3106_Final_Project

Nicki Camberg and Hart Rapaport

5/7/2022

```
#general packages
library(tidyverse)
library(stringi)
library(rmarkdown)
library(spotifyr)
library(knitr)
library(data.table)
library(jsonlite)
library(reshape2)
library(plyr)
library(formatR)
library(h2o)
library(pdp)
library(pls)
library(rsample)

#models
library(gbm)
library(xgboost)
library(caret)
library(lime)
library(MLmetrics)
library(glmnet)
library(Metrics)
library(rstanarm)
library(mlr3measures)

#graphing
library(tidygraph)
library(ggraph)
library(network)
library(igraph)
library(arcdiagram)
library(ggridges)
library(MetBrewer)
library(GGally)
library(scales)
library(ggcorrplot)
```

Redacted API keys

```
set.seed(1)
```

Note! A lot of our code took a very long time to run, so we are including the code here to show what the process looked like, but are not running them and instead are reopening presaved csv/RDS files with the corresponding output.

INTRODUCTION:

Spotify is one of the world's most popular music streaming services. For users, it has a number of differentiating features — one of which is its industry-leading algorithmic curation that powers auto-generated playlists and allows users to discover new songs based on their enjoyment of a given track. In this project, we will analyze whether algorithmic playlists that are similar to human-made playlists can be generated through the use of song features. More specifically, we'll find out whether user-generated "closeness" between two songs (as measured by their Jaccard Index of how often users put a pair of songs on the same playlist) is associated with the "closeness" of the features that Spotify generates for every song.

In terms of data-snooping, we don't believe that a substantial probability exists within our project. The specific issue of p-hacking could only really possibly be present within one of the algorithm types we use for our work — others (such as Principal Component Regression, for example), do not give p-values in a meaningful way for our features. Because we run only a single iteration for each algorithm containing the outcome variable and features of interest, p-hacking isn't particularly likely.

When thinking through this project, we considered a number of approaches before settling on this final framework for our analysis. We first considered doing a genre classification from song features, which would be potentially interesting but is slightly boring in its simplicity, and perhaps lacking a specific audience that would want this information. We then turned to assessing popularity, but that would have required us to pull in data that didn't exist as the Spotify API only provides popularity data at the time of the API call, rather than over time. While we could have brought in another dataset, there is to our knowledge no datasets that have comprehensive data on song popularity to the extent that we needed, and decided that the methodology and relationship described above would be the best avenue to examine these data.

In terms of iterations for this topic, we considered a number of different algorithms before settling on a mix of simpler and more complex regression methods (precise reasoning behind our selection of each model type will be outlined below). We also thought through which features to include as predictors for our algorithm (discussed below). Finally, we iterated through how to subset our data so as to balance execution time with more thorough results. This last part will be discussed in the explanation of the datasets below.

Our work has value for a number of different audiences. Most obviously, Spotify and similar music services with data on song features will want to know how well their algorithmic methods of playlist curation match up to how their millions of users actually group songs together within the playlists they make themselves. The best situation for each of these companies would be to create an algorithmic playlist that matches the quality of a hand selected, human curated playlist instantly without having to employ anyone besides data scientists. If song feature similarity is a great predictor of playlist pairings, then these services could just use it to save people all the trouble of building the playlist themselves. For example, if Spotify notices that you enjoy a given rock song, they could build an entirely customized playlist based on feature similarity that would be a similar quality to what you could build yourself. Such a feature would be a great selling point for a music service, and definitely attract new users and retain existing ones

Also, newer artists without a lot of data on how their songs are consumed on Spotify or similar platforms might be interested in this analysis. If the features of a song are reliable predictors of what types of tracks that song is paired up with in playlists, then musicians can tweak their music to have certain values for different features to try to get paired up with tracks that they, or their management, think (for whatever reason) might be better suited to their music and would expose them to new fans who would like their work.

We use two datasets for this project. The first is Spotify's Million Playlist Dataset, a collection of 1,000,000 playlists sourced from the platform. Per their documentation, the dataset "includes public playlists created by US Spotify users between January 2010 and November 2017" (https://www.aicrowd.com/challenges/spotify-million-playlist-dataset-challenge). Furthermore, "playlists are sampled with some randomization, are manually filtered for playlist quality and to remove offensive content, and have some dithering and fictitious tracks added to them." For our purposes, we're interested in the list of song IDs that each playlist contains. While the dithering and fictitious tracks are potentially worrying, we do not believe that they've impacted our analysis as any fictitious tracks would simply not return a result within our second dataset and so would not be included in the eventual regression.

The second dataset is Spotify's API, accessed through the spotifyr wrapper that allows us to run API calls within R (https://cran.r-project.org/web/packages/spotifyr/spotifyr.pdf). This is not a dataset per se, but it provides information about any of the songs we want to investigate by querying the Spotify database. We are primarily interested in the get_track_audio_features function, which takes in a list of song IDs and provides a dataframe that reports 12 audio features for each song along with a number of pieces of metadata: danceability, energy, key, loudness, mode, speechiness, acousticness, instrumentalness, liveness, valence, tempo, and time signature. More information about each of these features is not included here for reasons of brevity, but can be found at the following link: https://developer.spotify.com/documentation/web-api/reference/#/operations/get-several-audio-features.

We use these features and not others because we want to understand whether there is something about the way that a song sounds (as measured via these features) that people pick up on when making playlists and associating songs with each other. Including other features that are not directly related to this (data on the artist, for example) would, of course, make our model better. However, we do not do this as it would lead to worse actual results in terms of generating interesting musical combinations even while providing better predictive results in terms of error metrics. We care about generating unexpected insights as those are the types of things that power playlist generators that people pay for. A model including the artist would simply return that songs with the same artist are more likely to be matched together — not an interesting insight.

So as to get accurate results (and so that computational time would finish at least somewhat quickly and to avoid API rate limiting), we decided we had to subset the possible universe by including only the 2000 songs with the most occurences in the 100,000 playlists that had 681,805 songs in total. Including songs with only a single occurrence in the dataset, for example, would lead to noisy data as our algorithm would consider this to be the same as equal to data generated from a song with 500 occurrences, leading to faulty results. Accordingly, we include only the top 2000 songs (with a minimum of 503 occurrences in the playlists). The number 2000 is a bit arbitrary, but we thought that it struck a good balance between too much and too little data, because we will be analyzing every potential pair between these songs, so including 5000 songs, for example, would balloon the eventual dataset to 5000 choose 2 or 12.5 million observations — a number that would not provide considerably increased data quality but would significantly impact running time.

One downside is that this limits our universe of analysis to songs that are popular. The playlists, to a certain extent, reflect the music that is popular among Spotify users who make playlists — rap and pop, rather than classical music or hyperpop or any other number of more niche genres. We exacerbate that by choosing the songs with the most data, potentially limiting the external validity of our results when applied to types of music that are not as present in the top 2000 songs. We attempt to measure whether this is a problem below by testing our data on entirely different playlists from different files.

The outcome variable (and our created feature) is the Jaccard Index for each pair of songs. This is calculated by taking the cardinality of their intersection divided by the cardinality of their union, and is between 0 (no members of the intersection set) and 1 (the members of the intersection set are equal to the members of the union set). We considered a number of other metrics as our outcome variable. One was simply taking the number of times a pair appeared and dividing this number by the number of appearances of the first song in the pair to get one outcome variable and by the number of appearances of the second song in the pair to get the second. Of course, this would produce two different outcome variables for the same set of predictors, so this idea was discarded in favor of the Jaccard Index, which would produce a single number for each pair of songs.

Reading in and Cleaning Data

first set of playlists, playlists 0-99999

```
pls <- lapply(list.files(path="/Users/nickicamberg/Desktop/stat3106/spot_playlists/",
                         pattern=NULL, all.files=FALSE,
                         full.names=TRUE), read_json)

pls <- lapply(pls, `[[`, 2)

pls <- unlist(pls, recursive = FALSE, use.names = TRUE)

#getting a list of all the tracks in all the playlists
track_list <- vector(mode = "list", length = length(pls))
for(i in 1:length(pls)){
  track_list[[i]] <- pls[[i]]$tracks
}

#getting how many songs are in each playlist to split them later
length_playlists <- c()
for(i in 1:length(track_list)){
  length_playlists[i] <- length(track_list[[i]])
}


playlist_numbers <- rep(1:length(pls), length_playlists)

#melting the tracklist
tmelt<- unlist(track_list, recursive = FALSE)

#creating an empty vector that holds all the URIs
uri_vec <- vector(mode = "list", length = length(tmelt))

#creating a single vector that has all the URIs
for(i in 1:length(tmelt)){
  uri_vec[[i]] <- tmelt[[i]]$track_uri}

uri_vec <- unlist(uri_vec)

#getting a table of the 2000 songs that appear the most
#there are 681805 songs
top_uri <- sort(table(uri_vec),decreasing=TRUE)[1:2000]

#vector of the song uris that appear the most
top_uris <- names(top_uri)

#making a vector of only the top 2k songs, rest are NA
uri_vec2 <- ifelse(uri_vec %in% top_uris, uri_vec, NA)

#splitting the vector of tracks by playlist number
tracks_split <- split(uri_vec2, playlist_numbers)

#making that into a data frame
```

```
uri_frame <- t(stri_list2matrix(tracks_split))
uri_frame <- as.data.frame(uri_frame)

#because all of that data cleaning takes forever to run,
#we are saving them to csvs and then reopening them

#saving the uri frame and a few other things
saveRDS(pls, file = "/Users/nickicamberg/Desktop/stat3106/pls.rds")
saveRDS(uri_vec, file = "/Users/nickicamberg/Desktop/stat3106/uri_vec.rds")
write.csv(uri_frame,"/Users/nickicamberg/Desktop/stat3106/uri_frame_100000.csv")
saveRDS(top_uri, file = "/Users/nickicamberg/Desktop/stat3106/top_uri.rds")
saveRDS(uri_vec2, file = "/Users/nickicamberg/Desktop/stat3106/uri_vec2.rds")
```

```
#reading them back in
pls <- readRDS("/Users/nickicamberg/Desktop/stat3106/pls.rds")
uri_vec <- readRDS("/Users/nickicamberg/Desktop/stat3106/uri_vec.rds")
#uri_frame <- read_csv("/Users/nickicamberg/Desktop/stat3106/uri_frame_100000.csv")
top_uri <- readRDS("/Users/nickicamberg/Desktop/stat3106/top_uri.rds")
top_uris <- names(top_uri)
uri_vec2 <- readRDS("/Users/nickicamberg/Desktop/stat3106/uri_vec2.rds")
```

Making a df of all possible pairs of songs, and get how many times they actually occur

```
#Make dataframe where each top song is a row, other songs are columns,
#and track how many times the column song is in a playlist the row song is in:

duplicate <- uri_frame %>%
  dplyr::mutate(name = row_number()) %>%
  pivot_longer(!name, names_to = 'variable',
               values_to = 'element') %>%
  drop_na()
duplicate <- duplicate %>%
  select(-variable) %>%
  group_by(element) %>%
  dplyr::mutate(numdups = n()) %>%
  filter(numdups > 1) %>%
  select(-numdups)

duplicate <-setDT(duplicate)

pair_table <- duplicate[duplicate, on = c('name'), allow.cartesian = TRUE][
  element<i.element, .N, .(pair = paste0(element, " ", i.element))]

pair_split <- as.data.frame(str_split_fixed(pair_table$pair, " ", 2))

#Get all possible pairs:
all_pairs <- as.data.frame(t(combn(top_uris, 2, FUN = NULL, simplify = TRUE)))

#Split the possible pairs to compare them with the pair_split and all the ones we're missing:

#Get the first song in alphabetical order to be first in the pair
```

```r
#and do the same with the second to make sure we can match by comb:

all_pairs$comb <- ifelse(all_pairs$V1 > all_pairs$V2,
                         paste0(all_pairs$V1, "-", all_pairs$V2),
                         paste0(all_pairs$V2, "-", all_pairs$V1))


pair_table$pair1 <- pair_split$V1
pair_table$pair2 <- pair_split$V2

pair_table$comb <- ifelse(pair_table$pair1 > pair_table$pair2,
                          paste0(pair_table$pair1, "-", pair_table$pair2),
                          paste0(pair_table$pair2, "-", pair_table$pair1))

pair_table <- bind_rows(pair_table,anti_join(all_pairs,pair_table,by="comb"))

#Fill in missing values for pairs without occurrences:
pair_table$pair1 <- ifelse(is.na(pair_table$pair1),
                           pair_table$V1, pair_table$pair1)

pair_table$pair2 <- ifelse(is.na(pair_table$pair2),
                           pair_table$V2, pair_table$pair2)

pair_table$N <- ifelse(is.na(pair_table$N),
                       0, pair_table$N)

pair_table <- subset(pair_table, select=-c(pair, comb, V1, V2))

occurences<-as.data.frame(table(unlist(uri_frame2)))

pair_table <- merge(pair_table, occurences,
                    by.x = "pair1", by.y = "Var1")
pair_table <- merge(pair_table, occurences,
                    by.x = "pair2", by.y = "Var1")


pair_table$jaccard <- (pair_table$N)/(pair_table$Freq.x +
                                      pair_table$Freq.y -
                                      pair_table$N)

pair_table$pair1 <- gsub("spotify:track:", "", pair_table$pair1)
pair_table$pair2 <- gsub("spotify:track:", "", pair_table$pair2)

#saving it so we don't need to run it again
saveRDS(pair_table, file = "/Users/nickicamberg/Desktop/stat3106/pair_table.rds")
```

```r
#reading it back in
pair_table <- readRDS("/Users/nickicamberg/Desktop/stat3106/pair_table.rds")
```

Getting the audio features for all of the 2000 top songs

```
all_songs <- unique(c(unique(pair_table$pair1),
                      unique(pair_table$pair2)))

#making an empty dataframe to store the song features
all_songs_data<- data.frame(matrix(ncol = 18,
                                   nrow = length(all_songs)))

#rename the columns with the feature names
names(all_songs_data) <- names(get_track_audio_features(all_songs[1],
                                          authorization = get_spotify_access_token()))

#running a loop to get the features
for(i in 1:length(all_songs)){
  all_songs_data[i,] <- get_track_audio_features(all_songs[i],
                                          authorization = get_spotify_access_token())
}

#saving it so we don't need to run it again
write.csv(all_songs_data,"/Users/nickicamberg/Desktop/stat3106/all_songs_data.csv",
          row.names = FALSE)
```

```
#reading it back in
all_songs_data <- read_csv("/Users/nickicamberg/Desktop/stat3106/all_songs_data.csv")
```

Merging together the dataframe of pairs and the song features, and finding the difference in each value for each pair

```
#Merge them back with main frame:
merge_frame <- merge(pair_table, all_songs_data,
                     by.x = "pair1", by.y = "id",
                     all.x = TRUE)
merge_frame <- merge(merge_frame, all_songs_data,
                     by.x = "pair2", by.y = "id",
                     all.x = TRUE)

#Remove extra columns:
merge_frame <- merge_frame %>%
  dplyr::select(-contains(c("type", "id", "uri",
                            "track_href", "analysis_url",
                            "duration_ms")))

#Get difference for each feature:
merge_frame$danceability <- abs(merge_frame$danceability.x -
                                  merge_frame$danceability.y)
merge_frame$energy <- abs(merge_frame$energy.x - merge_frame$energy.y)
merge_frame$key <- abs(merge_frame$key.x - merge_frame$key.y)
merge_frame$loudness <- abs(merge_frame$loudness.x - merge_frame$loudness.y)
merge_frame$mode <- abs(merge_frame$mode.x - merge_frame$mode.y)
merge_frame$speechiness <- abs(merge_frame$speechiness.x - merge_frame$speechiness.y)
```

```
merge_frame$acousticness <- abs(merge_frame$acousticness.x - merge_frame$acousticness.y)
merge_frame$instrumentalness <- abs(merge_frame$instrumentalness.x -
                                      merge_frame$instrumentalness.y)
merge_frame$liveness <- abs(merge_frame$liveness.x - merge_frame$liveness.y)
merge_frame$valence <- abs(merge_frame$valence.x - merge_frame$valence.y)
merge_frame$tempo <- abs(merge_frame$tempo.x - merge_frame$tempo.y)
merge_frame$time_signature <- abs(merge_frame$time_signature.x -
                                    merge_frame$time_signature.y)

#Delete individual feature columns:
merge_frame <- merge_frame[,c(1:6, 31:42)]

#saving it so we don't need to run it again
write.csv(merge_frame,"/Users/nickicamberg/Desktop/stat3106/merge_frame.csv", row.names = FALSE)
```

```
#reading it back in
merge_frame <- read_csv("/Users/nickicamberg/Desktop/stat3106/merge_frame.csv")
```

Getting the data for each song for year released, artist, genre, etc

```
#code to get the date of the song and also the top genre of the artist and the artist/song name
#use all_songs
all_songs <- unique(c(unique(pair_table$pair1),
                      unique(pair_table$pair2)))
song_name <- c()
album_id <- c()
album_date <- c()
artist_id <- c()
artist_name <- c()
artist_genres <-  vector(mode = "list", length = length(all_songs))


for(i in 1:length(all_songs)){
  song_holder <- get_track(all_songs[i],
                           authorization = get_spotify_access_token())

  song_name[i] <- song_holder[["name"]]

  album_id[i] <- song_holder[["album"]][["id"]]

  album_holder <- get_album(album_id[i], authorization = get_spotify_access_token())

  album_date[i] <- album_holder[["release_date"]]

  artist_id[i] <- album_holder[["artists"]][["id"]]

  artist_name[i] <- album_holder[["artists"]][["name"]]

  artist_genres[[i]] <- get_artist(artist_id[i],
                                   authorization = get_spotify_access_token())[["genres"]]
```

```
}


#now finding the most commonly occuring genre for each song based on the frequency of
#genres in the whole df
table_ug <- table(unlist(artist_genres))
top_genre <- c()

for(i in 1:length(all_songs)){
  genre_holder <- unlist(c(artist_genres[[i]]))
  top_genre[i] <- ifelse(length(artist_genres[[i]]) > 0,
                         names(which.max(table_ug[names(table_ug) %in% genre_holder])),
                         "NONE")
}

#merging everything into a single dataframe
track_info <- as.data.frame(cbind(all_songs, song_name, artist_name,
                                  top_genre, album_date,
                                  artist_id, album_id))

#getting a column for year
track_info$album_year <- substr(track_info$album_date, 1, 4)

#saving these in so we don't need to run them again
write.csv(track_info,"/Users/nickicamberg/Desktop/stat3106/track_info.csv")
saveRDS(artist_genres, file = "/Users/nickicamberg/Desktop/stat3106/artist_genres.rds")
```

```
#reading it back in
track_info <- read_csv("/Users/nickicamberg/Desktop/stat3106/track_info.csv")
artist_genres <- readRDS("/Users/nickicamberg/Desktop/stat3106/artist_genres.rds")
```

When analyzing the dataset, we found a number of things. First, our initial data checks ("Does each playlist have the listed number of songs in it?" and "Does each playlist have the listed number of songs in it that's greater than or equal to the number of albums," for instance) came back with the expected result of TRUE. We expect the data quality to be high given Spotify's ownership of the data, but we checked to ensure we thoroughly reviewed the data. In terms of the "fake songs" that Spotify added to the playlists, our analysis did not pick up on them, as they were not the top 2000 songs by occurrences, so this was not a problem in our project.

Exploratory Data Analysis

```
#Exploratory data analysis:

#Data checks:

#Does each playlist have the listed number of songs in it?
num_tracks <- vector(length=length(pls))
listed_num_tracks <- vector(length=length(pls))

for(i in 1:length(pls))
{
```

```
  num_tracks[i] <- length(pls[[i]][["tracks"]])
  listed_num_tracks[i] <- pls[[i]][["num_tracks"]]
}
setequal(num_tracks,listed_num_tracks)
```

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

```
#Does each playlist have a number of songs that is greater than or equal to the number of albums?
listed_num_albums <- vector(length=length(pls))

for(i in 1:length(pls))
{
  listed_num_albums[i] <- pls[[i]][["num_albums"]]
}

sum(num_tracks >= listed_num_albums) == length(listed_num_albums)
```

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

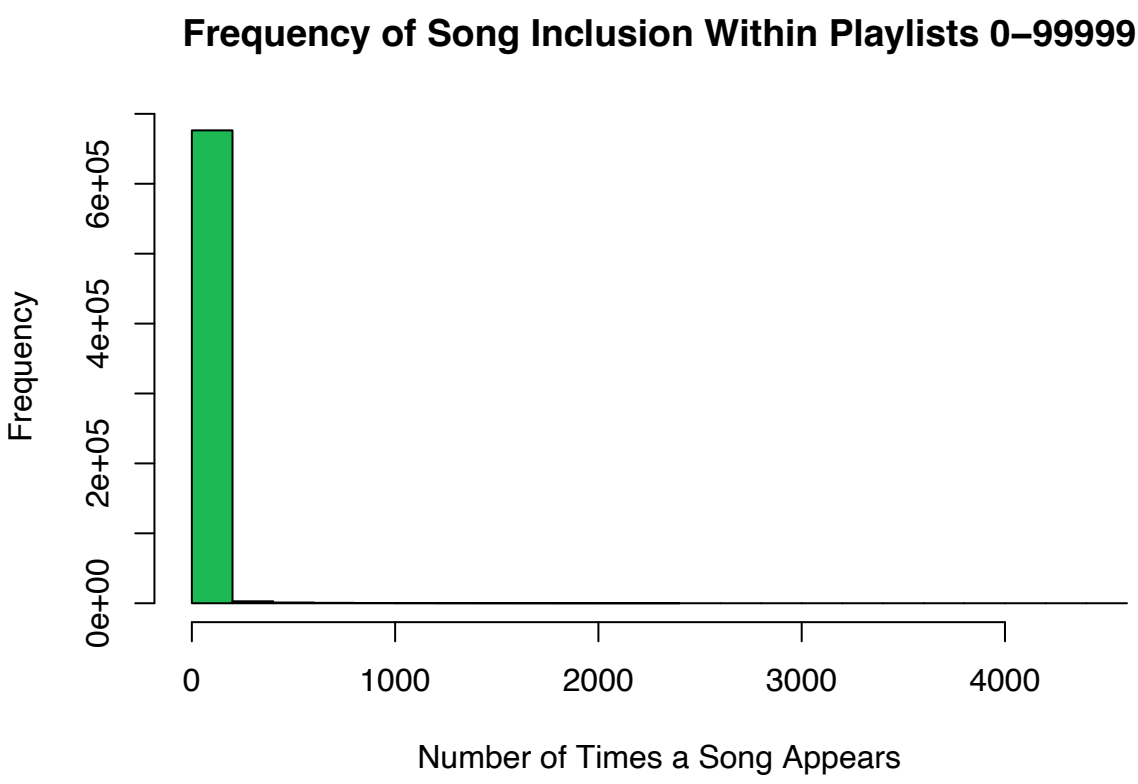
```
#Check number of unique tracks:
length(unique(uri_vec))
```

```
## [1] 681805
```

```
#How does the number of unique tracks compare to the number of total tracks?
length(unique(uri_vec))/length(uri_vec)
```

```
## [1] 0.1021002
```

```
#Information on how many times each song is included:
hist(as.numeric(table(uri_vec)),
     xlab="Number of Times a Song Appears",
     main="Frequency of Song Inclusion Within Playlists 0-99999",
     col = "#1DB954")
```



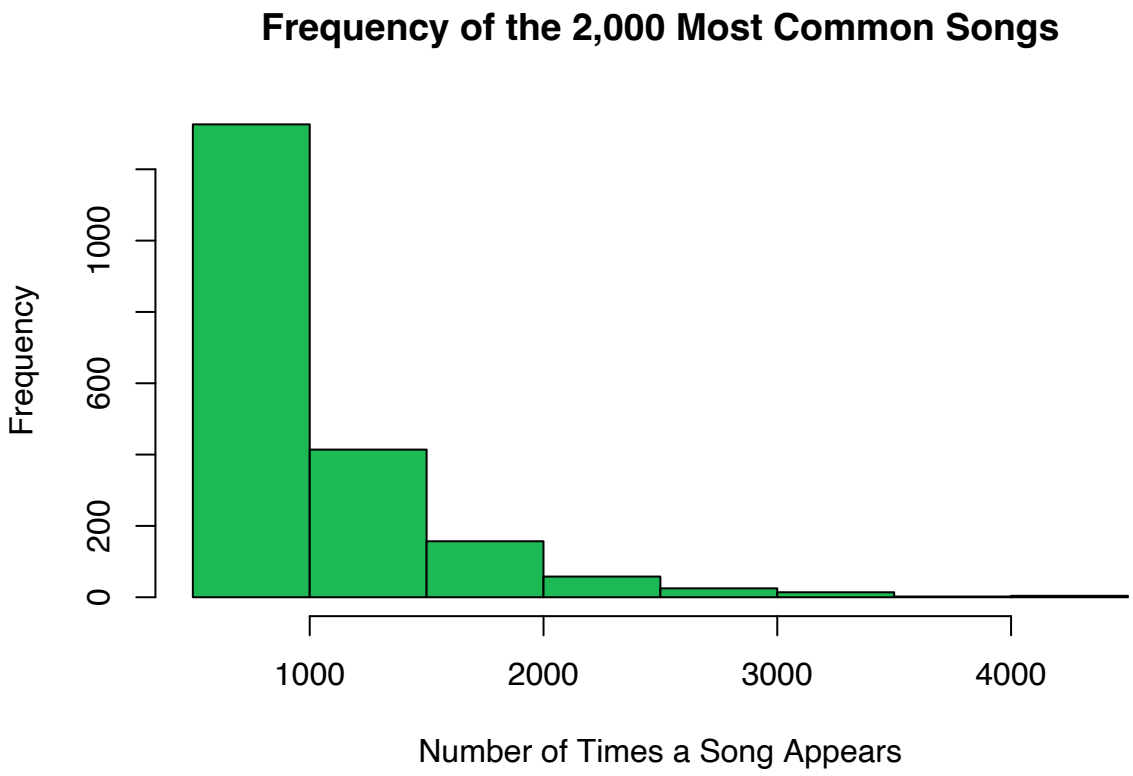

```
quantile(as.numeric(table(uri_vec)), probs = seq(0, 1, by = .1))
```

```
##   0%  10%  20%  30%  40%  50%  60%  70%  80%  90% 100%
##    1    1    1    1    1    1    2    3    4   11 4441
```

```
summary(as.numeric(table(uri_vec)))
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
##    1.000    1.000    1.000    9.794    3.000 4441.000
```

```
#Information on how many times each song is included within the top 2k
hist(as.numeric(top_uri),
    xlab="Number of Times a Song Appears",
    main="Frequency of the 2,000 Most Common Songs",
    col = "#1DB954")
```




```
quantile(as.numeric(top_uri), probs = seq(0, 1, by = .1))
```

```
##     0%    10%    20%    30%    40%    50%    60%    70%    80%    90%   100%
##  503.0  546.0  598.0  647.0  714.6  806.5  914.0 1054.9 1267.0 1615.0 4441.0
```

```
summary(as.numeric(top_uri))
```

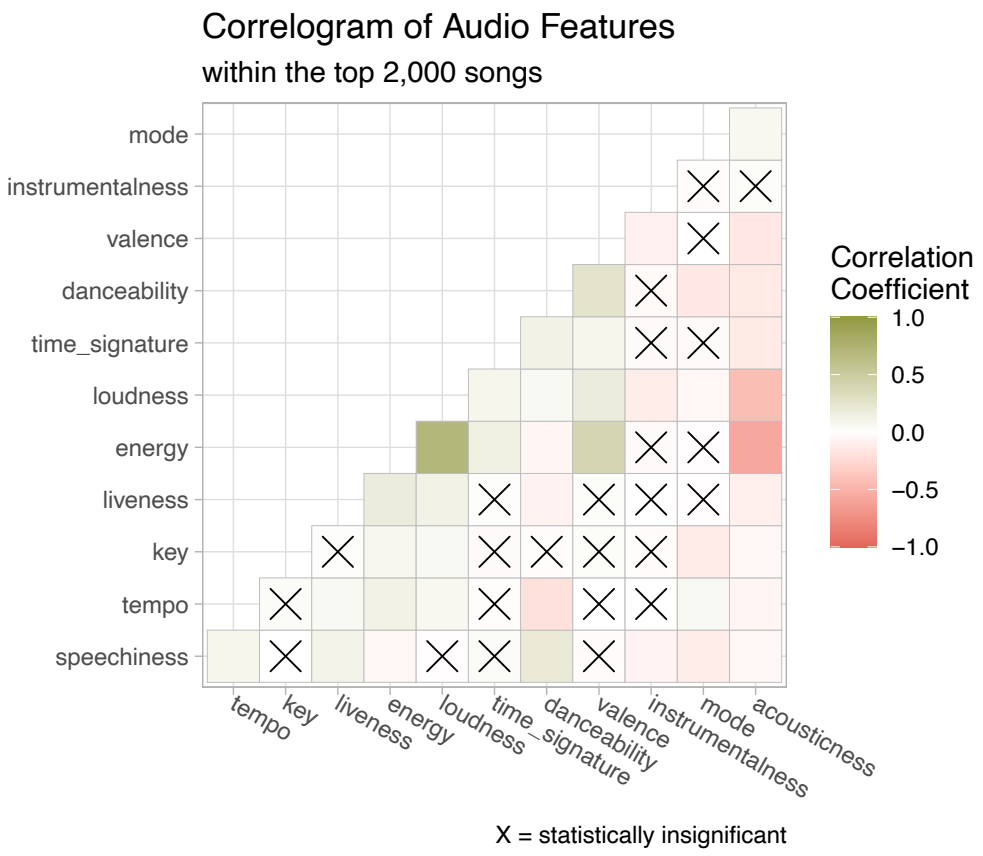
```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   503.0   620.0   806.5   977.2  1158.5  4441.0
```

Correlograms revealed substantial correlations in certain respects, especially with "energy" and other features. Because collinearity can be a problem, we'll be taking steps to address this later in our project.

Correlogram of All Audio Features

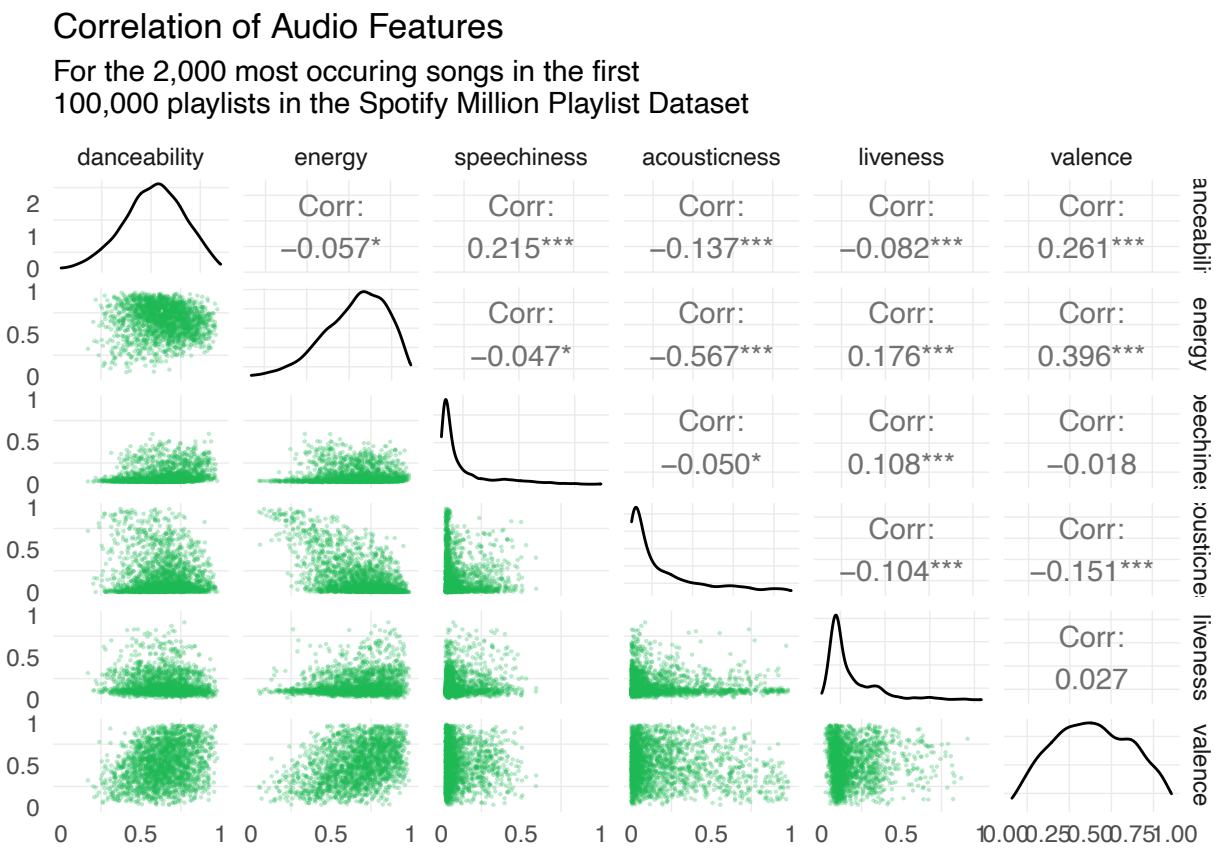
```
corr_asd <- all_songs_data %>% dplyr::select(-type, -id, -uri, -track_href,
                                             -analysis_url, -duration_ms)

nums <- sapply(corr_asd, is.numeric)

corrs <- round(cor(corr_asd[nums],
                   use="pairwise.complete.obs"), 2)
```

```
p_mats <- cor_pmat(corr_asd[nums],
                   use="pairwise.complete.obs")
corrs %>%
  ggcorrplot(type = "lower",
             hc.order = TRUE,
             p.mat = p_mats,
             lab_size = 3.5,
             legend.title = "Correlation\nCoefficient",
             colors = c("#e3655b",
                        "white",
                        "#929a42")) +
  labs(title = "Correlogram of Audio Features",
       subtitle = "within the top 2,000 songs",
       caption = "X = statistically insignificant") +
  theme_light() +
  theme(
    axis.text.x = element_text(angle = 330, vjust = 0.5,
                               hjust=0),
    axis.title.x = element_blank(),
    axis.title.y = element_blank())
```



Correlation Matrix of Some Audio Features

```
lowerfun <- function(data,mapping){
  ggplot(data = data, mapping = mapping)+
    geom_point(alpha = 0.3, size=0.1, color ="#1DB954")+
    scale_x_continuous(limits = c(0,1), breaks = seq(0, 1, by = .5), labels = c(0, .5, 1)) +
    scale_y_continuous(limits = c(0,1), breaks = seq(0, 1, by = .5), labels = c(0, .5, 1))
}

all_songs_data %>%
  dplyr::select(-uri, -id, -type, -track_href,
                -analysis_url, -duration_ms, -key,
                -instrumentalness, -mode, -tempo,
                -loudness, -time_signature) %>%
  ggpairs(color = "#1DB954",
          lower = list(continuous = wrap(lowerfun)),
          ) +
  theme_minimal() +
  theme(panel.grid.major = element_blank()) +
  labs(title="Correlation of Audio Features",
       subtitle = "For the 2,000 most occuring songs in the first\n100,000 playlists in the Spotify Mil
```



We were also curious to see which song pairings had the strongest relationship, both in terms of Jaccard index and how often they are paired together. It seems that more situation specific songs, like Christmas

songs or Disney songs had the highest Jaccard values, which makes sense as those kinds of songs tend to only be put on playlists with other songs in the same genre, as most people won't just put "Jingle Bell Rock" on their gym playlist or whatnot, but for a holiday specific playlists. Songs that were paired together the most were interestingly all rap songs and by a small handful of artists, which may just reflect their overall popularity and overrepresentation in this dataset.

Songs Pairings with the highest Jaccards

```
#getting a df of all the pairs by genre and how often they happen
genre_pairs <- merge_frame %>%
  filter(N > 0) %>%
  dplyr::select(c("pair2", "pair1", "N", "Freq.x", "Freq.y", "jaccard"))

genre_pairs <- left_join(genre_pairs, track_info, by=c("pair2" = "all_songs"))
genre_pairs <- left_join(genre_pairs, track_info, by=c("pair1" = "all_songs"))

genre_pairs %>%
  arrange(desc(jaccard)) %>%
  slice(1:10) %>%
  dplyr::select(jaccard, N,
                song_name.x, artist_name.x, top_genre.x,
                song_name.y, artist_name.y, top_genre.y) %>%
  dplyr::mutate(jaccard = round(jaccard, 3)) %>%
  kable(booktabs="T")
```

jaccard	N	song_name.x	artist_name.x	top_genre.x	song_name.y	artist_name.y	top_genre.y
0.763	487	Under the Sea	Alan Menken	hollywood	Kiss the Girl	Alan Menken	hollywood
0.730	553	A Holly Jolly Christmas - Single Version	Burl Ives	adult standards	Rockin' Around The Christmas Tree - Single Version	Brenda Lee	adult standards
0.716	524	Under the Sea	Alan Menken	hollywood	A Whole New World	Various Artists	NONE
0.681	436	Part of Your World	Alan Menken	hollywood	Kiss the Girl	Alan Menken	hollywood
0.680	479	Part of Your World	Alan Menken	hollywood	Under the Sea	Alan Menken	hollywood
0.679	526	Jingle Bell Rock	Anita Kerr Singers	easy listening	Rockin' Around The Christmas Tree - Single Version	Brenda Lee	adult standards
0.665	459	A Whole New World	Various Artists	NONE	Kiss the Girl	Alan Menken	hollywood
0.664	491	Under the Sea	Alan Menken	hollywood	Hakuna Matata - From "The Lion King"/Soundtrack	Various Artists	NONE
0.663	461	A Holly Jolly Christmas - Single Version	Burl Ives	adult standards	Jingle Bell Rock	Anita Kerr Singers	easy listening

jaccard	N	song_name.x	artist_name.x	top_genre.x	song_name.y	artist_name.y	top_genre.y
0.650	472	A Holly Jolly Christmas - Single Version	Burl Ives	adult standards	The Christmas Song (Merry Christmas To You)	Nat King Cole	adult standards

Pairings that occur the most often

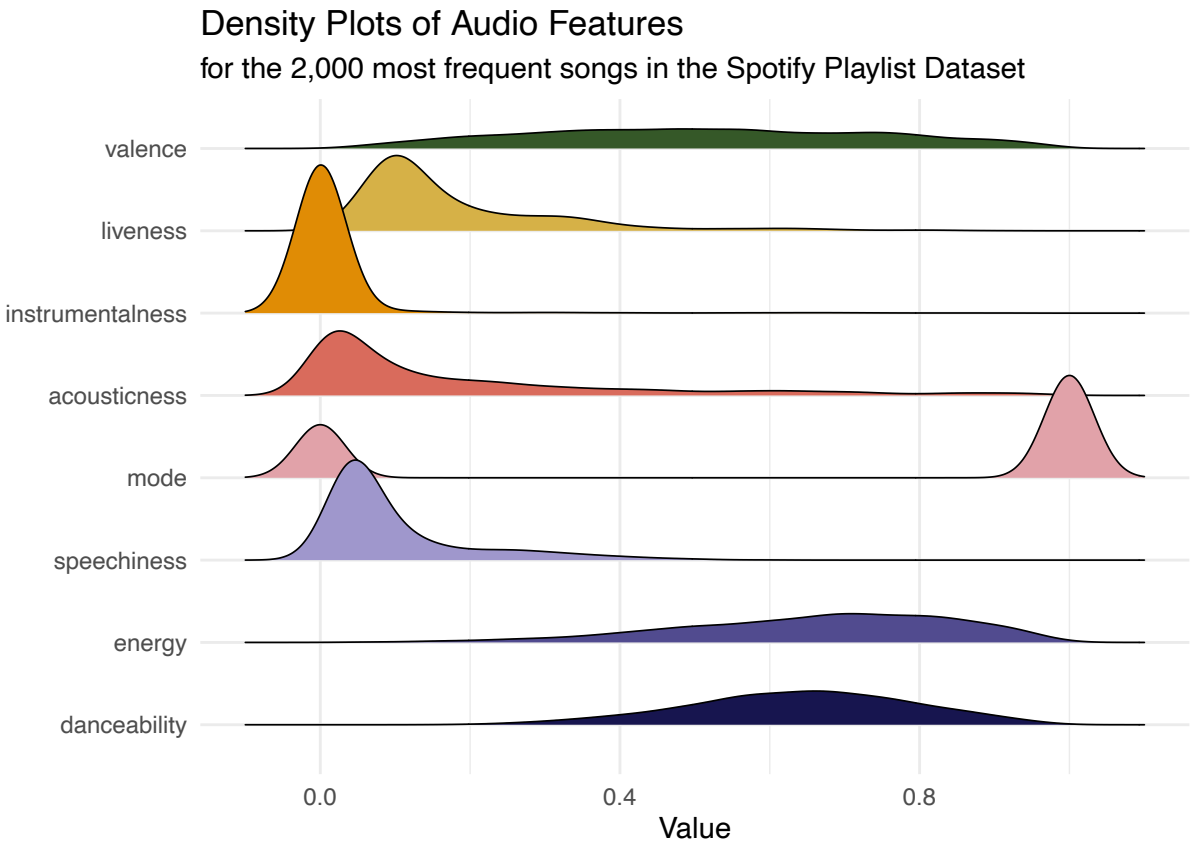
```
genre_pairs %>%
  arrange(desc(N)) %>%
  slice(1:10) %>%
  dplyr::select(jaccard, N,
                song_name.x, artist_name.x, top_genre.x,
                song_name.y, artist_name.y, top_genre.y) %>%
  dplyr::mutate(jaccard = round(jaccard, 3)) %>%
  kable(booktabs="T")
```

jaccard	N	song_name.x	artist_name.x	top_genre.x	song_name.y	artist_name.y	top_genre.y
0.428	2110	HUMBLE.	Kendrick Lamar	rap	DNA.	Kendrick Lamar	rap
0.359	2086	HUMBLE.	Kendrick Lamar	rap	XO Tour Llif3	Lil Uzi Vert	rap
0.355	1986	HUMBLE.	Kendrick Lamar	rap	Mask Off	Future	rap
0.307	1969	HUMBLE.	Kendrick Lamar	rap	Congratulations	Post Malone	rap
0.370	1775	Bad and Boujee (feat. Lil Uzi Vert)	Migos	rap	Bounce Back	Big Sean	pop
0.313	1761	XO Tour Llif3	Lil Uzi Vert	rap	Congratulations	Post Malone	rap
0.358	1738	XO Tour Llif3	Lil Uzi Vert	rap	Mask Off	Future	rap
0.296	1736	Broccoli (feat. Lil Yachty)	Shelley FKA DRAM	rap	Caroline	Aminé	pop
0.282	1630	Congratulations	Post Malone	rap	iSpy (feat. Lil Yachty)	KYLE	pop
0.273	1612	HUMBLE.	Kendrick Lamar	rap	goosebumps	Travis Scott	rap

We then embarked on a number of actions for our exploratory data analysis. We first report a density plot for the different features. These plots show us that some (instrumentalness, valence, tempo) are roughly normal while others (mode, for example) have discrete peaks. Mode, of course, is bimodal because it only has two values: 0 or 1.

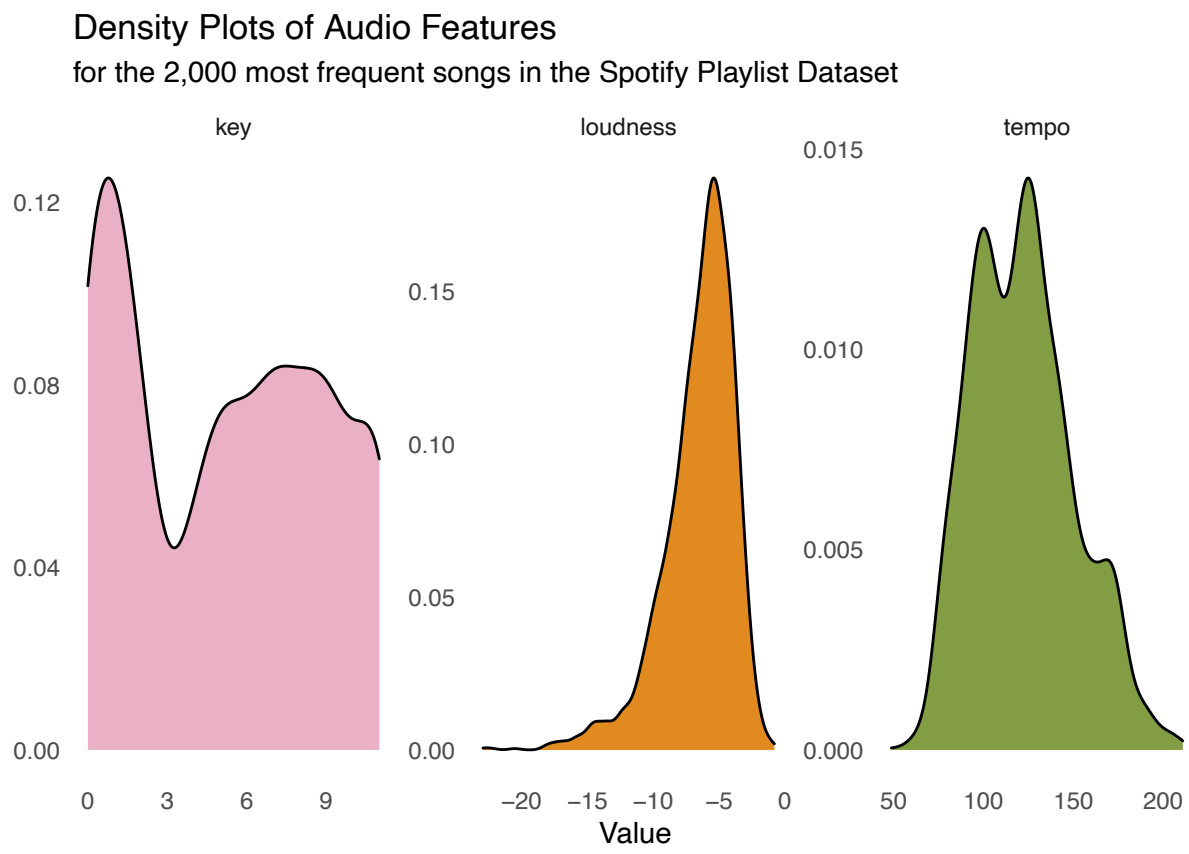
Density Plots

```r
all_songs_data %>%
  dplyr::select(id, danceability, energy, speechiness,
                mode, acousticness, instrumentalness,
                liveness, valence) %>%
  reshape2::melt(id.vars = c("id"),
                 variable.name = "Feature",
                 value.name = "Value") %>%
  ggplot(aes(x=Value, y=Feature, fill=Feature)) +
  labs(title = "Density Plots of Audio Features",
       subtitle = "for the 2,000 most frequent songs in the Spotify Playlist Dataset",
       y = "") +
  geom_density_ridges(size=.3) +
  scale_fill_manual(values=met.brewer("Renoir",
                                      8, "continuous")) +
  theme_minimal() +
  theme(legend.position = "none")
```




```r
asd2 <- left_join(track_info, all_songs_data, by = c("all_songs"="id"))
asd2 %>%
  dplyr::select(all_songs, key, loudness,
                tempo) %>%
  reshape2::melt(id.vars = c("all_songs"),
                 variable.name = "Feature",
                 value.name = "Value") %>%
```

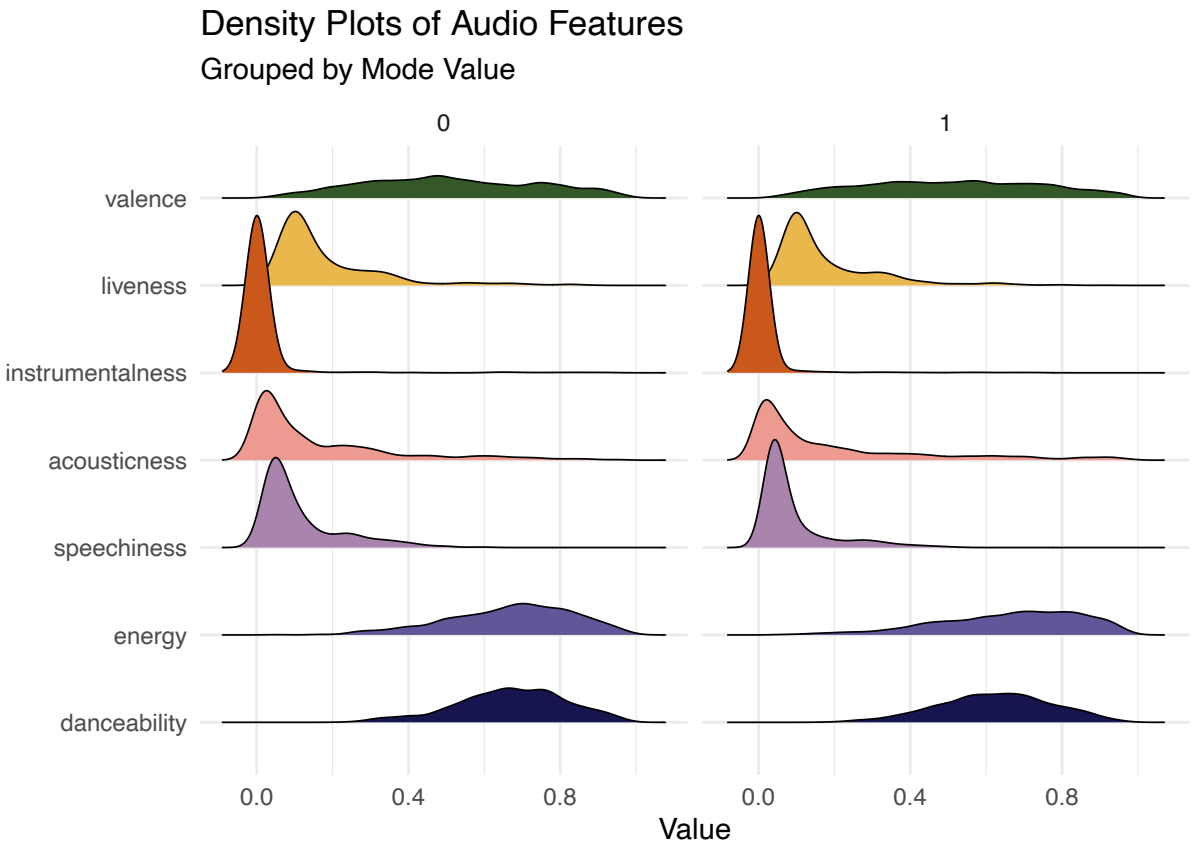
```
ggplot(aes(x=Value, fill=Feature)) +
labs(title = "Density Plots of Audio Features",
     subtitle = "for the 2,000 most frequent songs in the Spotify Playlist Dataset",
     y = "") +
geom_density(size=.5) +
scale_fill_manual(values=met.brewer("Tara",
                                    3, "continuous")) +
facet_wrap(~Feature, scales="free") +
theme_minimal() +
theme(legend.position = "none") +
theme(panel.grid.major = element_blank(),
      panel.grid.minor = element_blank(),
      panel.background = element_blank())
```



Questions of feature value overlap (i.e. are there values of valences across the valence spectrum both when mode = 0 and when mode = 1) are critical for causal inference. Even though that's not the goal of our project, besides the overall plot we additionally report one example of this for mode = 0 and mode = 1. This reveals two plots that are nearly the same, suggesting that (at least for this metric) there would be no inferential problems (if that was the goal of our project) related to examining the relationship of mode on other features. For our purposes, this is important because a key assumption of linear regression is linearity. If mode = 0 and mode = 1 had substantially different density plots, then adding them into the regression equation (without the use of interaction terms) would mean that linearity could be violated as a + 1 increase for energy might mean something different for songs with mode = 0 and mode = 1 in real life — though the model would treat that coefficient as precisely the same either way.

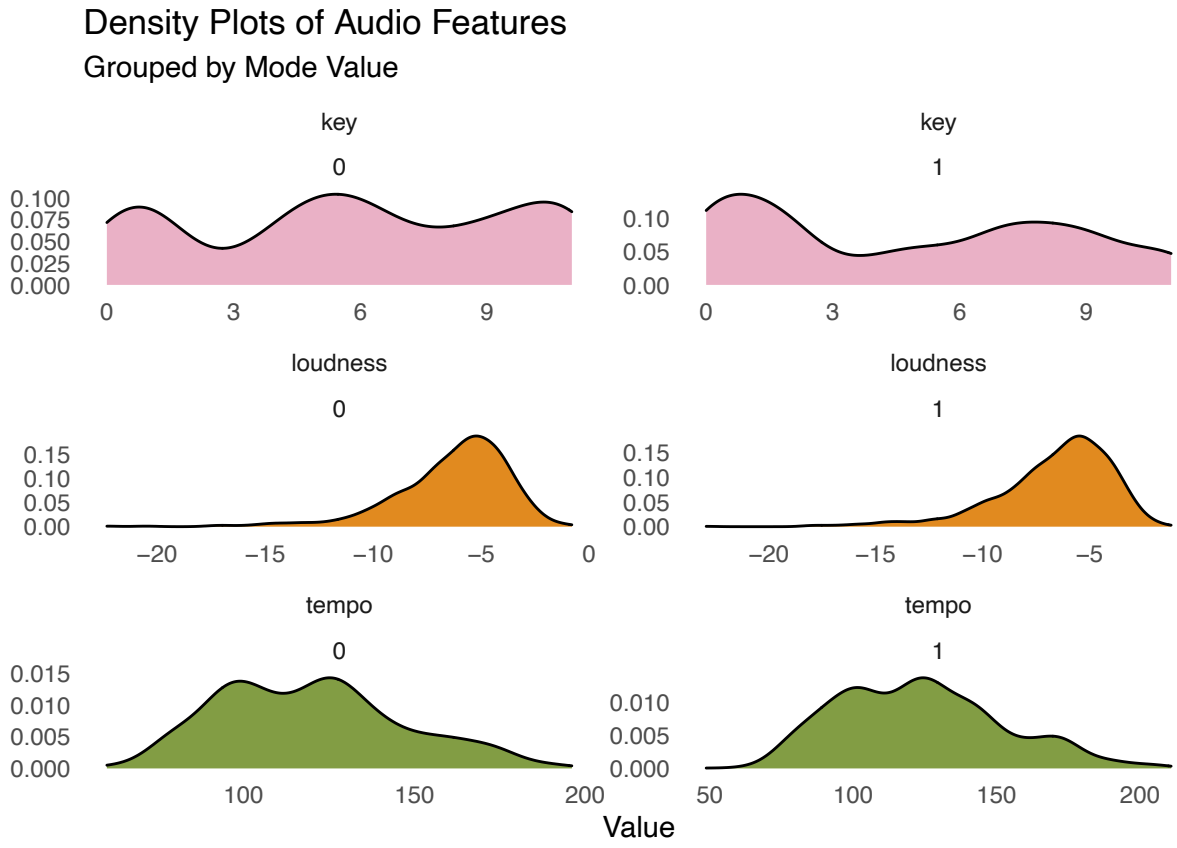
Density Plots by Mode

```
all_songs_data %>%
  dplyr::select(id, danceability, energy, speechiness,
                mode, acousticness, instrumentalness,
                liveness, valence) %>%
  reshape2::melt(id.vars = c("id", "mode"),
                 variable.name = "Feature",
                 value.name = "Value") %>%
  ggplot(aes(x=Value, y=Feature, fill=Feature)) +
  labs(title = "Density Plots of Audio Features",
       subtitle = "Grouped by Mode Value",
       y = "") +
  geom_density_ridges(size=.3) +
  scale_fill_manual(values=met.brewer("Renoir",
                                      7, "continuous")) +
  facet_wrap(~mode)+
  theme_minimal() +
  theme(legend.position = "none")
```




```
asd2 %>%
  dplyr::select(all_songs, mode, key, loudness,
                tempo) %>%
  reshape2::melt(id.vars = c("all_songs", "mode"),
```

```
                variable.name = "Feature",
                value.name = "Value") %>%
ggplot(aes(x=Value, fill=Feature)) +
labs(title = "Density Plots of Audio Features",
     subtitle = "Grouped by Mode Value",
     y = "") +
geom_density(size=.5) +
scale_fill_manual(values=met.brewer("Tara",
                                    3, "continuous")) +
facet_wrap(Feature~mode, scales="free", ncol=2) +
theme_minimal() +
theme(legend.position = "none") +
theme(panel.grid.major = element_blank(),
      panel.grid.minor = element_blank(),
      panel.background = element_blank())
```



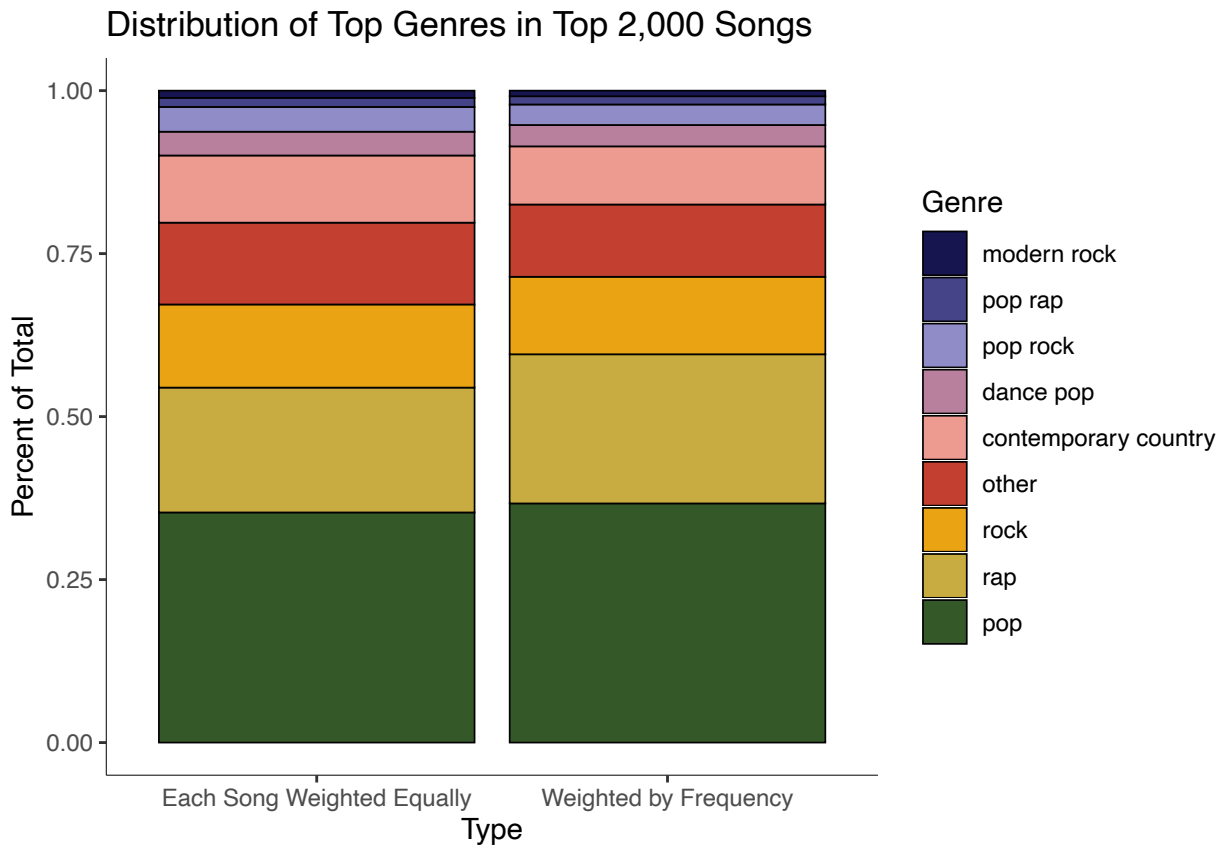
We also looked at the genre data for each of our songs. The Spotify API doesn't give genres for each song, but we generated songs for the corresponding artist by filtering out the list of genres for each artist to keep only the one that appeared the most often.

Within the top 2,000 songs, there were 76 unique genres, the most common of which were pop, rap, and rock. These genre distributions were basically the same within all the pairs, but there was an overrepresentation of pop and rap songs (and corresponding rap and pop musicians) in the pairs as certain pop and rap songs tended to be the most common songs in the playlist, with many of the top songs being either pop or rap. As would be expected, when we made network charts of pairings between genres, the most frequent pairings were between the most popular and similar genres, like pop and dance pop, or rap and pop.

Stacked Percentage Barplot of Top Genre Frequency

```
#frequency of genres within entire dataframe that has been subsetted to the 2k songs
uri_vec2<-uri_vec2[!is.na(uri_vec2)]
uri_vec2 <- as.data.frame(uri_vec2)
overall_frequency <- uri_vec2 %>%
  group_by(uri_vec2) %>%
  dplyr::summarise(count=n())

overall_frequency$uri_vec2 <- gsub("spotify:track:", "", overall_frequency$uri_vec2)

overall_frequency <- left_join(overall_frequency, track_info, by =c("uri_vec2" ="all_songs"))

overall_frequency <- overall_frequency %>%
  group_by(top_genre) %>%
  dplyr::summarise(overall = sum(count)) %>%
  arrange(desc(overall))

#frequency of genres within 2k songs
unique_genre_f <- track_info %>%
  group_by(top_genre) %>%
  dplyr::summarise(unique = n()) %>%
  arrange(desc(unique))

#merging them
genre_freq <- merge(overall_frequency, unique_genre_f)

#finding the top genres to make the rest be "other" for the viz
top_8_genres <- genre_freq %>%
  dplyr::mutate(overall_perc = (overall/1954359)*100,
                unique_perc = (unique/2000)*100) %>%
  arrange(desc(unique_perc)) %>%
  top_n(8,unique_perc) %>% .$top_genre


genre_freq$top_genre <- ifelse(genre_freq$top_genre %in% top_8_genres,
                               genre_freq$top_genre,
                               "other")

#some more cleanup, reordering factors
genre_freq <- genre_freq %>%
  group_by(top_genre) %>%
  dplyr::summarise(once = sum(unique),
                   with_all = sum(overall)) %>%
  arrange(desc(with_all))

genre_freq$top_genre <- reorder(genre_freq$top_genre , genre_freq$with_all)
genre_freq$top_genre <- factor(genre_freq$top_genre , levels=levels(genre_freq$top_genre))

#plotting!
genre_freq %>%
  reshape2::melt(id.vars = c("top_genre"),
                 variable.name = "Group",
```

```
                value.name = "Value") %>%
  ggplot(aes(x=Group, y=Value, fill=top_genre)) +
  geom_bar(position="fill", stat="identity", color = "black", size=.3) +
  scale_fill_manual(values=met.brewer("Renoir",
                                      9, "continuous"))+
    labs(title = "Distribution of Top Genres in Top 2,000 Songs",
         fill = "Genre",
         x="Type",
         y="Percent of Total") +
    scale_x_discrete(labels = c("Each Song Weighted Equally", "Weighted by Frequency")) +
    theme_bw() +
    theme(panel.grid.major = element_blank(),
          axis.line = element_line(colour = "black", size=.1),
          panel.grid.minor = element_blank(),
          panel.border = element_blank(),
          panel.background = element_blank())
```



Preliminary Code to make Network/Edge Diagrams

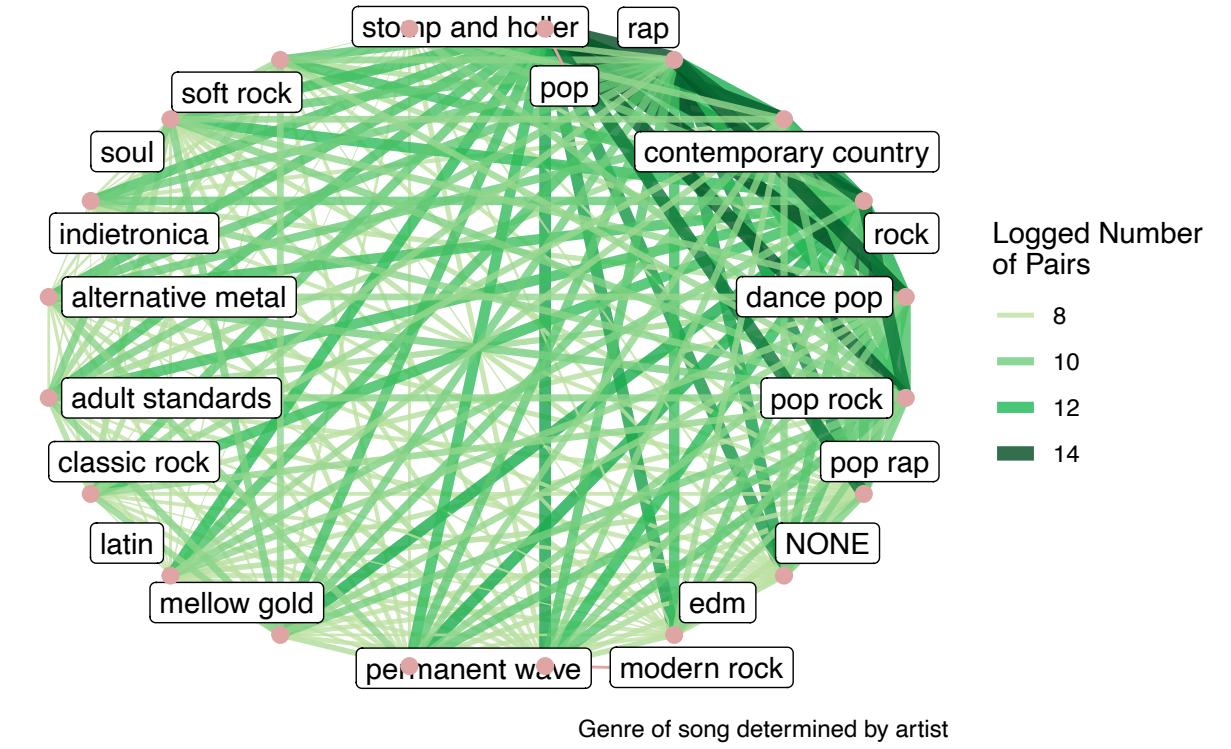
```
#getting a df of all the pairs by genre and how often they happen
genre_pairs <- merge_frame %>%
  filter(N > 0) %>%
  dplyr::select(c("pair2", "pair1", "N", "Freq.x", "Freq.y", "jaccard"))
```

```r
genre_pairs <- left_join(genre_pairs, track_info, by=c("pair2" = "all_songs"))
genre_pairs <- left_join(genre_pairs, track_info, by=c("pair1" = "all_songs"))

pairs_weight <- genre_pairs %>%
  dplyr::select(top_genre.x, top_genre.y, N) %>%
  group_by(top_genre.x, top_genre.y) %>%
  dplyr::summarise(weight = sum(N)) %>%
  ungroup() %>%
  arrange(desc(weight))

#subsetting to top 20 genres
top_genres <- names(sort(table(track_info$top_genre),
                         decreasing=TRUE)[c(1:20)])
pairs_weight_top <- pairs_weight %>%
  filter(top_genre.x %in% top_genres &
           top_genre.y %in% top_genres)

#getting rid of duplicate pairs
pwt2 <- as.data.frame(t(apply(pairs_weight_top, 1, sort)))

names(pwt2) <- c("N", "g1", "g2")
pwt2$N <- as.numeric(pwt2$N)

pwt2 <- pwt2 %>%
  group_by(g1, g2) %>%
  dplyr::summarise(weight = sum(N)) %>%
  ungroup() %>%
  arrange(desc(weight))

#code to get edges and nodes
genre1 <- pwt2 %>%
  distinct(g1) %>%
  dplyr::rename(label = g1)

genre2 <- pwt2 %>%
  distinct(g2) %>%
  dplyr::rename(label = g2)

nodes <- full_join(genre1, genre2, by = "label")

nodes <- nodes %>% rowid_to_column("id")

edges <- pwt2 %>%
  left_join(nodes, by = c("g1" = "label")) %>%
  dplyr::rename(g1_id = id)

edges <- edges %>%
  left_join(nodes, by = c("g2" = "label")) %>%
  dplyr::rename(g2_id = id)


edges2 <- select(edges, g1_id, g2_id, weight)
```

```
edges2 <- edges2 %>% filter(g1_id != g2_id)

routes_tidy <- tbl_graph(nodes = nodes, edges = edges2, directed = TRUE)
```

Linear Circular Network Graph

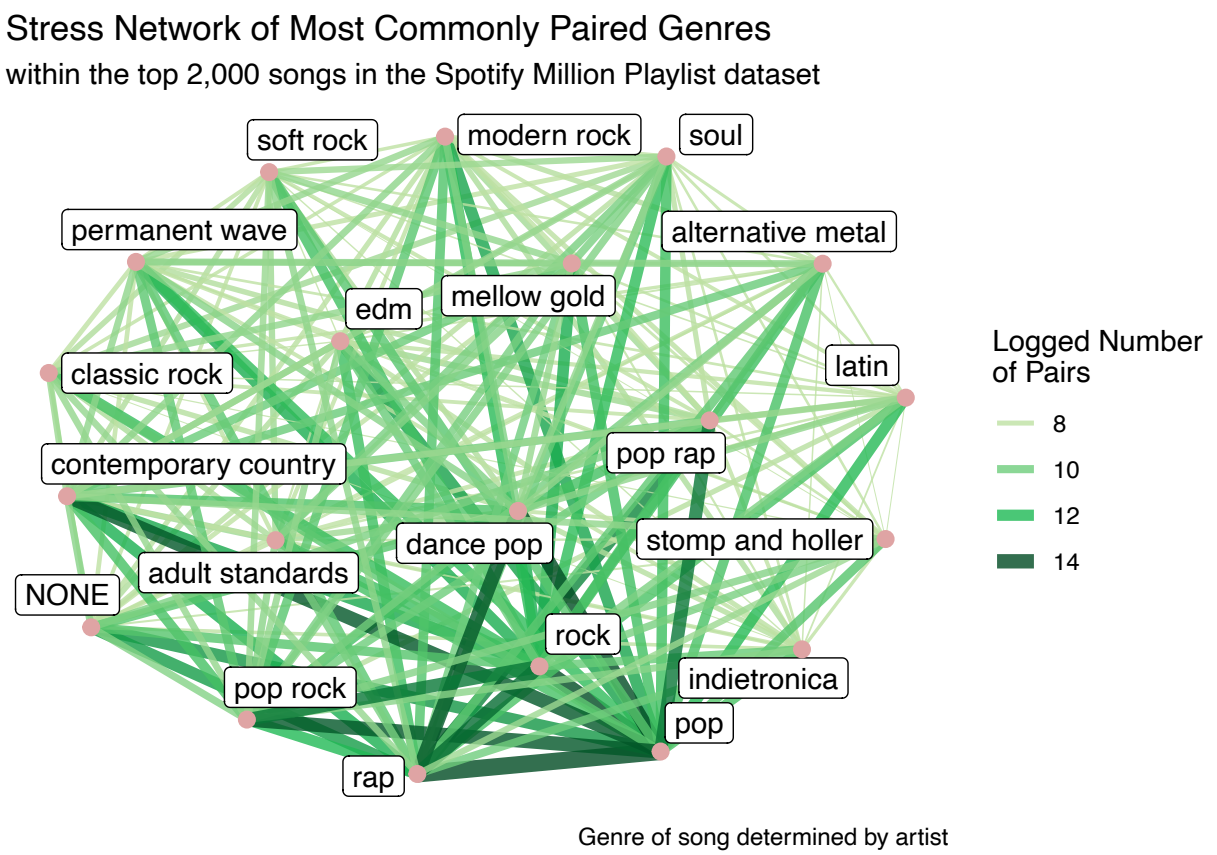
```
ggraph(routes_tidy, layout = "linear", circular = TRUE) +
  geom_edge_link(aes(width = log(weight),
                     color=log(weight)),
                 alpha = 0.8) +
  geom_node_label(aes(label = label), repel = TRUE,
                  segment.color = "#DFA4A4") +
  geom_node_point(color="#DFA4A4", size=2.5) +
  theme_graph() +
  scale_edge_color_gradientn(
    colors = c("#bee1a3", "#6ECD7C", "#1db954", "#004b23"),
    limits = c(8, 14),
    breaks=c(seq(8, 14, by=2)),
    labels=c(seq(8, 14, by=2)),
    oob = scales::squish,
    name = "Logged Number\nof Pairs",
    na.value="grey",
    guide="legend") +
  scale_edge_width_continuous(name="Logged Number\nof Pairs",
                          breaks=c(seq(8, 14, by=2)),
                          labels=c(seq(8, 14, by=2)),
                          range = c(0.15, 3)) +
  guides(color=guide_legend(),
         width=guide_legend()) +
  theme_bw() +
  theme(axis.title=element_blank(),
        axis.text=element_blank(),
        axis.ticks=element_blank(),
        panel.grid = element_blank(),
        panel.border = element_blank(),
        panel.background = element_blank()) +
  labs(title="Linear Circle Network of Most Commonly Paired Genres",
       subtitle="within the top 2,000 songs in the Spotify Million Playlist dataset",
       caption = "Genre of song determined by artist")
```



Stress Network Graph

```
ggraph(routes_tidy, layout = "stress") +
  geom_edge_link(aes(width = log(weight),
                     color=log(weight)),
                 alpha = 0.8) +
  geom_node_label(aes(label = label), repel = TRUE,
                  segment.color = "#DFA4A4") +
  geom_node_point(color="#DFA4A4", size=2.5) +
  theme_graph() +
  scale_edge_color_gradientn(
    colors = c("#bee1a3", "#6ECD7C", "#1db954", "#004b23"),
    limits = c(8, 14),
    breaks=c(seq(8, 14, by=2)),
    labels=c(seq(8, 14, by=2)),
    oob = scales::squish,
    name = "Logged Number\nof Pairs",
    na.value="grey",
    guide="legend") +
  scale_edge_width_continuous(name="Logged Number\nof Pairs",
                        breaks=c(seq(8, 14, by=2)),
                        labels=c(seq(8, 14, by=2)),
                        range = c(0.15, 3)) +
  guides(color=guide_legend(),
```

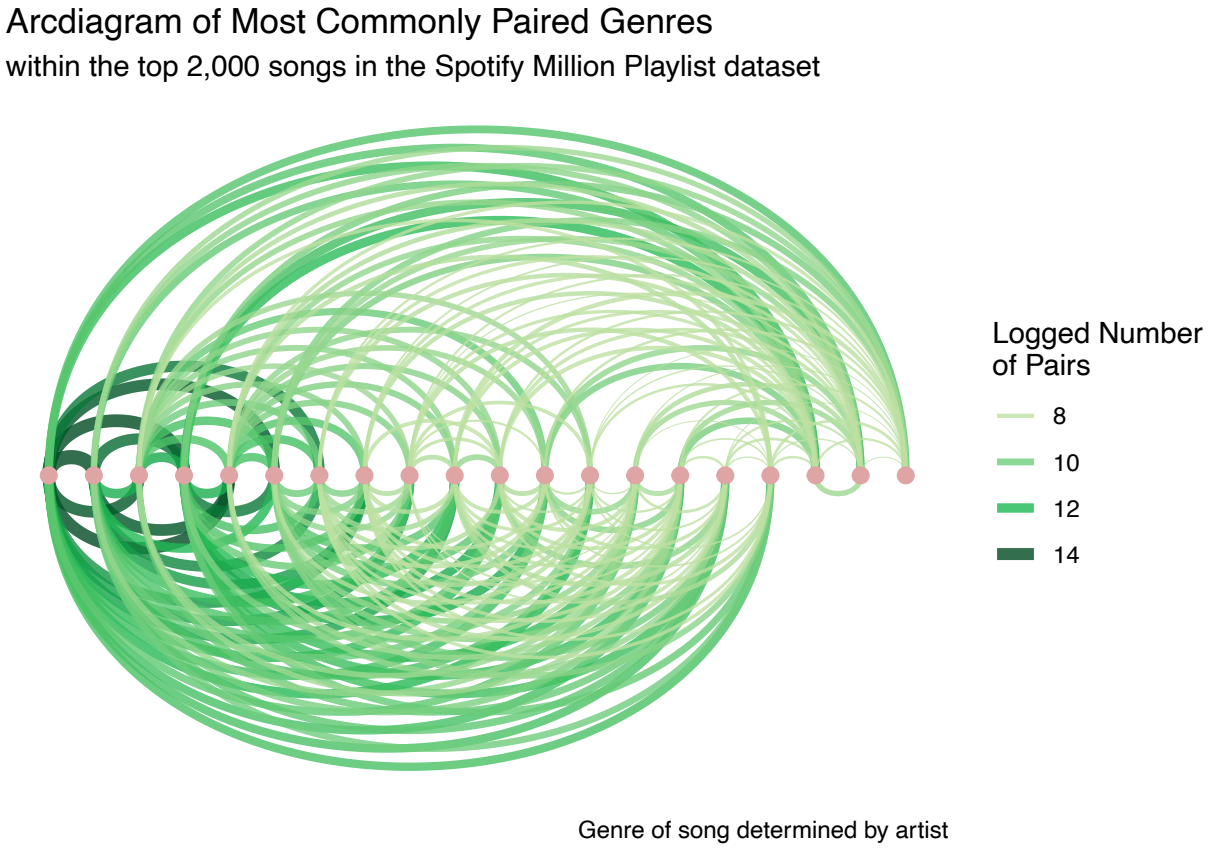
```
        width=guide_legend()) +
  theme_bw() +
  theme(axis.title=element_blank(),
        axis.text=element_blank(),
        axis.ticks=element_blank(),
        panel.grid = element_blank(),
        panel.border = element_blank(),
        panel.background = element_blank()) +
  labs(title="Stress Network of Most Commonly Paired Genres",
       subtitle="within the top 2,000 songs in the Spotify Million Playlist dataset",
       caption = "Genre of song determined by artist")
```



Arcdiagram Pairs

```
ggraph(routes_tidy, layout = "linear") +
  geom_edge_arc(aes(width = log(weight),
                    color=log(weight)),
                alpha = 0.8) +
  geom_node_point(color="#DFA4A4", size=2.5) +
  theme_graph() +
  scale_edge_color_gradientn(
    colors = c("#bee1a3", "#6ECD7C", "#1db954", "#004b23"),
    limits = c(8, 14),
```

```
  breaks=c(seq(8, 14, by=2)),
  labels=c(seq(8, 14, by=2)),
  oob = scales::squish,
  name = "Logged Number\nof Pairs",
  na.value="grey",
  guide="legend") +
scale_edge_width_continuous(name="Logged Number\nof Pairs",
                        breaks=c(seq(8, 14, by=2)),
                        labels=c(seq(8, 14, by=2)),
                        range = c(0.15, 2.5)) +
guides(color=guide_legend(),
       width=guide_legend()) +
theme_bw() +
theme(axis.title=element_blank(),
      axis.text=element_blank(),
      axis.ticks=element_blank(),
      panel.grid = element_blank(),
      panel.border = element_blank(),
      panel.background = element_blank()) +
labs(title="Arcdiagram of Most Commonly Paired Genres",
     subtitle="within the top 2,000 songs in the Spotify Million Playlist dataset",
     caption = "Genre of song determined by artist")
```



Interestingly, when we made network diagrams of the average Jaccard index between genres, the strongest relationships were between the less popular genres like adult standards and soul, or classic rock, soft rock,

and mellow gold, while pop and rap had much weaker relationships with all the other genres. This makes sense, because the music of the more "indie" genres are pretty similar to each other than to more mainstream music and are more likely to share audiences with each other and have similar fan bases, based on our own personal experience.

Showing which genres have the highest avg Jaccard with each other

```
gp_jac <- genre_pairs %>%
  dplyr::mutate(sum_jac = jaccard*N) %>%
  group_by(top_genre.x, top_genre.y) %>%
  dplyr::summarise(avg_jac = sum(sum_jac)/sum(N),
                   tot_N= sum(N)) %>%
  arrange(desc(avg_jac))

gp_jac2 <- as.data.frame(t(apply(gp_jac[,1:2], 1, sort)))
gp_jac <- data.frame(gp_jac2, gp_jac[,3:4])
names(gp_jac) <- c("genre1", "genre2", "avg_jac", "N")
gp_jac$N <- as.numeric(gp_jac$N)
gp_jac$avg_jac <- as.numeric(gp_jac$avg_jac)

genre_jac <- gp_jac %>%
  dplyr::mutate(sum_jac = avg_jac*N) %>%
  group_by(genre1, genre2) %>%
  dplyr::summarise(avg_jac = sum(sum_jac)/sum(N),
                   tot_N= sum(N)) %>%
  ungroup() %>%
  arrange(desc(avg_jac))

genre_jac %>%
  dplyr::slice_head(n=10) %>%
  dplyr::mutate(avg_jac = round(avg_jac, 3)) %>%
  kable(booktabs="T")
```

genre1	genre2	avg_jac	tot_N
hollywood	hollywood	0.709	1402
canadian pop	canadian pop	0.602	477
adult standards	easy listening	0.574	2169
hollywood	NONE	0.511	7832
easy listening	soft rock	0.325	285
reggae	reggae	0.324	2146
adult standards	adult standards	0.300	11518
NONE	NONE	0.274	10691
easy listening	pop	0.274	792
canadian pop	easy listening	0.269	500

Network Diagram by Average Genre Jaccarad

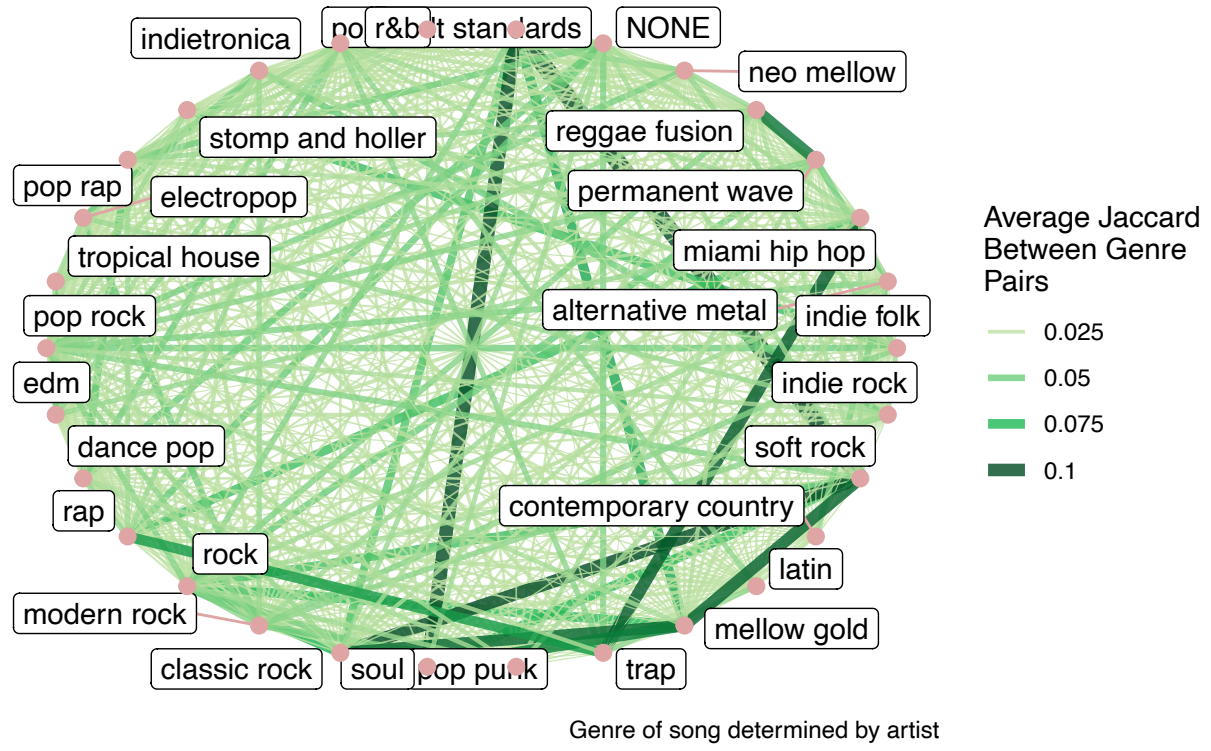
```
#subsetting to top 30 genres
top_genres <- names(sort(table(track_info$top_genre),
```

```
                        decreasing=TRUE)[c(1:30)])
genre_jac_top <- genre_jac %>%
  filter(genre1 %in% top_genres &
         genre2 %in% top_genres) %>%
  dplyr::select(genre1, genre2, avg_jac)

names(genre_jac_top) <- c("g1", "g2", "aj")
genre_jac_top$aj <- as.numeric(genre_jac_top$aj)

#code to get edges and nodes
genre1 <- genre_jac_top %>%
  distinct(g1) %>%
  dplyr::rename(label = g1)

genre2 <- genre_jac_top %>%
  distinct(g2) %>%
  dplyr::rename(label = g2)

nodes_j <- full_join(genre1, genre2, by = "label")

nodes_j <- nodes_j %>% rowid_to_column("id")

edges_j <- genre_jac_top %>%
  left_join(nodes_j, by = c("g1" = "label")) %>%
  dplyr::rename(g1_id = id)

edges_j <- edges_j %>%
  left_join(nodes_j, by = c("g2" = "label")) %>%
  dplyr::rename(g2_id = id)


edges2_j <- select(edges_j, g1_id, g2_id, aj)

edges2_j <- edges2_j %>% filter(g1_id != g2_id)

routes_tidy_j <- tbl_graph(nodes = nodes_j, edges = edges2_j, directed = TRUE)
```

Linear Circular Network Graph: Jaccard

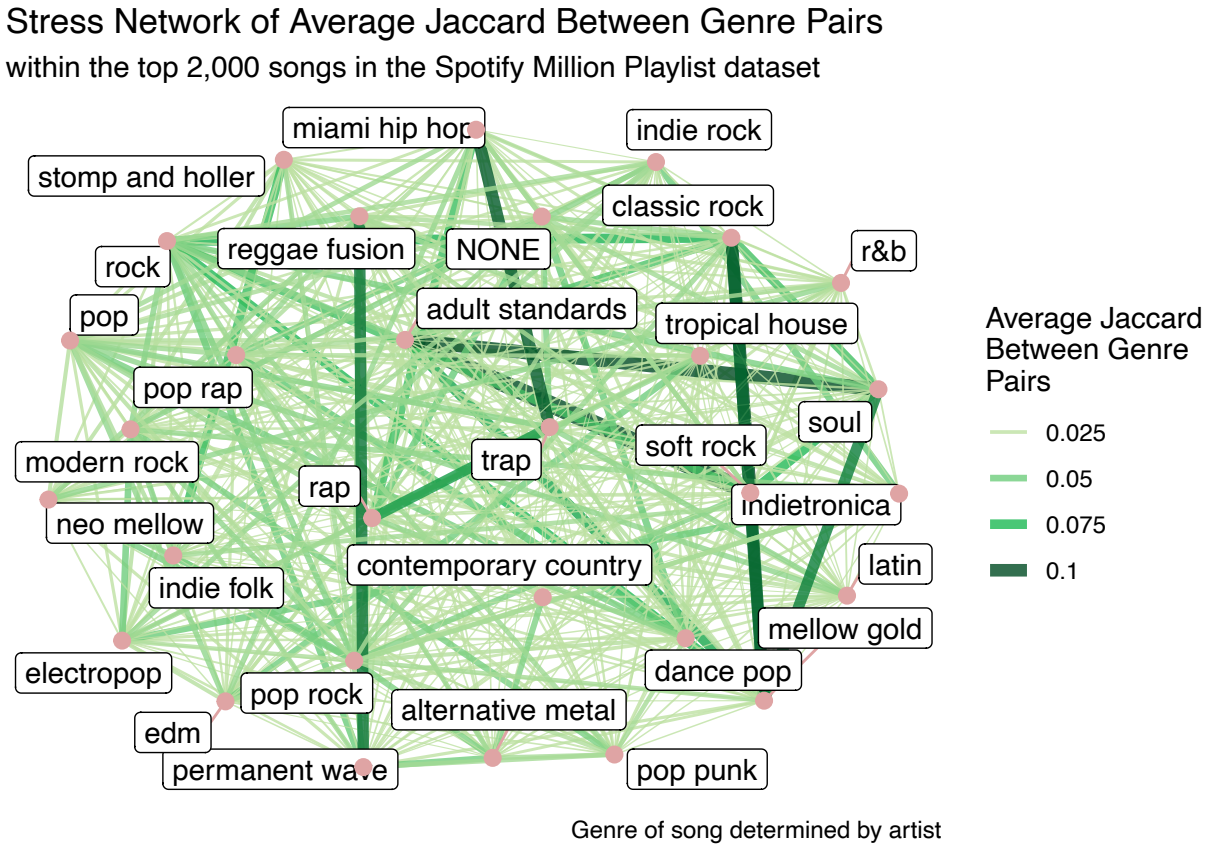
```
ggraph(routes_tidy_j, layout = "linear", circular = TRUE) +
  geom_edge_link(aes(width = aj,
                     color= aj),
                 alpha = 0.8) +
  geom_node_label(aes(label = label), repel = TRUE,
                  segment.color = "#DFA4A4") +
  geom_node_point(color="#DFA4A4", size=2.5) +
  theme_graph() +
  scale_edge_color_gradientn(
    colors = c("#bee1a3", "#6ECD7C", "#1db954", "#004b23"),
    limits = c(.025, .1),
    breaks=c(seq(0, .1, by=.025)),
```

```
  labels=c(seq(0, .1, by=.025)),
  oob = scales::squish,
  name = "Average Jaccard\nBetween Genre\nPairs",
  na.value="grey",
  guide="legend") +
scale_edge_width_continuous(name="Average Jaccard\nBetween Genre\nPairs",
                      breaks=c(seq(0, .1, by=.025)),
                      labels=c(seq(0, .1, by=.025)),
                      range = c(0.15, 3)) +
guides(color=guide_legend(),
       width=guide_legend()) +
theme_bw() +
theme(axis.title=element_blank(),
      axis.text=element_blank(),
      axis.ticks=element_blank(),
      panel.grid = element_blank(),
      panel.border = element_blank(),
      panel.background = element_blank()) +
labs(title="Linear Circle Network of Average Jaccard Between Genre Pairs",
     subtitle="within the top 2,000 songs in the Spotify Million Playlist dataset",
     caption = "Genre of song determined by artist")
```



Stress Network Graph: Jaccard

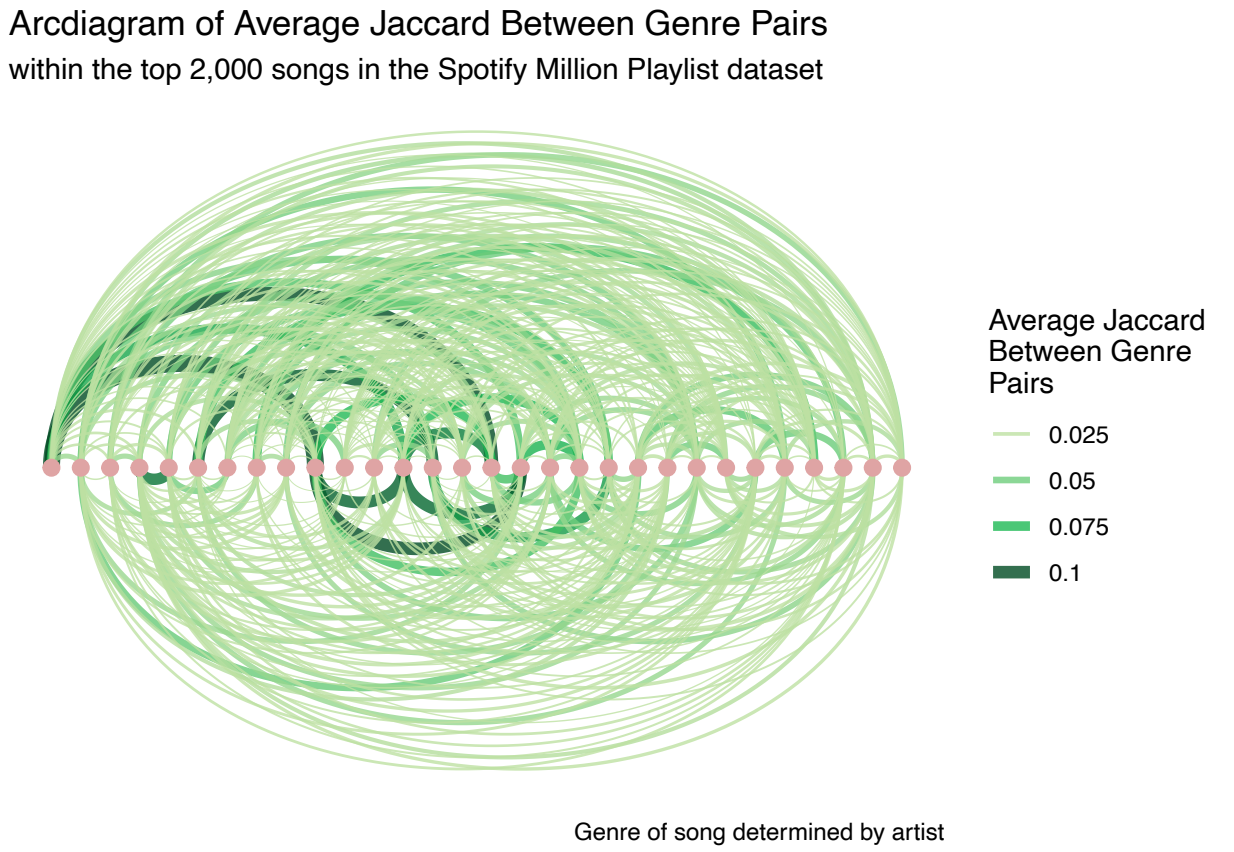
```
ggraph(routes_tidy_j, layout = "stress") +
  geom_edge_link(aes(width = aj,
                     color= aj),
                 alpha = 0.8) +
  geom_node_label(aes(label = label), repel = TRUE,
                  segment.color = "#DFA4A4") +
  geom_node_point(color="#DFA4A4", size=2.5) +
  theme_graph() +
  scale_edge_color_gradientn(
    colors = c("#bee1a3", "#6ECD7C", "#1db954", "#004b23"),
    limits = c(.025, .1),
    breaks=c(seq(0, .1, by=.025)),
    labels=c(seq(0, .1, by=.025)),
    oob = scales::squish,
    name = "Average Jaccard\nBetween Genre\nPairs",
    na.value="grey",
    guide="legend") +
  scale_edge_width_continuous(name="Average Jaccard\nBetween Genre\nPairs",
                        breaks=c(seq(0, .1, by=.025)),
                        labels=c(seq(0, .1, by=.025)),
                        range = c(0.15, 3)) +
  guides(color=guide_legend(),
         width=guide_legend()) +
  theme_bw() +
  theme(axis.title=element_blank(),
        axis.text=element_blank(),
        axis.ticks=element_blank(),
        panel.grid = element_blank(),
        panel.border = element_blank(),
        panel.background = element_blank()) +
  labs(title="Stress Network of Average Jaccard Between Genre Pairs",
       subtitle="within the top 2,000 songs in the Spotify Million Playlist dataset",
       caption = "Genre of song determined by artist")
```



Arcdiagram: Jaccard

```
ggraph(routes_tidy_j, layout = "linear") +
  geom_edge_arc(aes(width = aj,
                    color= aj),
                alpha = 0.8) +
  geom_node_point(color="#DFA4A4", size=2.5) +
  theme_graph() +
    scale_edge_color_gradientn(
    colors = c("#bee1a3", "#6ECD7C", "#1db954", "#004b23"),
    limits = c(.025, .1),
    breaks=c(seq(0, .1, by=.025)),
    labels=c(seq(0, .1, by=.025)),
    oob = scales::squish,
    name = "Average Jaccard\nBetween Genre\nPairs",
    na.value="grey",
    guide="legend") +
  scale_edge_width_continuous(name="Average Jaccard\nBetween Genre\nPairs",
                        breaks=c(seq(0, .1, by=.025)),
                        labels=c(seq(0, .1, by=.025)),
                        range = c(0.15, 3)) +
  guides(color=guide_legend(),
         width=guide_legend()) +
  theme_bw() +
```

```
  theme(axis.title=element_blank(),
        axis.text=element_blank(),
        axis.ticks=element_blank(),
        panel.grid = element_blank(),
        panel.border = element_blank(),
        panel.background = element_blank()) +
  labs(title="Arcdiagram of Average Jaccard Between Genre Pairs",
       subtitle="within the top 2,000 songs in the Spotify Million Playlist dataset",
       caption = "Genre of song determined by artist")
```



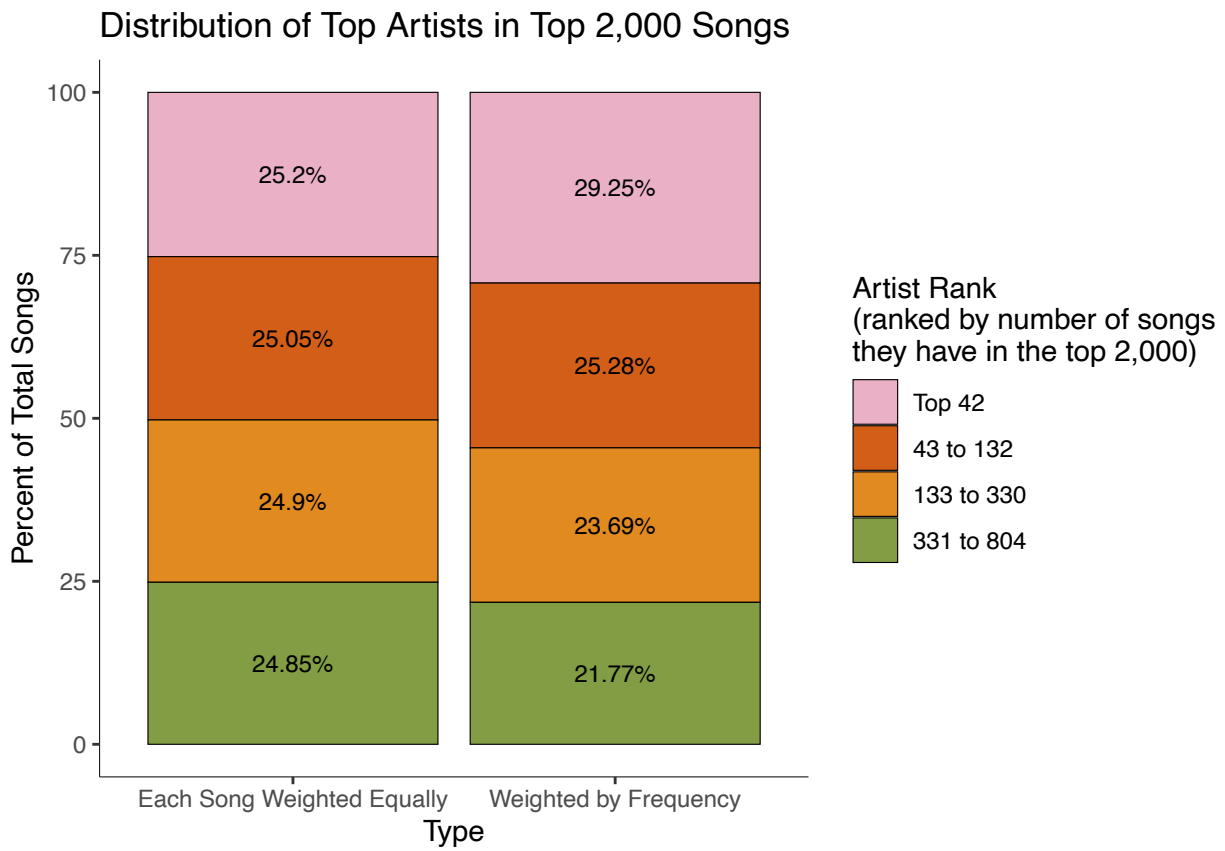
The 42 artists with the most songs included were responsible for 25% of all the top 2,000 songs, and accountated for 29.25% of the songs in the pairs, showing that these more popular artists are also more commonly listened to and their music is added to more playlists than less popular artists, which makes sense.

Stacked Percentage Barplot of Top Artist Frequency

```
#frequency of genres within entire dataframe that has been subsetted to the 2k songs
overall_frequency <- uri_vec2 %>%
  group_by(uri_vec2) %>%
  dplyr::summarise(count=n())

overall_frequency$uri_vec2 <- gsub("spotify:track:", "", overall_frequency$uri_vec2)
```

```
overall_frequency <- left_join(overall_frequency, track_info, by =c("uri_vec2" ="all_songs"))

overall_frequency <- overall_frequency %>%
  group_by(artist_name) %>%
  dplyr::summarise(overall = sum(count)) %>%
  arrange(desc(overall))

#frequency of genres within 2k songs
unique_artist_f <- track_info %>%
  group_by(artist_name) %>%
  dplyr::summarise(unique = n()) %>%
  arrange(desc(unique))

#merging them
artist_freq <- merge(overall_frequency, unique_artist_f)

#finding the top genres to make the rest be "other" for the viz
artists_1_50 <- artist_freq %>%
  dplyr::mutate(overall_perc = (overall/1954359)*100,
                unique_perc = (unique/2000)*100) %>%
  arrange(desc(unique_perc)) %>%
  slice(1:42) %>%
  .$artist_name

artists_51_200 <- artist_freq %>%
  dplyr::mutate(overall_perc = (overall/1954359)*100,
                unique_perc = (unique/2000)*100) %>%
  arrange(desc(unique_perc)) %>%
  slice(43:132) %>%
  .$artist_name

artists_201_500 <- artist_freq %>%
  dplyr::mutate(overall_perc = (overall/1954359)*100,
                unique_perc = (unique/2000)*100) %>%
  arrange(desc(unique_perc)) %>%
  slice(133:330) %>%
  .$artist_name

artists_rest <- artist_freq %>%
  dplyr::mutate(overall_perc = (overall/1954359)*100,
                unique_perc = (unique/2000)*100) %>%
  arrange(desc(unique_perc)) %>%
  slice(331:804) %>%
  .$artist_name


artist_freq <- artist_freq %>%
  dplyr::mutate(
    artist_name = case_when(
      artist_name %in% artists_1_50 ~ "Top 42",
      artist_name %in% artists_51_200 ~ "43 to 132",
      artist_name %in% artists_201_500 ~ "133 to 330",
      artist_name %in% artists_rest ~ "331 to 804",
```

```
      TRUE                        ~  "other"
    )
  )

#some more cleanup, reordering factors
artist_freq <- artist_freq %>%
  group_by(artist_name) %>%
  dplyr::summarise(once = sum(unique)/2000*100,
                   with_all = sum(overall)/1954359*100) %>%
  arrange(desc(with_all))

artist_freq$artist_name <- reorder(artist_freq$artist_name ,
                                   artist_freq$with_all)
artist_freq$artist_name <- factor(artist_freq$artist_name,
                                  levels=rev(levels(artist_freq$artist_name)))

#plotting!
artist_freq %>%
  reshape2::melt(id.vars = c("artist_name"),
                 variable.name = "Group",
                 value.name = "Value") %>%
  ggplot(aes(x=Group, y=Value, fill=artist_name, label=paste0(round(Value, 2), "%"))) +
  geom_bar(position="stack",
           stat="identity",
           color = "black", size=.2) +
  scale_fill_manual(values=met.brewer("Tara",
                                      4))+
    labs(title = "Distribution of Top Artists in Top 2,000 Songs",
         fill = "Artist Rank\n(ranked by number of songs\nthey have in the top 2,000)",
         x="Type",
         y="Percent of Total Songs") +
    geom_text(size = 3, position = position_stack(vjust = 0.5)) +
    scale_x_discrete(labels = c("Each Song Weighted Equally", "Weighted by Frequency")) +
    theme_bw() +
    theme(panel.grid.major = element_blank(),
          axis.line = element_line(colour = "black", size=.1),
          panel.grid.minor = element_blank(),
          panel.border = element_blank(),
          panel.background = element_blank())
```



Models (first set of playlists)

When our data processing had finished, we had 2000 choose 2 or 1,999,000 rows. This number is the result of needing every pair to be paired with every other song once, but not caring about the order in which the pairs are represented.

In order to assess our features' fit to the data, we used a variety of regression algorithms (linear, lasso, PCR, and GBM). Linear regression was used as the simplest algorithm to simply see the fit of the data with a general line. Lasso was used to see if certain features could be essentially discarded via regularization, as that could potentially improve our prediction quality on data from different playlist groups. PCR was utilized as some of our predictors (valance/danceability, energy/loudness, etc) had substantial correlations (though, of course, a principal components analysis is perhaps not as necessary as in HW3 given that we had many more songs than features). GBM was utilized as earlier results from other regression had much better results for smaller values than larger ones, so we believed that a boosted method would correct for that, given its focus on addressing poorly-modeled data within each tree.

Each algorithm analyzed the relationship between the outcome variable and the absolute difference between the features of the pair of songs. We thought that the absolute difference would be the best way to analyze the pairs of features, as this would allow us to get at the difference between the songs — i.e. if one song had much more energy than another this would show up as a large absolute difference. Before calculating the absolute difference, we scaled each of the features as some are on different scales (loudness, for example, is on a decibel scale while liveness is the probability that a song was performed live). This is necessary for things like lasso, and it's good practice to have the coefficients be comparable across algorithms when the regression type permits.

Making the test and training data

```
set.seed(1)
#Make test and train sets:
dt = sort(sample(nrow(merge_frame),
                 nrow(merge_frame)*.7))
train<-merge_frame[dt,]
test<-merge_frame[-dt,]
test_dat <- test[,7:18]
```

Baseline Model From Percentiles

```
set.seed(1)
baseline <- quantile(train$jaccard, probs = seq(.1, .9, by = .1))
baseline[1] <- 0.0000001
y_predicted_baseline <- sample(baseline, length(test$jaccard), replace = TRUE)

rmse_baseline <- rmse(as.numeric(test$jaccard), y_predicted_baseline)
rmse_baseline
```

```
## [1] 0.02341966
```

```
smape_baseline<- mlr3measures::smape(as.numeric(test$jaccard), y_predicted_baseline, na_value = 0)
smape_baseline
```

```
## [1] 1.228602
```

We also created a baseline score to which we could compare our models. This took the deciles (up to the 90% percentile) of the full set of Jaccard Indices and assigned them randomly over the test data with equal probability. The baseline's values were between 0 and 0.03 — reflecting that the vast majority of our data was present in this range even as the Jaccard Index did range up past 0.5 for a small number of values.

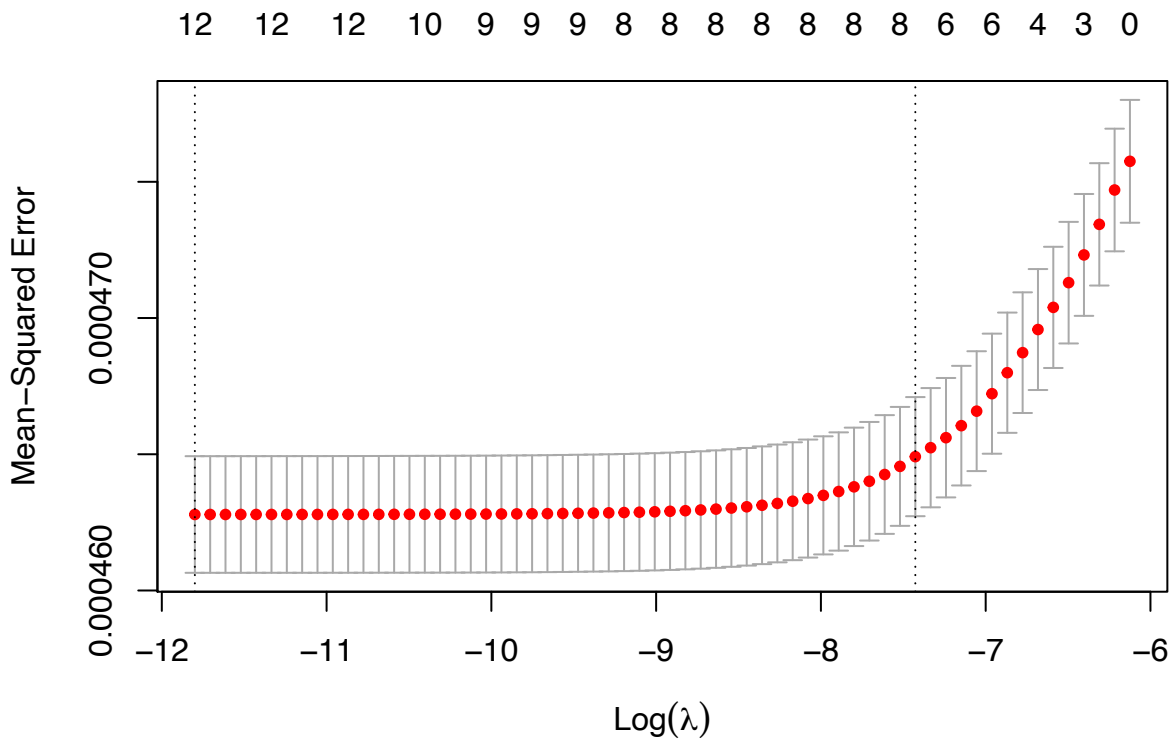
One change was made to the first value in this vector — it was changed from 0 to 0.0000001 as values of (or near, potentially due to some type of bit limitation for each number?) 0 produce exceptions in the sMAPE (one of our error metrics that will be explained below) formula under certain conditions. This change would not have a meaningful actual impact on the metric calculation — only its ability to run.

Lasso

```
y_lasso <- train$jaccard
x_lasso <- data.matrix(train[,7:18])

#cv_model <- cv.glmnet(x_lasso, y_lasso, alpha = 1)
##saving the output so we don't need to run again
#saveRDS(cv_model, file = "/Users/nickicamberg/Desktop/stat3106/cv_model.rds")
cv_model <- readRDS("/Users/nickicamberg/Desktop/stat3106/cv_model.rds")

#find optimal lambda value that minimizes test MSE
best_lambda <- cv_model$lambda.min
best_lambda
```

```
## [1] 7.508089e-06
```

```
#produce plot of test MSE by lambda value
plot(cv_model)
```




```
best_model_lasso <- glmnet(x_lasso, y_lasso,
                           alpha = 1, lambda = best_lambda)
coef(best_model_lasso)
```

```
## 13 x 1 sparse Matrix of class "dgCMatrix"
##                              s0
## (Intercept)       1.973069e-02
## danceability     -1.423719e-02
## energy           -6.928535e-03
## key               8.866356e-06
## loudness         -3.717174e-04
## mode             -1.167251e-03
## speechiness      -9.907492e-03
## acousticness     -4.197516e-03
## instrumentalness -5.332264e-03
## liveness         -1.540108e-04
## valence          -5.731610e-03
## tempo             3.574583e-06
## time_signature    7.308435e-05
```

```
y_predicted_lasso <- predict(best_model_lasso,
                             s = best_lambda,
                             newx = data.matrix(test_dat))

rmse_lasso <- rmse(as.numeric(test$jaccard),
                   as.numeric(y_predicted_lasso))

rmse_lasso
```

```
## [1] 0.02124369
```

```
smape_lasso <- mlr3measures::smape(as.numeric(test$jaccard),
                                   as.numeric(y_predicted_lasso),
                                   na_value = 0)
smape_lasso
```

```
## [1] 1.102694
```

In terms of the interpretation of the coefficients from the algorithms, it seems that they are indeed in the direction that we predicted. In most cases, the coefficients of the features were negative — signifying that a higher absolute difference leads to a lesser Jaccard Index and vice versa. Tempo, key, and time signature have positive coefficients in some of the models, but these are very close (and much closer than the negative coefficients) to zero so they could be potentially interpreted as noise in the data. It's unclear why these particular features have coefficients that are oriented in different directions than the others. Time signature and tempo describe fairly similar aspects of the song, so users could be perceiving them in the same way. Key, on the other hand, simply represents whether the key the track is in. Users may find the differences between different keys to be too small to notice. Moreover, it may be possible that key is a less meaningful feature to distinguish between songs because it doesn't neatly fit into a numerical frame — i.e. the difference between key of C and E might not sound twice as big as the difference between C and D when put into music.

Lasso didn't discard any of the features, though it did shrink the three positive coefficients as just described down to near 0.

Linear

```
linear <- lm(train$jaccard ~ ., data = as.data.frame(scale(train[,7:18])))
summary(linear)
```

```
##
## Call:
## lm(formula = train$jaccard ~ ., data = as.data.frame(scale(train[,
##     7:18])))
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.01948 -0.00981 -0.00605  0.00040  0.74943
##
## Coefficients:
##                    Estimate Std. Error t value Pr(>|t|)
```

```
## (Intercept)       1.101e-02  1.819e-05 605.254  < 2e-16 ***
## danceability     -1.786e-03  1.841e-05 -96.984  < 2e-16 ***
## energy           -1.068e-03  2.230e-05 -47.875  < 2e-16 ***
## key               3.507e-05  1.819e-05   1.928   0.0539 .
## loudness         -9.409e-04  2.123e-05 -44.325  < 2e-16 ***
## mode             -5.878e-04  1.822e-05 -32.269  < 2e-16 ***
## speechiness      -1.051e-03  1.842e-05 -57.075  < 2e-16 ***
## acousticness     -9.689e-04  2.016e-05 -48.061  < 2e-16 ***
## instrumentalness -5.442e-04  1.831e-05 -29.718  < 2e-16 ***
## liveness         -2.890e-05  1.827e-05  -1.582   0.1135
## valence          -1.076e-03  1.838e-05 -58.557  < 2e-16 ***
## tempo             9.456e-05  1.838e-05   5.145 2.67e-07 ***
## time_signature    3.784e-05  1.848e-05   2.048   0.0406 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.02151 on 1399287 degrees of freedom
## Multiple R-squared:  0.02729,    Adjusted R-squared:  0.02729
## F-statistic:  3272 on 12 and 1399287 DF,  p-value: < 2.2e-16
```

```
y_predicted_linear <- predict(linear, newdata = test_dat)

rmse_linear <- rmse(as.numeric(test$jaccard), as.numeric(y_predicted_linear))
rmse_linear
```

```
## [1] 0.02155994
```

```
smape_linear <- mlr3measures::smape(as.numeric(test$jaccard),
                                    as.numeric(y_predicted_linear),
                                    na_value = 0)

smape_linear
```

```
## [1] 1.095229
```

Linear regression reported nearly every feature as significant, though this could just be an artifact of the fact that our n being so big reduced the standard errors to such a small number as to inflate t-values. Substantive significance, on the other hand, is not present. A one standard deviation jump in energy, for instance, would produce only a 0.001 Jaccard Index difference. This presents a critical problem that will be explored further later — there's simply not enough difference in the features to account for the range of Jaccard Index.

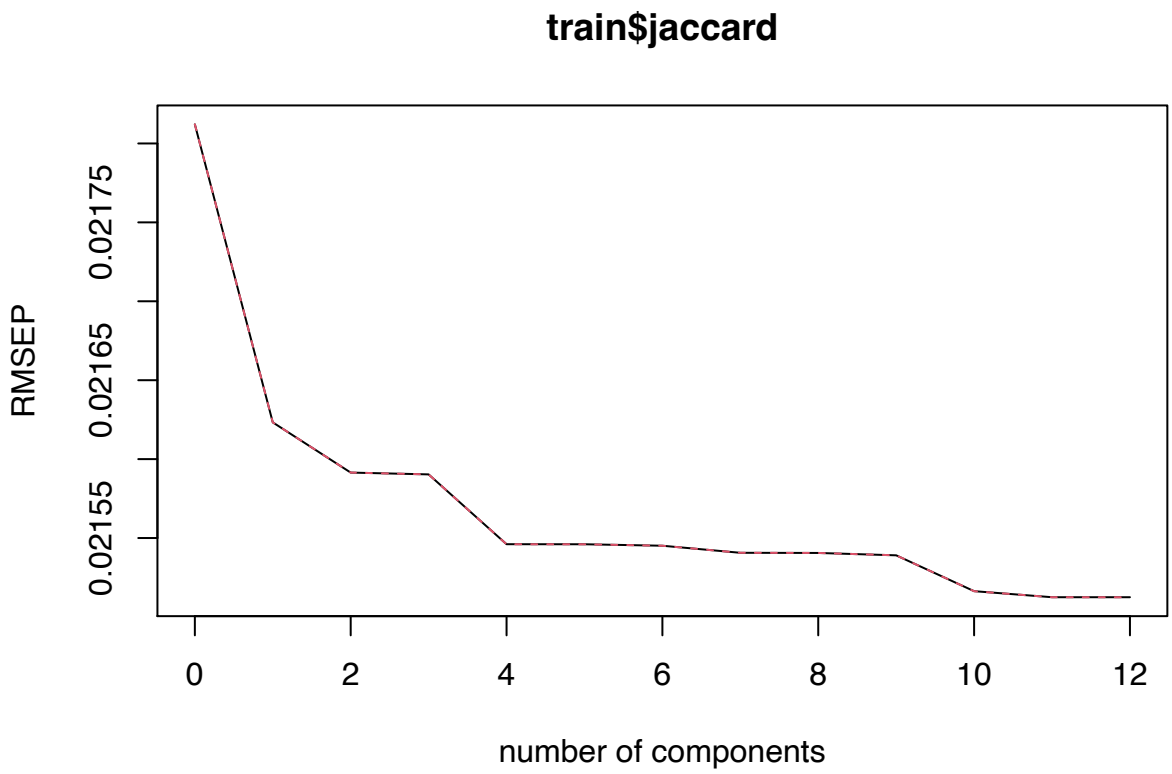
PCR

```
pcr_model <- pcr(train$jaccard~., data = train[,7:18],
                 scale = TRUE, validation = "CV")
#saving the output so we don't need to run again
saveRDS(pcr_model, file = "/Users/nickicamberg/Desktop/stat3106/pcr_model.rds")
```

```r
pcr_model <- readRDS("/Users/nickicamberg/Desktop/stat3106/pcr_model.rds")
summary(pcr_model)
```

```
## Data:    X dimension: 1399300 12
##  Y dimension: 1399300 1
## Fit method: svdpc
## Number of components considered: 12
##
## VALIDATION: RMSEP
## Cross-validated using 10 random segments.
##        (Intercept)  1 comps  2 comps  3 comps  4 comps  5 comps  6 comps
## CV         0.02181  0.02162  0.02159  0.02159  0.02155  0.02155  0.02155
## adjCV      0.02181  0.02162  0.02159  0.02159  0.02155  0.02155  0.02155
##        7 comps  8 comps  9 comps  10 comps  11 comps  12 comps
## CV     0.02154  0.02154  0.02154   0.02152   0.02151   0.02151
## adjCV  0.02154  0.02154  0.02154   0.02152   0.02151   0.02151
##
## TRAINING: % variance explained
##                1 comps  2 comps  3 comps  4 comps  5 comps  6 comps  7 comps
## X               15.767   25.838   34.692   43.318   51.730   59.972   68.004
## train$jaccard    1.724    2.013    2.024    2.425    2.425    2.434    2.474
##                8 comps  9 comps  10 comps  11 comps  12 comps
## X               75.886   83.203    90.340    96.094   100.000
## train$jaccard    2.475    2.489     2.695     2.729     2.729
```

```r
validationplot(pcr_model)
```




```
y_predicted_pcr <- predict(pcr_model, test_dat, ncomp = 4)

rmse_pcr <- rmse(as.numeric(test$jaccard), as.numeric(y_predicted_pcr))
rmse_pcr
```

```
## [1] 0.0212778
```

```
smape_pcr <- mlr3measures::smape(as.numeric(test$jaccard),
                                 as.numeric(y_predicted_pcr),
                                 na_value = 0)

smape_pcr
```

```
## [1] 1.104658
```

PCA regression, on the other hand, found that roughly four components had the right balance between including too many components and minimizing error. This, namely, PCR being useful in shrinking the number of components down, is not unexpected, as we determined that a number of the features had substantial correlations and collinearity.

GBM

```
gbm.fit <- gbm(
  formula = train$jaccard ~ .,
  distribution = "gaussian",
  data = as.data.frame(scale(train[,7:18])),
  interaction.depth = 3,
  shrinkage = 0.1,
  cv.folds = 5,
  n.cores = NULL, # will use all cores by default
  verbose = FALSE
  )

##saving the output so we don't need to run again
saveRDS(gbm.fit, file = "/Users/nickicamberg/Desktop/stat3106/gbm_fit.rds")
```

```
gbm.fit <- readRDS("/Users/nickicamberg/Desktop/stat3106/gbm_fit.rds")
y_predicted_gbm <- predict(gbm.fit,
                           test,
                           n.trees = gbm.fit$n.trees)

rmse_gbm <- rmse(as.numeric(test$jaccard),
                 as.numeric(y_predicted_gbm))
rmse_gbm
```

```
## [1] 0.02184009
```

```
smape_gbm <- mlr3measures::smape(as.numeric(test$jaccard),
                                 as.numeric(y_predicted_gbm),
                                 na_value = 0)

smape_gbm
```

```
## [1] 1.03109
```

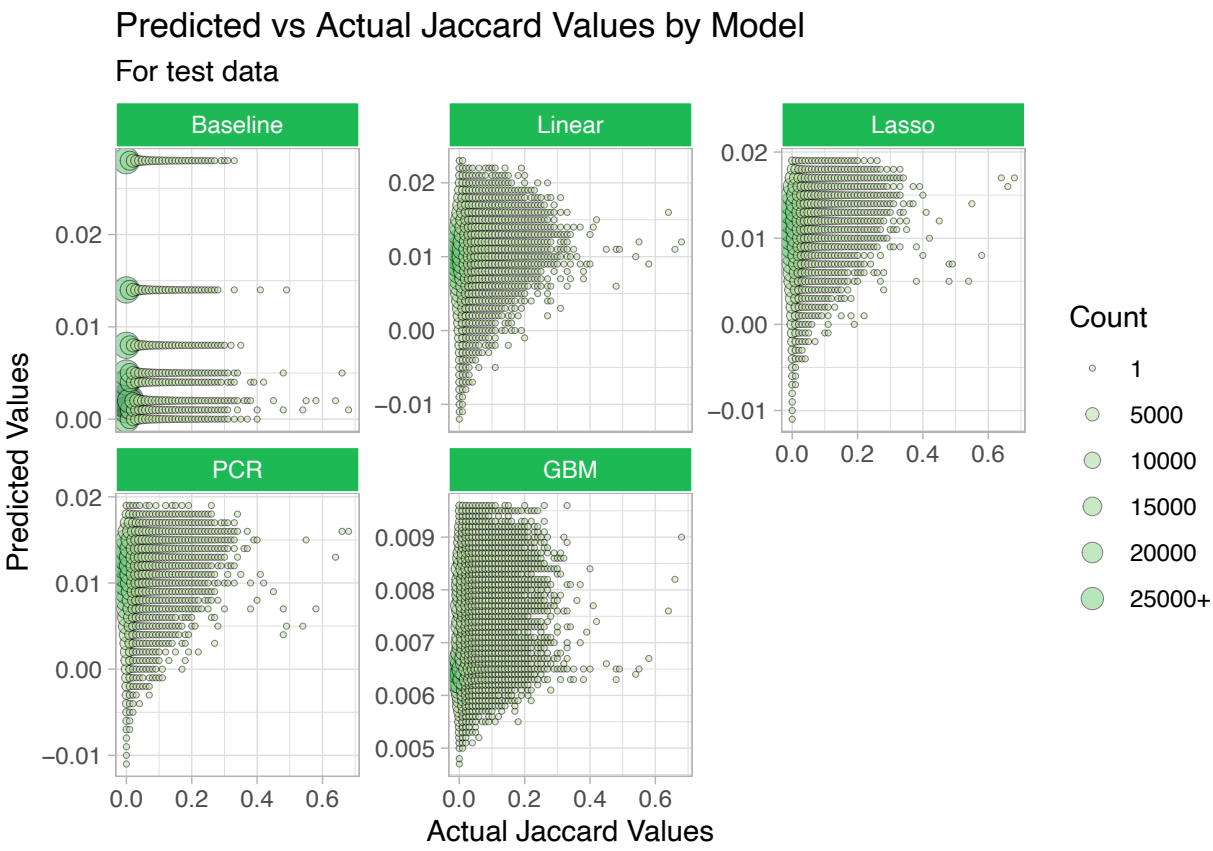
We assessed these algorithms quantitatively via two metrics: RMSE and sMAPE. RMSE is used as the default for continuous regression problems for a good reason — it provides an accurate representation of how far our predictions were from the actual values. However, given that so many of the Jaccard Index values were small (as we outlined in the percentile section of the exploratory data analysis), the RMSE values are actually somewhat misleading. While an RMSE in the hundredths place seems tiny, that's only because the values we're analyzing are similarly small. An RMSE value of 0.03 for predicted values around 100 is much better than the same RMSE for predicted values around 0.1.

Because of this, sMAPE stands out as a better choice. This is a variation of MAPE and we use it instead because MAPE returns infinite or NaN values when the actual value is equal to 0. Unlike RMSE, which simply focuses on the difference between actual and predicted values, sMAPE divides that difference by the sum of the two values (divided by two), which produces a % error. More generally, this error is relative rather than absolute as RMSE is. This fixes the problem identified in the last sentence of the previous paragraph — while those values would produce identical RMSEs, the sMAPEs would instead be different by several orders of magnitude.

For each algorithm, we performed two different tests with the two quantitative metrics. First, we did the normal test/train split with the metrics being assessed on the test set. Validation was either not needed for algorithms like linear regression that don't have hyperparameters or handled by the function itself in the case of the algorithms that do have hyperparameters (lasso, for example).

Jaccard Plots for Each Model

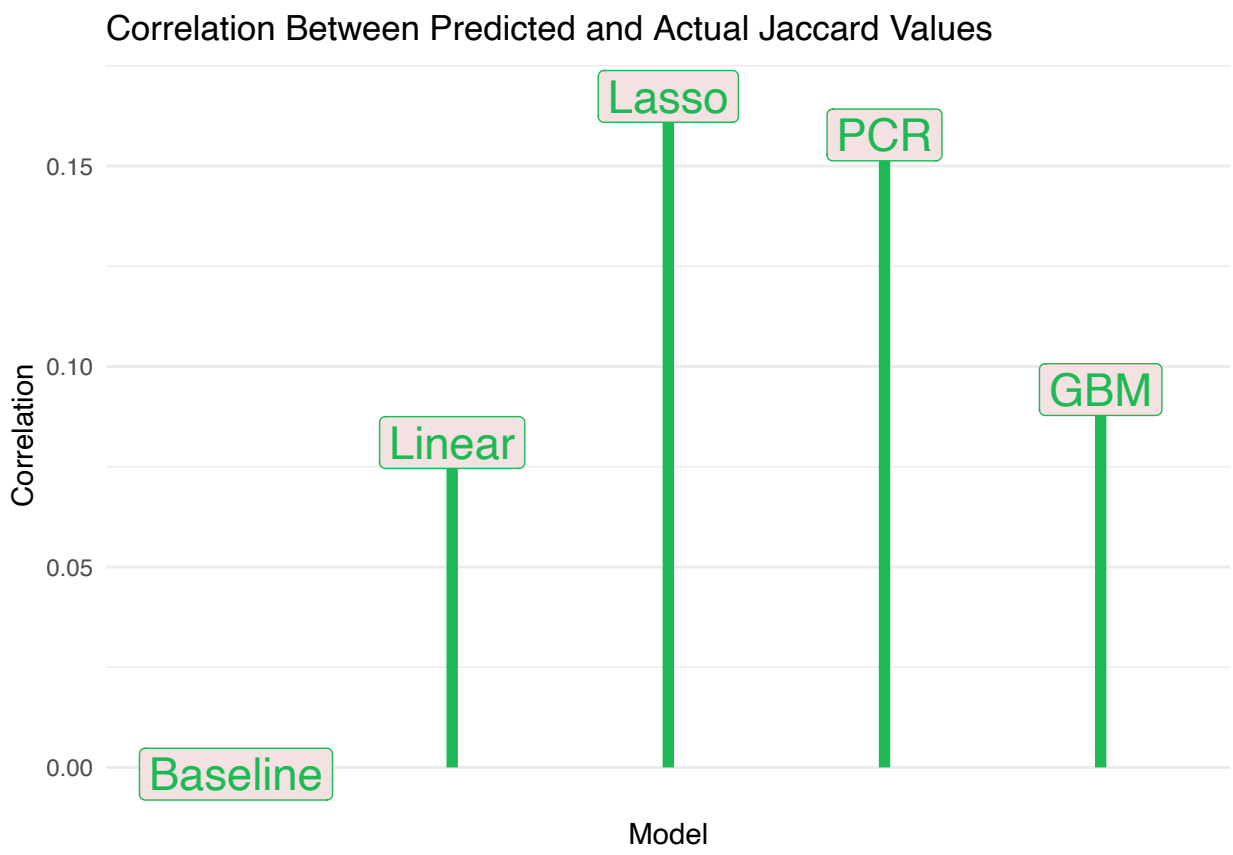
```
jacc1 <- data.frame(test$jaccard, as.numeric(y_predicted_baseline),
                    as.numeric(y_predicted_linear), as.numeric(y_predicted_lasso),
                    as.numeric(y_predicted_pcr), as.numeric(y_predicted_gbm))
names(jacc1) <- c("jaccard", "Baseline", "Linear", "Lasso", "PCR", "GBM")

jacc1 %>%
  reshape2::melt(id.vars = c("jaccard"),
                 variable.name = "model",
                 value.name = "predicted") %>%
  dplyr::mutate(jaccard = round(jaccard, 2),
                predicted = ifelse(model=="GBM",
                                   round(predicted, 4),
                                   round(predicted, 3))) %>%
  group_by(jaccard, predicted, model) %>%
  dplyr::summarise(count=n()) %>%
  ggplot(aes(x=jaccard, y=predicted, group=model, size=count)) +
  geom_point(aes(fill=count, size=count),
             color = "black", pch=21, alpha=.5, stroke=.1) +
  labs(x="Actual Jaccard Values",
       y="Predicted Values") +
  facet_wrap(~model, scales = "free_y") +
  scale_fill_gradientn(
    colors = c("#bee1a3", "#6ECD7C", "#1db954", "#004b23"),
    breaks=c(1, seq(5000, 25000, by=5000)),
    labels=c(1, seq(5000, 20000, by=5000), "25000+"),
    oob = scales::squish,
    name = "Count",
    na.value="grey",
    guide="legend") +
  scale_size_continuous(name="Count",
                        breaks=c(1, seq(5000, 25000, by=5000)),
                        labels=c(1, seq(5000, 20000, by=5000), "25000+")) +
  guides(fill= guide_legend(),
         size=guide_legend()) +
  labs(title="Predicted vs Actual Jaccard Values by Model",
       subtitle="For test data") +
  theme_light() +
  theme(strip.background = element_rect(color="white", fill = "#1db954"))
```



Correlation Lollipop Graph

```
jacc1 %>%
  reshape2::melt(id.vars = c("jaccard"),
                 variable.name = "model",
                 value.name = "predicted") %>%
  group_by(model) %>%
  dplyr::summarise(correlation = cor(jaccard, predicted)) %>%
  ggplot(aes(x=model, y=correlation)) +
  geom_point() +
  geom_segment(aes(x=model, xend=model, y=0, yend=correlation),
               color="#1DB954", size =2) +
  geom_label(aes(label = model),
             color = "#1DB954", fill = "#F4E1E1",
             size=6) +
  labs(x="Model",
       y="Correlation",
       title="Correlation Between Predicted and Actual Jaccard Values")+
  theme_bw()+
  theme(axis.text.x=element_blank(),
        axis.ticks=element_blank(),
        panel.grid.major.x = element_blank(),
        panel.grid.minor.x = element_blank(),
        panel.border = element_blank(),
```

```
        panel.background = element_blank())
```



Correlation and Jaccard predicted vs actual plots will be further discussed below.

Reading in and Cleaning Data

To quantitatively answer the question of whether the results we've observed are chance or are a real pattern, we also tested our results on an entire separate group of playlists. The structure of the dataset is such that the million playlists are divided up into 1000 files of 1000 playlists each. With our first training and test sets being composed of data from 100 of the 1000 files (100K playlists overall), we then took 20 more files (20K playlists) and performed the same operations on them as with the previous data before testing the models that had been trained on the previous data on the new playlists. Our reasoning behind this was that the first group of playlists might have been biased (if Spotify constructed the files in a certain way) towards certain types of music, providing skewed results for our model and meaning that it would perform worse in the real world over a more complete dataset. By testing the same models on entirely different playlist data, we can get a sense of whether this is the case.

Second set of playlists, playlists 810000-830999

```
pls2 <- lapply(list.files(path="/Users/nickicamberg/Desktop/stat3106/spot_playlists2/",
                          pattern=NULL, all.files=FALSE,
                          full.names=TRUE), read_json)
```

```
pls2 <- lapply(pls2, `[[`, 2)


pls2 <- unlist(pls2, recursive = FALSE, use.names = TRUE)

#getting a list of all the tracks in all the playlists
track_list2 <- vector(mode = "list", length = length(pls2))
for(i in 1:length(pls2)){
  track_list2[[i]] <- pls2[[i]]$tracks
}

#getting how many songs are in each playlist to split them later
length_playlists2 <- c()
for(i in 1:length(track_list2)){
  length_playlists2[i] <- length(track_list2[[i]])
}

playlist_numbers2 <- rep(1:length(pls2), length_playlists2)

#melting the tracklist
tmelt2<- unlist(track_list2, recursive = FALSE)

#creating an empty vector that holds all the URIs
uri_vec_pl2 <- vector(mode = "list", length = length(tmelt2))

#creating a single vector that has all the URIs
for(i in 1:length(tmelt2)){
  uri_vec_pl2[[i]] <- tmelt2[[i]]$track_uri}

uri_vec_pl2 <- unlist(uri_vec_pl2)

#getting a table of the 2000 songs that appear the most
#there are 681805 songs
top_uri2 <- sort(table(uri_vec_pl2),decreasing=TRUE)[1:2000]

#vector of the song uris that appear the most
top_uris2 <- names(top_uri2)

#making a vector of only the top 2k songs, rest are NA
uri_vec_pl2_2 <- ifelse(uri_vec_pl2 %in% top_uris2, uri_vec_pl2, NA)

#splitting the vector of tracks by playlist number
tracks_split2 <- split(uri_vec_pl2_2, playlist_numbers2)

#making that into a data frame
uri_frame2 <- t(stri_list2matrix(tracks_split2))
uri_frame2 <- as.data.frame(uri_frame2)

#saving the uri frame and a few other things
write.csv(uri_frame2,"/Users/nickicamberg/Desktop/stat3106/uri_frame2.csv")
saveRDS(top_uri2, file = "/Users/nickicamberg/Desktop/stat3106/top_uri2.rds")
saveRDS(uri_vec_pl2_2, file = "/Users/nickicamberg/Desktop/stat3106/uri_vec_pl2_2.rds")
```

```
#reading them back in
#uri_frame2 <- read_csv("/Users/nickicamberg/Desktop/stat3106/uri_frame2.csv")
top_uri2 <- readRDS("/Users/nickicamberg/Desktop/stat3106/top_uri.rds")
top_uris2 <- names(top_uri2)
```

Making a df of all possible pairs of songs in the second set of playlists, and get how many times they actually occur

```
#Make dataframe where each top song is a row, other songs are columns,
#and track how many times the column song is in a playlist the row song is in:
duplicate2 <- uri_frame2 %>%
  dplyr::mutate(name = row_number()) %>%
  pivot_longer(!name, names_to = 'variable',
               values_to = 'element') %>%
  drop_na()

duplicate2 <- duplicate2 %>%
  dplyr::select(-variable) %>%
  group_by(element) %>%
  dplyr::mutate(numdups = n()) %>%
  filter(numdups > 1) %>%
  dplyr::select(-numdups)

duplicate2 <-setDT(duplicate2)

pair_table2 <- duplicate2[duplicate2, on = c('name'), allow.cartesian = TRUE][
  element<i.element, .N, .(pair = paste0(element, " ", i.element))]

pair_split2 <- as.data.frame(str_split_fixed(pair_table2$pair, " ", 2))

#Get all possible pairs:
all_pairs2 <- as.data.frame(t(combn(top_uris2, 2, FUN = NULL, simplify = TRUE)))

#Split the possible pairs to compare them with the pair_split and all the ones we're missing:

#Get the first song in alphabetical order to be first in the pair
#and do the same with the second to make sure we can match by comb:

all_pairs2$comb <- ifelse(all_pairs2$V1 > all_pairs2$V2,
                          paste0(all_pairs2$V1, "-", all_pairs2$V2),
                          paste0(all_pairs2$V2, "-", all_pairs2$V1))


pair_table2$pair1 <- pair_split2$V1
pair_table2$pair2 <- pair_split2$V2

pair_table2$comb <- ifelse(pair_table2$pair1 > pair_table2$pair2,
                           paste0(pair_table2$pair1, "-", pair_table2$pair2),
                           paste0(pair_table2$pair2, "-", pair_table2$pair1))

pair_table2 <- bind_rows(pair_table2,
                         anti_join(all_pairs2,
```

```
                                  pair_table2,
                                  by="comb"))

#Fill in missing values for pairs without occurrences:
pair_table2$pair1 <- ifelse(is.na(pair_table2$pair1),
                            pair_table2$V1,
                            pair_table2$pair1)

pair_table2$pair2 <- ifelse(is.na(pair_table2$pair2),
                            pair_table2$V2,
                            pair_table2$pair2)

pair_table2$N <- ifelse(is.na(pair_table2$N),
                        0,
                        pair_table2$N)

pair_table2 <- subset(pair_table2,
                      select=-c(pair, comb, V1, V2))

occurences2 <-as.data.frame(table(unlist(uri_frame2)))

pair_table2 <- merge(pair_table2, occurences2,
                     by.x = "pair1", by.y = "Var1")
pair_table2 <- merge(pair_table2, occurences2,
                     by.x = "pair2", by.y = "Var1")


pair_table2$jaccard <- (pair_table2$N)/(pair_table2$Freq.x +
                                        pair_table2$Freq.y -
                                        pair_table2$N)

pair_table2$pair1 <- gsub("spotify:track:", "", pair_table2$pair1)
pair_table2$pair2 <- gsub("spotify:track:", "", pair_table2$pair2)

saveRDS(pair_table2, file = "/Users/nickicamberg/Desktop/stat3106/pair_table2.rds")
```

```
#reading it back in
pair_table2 <- readRDS("/Users/nickicamberg/Desktop/stat3106/pair_table2.rds")
```

Getting the audio features for all of the 2000 top songs in the second set of playlists

```
#Get audio features for all of the 2000 top songs

all_songs2 <- unique(c(unique(pair_table2$pair1),
                       unique(pair_table2$pair2)))

#making an empty dataframe to store the song features
all_songs_data2<- data.frame(matrix(ncol = 18,
                                    nrow = length(all_songs2)))

#rename the columns with the feature names
```

```
names(all_songs_data2) <- names(get_track_audio_features(all_songs2[1],
                                          authorization = get_spotify_access_token()))

#running a loop to get the features
for(i in 1:length(all_songs2)){
  all_songs_data2[i,] <- get_track_audio_features(all_songs2[i],
                                          authorization = get_spotify_access_token())
}

#saving it so we don't need to run it again
write.csv(all_songs_data2,"/Users/nickicamberg/Desktop/stat3106/all_songs_data2.csv",
          row.names = FALSE)
```

```
#reading it back in
all_songs_data2 <- read_csv("/Users/nickicamberg/Desktop/stat3106/all_songs_data2.csv")
```

Merging together the dataframe of pairs and the song features, and finding the difference in each value for each pair

```
#Merge them back with main frame:
merge_frame2 <- merge(pair_table2, all_songs_data2,
                      by.x = "pair1", by.y = "id",
                      all.x = TRUE)
merge_frame2 <- merge(merge_frame2, all_songs_data2,
                      by.x = "pair2", by.y = "id",
                      all.x = TRUE)

#Remove extra columns:
merge_frame2 <- merge_frame2 %>%
  dplyr::select(-contains(c("type", "id", "uri",
                            "track_href", "analysis_url",
                            "duration_ms")))

#Get difference for each feature:
merge_frame2$danceability <- abs(merge_frame2$danceability.x -
                                   merge_frame2$danceability.y)
merge_frame2$energy <- abs(merge_frame2$energy.x - merge_frame2$energy.y)
merge_frame2$key <- abs(merge_frame2$key.x - merge_frame2$key.y)
merge_frame2$loudness <- abs(merge_frame2$loudness.x - merge_frame2$loudness.y)
merge_frame2$mode <- abs(merge_frame2$mode.x - merge_frame2$mode.y)
merge_frame2$speechiness <- abs(merge_frame2$speechiness.x -
                                  merge_frame2$speechiness.y)
merge_frame2$acousticness <- abs(merge_frame2$acousticness.x -
                                   merge_frame2$acousticness.y)
merge_frame2$instrumentalness <- abs(merge_frame2$instrumentalness.x -
                                       merge_frame2$instrumentalness.y)
merge_frame2$liveness <- abs(merge_frame2$liveness.x - merge_frame2$liveness.y)
merge_frame2$valence <- abs(merge_frame2$valence.x - merge_frame2$valence.y)
merge_frame2$tempo <- abs(merge_frame2$tempo.x - merge_frame2$tempo.y)
merge_frame2$time_signature <- abs(merge_frame2$time_signature.x -
                                     merge_frame2$time_signature.y)
```

```r
#Delete individual feature columns:
merge_frame2 <- merge_frame2[,c(1:6, 31:42)]

#saving it so we don't need to run it again
write.csv(merge_frame2,"/Users/nickicamberg/Desktop/stat3106/merge_frame2.csv",
          row.names = FALSE)
```

```r
#reading it back in
merge_frame2 <- read_csv("/Users/nickicamberg/Desktop/stat3106/merge_frame2.csv")
```

Getting the data for each song for year released, artist, genre, etc

```r
#code to get the date of the song and also the
#top genre of the artist and the artist/song name
#use all_songs and all_songs2 so we only pull data
#for songs we don't have data for yet
`%!in%` = Negate(`%in%`)
all_songs <- unique(c(unique(pair_table$pair1),
                      unique(pair_table$pair2)))
all_songs2 <- unique(c(unique(pair_table2$pair1),
                      unique(pair_table2$pair2)))
all_songs_only_in2 <- all_songs2[all_songs2 %!in% all_songs]

song_name <- c()
album_id <- c()
album_date <- c()
artist_id <- c()
artist_name <- c()
artist_genres2 <-  vector(mode = "list", length = length(all_songs_only_in2))


for(i in 1:length(all_songs_only_in2)){
  song_holder <- get_track(all_songs_only_in2[i],
                           authorization = get_spotify_access_token())

  song_name[i] <- song_holder[["name"]]

  album_id[i] <- song_holder[["album"]][["id"]]

  album_holder <- get_album(album_id[i], authorization = get_spotify_access_token())

  album_date[i] <- album_holder[["release_date"]]

  artist_id[i] <- album_holder[["artists"]][["id"]]

  artist_name[i] <- album_holder[["artists"]][["name"]]

  artist_genres2[[i]] <- get_artist(artist_id[i],
                                    authorization = get_spotify_access_token())[["genres"]]

}
```

```
#now finding the most commonly occuring genre for each song based on the frequency of
#genres in the whole df
table_ug2 <- table(unlist(artist_genres))
top_genre2 <- c()

for(i in 1:length(all_songs_only_in2)){
  genre_holder <- unlist(c(artist_genres2[[i]]))
  top_genre2[i] <- ifelse(length(artist_genres2[[i]]) > 0,
                          names(which.max(table_ug2[names(table_ug2) %in% genre_holder])),
                          "NONE")
}

#merging everything into a single dataframe
track_info_only_2 <- as.data.frame(cbind(all_songs_only_in2, song_name, artist_name,
                                  top_genre2, album_date,
                                  artist_id, album_id))

#getting a column for year
track_info_only_2$album_year <- substr(track_info_only_2$album_date, 1, 4)

names(track_info_only_2) <- names(track_info)
#merging all the song data together
track_info_all_songs <- as.data.frame(rbind(track_info, track_info_only_2))
#filtering it so we only get the songs that are in the second set
track_info2 <- track_info_all_songs %>%
  filter(all_songs %in% all_songs2)

#saving it so we don't need to run it again
write.csv(track_info2,"/Users/nickicamberg/Desktop/stat3106/track_info2.csv",
          row.names = FALSE)
```

```
#reading it back in
track_info2 <- read_csv("/Users/nickicamberg/Desktop/stat3106/track_info2.csv")
```

Testing the old models on the new data

Baseline

```
#BASELINE:
y_predicted_baseline2 <- sample(baseline,
                                length(merge_frame2$jaccard),
                                replace = TRUE)

rmse_baseline_2 <- rmse(as.numeric(merge_frame2$jaccard),
                        y_predicted_baseline2)
rmse_baseline_2
```

```
## [1] 0.02499013
```

```
smape_baseline_2<- mlr3measures::smape(as.numeric(merge_frame2$jaccard),
                                       as.numeric(y_predicted_baseline2),
                                       na_value = 0)
smape_baseline_2
```

```
## [1] 1.415952
```

Lasso

```
y_predicted_lasso2 <- predict(best_model_lasso,
                              s = best_lambda,
                              newx = data.matrix(merge_frame2[,7:18]))

rmse_lasso_2 <- rmse(as.numeric(merge_frame2$jaccard),
                     as.numeric(y_predicted_lasso2))

rmse_lasso_2
```

```
## [1] 0.022907
```

```
smape_lasso_2 <- mlr3measures::smape(as.numeric(merge_frame2$jaccard),
                                     as.numeric(y_predicted_lasso2), na_value = 0)
smape_lasso_2
```

```
## [1] 1.220765
```

```
smape_lasso2 <- abs(y_predicted_lasso2-merge_frame2$jaccard)/
                    ((y_predicted_lasso2+merge_frame2$jaccard)/2)
smape_lasso2 <- sort(smape_lasso2, index.return=TRUE, decreasing=TRUE)
smape_lasso2 <- smape_lasso2$ix[1:(nrow(merge_frame2)*0.0001)]
```

Linear

```
y_predicted_linear2 <- predict(linear, newdata = merge_frame2[,7:18])

rmse_linear_2<- rmse(as.numeric(merge_frame2$jaccard),
                     as.numeric(y_predicted_linear2))
rmse_linear_2
```

```
## [1] 0.02323218
```

```
smape_linear_2 <- mlr3measures::smape(as.numeric(merge_frame2$jaccard),
                                      as.numeric(y_predicted_linear2),
                                      na_value = 0)
smape_linear_2
```

```
## [1] 1.221728
```

```
smape_linear2 <- abs(y_predicted_linear2-merge_frame2$jaccard)/
                    ((y_predicted_linear2+merge_frame2$jaccard)/2)
smape_linear2 <- sort(smape_linear2, index.return=TRUE, decreasing=TRUE)
smape_linear2 <- smape_linear2$ix[1:(nrow(merge_frame2)*0.0001)]
```

PCR

```
y_predicted_pcr2 <- predict(pcr_model,
                            merge_frame2[,7:18],
                            ncomp = 4)

rmse_pcr_2<- rmse(as.numeric(merge_frame2$jaccard),
                  as.numeric(y_predicted_pcr2))
rmse_pcr_2
```

```
## [1] 0.02294309
```

```
smape_pcr_2 <- mlr3measures::smape(as.numeric(merge_frame2$jaccard),
                                   as.numeric(y_predicted_pcr2),
                                   na_value = 0)
smape_pcr_2
```

```
## [1] 1.220848
```

```
smape_pcr2 <- abs(y_predicted_pcr2-merge_frame2$jaccard)/
                  ((y_predicted_pcr2+merge_frame2$jaccard)/2)
smape_pcr2 <- sort(smape_pcr2, index.return=TRUE, decreasing=TRUE)
smape_pcr2 <- smape_pcr2$ix[1:(nrow(merge_frame2)*0.0001)]
```

GBM

```
y_predicted_gbm2 <- predict(gbm.fit,
                            merge_frame2[,7:18],
                            n.trees = gbm.fit$n.trees)

rmse_gbm_2 <- rmse(as.numeric(merge_frame2$jaccard),
                   as.numeric(y_predicted_gbm2))

rmse_gbm_2
```

```
## [1] 0.023527
```

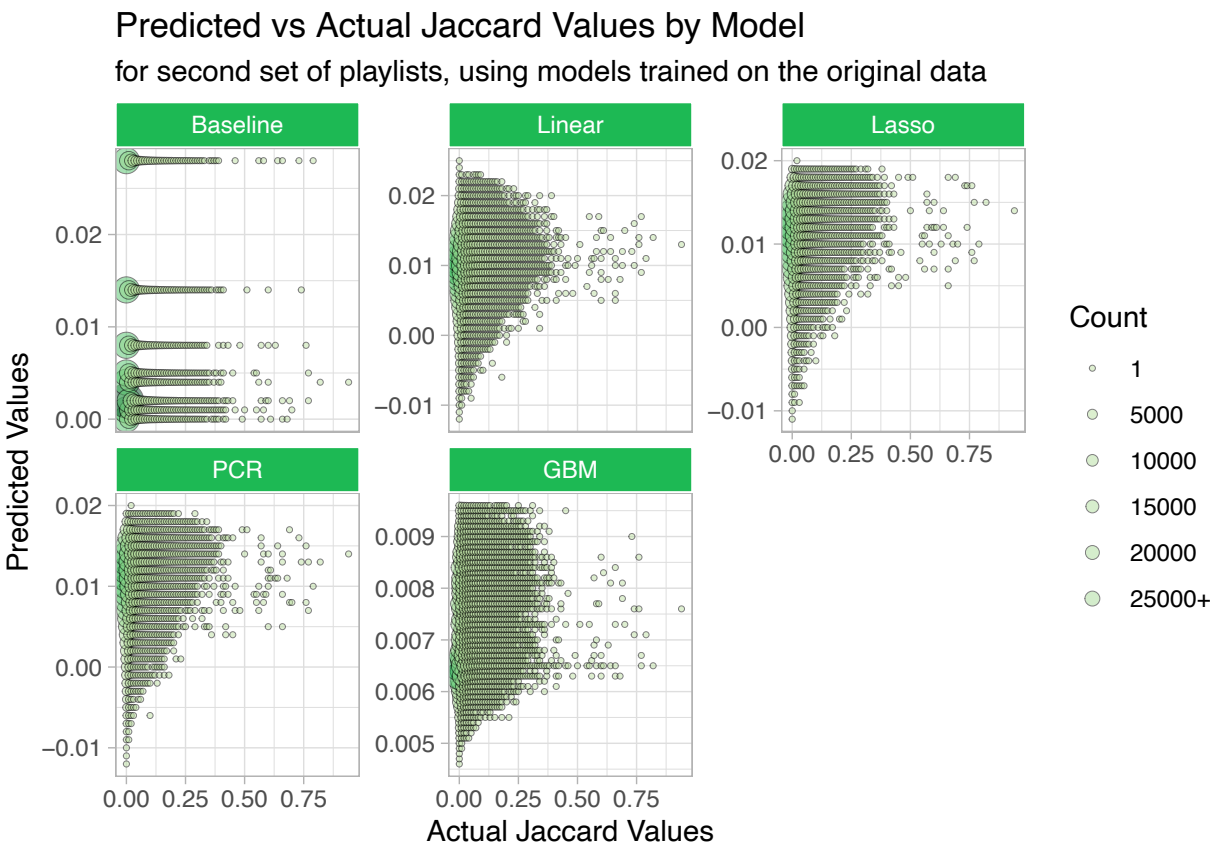
```
smape_gbm_2 <- mlr3measures::smape(as.numeric(merge_frame2$jaccard),
                                   as.numeric(y_predicted_gbm2),
                                   na_value = 0)

smape_gbm_2
```

```
## [1] 1.185025
```

```
smape_gbm2 <- abs(y_predicted_gbm2-merge_frame2$jaccard)/
                  ((y_predicted_gbm2+merge_frame2$jaccard)/2)
smape_gbm2 <- sort(smape_gbm2, index.return=TRUE, decreasing=TRUE)
smape_gbm2 <- smape_gbm2$ix[1:(nrow(merge_frame2)*0.0001)]
```

Jaccard Plots for Each Model

```
jacc2 <- data.frame(merge_frame2$jaccard, as.numeric(y_predicted_baseline2),
                      as.numeric(y_predicted_linear2), as.numeric(y_predicted_lasso2),
                      as.numeric(y_predicted_pcr2), as.numeric(y_predicted_gbm2))
names(jacc2) <- c("jaccard", "Baseline", "Linear", "Lasso", "PCR", "GBM")

jacc2 %>%
  reshape2::melt(id.vars = c("jaccard"),
                 variable.name = "model",
                 value.name = "predicted") %>%
  dplyr::mutate(jaccard = round(jaccard, 2),
                predicted = ifelse(model=="GBM",
                                   round(predicted, 4),
                                   round(predicted, 3))) %>%
  group_by(jaccard, predicted, model) %>%
  dplyr::summarise(count=n()) %>%
  ggplot(aes(x=jaccard, y=predicted, group=model, size=count)) +
  geom_point(aes(fill=count, size=count),
             color = "black", pch=21, alpha=.5, stroke=.1) +
  labs(x="Actual Jaccard Values",
       y="Predicted Values") +
  facet_wrap(~model, scales = "free_y") +
  scale_fill_gradientn(
    colors = c("#bee1a3", "#6ECD7C", "#1db954", "#004b23"),
    breaks=c(1, seq(5000, 25000, by=5000)),
    labels=c(1, seq(5000, 20000, by=5000), "25000+"),
    oob = scales::squish,
    name = "Count",
    na.value="grey",
    guide="legend") +
  scale_size_continuous(name="Count",
                        breaks=c(1, seq(5000, 25000, by=5000)),
                        labels=c(1, seq(5000, 20000, by=5000), "25000+")) +
  guides(fill= guide_legend(),
         size=guide_legend()) +
  labs(title="Predicted vs Actual Jaccard Values by Model",
       subtitle="for second set of playlists, using models trained on the original data") +
  theme_light() +
  theme(strip.background = element_rect(color="white", fill = "#1db954"))
```



Beyond our quantitative metrics, we also produced a graphical analysis of how our algorithm is working by plotting the predicted and the actual values for both the test set of the first group and second groups. In an ideal situation, our model would produce predicted values that precisely align with the actual figures, leading to a plot that looks like the line y=x.

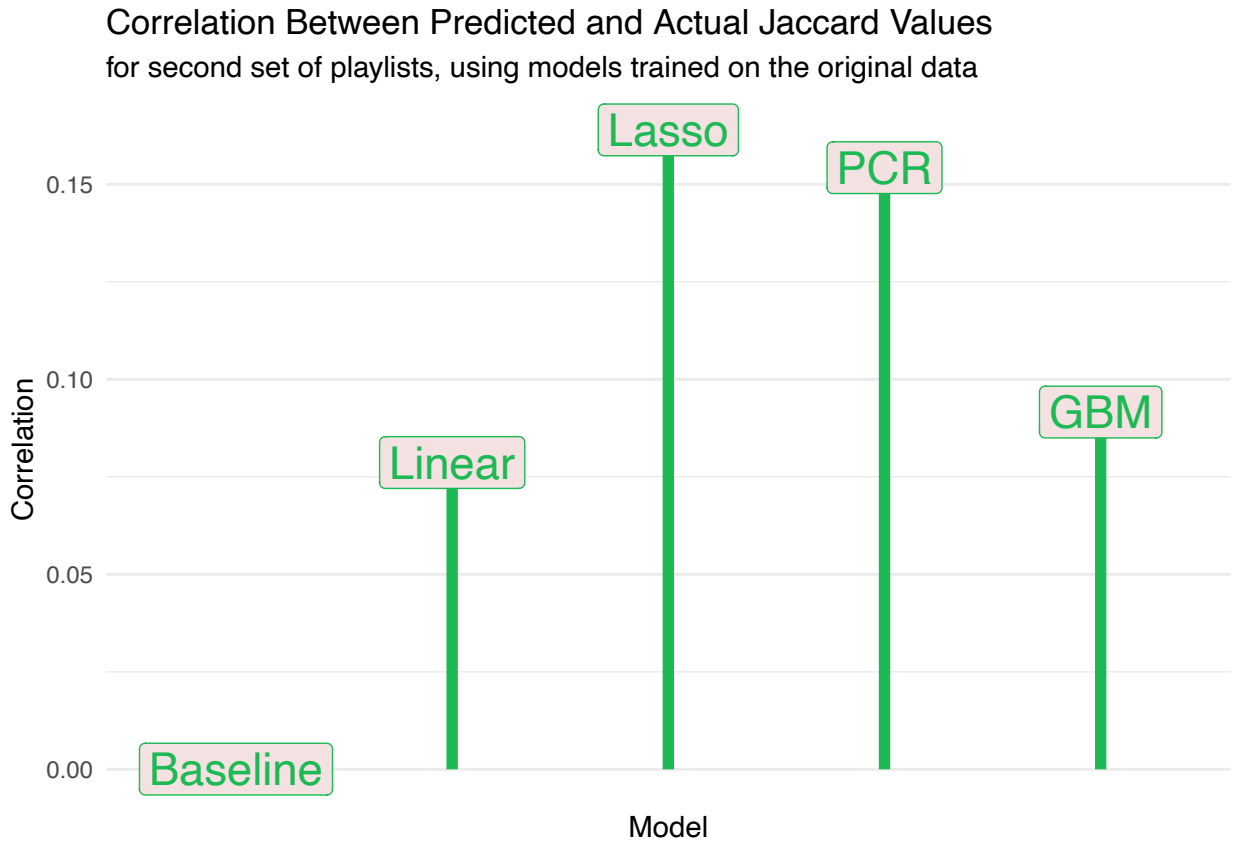
Unfortunately, that is not the case with our models. Instead, the plots show that our models simply cannot predict higher Jaccard values. This is the case both when comparing predicted Jaccard values to the test data from the first set of playlists as well as (even more so) to the data from the second set of playlists. Because the scales are different for the x- and y-axis (doing otherwise would result in every point being near 0 on the y-axis), the true level of dissimilarity between the predicted and actual values isn't readily apparent. However, closer inspection reveals that none of the models can predict Jaccard values above around 0.02.

This reflects the fact that there simply isn't a lot of variation in the feature scales we have. The high Jaccard and low Jaccard pairings are close enough alike (in terms of their absolute difference of features) so that the models simply cannot separate out enough difference between features to produce predictions that are high enough.

Correlation Lollipop Graph

```
jacc2 %>%
  reshape2::melt(id.vars = c("jaccard"),
                 variable.name = "model",
                 value.name = "predicted") %>%
  group_by(model) %>%
  dplyr::summarise(correlation = cor(jaccard, predicted)) %>%
```

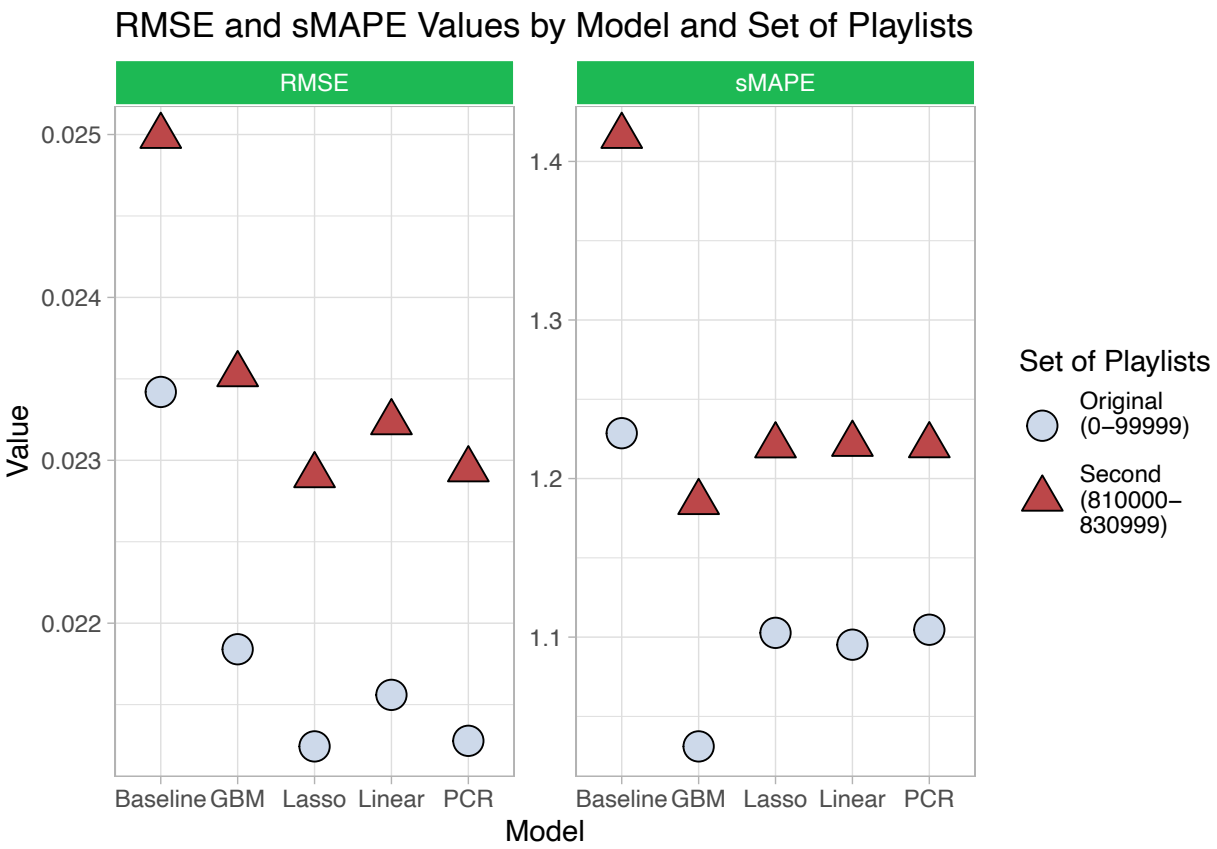
```
ggplot(aes(x=model, y=correlation)) +
geom_point() +
geom_segment(aes(x=model, xend=model, y=0, yend=correlation),
             color="#1DB954", size =2) +
geom_label(aes(label = model),
           color = "#1DB954", fill = "#F4E1E1",
           size=6) +
labs(x="Model",
     y="Correlation",
     title="Correlation Between Predicted and Actual Jaccard Values",
     subtitle="for second set of playlists, using models trained on the original data")+
theme_bw()+
theme(axis.text.x=element_blank(),
      axis.ticks=element_blank(),
      panel.grid.major.x = element_blank(),
      panel.grid.minor.x = element_blank(),
      panel.border = element_blank(),
      panel.background = element_blank())
```



However, it's fair to say that our models still perform better than the baseline in terms of correlation, which is positive in all cases compared to the baseline's ~0 correlation. At the end of the day, our models are still somewhat useful even if broadly so for smaller Jaccard values rather than larger ones.

RMSE and sMAPE for all models

```
values <- c(rmse_baseline,
            rmse_linear,
            rmse_lasso,
            rmse_pcr,
            rmse_gbm,
            rmse_baseline_2,
            rmse_linear_2,
            rmse_lasso_2,
            rmse_pcr_2,
            rmse_gbm_2,
            smape_baseline,
            smape_linear,
            smape_lasso,
            smape_pcr,
            smape_gbm,
            smape_baseline_2,
            smape_linear_2,
            smape_lasso_2,
            smape_pcr_2,
            smape_gbm_2)

model_name <- rep(c("Baseline", "Linear", "Lasso", "PCR", "GBM"), 4)

pl_set <- rep(c(rep("one", 5), rep("two", 5)), 2)

measure_type <- c(rep("RMSE", 10), rep("sMAPE", 10))

rmse_smape_vals <- data.frame(values, model_name, pl_set, measure_type)

rmse_smape_vals %>%
  ggplot(aes(x=model_name, y=values)) +
  geom_point(aes(fill=pl_set, shape=pl_set), size=5, color="black") +
  facet_wrap(~measure_type, scales = "free") +
  labs(x="Model",
       y="Value",
       fill="Set of Playlists",
       shape="Set of Playlists",
       title="RMSE and sMAPE Values by Model and Set of Playlists") +
  scale_fill_manual(values = c("#CDD9EA", "#bc4749"),
                    labels = c("Original\n(0-99999)\n ",
                               "Second\n(810000-\n830999)")) +
  scale_shape_manual(values = c(21, 24),
                     labels = c("Original\n(0-99999)\n ",
                                "Second\n(810000-\n830999)"))+
  theme_light() +
  theme(strip.background = element_rect(color="white",
                                        fill = "#1db954"))
```



These plots reveal a number of things. First, each of our models are better than the baseline on both metrics. Second, our models do worse on the second set of playlists than the test set of the first set of playlists by around 10% in most cases. However, because our models still performed fairly well on the second set of models, we can conclude that this relationship is real — not just due to chance in the first set of playlists we analyzed. Of course, there is some amount of uncertainty given that so many of the songs in this dataset were from more modern genres and pop and rap. If we had to test our models on data with classical music, our results might be no better than randomness. Third, model performance differs on the metrics — GBM is best on sMAPE and lasso and PCR are best on RMSE. (Note that lasso was tuned to find a lambda value that minimizes MSE, so a lasso tuned on sMAPE may be similarly strong at that metric). This is because, as outlined above, the two metrics measure slightly different types of error. GBM performs better on sMAPE because it produces predicted values that are broadly lower than the other models. Thus, it has better performance when thinking about relative error relative (sMAPE), but worse performance when considering error in absolute terms (RMSE). Put simply, it gets the smaller values (where relative error potential is higher but absolute error potential is lower) right at the expense of the other values. The best model to use for this purpose then depends on what the user's goal is in terms of measuring higher or smaller values more accurately.

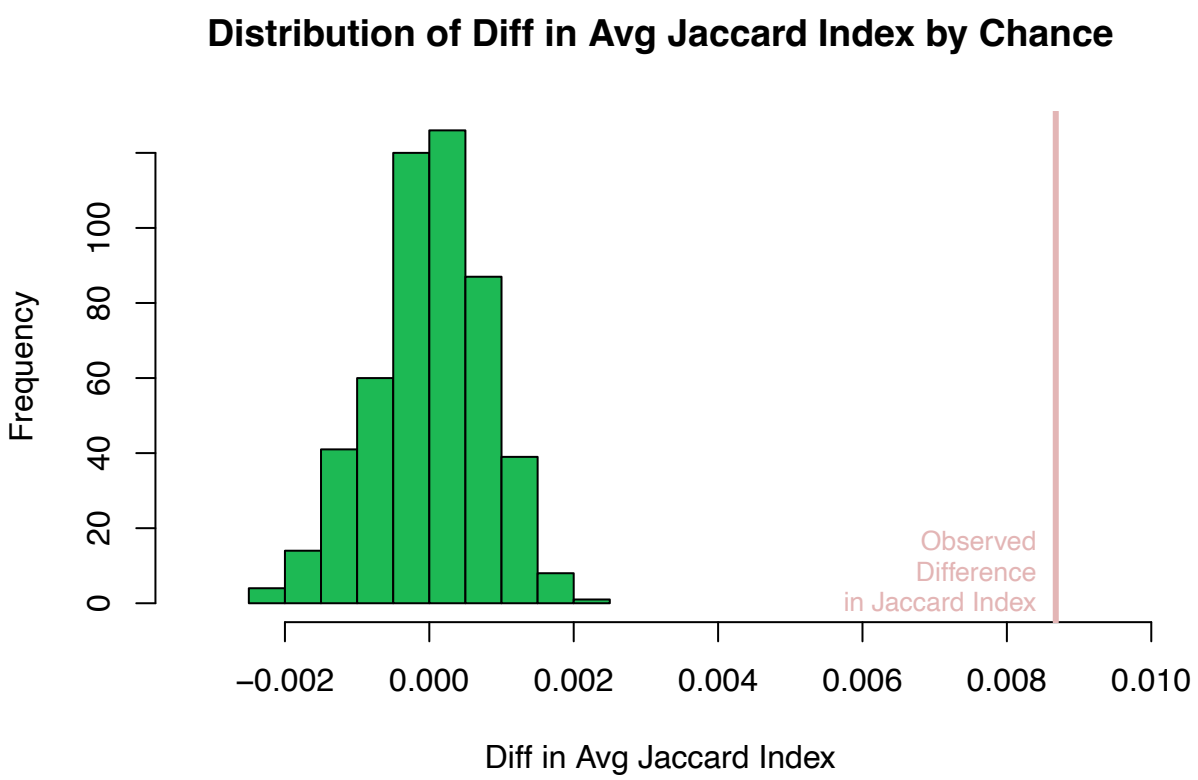
Looking at Top Errors

```
top_errors <- union(c(smape_lasso2, smape_linear2),
                    c(smape_gbm2, smape_pcr2))
merge_frame2$top_errors <- 0
merge_frame2$top_errors[top_errors] <- 1
```

```
merge_top_err2 <- merge_frame2[merge_frame2$top_errors == 1,]
top_error_songs2 <- unique(c(unique(merge_top_err2$pair1),
                             unique(merge_top_err2$pair2)))
track_info2$top_errors <- 0
track_info2$top_errors[track_info2$all_songs %in% top_error_songs2] <- 1
```

Jaccard Permutation Test and Histogram

```
#Compare top error frame to merge_frame overall:
set.seed(1)

avg_jaccard <- union(mean(merge_frame2$jaccard[merge_frame2$top_errors==0]),
                     mean(merge_frame2$jaccard[merge_frame2$top_errors==1]))
observed_diff <- avg_jaccard[1] - avg_jaccard[2]

B <- 500
permuted_err <- sample(merge_frame2$top_errors)
perm_avg_jaccard <- vector(length=2)
permuted_diffs <- vector(length=B)
for(i in 1:500){
    permuted_err <- sample(merge_frame2$top_errors)
    #perm_avg_jaccard <- tapply(merge_frame2$jaccard, permuted_err, mean)
    perm_avg_jaccard <- union(mean(merge_frame2$jaccard[permuted_err==0]),
                         mean(merge_frame2$jaccard[permuted_err==1]))
    permuted_diffs[i] <- perm_avg_jaccard[1] - perm_avg_jaccard[2]
}

hist(permuted_diffs,
     xlab="Diff in Avg Jaccard Index",
     main="Distribution of Diff in Avg Jaccard Index by Chance",
     col = "#1DB954",
     xlim = c(min(permuted_diffs)-0.001, observed_diff+0.001))
abline(v=observed_diff, col="#E3B5B5", lwd=3)
text(observed_diff, 5,
     paste("Observed\n Difference\nin Jaccard Index"),
     col = "#E3B5B5", cex=.8, pos=2)
```



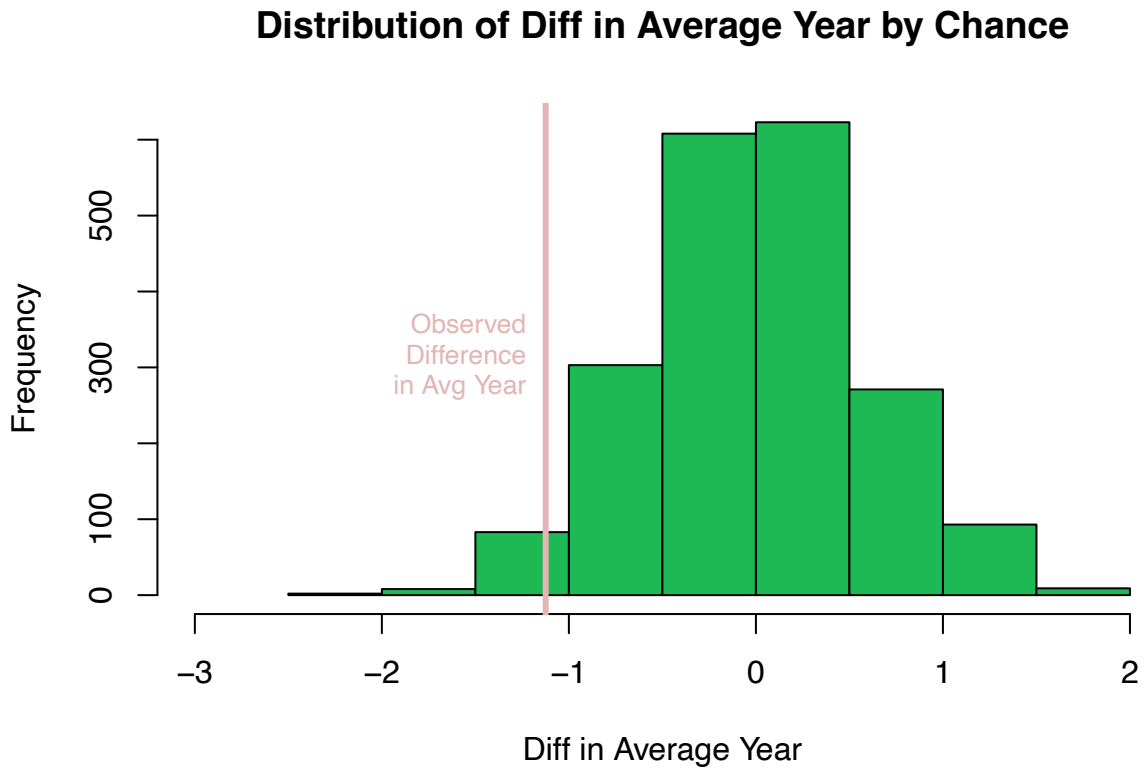
We also conducted some analyses to see how the pairings with the 0.01% top sMAPE errors from each of the four models (when run on data from the second group of playlists) were different from the rest of our pairings. First, we conducted a permutation test to see whether the mean of the Jaccard Index was similar or different for these two groups. The permutation test revealed that it was considerably different — the observed difference was far higher than distribution of the differences by chance. This means that the Jaccard value for the songs with the highest sMAPE values are much lower than those with other sMAPE values.

This is counterintuitive, given our previous discussion of how our models cannot predict high Jaccard values. One might reasonably think that the pairs with the highest sMAPE would have higher Jaccard values than the others. However, the reason the opposite is true is either a feature or a bug of sMAPE, depending on your perspective. Assuming that our predictions are around 0.01 and the actual values range from 0 (or slightly above) to 0.75, an analysis of the sMAPE formula will reveal that it produces a value of 2 (the maximum sMAPE value) when the actual value is 0. No matter how big the difference between 0.01 and 0.75 is, it will never reach 2, or even the sMAPE between 0.01 and 0.0001. So, small Jaccard values have the largest relative errors. Calculating top errors from RMSE would produce a much different result.

Year Permutation Test and Histogram

```
set.seed(1)
avg_year <- union(mean(track_info2$album_year[track_info2$top_errors==0]),
                  mean(track_info2$album_year[track_info2$top_errors==1]))
observed_diff <- avg_year[1] - avg_year[2]

B <- 2000
```

```r
permuted_err <- sample(track_info2$top_errors)
perm_avg_year <- vector(length=2)
permuted_diffs <- vector(length=B)
for(i in 1:2000){
    permuted_err <- sample(track_info2$top_errors)
    #perm_avg_jaccard <- tapply(merge_frame2$jaccard, permuted_err, mean)
    perm_avg_year <- union(mean(track_info2$album_year[permuted_err==0]),
                           mean(track_info2$album_year[permuted_err==1]))
    permuted_diffs[i] <- perm_avg_year[1] - perm_avg_year[2]
}

hist(permuted_diffs,
     xlab="Diff in Average Year",
     main="Distribution of Diff in Average Year by Chance",
     col = "#1DB954",
     xlim = c(round(min(permuted_diffs)-1), round(max(permuted_diffs))))
abline(v=observed_diff, col="#E3B5B5", lwd=3)
text(observed_diff, 300,
     paste("Observed\n Difference\nin Avg Year"),
     col = "#E3B5B5", cex=.8, pos=2)
```



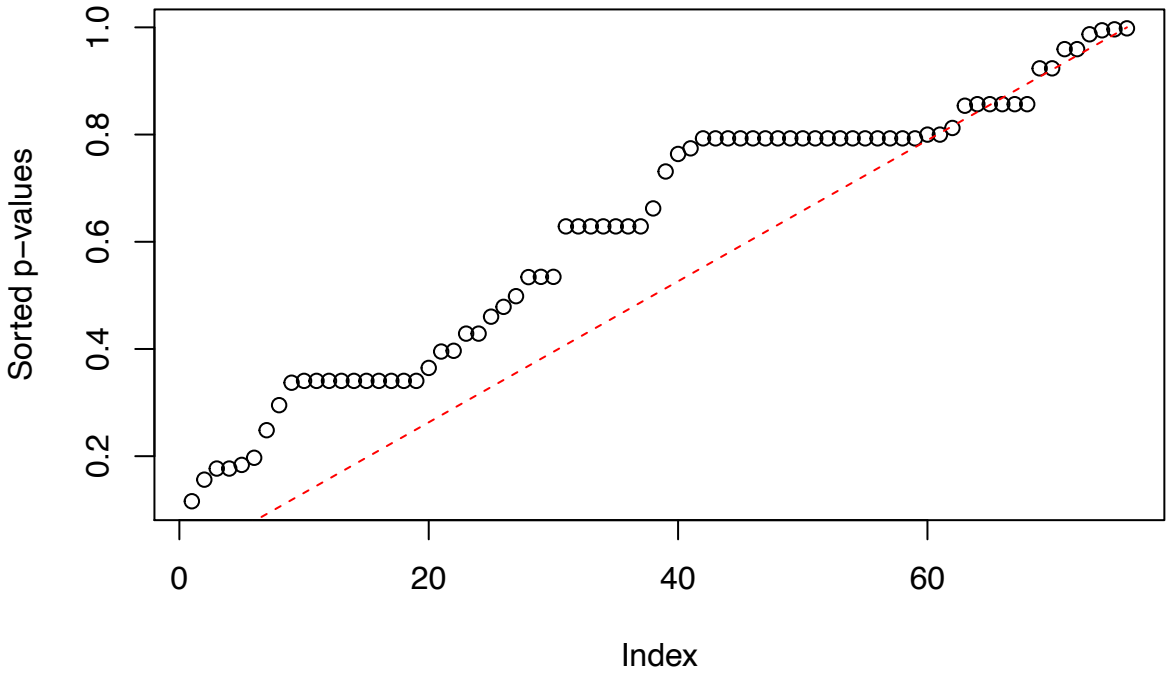
We also analyzed the average year that a song was released between the two groups (songs that were in at least one of the top error pairs and those that were not). Unlike Jaccard, this difference was not large compared to the differences by chance — the top error group had a slightly lower value, but only by around a year — our predictions are not particularly worse on more recent or more distant years.

Genre Chisq Test and Histogram

```
genre_error <- as.data.frame.matrix(table(track_info2$top_genre,
                                          track_info2$top_errors))

chisq.test(genre_error)
```

```
##
##  Pearson's Chi-squared test
##
## data:  genre_error
## X-squared = 88.879, df = 75, p-value = 0.1306
```

```
genre_error$sums <- rowSums(genre_error)

overall_prop <- genre_error %>%
    summarize_all(sum)

overall_prop <- as_vector(overall_prop)
overall_prop <- overall_prop[1:(length(overall_prop)-1)] / overall_prop[length(overall_prop)]

p_vals <- c()

for (i in 1:nrow(genre_error)) {
    n_genre <- as_vector(genre_error[i, ncol(genre_error)])
    error_props <- as_vector(genre_error[i, 1:(length(genre_error)-1)])
    expected_props <- overall_prop * n_genre
    chi_sq <- sum((error_props - expected_props)^2 / expected_props)
    p_vals[i] <- pchisq(chi_sq, df = ncol(genre_error)- 1, lower.tail = FALSE)
}

plot(1:nrow(genre_error), sort(p_vals),
     xlab = "Index", ylab = "Sorted p-values")
lines(1:nrow(genre_error), (1:nrow(genre_error))/nrow(genre_error),
      col = "red", lty = 2)
```




```
genre_error$p_value <- p_vals

#showing the genres with the ten highest and lowest p values
genre_error %>%
  arrange(desc(p_value)) %>%
  slice(1:10, 67:76) %>%
  kable(booktabs="T")
```

	0	1	sums	p_value
arkansas country	2	1	3	0.9981535
miami hip hop	4	2	6	0.9963103
latin	6	3	9	0.9944706
dance pop	45	20	65	0.9870799
neo mellow	3	1	4	0.9593813
reggae	3	1	4	0.9593813
pop punk	3	2	5	0.9235359
trap	3	2	5	0.9235359
art pop	1	1	2	0.8566929
canadian pop	1	1	2	0.8566929
singer-songwriter	0	1	1	0.3405181
modern rock	19	4	23	0.3370676
contemporary country	152	87	239	0.2952027
r&b	6	0	6	0.2484816
classic rock	7	0	7	0.1970198

	0	1	sums	p_value
stomp and holler	2	4	6	0.1837569
electropop	1	3	4	0.1769458
indie folk	1	3	4	0.1769458
soft rock	8	0	8	0.1562160
australian psych	0	2	2	0.1159526

Finally, we conduct a chi-squared test (as well as individual tests of independence) on the relationship between genre and top errors. Both the overall chi-squared test as well as each of the tests of independence fail to reject the null hypothesis (some due more to low sample size rather than to truly similar results), meaning that we cannot say that our data is significantly worse (at least in terms of songs that were included in the top error pairs) at predicting certain types of genres than others.

CONCLUSION

In conclusion, it seems that some relationship between song similarity as determined by humans (playlists) and song similarity as determined by track features (the song metrics) exists. However, because the range of the track features is inherently limited, this relationship cannot explain the full range of playlist co-occurrence as measured by the Jaccard Index. This is a big problem, as we care quite a bit about predicting the top co-occurring songs (potentially even more than about other pairings) as they would populate the automatically created playlists.

To have a predictive algorithm with substantial accuracy, we could include information about the artist (and album, etc) that made the song. However, this would render our work fundamentally useless. That algorithm would undoubtedly find that songs with the same artist appear together in the same playlist more often than other songs holding all else equal. But that's not an important insight for Spotify or music creators. If the automated playlist creator used that algorithm, it would only ever return songs from a single artist.

So, we are forced to sacrifice quantitative accuracy for the sake of externally important results. This is a common problem in industry — finding things that are alike, but not too much alike. We explore one approach to this and find some success (coefficients that are mostly in the right direction) but significant problems handling song pairings with higher Jaccard Indices. Future work should explore this critical question and how to get around it. Our methods provide a good start, but other features (maybe artist similarity or song popularity) are needed to generate customer-usable predictions.

Reviewing another group's project

We reviewed Ohad and Joe's project about nonprofit finances. The motivation for this project is clear, namely, to find a way to help nonprofit groups better successfully use their donations to run their organizations, and more importantly, how to raise more money to begin with. Ohad and Joe used data from the IRS, specifically the 2020 nonprofit tax returns in the Form 990 extract for more than 200,000 of the largest organizations nationwide. The data mining aspect here is algorithmically scanning a vast amount of data to determine what features have the most impact on the success of an organization, and which have no impact. They found that of the 255 features, their algorithm retained only 10 as being impactful to the outcome, with some being non-functional expense variables, some functional expense variables, and a few related to total revenue, among other things. Between the use of Lasso and K means, they found that across different subsections, most of the organizations had overall similar finances, and nothing stood out as exceptionally irregular or impactful.

In terms of things we would have differently, it would have been beneficial to perhaps see a bit more viz throughout to get a true sense of what the data was showing, or perhaps a step by step walkthrough of the

analysis with a specific organization to create a narrative through which to understand the algorithm. They also log-transformed many of their values, and it may have made more sense to just simply scale them within their models since log transformations can make interpretations more difficult after the fact while scaling allows us to think of things in terms of standard deviations or a similar common factor.