Predicting song popularity using pseudo-random sampling of spotify catalog

prj-pcbarko-schen176–mettler3-yc62-gianghl2

2022-12-07

Abstract

Spotify is an audio streaming service used by hundreds of millions of people. The popularity of Spotify songs is described by a popularity index and each song is associated with features that quantify various sonic attributes (e.g. pitch, danceability, etc.). For this group project, we sought to generate statistical models that can predict the popularity of a song based on these sonic attributes. We constructed a Spotify songs database using custom R and Python scripts to query the Spotify API and appended it to a similar, publicly available database. Song popularity was predicted using several models, whose performance was compared.

Introduction

Spotify (https://open.spotify.com) is a web-based audio streaming service, first launched in 2008, with 456 million users, including 195 million subscribers across 183 markets. The popularity of each song is quantified using a numeric popularity index from 0-100. Each song is also associated with metadata including genre, artist, and several attributes that describe various acoustic/sonic features:

- acousticness
- danceability
- durations
- energy
- instrumentalness
- key
- liveness
- loudness
- mode
- speechiness
- tempo
- valence

We sought to model the song popularity index using the acoustic attributes in a large database of Spotify songs. There are several publicly available Spotify song datasets, but these are outdated. Additionally, we sought to develop a novel means of accessing song data from the Spotify API.

Our *fist objective* was to create a new, updated dataset of Spotify songs. We attempted to accomplish this by generating random song IDs and using these to search the Spotify API.

Our *second objective* was to model song popularity (dependent variable) from the acoustic attributes (independet variables). Others have attempted to model popularity from the acoustic attributes and genre, but most used linear models that had poor performance. For this project, we utilized alternative approaches to modeling/predicting song popularity from acoustic attributes and compared them with respect to performance.

Sampling Techniques

Issues generating random sample

Randomly sampling the spotify catalog is a difficult task that is beyond the timeframe and computing power available to our cohort. Spotify randomly assigns each music track a twenty one character id. The first character is always numeric but the remaining characters are case sensitive alphanumeric characters with repetition. Thus, we have a sample space of $(62^{20})(10)$ but the actual sample space of spotify tracks is a much smaller subset of the total sample space (approximately 1.41^{-28} of the sample space). Below we implemented the stringi package to generate random id's that are then pulled from Spotify's api using the Spotifyr package. Unfortunately, due to the time it would take to generate a random sample this way and that the spotify API limits the number of inquiries a developer can make we would not be able to generate a sample this way.

```
# generates 21 character long IDs that were used to query the API
spot_id_track_check <- function(x) get_tracks(x)

spot_ids <- function(Length) {

    st_int <- stri_rand_strings(1, 1, "[0-9]")
    st_char <- stri_rand_strings(as.integer(Length), 21, pattern = "[A-Za-z0-9]")
    spot_id <- seq(st_int)
    for (i in spot_id) {
        spot_id[i] <- paste(st_int[i], st_char[i], sep = "")
    }

    xt <- do.call(c, replicate(n = Length, mclapply(spot_id, function(x) {
        spot_id_track_check(x)
    }, mc.cores = 48)))
    return(xt)
}

songs <- spot_ids("1000")

# this does not return any valid song data because of the large sample space
songs
```

Without the ability to draw a sample from randomly generating string id's, We decided to use the function *search_spotify* to return tracks from randomly generated strings that the function uses to search for matches with track id's. For instance, if we use a generated string *"abd"* then tracks that include these characters in the title will be returned from the search. Additionally, we used a vector of each upper/lower case and number to return tracks that match each character. This sampling method is effective but has some drawbacks. The search function is a subset of all Spotify tracks. When iterating over characters we are likely to get repeat tracks if we draw a large enough sampling. Thus, not all tracks are able to be sampled and the sample size we can actually draw is limited to several hundred thousand tracks. Below are several examples of pseudo random samples using shell script, R, and Python.

Bash sampling script

```
#!/bin/bash
# Bash script to search for random ids for Spotify tracks using the web API.
# Usage: bash spotify_scraping.sh $CLIENT_ID $CLIENT_SECRET

CLIENT_ID=$1
CLIENT_SECRET=$2

creds=$CLIENT_ID:$CLIENT_SECRET
encoded_creds=$(echo -n $creds | base64)

access_token=$(curl -s -X "POST" -H "Authorization: Basic $encoded_creds" -d grant_type=client_c
redentials https://accounts.spotify.com/api/token | awk -F"\"" '{print $4}')
echo $access_token
rm result.json extracted_ids.txt

for x in {a..z}
do
for y in {0..9}
do
curl --request GET --url "https://api.spotify.com/v1/search?q=$x$y&type=track&limit=50&year=1970
-2023" --header "Authorization: Bearer $access_token" --header "Content-Type: application/json"
>> result.json
done
done
cat result.json | grep -o "\"id\" :.*" | sed 's/"id" : "//g' | sed 's/",$//g' >> extracted_ids.t
xt
cat extracted_ids.txt | sort | uniq > tmp
mv tmp extracted_ids.txt ## extract id's to use in R spotifyr package

#### use R package to gain audio track features using R spotify package
Sys.setenv(SPOTIFY_CLIENT_ID = '')
Sys.setenv(SPOTIFY_CLIENT_SECRET = '')

access_token <- get_spotify_access_token()

setwd("/Users/gianghale/Documents/GitHub/fa22-prj-pcbarko-schen176--mettler3-yc62-gianghl2/Old_D
ata")
ids <- read.table("extracted_ids.txt")
ids <- apply(ids, MARGIN=1,FUN=toString)
ids_list <- as.list(strsplit(ids, " "))

# write track audio features to csv
lapply(ids_list, function(x) write.table(get_track_audio_features(x), 'track_features_25k.csv',
append=TRUE,col.names = FALSE, sep=',' ))
```

R function below makes use of the Spotifyr, parallel, and stringi packages to generate track id's from Spotify's API. The track id's are then used in Spotifyr's *get_track_audio_features* to return the audio features we will use to predict popularity.

R sampling script

```
Sys.setenv(SPOTIFY_CLIENT_ID = "")
Sys.setenv(SPOTIFY_CLIENT_SECRET = "")


# function ran inside mylist function to get audio features of sampled tracks
spot_id_track_check <- function(x) get_track_audio_features(x)

mylist <- function(x) {
    xL <- c()
    st_char <- stri_rand_strings(2, x, pattern = "[A-Za-z0-9]")  ## generate random strings to m
atch and return from Spotify API
    xL <- foreach(i = st_char) %do% {
        # search track returns tracks from spotify API
        search_spotify(i, type = "track", market = NULL, limit = 50, offset = 0,
            include_external = NULL, include_meta_info = FALSE)

    }
    xt <- xL
    # bind
    sp_s <- bind_rows(xt, .id = "column_label")
    x_id <- sp_s$id
    ## get audio track features
    sp_f <- spot_id_track_check(x_id)
    sp_s <- sp_s[-c(1, 3, 17, 18, 21)]
    artists <- do.call(cbind.data.frame, lapply(sp_s, function(x) {
        # check if list
        if (is.list(x)) {
            data.frame(t(sapply(x, "[", seq(max(lengths(x))))))
        } else {
            x
        }
    }))

    spot_data <- cbind(artists, sp_f)

    ## return two dataframes,track information and audio features
    return(spot_data)
}


### Replicate function to get large dataset of spotify tracks
x <- replicate(n = 2, mclapply(3, function(x) {
    mylist(x)
}, mc.cores = 48))

x1 <- bind_rows(x, .id = "column_label")

# Including only wanted parameters
sp_all_unique <- x1[, c(12, 14, 4, 22, 23, 24, 15, 32, 33, 34, 35, 36, 37, 38, 39,
    40, 41, 42, 9)]
```

```
## remove duplicates
sp_all_unique <- sp_all_unique[!duplicated(sp_all_unique[1]), ]

sp_all_unique <- sp_all_unique %>%
    mutate(Popularity_Quantized = cut(popularity, breaks = c(0, 25, 50, 75, 100),
        include.lowest = TRUE))

colnames(sp_all_unique) <- c("TrackID", "SongName", "Artist", "track_album_id", "Album_Name",
    "Release_Date", "Popularity", "Danceability", "Energy", "Key", "Loudness", "Mode",
    "Speachiness", "Acousticness", "Instrumentalness", "Liveness", "Valence", "Tempo",
    "Duration", "Popularity_Quantized")

# sample of songs saved as songs.tsv
write_tsv(sp_all_unique, "songs.tsv")
```

Python sampling script

```
# Python script to sample tracks

import csv
import itertools
import string

import pandas as pd
import spotipy
from spotipy.oauth2 import SpotifyClientCredentials


class CrawlerSpotify:
    def __init__(self, client_id, client_secret):
        # authorization information
        self.id = client_id
        self.secret = client_secret
        self.spotify = spotipy.Spotify(client_credentials_manager=
                                       SpotifyClientCredentials(client_id=self.id, client_secret
=self.secret))

    def save_tracks(self, output_path: str = None):
        search_words = self._generate_search_words()  # generate search words

        results = []
        for sq in search_words:  # iterate search words
            search_results = self.spotify.search(q=sq, type='track', limit=50)
            total = search_results['tracks']['total']
            print(f'===== A total of {total} results from search word: {sq} =====')

            for offset in range(0, total if total < 1000 else 1000, 50):  # iterate pages.
                # for each search word, the API returns 1000 results at most,
                # so we can get maximum 20 pages for each search word (50 tracks each page)
                # see https://developer.spotify.com/documentation/web-api/reference/#/operation
s/search
                print(f'results from page {int(offset / 50 + 1)}')
                tracks = self._parse_tracks(self.spotify.search(q=sq, type='track', limit=50, of
fset=offset))
                audio_features = self.spotify.audio_features([track['track_id'] for track in tra
cks])  # audio features
                result = [{**track, **self._parse_audio_features(audio_feature)} if audio_featur
e else track for
                          track, audio_feature in
                          zip(tracks, audio_features)]  # combine track details and audio featur
es
                self._save_csv(result, output_path)  # save to csv
                results.append(result)

        return pd.DataFrame(results)

    @staticmethod
```

```
    def _save_csv(result, output_path: str):
        csv_header = ['track_id', 'track_name', 'track_artist', 'track_popularity', 'track_album
_id',
                      'track_album_name', 'track_album_release_date', 'danceability', 'energy',
'key',
                      'loudness', 'mode', 'speechiness', 'acousticness', 'instrumentalness', 'li
veness',
                      'valence', 'tempo', 'duration_ms']

        with open(output_path, 'a', newline='', encoding='utf-8-sig') as fp:
            csv_writer = csv.DictWriter(fp, csv_header)
            if fp.tell() == 0:
                csv_writer.writeheader()
            csv_writer.writerows(result)

    @staticmethod
    def _parse_tracks(search_results):
        """
        extract useful data from search results
        """
        details = []
        for track in search_results['tracks']['items']:
            details.append({
                'track_id': track['id'],
                'track_name': track['name'],
                'track_artist': track['artists'][0]['name'],
                'track_popularity': track['popularity'],
                'track_album_id': track['album']['id'],
                'track_album_name': track['album']['name'],
                'track_album_release_date': track['album']['release_date']
            })
        return details

    @staticmethod
    def _parse_audio_features(audio_feature_results):
        """
        delete unnecessary data from audio feature results
        """
        del_key = ['type', 'uri', 'track_href', 'analysis_url', 'time_signature', 'id']
        return {key: audio_feature_results[key] for key in audio_feature_results if key not in d
el_key}

    @staticmethod
    def _generate_search_words():
        """
        generate search words. from letters a-z and numbers 0-9, with years 1985-2022
        """
        search_words = []
        for word, year in itertools.product(list(string.ascii_lowercase) + list(range(0, 10)), l
ist(range(1985, 2023))):
            search_words.append(f'{word} year:{year}')
        return search_words
```

```
if __name__ == '__main__':
    my_client_id = 'Your Client ID'
    my_client_secret = 'Your Client Secret'
    my_output_path = r'result_spotify.csv'

    crawler = CrawlerSpotify(client_id=my_client_id, client_secret=my_client_secret)
    data_tracks = crawler.save_tracks(output_path=my_output_path)  # 1,368,207 tracks

    # delete duplicates
    data = pd.read_csv(my_output_path)
    data_final = data.drop_duplicates().reset_index(drop=True)  # 573,131 tracks
    data_final.to_csv(r'result_spotify_noduplicates.csv', index=False, encoding='utf-8-sig')
    # data_final['loudness'].isna().sum()  # 615 tracks can not extract audio features


# Returns 573k tracks which we split into tsv.gz files to upload to github
```

Data Preparation and Cleaning

Scraping the Spotify API resulted in the generation of three datasets of Spotify Songs. These datasets were cleaned and combined to generate a single database of songs.

```
list.files()
```

```
##  [1] "cl_summary.png"                 "Descriptive Plots"
##  [3] "Final_Presentation.html"        "Final_Presentation.md"
##  [5] "Final_Presentation.Rpres"       "Final_Project_Final_Version.Rmd"
##  [7] "gbm_cl_int_depth_iter.png"      "gbm_reg_int_depth_iter.png"
##  [9] "Older_Versions"                 "reg_summary.png"
## [11] "rf_cl_mtry.png"                 "rf_cl_varimp.png"
## [13] "rf_reg_mtry.png"                "rf_reg_varimp.png"
## [15] "songs.tsv"                      "songs2.tsv.gz"
## [17] "songs3.tsv.gz"                  "svm_cl_cost.png"
## [19] "svm_reg_cost.png"
```

```
# this was generated in R - the files have been renamed for convenience
songs <- read_tsv("songs.tsv", col_names = TRUE, show_col_types = FALSE)

# These were generated in Python - the files have been renamed for convenience
songs2 <- read_tsv("songs2.tsv.gz", col_names = TRUE, show_col_types = FALSE)

songs3 <- read_tsv("songs3.tsv.gz", col_names = TRUE, show_col_types = FALSE)
```

Synchronize column names and order:

```
colnames(songs)
```

```
##  [1] "X"                  "TrackID"            "SongName"
##  [4] "Artist"             "Popularity"         "Danceability"
##  [7] "Energy"             "Key"                "Loudness"
## [10] "Mode"               "Speachiness"        "Acousticness"
## [13] "Instrumentalness"   "Liveness"           "Valence"
## [16] "Tempo"              "Duration"           "Popularity_Quantized"
```

```
table(colnames(songs2) == colnames(songs3))
```

```
##
## TRUE
##   20
```

```
songs <- songs[, -c(1, 18)]
songs2 <- songs2[, -c(1, 6:8)]
songs3 <- songs3[, -c(1, 6:8)]
```

```
names(songs2) <- colnames(songs)

names(songs3) <- colnames(songs)
```

Combine all datasets:

```
songs_combined <- data.frame(rbind(songs, songs2, songs3))

str(songs_combined)
```

```
## 'data.frame':    661956 obs. of  16 variables:
##  $ TrackID         : chr  "2uflssWlCaJ6CbTMOsUpNI" "46M2hXnaQpueG7vSvgVtVH" "0wihfILRNOwE2156
Shezc8" "2Y0iGXY6m6immVb2ktbseM" ...
##  $ SongName        : chr  "Aguacero" "GTG" "Agosto" "Little Dark Age" ...
##  $ Artist          : chr  "Bad Bunny" "Freddie Dredd" "Bad Bunny" "MGMT" ...
##  $ Popularity      : num  83 76 82 82 70 76 66 51 73 67 ...
##  $ Danceability    : num  0.861 0.88 0.849 0.705 0.708 0.822 0.547 0.686 0.603 0.815 ...
##  $ Energy          : num  0.645 0.777 0.584 0.712 0.912 0.59 0.453 0.433 0.204 0.697 ...
##  $ Key             : chr  "6" "7" "1" "6" ...
##  $ Loudness        : num  -6.96 -6.88 -8.2 -6.16 -3.19 ...
##  $ Mode            : chr  "0" "0" "0" "1" ...
##  $ Speachiness     : num  0.0743 0.125 0.115 0.0385 0.0838 0.18 0.0358 0.0339 0.112 0.219 ...
##  $ Acousticness    : num  0.423 0.164 0.0929 0.0102 0.487 0.429 0.189 0.239 0.479 0.778 ...
##  $ Instrumentalness: num  5.92e-04 1.71e-02 6.18e-06 8.55e-04 1.77e-02 0.00 3.29e-02 1.80e-01
7.14e-02 3.98e-04 ...
##  $ Liveness        : num  0.349 0.0957 0.492 0.1 0.79 0.102 0.119 0.0898 0.642 0.102 ...
##  $ Valence         : num  0.668 0.974 0.724 0.62 0.571 0.383 0.412 0.389 0.71 0.147 ...
##  $ Tempo           : num  121.4 155 115 97.5 118 ...
##  $ Duration        : num  210989 93893 139041 299960 150466 ...
```

Scan for duplicates:

```
dupes <- songs_combined %>%
    janitor::get_dupes(TrackID)

# nrow(dupes)
```

There were 3.23125^{4} duplicate track IDs in the combined dataset. These duplicated rows were removed.

```
songs_combined <- songs_combined %>%
    filter(duplicated(TrackID) == FALSE)
```

Confirm that duplicate rows have been removed:

```
songs_combined %>%
    janitor::get_dupes(TrackID)
```

```
## No duplicate combinations found of: TrackID
```

The “mode” attribute codes whether the song is written in a major or minor key and is stored as a binary numeric variable (0 = minor; 1 = mojor). For interpretability, this attribute was decoded.

```
songs_combined$Mode <- case_when(songs_combined$Mode == 1 ~ "Major", songs_combined$Mode ==
    0 ~ "Minor")
```

Similarly, each song’s key is stored as a numeric variable and was decoded for interpretability:

```
songs_combined$Key <- case_when(songs_combined$Key == 0 ~ "C", songs_combined$Key ==
    1 ~ "C#/Db", songs_combined$Key == 2 ~ "D", songs_combined$Key == 3 ~ "D#/Eb",
    songs_combined$Key == 4 ~ "E", songs_combined$Key == 5 ~ "F", songs_combined$Key ==
        6 ~ "F#/Gb", songs_combined$Key == 7 ~ "G", songs_combined$Key == 8 ~ "G#/Ab",
    songs_combined$Key == 9 ~ "A", songs_combined$Key == 10 ~ "A#/Bb", songs_combined$Key ==
        11 ~ "B")
```

Finally, the popularity index was quantized in order to convert it from a numeric to a categorical variable. This will be useful for classification models that require categorical independent variables.

```
songs_combined <- songs_combined %>%
    mutate(Popularity_quantized = cut(Popularity, breaks = c(0, 25, 50, 75, 100),
        include.lowest = TRUE))

# table(songs_combined$Popularity_quantized)

songs_combined$Popularity_quantized <- factor(songs_combined$Popularity_quantized)

# levels(songs_combined$Popularity_quantized)
```

Finally, remove rows with NA values

```
songs_combined <- songs_combined %>%
    drop_na()
```

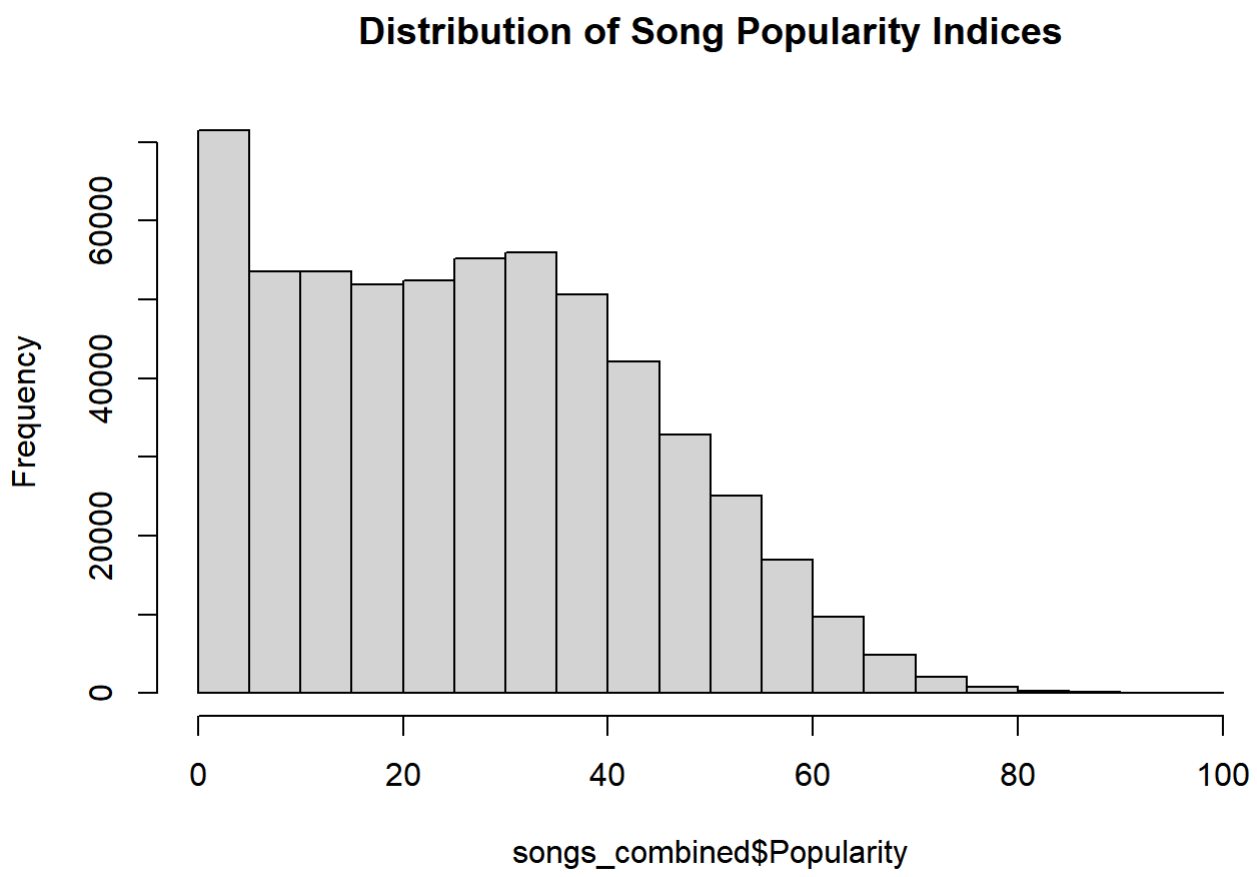
Exploratory Analysis

There were 579108 songs from 98589 artists in the combined Spotify Songs dataset.

Song Popularity

Distribution of song popularity indices:

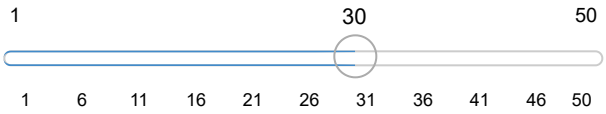
```
hist(songs_combined$Popularity, main = "Distribution of Song Popularity Indices")
```



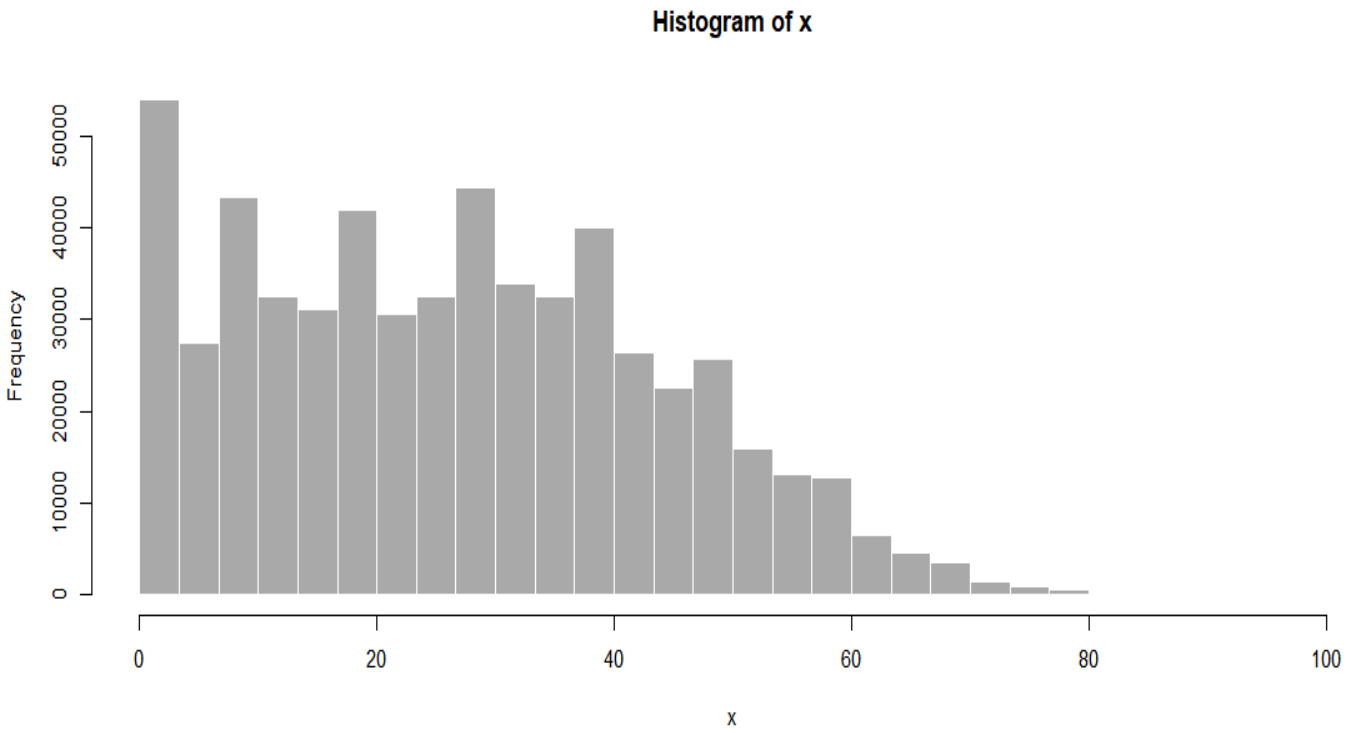
Here is Shiny capability to try out

```
sliderInput("bins", "Number of bins:", 30, min = 1, max = 50)
```

Number of bins:

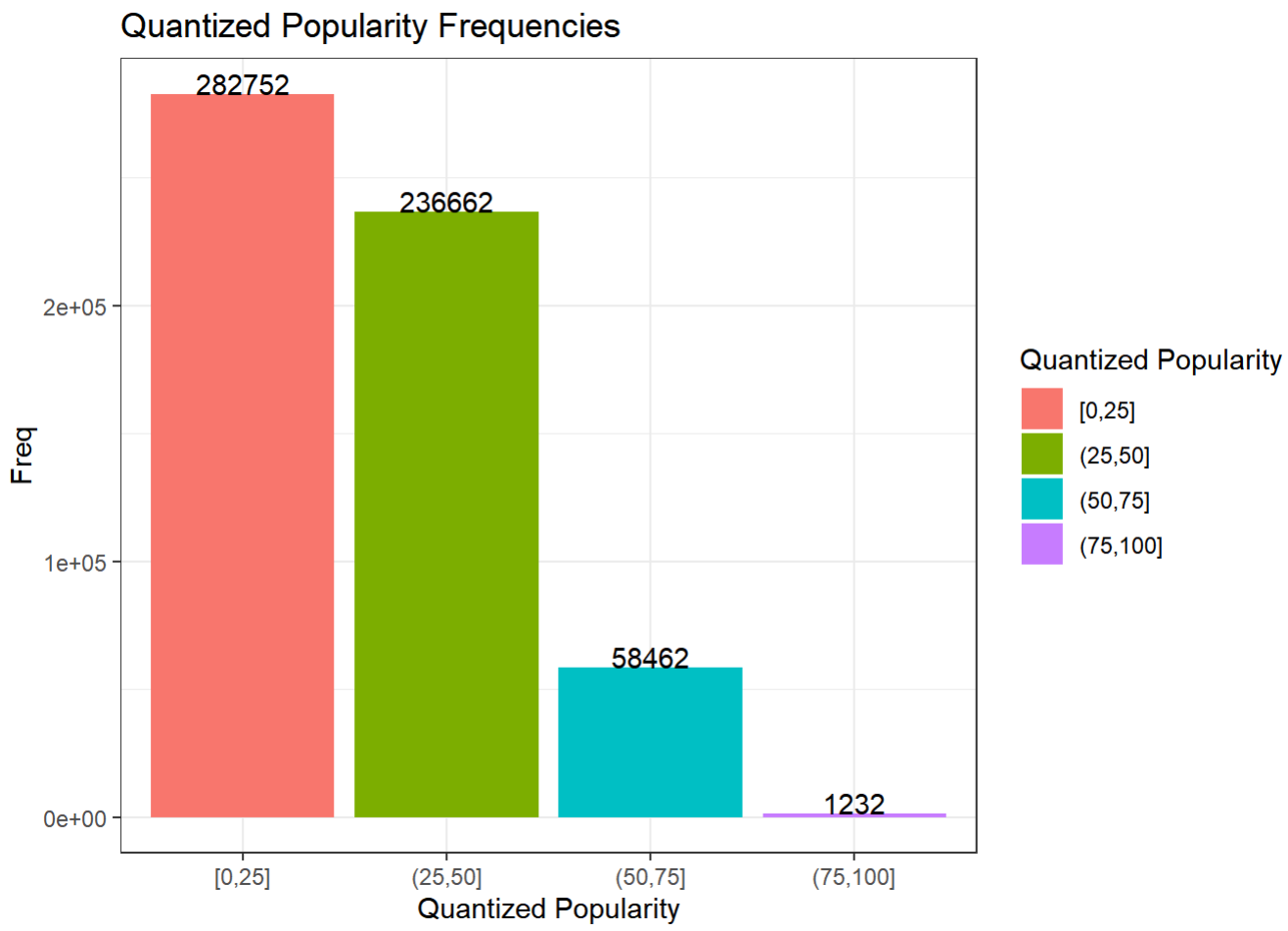



```
renderPlot({
    x = songs_combined$Popularity
    bins = seq(min(x), max(x), length.out = input$bins + 1)

    # draw the histogram with the specified number of bins
    hist(x, breaks = bins, col = "darkgray", border = "white")
})
```



Distribution of quantized popularity indices:

```
quant_pop <- data.frame(table(songs_combined$Popularity_quantized))

names(quant_pop)[1] <- "Quantized Popularity"

ggplot(quant_pop, aes(x = `Quantized Popularity`, y = Freq, fill = `Quantized Popularity`)) +
    geom_bar(stat = "identity") + geom_text(aes(label = Freq), vjust = 0) + theme_bw() +
    ggtitle("Quantized Popularity Frequencies")
```




```
kable(quant_pop)
```

Quantized Popularity	Freq
[0,25]	282752
(25,50]	236662
(50,75]	58462
(75,100]	1232

Clearly, popular songs are rare in this dataset and the majority of songs had low popularity.

The top twenty most popular songs:

```
songs_combined %>%
    arrange(desc(Popularity)) %>%
    dplyr::select(SongName, Artist, Popularity) %>%
    top_n(20) %>%
    kable
```

```
## Selecting by Popularity
```

SongName	Artist	Popularity
Unholy (feat. Kim Petras)	c(""Sam Smith"", ""Kim Petras"")	100
Quevedo: Bzrp Music Sessions, Vol. 52	c(""Bizarrap"", ""Quevedo"")	97
La Bachata	Manuel Turizo	97
I'm Good (Blue)	c(""David Guetta"", ""Bebe Rexha"")	97

SongName	Artist	Popularity
Tití Me Preguntó	Bad Bunny	96
Me Porto Bonito	c(""Bad Bunny"", ""Chencho Corleone"")	96
I Ain’t Worried	OneRepublic	95
Under The Influence	Chris Brown	95
Anti-Hero	Taylor Swift	94
Efecto	Bad Bunny	94
As It Was	Harry Styles	94
Ojitos Lindos	c(""Bad Bunny"", ""Bomba Estéreo"")	93
Glimpse of Us	Joji	93
CUFF IT	Beyoncé	93
Romantic Homicide	d4vd	93
SNAP	Rosa Linn	93
Moscow Mule	Bad Bunny	92
Neverita	Bad Bunny	92
PROVENZA	KAROL G	92
Calm Down (with Selena Gomez)	c(""Rema"", ""Selena Gomez"")	92
DESPECHÁ	ROSALÍA	92
Another Love	Tom Odell	92

Now, the interactive, shiny version of the table - the user can select how many songs to look at.

How Many Songs?

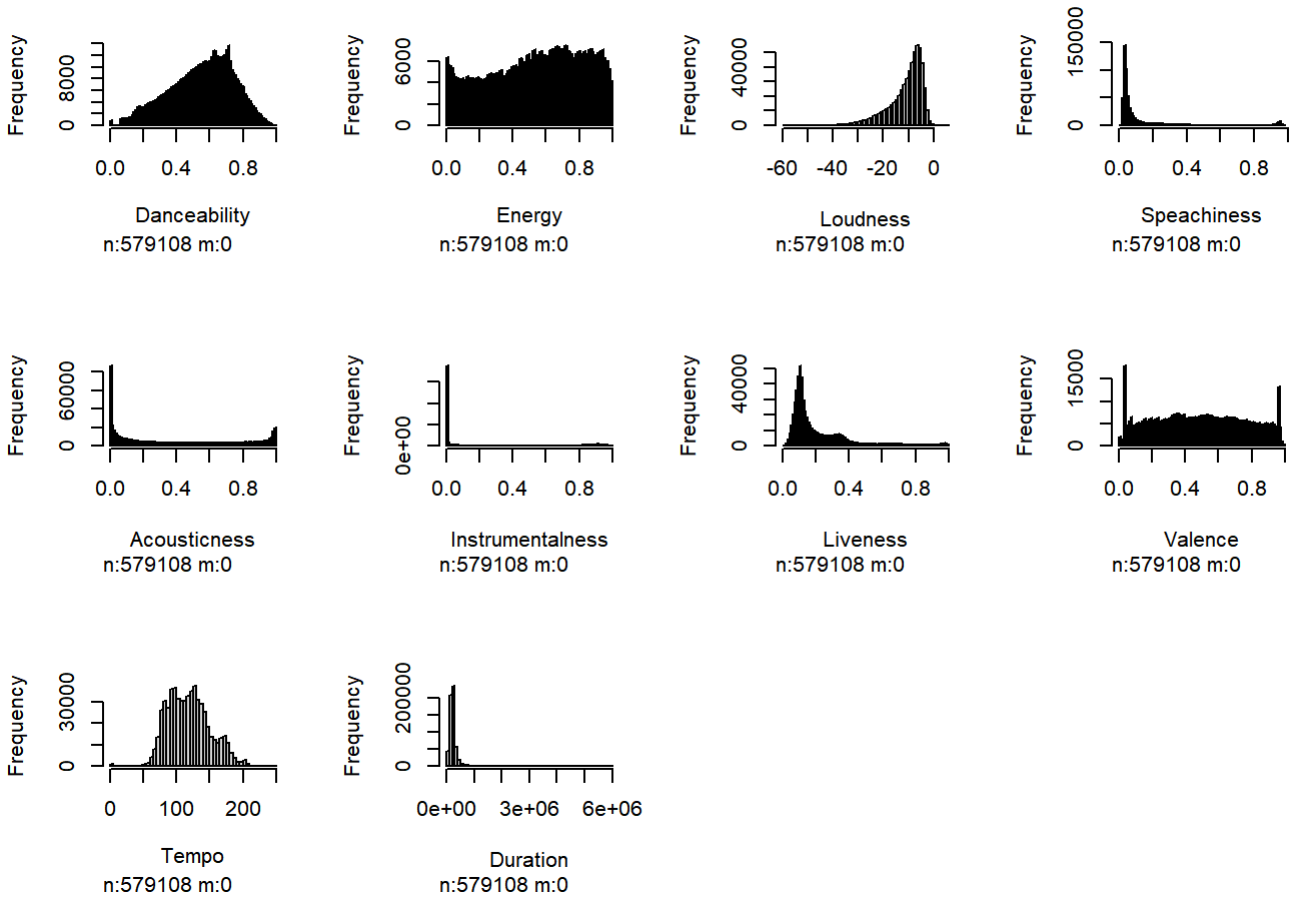
5

SongName	Artist	Popularity
Unholy (feat. Kim Petras)	c(""Sam Smith"", ""Kim Petras"")	100.00
Quevedo: Bzrp Music Sessions, Vol. 52	c(""Bizarrap"", ""Quevedo"")	97.00
La Bachata	Manuel Turizo	97.00
I'm Good (Blue)	c(""David Guetta"", ""Bebe Rexha"")	97.00
Tití Me Preguntó	Bad Bunny	96.00
Me Porto Bonito	c(""Bad Bunny"", ""Chencho Corleone"")	96.00

Acoustic Attributes

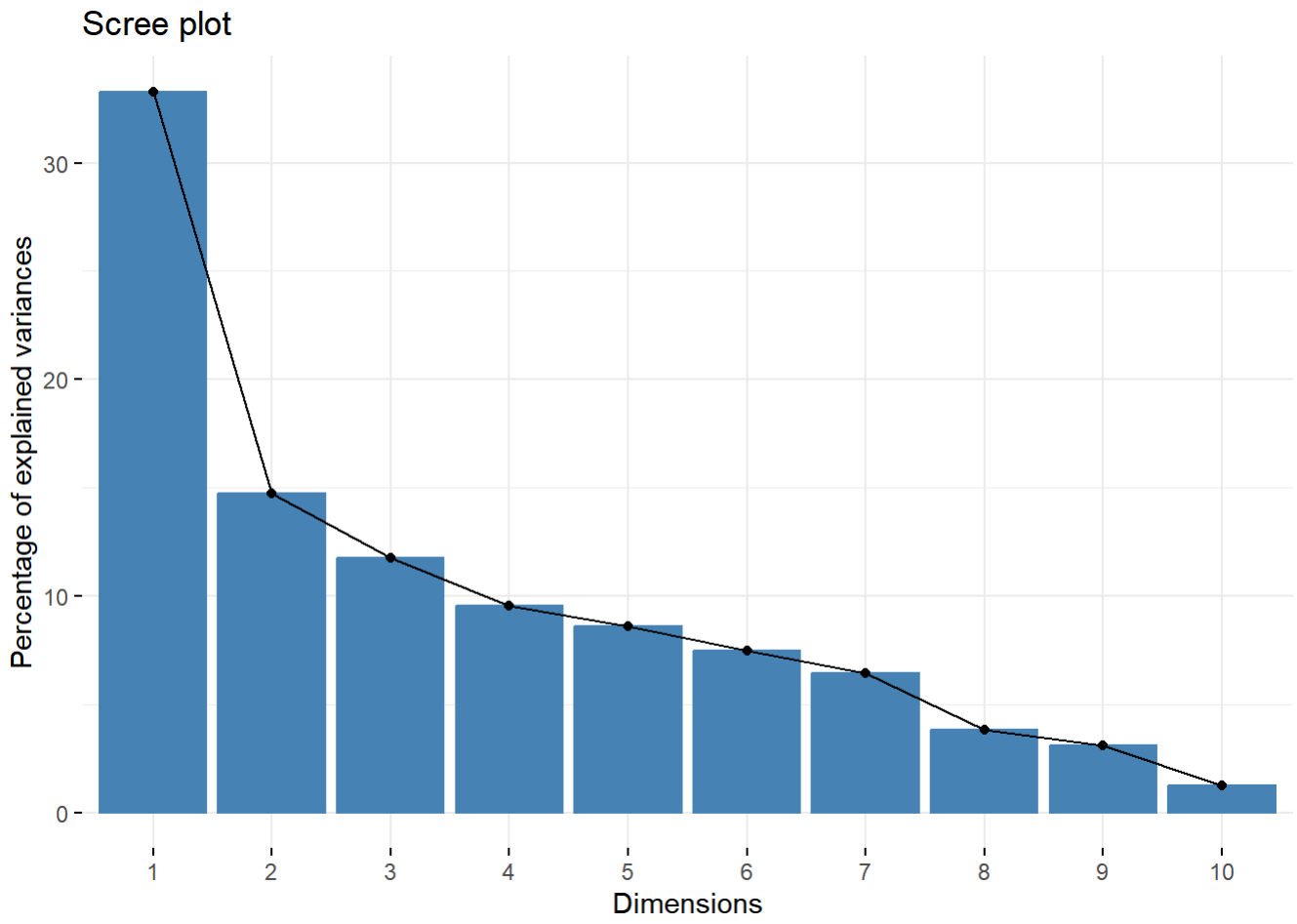
Distributions of acoustic variables (raw data):

```
hist.data.frame(songs_combined[, c(5:6, 8, 10:16)])
```



Principal Component Analysis

```
num_cols <- unlist(lapply(songs_combined, is.numeric))

pca <- PCA(songs_combined[, c(5:6, 8, 10:16)], graph = F, scale.unit = T)

fviz_eig(pca)
```


Scree plot
Percentage of explained variances
30
20
10
0
1
2
3
4
5
6
7
8
9
10
Dimensions

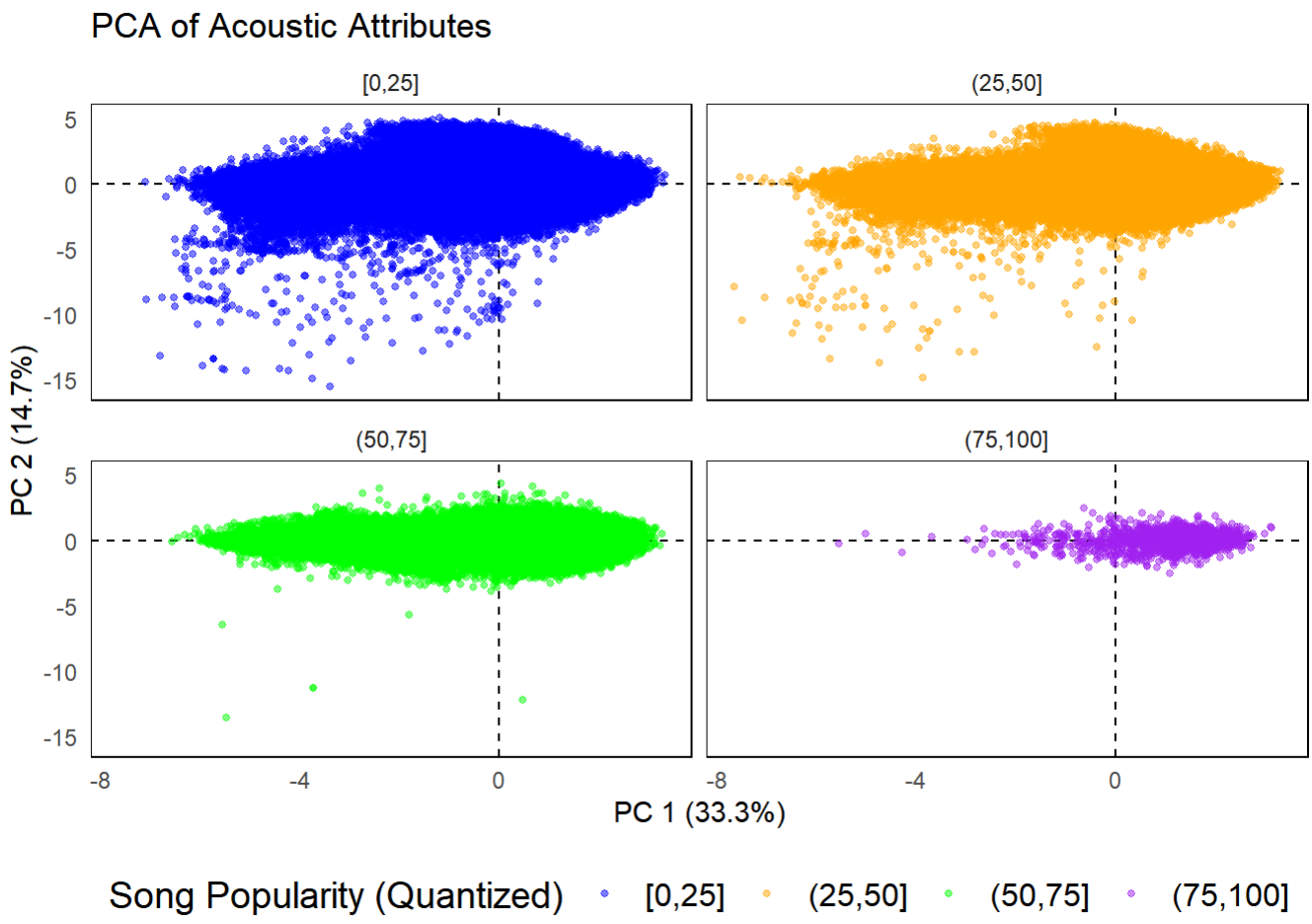
```
summary(pca)
```

```
##
## Call:
## PCA(X = songs_combined[, c(5:6, 8, 10:16)], scale.unit = T, graph = F)
##
##
## Eigenvalues
##                        Dim.1   Dim.2   Dim.3   Dim.4   Dim.5   Dim.6   Dim.7
## Variance               3.330   1.474   1.178   0.954   0.859   0.749   0.643
## % of var.             33.297  14.740  11.776   9.542   8.590   7.489   6.425
## Cumulative % of var.  33.297  48.037  59.813  69.355  77.945  85.434  91.859
##                        Dim.8   Dim.9  Dim.10
## Variance               0.380   0.309   0.126
## % of var.              3.797   3.086   1.258
## Cumulative % of var.  95.656  98.742 100.000
##
## Individuals (the 10 first)
##                       Dist    Dim.1    ctr   cos2    Dim.2    ctr   cos2
## 1                  |  2.126 |  1.460  0.000  0.471 |  0.697  0.000  0.108 |
## 2                  |  3.247 |  2.507  0.000  0.596 |  0.996  0.000  0.094 |
## 3                  |  2.653 |  1.777  0.000  0.449 |  0.888  0.000  0.112 |
## 4                  |  2.124 |  1.510  0.000  0.505 | -0.140  0.000  0.004 |
## 5                  |  3.727 |  1.899  0.000  0.260 | -0.042  0.000  0.000 |
## 6                  |  1.988 |  0.596  0.000  0.090 |  0.920  0.000  0.214 |
## 7                  |  2.062 |  0.080  0.000  0.002 |  0.343  0.000  0.028 |
## 8                  |  1.701 |  0.097  0.000  0.003 |  0.357  0.000  0.044 |
## 9                  |  3.132 | -0.300  0.000  0.009 |  1.181  0.000  0.142 |
## 10                 |  2.583 |  0.038  0.000  0.000 |  1.106  0.000  0.183 |
##                     Dim.3    ctr   cos2
## 1                   0.000  0.000  0.000 |
## 2                  -1.485  0.000  0.209 |
## 3                   0.573  0.000  0.047 |
## 4                  -0.540  0.000  0.065 |
## 5                   2.006  0.001  0.290 |
## 6                  -0.274  0.000  0.019 |
## 7                  -0.404  0.000  0.038 |
## 8                  -0.713  0.000  0.176 |
## 9                   1.429  0.000  0.208 |
## 10                 -0.151  0.000  0.003 |
##
## Variables
##                      Dim.1    ctr   cos2    Dim.2    ctr   cos2    Dim.3    ctr
## Danceability     |  0.598 10.726  0.357 |  0.556 20.990  0.309 | -0.204  3.543
## Energy           |  0.851 21.759  0.725 | -0.334  7.583  0.112 |  0.089  0.680
## Loudness         |  0.869 22.686  0.755 | -0.228  3.512  0.052 | -0.013  0.013
## Speachiness      |  0.052  0.081  0.003 |  0.663 29.849  0.440 |  0.501 21.355
## Acousticness     | -0.780 18.290  0.609 |  0.284  5.461  0.080 | -0.064  0.352
## Instrumentalness | -0.617 11.429  0.381 | -0.290  5.715  0.084 | -0.177  2.657
## Liveness         |  0.156  0.734  0.024 | -0.070  0.331  0.005 |  0.818 56.769
## Valence          |  0.605 10.976  0.365 |  0.345  8.055  0.119 | -0.302  7.723
## Tempo            |  0.315  2.978  0.099 | -0.275  5.149  0.076 | -0.154  2.018
## Duration         | -0.106  0.340  0.011 | -0.444 13.356  0.197 |  0.240  4.891
##                     cos2
```

```
## Danceability      0.042 |
## Energy            0.008 |
## Loudness          0.000 |
## Speachiness       0.251 |
## Acousticness      0.004 |
## Instrumentalness  0.031 |
## Liveness          0.669 |
## Valence           0.091 |
## Tempo             0.024 |
## Duration          0.058 |
```

```
# summary(pca)
```

```
# format pca data for plotting

dat_pca <- data.frame(pca$ind$coord[, 1], pca$ind$coord[, 2])

names(dat_pca) <- c("PC1", "PC2")

pca.vars <- pca$var$coord %>%
    data.frame

pca.vars$vars <- rownames(pca.vars)

pca.vars.m <- melt(pca.vars, id.vars = "vars")

dat_pca$Popularity_quantized <- factor(songs_combined$Popularity_quantized)

dat_pca <- dat_pca %>%
    na.omit()

ggplot(data = dat_pca, aes(x = PC1, y = PC2, color = Popularity_quantized)) + scale_color_manual
(values = c("blue",
    "orange", "green", "purple")) + geom_hline(yintercept = 0, lty = 2) + geom_vline(xintercept
= 0,
    lty = 2) + guides(color = guide_legend(title = "Song Popularity (Quantized)")) +
    scale_shape_manual(values = c(15, 16, 16, 17, 18)) + geom_point(alpha = 0.5,
    size = 1) + xlab("PC 1 (33.3%)") + ylab("PC 2 (14.7%)") + facet_wrap(vars(Popularity_quantiz
ed)) +
    theme_minimal() + theme(panel.grid = element_blank(), panel.border = element_rect(fill = "tr
ansparent")) +
    theme(legend.position = "bottom") + theme(legend.title = element_text(size = 14),
    legend.text = element_text(size = 12)) + ggtitle("PCA of Acoustic Attributes")
```



Because there are so many datapoints, the PCA plots are faceted by the quantized popularity indices.

Interactive Shiny PCA plot:

```
fluidRow(style = "padding-bottom: 20px;", column(4, selectInput("group", "Popularity Index (Quan
tized) Group",
    dat_pca$Popularity_quantized)), column(4, selectInput("color", "Group Color",
    c("blue", "orange", "green", "purple"))))
```

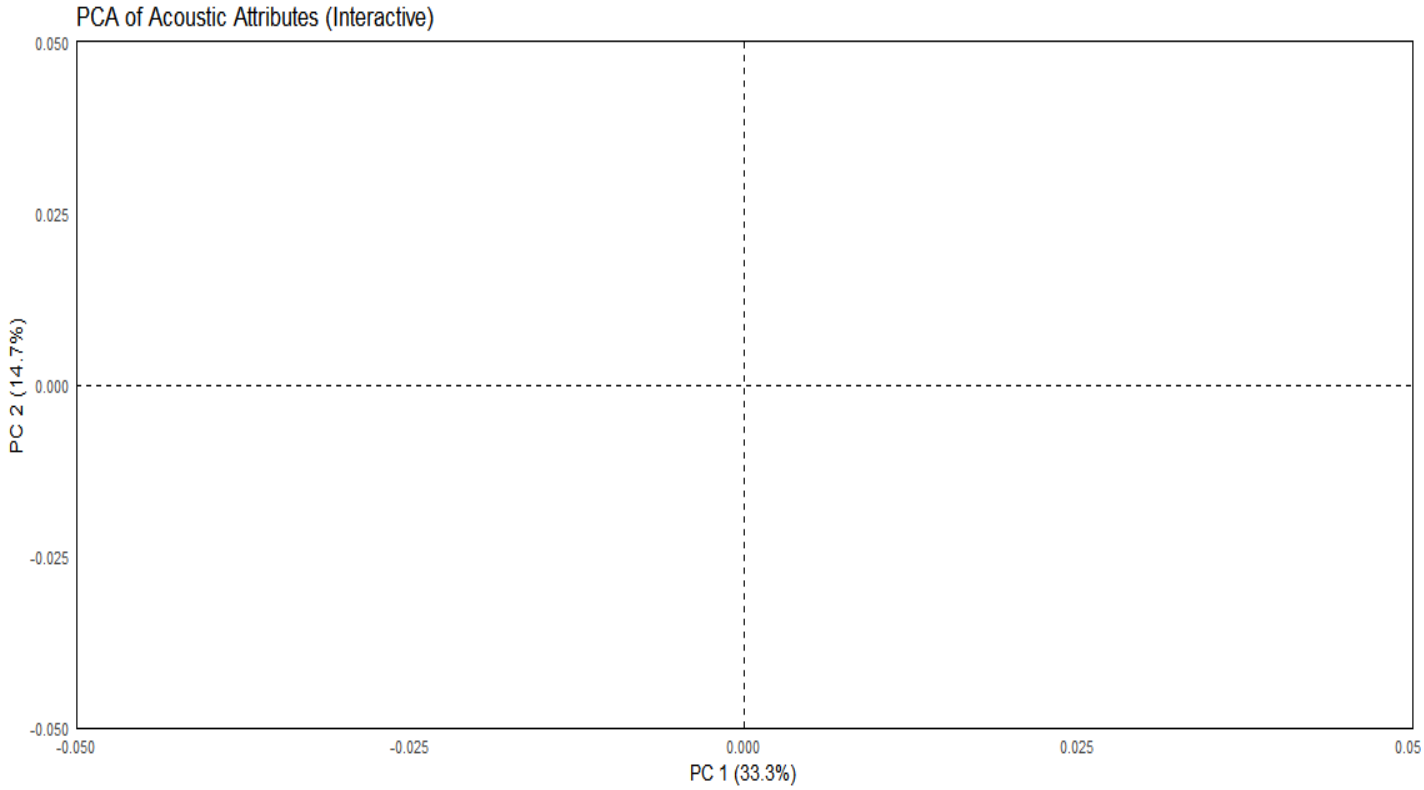
```
## Warning: The select input "group" contains a large number of options; consider
## using server-side selectize for massively improved performance. See the Details
## section of the ?selectizeInput help topic.
```

Popularity Index (Quantized) Group

Group Color

blue

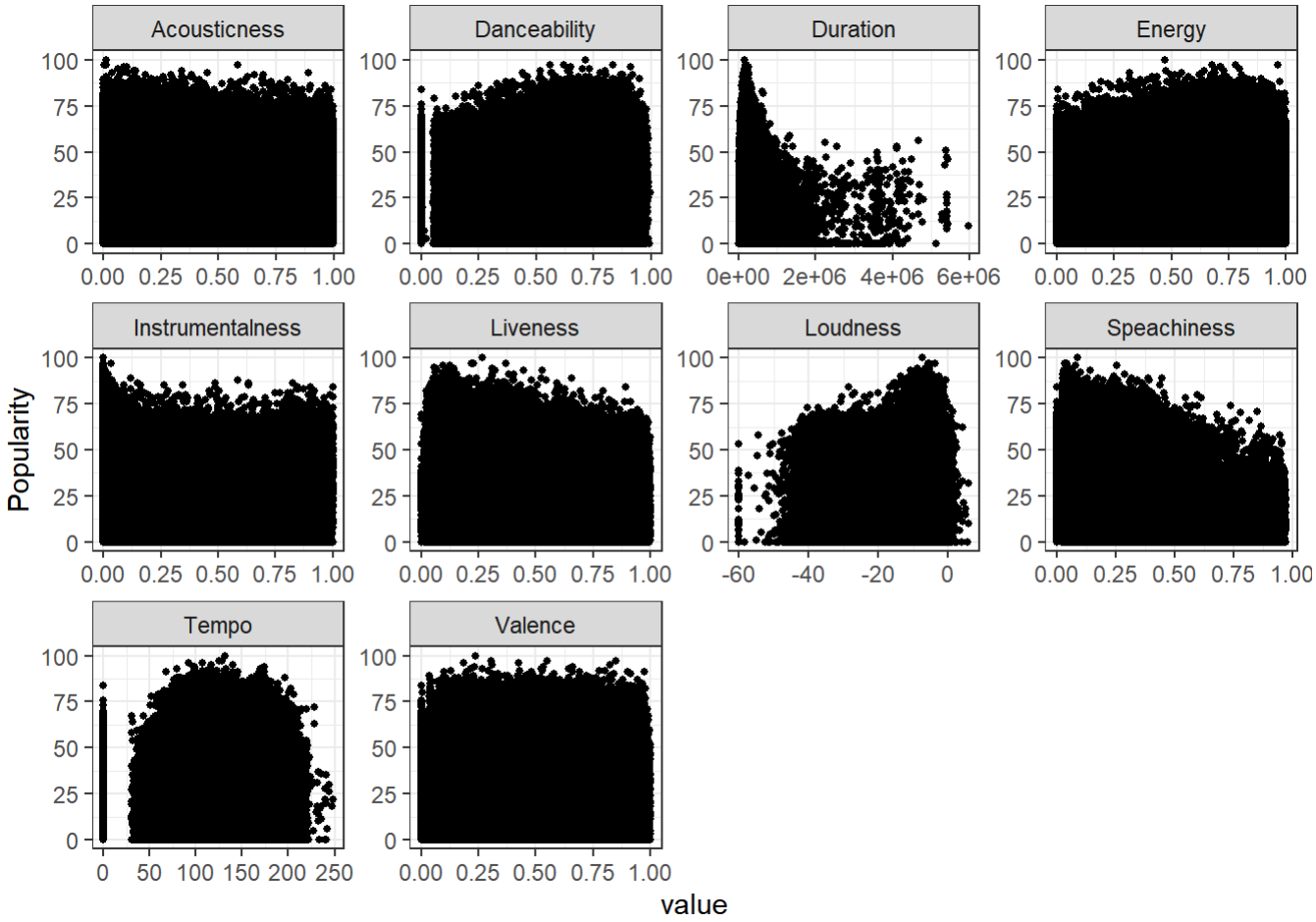
```
renderPlot({
    ggplot(data = dat_pca[dat_pca$Popularity_quantized == input$group, ], aes(x = PC1,
        y = PC2, color = Popularity_quantized)) + geom_point(alpha = 0.5, size = 1) +
        scale_color_manual(values = input$color) + geom_hline(yintercept = 0, lty = 2) +
        geom_vline(xintercept = 0, lty = 2) + scale_shape_manual(values = c(15, 16,
        16, 17, 18)) + xlab("PC 1 (33.3%)") + ylab("PC 2 (14.7%)") + theme_minimal() +
        theme(panel.grid = element_blank(), panel.border = element_rect(fill = "transparent")) +
        theme(legend.position = "bottom") + theme(legend.title = element_text(size = 14),
        legend.text = element_text(size = 12)) + ggtitle("PCA of Acoustic Attributes (Interactiv
e)")


})
```



Correlation Analysis

Here we sought to determine if sonic attributes are correlated with song popularity.

```
# this takes a couple of minutes to run

songs_combined[, c(4, 5:6, 8, 10:16)] %>%
    gather(-Popularity, key = "var", value = "value") %>%
    ggplot(aes(x = value, y = Popularity)) + geom_point(size = 1) + facet_wrap(~var,
    scales = "free") + theme_bw()
```



Most acoustic variables do not appear to be correlated with song popularity. However a few possible relationships are apparent:

- Danceability is positively correlated with popularity
- Duration is negatively correlated with popularity
- Loudness is positively correlated with popularity
- Speachiness is negatively correlated with popularity

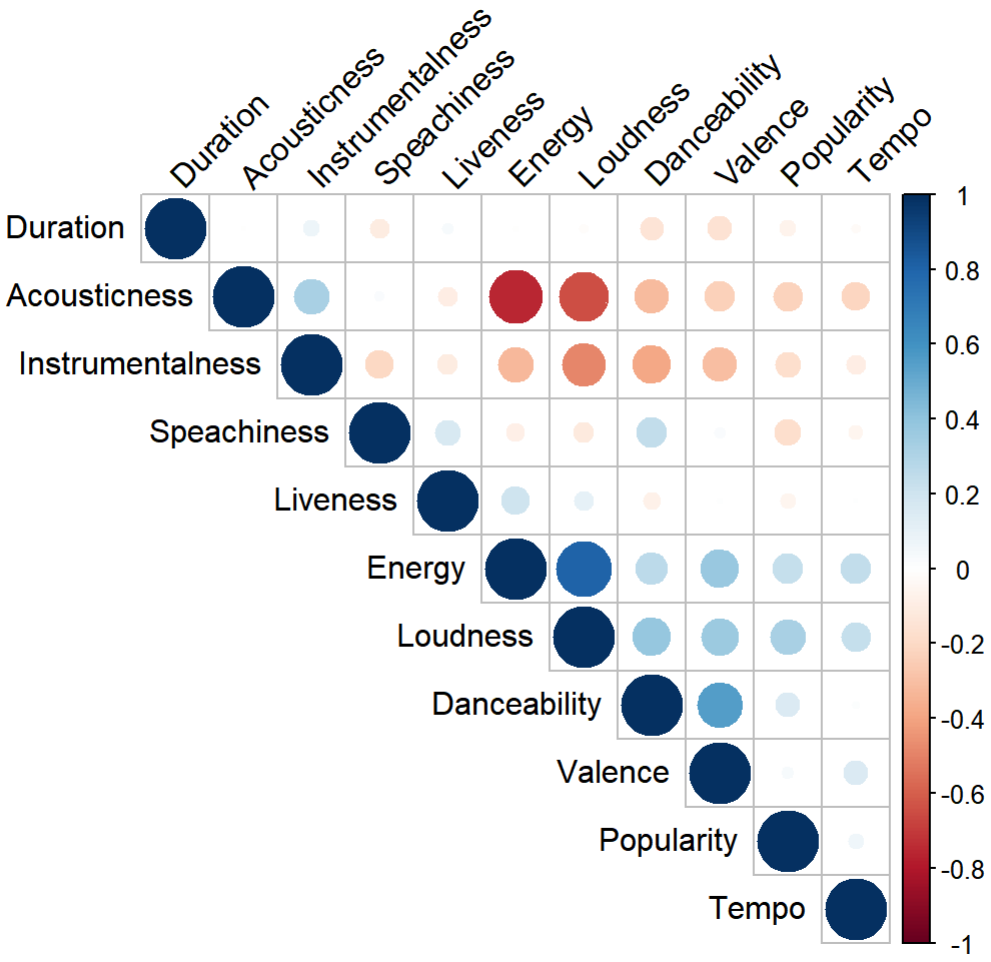
Pearson Correlation Coefficients:

```
cor.res <- songs_combined[, c(4, 5:6, 8, 10:16)] %>%
    cor_test(Popularity, method = "pearson")

cor.res$FDR <- qvalue::qvalue(cor.res$p, pi0 = 1)$qvalues

cor.res
```

```
# cor.res[cor.res$FDR<0.05, c(1:3, 5, 9)] %>% arrange(cor)
```

Plot correlations among all acoustic variables and popularity (Pearson)

```
all_cor <- cor(songs_combined[, c(4, 5:6, 8, 10:16)], method = "pearson")

corrplot(all_cor, type = "upper", order = "hclust", tl.col = "black", tl.srt = 45)
```


Duration
Acousticness
Instrumentalness
Speachiness
Liveness
Energy
Loudness
Danceability
Valence
Popularity
Tempo
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1

Modeling Analysis

Using the whole dataset, the models take too long to run. With greater computational resources, the models can be run on the entire dataset. For the purposes of this project, the models were run on a smaller subset of rows selected randomly from the whole data.

```
set.seed(1221)
obs <- sample(1:nrow(songs_combined), 80000)
subdata <- songs_combined[obs, ]
```

The data was then divided into a training set and a testing set in an 80%/20% split, respectively.

Regression models

For the regression analysis, the response variable is *Popularity*. The predictors are listed as follows:

- acousticness
- danceability
- durations
- energy
- instrumentalness
- key
- liveness
- loudness
- mode
- speechiness
- tempo
- valence

The following models were processed using the predictors listed above in the training data, and model predictions were evaluated against the testing data. The metrics used to evaluate model fit were root mean squared error (RMSE) and R^2. A good model fit will have a lower RMSE and a higher R^2 value. Model parameters were then tuned to try to optimize the model to obtain a better fit.

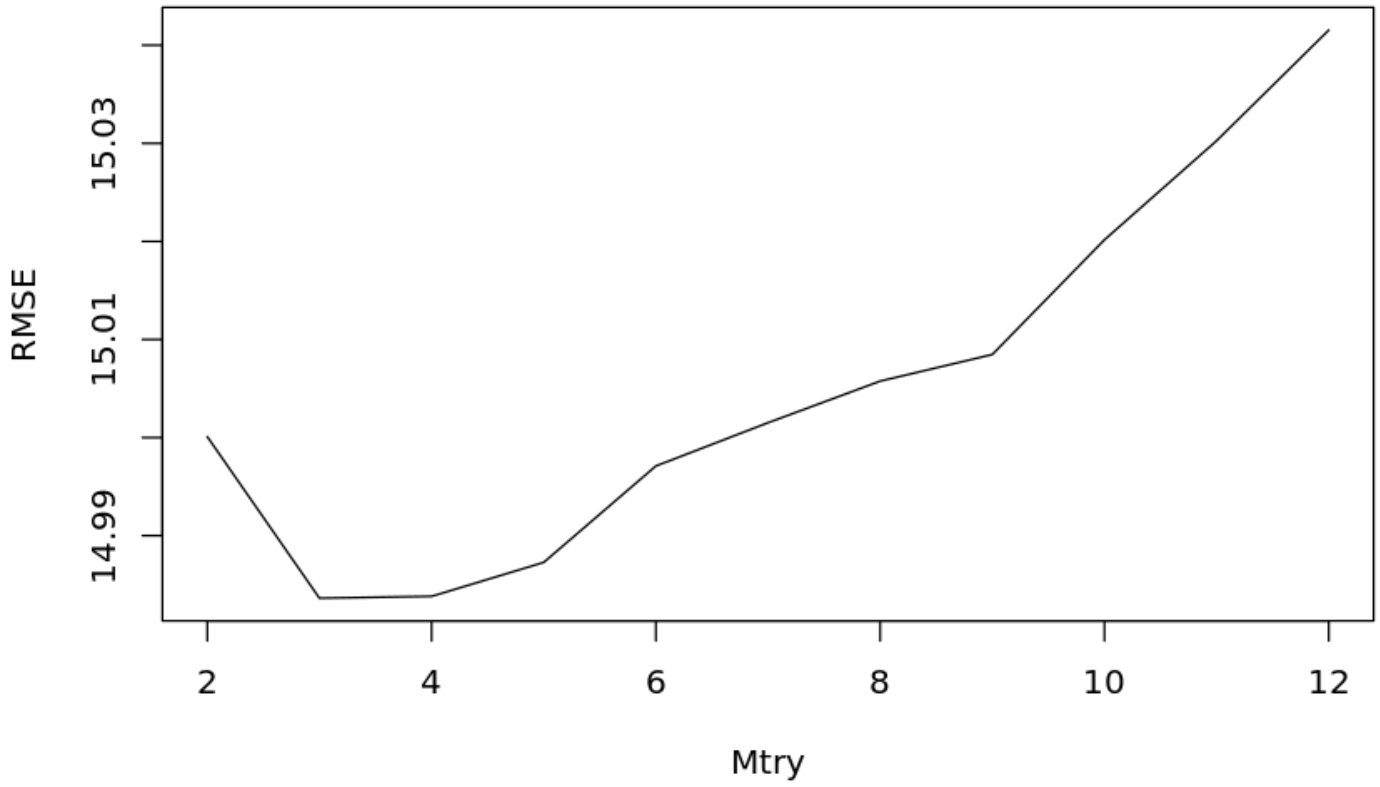
```
reg_train <- train[, 4:16]
reg_test <- test[, 4:16]
reg_test_x <- reg_test[, -1]
reg_test_y <- as.matrix(reg_test[, 1], nrow = nrow(reg_test[, 1]), ncol = 1)
```

1) Random Forest

The random forest model is an ensemble learning method that constructs decision trees with a random selection of a subset of the predictors. The number of predictors selected for this subset is a parameter in this model known as mtry. For regression, the output of the random forest is the number selected by the majority of the trees based on the predictor values. The initial random forest model that was run included the default mtry parameter = 3 using 500 trees. The resulting RMSE and R^2 values evaluated against the testing data were 14.99 and 0.254, respectively.

The random forest model was then tuned via 5-fold cross validation to determine the optimal value of mtry. Since there are 12 predictors, the model was re-run using mtry values ranging from 2-12, and the model that had the lowest RMSE was selected as the optimal model. The model with mtry = 3 had the lowest RMSE, so our initial model was already the optimal model.

```r
# Fit random forest regression model
rf_reg_fit <- ranger(Popularity ~ ., data = reg_train, importance = "impurity")
rf_reg_fit
# Function that computes RMSE and R^2 from true and predicted values
eval_results <- function(true, predicted, df) {
    SSE <- sum((predicted - true)^2)
    SST <- sum((true - mean(true))^2)
    R_square <- 1 - SSE/SST
    RMSE <- sqrt(SSE/nrow(df))
    # Summarize model performance metrics
    data.frame(RMSE <- RMSE, Rsquare <- R_square)
}
# Prediction and evaluation on test data
rf_reg_pred_y <- predict(rf_reg_fit, reg_test_x)

rf_reg_res <- eval_results(as.matrix(reg_test[, 1]), rf_reg_pred_y$predictions, as.matrix(reg_te
st))

# Get variable importance from the model fit
importance <- as.vector(rf_reg_fit$variable.importance)
variable <- as.vector(colnames(reg_train)[2:13])
var_imp <- cbind(variable, importance)
var_imp <- as.data.frame(var_imp)
var_imp$importance <- as.numeric(var_imp$importance)
var_imp

ggplot(var_imp, aes(x = reorder(variable, importance), y = importance, fill = importance)) +
    geom_bar(stat = "identity", position = "dodge") + coord_flip() + ylab("Variable Importance")
+
    xlab("") + ggtitle("Variable Importance Plot")

# Tune mtry parameter (# of vars to randomly sample as candidates at each
# split)
rf_grid <- expand.grid(mtry = c(2:12), splitrule = "variance", min.node.size = 5)

fitControl <- trainControl(method = "CV", number = 5, verboseIter = TRUE)

class(reg_train$Popularity)

## below does not work

rf_reg_tune <- train(x = reg_train[, 2:13], y = reg_train[, 1], method = "ranger",
    tuneGrid = rf_grid, metric = "RMSE", trControl = fitControl)
rf_reg_tune

plot(rf_reg_tune$results[, 1], rf_reg_tune$results[, 4], xlab = "Mtry", ylab = "RMSE",
    type = "l")
```

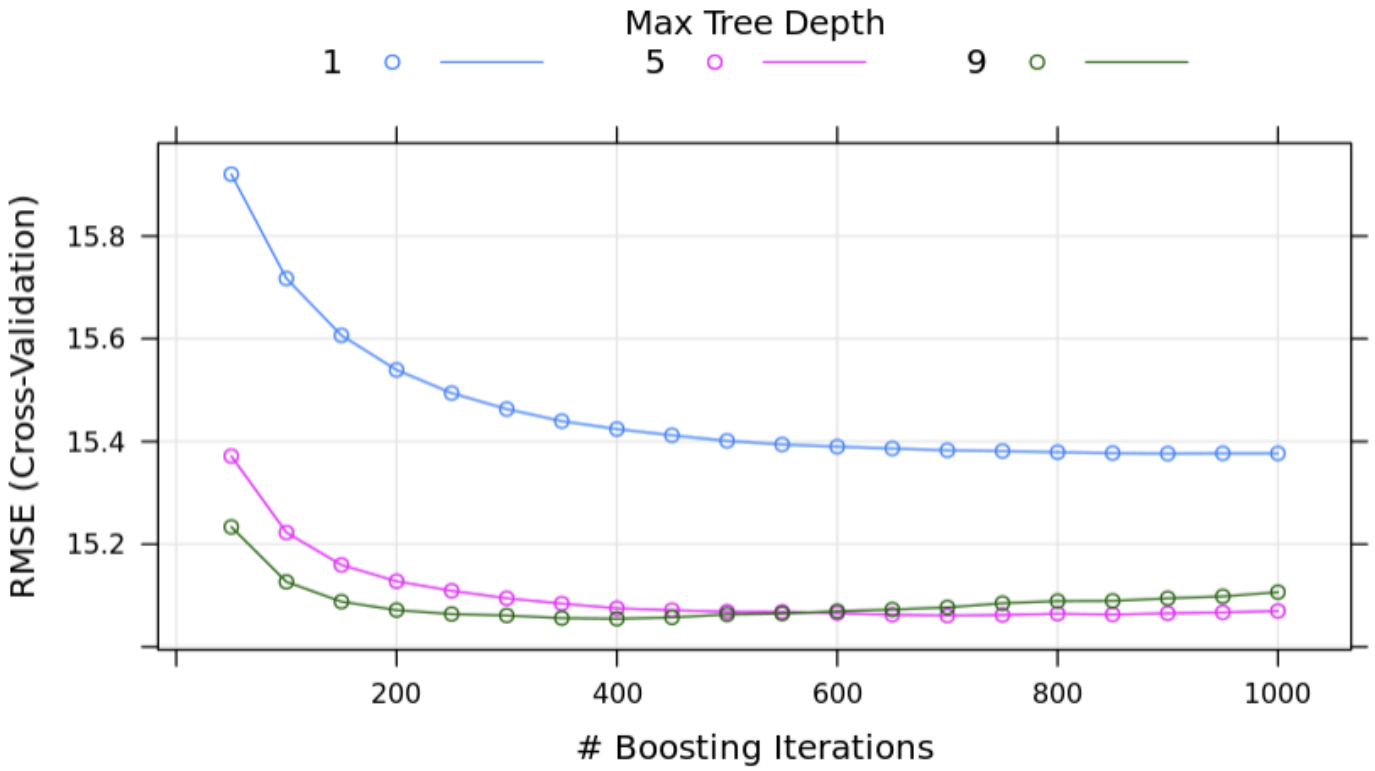


2) Gradient Boosting Machine

Gradient boosting machines (GBM) are another type of ensemble machine learning algorithms that can be used for regression. Gradient boosting is an optimization problem with the goal of minimizing the model's loss by adding one weak learner at a time using a gradient descent algorithm. The weak learners in gradient boosting are decision trees. The decision trees are built such that their split values are determined based on minimizing the loss. The GBM that was run included the default max tree depth parameter = 3 and number of iterations parameter = 150. The resulting RMSE and R^2 values evaluated against the testing data were 15.28 and 0.225, respectively.

The GBM was then tuned via 5-fold cross validation to determine the optimal parameter values. The max tree depth values that were tested were 1, 5, and 9, and the number of iterations tested ranged from 50 to 1000 by increments of 50. The model with max tree depth = 9 and number of iterations = 400 had the lowest RMSE, so the GBM was re-run using these parameters. The resulting RMSE and R^2 values were 15.07 and 0.245, respectively.

```
# Fit GBM
gbm_reg_fit <- train(Popularity ~ ., data = reg_train, method = "gbm", trControl = fitControl,
    verbose = FALSE)

gbm_reg_fit
# Prediction and evaluation on test data
gbm_reg_pred_y <- predict(gbm_reg_fit, reg_test_x)

gbm_reg_res <- eval_results(as.matrix(reg_test[, 1]), gbm_reg_pred_y, as.matrix(reg_test))

# Tune interaction.depth (tree complexity) and n.trees (# of iterations)
# parameters
gbm_grid <- expand.grid(interaction.depth = c(1, 5, 9), n.trees = (1:20) * 50, shrinkage = 0.1,
    n.minobsinnode = 20)

gbm_reg_tune <- train(Popularity ~ ., data = reg_train, method = "gbm", trControl = fitControl,
    verbose = FALSE, tuneGrid = gbm_grid)

gbm_reg_tune

plot(gbm_reg_tune)

gbm_grid_tune <- expand.grid(interaction.depth = 9, n.trees = 400, shrinkage = 0.1,
    n.minobsinnode = 20)

gbm_reg_tuned_fit <- train(Popularity ~ ., data = reg_train, method = "gbm", trControl = fitCont
rol,
    verbose = FALSE, tuneGrid = gbm_grid_tune)

gbm_reg_tuned_fit
# Tuned prediction and evaluation on test data
gbm_reg_tuned_pred_y <- predict(gbm_reg_tuned_fit, reg_test_x)

gbm_reg_tuned_res <- eval_results(as.matrix(reg_test[, 1]), gbm_reg_tuned_pred_y,
    as.matrix(reg_test))
```

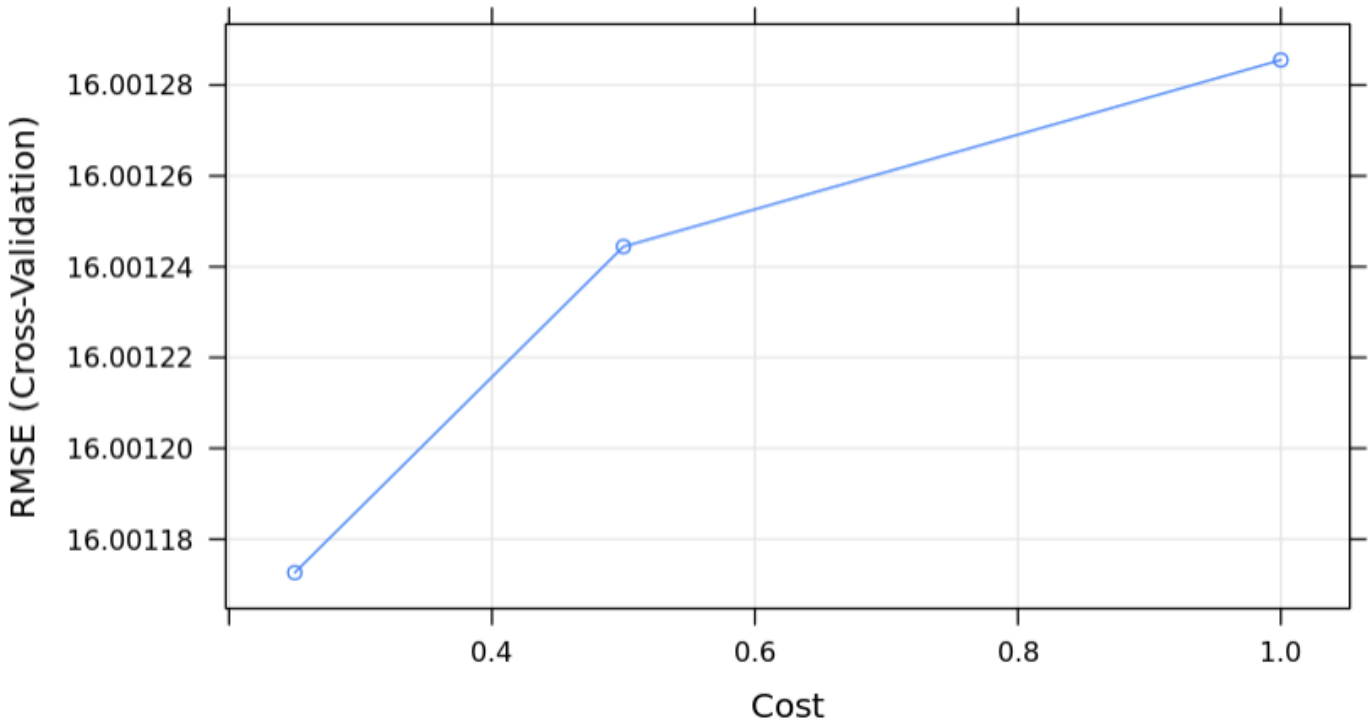


3) Support Vector Regression

Support vector machines (SVM) are large margin classifiers. Support vector regression is a machine learning algorithm that finds an appropriate line or hyperplane that fits the data within a specified threshold. Here, we are fitting a linear kernel. The support vector regression model that was run included the default cost parameter = 1. This cost parameter can adjust for how much the large margin classification should be. The resulting RMSE and R^2 values were 15.98 and 0.152, respectively.

The support vector regression was then tuned via 5-fold cross validation to determine the optimal cost parameter. The cost values that were tested were 0.25, 0.5, and 1. The model with cost = 0.25 had the lowest RMSE, so the support vector regression was re-run using this parameter value. The resulting RMSE and R^2 values were 15.98 and 0.152, respectively.

```
# Fit support vector regression
svm_reg_fit <- train(Popularity ~ ., data = reg_train, method = "svmLinear", trControl = fitCont
rol,
    preProcess = c("center", "scale"))

svm_reg_fit

# Prediction and evaluation on test data
svm_reg_pred_y <- predict(svm_reg_fit, reg_test_x)

svm_reg_res <- eval_results(as.matrix(reg_test[, 1]), svm_reg_pred_y, as.matrix(reg_test))

# Tune C parameter (cost - penalty to model for making error)
svm_grid <- expand.grid(C = c(0.25, 0.5, 1))

svm_reg_tune <- train(Popularity ~ ., data = reg_train, method = "svmLinear", trControl = fitCon
trol,
    preProcess = c("center", "scale"), tuneGrid = svm_grid)
svm_reg_tune

plot(svm_reg_tune)

svm_grid_tune <- expand.grid(C = 0.25)

svm_reg_tuned_fit <- train(Popularity ~ ., data = reg_train, method = "svmLinear",
    trControl = fitControl, preProcess = c("center", "scale"), tuneGrid = svm_grid_tune)

svm_reg_tuned_fit
# Tuned prediction and evaluation on test data
svm_reg_tuned_pred_y <- predict(svm_reg_tuned_fit, reg_test_x)

svm_reg_tuned_res <- eval_results(as.matrix(reg_test[, 1]), svm_reg_tuned_pred_y,
    as.matrix(reg_test))
```


RMSE (Cross-Validation)
16.00128
16.00126
16.00124
16.00122
16.00120
16.00118
0.4
0.6
0.8
1.0
Cost

Classification Models

```
cl_train <- train[, 5:17]
cl_test <- test[, 5:17]
cl_test_x <- cl_test[, -13]
cl_test_y <- as.matrix(cl_test[, 13], nrow = nrow(cl_test[, 13]), ncol = 1)
```

For the classification analysis, the response variable is *Popularity_quantized*. The predictors remain the same as regression.

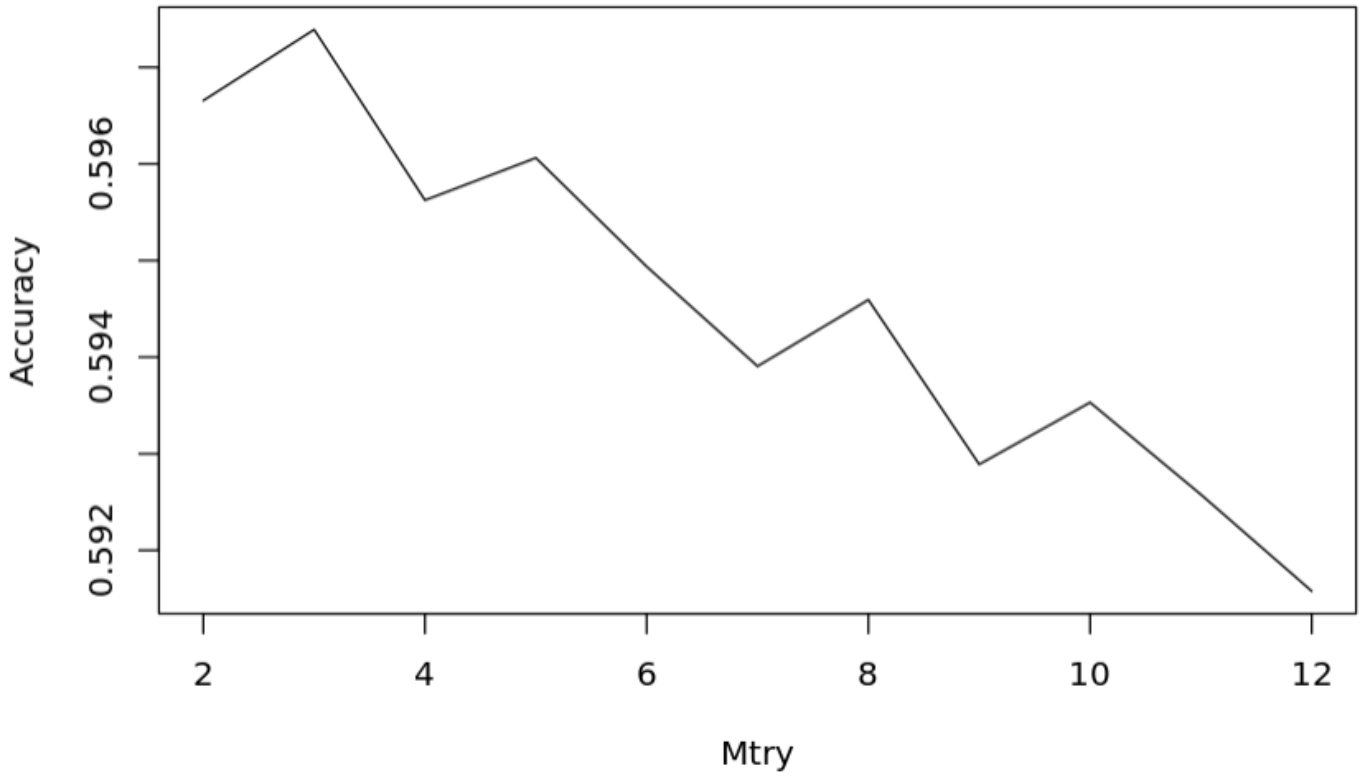
The metric used to evaluate model fit was accuracy. A good model fit will have a higher accuracy value. Similar to regression, model parameters were then tuned to try to optimize the model to obtain a better fit.

1) Random Forest

The random forest model for classification is similar to the random forest model for regression except the output of the random forest is the class selected by the majority of the trees based on the predictor values. The initial random forest model was run, and the resulting accuracy evaluated against the testing data was 0.592.

The random forest model was then tuned similar to regression, and the model with mtry = 3 had the highest accuracy, so our initial model was already the optimal model again.

```
# Fit random forest regression model
rf_cl_fit <- ranger(Popularity_quantized ~ ., data = cl_train, importance = "impurity")
rf_cl_fit
# Prediction and evaluation on test data
rf_cl_pred_y <- predict(rf_cl_fit, cl_test_x)
table <- table(rf_cl_pred_y$predictions, cl_test_y)[1:4, c(4, 1:3)]
table
rf_table <- confusionMatrix(table)
# Get variable importance from the model fit
importance <- as.vector(rf_cl_fit$variable.importance)
variable <- as.vector(colnames(cl_train)[1:12])
var_imp <- cbind(variable, importance)
var_imp <- as.data.frame(var_imp)
var_imp$importance <- as.numeric(var_imp$importance)
var_imp
ggplot(var_imp, aes(x = reorder(variable, importance), y = importance, fill = importance)) +
    geom_bar(stat = "identity", position = "dodge") + coord_flip() + ylab("Variable Importance")
+
    xlab("") + ggtitle("Variable Importance Plot")
# Tune mtry parameter (# of vars to randomly sample as candidates at each
# split)
rf_grid <- expand.grid(mtry = c(2:12), splitrule = "gini", min.node.size = 1)
fitControl <- trainControl(method = "CV", number = 5, verboseIter = TRUE)
rf_cl_tune <- train(x = cl_train[, 1:12], y = cl_train[, 13], method = "ranger",
    tuneGrid = rf_grid, metric = "Accuracy", trControl = fitControl)
rf_cl_tune
plot(rf_cl_tune$results[, 1], rf_cl_tune$results[, 4], xlab = "Mtry", ylab = "Accuracy",
    type = "l")
# Optimal model is mtry = 3 which is the random forest classification model
# that was already run
```

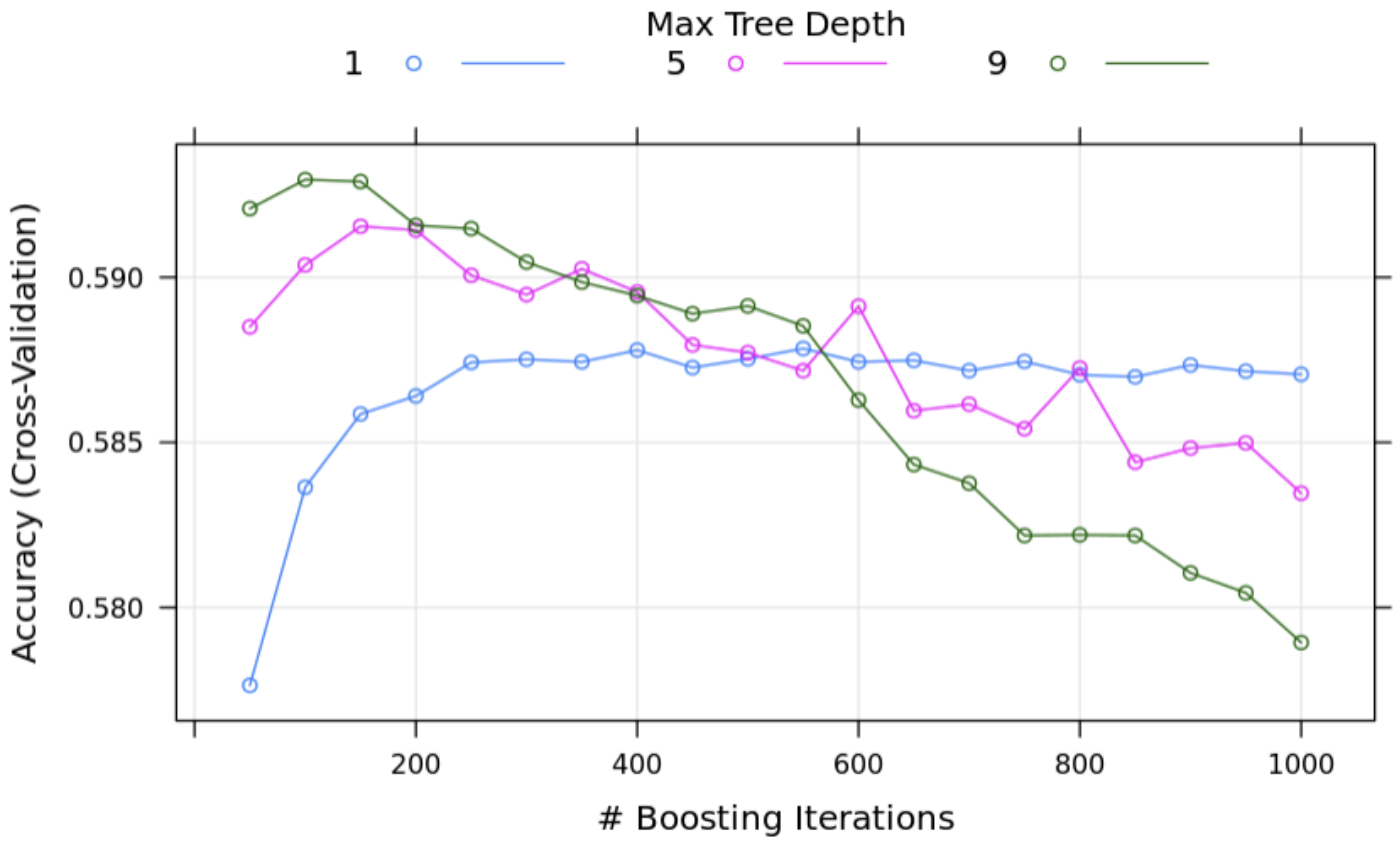


2) Gradient Boosting Machine

GBM can also be used for classification. The initial GBM was run, and the resulting accuracy evaluated against the testing data was 0.584.

The GBM was then tuned similar to regression, and the model with max tree depth = 9 and number of iterations = 100 had the lowest accuracy, so the GBM was re-run using these parameters. The resulting accuracy was 0.590.

```
# Fit GBM
gbm_cl_fit <- train(Popularity_quantized ~ ., data = cl_train, method = "gbm", trControl = fitCo
ntrol,
    verbose = FALSE)
gbm_cl_fit
# Prediction and evaluation on test data
gbm_cl_pred_y <- predict(gbm_cl_fit, cl_test_x)
table <- table(gbm_cl_pred_y, cl_test_y)[1:4, c(4, 1:3)]
table
gbm_table <- confusionMatrix(table)
# Tune interaction.depth (tree complexity) and n.trees (# of iterations)
# parameters
gbm_grid <- expand.grid(interaction.depth = c(1, 5, 9), n.trees = (1:20) * 50, shrinkage = 0.1,
    n.minobsinnode = 20)
gbm_cl_tune <- train(Popularity_quantized ~ ., data = cl_train, method = "gbm", trControl = fitC
ontrol,
    verbose = FALSE, tuneGrid = gbm_grid)
gbm_cl_tune
plot(gbm_cl_tune)
gbm_grid_tune <- expand.grid(interaction.depth = 9, n.trees = 100, shrinkage = 0.1,
    n.minobsinnode = 20)
gbm_cl_tuned_fit <- train(Popularity_quantized ~ ., data = cl_train, method = "gbm",
    trControl = fitControl, verbose = FALSE, tuneGrid = gbm_grid_tune)
gbm_cl_tuned_fit
# Tuned prediction and evaluation on test data
gbm_cl_tuned_pred_y <- predict(gbm_cl_tuned_fit, cl_test_x)
table <- table(gbm_cl_tuned_pred_y, cl_test_y)[1:4, c(4, 1:3)]
table
gbm_tuned_table <- confusionMatrix(table)
```

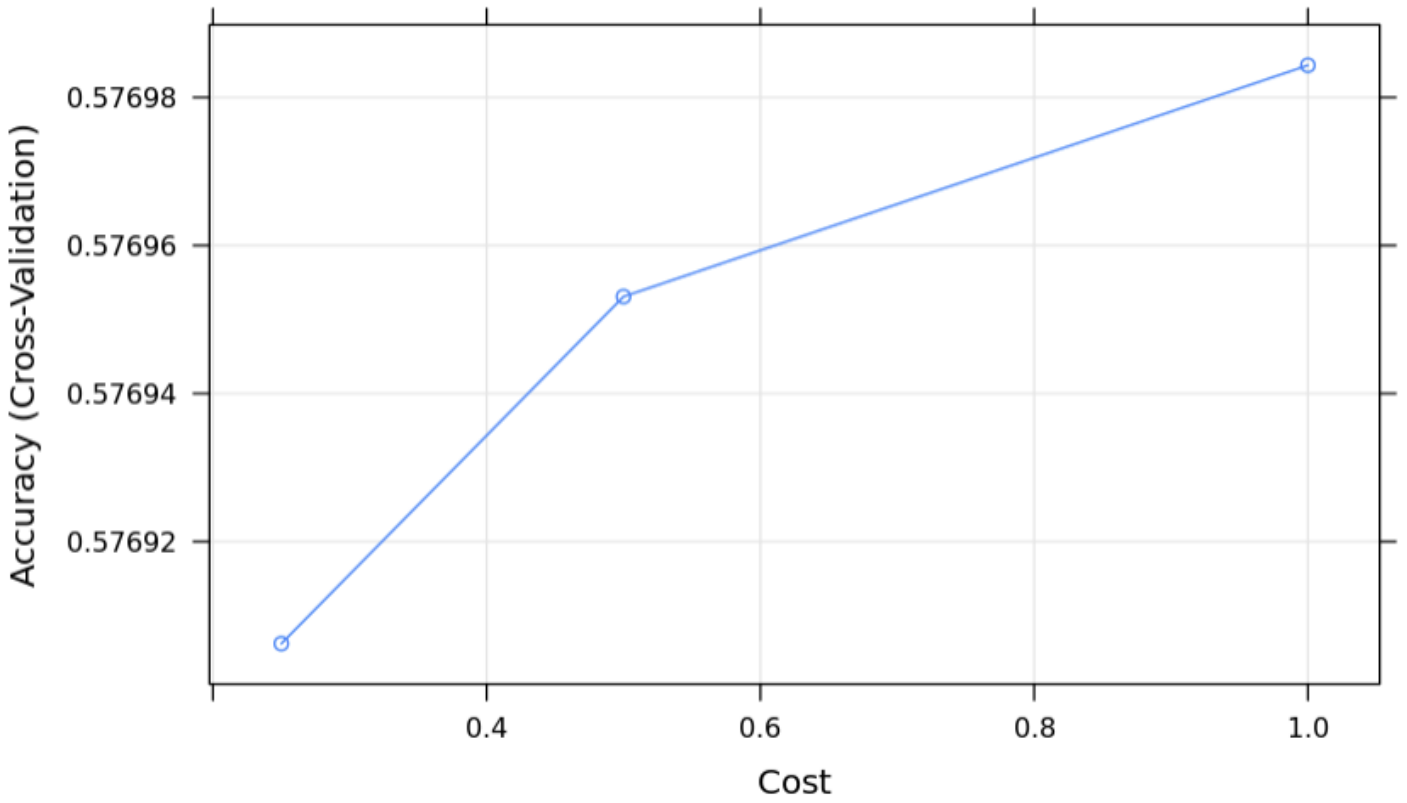


3) Support Vector Machine

SVM was used for classification. Here it tries to find an appropriate margin that separates the classes and also reduces the risk of error on the data. A linear classifier was used similar to regression. The initial SVM model was run, and the resulting accuracy evaluated against the testing data was 0.572.

The SVM model was then tuned similar to regression, and the model with cost = 1 had the highest accuracy, so our initial model was already the optimal model.

```
# Fit support vector regression
svm_cl_fit <- train(Popularity_quantized ~ ., data = cl_train, method = "svmLinear",
    trControl = fitControl, preProcess = c("center", "scale"))
svm_cl_fit
# Prediction and evaluation on test data
svm_cl_pred_y <- predict(svm_cl_fit, cl_test_x)
table <- table(svm_cl_pred_y, cl_test_y)[1:4, c(4, 1:3)]
table
svm_table <- confusionMatrix(table)
# Tune C parameter (cost - penalty to model for making error)
svm_grid <- expand.grid(C = c(0.25, 0.5, 1))
svm_cl_tune <- train(Popularity_quantized ~ ., data = cl_train, method = "svmLinear",
    trControl = fitControl, preProcess = c("center", "scale"), tuneGrid = svm_grid)
svm_cl_tune
plot(svm_cl_tune)
# Optimal model is C = 1 which is the svm classification model that was already
# run
```


Accuracy (Cross-Validation)
0.57698
0.57696
0.57694
0.57692
0.4
0.6
0.8
1.0
Cost

Results

Below is a summary table of the regression and classification model metrics. The random forest model appears to perform the best for both regression and classification because it has the lowest RMSE and highest accuracy, respectively. This means it is a good model to predict both a popularity score as well as a popularity class given these predictors.

On the other hand, the SVM model appears to perform the worst for both regression and classification. This suggests a linear classifier may not be good enough to predict popularity. However, the model metrics are still pretty close to each other and aren't too far apart. Given more time, we could look into other possible kernels for SVM such as a polynomial or radial kernel, for example, to better match the shape of the data. We could also look into tuning other parameters in the random forest model such as the number of trees parameter to further improve the model fit.

```
# Make a table of regression results
tab <- matrix(c(round(rf_reg_res, 3), round(rf_reg_res, 3), round(gbm_reg_res, 3),
    round(gbm_reg_tuned_res, 3), round(svm_reg_res, 3), round(svm_reg_tuned_res,
        3)), nrow = 6, ncol = 2, byrow = TRUE)
rownames(tab) <- c("Random Forest", "Random Forest Tuned", "GBM", "GBM Tuned", "SVM",
    "SVM Tuned")
colnames(tab) <- c("RMSE", "R^2")
tab
```

```
# Make a table of classification results
tab <- matrix(c(round(rf_table$overall[1], 3), round(rf_table$overall[1], 3), round(gbm_table$ov
erall[1],
    3), round(gbm_tuned_table$overall[1], 3), round(svm_table$overall[1], 3), round(svm_table$ov
erall[1],
    3)), nrow = 6, ncol = 1, byrow = TRUE)
rownames(tab) <- c("Random Forest", "Random Forest Tuned", "GBM", "GBM Tuned", "SVM",
    "SVM Tuned")
colnames(tab) <- c("Accuracy")
tab
```

Summary statistics for regression models:

	RMSE	R^2
Random Forest	14.991	0.254
Random Forest Tuned	14.991	0.254
GBM	15.28	0.225
GBM Tuned	15.072	0.245
SVM	15.979	0.152
SVM Tuned	15.979	0.152

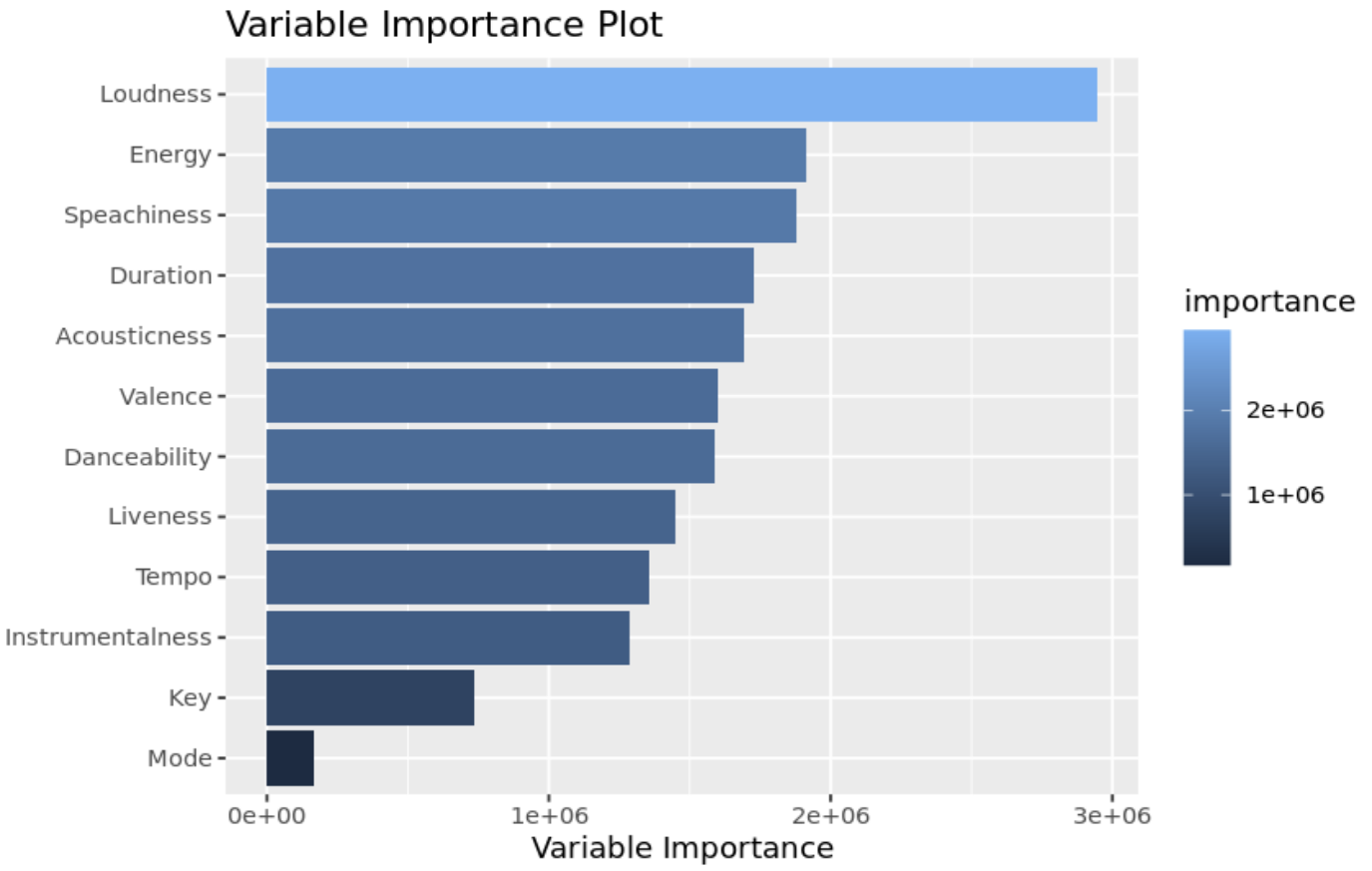
Summary statistics for classification models:

	Accuracy
Random Forest	0.592
Random Forest Tuned	0.592
GBM	0.584
GBM Tuned	0.590
SVM	0.572
SVM Tuned	0.572

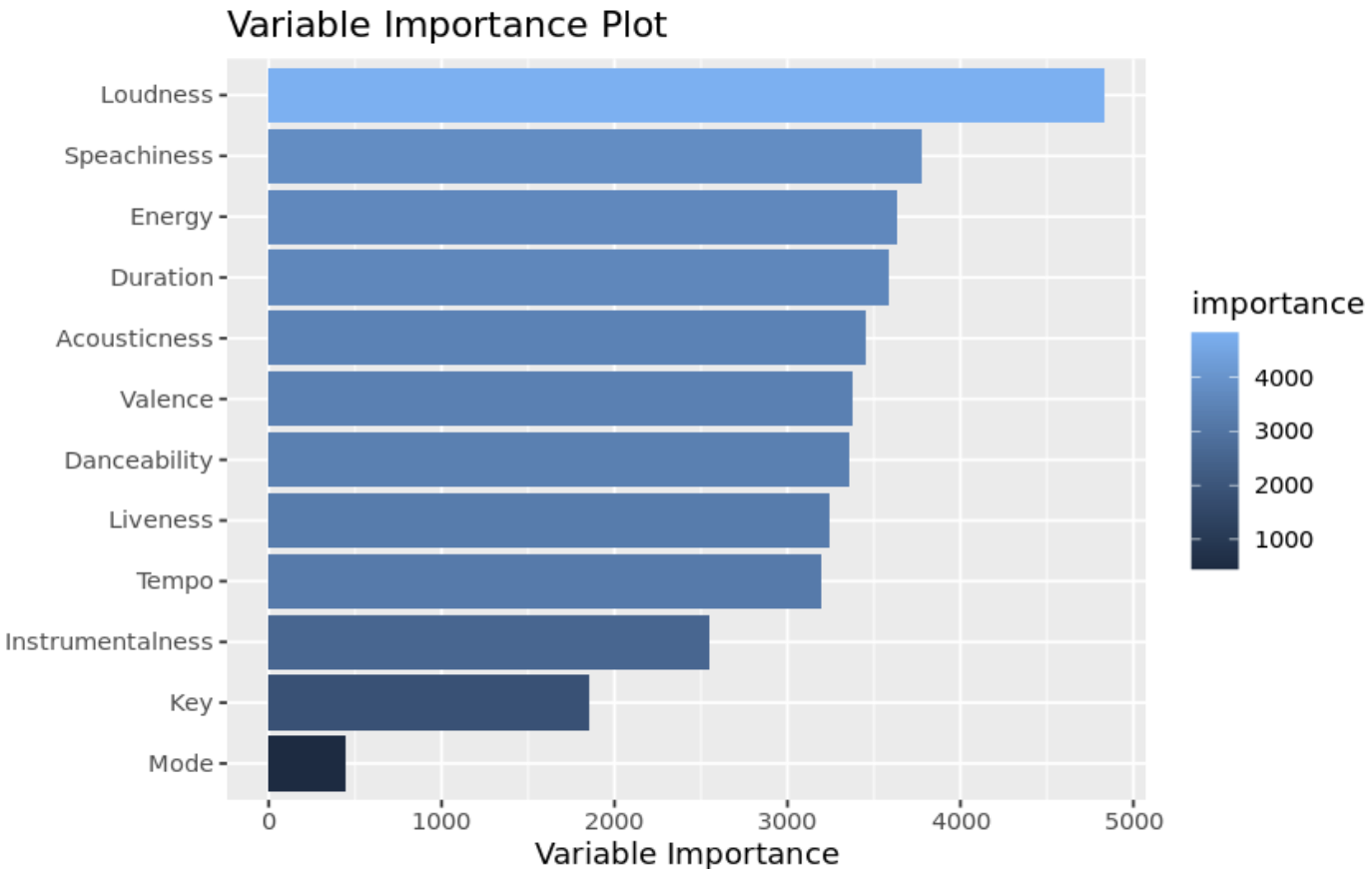
Here's a breakdown of the importance of each of the predictors from the random forest model for both regression and classification. The plots are similar except for the small discrepancy in the order of importance for *Energy* and *Speachiness*.

The variable that has the most impact in predicting song popularity is *loudness* with a huge margin. *Key* and *Mode* have the least impact in predicting song popularity. This makes sense because the average person would not normally know the key or mode of a song.

Regression



Classification



Conclusion

In summary our project accomplishes two objectives. First, we were able to create novel date using a combination of python, bash and R packages we learned throughout the semester. Second, we completed an exploratory and machine learning analysis on our novel dataset to understand what parameters of songs predict popularity. From our analysis we found that loudness was by far the best predictor of song popularity, followed by speechiness, energy and duration. While we don't intend to make any theoretical argument for these predictors they seem to make sense if we consider that one way (but certainly not the only way) music popularity spreads is radio play and songs that are loud, have lyrics and are not twelve minutes long tend to get radio time.