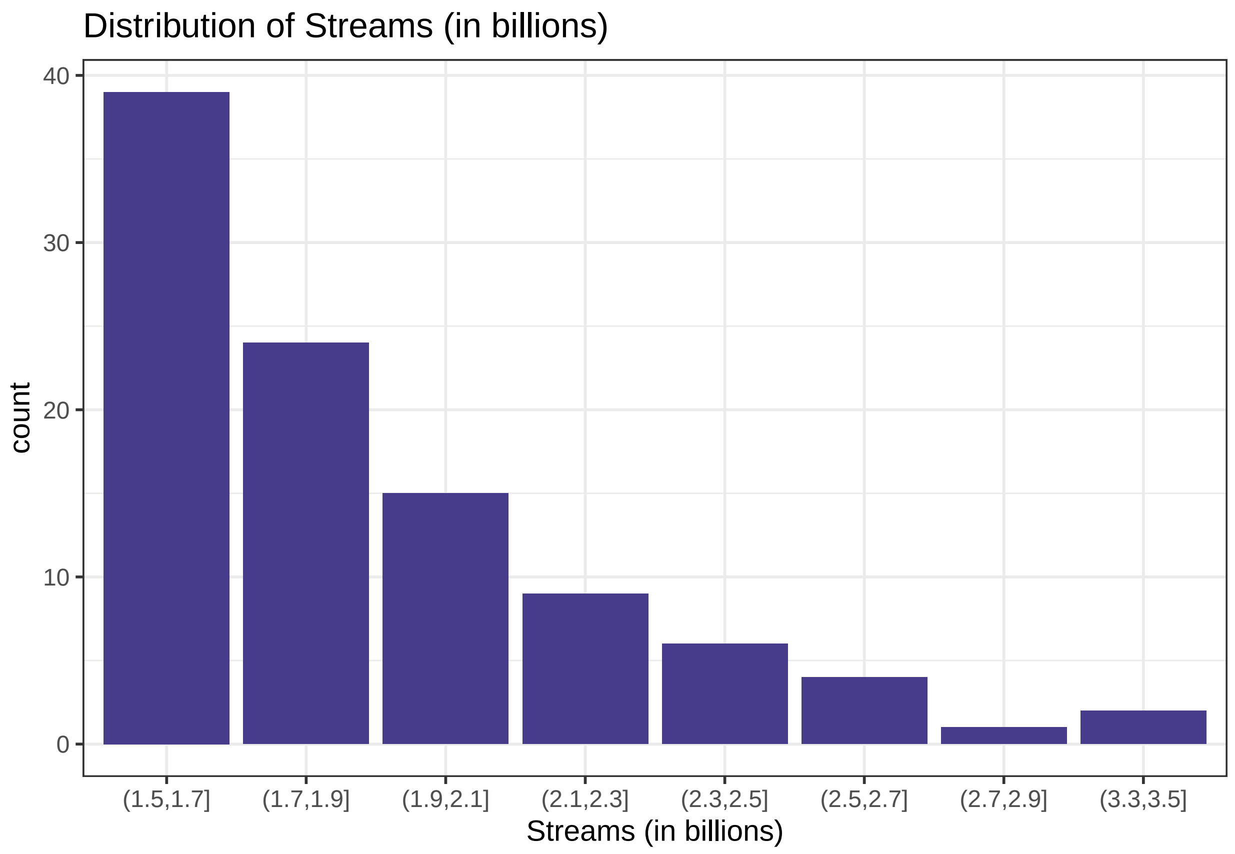
Looks like we have no missing data! This makes things much easier for us the following steps.

Visual EDA

Streams in Billions

First, let's explore the distribution of our response variable, streams_billions.

```r
# Distribution of streams

# we split the x axis values into intervals for better data representation
ggplot(songs, aes(cut(streams_billions, breaks=seq(1.5,3.5,0.2)))) +
  geom_bar(fill='slateblue4') +
  labs(
    title = "Distribution of Streams (in billions)",
    x = "Streams (in billions)"
  )
```

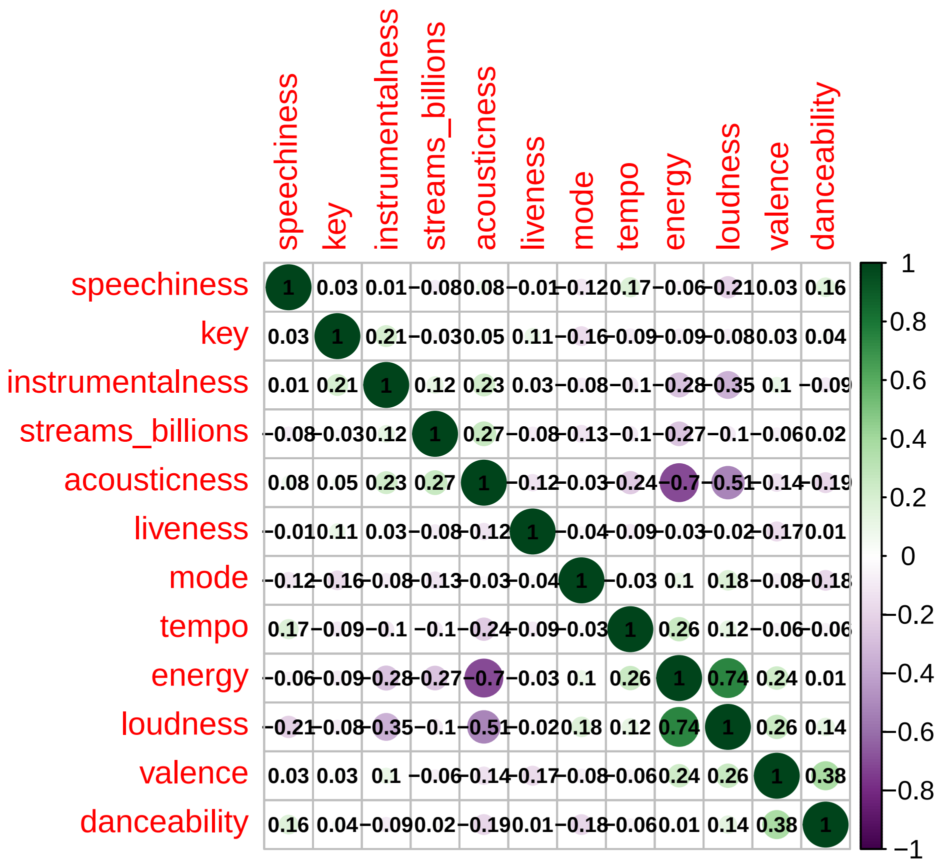


According to our graph, we can see as the stream count goes up, there's a smaller frequency of songs. This is expected, of course, since we are looking at the highest number of streams on Spotify. Only a handful of songs in our data reach past 2.5 billion streams. There is an obvious peak in the 1.5-1.7 billion stream count interval.

Variable Correlation Plot

Let's make a correlation heat map of the numeric variables to get a sense of each of their relationships. Since all of our variables are numeric, we don't need to filter out non-numeric variables!

```
songs_cor <- cor(songs)  # calculating the correlation between each variable
songs_corrplot <- corrplot(songs_cor,  # making the correlation plot
                           order = 'AOE',
                           col = COL2("PRGn"), # purple and green color combination
                           addCoef.col=1, # displaying a numerical value
                           number.cex=0.7)
```



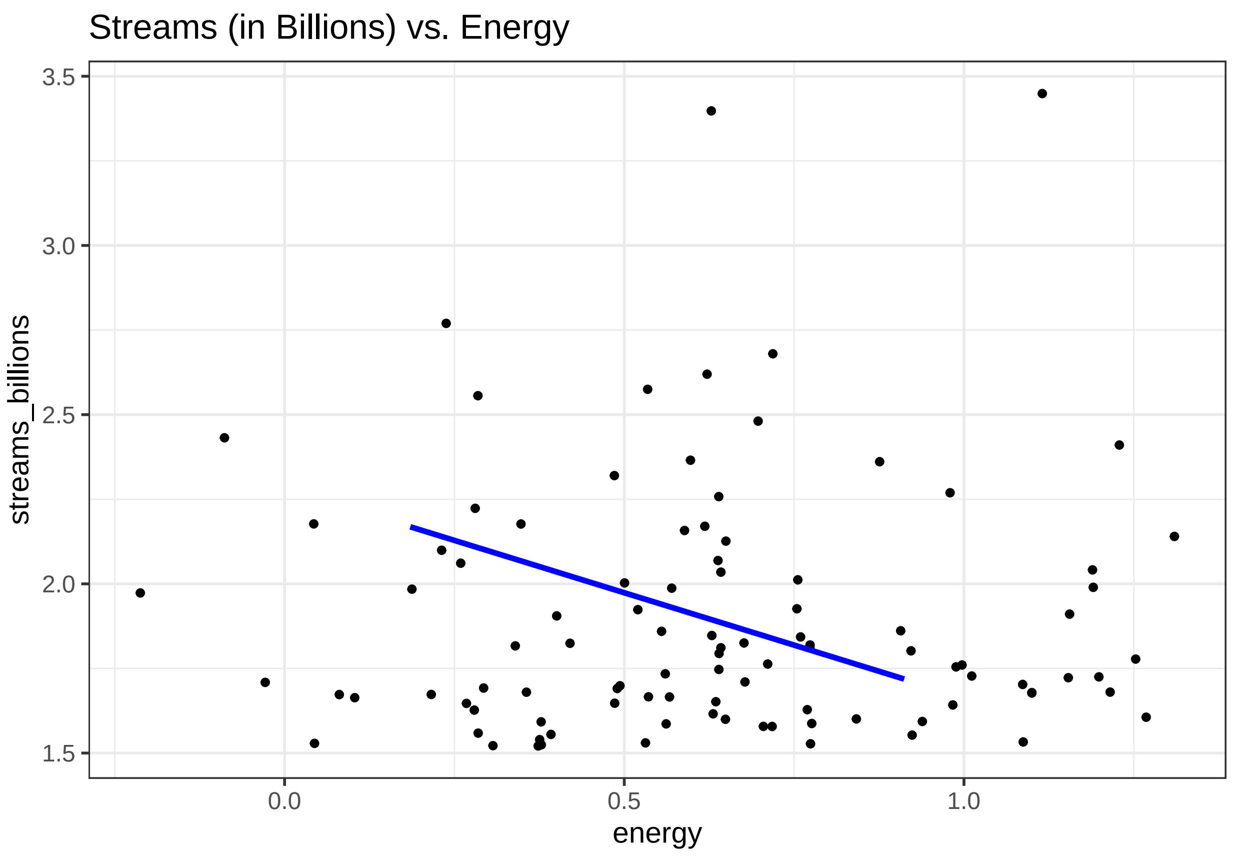
As seen in the plot, there is not a high correlation between a lot of our variables. In particular, streams and energy have a very low correlation. We also see that our response variable has the highest correlation with acousticness and instrumentalness. Both variables also seem to have a low correlation with energy and loudness. We are starting to get a feel for which variables will end up having the biggest impact on our outcome. Let's explore these relationships some more!

Energy

Since we discovered that streams and energy have the lowest correlation, let's dig into their relationship further.

```r
songs %>%
  ggplot(aes(x=energy, y=streams_billions)) +
  geom_jitter(width = 0.5, size = 1) +
  geom_smooth(method = "lm", se =F, col="blue") +
  labs(title = "Streams (in Billions) vs. Energy")
```

```
## `geom_smooth()` using formula = 'y ~ x'
```



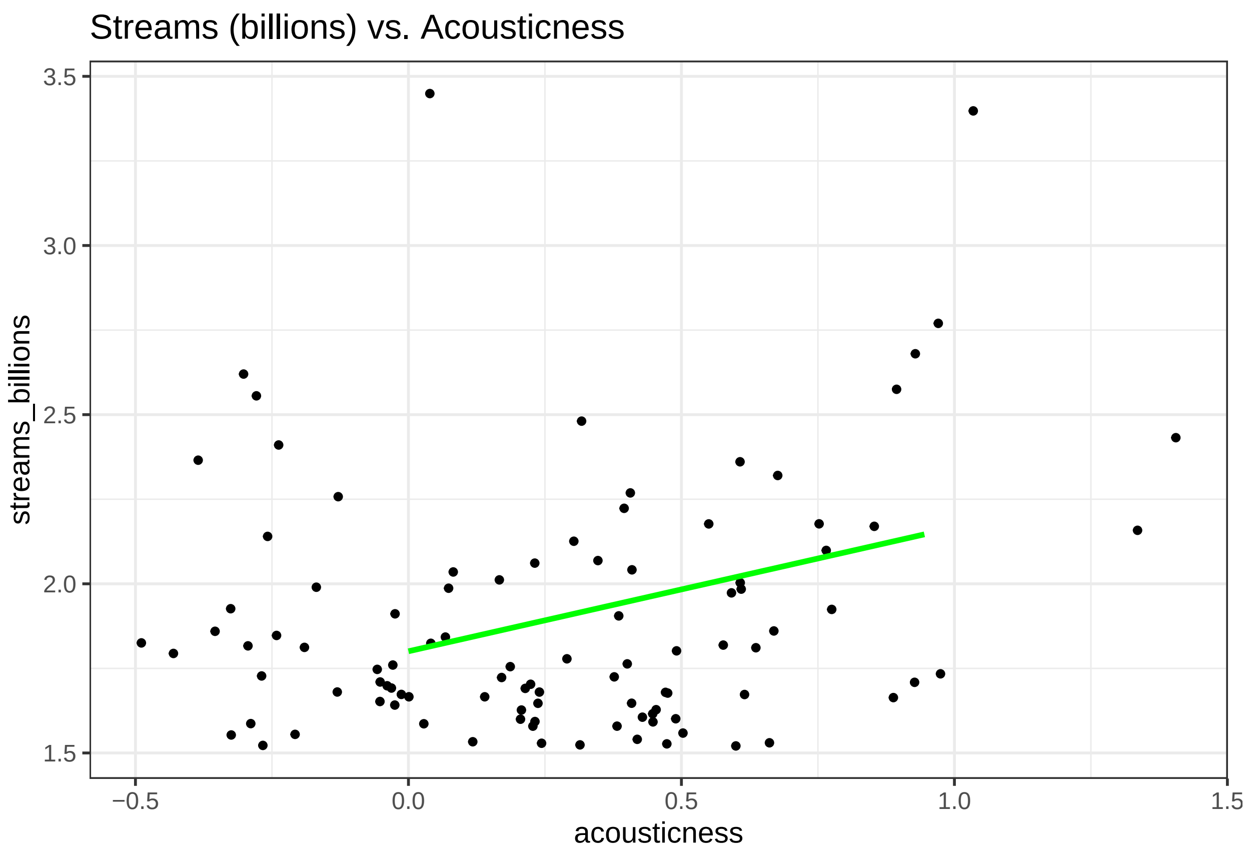
The data points are quite scattered, but ultimately there is a very obvious negative relationship between streams and energy. This means that the higher energy in the songs, the less amount of times streamed. This makes sense if you consider that high energy songs are normally played in specific occasions such as working out and parties. These songs could be popular but people would most likely repeat more calm songs in their day-to-day lives. Therefore, the lower-energy songs typically have more streams. I would like to note that there are two outliers on the top right and top middle that stand out. These songs have very high energy levels, but also have the two highest streams. Looking at our data, we can see that those two songs are Blinding Lights by the Weeknd and Shape of You by Ed Sheeran! This makes me wonder what other characteristics about these songs led to such a high number of streams.

Acousticness

We can see that streams and acousticness have the highest correlation. This relationship backs up our previous relationship between streams and energy, since songs that have more acoustics are typically lower-energy. Let's explore this relationship further.

```
songs %>%
  ggplot(aes(x=acousticness, y=streams_billions)) +
  geom_jitter(width = 0.5, size = 1) +
  geom_smooth(method = "lm", se =F, col="green") +
  labs(title = "Streams (billions) vs. Acousticness")
```

```
## `geom_smooth()` using formula = 'y ~ x'
```



We can see that the relationship here is almost directly opposite to the previous relationship. This observation is backed up by looking at the extremely low correlation between energy and acousticness in our correlation plot. Thus, we conclude that the lower the energy, the higher the acousticness and number of streams! Again, we see two outliers on the top area. They are the same songs as before, meaning those two songs go against the usual correlation and made it to the top of the charts with some other qualities.

Setting up for the Models

Now we can start fitting models to our song data to see if our predictors can successfully predict the number of streams in billions. We will start by splitting our data, creating the recipe, and creating the folds for k-fold cross validation.

Data Split

Before we fit any of our models, we need to split our data into a training and testing set. The training set will train our models, and as for the testing data, our models will not be able to train on the testing data. Then, we fit the best model to our testing data using the lowest RMSE value. We will then be able to see how it performs on new data. The purpose of splitting our data is to prevent over-fitting. I split the data 70/30, meaning that 70% of the data used will be for training data and 30% will be for testing data. That means most of our data will be used for training and we will still have enough data to test the models. I also stratified the data on the response variable streams_billions, so that both data have an equal distribution of the response.

```
# setting the seed for consistent results
set.seed(1541)

# splitting the data and stratifying on streams)
```

```r
song_split <- initial_split(songs, prop = 0.7, strata = streams_billions)
song_train <- training(song_split)
song_test <- testing(song_split)

# verify we split the data correctly, must equal 1
nrow(song_train)/nrow(songs)
```

```
## [1] 0.68
```

```r
nrow(song_test)/nrow(songs)
```

```
## [1] 0.32
```

Since our training data contains 68% of the data and the testing data contains 32% of the data, we have verified that we split our data correctly!

Recipe Creation

We will now create our recipe for all the models using our predictor variables and response variable. Each model will work with this recipe using whatever methods are associated with that model. We will be using all of our predictors: “energy”, “key”, “loudness”, “mode”, “speechiness”, “acousticness”, “instrumentalness”, “liveness”, “valence”, “tempo”, and “danceability”.

We will also center and scale all predictor variables to normalize them.

```r
# creating the recipe
song_recipe <- recipe(streams_billions ~ ., data = songs) %>%
  # normalizing
  step_center(all_predictors()) %>%
  step_scale(all_predictors())
```

K-Fold Cross Validation

We will conduct k-fold stratified cross validation using 10 folds. This means that each observation in the training data will be assigned to 1 of 10 folds. A testing set is created for each fold and the remaining k-1 folds will act as the training data for that fold. Then, we will end up with a total of k folds. Cross validation provides us with more testing accuracy rather than just fitting and testing models on the training set. Since variation reduces as n increases, we take the mean accuracy from multiple samples instead of just one.

Again, we stratify on the response, streams_billions, to ensure that the data in each fold is balanced.

```r
# creating folds
song_folds <- vfold_cv(song_train, v = 10, strata = streams_billions)
```

Model Building

Now we can start building our models! Some models take a lot more time to run than others, so we will save the results from each model so that we don’t have to rerun them each time. We will use the RMSE (root mean squared error) value as the metric because it typically works for all models. It is commonly used in regression and displays how close the model’s predicitons are from the true values. Therefore, we are aiming for a lower RMSE because that means our models are performing well. RMSE measures distance, which is why we normalized our data in previous steps. I have chosen 4 models to fit to our data: Linear regression, ridge regression, K Nearest Neighbors, and random forest. We will see which models performed the best and perform a deeper analysis on it!

Fitting the models

We will fit each model using the following steps:

1. Specify which model you are fitting, the parameters we are tuning, the engine the model comes from, and the mode if applicable (in our case regression).

```r
# Linear regression
lm_model <- linear_reg() %>%
  set_engine("lm")

# Ridge regression
# Tuning parameter: penalty
# Setting mixture to 0 to specify that it's ridge regression
ridge_spec <- linear_reg(mixture = 0,
                         penalty = tune()) %>%
  set_mode("regression") %>%
  set_engine("glmnet")

# K Nearest Neighbors
# Tuning parameter: the # of neighbors
knn_model <- nearest_neighbor(neighbors = tune()) %>%
  set_mode("regression") %>%
  set_engine("kknn")

# Random forest
# Tuning parameters: mtry (# of predictors), trees, and min_n (# of minimum values in each node)
rf_spec <- rand_forest(mtry = tune(),
                       trees = tune(),
                       min_n = tune()) %>%
  set_engine("ranger", importance = "impurity") %>%
  set_mode("regression")
```

2. Define the model workflow and add the model/recipe.

```r
# Linear regression
lm_workflow <- workflow() %>%
  add_model(lm_model) %>%
  add_recipe(song_recipe)

# Ridge regression
ridge_workflow <- workflow() %>%
  add_recipe(song_recipe) %>%
  add_model(ridge_spec)

# K Nearest Neighbors
knn_workflow <- workflow() %>%
  add_model(knn_model) %>%
  add_recipe(song_recipe)

# Random forest
rf_workflow <- workflow() %>%
  add_recipe(song_recipe) %>%
  add_model(rf_spec)
```

3. Create a tuning grid to specify the ranges of the parameters you wish to tune and how many levels of

each. ** Note that linear regression does not have a grid because there are no tuning parameters.

```
# Ridge regression
ridge_penalty_grid <- grid_regular(penalty(range = c(0,1)), levels = 5)

# K Nearest Neighbors
knn_grid <- grid_regular(neighbors(range = c(1,15)), levels = 5)

# Random forest
rf_grid <- grid_regular(mtry(range = c(1, 11)), trees(range = c(200,1000)), min_n(range = c(5,20)), leve
```

4. Tune the model and specify the workflow, k-fold cross validation folds, and tuning grid for our chosen parameters. ** Again, note there no tuning parameters for linear regression.

```
# Ridge regression
ridge_tune <- tune_grid(
  ridge_workflow,
  resamples = song_folds,
  grid = ridge_penalty_grid
)

# K Nearest Neighbors
knn_tune <- tune_grid(
    knn_workflow,
    resamples = song_folds,
    grid = knn_grid
)

# Random forest
rf_tune <- tune_grid(
  rf_workflow,
  resamples = song_folds,
  grid = rf_grid
)
```

5. Save the tuned models to an RDS file to keep our results and save running time.

```
# Ridge regression
write_rds(ridge_tune, file = "~/Documents/Pstat 131/Final project/tuned_models/ridge.rds")

# K Nearest Neighbors
write_rds(knn_tune, file = "~/Documents/Pstat 131/Final project/tuned_models/knn.rds")

# Random forest
write_rds(rf_tune, file = "~/Documents/Pstat 131/Final project/tuned_models/rf.rds")
```

6. Load back in the saved files.

```
# Ridge regression
ridge_tuned <- read_rds(file = "~/Documents/Pstat 131/Final project/tuned_models/ridge.rds")

# K Nearest Neighbors
knn_tuned <- read_rds(file = "~/Documents/Pstat 131/Final project/tuned_models/knn.rds")

# Random forest
rf_tuned <- read_rds(file = "~/Documents/Pstat 131/Final project/tuned_models/rf.rds")
```

7. Collect the metrics (RMSE) of the tuned models and arrange in ascending order of mean to display the lowest RMSE value. Then, save the RMSE to a variable for comparison in the following steps.

```
# Linear regression
# Fitting the linear regression to the folds first since it didn't have tuning parameters
lm_fit <- fit_resamples(lm_workflow, resamples = song_folds)
lm_rmse <- collect_metrics(lm_fit) %>%
  arrange(mean)

# Ridge regression
ridge_rmse <- collect_metrics(ridge_tuned) %>%
  arrange(mean)

# K Nearest Neighbors
knn_rmse <- collect_metrics(knn_tuned) %>%
  arrange(mean)

# Random forest
rf_rmse <- collect_metrics(rf_tuned) %>%
  arrange(mean)
```

Model Results

Let's compare the results of all our models and see which one performed the best!

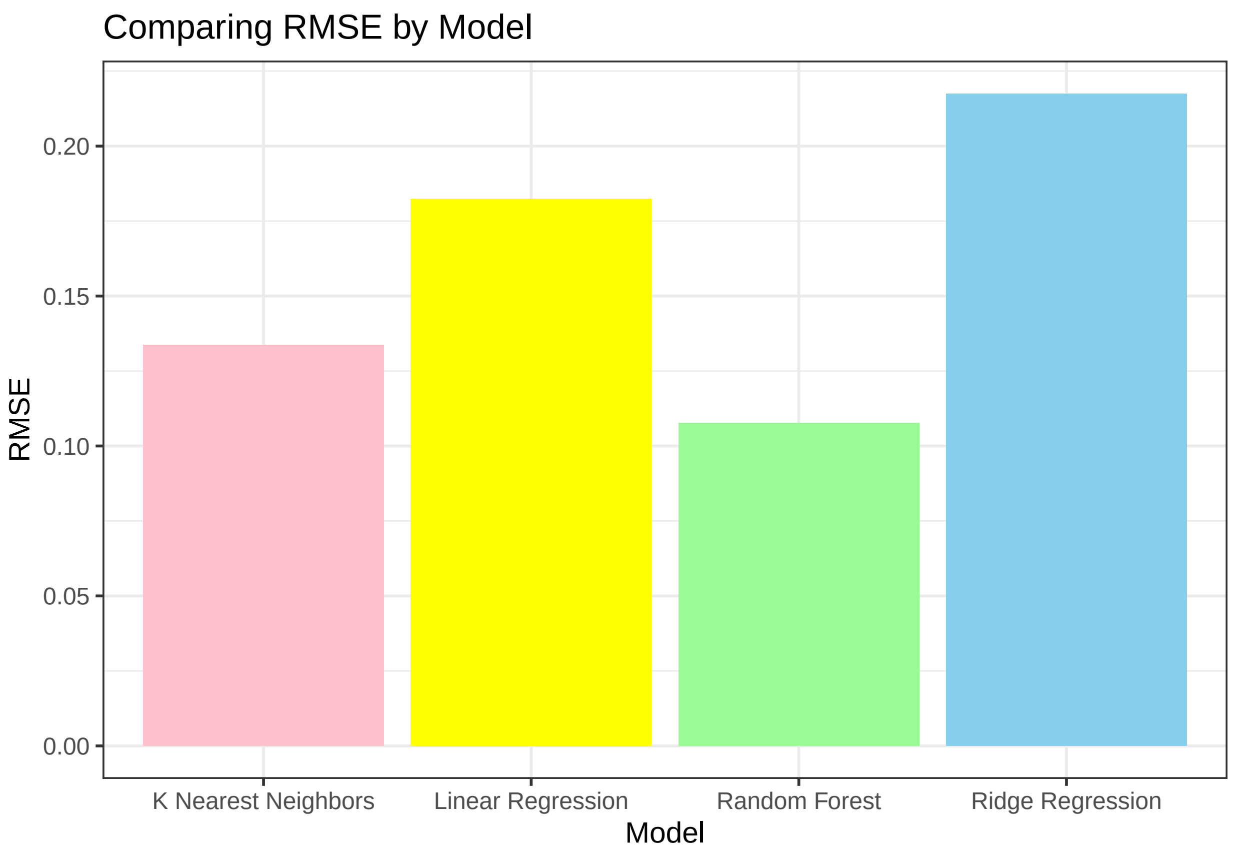
```
# Creating a tibble of all the models and their RMSE
best_model<- tibble(Model=c('Linear Regression', 'Ridge Regression', 'K Nearest Neighbors', 'Random Fore

# Arranging by lowest RMSE
best_model <- best_model %>%
  arrange(RMSE)

best_model
```

```
## # A tibble: 4 x 2
##   Model                RMSE
##   <chr>               <dbl>
## 1 Random Forest       0.108
## 2 K Nearest Neighbors 0.134
## 3 Linear Regression   0.182
## 4 Ridge Regression    0.218
```

Visualization of the Models and their RMSE

Let's graph our results for better visualization!

```
# Creating a data frame of the RMSE's
models <- data.frame(Model = c('Linear Regression', 'Ridge Regression', 'K Nearest Neighbors', 'Random 

# Creating a barplot of the RMSE values for easy comparison
ggplot(models, aes(x=Model, y=RMSE)) +
  geom_bar(stat = "identity", aes(fill = Model)) +
  scale_fill_manual(values = c("pink", "yellow", "palegreen", "skyblue")) +
  theme(legend.position = "none") +
  labs(title = "Comparing RMSE by Model")
```



Based on the graph and the tibble above, we can see that our random forest model performed the best with the lowest RMSE value of 0.1077443! Following random forest, we have K Nearest Neighbors with a RMSE of 0.1336420, Linear Regression with a RMSE of 0.182, and Ridge Regression with a RMSE of 0.2175064! With the exception of ridge regression, our values are very close. This means all our models performed very well.

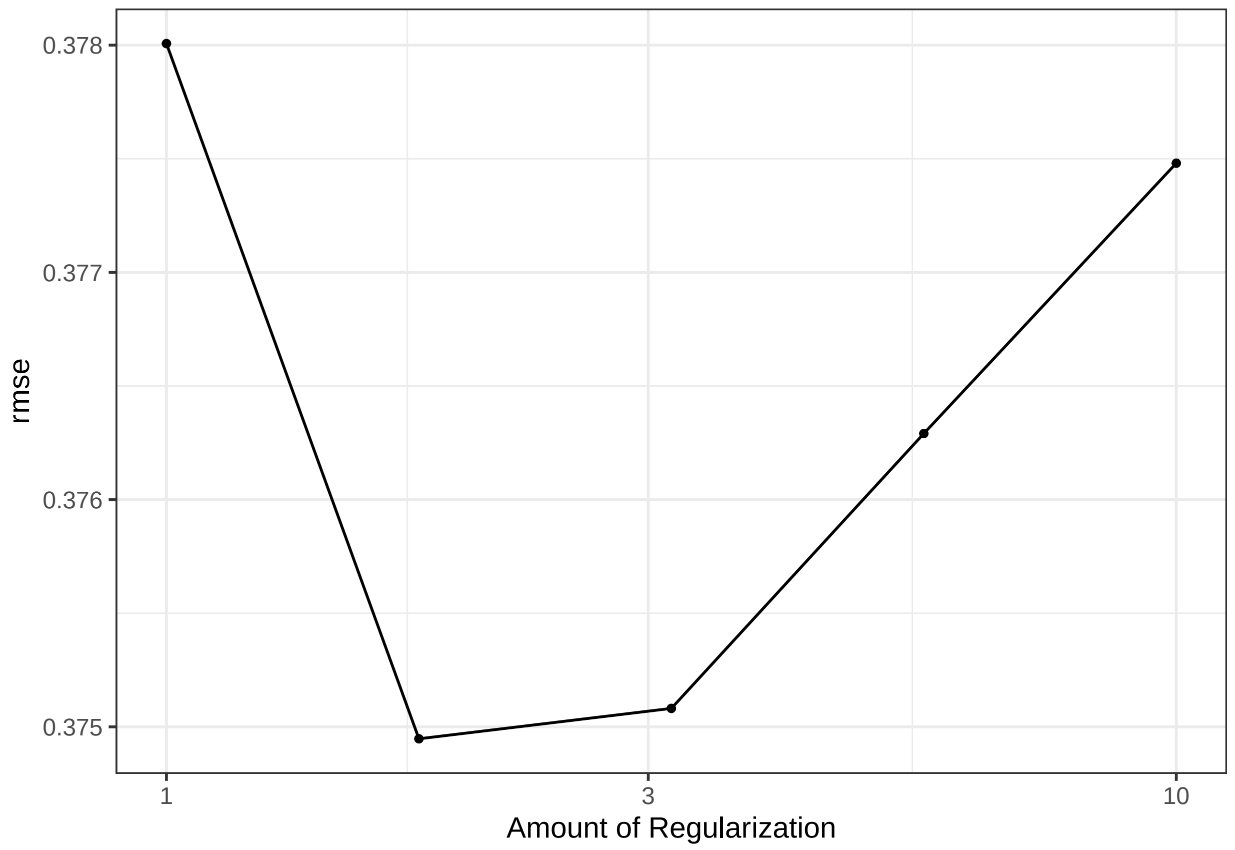
Model Autoplots

We will now apply autoplot to our models to visualize the effects of each tuned parameter on the performance of the models. It is measured by RMSE, so the lower the RMSE the better the performance of the model.

Since we did not tune the linear regression, we cannot apply autoplot.

Ridge Regression Plot

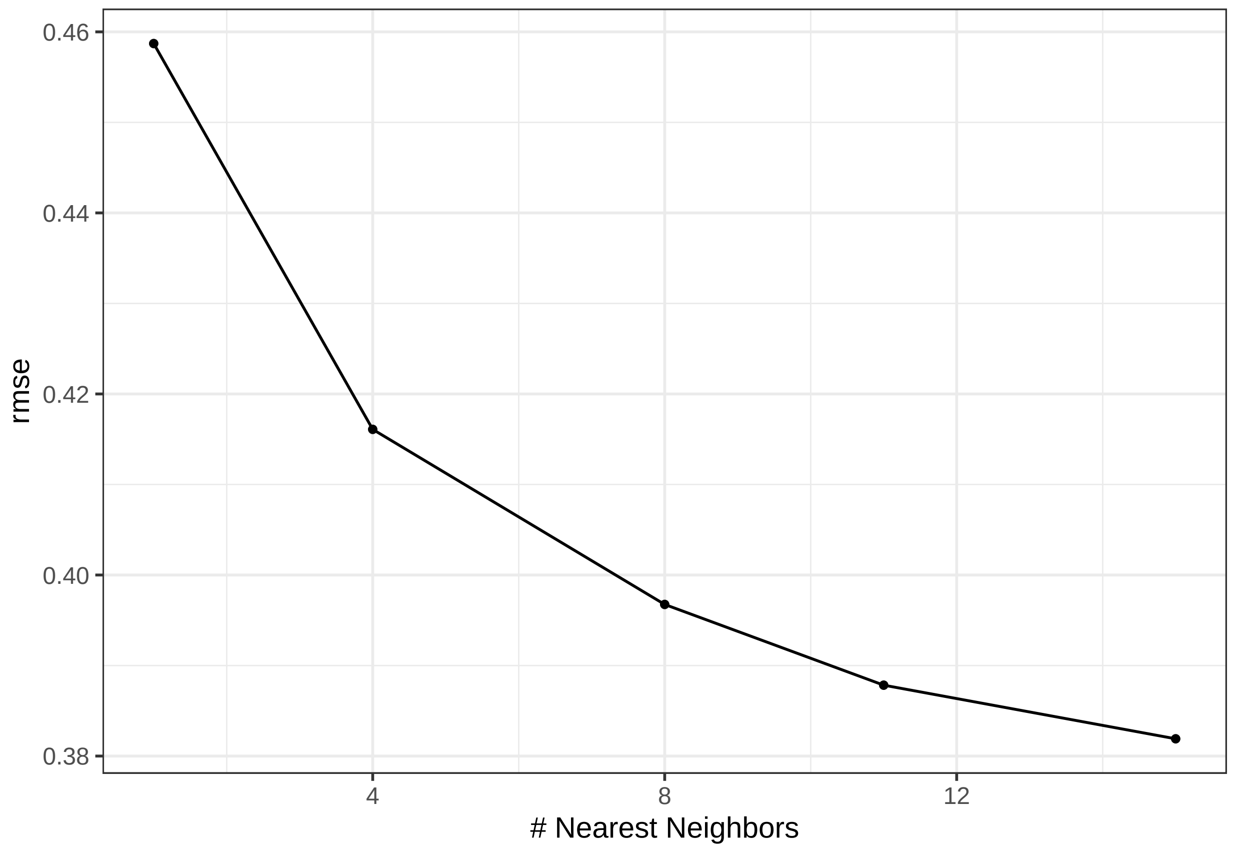
```
autoplot(ridge_tuned, metric = 'rmse')
```



In our ridge regression model, we tuned the penalty at 5 different levels. We can see that at the level 2 of amount of regularization, the model performed the best. However, it seems that when the amount of regularization was at 1 or 10, the model performed the worst.

K Nearest Neighbors Plot

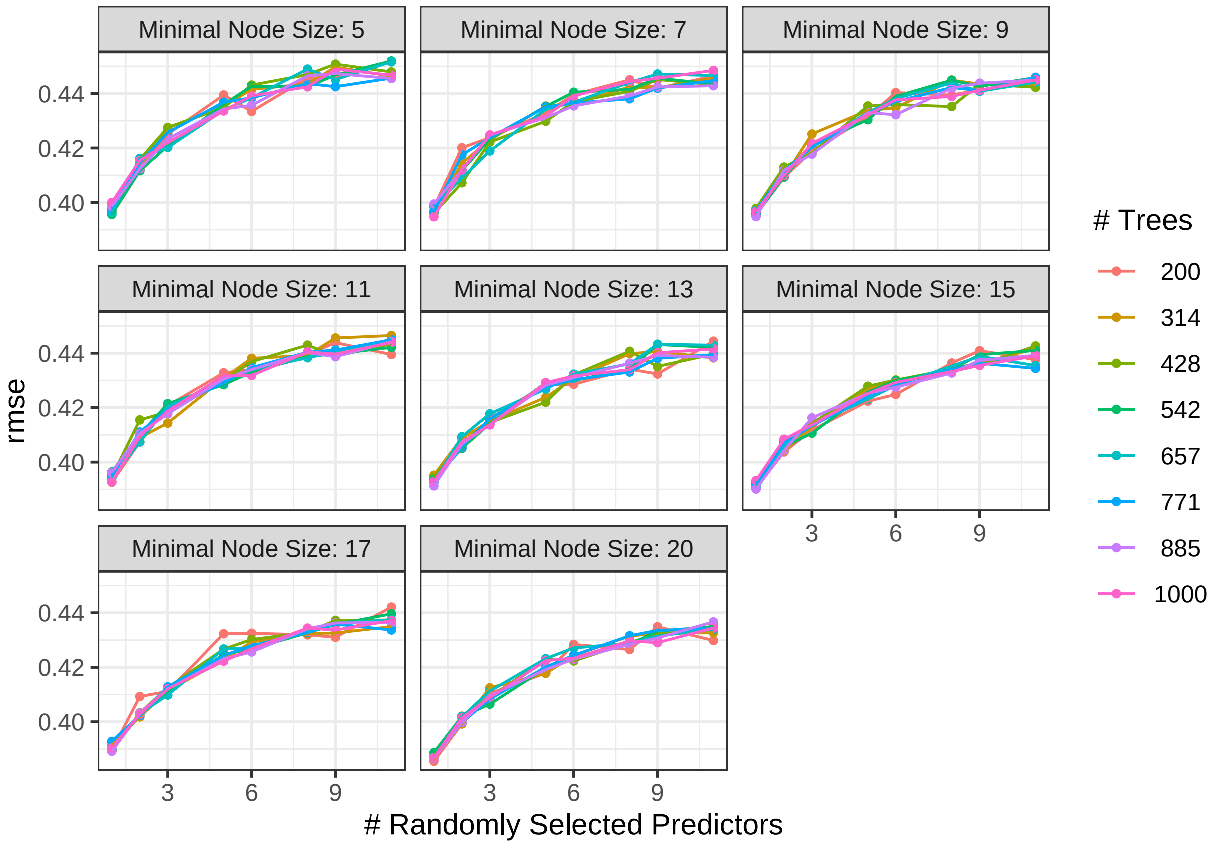
```
autoplot(knn_tuned, metric = 'rmse')
```



For the KNN model, we tuned 15 nearest neighbors at 5 different levels. From the graph, we can see that as we add a nearest neighbor, the model performs much better. There is an exponential decrease of the RMSE value as the number of nearest neighbors increases. This is expected because in most cases, adding more neighbors improves the model performance.

Random Forest Plot

```
autoplot(rf_tuned, metric = 'rmse')
```



For the random forest, we tuned the minimal node size, the number of randomly selected predictors, and the number of trees. We used all of our predictors in building this model, so there are 11 per minimal node. The graphs are a little unexpected since the RMSE values goes up as more predictors are added. This could mean that some predictors are almost detrimental in predicting our outcome. Looking at the correlation plot from before, we see that the predictors did not have a particularly high correlation with the outcome. Thus, this result could make sense. The number of trees stay consistent, so we can assume that it does not have much effect on the performance of the model. Overall, we can see that as the minimal node size increases, the RMSE value decreases meaning the model performed better. Specifically, the model performed the best when the minimal node size was 20 and when there were only 1-3 predictors.

Results of the Best Model

Performance on the Folds

The random forest model performed the best out of the 3 other models. Let's find out which tuned parameters were chosen as the best random forest model.

```
# Collecting metrics on the tuned random forest and slicing the lowest RMSE
best_tuned_rf <- rf_tuned %>%
  collect_metrics() %>%
  arrange(mean)

best_tuned_rf[1,]
```

```
## # A tibble: 1 x 9
##    mtry trees min_n .metric .estimator  mean     n std_err .config
##   <int> <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1     1   657     7 rsq     standard   0.108    10  0.0336 Preprocessor1_Model0~
```

The random forest with 657 trees, a minimal mode size of 7, and 10 predictors performed the best! It had the lowest RMSE of 0.1077443.

Fitting to Training Data

Now that we have the best model from the tuned random forest, we will fit it to our training data. Then, the random forest will be trained on the whole training data set and it will be ready for testing!

```
# Fitting to the training data
best_train_rf <- select_best(rf_tuned, metric = 'rmse')
rf_final_workflow_train <- finalize_workflow(rf_workflow, best_train_rf)
final_fit_train_rf <- fit(rf_final_workflow_train, data = song_train)

# Save
write_rds(final_fit_train_rf, file = "~/Documents/Pstat 131/Final project/final_rf/final_train_rf.rds")
```

Testing the Model

Let's test the performance of our random forest on the testing data set. Note it has not been trained at all!

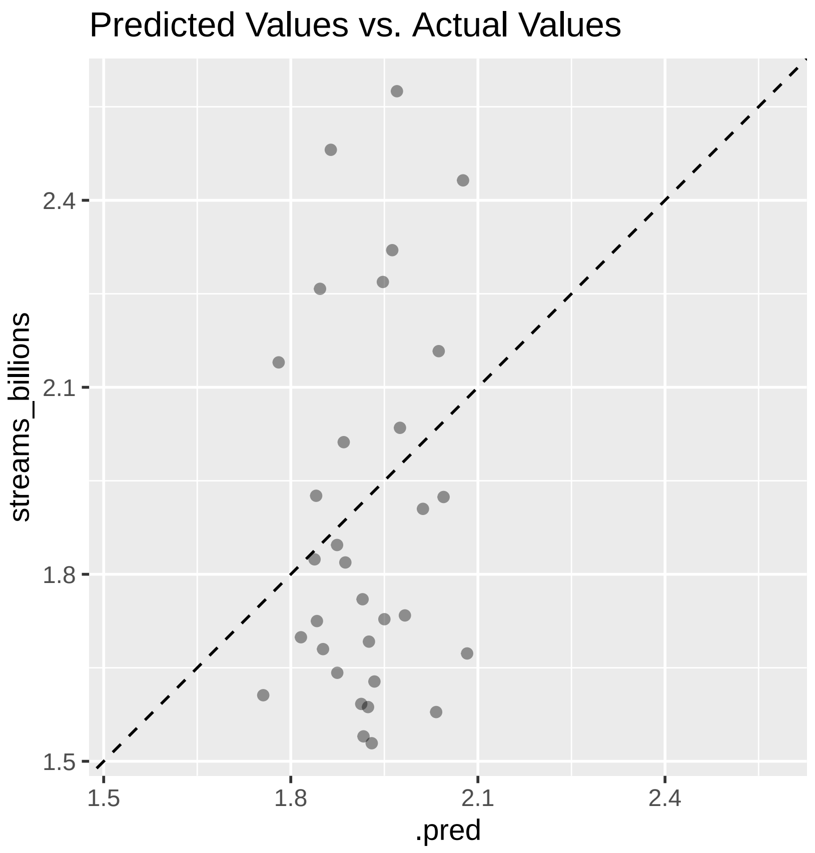
```
# Loading in the training data fit
final_fit_train_rf <- read_rds(file = "~/Documents/Pstat 131/Final project/final_rf/final_train_rf.rds"

# Creating the predicted vs. actual value tibble
song_tibble <- predict(final_fit_train_rf, new_data = song_test %>% select(-streams_billions))
song_tibble <- bind_cols(song_tibble, song_test %>% select(streams_billions))

# Save the file
write_rds(song_tibble, file = "~/Documents/Pstat 131/Final project/final_rf/final_model.rds")

# Load in final model
song_tibble <- read_rds(file = "~/Documents/Pstat 131/Final project/final_rf/final_model.rds")

# RMSE as the metric
song_metric <- metric_set(rmse)

# Collecting the RMSE of the model
song_metric_tibble <- song_metric(song_tibble, truth = streams_billions, estimate = .pred)
song_metric_tibble
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard       0.295
```

Our random forest model performed better on the training data set with a RMSE of 0.10362 versus a RMSE of 0.295 with the testing data set. Both values are very low, meaning our model performed very well overall!

Predicted values versus actual values

```
# Creating plot of predicted values vs. actual values
song_tibble %>%
  ggplot(aes(x = .pred, y = streams_billions)) +
  geom_point(alpha = 0.4) +
  geom_abline(lty = 2) +
  theme_grey() +
  coord_obs_pred() +
  labs(title = "Predicted Values vs. Actual Values")
```

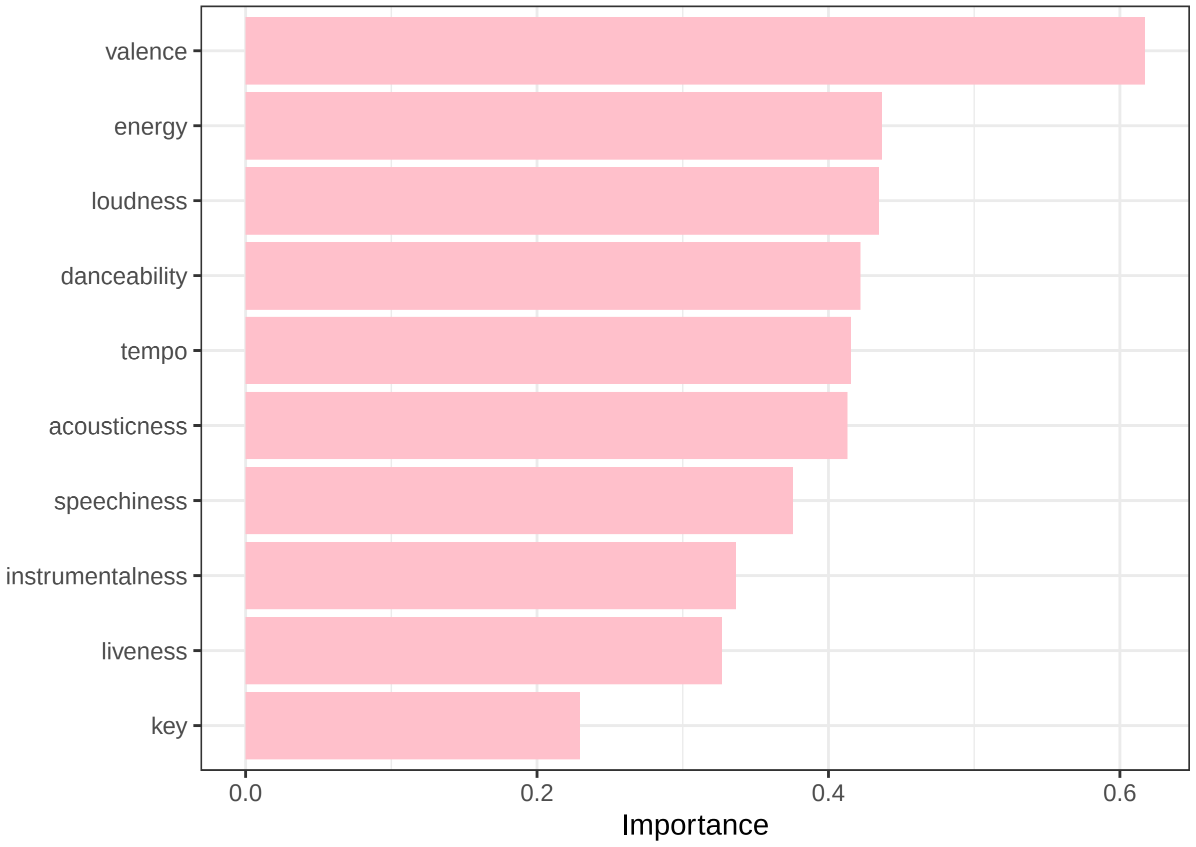


It looks like the predicted values are bunched within the 1.8-2.1 interval. Thus, around half the data was underestimated and the other half was overestimated. There are a couple dots that fall within a good range of the fit line, but for the most part the values were not predicted as accurately. However, after close inspection, the predicted values were not that far off from the actual values. Although they may not be exact, they fall within about a 0.5 radius from the actual values. Overall, the random forest performed well with a few areas that could be worked on for improvement.

Variable Importance

Using the variable importance plot (VIP), let's see which variables had the most impact in predicting the outcome!

```
# Loading in the training data fit
final_fit_train_rf <- read_rds(file = "~/Documents/Pstat 131/Final project/final_rf/final_train_rf.rds"

# Using the training fit to create the VIP because the model was not actually fit to the testing data
final_fit_train_rf %>%
  extract_fit_engine() %>%
  vip(aesthetics = list(fill = "pink"))
```



From our graph, we can see that the variable that had the most impact on the outcome is valence. This is surprising considering that in our previous correlation plot, streams and valence had a correlation of -0.06. Valence describes the positivity tied to a song. The higher the valence, the happier and more cheerful. The lower the valence, the more negative the song sounds. This does make sense considering that most people keep songs with a higher valence in their daily playlists. There is a time and place for any kind of genre, but for the majority of the time it is expected that people gear towards happier songs. Acousticness, energy, and danceability are very close after valence. We did see in our visual EDA that these variables either had a high or low correlation with the number of streams, so this result makes sense.

Conclusion

We fitted four different models to our training data, and concluded that the random forest performed the best. With the lowest RMSE value, it was able to predict the number of streams a song receives on Spotify better than all other models. This is expected because from previous assignments and labs, we could see that random forests work well with a majority of data sets because its nonparametric. Additionally, it does not assume anything about the outcome. The low RMSE value leads me to believe it did a good job in predicting the stream count.

Our worst model was the ridge regression model. This is not a surprise because ridge regression models usually work well with data sets that deal with multicollinearity. As we saw in our correlation plot, our variables did not have an overall high correlation with one another. Thus, it makes sense that this model performed the worst.

In the future, I would want to expand my data set so that I have more data to work with. Having only 100 rows of data posed as a small issue when creating folds. Also, with more data the data would be more accurate. The sample group of songs was not representative of songs in general, since it was the top 100. A lot of popular songs go viral from social media platforms such as Tik Tok and Instagram. Since we were just looking at features of the music itself, we did not take into consideration outside influences that could increase the number of streams. This could explain why the increase in predictors led to a decrease in model performance. Perhaps if there was data collected on whether the song was made a sound on Tik Tok, we could incorporate that into our analysis for better results.

I originally took out the song name and release data variables, assuming it would not have as much of an effect on the outcome. However, the song name could be useful for songs that are less popular. For example, if someone is browsing through a Spotify-generated playlist, the title is the first thing they see. It could result in more clicks and therefore streams. As for the release date, the timing could be a potential predictor variable. If two songs from two different big artists were released at the same time, one could overshadow the other. There are many possibilities that were not considered in this project, so I would like to explore those more in the future.

Overall, this data analysis allowed me to further understand model building. I was glad to work with data I had a genuine interest for, since it is something I use every day. I never considered factors such as valence or danceability in determining the number of streams a song gets. After working with the variables, models, and graphs, I have a much better understanding of the statistics behind why the top 100 songs got to where they are. Despite outside factors that could influence the stream count, I was able to develop a model using the variables I had! Thank you :D

Sources

This data was taken from the Kaggle data set, “Most Streamed Songs (All Time)” uploaded by user Amaan Ansari.

Images taken from google images.