Spotify Popularity Prediction Project

HarvardX: PH125.9x Data Science - Choose Your Own Project

Jeffrey Chin

1/1/2021

Contents

1 Executive Summary **2**

1.1 Background . . . 2

1.2 Project Goal . . . 2

1.3 About the Data . . . 2

1.4 Loading the Data . . . 3

2 Data Exploration **5**

2.1 Pre-processing . . . 5

2.2 Structure . . . 5

2.3 Data Visualizations . . . 5

3 Methods and Analysis **12**

3.1 Evaluation . . . 12

3.2 Model 1: Just the Average . . . 13

3.3 Model 2: Energy Effect Model . . . 14

3.4 Model 3: Energy and Year Effect Model . . . 15

3.5 Model 4: Regularized Energy and Year Effect Model . . . 17

3.6 Model 5: Random Forest Model Using Energy and Year as Predictors . . . 21

4 Results **22**

5 Conclusion **23**

5.1 Summary . . . 23

5.2 Limitations . . . 23

5.3 Future Work and Impact . . . 23

6 References **24**

1 Executive Summary

1.1 Background

Not long before online music-streaming services prevailed the music industry, people listened to music on the radio, on mp3 players, or at live concerts or events. One of the problems with earlier forms of music entertainment was that they did not allow for widespread music discovery. Radio channels often only featured the most popular songs from well-known and established artists, and people often only downloaded mp3 songs from artists that they already knew because portable music was not a free commodity at the time. To solve these issues, people have turned to using music streaming services such as Pandora and Spotify to help them discover new music. Spotify is a digital music streaming service featuring millions of tracks, podcasts, and videos from artists all over the world. However, unlike many other music streaming services, Spotify features a robust recommendation system that curates personalized playlists for users based on users' listening habits (what users like, share, save, and skip) as well as listening habits of others with similar music tastes. For example, users who have used the service for a few weeks are able to receive a "Discover Weekly" playlist, which is a personalized playlist generated using Spotify's recommendation algorithm and that updates every Monday, featuring 30 new tracks every week. Spotify also curates "Daily Mix" playlists for users, which are sets of up to 6 different mixes generated based on users' favorite music genres. Spotify's recommendation system not only helps users discover new music on a weekly or even daily basis, but also helps new and rising artists gain the recognition they deserve by showcasing their work to a large audience. Before the rise of machine-learning algorithms for personalizing and improving upon the user experience, relatively unknown artists were limited in the number of ways to go about promoting their work online, especially to individuals outside of their fanbase. Streaming services like Spotify aim to increase exposure for relatively unknown artists by giving every artist an equal chance at being heard, regardless of the size of his or her fanbase. Without the invention of algorithmic-based music streaming services, the digital music scene today would likely be very different as many popular streaming artists would not have nearly as much exposure to new listeners as they have today.

1.2 Project Goal

The goal of this project is to build an algorithm that can predict the popularity of a Spotify track given some of its audio and track features. The dataset that will be used to achieve this goal is the Spotify 1921 – 2020 dataset on Kaggle containing the audio and track features of over 160,000 tracks released from 1921 to 2020. According to Spotify, the popularity of a track is a measurement ranging from 0 to 100 calculated based on the total number of plays a track has, and how recent those plays are. In order to determine a final algorithm that can accurately predict popularity rankings, five models will be evaluated on accuracy using the root mean squared error (RMSE), and each algorithm will follow the previous model in complexity and predictive accuracy. To determine the features that will be used as predictors, the data will be visualized to extract useful relationships and biases among predictive variables.

1.3 About the Data

The Spotify 1921 – 2020 dataset contains a little over 170,000 tracks released from 1921 to 2020. The dataset covers a wide range of music genres, ranging from classical and jazz to heavy metal and techno. Each track has 19 features associated with it, including track features such as title, artist, and release year, and audio features such as acousticness, energy, and loudness. Several audio features used in the data are a combination of audio features that contribute to the overall measurement. For example, Spotify defines "energy" as a measurement from 0.0 to 1.0 that is based on perceptual features such as dynamic range, timbre, onset rate, and general entropy. "Danceability" is defined as how suitable a track is for dancing based on musical elements such as tempo, rhythm stability, and beat strength. Besides audio and track features, the dataset does not contain user or playlist information, which would make it possible to create a recommendation system based on users' listening behaviors and preferences.

1.4 Loading the Data

To start, we will first load the data. This code loads all the required packages, downloads the Spotify dataset, and then creates a partition of the data into a train and test set. The test set will be 10% of the original data and will only be used to test the accuracy of each model. The reason why a 90-10 train-test split ratio was chosen is because doing so allows enough data to be trained for generating accurate predictions while also leaving plenty of data in the test set to evaluate our models' tendencies to overfit the train data.

```
if(!require(tidyverse)) install.packages("tidyverse", repos = "http://cran.us.r-project.org")
```

```
## Loading required package: tidyverse
```

```
## -- Attaching packages ----------------------- tidyverse 1.3.0 --
```

```
## v ggplot2 3.3.2     v purrr   0.3.4
## v tibble  3.0.3     v dplyr   1.0.0
## v tidyr   1.1.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.5.0
```

```
## -- Conflicts -------------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
if(!require(caret)) install.packages("caret", repos = "http://cran.us.r-project.org")
```

```
## Loading required package: caret
```

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

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

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

```
if(!require(data.table)) install.packages("data.table", repos = "http://cran.us.r-project.org")
```

```
## Loading required package: data.table
```

```
##
## Attaching package: 'data.table'
```

```
## The following objects are masked from 'package:dplyr':
##
##     between, first, last
```

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

```
if(!require(ggcorrplot)) install.packages("ggcorrplot", repos = "http://cran.us.r-project.org")
```

```
## Loading required package: ggcorrplot
```

```
## Warning: package 'ggcorrplot' was built under R version 4.0.3
```

```
library(tidyverse)
library(caret)
library(data.table)
library(ggcorrplot)

dl <- tempfile()
download.file("https://github.com/jeffreyrchin/Spotify-Dataset/raw/master/archive.zip", dl)

Spotify <- read.csv(unzip(dl, "data.csv"))

# Test set will be 10% of Spotify data
set.seed(1, sample.kind="Rounding") # if using R 3.5 or earlier, use `set.seed(1)`
```

```
## Warning in set.seed(1, sample.kind = "Rounding"): non-uniform 'Rounding' sampler
## used
```

```
test_index <- createDataPartition(y = Spotify$popularity, times = 1, p = 0.1, list = FALSE)
train <- Spotify[-test_index,]
temp <- Spotify[test_index,]

# Make sure energy and year entries in test set are also in the train set
test <- temp %>%
  semi_join(train, by = "energy") %>%
  semi_join(train, by = "year")

# Add rows removed from test set back into train set
removed <- anti_join(temp, test)
```

```
## Joining, by = c("valence", "year", "acousticness", "artists", "danceability", "duration_ms", "energy
```

```
train <- rbind(train, removed)

rm(dl, temp, test_index, removed)
```

2 Data Exploration

2.1 Pre-processing

Because the name and id columns are unique to each track and do not provide much predictive assistance, we will be removing those columns from our data. Additionally, the release_date column will be removed because there is already a column named "year" that provides the release year of a track. Although the "artists" column may provide some assistance for predicting popularity rankings, we will not be including the artist effect in any of our models for the purposes of this project.

```
Spotify <- Spotify %>% select(-c(artists, id, name, release_date))
```

2.2 Structure

```
dim(Spotify)
```

```
## [1] 170653     15
```

As shown above, the Spotify dataset contains a little over 170,000 observations for 15 variables. Details about each variable are shown below:

```
str(Spotify)
```

```
## 'data.frame':    170653 obs. of  15 variables:
##  $ valence         : num  0.0594 0.963 0.0394 0.165 0.253 0.196 0.406 0.0731 0.721 0.771 ...
##  $ year            : int  1921 1921 1921 1921 1921 1921 1921 1921 1921 1921 ...
##  $ acousticness    : num  0.982 0.732 0.961 0.967 0.957 0.579 0.996 0.993 0.996 0.982 ...
##  $ danceability    : num  0.279 0.819 0.328 0.275 0.418 0.697 0.518 0.389 0.485 0.684 ...
##  $ duration_ms     : int  831667 180533 500062 210000 166693 395076 159507 218773 161520 196560 ...
##  $ energy          : num  0.211 0.341 0.166 0.309 0.193 0.346 0.203 0.088 0.13 0.257 ...
##  $ explicit        : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ instrumentalness: num  8.78e-01 0.00 9.13e-01 2.77e-05 1.68e-06 1.68e-01 0.00 5.27e-01 1.51e-01 0
##  $ key             : int  10 7 3 5 3 2 0 1 5 8 ...
##  $ liveness        : num  0.665 0.16 0.101 0.381 0.229 0.13 0.115 0.363 0.104 0.504 ...
##  $ loudness        : num  -20.1 -12.44 -14.85 -9.32 -10.1 ...
##  $ mode            : int  1 1 1 1 1 1 1 0 0 1 ...
##  $ popularity      : int  4 5 5 3 2 6 4 2 0 0 ...
##  $ speechiness     : num  0.0366 0.415 0.0339 0.0354 0.038 0.07 0.0615 0.0456 0.0483 0.399 ...
##  $ tempo           : num  81 60.9 110.3 100.1 101.7 ...
```

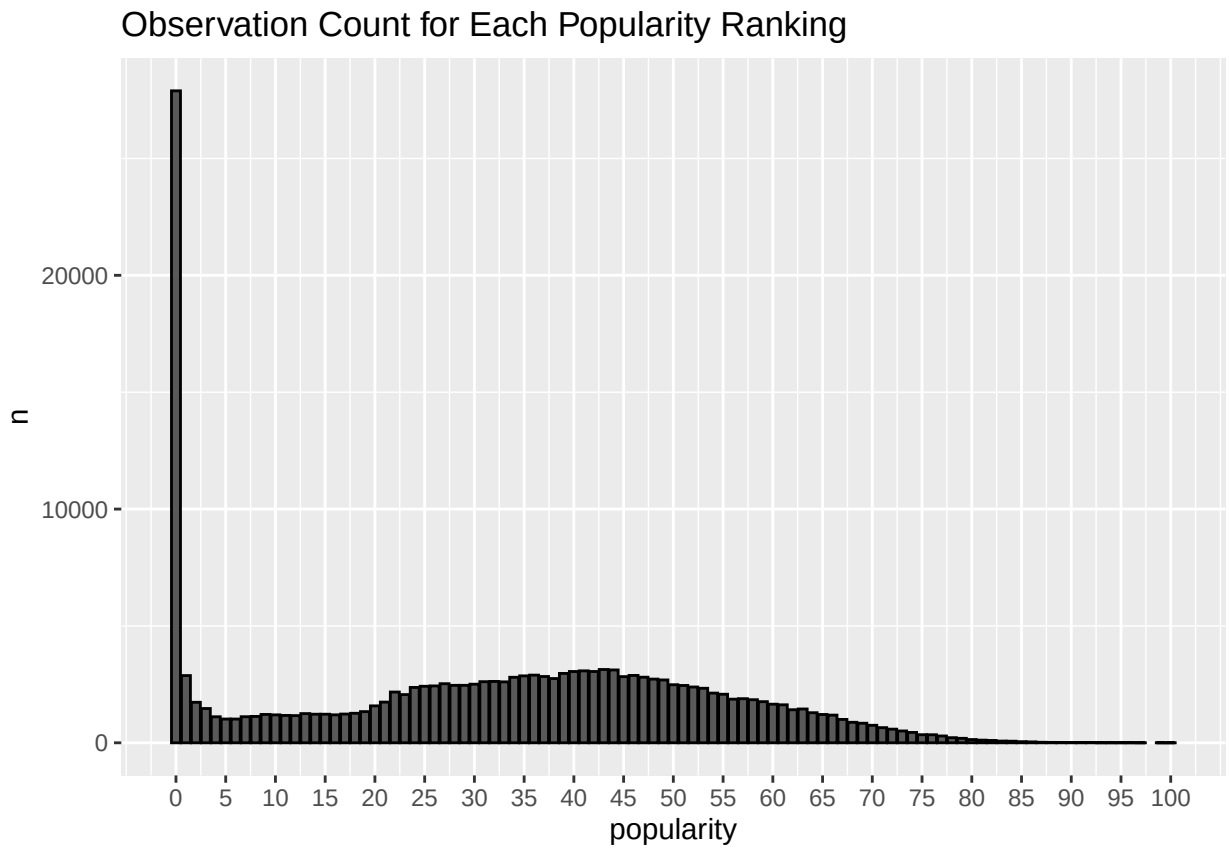
From this table, it can be observed that there are 15 variables: valence, year, acousticness, danceability, duration_ms, energy, explicit, instrumentalness, key, liveness, loudness, mode, popularity, speechiness, and tempo. For this project, we will mainly be using popularity, energy, and the release year.

2.3 Data Visualizations

Before applying algorithmic models to our data, we will first need to analyze it. The code below visualizes the number of observations for each individual popularity ranking from 0 to 100.

```r
# Number of Observations for Each Popularity Ranking:

Spotify %>% count(popularity) %>%
  ggplot(aes(popularity, n)) +
  geom_col(color = "black") +
  scale_x_continuous(breaks = seq(0, 100, 5)) +
  ggtitle("Observation Count for Each Popularity Ranking")
```



From this graph, it can be observed that most tracks have a popularity of zero and that there are very few tracks with a popularity of above 80.

```r
# Number of observations for high popularity rankings (above 80):

Spotify %>% count(popularity) %>% tail(20)
```

```
##     popularity   n
## 81          80 139
## 82          81 114
## 83          82 100
## 84          83  74
## 85          84  69
## 86          85  48
## 87          86  39
## 88          87  23
## 89          88  17
```

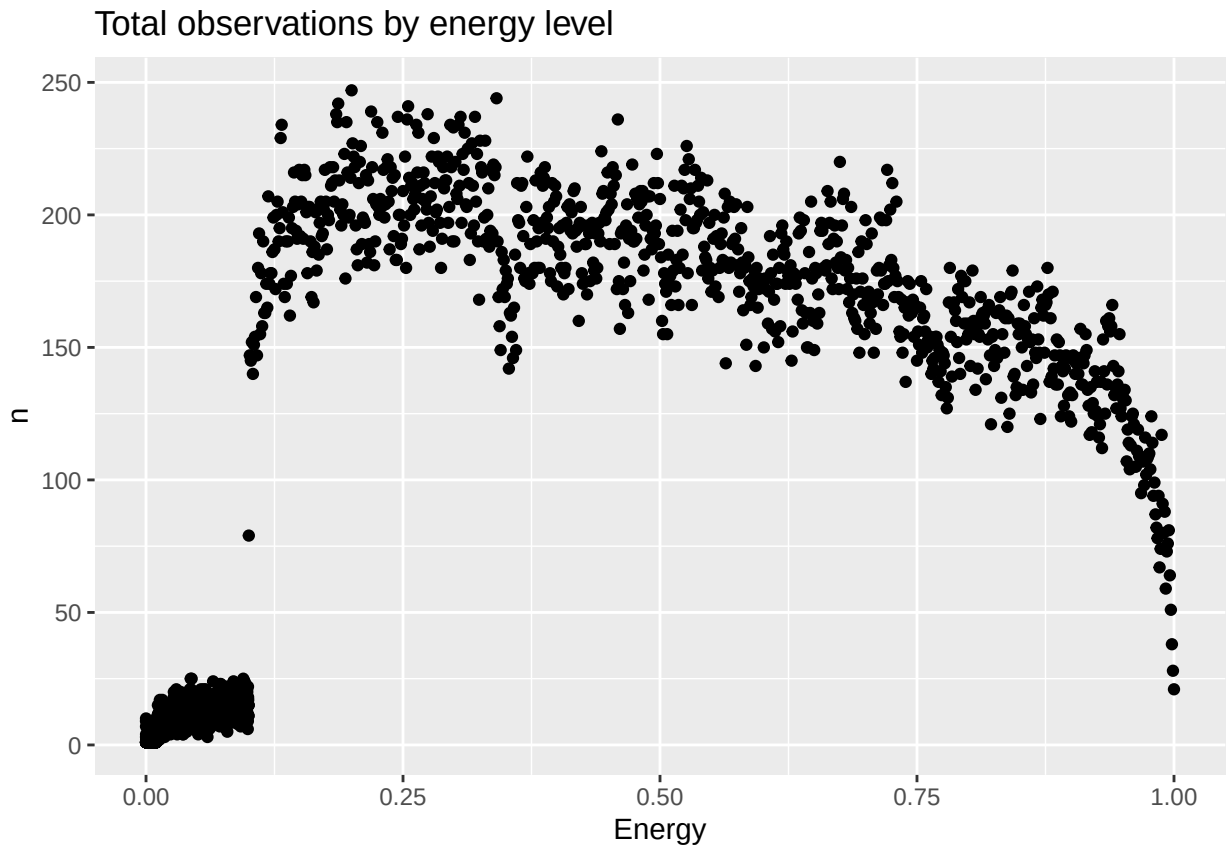
```
## 90           89  15
## 91           90  11
## 92           91   9
## 93           92  11
## 94           93   4
## 95           94   4
## 96           95   4
## 97           96   4
## 98           97   1
## 99           99   1
## 100         100   1
```

We can see that the total number of observations having a popularity of greater than or equal to 80 is around 1% of the original data. This makes sense because tracks associated with artists in the top 1% must not only have a high stream count, but must also have a high number of recent streams relative to other popular tracks.

```
# Popularity rankings grouped by energy level, totaled:

Spotify %>% group_by(energy) %>%
  summarize(n = n()) %>%
  ggplot(aes(energy, n)) +
  xlab("Energy") +
  geom_point() +
  ggtitle("Total observations by energy level")
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

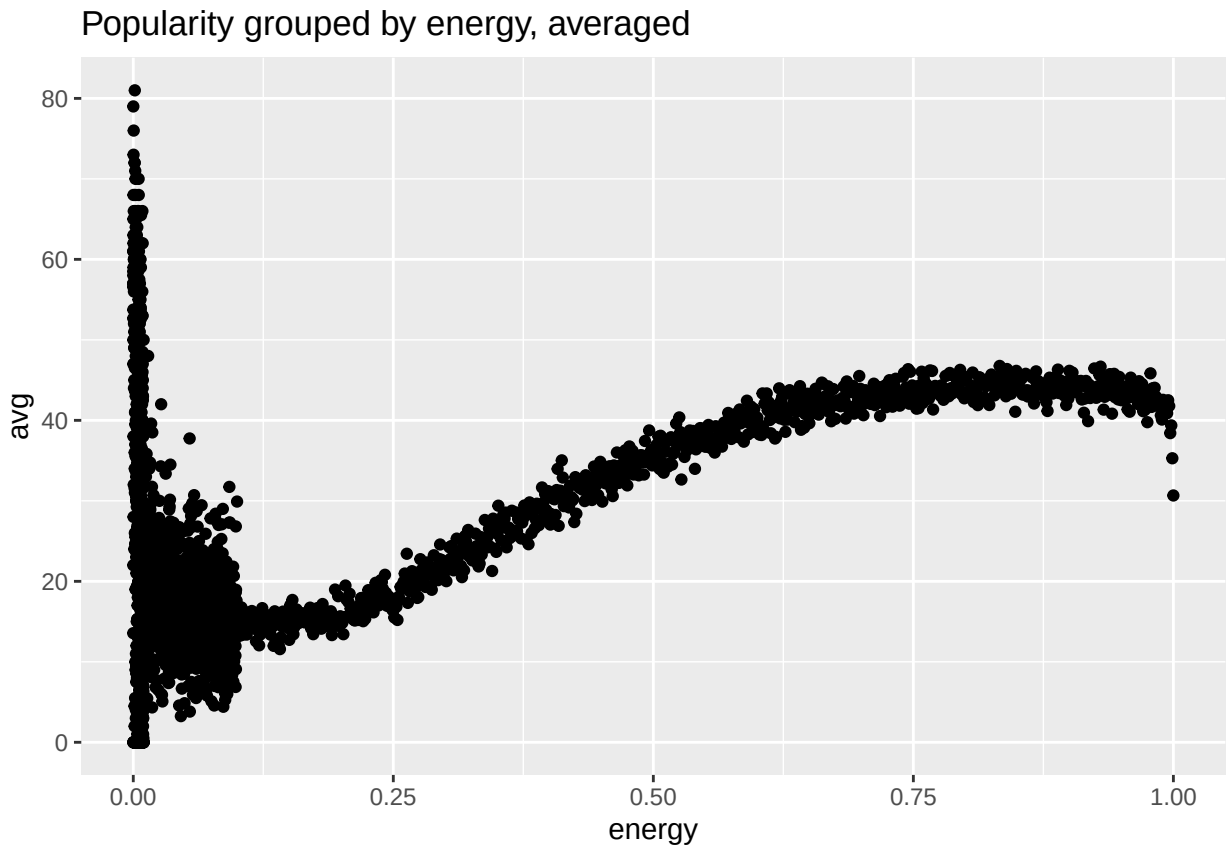


The graph above shows that most tracks have a low energy level and that there is a downward trend for which the number of tracks decreases for each subsequent energy level. This could be explained by the fact that the energy measurement for a track is defined by its dynamic range, which seems to indicate that there is a high number of tracks within the data that have a limited dynamic range. A limited dynamic range can be indicative of a limited audio equipment budget among other factors, which may possibly explain the higher number of tracks in the data with low energy levels.

```
# Popularity rankings grouped by energy level, averaged:

Spotify %>% group_by(energy) %>%
  summarize(avg = mean(popularity)) %>%
  ggplot(aes(energy, avg)) +
  xlab("energy") +
  geom_point() +
  ggtitle("Popularity grouped by energy, averaged")
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

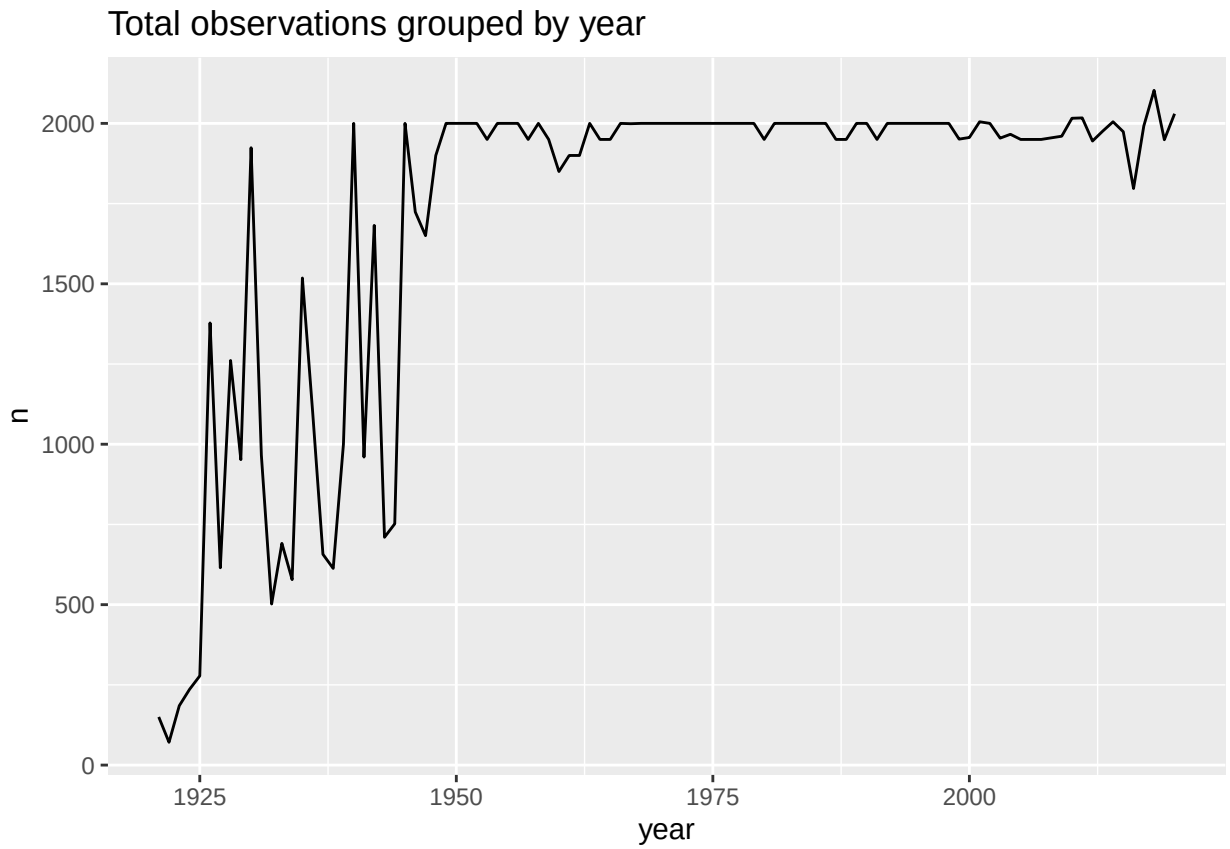


In this graph, we see that there is a notable rise in average popularity rankings for tracks with higher energy levels. The large cluster of popularity averages around energy level 0 can be explained by small sample sizes observed in the previous graph for energy levels close to 0. Small sample sizes tend to generate high variability among averages, whereas larger sample sizes generate more stability among averages. The correlation between increased observation count and increased average stability can be explained by the Law of Large Numbers, which states that as the sample size of a random distribution increases, the mean of the distribution approaches the mean of the population. In the context of our analysis, as the number of observations for tracks of energy j increases, the average popularity for tracks of that energy level converges to the average popularity of all tracks, μ.

```
# Popularity rankings grouped by year, totaled:

Spotify %>% group_by(year) %>%
  summarize(n = n()) %>%
  ggplot(aes(year, n)) +
  xlab("year") +
  geom_line() +
  ggtitle("Total observations grouped by year")
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

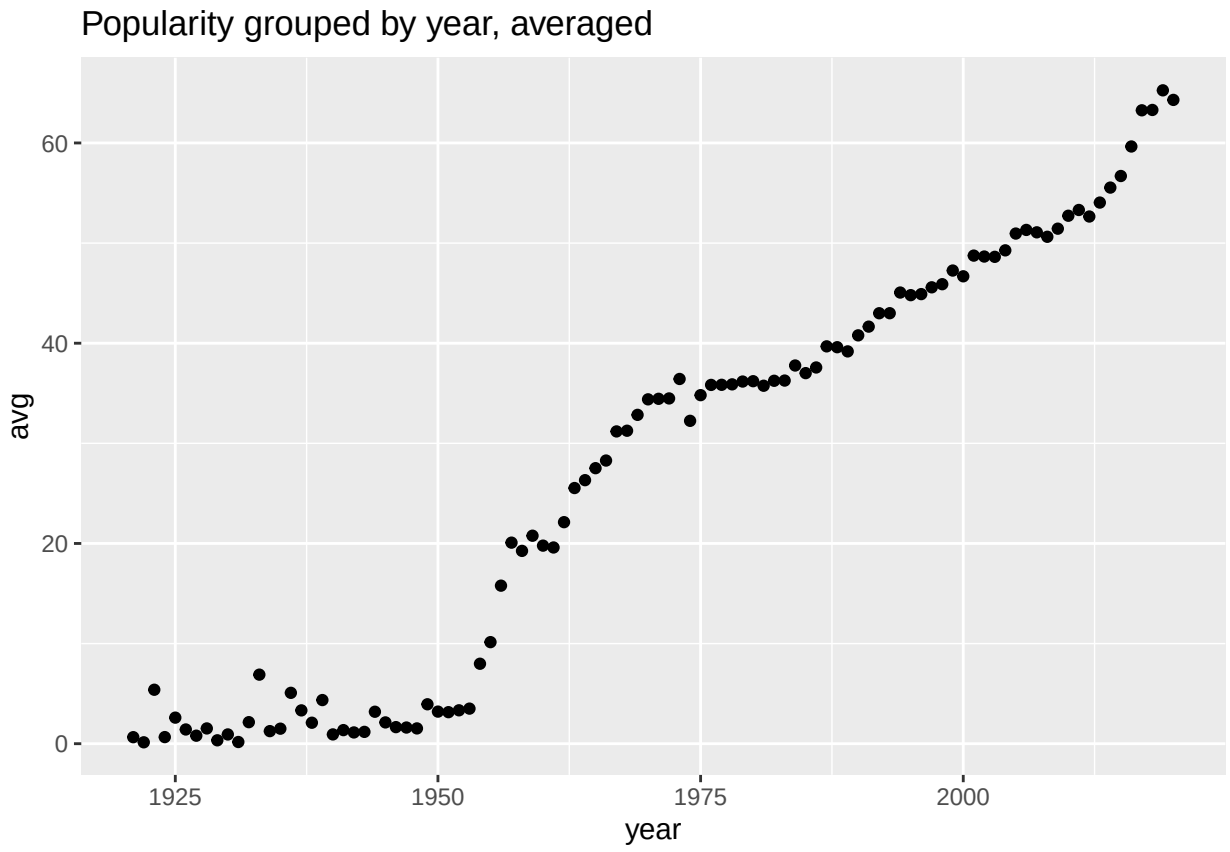


In this graph, we notice an interesting trend. The number of observations for track release years around and after 1950 caps at 2000. Because such a phenomenon is not likely a natural occurrence, it is assumed that observations for years after 1950 are the top 2000 most popular tracks for each year. This introduces a critical problem in our data, in that there is an unnatural popularity bias for tracks released around and after 1950.

```
# Popularity rankings grouped by year, averaged:

Spotify %>% group_by(year) %>%
  summarize(avg = mean(popularity)) %>%
  ggplot(aes(year, avg)) +
  xlab("year") +
  geom_point() +
  ggtitle("Popularity grouped by year, averaged")
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

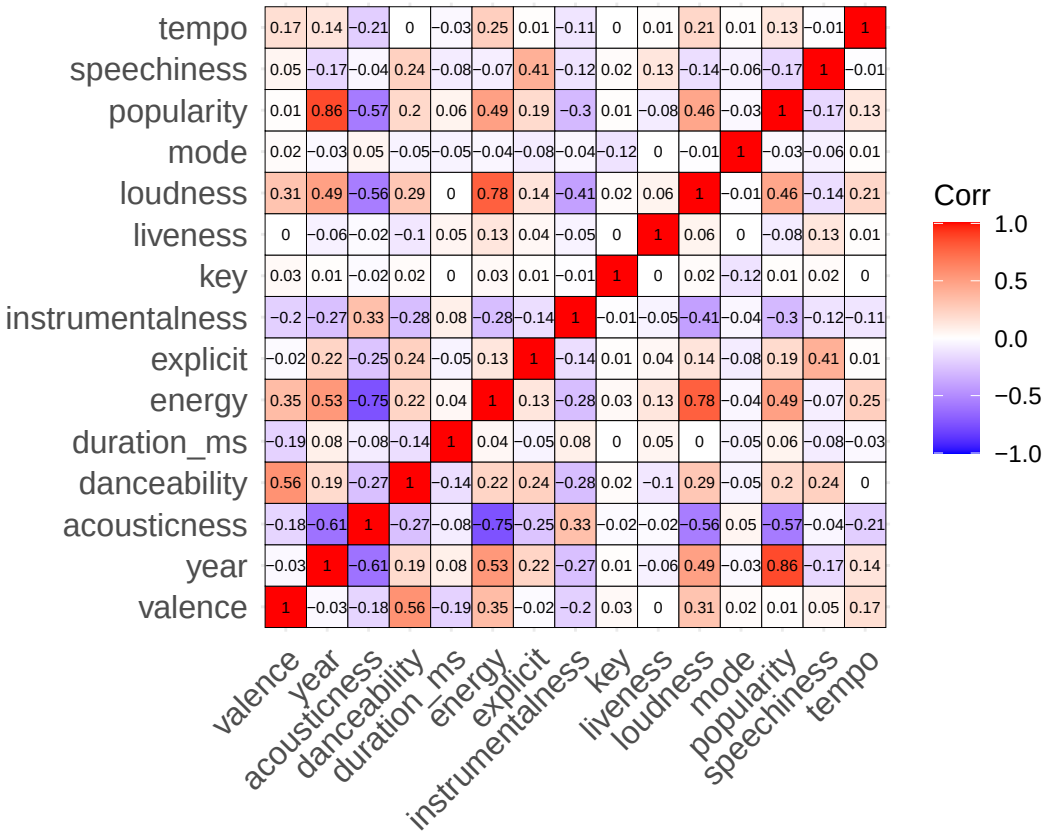


As expected from our previous analysis, tracks released on or after 1950 tend to have higher popularity rankings compared to tracks released before 1950.

In order to determine the features that will be used as predictors for our models, it is helpful for us to create a correlation table showing the strongest correlations between all predictors.

```
# Correlation table:

cor(Spotify) %>% ggcorrplot(lab = T, lab_size = 2, outline.color = "black")
```



From this correlation table, we notice that of all predictors, energy and year positively influence the popularity of a track the most. This conclusion confirms and aligns with our visualizations because we saw that the popularity of a track generally increases based on how recent its release date is and how energetic it is.

3 Methods and Analysis

3.1 Evaluation

After analyzing the data, we can see that there are some general trends and biases among the relationships between a track's popularity, energy, and release year. These biases will all be considered in our final predictive model. To begin applying machine-learning algorithms to our dataset, we must first understand how they will be evaluated. We will be evaluating our predictions with the root mean squared error (RMSE), or loss function. This function is used to determine the accuracy of a predictive model based on the root mean squared difference between a predicted popularity and the actual popularity, and can be formally notated as follows:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}$$

where y_i represents the actual popularity of track i, $\hat{y}_i$ represents the predicted popularity of track i, and N represents the number of observations in the dataset. The following code is used to calculate the RMSE in R:

```r
# Loss Function (RMSE):

RMSE <- function(true_ratings, predicted_ratings){
  sqrt(mean((true_ratings - predicted_ratings)^2))
}
```

3.2 Model 1: Just the Average

A simple recommendation algorithm would be to predict popularity using the average of all popularity rankings across all tracks. This algorithm can be mathematically represented by the following equation:

$$Y_i = \mu + \epsilon_i$$

where μ represents the average popularity across all tracks and ϵ_i represents track-to-track variation due to random error. Statistically, the average ranking is chosen due to the fact that the average is a least-squares estimate that minimizes the RMSE. This phenomenon can be illustrated by making predictions using any other number besides the average ranking. For example, random popularity predictions of 1 and 40 both result in RMSEs higher than that of our first predictive model.

```r
# Model 1: Just The Average

mu_hat <- mean(train$popularity) # Average of all popularity ratings of tracks in train

prediction_1 <- mu_hat # using just the average to predict popularity ratings in test
prediction_1
```

```
## [1] 31.43071
```

```r
model_1_rmse <- RMSE(test$popularity, prediction_1) # Calculate the RMSE
model_1_rmse
```

```
## [1] 21.80934
```

```r
# RMSE of using any other number besides the average popularity rating:

RMSE(test$popularity, 40)
```

```
## [1] 23.42849
```

```r
RMSE(test$popularity, 40) > model_1_rmse
```

```
## [1] TRUE
```

```r
RMSE(test$popularity, 1)
```

```
## [1] 37.44776
```

```
RMSE(test$popularity, 1) > model_1_rmse
```

```
## [1] TRUE
```

3.3 Model 2: Energy Effect Model

We saw in our previous model that using the average popularity of all tracks to predict rankings resulted in an RMSE of 21.8. Although an improvement to random prediction, we can definitely do better. In exploring and visualizing the relationship between track popularity and energy level, it was observed that popular tracks generally tend to have higher energy levels. This bias should not be ignored because incorporating it into our model has the potential to greatly improve our RMSE. This bias is called the "energy effect" and is represented by the term b_j. Including the energy effect into our model produces this equation:

$$Y_i = \mu + b_j + \epsilon_i$$

where b_j represents the average popularity ranking for tracks of energy j. Because some tracks have the same energy levels, we will need to group the data by energy, which we can do now because this model does not involve year-to-year popularity variation yet. Additionally, since we know that the estimated average popularity for each energy level j is the average of $Y_i - \hat{\mu}$, we can calculate $\hat{b}_j$ in the following way:

$$\hat{b}_j = \frac{1}{N}\sum_{j=1}^{N}(Y_i - \hat{\mu})$$

where N represents the number of distinct energy levels in the dataset.

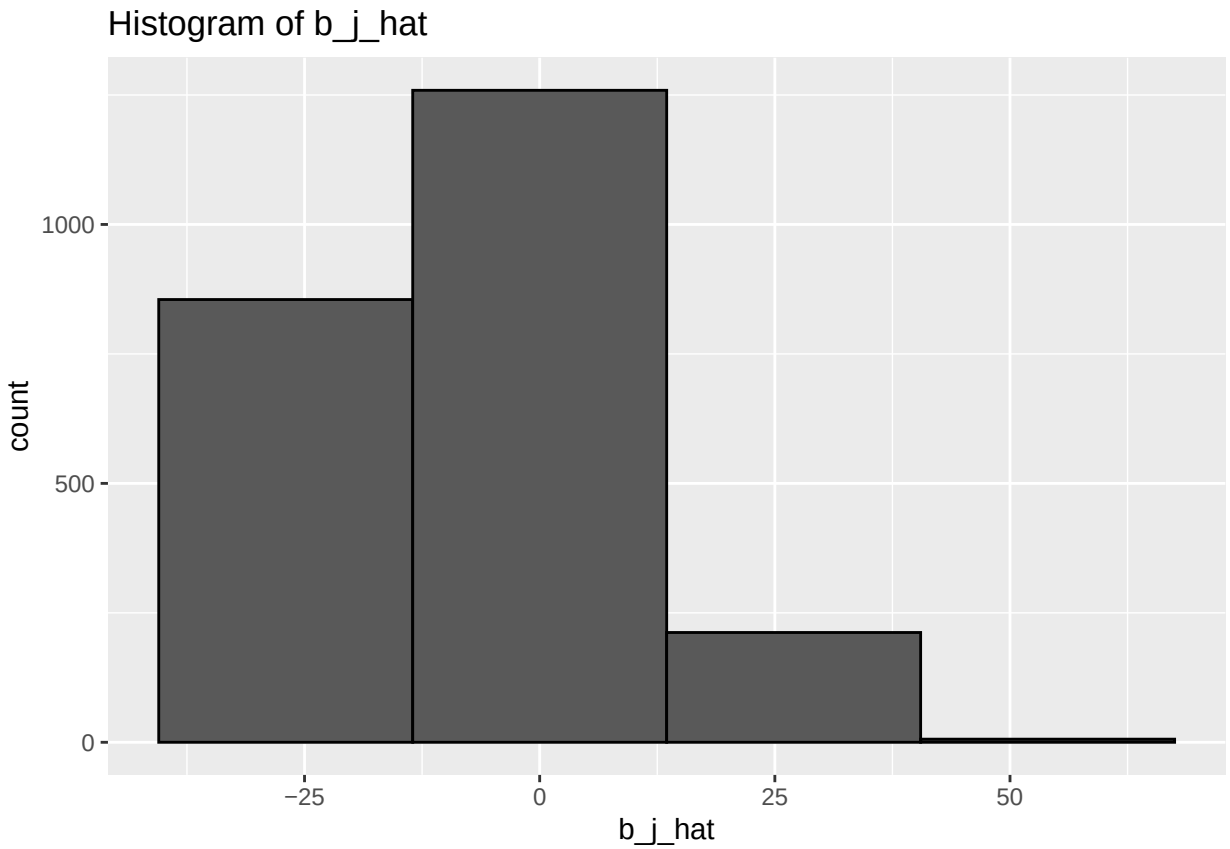
The following code calculates the average popularity ranking (centered around 0) of all distinct energy levels in the train set, subsets these averages for energy levels in the test set, combines $\hat{\mu}$ and $\hat{b}_j$ to make predictions in the test set, and calculates the RMSE of those predictions.

```
# Model 2: Energy Effect Model

b_j_hat <- train %>%
  group_by(energy) %>%
  summarize(b_j_hat = mean(popularity - mu_hat)) # b_j_hat represents the adjusted average popularity r
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
b_j_hat %>%
  ggplot(aes(b_j_hat)) +
  geom_histogram(bins = 4, color = "black") +
  ggtitle("Histogram of b_j_hat")
```



As expected, this graph shows that most energy levels have an average popularity ranking of around $\hat{\mu}$.

```
b_j_hat_test <- test %>%
  left_join(b_j_hat, by = "energy") %>%
  .$b_j_hat # b_j estimates for each energy level in test

prediction_2 <- mu_hat + b_j_hat_test # combining the average and the average popularity rating bias to

model_2_rmse <- RMSE(test$popularity, prediction_2) # Calculate the RMSE
model_2_rmse
```

```
## [1] 18.89756
```

Incorporating the energy effect into our first model improved our RMSE by 13%, a significant improvement. But can we do even better?

3.4 Model 3: Energy and Year Effect Model

In addition to the energy effect bias, another bias called the "year effect" can also be added to our model. This bias is represented by b_y, which will change our formula into:

$$Y_i = \mu + b_j + b_y + \epsilon_i$$

where b_y represents the average popularity ranking for year y.

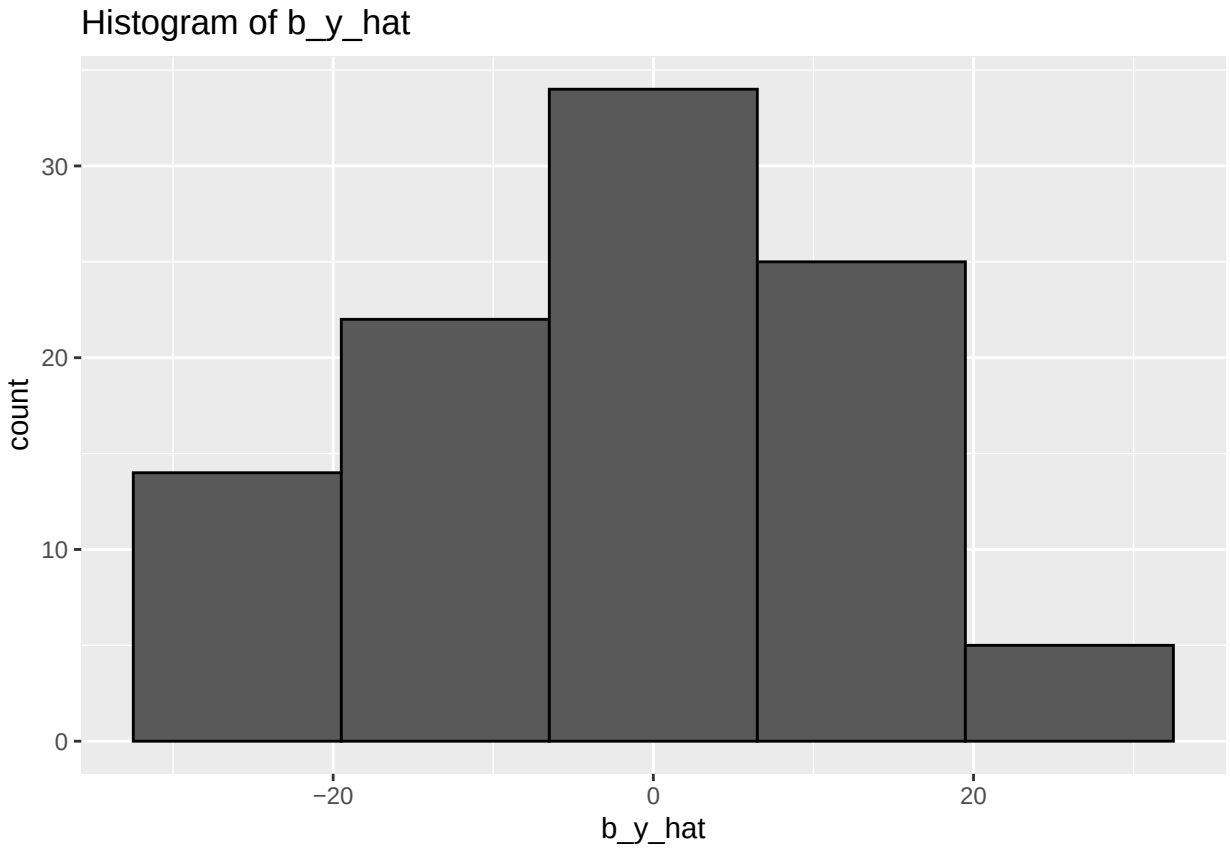
Similar to how we estimated $\hat{b}_j$, $\hat{b}_y$ can be estimated by calculating the average of $Y_i - \hat{\mu} - \hat{b}_j$:

```
# Model 3: Energy and Year Effect Model

b_y_hat <- train %>%
  left_join(b_j_hat, by = "energy") %>%
  group_by(year) %>%
  summarize(b_y_hat = mean(popularity - mu_hat - b_j_hat)) # b_y_hat represents the adjusted average po
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
b_y_hat %>%
  ggplot(aes(b_y_hat)) +
  geom_histogram(bins = 5, color = "black") +
  ggtitle("Histogram of b_y_hat")
```



From this graph, it appears that some years yield wildly popular tracks while most years have tracks with average popularity rankings.

```
b_y_hat_test <- test %>%
  left_join(b_y_hat, by = "year") %>%
  .$b_y_hat # b_y estimates for each track release year in test

prediction_3 <- mu_hat + b_j_hat_test + b_y_hat_test

model_3_rmse <- RMSE(test$popularity, prediction_3) # Calculate the RMSE
model_3_rmse
```

```
## [1] 13.6582
```

Adding the year effect to our model improved its RMSE by about 28%, a significant improvement. This is to be expected because as we saw in the correlation plot earlier, the relationship between the popularity ranking and release year of a track had a high positive correlation coefficient of 0.86.

3.5 Model 4: Regularized Energy and Year Effect Model

When we visualized the number of observations for each popularity ranking, we saw that the number of observations for rankings varied widely, in that some rankings had more than 20,000 observations, while others had only one or two observations. This introduces a problem in our data because although popularity rankings with high observation counts generate more precise estimates, rankings with very few observations are harder to fit, and consequently result in larger errors. To solve this issue, we turn to regularization techniques. The concept of regularization refers to adding a penalty term to our loss function in order to penalize large estimates stemming from small sample sizes. To illustrate this concept, we will use the "energy effect" model and add a penalty term to give us this function:

$$\sum_i (y_i - \mu - b_j)^2 + \lambda \sum_j b_j^2$$

where λ represents a regularization parameter used to control the amount of regularization to be applied on the model. As λ increases, the estimated value(s) that minimize the equation shrink towards zero. Conversely, as λ decreases, penalties for each effect become less severe. The above equation can be expanded to include the year effect:

$$\sum_i (y_i - \mu - b_j - b_y)^2 + \lambda \left(\sum_j b_j^2 + \sum_y b_y^2 \right)$$

To minimize this equation, the values of $\hat{b}_j$ and $\hat{b}_y$ are calculated by the following equations:

$$\hat{b}_j = \frac{1}{\lambda + n_j} \sum_{n=1}^{n_j} (Y_i - \hat{\mu})$$

$$\hat{b}_y = \frac{1}{\lambda + n_y} \sum_{n=1}^{n_y} (Y_i - \hat{\mu} - \hat{b}_j)$$

where n_j represents the number of observations for energy level j, and n_y represents the number of observations for year y.

In R, we can create a function to calculate the RMSE of a regularized energy and year effect model given a regularization parameter, a train set, and a test set. This function will be used to cross-validate λ to optimize our model by means of this parameter.

```
# Model 4: Regularized Energy and Year Effect Model

Regularize <- function(lambda, train_set, test_set) { # Function used to choose the best lambda given a

  b_j_hat_train <- train_set %>%
    group_by(energy) %>%
    summarize(b_j_hat = sum(popularity - mu_hat)/(n() + lambda))

  b_y_hat_train <- train_set %>%
```

```r
    left_join(b_j_hat, by = "energy") %>%
    group_by(year) %>%
    summarize(b_y_hat = sum(popularity - mu_hat - b_j_hat)/(n() + lambda))

  prediction <- test_set %>%
    left_join(b_j_hat_train, by = "energy") %>%
    left_join(b_y_hat_train, by = "year") %>%
    mutate(pred = mu_hat + b_j_hat + b_y_hat) %>%
    pull(pred)

  RMSE(test_set$popularity, prediction)

}
```

Now we will perform cross-validation using R's sapply function to choose the best lambda for our model:

```r
# Using cross-validation to choose the best lambda:
rmses <- sapply(seq(0, 10, 0.25), Regularize, train, test)
```

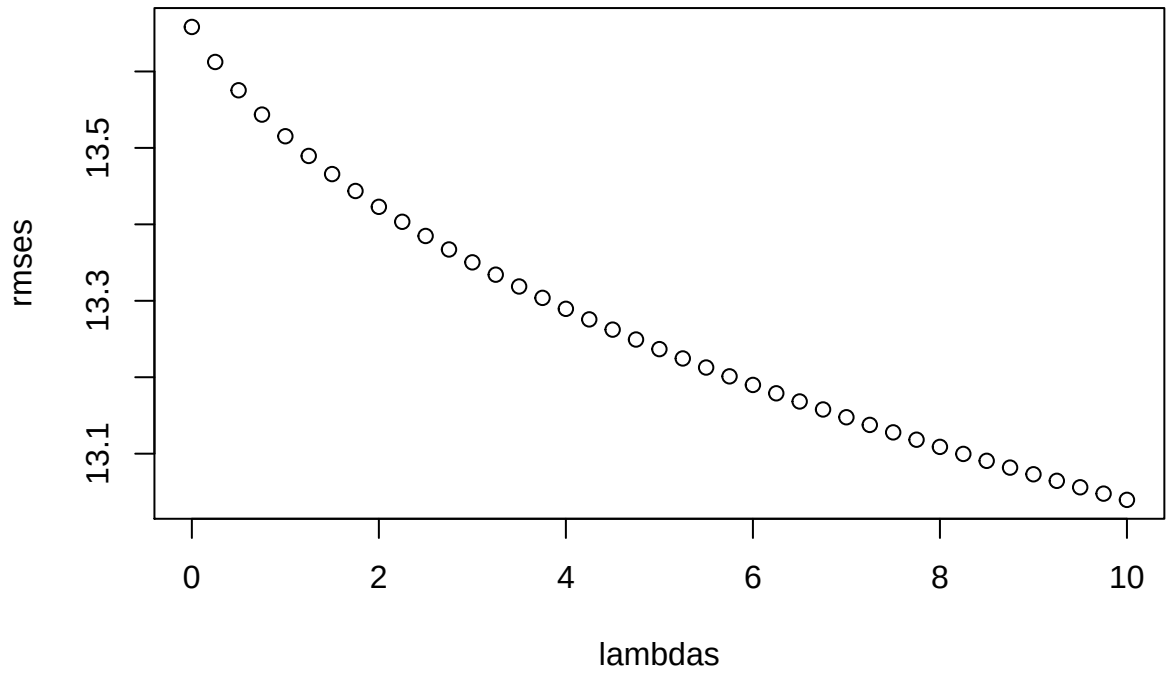
```
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
```

```
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
## `summarise()` ungrouping output (override with `.groups` argument)
```

```r
Lambda <- seq(0, 10, 0.25)[which.min(rmses)] # lambda that yields the lowest RMSE
Lambda
```

```
## [1] 10
```

```
plot(rmses, x = seq(0, 10, 0.25), xlab = "lambdas")
```




```
model_4_rmse <- min(rmses) # RMSE calculated using best lambda
model_4_rmse
```

```
## [1] 13.0396
```

Regularizing our energy and year effect model using a lambda value of 10 improved its RMSE by 4.5%.

3.6 Model 5: Random Forest Model Using Energy and Year as Predictors

Apart from linear regression models, another model we can use to predict track popularity is the random forest model. The random forest is a popular ensemble algorithm, in that it takes a collection of several smaller decision trees and combines them to produce a final prediction. Random forests are known for being able to solve classification problems as well as regression problems. In a random forest, decision trees are grown from a data sample drawn with replacement, which is known as the bootstrap sampling method. Each decision tree is made up of nodes that determine the best split of the data based on a random subset of features, m. The accuracy of the model increases as more trees are grown due to it being an ensemble method. The random forest offers many benefits that contribute to its popularity, including reduced risk of overfitting from averaging multiple decision trees, as well as high accuracy and flexibility for a variety of machine-learning problems.

To train our random forest model, we will need to specify cross-validation parameters. The number of trees that will be grown will be limited to 2 due to the relatively high computational expense of training a random forest. In addition, we will only be using two features for determining accuracy relative to our previous model, which also only used two features.

```
# Model 5: Random Forest Model
# takes a few minutes to run
set.seed(1, sample.kind = "Rounding") # if using R 3.5 or earlier, use set.seed(1)
```

```
## Warning in set.seed(1, sample.kind = "Rounding"): non-uniform 'Rounding' sampler
## used
```

```
control <- trainControl(method = "cv", number = 5, p = 0.8)
Train_fit <- train(popularity ~ energy + year,
                   method = "rf",
                   ntree = 2,
                   data = train,
                   trControl = control,
                   tuneGrid = data.frame(mtry = seq(1, 2, 1)))

model_5_rmse <- RMSE(test$popularity, predict(Train_fit, test))
model_5_rmse
```

```
## [1] 10.78779
```

Predicting popularity ratings using a random forest model improved our RMSE by about 17%.

4 Results

Now, we will combine the RMSEs of all five models into a table:

```
RMSE_table <- data.table(methods = c("Using mu_hat",
                                     "Energy effect model",
                                     "Energy + year effect model",
                                     "Regularized energy + year effect model",
                                     "Random forest model using energy and year as predictors"),
                         RMSEs = c(model_1_rmse,
                                   model_2_rmse,
                                   model_3_rmse,
                                   model_4_rmse,
                                   model_5_rmse),
                         "Improvement Percentage" = c(NA,
                                                      (1 - (model_2_rmse / model_1_rmse)) * 100,
                                                      (1 - (model_3_rmse / model_2_rmse)) * 100,
                                                      (1 - (model_4_rmse / model_3_rmse)) * 100,
                                                      (1 - (model_5_rmse / model_4_rmse)) * 100))

knitr::kable(RMSE_table)
```

methods	RMSEs	Improvement Percentage
Using mu_hat	21.80934	NA
Energy effect model	18.89756	13.351080
Energy + year effect model	13.65820	27.725070
Regularized energy + year effect model	13.03960	4.529125
Random forest model using energy and year as predictors	10.78779	17.268963

In summary, we were able to improve the RMSE with each subsequent model, and successfully brought the RMSEs of models 3, 4, and 5 down to below 15. The model that made the most improvement to the RMSE was the energy and year effect model, improving the energy effect model by nearly 28%. The model that made the least improvement to the RMSE was the regularized energy + year effect model, improving the energy + year effect model by 4.5%. Compared to the initial model, the final random forest model improved the RMSE by over 50%, a significant improvement.

5 Conclusion

5.1 Summary

In conclusion, this project succeeded in building an algorithm that can accurately predict the popularity of a Spotify track given its energy level and release year. Additionally, a target RMSE of below 15 was reached with the final random forest algorithm. To determine this final algorithm, five models were tested on the train data, where each model methodically improved upon the last in predictive accuracy by means of adding effects, regularizing, and training a random forest. Each effect that was added to the models was an observed bias seen in visualizations of the data, such as the fact that some energy levels in the data had more observations than others, and that some years saw a higher number of tracks. The results section of this report showed that of the five models tested, the random forest model generated the most accurate predictions and improved the RMSE of the first model by more than 50%, achieving a final RMSE of about 10.79. The fact that this model performed well on the test set indicates that the model benefitted from the bootstrap method and did not overfit the train data.

5.2 Limitations

Although the final model performed well in accuracy for the test set, it is limited by the number of effects it uses to make predictions. The original data included variables that were not incorporated as effects into the five models, such as acousticness, loudness, valence, and danceability, among several others. Similar to the energy and year effects, there is most likely a relationship between the popularity ranking of a track and its acousticness, and a relationship between the popularity of a track and its danceability measurement. Adding these effects to the final model could potentially have a significant impact on the final RMSE, depending on how strong these effects are. There is a limitation to this approach, however, in that the number of effects to be added to the final model is limited by the number of variables provided by the data. Features not given by the data could include how many times a track was skipped, average listening time, and even how many times a track was shared on social media. Apart from effects, the choice of algorithm for a data science project can also be a limitation depending on how much computer memory an algorithm requires to run, and the practical capability of a computer to compute such algorithms.

5.3 Future Work and Impact

Based on the limitations of this project in regards to the data and strategies used, future work needs to be done to further improve upon the final model. To improve the usefulness of the data, it would be beneficial to include user information not only to generate more accurate predictions by including more effects, but also to make it possible to create a recommendation system based on users' listening behaviors. Additionally, a wider range of algorithms should be tested on the data, some of which may help improve the predictive power of the model by a significant margin. In consideration of the large amounts of memory required to run complex recommendation algorithms such as random forests, improvements in computational capability may open the doors to new, more powerful algorithmic approaches that paint a clearer picture of whether or not a track will gain popularity given its audio and informational attributes.

6 References

Davis, Jason and Vikas A. Aggarwal. "How Spotify and TikTok Beat Their Copycats." Harvard Business Review. Harvard Business Review, July 21, 2020. Accessed December 31, 2020. https://hbr.org/2020/07/how-spotify-and-tiktok-beat-their-copycats

Mejia, Paula. "The Success Of Streaming Has Been Great For Some, But Is There A Better Way?" NPR. NPR, July 22, 2019. Accessed December 31, 2020. https://www.npr.org/2019/07/22/743775196/the-success-of-streaming-has-been-great-for-some-but-is-there-a-better-way

"Get a Track." Spotify for Developers. Spotify AB. Accessed December 31, 2020. https://developer.spotify.com/documentation/web-api/reference/tracks/get-track/

"Get Audio Features for a Track." Spotify for Developers. Spotify AB. Accessed December 31, 2020. https://developer.spotify.com/documentation/web-api/reference/tracks/get-audio-features/

Irizarry, Rafael A. Introduction to Data Science: Data Analysis and Prediction Algorithms with R. Boca Raton, FL: Taylor & Francis Group, LLC, 2020.

Vangumalli, Dinesh. "Regularization." Refactored.ai. Colaberry Inc., 2019. Accessed December 24, 2020. https://refactored.ai/microcourse/notebook?path=content%2F05-Regression_models_in_Machine_Learning%2F05-Regularization%2F01-Regularization.ipynb