Predicting MBTI Personality Type

Angel Alcantara

2025-06-10

Contents

Introduction

The purpose of this project is to develop a model that will accurately predict a person's Myers Briggs Type Indicator (MBTI) Function Pair based on the audio features of their Spotify playlists. The data I will be using is from Kaggle: https://www.kaggle.com/datasets/xtrnglc/spotify-mbti-playlists.

What is the MBTI Personality Type?

The Myers Briggs Type Indicator (or MBTI for short) is a personality type system that can describe someone's personality from 16 distinct personality types. The distinct personality types are based on four dichotomies: introversion vs. extroversion, thinking vs. feeling, judging vs. perceiving, and sensing vs. intuition. Ultimately, this tool will categorize an individual into one of the 16 personality types.

What's the Objective?

Music has existed for at least 40000 years ago. Approximately between 200 BC and 100 AD, there was the discovery of the oldest surviving complete musical composition, the Seikilos epitaph. This means that music has a deep root in humanity and is a part of our individual and cultural identities.

Throughout my childhood and early adulthood, my music taste has changed significantly. Growing up all I listened to was pop, then in high school I started to get more into hip-hop/rap, and now I listen to a lot of new metal and indie music. As I reflect back on my childhood and my time at UCSB, I think about the type of person I was in each of these periods of my life. That's when I came to the realization that my music taste reflects how I feel, what my identity is, and so many other things about myself. While music may mean different things to different people, it is a part of an individual's identity whether they realize it or not. This is when I decided that I wanted to take a deep dive into music and personality types. My main objective is to answer: **can we predict personality function pair type based off of someone's music audio features?**

Loading Packages and Raw Data

Let's load in all of the packages we will need and the raw Spotify data.

```
# Loading necessary libraries
library(janitor) # for clean_names()
library(ggplot2)
library(dplyr)
library(corrplot)
library(corrr)
library(tidymodels)
library(naniar) # for missing values plot
library(readr)

# Importing data and assigning it to a variable
spotify <- read.csv("Data/combined_mbti_df.csv")
tidymodels_prefer() # To avoid function name conflicts
set.seed(4444) # To reproduce results

# To see the first few rows of the raw data
head(spotify)
```

```
##   mbti function_pair danceability_mean danceability_stdev energy_mean
## 1 INFP            NF         0.5578409         0.15501057   0.5533250
## 2 INFP            NF         0.5876364         0.13564425   0.5562727
## 3 INFP            NF         0.6770000         0.12800941   0.8512800
## 4 INFP            NF         0.5170000         0.16947739   0.5134119
## 5 INFP            NF         0.5604000         0.14145007   0.4458620
## 6 INFP            NF         0.6139355         0.09817941   0.7283871
##   energy_stdev loudness_mean loudness_stdev mode_mean mode_stdev
## 1    0.2251780     -8.352591       3.273317 0.6590909  0.4794950
## 2    0.1916420     -8.215697       3.356867 0.6363636  0.4885042
## 3    0.1283365     -5.046100       2.180554 0.5800000  0.4985694
## 4    0.2583449    -10.172833       4.935140 0.7857143  0.4152997
## 5    0.2425917    -10.572240       5.685179 0.8200000  0.3880879
## 6    0.1280338     -6.986581       1.639672 0.5483871  0.5058794
##   speechiness_mean speechiness_stdev acousticness_mean acousticness_stdev
## 1       0.06734091        0.06108258        0.29346909         0.28608805
## 2       0.07427273        0.08338823        0.34143394         0.26707676
## 3       0.27254600        0.17601286        0.05808475         0.07481759
## 4       0.05150952        0.03391090        0.44216304         0.36702457
## 5       0.06517800        0.05868432        0.47967740         0.36140648
## 6       0.28977097        0.12265243        0.18835484         0.18640594
##   liveness_mean liveness_stdev valence_mean valence_stdev tempo_mean
## 1     0.1634023     0.08626590    0.4488841     0.2217635  120.95480
## 2     0.1545424     0.11249567    0.4476030     0.2160687  131.87124
## 3     0.2645200     0.20809663    0.2650300     0.1705175  128.20602
## 4     0.1526452     0.09595586    0.3378452     0.2118900  120.59357
## 5     0.1690920     0.12663451    0.3513660     0.1951161  114.69308
## 6     0.2204097     0.18585675    0.5082903     0.1634673   96.37294
##   tempo_stdev instrumentalness_mean instrumentalness_stdev Cminor_count
## 1    32.16578            0.01997099             0.08180459            1
## 2    33.21601            0.06104086             0.17755873            1
```

```
## 3    26.92719            0.25035756             0.33180466            0
## 4    34.33522            0.21092715             0.29434621            2
## 5    30.03414            0.06953727             0.20627703            2
## 6    22.35451            0.02886392             0.16056249            1
##   CMajor_count C..Dbminor_count C..DbMajor_count DMajor_count D._EbMajor_count
## 1            5                3                5           4                3
## 2            5                0                1           1                0
## 3            0                3               13           4                0
## 4            6                0                2           3                0
## 5           12                0                4           2                2
## 6            1                0                6           3                0
##   Eminor_count EMajor_count Fminor_count FMajor_count F..Gbminor_count
## 1            5           1            1           2                1
## 2            4           0            1           2                0
## 3            2           0            2           1                2
## 4            1           5            1           3                2
## 5            2           0            0           2                0
## 6            1           0            1           0                1
##   GMajor_count G..Abminor_count G..AbMajor_count Aminor_count AMajor_count
## 1            2                1                1           2            2
## 2            3                1                2           1            2
## 3            2                1                2           0            1
## 4            1                0                3           2            5
## 5            4                1                1           1            7
## 6            3                0                1           0            1
##   A..Bbminor_count BMajor_count Dminor_count D._Ebminor_count Gminor_count
## 1                1           4            0                0            0
## 2                1           4            1                1            1
## 3                1           3            1                0            0
## 4                0           2            0                1            0
## 5                2           1            0                0            0
## 6                3           1            1                2            0
##   A..BbMajor_count F..GbMajor_count Bminor_count
## 1                0                0           0
## 2                1                0           0
## 3                1                2           9
## 4                3                0           0
## 5                5                1           1
## 6                0                1           4
```

This data is from a Kaggle data set,Spotify MBTI Playlists.

Exploring and Tidying the Raw Data

Variable Selection

Let's take a look at the data and it's characteristics, so we know what we're working with here.

```
# To see how many rows and columns are in the raw data
dim(spotify) # output: row columns
```

```
## [1] 4081   46
```

The Spotify data set has 4081 rows and 46 columns. In a dataset, rows represent the observations and columns represent the variables. This means that we have 4081 Spotify playlists and 46 audio features in this dataset.

There are some variables in this dataset that will not be relevant in our model, therefore it's safe to say that we can "drop" them from our dataset. The columns that count the amount of major or minor key songs a playlist has (e.g., `Eminor_count`, `FMajor_count`) will not be useful in this model since they are too specific and difficult for the common music listener to understand. Therefore, we will remove these columns.

Also, in this dataset we are given the standard deviation and mean in separate variables for each audio feature (e.g., `tempo_mean`, `tempo_stdvev`). We will only be focusing on the mean values of these audio features, therefore we can remove all of these columns from our data.

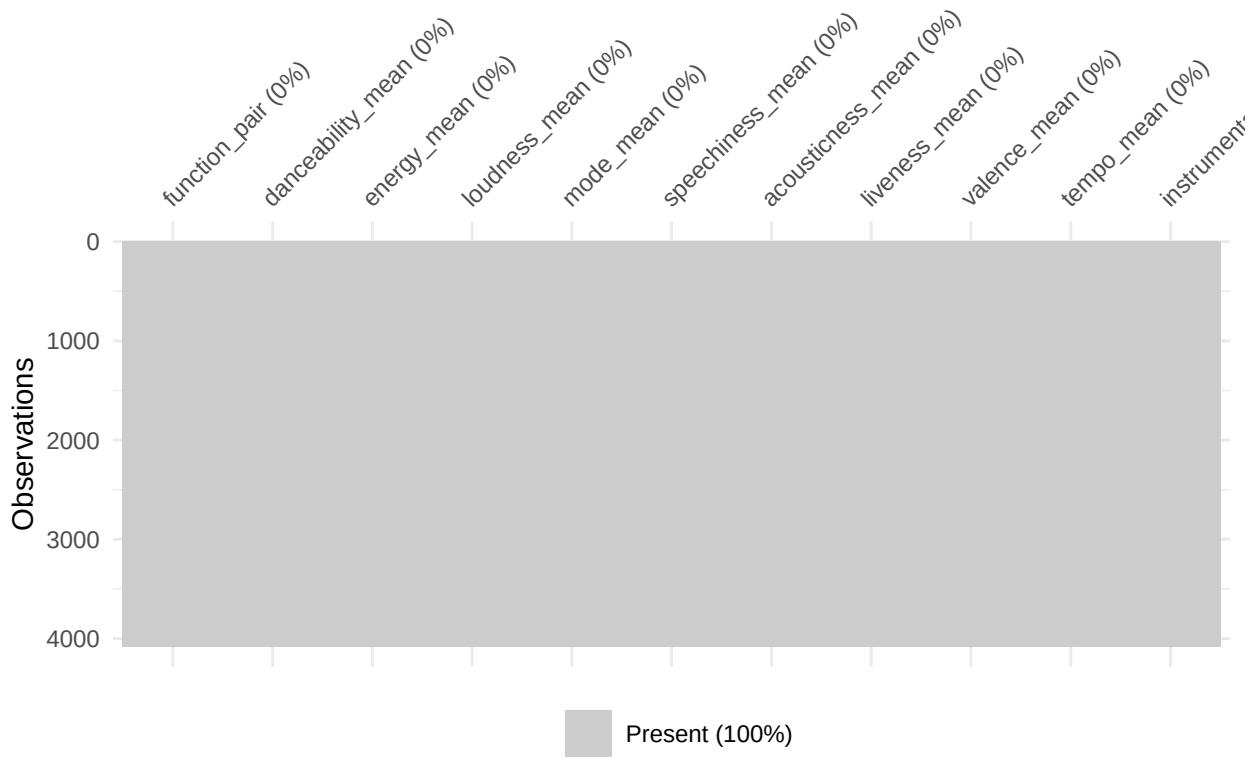
```
# Selecting variables we want
spotify <- spotify %>%
  select(c("function_pair", "danceability_mean", "energy_mean", "loudness_mean", "mode_mean", "speechin
  clean_names()  #to make variable names look nicer

# Converting function pair to a factor
spotify <- spotify %>%
  mutate(function_pair = as.factor(function_pair))
```

Missing Observations

Before we start visualizing our data, it is important to check for missing observations. It is normal for data to have a few missing observations (NA), but that doesn't mean we ignore it. Therefore, we will plot the missing values and attain the number of missing values in our data.

```
spotify %>%
  ungroup() %>%
  vis_miss()
```




```
sum(is.na(spotify))
```

```
## [1] 0
```

This looks great! After plotting the missing observations and attaining the sum of missing observations in our dataset, we can see that there are 0. This means that we don't need to do anything to fix this because there is no issue.

Describing the Predictors

Since we have removed the unnecessary variables from our raw data, we now have our true dataset that we will be working with from now on. To get a better idea of what predictors we will be using, here is a description of each one:

- `danceability_mean`: Based on factors like tempo, rhythm, stability, beat strength, and overall consistency, this will measure how well a track is suited for dancing. The score ranges from 0.0 to 1.0, where 1.0 means that it is very danceable.
- `energy_mean`: The energy of the track, ranging from 0.0 (low energy) to 1.0 (high energy).
- `loudness_mean`: The overall loudness of a track in decibels (dB). Loudness values are averaged across the entire track and are useful for comparing relative loudness of tracks. Loudness is the quality of a sound that is the primary psychological correlate of physical strength (amplitude). Values typically range between -60 and 0 db.

- `mode_mean`: Mode indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived. Major is represented by 1 and minor is 0.
- `speechiness_mean`: Speechiness detects the presence of spoken words in a track. The more exclusively speech-like the recording (e.g. talk show, audio book, poetry), the closer to 1.0 the attribute value. Values above 0.66 describe tracks that are probably made entirely of spoken words. Values between 0.33 and 0.66 describe tracks that may contain both music and speech, either in sections or layered, including such cases as rap music. Values below 0.33 most likely represent music and other non-speech-like tracks.
- `acousticness_mean`: The mean value between 0.0 and 1.0 that estimates how likely a track is to be acoustic. The closer the value is to 1.0, then this would indicate the song is most likely acoustic.
- `liveness_mean`: Detects the presence of an audience in the recording. Higher liveness values represent an increased probability that the track was performed live. A value above 0.8 provides strong likelihood that the track is live.
- `valence_mean`: A measure from 0.0 to 1.0 describing the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry).
- `tempo_mean`: The overall estimated tempo of a track in beats per minute (BPM). In musical terminology, tempo is the speed or pace of a given piece and derives directly from the average beat duration.
- `instrumentalness_mean`: Predicts whether a track contains no vocals. "Ooh" and "aah" sounds are treated as instrumental in this context. Rap or spoken word tracks are clearly "vocal". The closer the instrumentalness value is to 1.0, the greater likelihood the track contains no vocal content. Values above 0.5 are intended to represent instrumental tracks, but confidence is higher as the value approaches 1.0.

The descriptions for this dataset are from the Kaggle website where I got the data from.

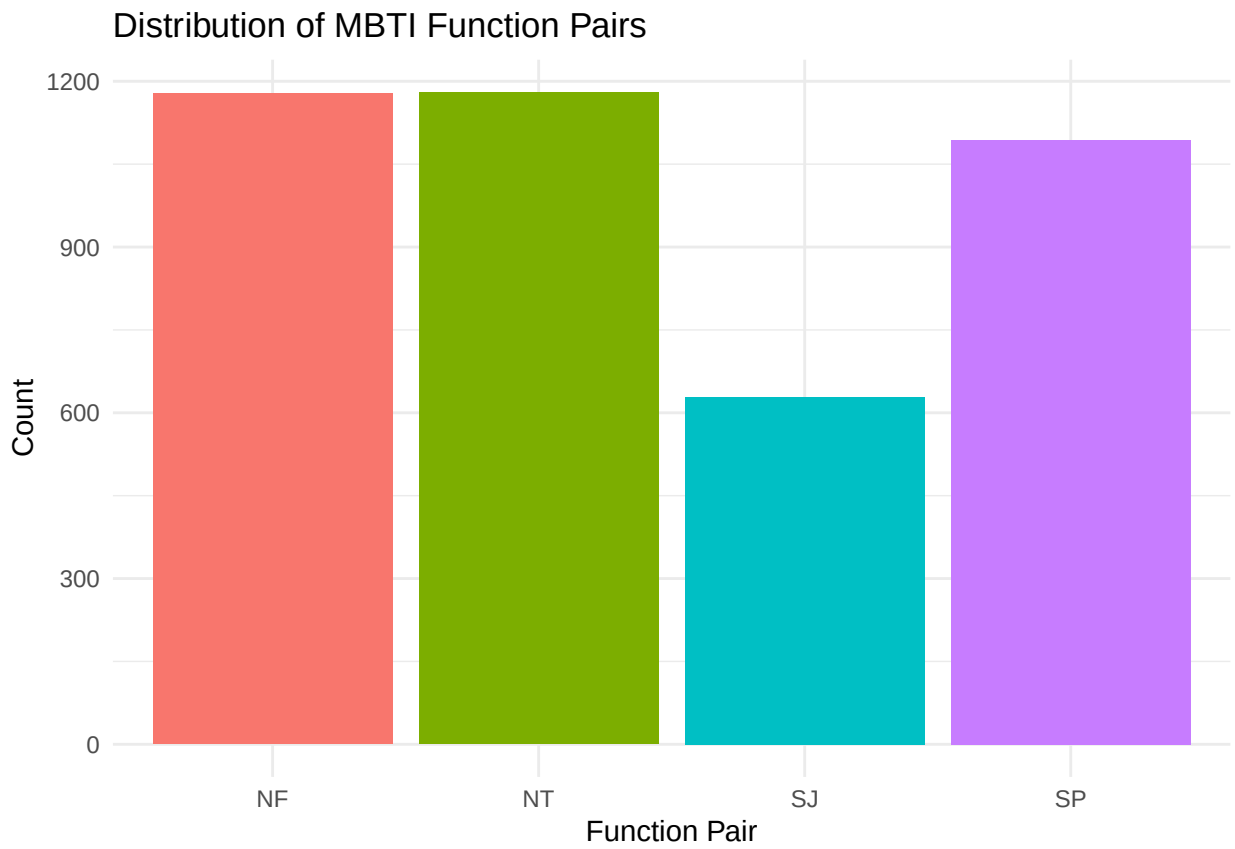
Visual EDA

Now, we will shift our focus into some visual exploratory data analysis. In this part we will focus on understanding the distribution of our outcome variable `function_pair`, as well as identifying potential patterns/relationships/trends across the predictors.

Function Pair Distribution

We will see a plot of the distribution of function pairs in our data. This will allow us to see if there is a function pair that has very few or too many observations compared to the others.

```
# Plotting distribution of function_pair
ggplot(spotify, aes(x = function_pair,
                    fill = function_pair
                    )) +
  geom_bar() +
  xlab("Function Pair") +
  ylab("Count") +
  ggtitle("Distribution of MBTI Function Pairs") +
  theme_minimal() +
  theme(legend.position = "none")
```

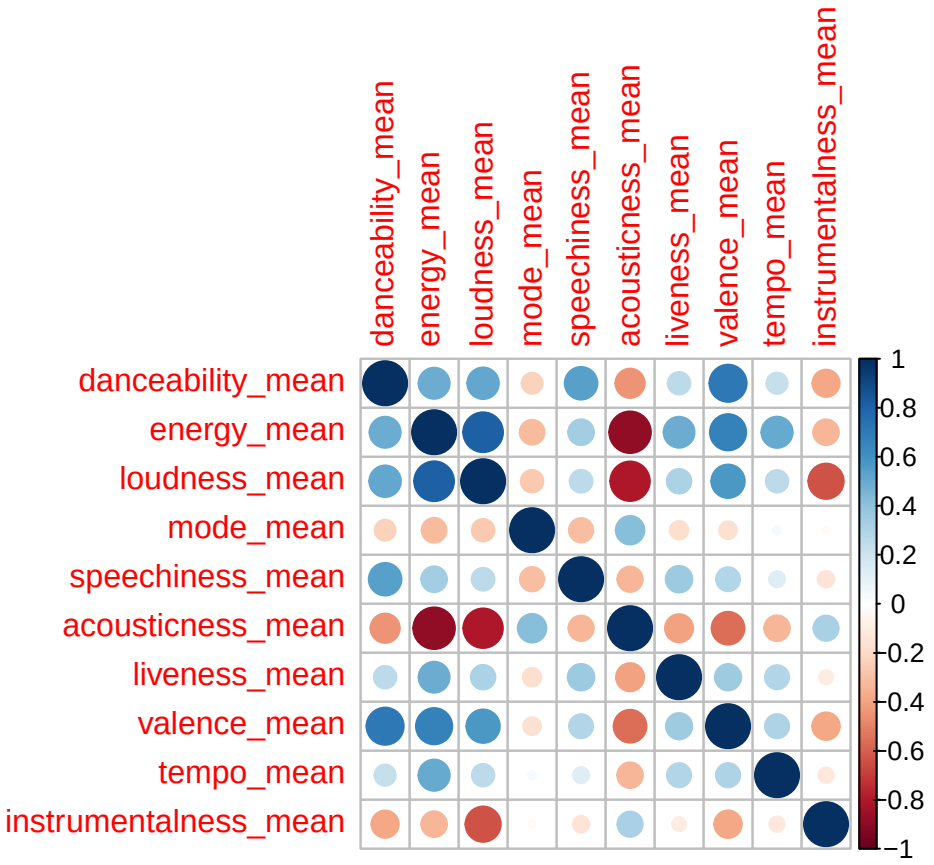


From this barplot, it is clear that the function pair SJ has an underwhelming amount of observations compared to the rest of the function pairs.

Variable Correlation Plot

In order to get more insight into our predictors, we will create a correlation matrix and use that to create a correlation plot that will show us the correlation of these predictors. All of the predictors are numeric, so we will be able to use all of our predictors in this matrix.

```
spotify %>%
  select(where(is.numeric)) %>% # select numeric variables
  cor() %>%
  corrplot()
```



We can see that there are some strong positive and negative correlations. Let's start with the positive ones (the numbers are estimations):

- `energy_mean` and `loudness_mean` (0.8): This makes sense to me since louder songs are typically more energetic. The first genre that comes to me is heavy metal, it's loud and energetic!
- `danceability_mean` and `energy_mean` (0.5-0.6): This makes sense because there are a fair amount of songs that are danceable and also have a lot of energy in them. An example of this would be reggaeton.
- `loudness_mean` and `danceability_mean` (0.6): This makes sense since loud music tends to bring people's energy up, and will therefore lead to people to move, or dance.

Now, some negative correlations:

- `acousticness_mean` and `energy_mean` (-0.9): This makes sense because acoustic tracks have a tendency to be calmer and laid-back
- `acousticness_mean` and `loudness_mean` (-0.8): This makes sense because by design, acoustic tracks are typically quieter

Due to the high positive correlation for `energy_mean` and `loudness_mean`, I will only keep `energy_mean` for my model fitting in order to reduce multicollinearity. Many models are sensitive to this and both of these predictors are pretty similar in what they represent, so removing one of them is not a significant loss.

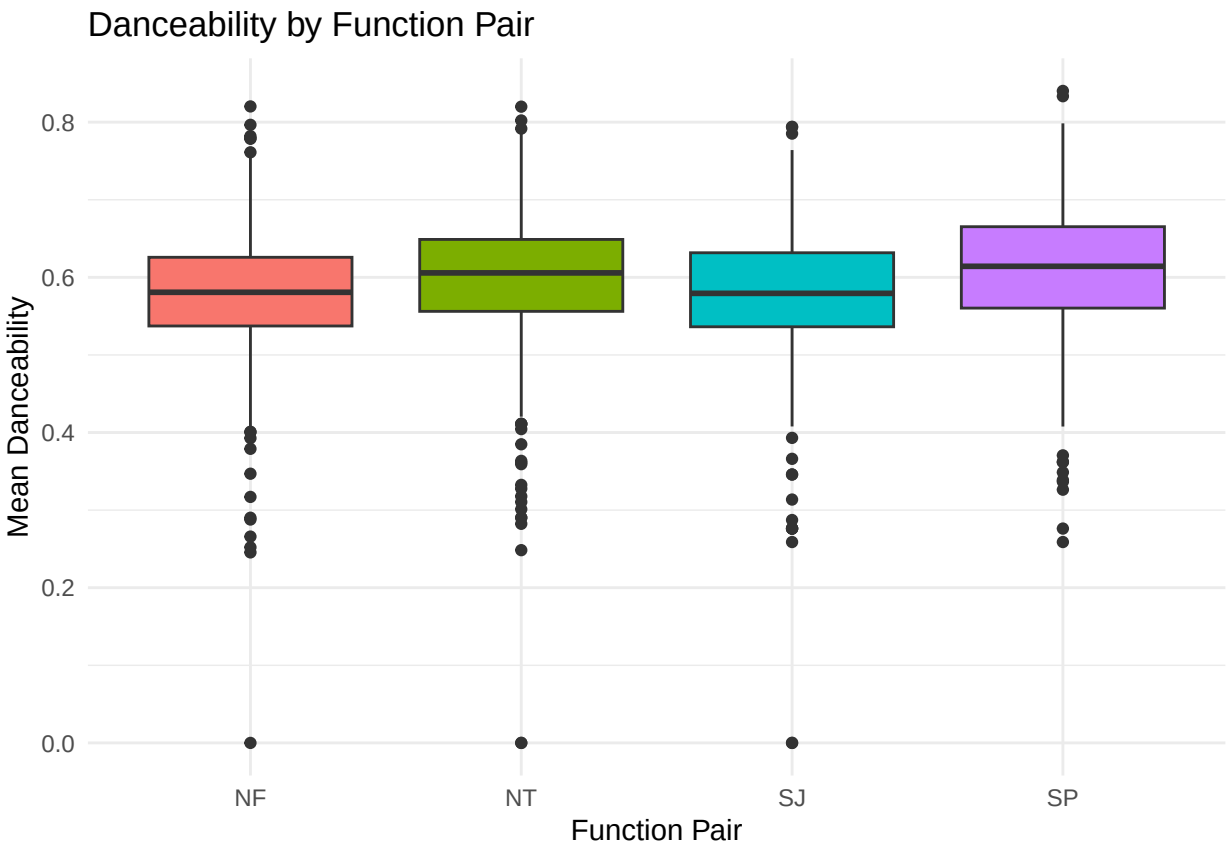
Comparing Audio Features by Function Pair

Danceability

Music is not only for listening and singing along with friends, it also makes you want to move your hips and dance! This is a significant characteristic of music, and therefore we will take a look at how it varies across personality types.

There isn't a significant difference among the function pairs when it comes to danceability. I'm a bit surprised because I'd expect at least one function pair to have a significantly lower danceability mean.

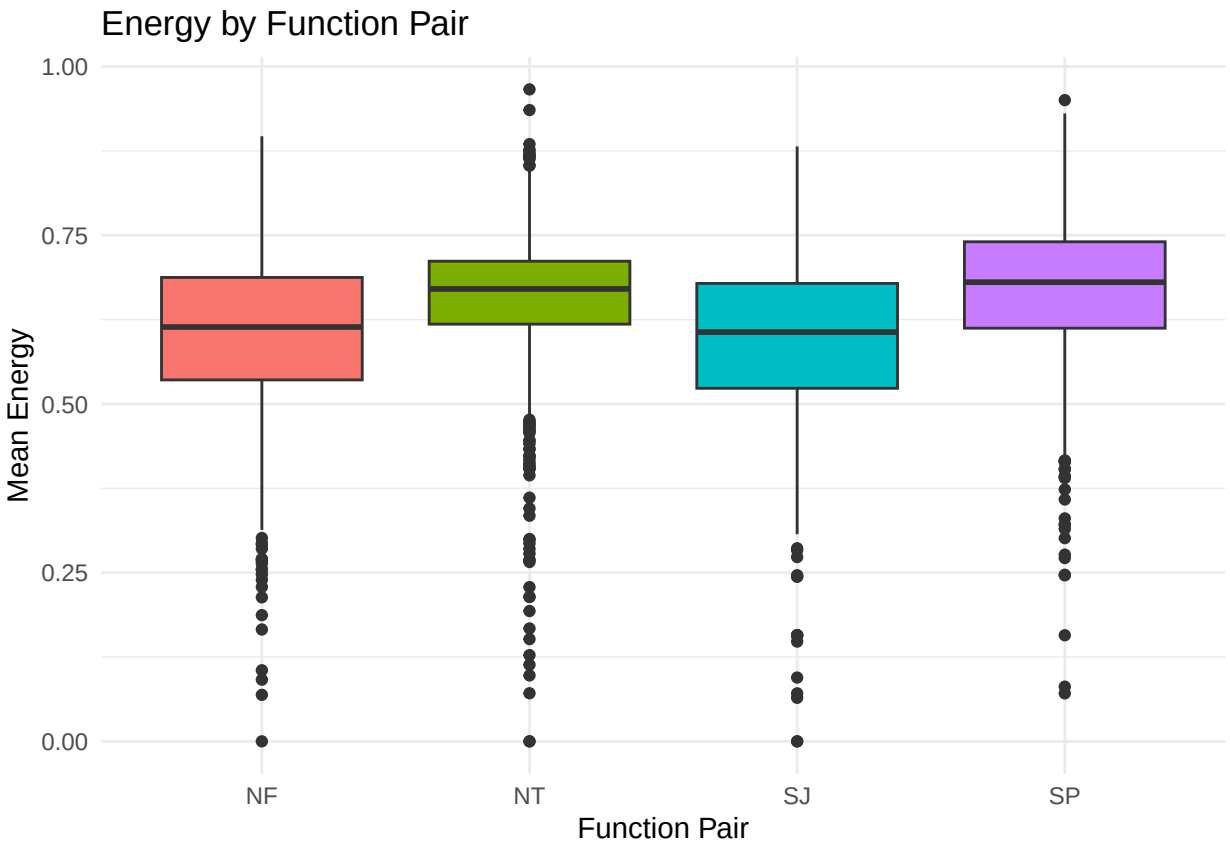
```
ggplot(spotify, aes(x = function_pair, y = danceability_mean, fill = function_pair)) +
  geom_boxplot() +
  xlab("Function Pair") +
  ylab("Mean Danceability") +
  ggtitle("Danceability by Function Pair") +
  theme_minimal() +
  theme(legend.position = "none")
```



Energy

There isn't any significant observations about the energy levels of the function pairs. Their means are close and the sizes of their boxes as well.

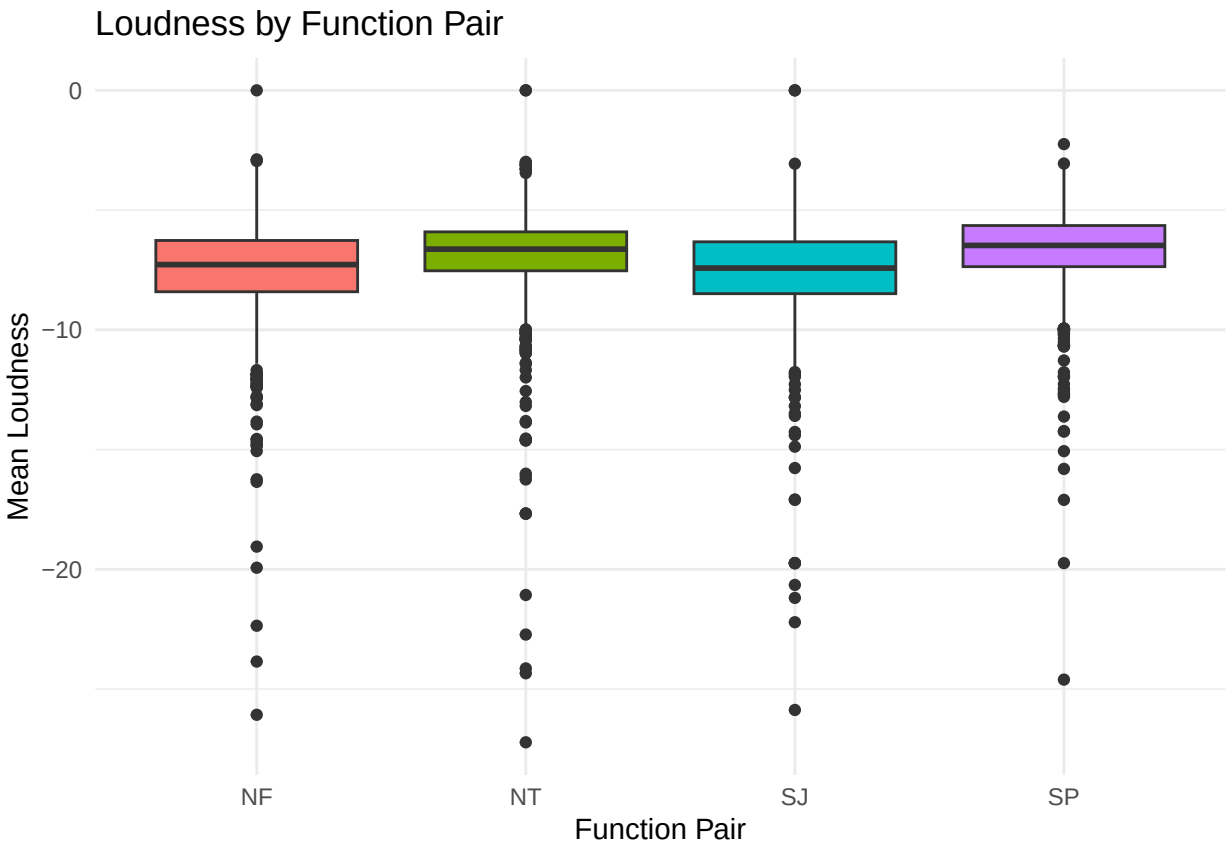
```r
ggplot(spotify, aes(x = function_pair, y = energy_mean, fill = function_pair)) +
  geom_boxplot() +
  xlab("Function Pair") +
  ylab("Mean Energy") +
  ggtitle("Energy by Function Pair") +
  theme_minimal() +
  theme(legend.position = "none")
```



Loudness

Loudness seems to not have a big contribution since all of the boxes are thin, but they all have around the same mean, which means loudness is consistent in all of the boxplots.

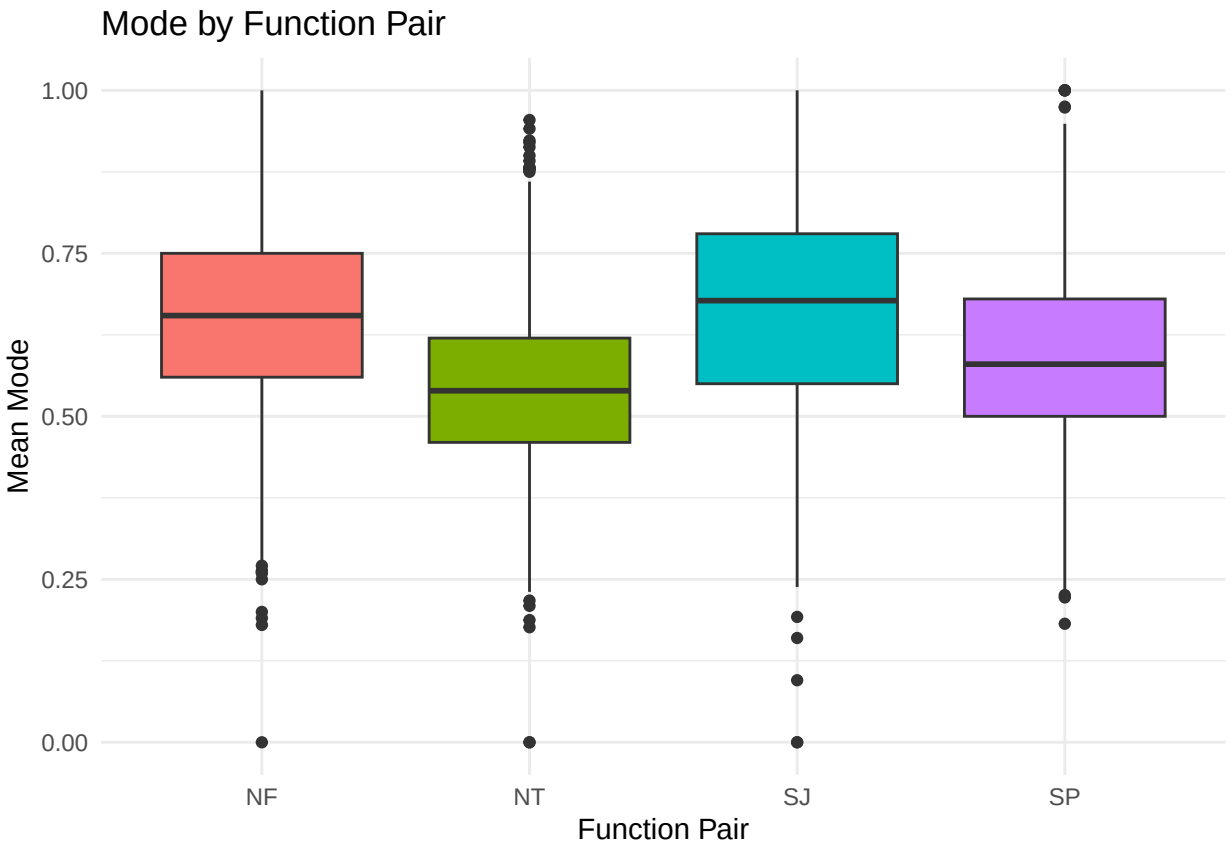
```r
ggplot(spotify, aes(x = function_pair, y = loudness_mean, fill = function_pair)) +
  geom_boxplot() +
  xlab("Function Pair") +
  ylab("Mean Loudness") +
  ggtitle("Loudness by Function Pair") +
  theme_minimal() +
  theme(legend.position = "none")
```



Mode

Mode seems to have a big contribution since all of the boxes are thin, but they all have around the same mean, which means mode is consistent in all of the boxplots.

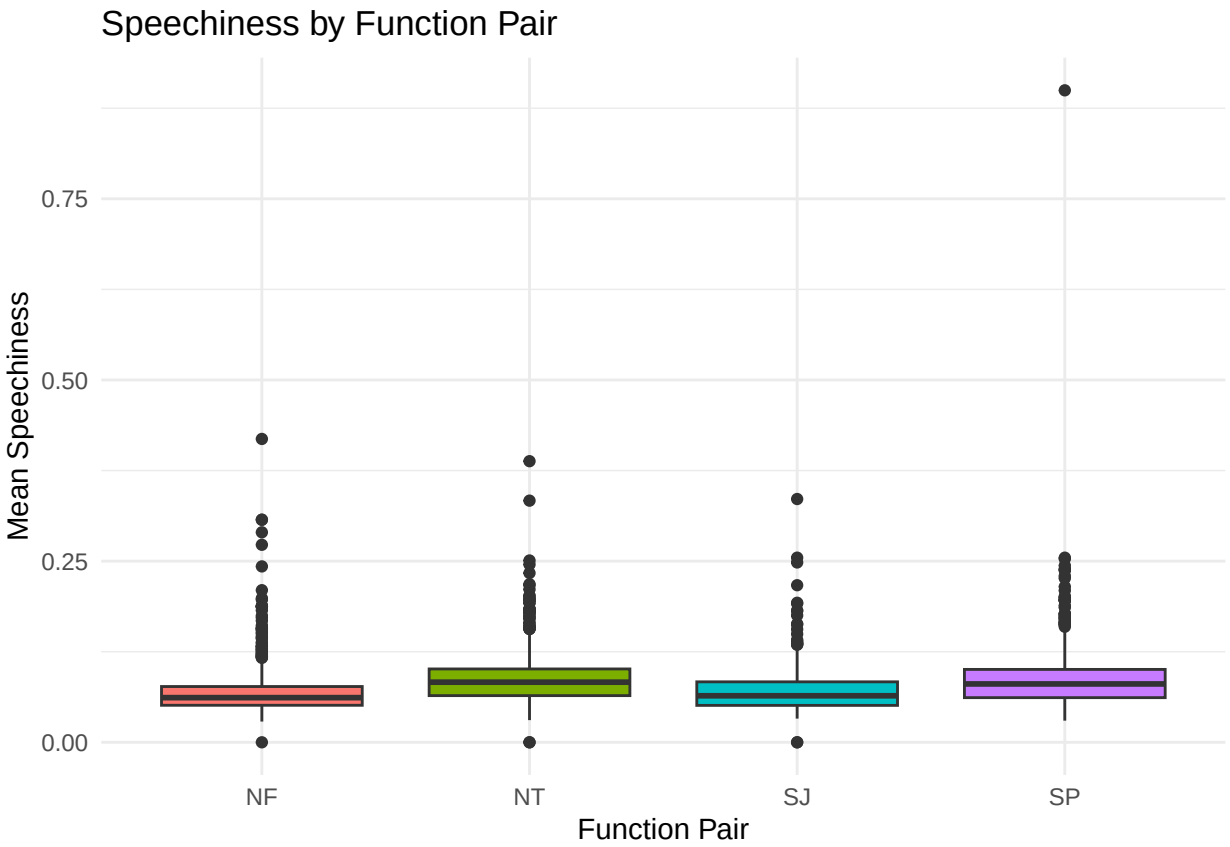
```
ggplot(spotify, aes(x = function_pair, y = mode_mean, fill = function_pair)) +
  geom_boxplot() +
  xlab("Function Pair") +
  ylab("Mean Mode") +
  ggtitle("Mode by Function Pair") +
  theme_minimal() +
  theme(legend.position = "none")
```



Speechiness

Speechiness seems to not have a insignificant contribution since all of the boxes are thin, but they all have around the same mean, which means speechiness is consistent in all of the boxplots.

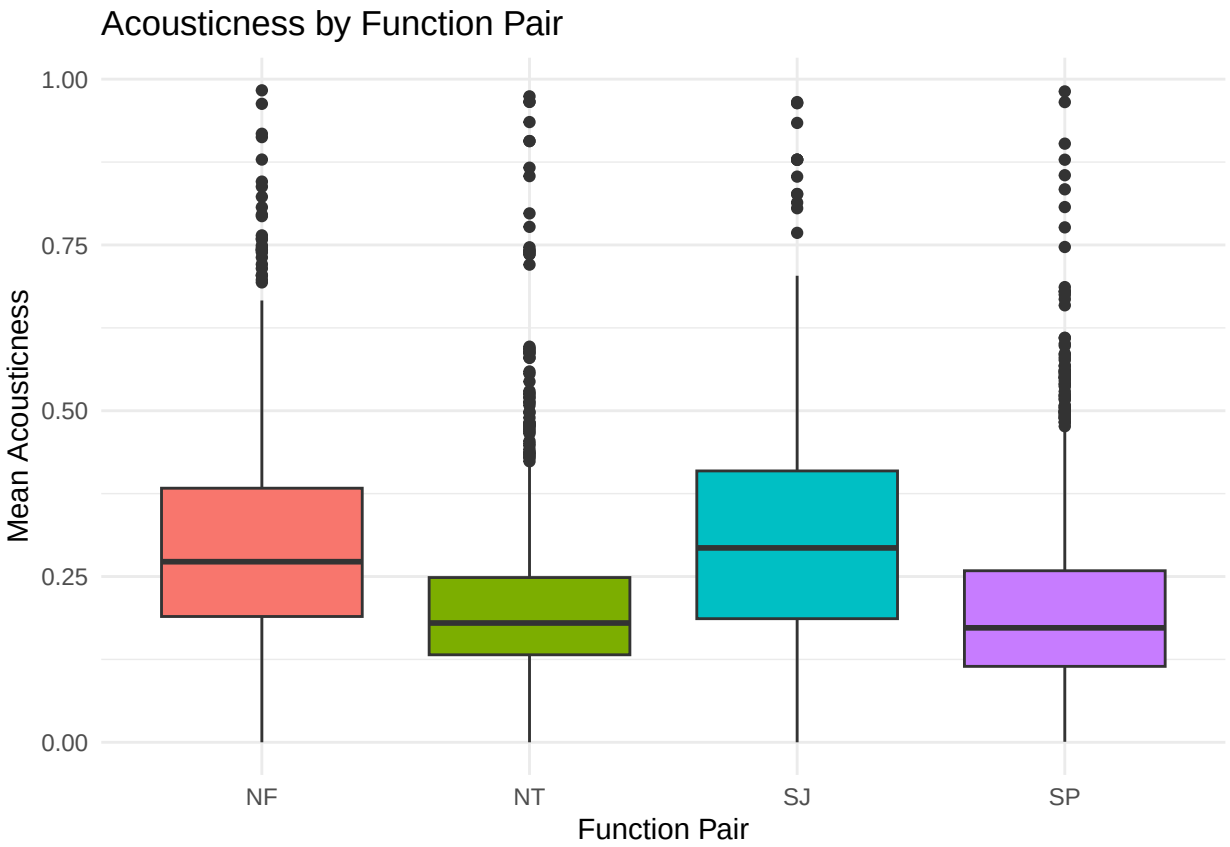
```
ggplot(spotify, aes(x = function_pair, y = speechiness_mean, fill = function_pair)) +
  geom_boxplot() +
  xlab("Function Pair") +
  ylab("Mean Speechiness") +
  ggtitle("Speechiness by Function Pair") +
  theme_minimal() +
  theme(legend.position = "none")
```



Acousticness

Acousticness seems to not have a contribution since all of the boxes are thin, but they all have around the same mean, which means acousticness is consistent in all of the boxplots.

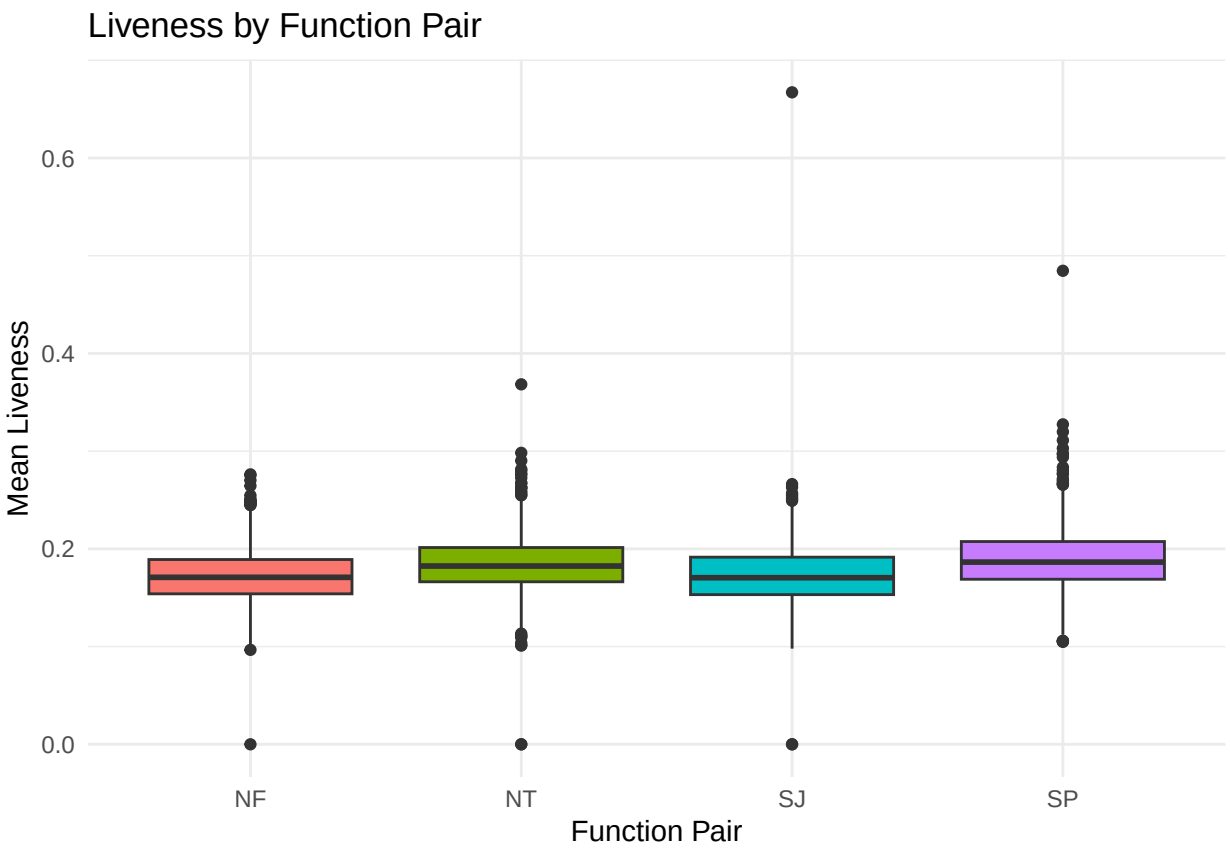
```
ggplot(spotify, aes(x = function_pair, y = acousticness_mean, fill = function_pair)) +
  geom_boxplot() +
  xlab("Function Pair") +
  ylab("Mean Acousticness") +
  ggtitle("Acousticness by Function Pair") +
  theme_minimal() +
  theme(legend.position = "none")
```



Liveness

Liveness seems to not have a big contribution since all of the boxes are thin, but they all have around the same mean, which means liveness is consistent in all of the boxplots.

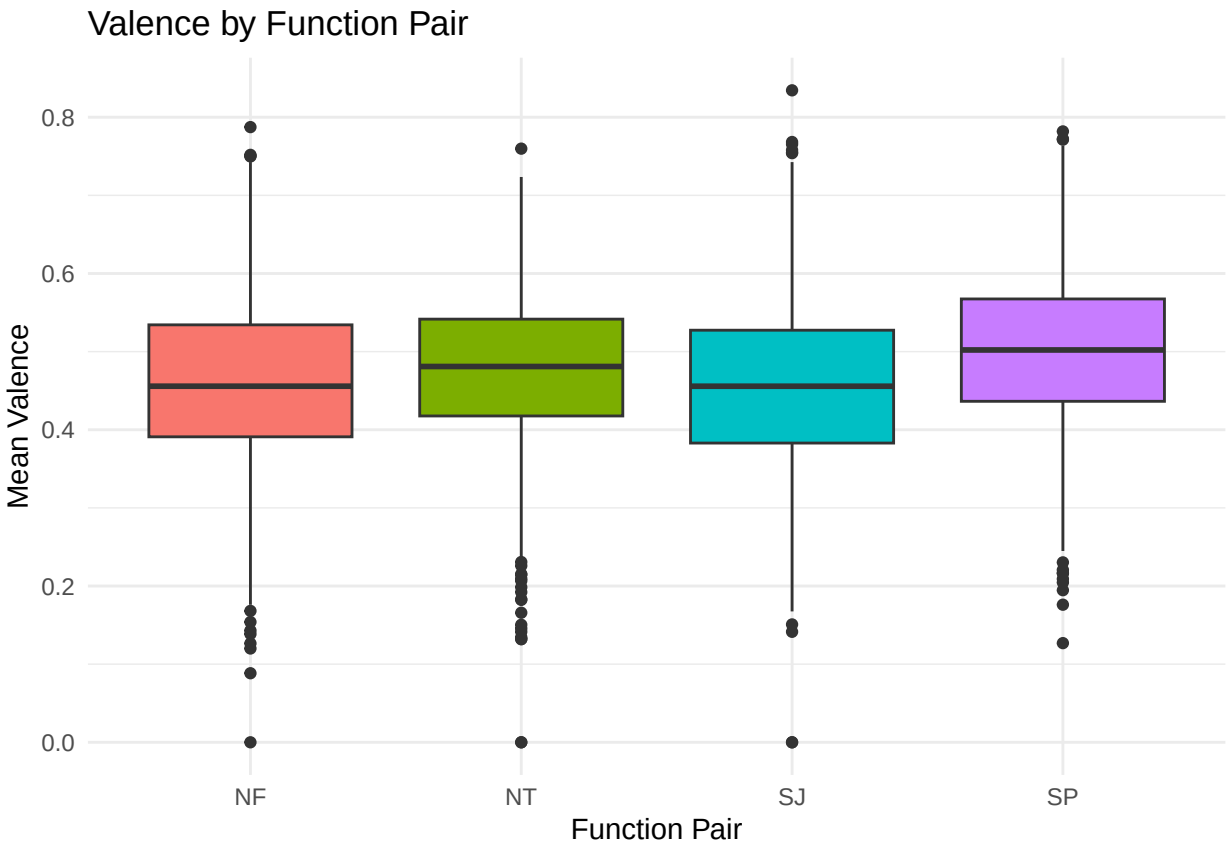
```
ggplot(spotify, aes(x = function_pair, y = liveness_mean, fill = function_pair)) +
  geom_boxplot() +
  xlab("Function Pair") +
  ylab("Mean Liveness") +
  ggtitle("Liveness by Function Pair") +
  theme_minimal() +
  theme(legend.position = "none")
```



Valence

Valence seems to have a effect since all of the boxes are thin, but they all have around the same mean, which means valence is consistent in all of the boxplots.

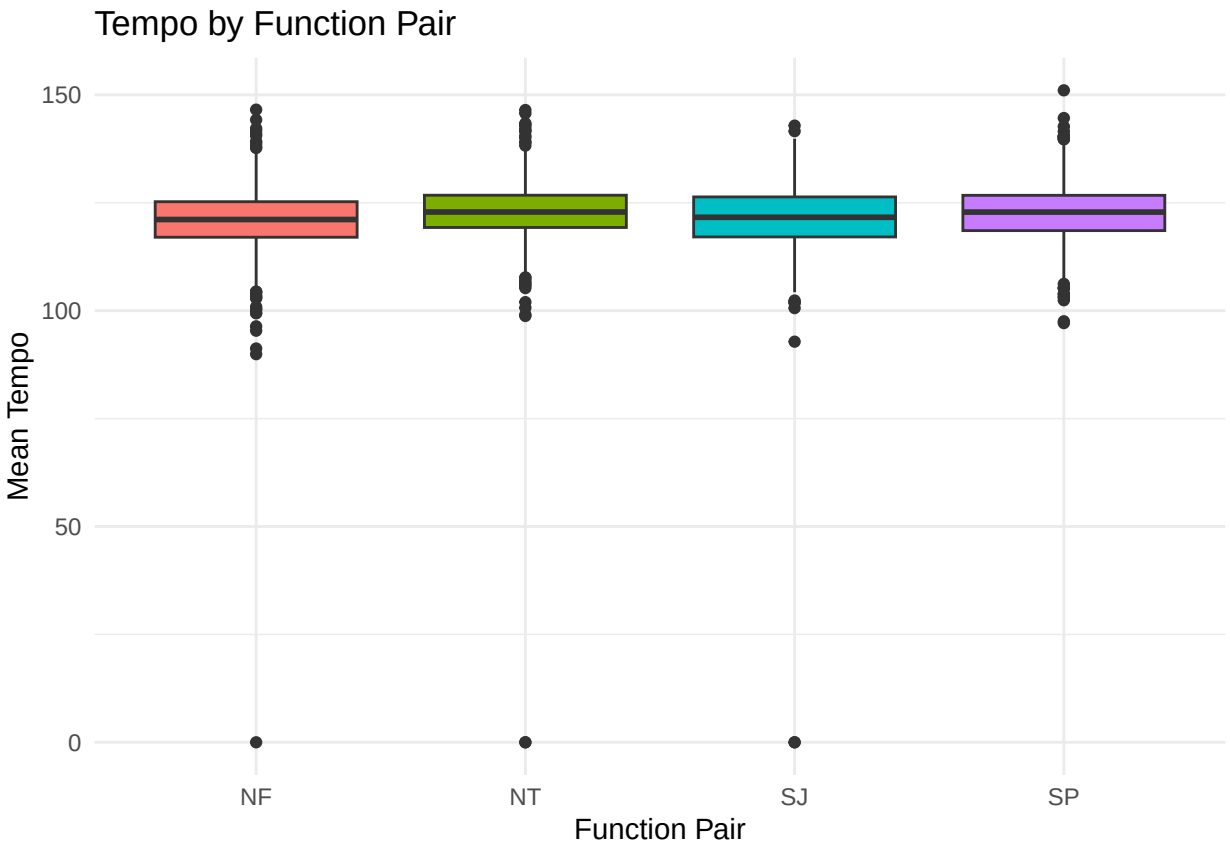
```
ggplot(spotify, aes(x = function_pair, y = valence_mean, fill = function_pair)) +
  geom_boxplot() +
  xlab("Function Pair") +
  ylab("Mean Valence") +
  ggtitle("Valence by Function Pair") +
  theme_minimal() +
  theme(legend.position = "none")
```



Tempo

Tempo seems to not have a contribution since all of the boxes are thin, but they all have around the same mean, which means tempo is consistent in all of the boxplots.

```
ggplot(spotify, aes(x = function_pair, y = tempo_mean, fill = function_pair)) +
  geom_boxplot() +
  xlab("Function Pair") +
  ylab("Mean Tempo") +
  ggtitle("Tempo by Function Pair") +
  theme_minimal() +
  theme(legend.position = "none")
```



Setting up Models

After some EDA, we now have better knowledge about our data and the variables we believe will be important in obtaining our objective. With this new knowledge, we can start building models. We will be splitting our data into testing/training sets, creating a recipe, and going through the process of cross-validation.

Training/Testing Split

An essential part of effective model building, is to divide the dataset into a training and testing set. The training set will be the data that is used to train our model into accurately predicting our outcome variable. The testing set is what we will use to check how our model perform with "new" data. I will be splititng the data to be 75% training and 25% testing.

```
# To reproduce results
set.seed(5566)

# Splitting data into two sets (training and testing)
spotify_split <- initial_split(spotify,
                               prop = 0.75,
                               strata = "function_pair")

# Creating testing and training set
spotify_train <- training(spotify_split)
```

```
spotify_test <- testing(spotify_split)

# Checking number of rows and columns of each set
dim(spotify_train)
```

```
## [1] 3059   11
```

```
dim(spotify_test)
```

```
## [1] 1022   11
```

There are now 3059 observations in the training set, and 1022 observations in the testing set. These are reasonable amounts for our model building.

Creating a Recipe

We are now building a recipe to use with all of the models. We eliminated a couple of variables earlier in the project in order to tidy the data. Our recipe will be used in order to help the models to better understand our dataset and the variables (predictors) it is dealing with.

```
spotify_recipe <- recipe(function_pair ~ .,
                         data = spotify_train) %>%
  step_center(all_predictors()) %>% # center all predictors
  step_scale(all_predictors()) # standardizing our predictors
```

K-Fold Cross Validation

We will use 5 folds to perform stratify our cross validation on our response variable, `function_pair`.

```
spotify_folds <- vfold_cv(spotify_train,
                          v = 5,
                          strata = function_pair)
```

Since models take a long time to run, I decided to save the results to a RDA file. Therefore, when I had the model I wanted I could load it back without having to wait a long time for it to run.

```
save(spotify_folds, spotify_recipe, spotify_train, spotify_test, file = "/Users/angelalcantara/Downloads
```

Model Building

We are now going to start building our models! The dataset that we are working with is very large, therefore it will be difficult to be able to run the tuned versions, since they will take a lot of time. The files for the tuned versions of these models are in their own file to limit the runtime. The models that we are fitting are: Multinomial Logistic classification, Random Forest, K-nearest neighbors, and Boosted Trees.

Fitting the Models

All of the models that I fitted my data too have a similar process, therefore I will show each step with the code. I will not be running the code here since it would take a lot of computing time.

Here are the steps for fitting the models:

1. Setting up the model: specify the model and it's parameters that we are fitting. Then, put the engine model it is coming from, and finally the mode, it will be either regression or classification. Since the outcome variable is categorical, then we will be doing classification

```
# Multinomial Logistic classification
log_model <- multinom_reg() %>%
  set_engine("nnet") %>%
  set_mode("classification")

# Random Forest
rf_spec <- rand_forest(mtry = tune(), # number of predictors
                       trees = tune(),
                       min_n = tune()) %>% # number of minimum values in each node
  set_engine("ranger", importance = "impurity") %>%
  set_mode("classification")

# K-nearest neighbors
knn_model <- nearest_neighbor(neighbors = tune()) %>%
  set_mode("classification") %>%
  set_engine("kknn")

# Boosted Trees
boosted_spec <- boost_tree(trees = tune(),
                           learn_rate = tune(), # learning rate
                           min_n = tune()) %>%
  set_engine("xgboost") %>%
  set_mode("classification")
```

2. In this step, we will combine the model and recipe into a worklfow

```
# Multinomial Logistic classification
log_workflow <- workflow() %>%
  add_model(log_model) %>%
  add_recipe(spotify_recipe)

# Random Forest
rf_workflow <- workflow() %>%
  add_model(rf_spec) %>%
  add_recipe(spotify_recipe)

# K-nearest neighbors
knn_workflow <- workflow() %>%
  add_model(knn_model) %>%
  add_recipe(spotify_recipe)

# Boosted Trees
```

```r
boosted_workflow <- workflow() %>%
  add_model(boosted_spec) %>%
  add_recipe(spotify_recipe)
```

3. Now, we will create a tuning grid. This will allow us to define ranges and levels for the parameters we wish to tune.

```r
# Multinomial Logistic Classification
## Fits perfectly, no tuning grid required

# Random forest
rf_grid <- grid_regular(
  mtry(range = c(2, 5)),
  trees(range = c(100,300)),
  min_n(range = c(2,10)),
  levels = 4
)

# K-nearest neighbors
knn_grid <- grid_regular(
  neighbors(range = c(3, 25)),
  levels = 6)

# Boosted Trees
boosted_grid <- grid_regular(
  trees(range = c(100,1000)),
  learn_rate(range = c(0.01, 0.3)),
  min_n(range = c(2, 10)),
  levels = 3
)
```

4. In this step, we will be specifying the workflow, k-fold cross validation folds, and tuning grid for our desired parameter, as well as tuning the model.

```r
# Multinomial Logistic Classification
## Fits perfectly, no tuning required

# Random forest
#rf_tune_res <- tune_grid(
  #rf_workflow,
  #resamples = spotify_folds,
  #grid = rf_grid
#)

# K-nearest neighbors
#knn_tune <- tune_grid(
  #knn_workflow,
  #resamples = spotify_folds,
  #grid = knn_grid
#)

# Boosted Trees
```

```
#boosted_tune_res <- tune_grid(
  #boosted_workflow,
  #resamples = spotify_folds,
  #grid = boosted_grid
#)
```

5. Running these models can take 5 or up to 15 minutes, therefore we will save these tuned models to an RDS file to avoid rerunning it.

```
# Save entire R environment to an .RData file
save.image("tuning_environment.RData")

# Load tuning results
#rf_tuned <- readRDS("rf_tuned.rds")

#knn_tuned <-
  #readRDS("knn_tuned.rds")

#boosted_tuned <-
  #readRDS("boosted_tuned.rds")


# Multinomial Logistic Classification
## Fits perfectly, no tuning required

# Random forest
#write_rds(rf_tune_res,
          #file = "Data/TunedModels/rf.rds")

# K-nearest neighbors
#write_rds(knn_tune,
          #file = "Data/TunedModels/knn.rds")

# Boosted Trees
#write_rds(boosted_tune_res,
          #file = "Data/TunedModels/boosted.rds")
```

6. Loading the RDS files in with their results, so we can use them later.

```
# Multinomial Logistic Classification
## Fits perfectly, no tuning required

# Random forest
rf_tuned <- read_rds(file = "/Users/angelalcantara/Downloads/PSTAT 131/Final Project/rf_tuned.rds")

# K-nearest neighbors
knn_tuned <- read_rds(file = "/Users/angelalcantara/Downloads/PSTAT 131/Final Project/knn_tuned.rds")

# Boosted trees
boosted_tuned <- read_rds(file = "/Users/angelalcantara/Downloads/PSTAT 131/Final Project/boosted_tuned
```

Model Results

We will now be loading the results from our models and collecting the important results (metrics) from each of them. We will then store them to figure what our best perfoming model was.

```
# Multinomial Logistic Classification
log_fit <- fit_resamples(log_workflow, resamples = spotify_folds)

select_best(log_fit, metric = "roc_auc")
```

```
## # A tibble: 1 x 1
##   .config
##   <chr>
## 1 Preprocessor1_Model1
```

```
select_best(knn_tuned, metric = "roc_auc")
```

```
## # A tibble: 1 x 2
##   neighbors .config
##       <int> <chr>
## 1        25 Preprocessor1_Model6
```

```
select_best(rf_tuned, metric = "roc_auc")
```

```
## # A tibble: 1 x 4
##    mtry trees min_n .config
##   <int> <int> <int> <chr>
## 1     2   233     7 Preprocessor1_Model41
```

```
select_best(boosted_tuned, metric = "roc_auc")
```

```
## # A tibble: 1 x 4
##   trees min_n learn_rate .config
##   <int> <int>      <dbl> <chr>
## 1   100     2       1.02 Preprocessor1_Model01
```

```
log_roc <- collect_metrics(log_fit) %>%
  filter(.metric == "roc_auc") %>%
  arrange(desc(mean)) %>%
  slice(1) %>%
  pull(mean)

knn_roc <- collect_metrics(knn_tuned) %>%
  filter(.metric == "roc_auc") %>%
  arrange(desc(mean)) %>%
  slice(1) %>%
  pull(mean)

rf_roc <- collect_metrics(rf_tuned) %>%
  filter(.metric == "roc_auc") %>%
```

```
  arrange(desc(mean)) %>%
  slice(1) %>%
  pull(mean)

boosted_roc <- collect_metrics(boosted_tuned) %>%
  filter(.metric == "roc_auc") %>%
  arrange(desc(mean)) %>%
  slice(1) %>%
  pull(mean)

library(tibble)

roc_auc_comparison <- tibble(
  Model = c("Multinomial Logistic", "KNN", "Random Forest", "Boosted Trees"),
  ROC_AUC = c(log_roc, knn_roc, rf_roc, boosted_roc)
)

# View the table
print(roc_auc_comparison)
```

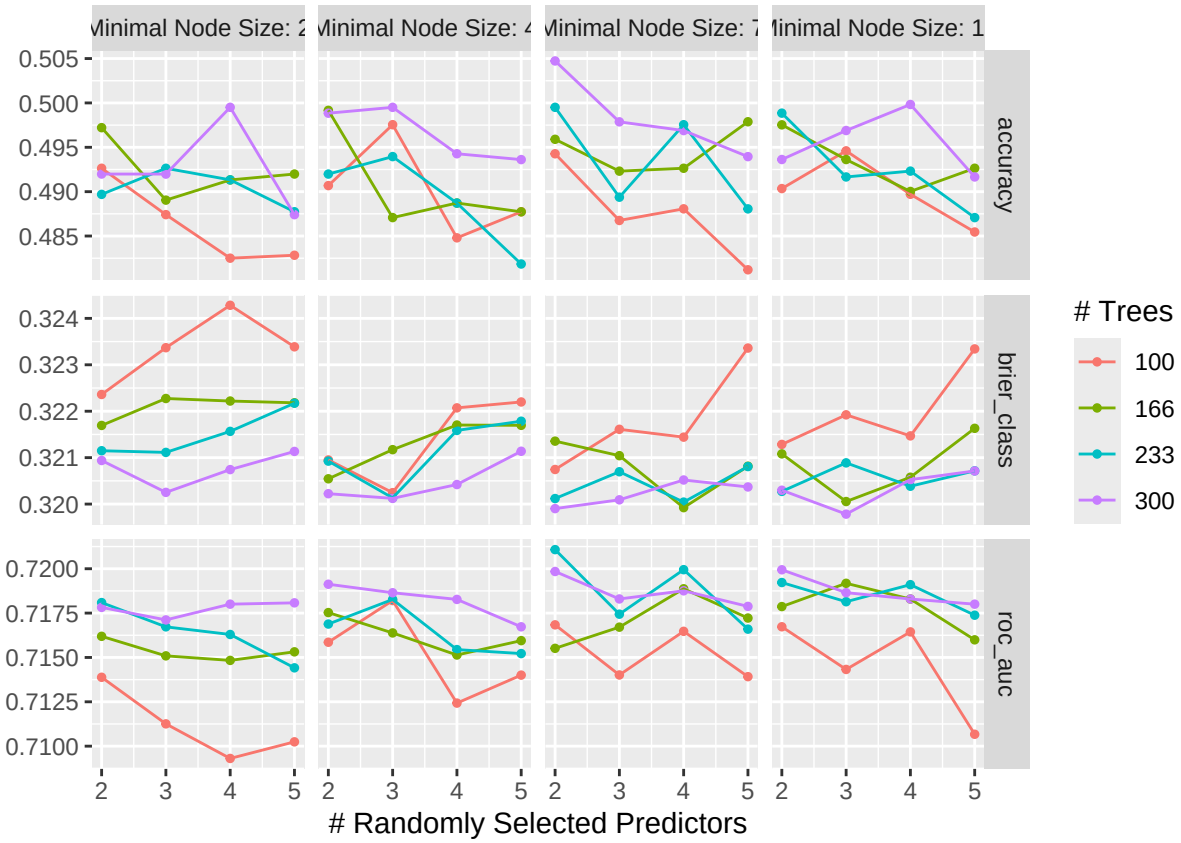
```
## # A tibble: 4 x 2
##   Model                ROC_AUC
##   <chr>                  <dbl>
## 1 Multinomial Logistic   0.679
## 2 KNN                    0.680
## 3 Random Forest          0.721
## 4 Boosted Trees          0.678
```

This table tells us that a random forest model performed the best on our training data because it has the highest ROC AUC.

Visualizing Results

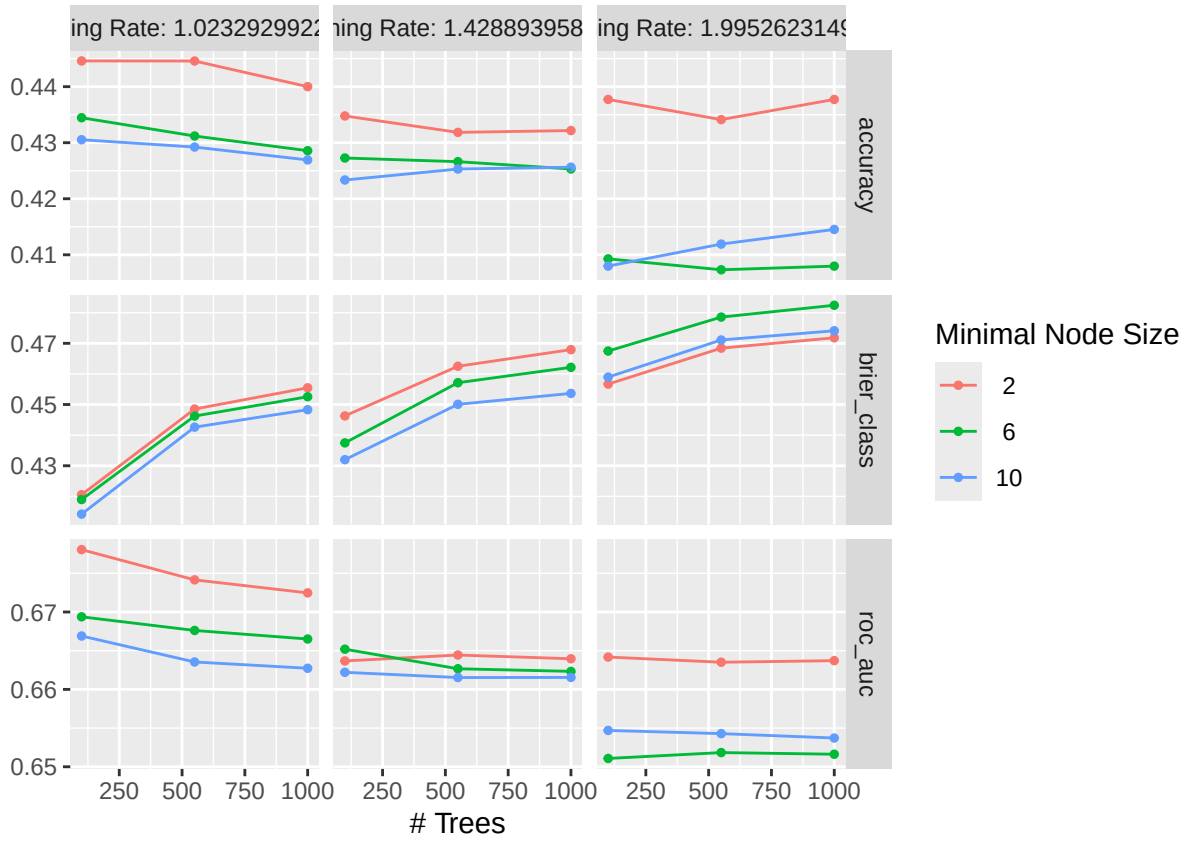
Random Forest Autoplot

```
autoplot(rf_tuned)
```



Boosted Trees Autoplot

```
autoplot(boosted_tuned)
```



Results of the Best Model

Performance on the Folds

The random forest model performed the best for the training data. What are the tuned parameters required to achieve the best model?

```
#
best_rf <- select_best(rf_tuned, metric = "roc_auc")

best_rf
```

```
## # A tibble: 1 x 4
##    mtry trees min_n .config
##   <int> <int> <int> <chr>
## 1     2   233     7 Preprocessor1_Model41
```

A random forest model with `mtry` = 2, `trees` = 233, and `min_n` = 7 are the parameters that gave the best model.

Fitting to Training Data

Now, we will take the best trained model to test it on our testing set.

Testing the Model

```
final_rf_wf <- finalize_workflow(
  rf_workflow,      # your original random forest workflow
  best_rf           # best tuning results (e.g., mtry, trees, min_n)
)

final_rf_fit <- fit(final_rf_wf, data = spotify_train)

# Predict class labels
rf_test_preds <- predict(final_rf_fit, new_data = spotify_test) %>%
  bind_cols(spotify_test %>% select(function_pair))

# Evaluate performance
library(yardstick)

# Accuracy, ROC AUC, etc.
metrics(rf_test_preds, truth = function_pair, estimate = .pred_class)
```

```
## # A tibble: 2 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.486
## 2 kap      multiclass     0.290
```

```
# Confusion matrix
conf_mat(rf_test_preds, truth = function_pair, estimate = .pred_class)
```

```
##           Truth
## Prediction  NF  NT  SJ  SP
##         NF 179  54  68  84
##         NT  59 182  36  78
##         SJ  16   4  30   6
##         SP  41  55  24 106
```

According to our table above, we achieved about a 49% accuracy in predicting function pairs. The confusion matrix tells us that NF and NT function pairs predicted the best, while SJ only had 29 correct, while SP was moderate with 104 correct.

Conclusion

Overall, this project was exciting to do and I truly enjoyed being able to use my machine learning skills in this course to be able to understand the connection between personlaity types and music taste. The model did not do a great job, but a 49% accuracy rate is not the worst. Therefore, I don't think that music will be sufficient enough to determine someone's personality.

Sources

Throughout the project I mentioned the sources that I used with links or I explained where I got it from.