ST310: Final Project

Spotification

1) Executive Summary

This analysis is an application of machine learning to a prediction of songs' popularity based on the characteristics of the audio. Overall, we fit 4 different models: linear regression, penalised regression, gradient descent on a more complicated linear regression model and random forest. While none of the model explains more than a third of the variation in popularity, there is a significant improvement moving from linear models to trees-based model, supporting the conjecture that the relationship may not be linear. However, since the RMSE is quite high, it might be the case that the popularity of a song is being influenced by the artist, more than the nature of the track itself.

2) Motivation

Have you ever wondered what makes a song popular? Does the success comes from the characteristics of the song or the artist name? In this project, we attempt to isolate any impacts from the artists from the features of the song and determine whether songs with certain features are likely to be more popular than others.

We conjecture that factors affecting 'popularity' are likely to be from other audio characteristics of a song such as 'energy', 'danceability', 'valence' or 'genre' etc. If a relationship exists, this could produce valuable models by predicting which songs people will enjoy before they've become popular, based on the 'intrinsic' value of the song and less so about the artist names attached to it. Hence this can help with the **song recommendation** features.

Whether audio features and popularity are related through a linear relationship or not is also of our interest. Given that the true relationship is unknown, different models are proposed and their predictive performance are compared. We start out with a **multiple linear regression** model with a few chosen predictors as our benchmark. Then, we extend this to produce a relatively interpretable models using **penalised regression** (ridge, Lasso and elastic net). **Mini-batch gradient descent** algorithm is also implemented on linear regression to see if it gives any improvement. Finally, a **random forest** model is built on the higher-dimensional version of the dataset (details are discussed below), focusing solely on the prediction accuracy.

It is reasonable to assume each track is independent of another, given that songs are usually written based on new concepts. We can also assume they share the same probability distribution, since all songs are judged based on the same criteria, all of which are normalised to the same scale. Hence it is reasonable to assume they are identically distributed.

3) Description of the Dataset

In this project, we analyse determinants of song popularity from a (dataset on Spotify tracks)[https://www.kaggle.com/datasets/maharshipandya/-spotify-tracks-dataset].

In particular, our original dataset covers 114,000 tracks. Each track has 21 audio features associated with it, ranging from artist name, popularity, duration, genre, 'acousticness', and tempo. All 'intangible' measures that cannot be measured directly such as 'acousticness', 'danceability', 'instrumentalness' have been normalised to a scale of 0-1.

The following lists 16 variables used in the analysis.

1. **popularity**: The popularity of a track is a value between 0 and 100, with 100 being the most popular. The popularity is calculated by algorithm and is based, in the most part, on the total number of plays the track has had and how recent those plays are. Generally speaking, songs that are being played a lot now will have a higher popularity than songs that were played a lot in the past. Duplicate tracks (e.g. the same track from a single and an album) are rated independently. Artist and album popularity is derived mathematically from track popularity.
2. **duration_ms**: The track length in milliseconds
3. **explicit**: Whether or not the track has explicit lyrics (true = yes it does; false = no it does not OR unknown)
4. **danceability**: Danceability describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity. A value of 0.0 is least danceable and 1.0 is most danceable
5. **energy**: Energy is a measure from 0.0 to 1.0 and represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy. For example, death metal has high energy, while a Bach prelude scores low on the scale
6. **key**: The key the track is in. Integers map to pitches using standard Pitch Class notation. E.g. 0 = C, 1 = C /D , 2 = D, and so on. If no key was detected, the value is -1
7. **loudness**: The overall loudness of a track in decibels (dB)
8. **mode**: Mode indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived. Major is represented by 1 and minor is 0
9. **speechiness**: Speechiness detects the presence of spoken words in a track. The more exclusively speech-like the recording (e.g. talk show, audio book, poetry), the closer to 1.0 the attribute value. Values above 0.66 describe tracks that are probably made entirely of spoken words. Values between 0.33 and 0.66 describe tracks that may contain both music and speech, either in sections or layered, including such cases as rap music. Values below 0.33 most likely represent music and other non-speech-like tracks
10. **acousticness**: A confidence measure from 0.0 to 1.0 of whether the track is acoustic. 1.0 represents high confidence the track is acoustic
11. **instrumentalness**: Predicts whether a track contains no vocals. "Ooh" and "aah" sounds are treated as instrumental in this context. Rap or spoken word tracks are clearly "vocal". The closer the instrumentalness value is to 1.0, the greater likelihood the track contains no vocal content
12. **liveness**: 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
13. **valence**: 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)
14. **tempo**: 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
15. **time_signature**: An estimated time signature. The time signature (meter) is a notational convention to specify how many beats are in each bar (or measure). The time signature ranges from 3 to 7 indicating time signatures of 3/4, to 7/4.
16. **track_genre**: The genre in which the track belongs.

Source: Pandya, Maharshi."Spotify Tracks Dataset." Kaggle, 22 Oct. 2022, https://www.kaggle.com/datasets/maharshipandya/-spotify-tracks-dataset.

4) Data cleaning

We did this data cleaning and formatting in python and then exported the resulting data sets as CSVs. Please find the python code used in the code file.

The 114,000 entries are comprised of samples of size 1000 for 114 unique values of 'track_genre'.

Firstly, note the difference between the variables `track_id` and `track_name`. `track_id` gives a string that identifies a track to a specific track on Spotify, whereas `track_name` gives the name of the song. `track_id` is closer to being a unique identifier than `track_name` so it would seem more plausible to use this as a means to identify entries. However this is not unique in the data because a specific track may be included multiple times under different `track_genre` values.

For example, here we can see that this song is listed under 9 genres. This means we can't drop duplicates based on `track_id` as doing so will delete data we wish to use in our predictions.

In addition if we use `track_name` to identify tracks, we also have duplicates of tracks as some tracks may be listed multiple times but have different `album_name` values. However, we cannot simply drop duplicates of `track_name` as some songs may be recorded by multiple artists and have different metrics in the variables we wish to use as predictors.

For example, the `track_name` value `halloween` appears 88 times with 21 different `track_id`'s, 6 artists and 12 albums.

To deal with this in a way that we don't lose data but also don't have duplicate entries, we aggregated the 114 values for `track_genre` into wider umbrella genres, creating a dummy variable for each. There is no need to drop one variable here as each `track_id` can have multiple genres, so there isn't a problem with multi-collinearity. We also dropped duplicates based on the `track_name` and `artists` variables, as this got rid of duplicates with the same metrics used for prediction, but kept duplicate tracks that contain different metrics.

Another reason why we aggregated the genres was to make it possible to use the `track_genre` variable without creating 113 dummy variables as `tidymodels` has a limit of 58 categories when converting qualitative data into factors. In addition, it allowed us to list tracks under multiple genres for songs which had fusion genres, such as `Rockabilly` which is a mixture of country and rock.

5) Exploratary Data Analysis

5.1) Load Libraries and Read in the Data

We will use the `tidymodels` package to facilitate our workflow, particularly with tuning the parameters. In addition, after the optimal paramters are chosen, some models may be fitted using their own stand-alone package for further analysis.

```r
library(tidyverse)
library(tidymodels) # initial_split()

library(GGally) # ggpairs()
library(corrplot) # corrplot()
library(gridExtra) # grid.arrange()
library(ggplot2) # gm_scatterplot

library(glmnet) # glmnet()
library(randomForest) # randomForest()
```

Since we exclude the analysis of popularity based on artist names, we first filter these irrelevant columns out.

```
data_raw <- read.csv("https://raw.githubusercontent.com/pparkkk/ST310_Spotify_Project/main/unique_data.
                     header = TRUE)
data_full <- select(data_raw, -X, -track_id, -artists, -album_name,
                    -track_name, -track_genre)

# Convert variables into numerical values
# 0 for FALSE, 1 for TRUE
data_full$explicit <- as.numeric(as.factor(data_full$explicit))-1

# Create new dummy showing if track has >1 genre
genres <- c("country", "electronic", "jazz", "pop", "r_n_b", "rock",
            "classical", "metal", "other", "world", "latin")

# Create a new variable counting the number of genres it has
data_full$ngenre <- as.numeric(rowSums(data_full[, genres]))

rm(data_raw)  # to free up the memory
```

Then, for a fair and accurate performance evaluation, the 'training' and 'test' sets of the data is being created.

```
# Split data into training and test set
set.seed(12345)
data_split <- initial_split(data_full)
data_train <- training(data_split)
data_test <- testing(data_split)
```

5.2) Exploratary Data Analysis

Before building any model, we perform an exploratory data analysis to see if there is any notable relationship that is worth exploring.

Distribution of Each Variable

```
# Histogram plots (for the popularity)
hist(data_train$popularity, main = "Popularity")

# Histogram plots (for the non-binary features)
par(mar = c(2,2,2,2))
col_names_non_binary <- c("duration_ms", "danceability", "energy", "key",
                          "loudness", "speechiness", "acousticness",
                          "instrumentalness", "liveness", "valence", "tempo",
                          "time_signature")

par(mfrow = c(4, 3))
for (i in seq_along(col_names_non_binary)){
  hist(data_train[,col_names_non_binary[i]], main=col_names_non_binary[[i]])
}

par(mfrow = c(1, 1))
# Select the binary columns from the data frame
binary_cols <- c("explicit", "mode", genres)

# Create frequency table for each column
freq_tables <- lapply(data_train[, binary_cols], table)
# Combine all frequency tables into one table
combined_freq_table <- do.call(cbind, freq_tables)
t(combined_freq_table) # View the combined frequency table
```

The distribution of `popularity` is slightly positively skewed with a lot of tracks having very low popularity and very few that are extremely popular. Inspecting the continuous predictors, we saw that some variables, such as `duration_ms`, `loudness`, `speechiness`, are highly positively skewed. Therefore, we may consider log-transforming the data before fitting a linear regression model to improve its predictive performance.

As for the binary categorical variables, there might be some class imbalances in the `explicit`, `mode` and different genres. However, we didn't consider any transformation given that the absolute frequency count should be sufficient when fitting the model and regression may be less affected by class imbalance, relative to classification problems.

Correlation plot

```
data_cor1 <- cor(data_train)
corrplot(data_cor1, method="square",
         col = rev(colorRampPalette(c("#B40F20", "#FFFFFF", "#2E3A87"))(100)),
         type="lower", tl.col="black", tl.srt=60, tl.cex = 0.6)
```

It seems that popularity doesn't seem to be linearly correlated with any of the predictors, but there are some features that seem to be more 'important' than others. So, we select certain features with high absolute correlation with popularity and explore their associations further.

ggpairs plot

Even though the magnitude of correlation seems to be low, hypothesis tests confirm that they are indeed significantly different from 0. The density plot shows that a proportion of the tracks have near-zero popularity, whilst very few are super popular. The distribution of each regressor also differ, with danceability being more symmetric whereas loudness and instrumentalness are highly negatively and postively skewed, respectively. With highly skewed distributions, log-transformation could help improve the performance.

```
ggpairs(data_train,
        columns = c("popularity", "danceability", "loudness", "instrumentalness"),
        lower = list(continuous = "smooth"), upper = list(continuous = "cor"))
```

Genres

Next, we explore potential relationships between popularity and genre. In particular, is there any genre that tends to be more popular than others? To address the issue of multiple genres, we first filter those with multiple genres, then specifically spell out the genre of tracks with only 1 genre.

```r
# Assign the genre name based on the dummy variables
get_genre_name <- function(x) {
  ifelse(x["ngenre"] == 2, "2_genres",
    ifelse(x["ngenre"] == 3, "3_genres",
      ifelse(x["ngenre"] == 4, "4_genres",
        ifelse(x["ngenre"] == 5, "5_genres",
          ifelse(x["country"] == 1, "country",
            ifelse(x["electronic"] == 1, "electronic",
              ifelse(x["jazz"] == 1, "jazz",
                ifelse(x["pop"] == 1, "pop",
                  ifelse(x["r_n_b"] == 1, "r_n_b",
                    ifelse(x["rock"] == 1, "rock",
                      ifelse(x["classical"] == 1, "classical",
                        ifelse(x["metal"] == 1, "metal",
                          ifelse(x["other"] == 1, "other",
                            ifelse(x["world"] == 1, "world", "latin"))))))))))))))
}
# Apply the function to each row of the data frame and create a new column with the genre names
temp_data <- data.frame(data_train)
temp_data$genre_name <- apply(data_train[, -1], 1, get_genre_name)

# Create a bar plot of mean popularity by genre
mean_popularity <- tapply(temp_data$popularity, temp_data$genre_name, mean)
barplot(mean_popularity, xlab = "Genre", ylab = "Mean Popularity",
        col = "steelblue", main = "Mean Popularity by Genre", las = 2,
        cex.names = 0.8)
```

It turns out that apart from `classical`, which tends to be the least popular if being on its own, and `pop` and `r_n_b`, which tend to be the most popular, genre doesn't seem to be a significant determinant of popularity.

6) Modelling

6.1) Set up the Tidymodels Framework

We first note that since the newly created variable, `ngenre`, is a linear combination of all the `genres` columns, we remove these in order to avoid the perfect collinearity issue.

```r
data_train <- data_train[,-28]
data_test <- data_test[,-28]

# Cross-validation for tuning the parameters
data_cv <- vfold_cv(data_train, v = 10)

# Pre-process the model
data_recipe <- data_train %>%
  recipe(popularity ~ .) %>%
  prep()
```

6.2) Baseline Model

We chose a multiple linear regression model with 3 parameters for a simple baseline for comparison to the more sophisticated models later.

The reason for choosing "danceability" is because it is one of the stongest (positively) correlated with popularity. Similarly for "classical", it is the strongest negatively correlated variable to popularity. "explicit" was chosen as we thought it could be interesting to see how this (indicator) variable affects popularity (if at all).

```r
# Fit baseline model on training set
baseline <- lm(popularity ~ explicit + danceability + speechiness,
               data = data_train)
summary_stats <- summary(baseline)
summary_stats
ggcoef(baseline)
```

Interpretation: As all slope coefficients are significantly different from zero, each predictor influences the popularity to a certain extent. Of the three regressors used in the baseline model, `speechiness` has the greatest impact, though with the widest confidence interval. Note that `speechiness` is normalised to be between 0 and 1. So, we may interpret its slope coefficient as: when `speechiness` increases by 0.1, it is associated with a decline in `popularity` by around 1.7. On the other hand, `explicit` and `danceability` affect popularity positively. The R-Squared is incredibly low, supporting the initial hypothesis that there is no strong linear relation between the predictors and outcome of interest: `popularity`.

Then, we use the model to predict on the test set.

```r
# Prediction
X <- cbind(rep(1, nrow(data_test)), data_test[,c(3,4,9)])
predictions_baseline <- as.matrix(X) %*% as.vector(coef(baseline))

# Calculate performance metrics
RMSE_baseline <- sqrt(mean((data_test$popularity - predictions_baseline)^2))
SSR <- sum((predictions_baseline - mean(data_test$popularity))^2)
SSE <- sum((predictions_baseline - data_test$popularity)^2)
SST <- sum((data_test$popularity - mean(data_test$popularity))^2)
RSQ_baseline <- SSR/SST

print("Testing: ")
cat("RMSE:", RMSE_baseline, "\n")
cat("R-squared:", RSQ_baseline, "\n")
```

Comment: The RMSE increases slightly compared to the training set, which is expected for a generalization. In addition, the R-squared decreases slightly to 2.1%. This implies that only 2.1% of the variation in popularity is being explained by the 3 predictors above. Although this is very poor, the model is very simple. So, we consider some extensions on this below.

6.3) Penalised Regression (Ridge / Lasso / Elastic Net)

This is a non-baseline model that is (relatively) interpretable. We perform the following steps:

- Defines a linear regression model with Lasso regularization using the `linear_reg()` function from the `parsnip` package

- `tune()` used to specify the hyperparameters `penalty` (P) and `mixture` (M)
- `set_engine()` used to specify the modeling engine used to fit the model (here we use `glmnet`)
- The resulting object `pen_reg_y` is a model specification object that can be further used for model training, tuning and prediction

```r
# Model specification for penalised linear regression
pen_reg_y <- linear_reg(penalty = tune('P'), mixture = tune('M')) %>%
  set_engine('glmnet')

# Set up the workflow
pen_reg_wf <- workflow() %>%
  add_recipe(data_recipe) %>%
  add_model(pen_reg_y)

# Tune the parameters
fit_pen_reg <- tune_grid(pen_reg_wf,
                         data_cv,
                         metrics = metric_set(rmse, mae, rsq),
                         control = control_grid(save_pred = TRUE))

# Select the best model with the smallest cross-validation RMSE
pen_reg_best <- fit_pen_reg %>%
  select_best(metric = 'rmse')

# Extract the optimal parameters
P_best <- pen_reg_best[1]
M_best <- pen_reg_best[2]

# Fit the final model
pen_reg_final <- finalize_model(pen_reg_y, pen_reg_best)
```

With the optimal parameters, we use the normal `glmnet` function to investigate which predictors are being treated as 'important' in predicting popularity.

```r
# Define X, y, data
ytrain <- data_train$popularity
Xtrain <- as.matrix(select(data_train, -1))

# Using best mixture (elastic net) to inspect coefficient path
glmnet_pre_best <- glmnet(Xtrain, ytrain,
                    family = "gaussian",
                    alpha = M_best)
x <- as.matrix(glmnet_pre_best$beta)
plot(glmnet_pre_best, xvar = "lambda")
```

Although the names of the variables didn't show up, but if we inspect the output manually, we see that `instrumentalness` (blue line) and `classical` (green line) are the last 2 predictors that survive, when heavy penalty is applied. This is followed by `speechiness` (red line). We may interpret these as the 3 most important predictors of popularity, given that their coefficients still show large magnitude when we restrict the number of non-zero slope coefficients.

Then, we use the best mixture and best lambda to predict on the test set.

```r
glmnet_best <- glmnet(Xtrain, ytrain,
                      family = "gaussian",
                      lambda = P_best,
                      alpha = M_best)
predictions_glmnet <- predict(glmnet_best, newx = as.matrix(data_test[,-1]))
```

Lastly, we inspect the coefficients to see which is the most important.

```r
# Inspect the coefficients
beta <- data.frame(Predictors = c("Intercept", colnames(Xtrain)),
                   Beta = as.vector(coef(glmnet_best)))
names(beta)[2] <- "Beta"

# Sort by the highest magnitude of beta
beta <- beta[order(abs(beta$Beta), decreasing = TRUE), ]
ggplot(beta, aes(x = Predictors, y = Beta)) +
  geom_bar(stat = "identity", color = "midnightblue", fill = "midnightblue") +
  theme_bw() +
  theme(axis.text.x = element_text(angle = 90, hjust = 0.9, vjust = 0.2),
        panel.grid.major = element_blank()) +
  scale_x_discrete(limits = beta$Predictors) +
  ggtitle("Slope Coefficient for Each Regressor") +
  labs(x = "Predictor", y = "Beta Coefficient")
```

Interpretation: As expected, `speechiness`, `classicial` and `instrumentalness` are the 3 most outstanding predictors and all are negatively correlated with popularity.

```r
# Calculate performance metrics
RMSE_glmnet <- sqrt(mean((data_test$popularity - predictions_glmnet[,1])^2))
SSR <- sum((predictions_glmnet - mean(data_test$popularity))^2)
SSE <- sum((predictions_glmnet - data_test$popularity)^2)
SST <- sum((data_test$popularity - mean(data_test$popularity))^2)
RSQ_glmnet <- SSR/SST

print("Testing: ")
cat("RMSE:", RMSE_glmnet, "\n")
cat("R-squared:", RSQ_glmnet, "\n")
```

Comparison to baseline model: The elastic net model gives better accuracy. This can be seen through the lower RMSE. The R-Squared has also improved from 0.02 to 0.12. Despite this, it is still very low. This agrees, to some extent, with the correlation plot that perhaps the relationship is non-linear. Therefore, this motivates one of the models that followed that was complex and non-linear: Random forest.

6.4) Mini-batch Gradient Descent

Although it seems that the relationship is not linear, we have another go at implementing mini-batch gradient descent on linear regression with a (pseudo-Huber loss)[https://en.wikipedia.org/wiki/Huber_loss] function. For conciseness, the source code in this section is hidden from the PDF version of the report, but it can be found in the Rmd file.

Pseudo-Huber Loss Function

Instead of minimising the sum of squares in linear regression or using the L_1-loss function in Lasso, we use a differentiable mixture of the two. It is a convex function, especially when close to the minimum, so ensuring that we will get a global minimum rather than a local one. The parameter δ controls the steepness of the loss function.

Source: Wikipedia, "Huber Loss." Wikipedia, 17 Feb. 2022, https://en.wikipedia.org/wiki/Huber_loss.

The pseudo-Huber loss function is defined as:

$$L_\delta(a) = \delta^2\left(\sqrt{1+\left(\frac{a}{\delta}\right)^2}-1\right)$$

where a is the residuals from linear regression.

```r
pseudo_huber_loss <- function(y, X, beta, delta = 1.2) {
  a <- y - X %*% beta
  loss <- delta^2 * (sqrt(1 + (a / delta)^2) - 1)
  return(sum(loss))
}
```

We illustrate the pseudo-Huber loss function for different values of δ, with particular emphasis on the change of curvature from L_1 loss function to L_2 around $a = 0$.

```r
sim_data <- data.frame(a = seq(-4, 4, length.out = 1000),
                       L1 = numeric(1000),
                       L2 = numeric(1000),
                       delta_2 = numeric(1000),
                       delta_5 = numeric(1000),
                       delta_10 = numeric(1000))

sim_data$L1 = abs(sim_data$a)
sim_data$L2 = sim_data$a^2
i <- 4
for (delta in c(2, 5, 10)) {
  sim_data[, i] <- delta^2 * (sqrt(1 + (sim_data$a / delta)^2) - 1)
  i <- i + 1
}

ggplot(sim_data) +
  geom_line(aes(a, L1, color = "L1"), linetype = 2) +
  geom_line(aes(a, L2, color = "L2"), linetype = 2) +
  geom_line(aes(a, delta_2, color = "2")) +
  geom_line(aes(a, delta_5, color = "5")) +
  geom_line(aes(a, delta_10, color = "10")) +
  labs(title = "Pseudo-Huber VS L1 VS L2 Loss Function",
       x = "Residuals", y = "Loss", color = "Delta") +
  ylim(0, 16)
```

Comment: The higher the value of δ, the steeper the loss function gets for residuals of greater magnitude, thus penalising them more and approximate the L_2 loss function more closely. Alternatively, δ can also be thought of as the point where the pseudo-Huber loss function changes from L_1 to L_2 loss. However, even if $\delta \to \infty$, there is a limit to how the loss function tends towards L_2. In fact, the limiting loss function is not far from that of $\delta = 10$.

Ideally, δ is another tuning parameter in the gradient descent algorithm. However, given the amount of data and the need to optimise the execution time, we decided to fix $\delta = 5$. The rationale behind this is that we don't want outliers to have as much impact as it was in L_2 loss function, but still want to penalise them more heavily than others. Hence, we seek a balance between L_1 and L_2 loss function, but closer to L_1.

Gradient Function

Since the pseudo-Huber loss function is differentiable everywhere, we use exact differentiation in the algorithm. Note that the result is being cross-checked with numeric differentiation. It's partial derivative is given by:

$$\frac{\partial L_\delta(a, \beta_j)}{\partial \beta_j} = \frac{-a x_j}{\sqrt{1 + \left(\frac{a}{\delta}\right)^2}} \qquad \text{for } j = 1, 2, \dots, p$$

```r
gradient_vector <- function(y, X, beta, delta = 1.2) {
  gradient <- rep(1, ncol(X))

  a <- y - X %*% beta
  scale_factor <- -a / sqrt(1 + (a / delta)^2)
  for (j in 1:ncol(X)) {
    gradient[j] <- sum(scale_factor * X[,j])
  }
  return(gradient)
}
```

Updating the Weights

For each batch of data, we then compute the gradient and update the weights (betas). The direction of the update is being controlled by the sign of the gradient. By normalising the gradient to have magnitude of 1, we control the size of the step by the learning rate. The weights are updated according to the equation below.

$$\beta_{i+1} = \beta_i - \alpha \frac{L'_\delta(a; \beta_i)}{||L'_\delta(a; \beta_i)||_2}$$

where the learning rate, α is a tuning parameter.

```r
update_beta <- function(y_batch, X_batch, beta, delta = 1.2, alpha = 0.8) {
  # Calculate gradient
  gradient <- gradient_vector(y_batch, X_batch, beta, delta)

  l2_norm <- sqrt(sum(gradient)^2)

  # Update beta
  new_beta <- beta - alpha * gradient / (l2_norm + 10^-9)  # to avoid division by 0

  return(new_beta)
}
```

Mini-batch Gradient Descent for 1 Iteration

We then write a function to perform mini-batch gradient descent, where a batch of data is randomly sampled and used to update the weights. We repeat this until all batches have been used. The number of batches is calculated such that each data point is in exactly 1 batch, in expectation.

```r
mini_batch_gd <- function(y, X, batch_size = 10000,
                          init_beta = rep(0.1, ncol(X)),
                          delta = 1.2, alpha = 0.8) {
  # Number of batches
  num_batches <- ceiling(nrow(X)/batch_size)

  for (i in 1:num_batches) {
    # Randomly sample the batch with replacement
    batch_index <- sample(1:nrow(X), batch_size, replace = TRUE)
    y_batch <- y[batch_index]
    X_batch <- X[batch_index, ]

    # Update the beta
    updated_beta <- update_beta(y_batch, X_batch, init_beta,
                                delta = delta, alpha = alpha)

    init_beta <- updated_beta
  }

  return(init_beta)
}
```

Adding Stopping Criteria

Next, we add a stopping criteria for the algorithm. It will stop when either a certain error tolerance is reached, where an iteration is counted when all batches have been used to update the beta. So, the number of iterations here is equivalent to an epoch.

To avoid over-stepping when getting closer to the minimum, we use a decaying learning rate in the algorithm. Note that the same learning rate α is used to update each batch in an epoch, but this will decay as the number of iterations increases.

```r
gradient_descent_alg <- function(y, X, tol = 1e-06, batch_size = 10000,
                                 init_beta = rnorm(ncol(X)),
                                 delta = 1.2, alpha = 0.8) {
  # 1st iteration
  iter <- 1
  prev_beta <- mini_batch_gd(y, X, batch_size, init_beta, delta, alpha)
  prev_loss <- pseudo_huber_loss(y, X, prev_beta, delta)

  err_reduction <- 10 * tol
  while (err_reduction > tol) {
    # Next iteration of mini-batch GD
    # Use decaying learning rate
    new_beta <- mini_batch_gd(y, X, batch_size, init_beta = prev_beta,
                              delta = delta, alpha = alpha^iter)
    new_loss <- pseudo_huber_loss(y, X, new_beta, delta)
    # Quantify the improvement
    err_reduction <- abs(prev_loss - new_loss)

    # Update the variables
    iter <- iter + 1
    prev_beta <- new_beta
    prev_loss <- new_loss
  }

  return(prev_beta)
}
```

Cross-validation to Select the Optimal Hyperparameters

We have 2 parameters to tune: 1. Batch size 2. Learning rate α

The optimal parameters are chosen using the k-fold cross validation based on the RMSE. Note that the algorithm is still being calculated on the pseudo-Huber loss function. The reason behind the choice of RMSE is for the comparability with other methods and it has a much lower magnitude than the corresponding loss from the pseudo-Huber loss function.

```r
kfold_cv <- function(y, X, k = 10, batch_size = 10000, tol = 1e-05,
                     init_beta = rnorm(ncol(X)),
                     delta = 1.2, alpha = 0.8) {
  # Divide the dataset into k folds
  index <- ceiling(nrow(X) / k)

  # Empty vector to store test error for each fold
  error_vec <- c()

  # Cross-validation
  for (fold in 1:k) {
    start <- (fold - 1)*index + 1
    end <- min(nrow(X), start + index - 1)
    yvalid <- y[start:end]
    Xvalid <- X[start:end, ]

    ytrain <- y[-(start:end)]
    Xtrain <- X[-(start:end), ]

    # Mini-batch gradient descent and calculate test error
    beta_GD <- gradient_descent_alg(ytrain, Xtrain, tol = tol, batch_size,
                                    init_beta = init_beta, delta, alpha)
    test_error <- pseudo_huber_loss(yvalid, Xvalid, beta_GD, delta = delta)
    error_vec <- c(error_vec, test_error)
  }

  # Return mean of the test error
  return(mean(error_vec))
}
```

Implementing the algorithm

From the EDA stage, some of the variables are highly skewed. Therefore, log-transformation is applied to them before fitting the algorithm. Note that `loudness` is being measured in decibels with negative values, so we first need to translate it such that the minimum value is 1 before applying the transformation. Morever, a column of intercept is being added to the matrix of regressors.

```r
# Log transform highly skewed variables
data_train$duration_ms <- log(data_train$duration_ms + 1)
data_train$loudness <- log(data_train$loudness - min(data_train$loudness) + 1)
data_train$speechiness <- log(data_train$speechiness + 1)
data_train$acousticness <- log(data_train$acousticness + 1)
data_train$instrumentalness <- log(data_train$instrumentalness + 1)
data_train$liveness <- log(data_train$liveness + 1)

y <- data_train$popularity
X <- as.matrix(data_train[,-1])

# Add the intercept
X <- cbind(rep(1, length(y)), X)
```

Finally, we are able to implement the algorithm across a grid of parameters and visualise the results.

```r
# Grid of parameters
batch_size <- seq(1000, 2000, by = 500)
alpha <- seq(0.05, 0.3, by = 0.05)
delta <- 5

# Empty matrix to store data
result <- array(dim = c(length(batch_size), length(alpha)),
                dimnames = list(batch_size, alpha))
```

As the starting point, we initialise the algorithm with the coefficients of a usual linear regression model (minimising RSS). This is because we expect the optimal vector of weights to not be far away from these values, hence help reducing the execution time.

```r
# Initialise beta with the coefficient from lm
fit <- lm(popularity ~ ., data = data_train[,-28])
start_beta <- as.vector(coef(fit))

# By batch_size
for (i in 1:length(batch_size)) {
  # By alpha
  for (j in 1:length(alpha)) {
    result[i, j] <- kfold_cv(y, X, k = 10,
                             batch_size = batch_size[i],
                             tol = 1e-05,
                             init_beta = start_beta,
                             delta = delta, alpha = alpha[j])
  }
}
```

Then, we visualise the results by plotting the RMSE against the batch size for each learning rate α. Given that there is an extremely high value, we discard that from the plot to zoom into the main part.

```r
result <- as.data.frame(result)
result

# Discard the one with very high RMSE for visualisation purpose
ymax <- sort(as.matrix(result), decreasing = TRUE)[2]

ggplot(result) +
  geom_line(aes(x = batch_size, y = result[, 1], col = alpha[1])) +
  geom_line(aes(x = batch_size, y = result[, 2], col = alpha[2])) +
  geom_line(aes(x = batch_size, y = result[, 3], col = alpha[3])) +
  geom_line(aes(x = batch_size, y = result[, 4], col = alpha[4])) +
  geom_line(aes(x = batch_size, y = result[, 5], col = alpha[5])) +
  ggtitle(paste("RMSE for Each Batch Size and Learning Rate with Delta = ",
              delta, sep = "")) +
  labs(x = "Batch Size", y = "RMSE", color = "Learning Rate") +
  ylim(min(result), ymax)
```

Overall, the RMSE are quite similar in magnitude, with a few outliers. Then, we select the optimal parameters ie. the pair that minimises RMSE.

```r
paste("Minimum Loss: ", round(min(result),4))

# Identify the index
best_size = batch_size[which(result == min(result), arr.ind = TRUE)[1]]
best_alpha = alpha[which(result == min(result), arr.ind = TRUE)[2]]

print("Optimal Parameters")
cat("Best batch size:", best_size, "\n")
cat("Best learning rate:", best_alpha, "\n")
```

Refit the Algorithm on the Full Training Data

```r
gd_beta <- gradient_descent_alg(y, X, tol = 1e-10, batch_size = best_size,
                                  init_beta = start_beta,
                     delta = delta, alpha = best_alpha)
```

Analyse the coefficients

```r
beta <- data.frame(Predictors = c("Intercept", colnames(X)[-1]),
                   Beta = gd_beta)
names(beta)[2] <- "Beta"

# Sort by the highest magnitude of beta
beta <- beta[order(abs(beta$Beta), decreasing = TRUE), ]

ggplot(beta, aes(x = Predictors, y = Beta)) +
  geom_bar(stat = "identity", color = "midnightblue", fill = "midnightblue") +
  theme_bw() +
  theme(axis.text.x = element_text(angle = 90, hjust = 0.9, vjust = 0.2),
        panel.grid.major = element_blank()) +
  scale_x_discrete(limits = beta$Predictors) +
  ggtitle("Slope Coefficient for Each Regressor") +
  labs(x = "Predictor", y = "Beta Coefficient")
```

Comment: The top 6 predictors, as measured by the magnitude of its corresponding slope coefficients, include 3 audio characteristics: `speechiness`, `instrumentalness` and `danceability`, and 3 genres: `classical`, `acoustic` and `r_n_b`. Afterwards, the magnitude of the coefficients are significantly lower. This agrees with the correlation plot since these variables seem to have higher correlation with popularity, despite being quite low in absolute value. It also agrees with the elastic net model.

Evaluate the Performance on the Training Set

```r
# Prediction
ytrain_pred <- X %*% gd_beta

# Evaluation
residual <- y - ytrain_pred
RMSE_gd_train <- sqrt(mean(residual^2))
TSS_train <- sum((y - mean(y))^2)
SSR_train <- sum((ytrain_pred - mean(y))^2)
RSQ_gd_train <- SSR_train / TSS_train
ntrain <- length(y)
K <- ncol(X)
adRSQ_gd_train <- 1 - (1 - RSQ_gd_train) * (ntrain-1) / (ntrain - K)

# Print the value
print("Gradient Descent: ")
cat("RMSE:", RMSE_gd_train, "\n")
cat("R-squared:", RSQ_gd_train, "\n")
cat("Adjusted R-squared:", adRSQ_gd_train, "\n")
```

Comment: The RMSE of 18.22 is very similar to the elastic net model. It can explain about 12.3% of the variation in popularity, having adjusted for the number of predictors used. However, the performance is evaluated on the same data set that was used to train the model. Therefore, to better assess the predictive accuracy, we evaluate the model on the unseen test data.

Evaluate the Performance on the Test Set

Firstly, we perform the same transformations to the regressors before predicting the popularity.

```r
# Log transformation
data_test$duration_ms <- log(data_test$duration_ms + 1)
data_test$loudness <- log(data_test$loudness - min(data_test$loudness) + 1)
data_test$speechiness <- log(data_test$speechiness + 1)
data_test$acousticness <- log(data_test$acousticness + 1)
data_test$instrumentalness <- log(data_test$instrumentalness + 1)
data_test$liveness <- log(data_test$liveness + 1)

ytest <- data_test$popularity
```

```r
Xtest <- as.matrix(data_test[,-1])
Xtest <- cbind(rep(1, length(ytest)), Xtest)  # Add the intercept

# Prediction
gd_pred <- Xtest %*% gd_beta
```

Then, we evaluate the test performance using RMSE and adjusted R-squared.

```r
# Compute the metrics
RMSE_gd <- sqrt(mean((ytest - gd_pred)^2))
TSS <- sum((ytest - mean(ytest))^2)
SSR <- sum((gd_pred - mean(ytest))^2)
RSQ_gd <- SSR / TSS
n <- length(ytest)
K <- ncol(X)
adRSQ_gd <- 1 - (1 - RSQ_gd) * (n-1) / (n - K)

# Print the value
print("Gradient Descent: ")
cat("RMSE:", RMSE_gd, "\n")
cat("R-squared:", RSQ_gd, "\n")
cat("Adjusted R-squared:", adRSQ_gd, "\n")
```

Interpretation: RMSE is now 18.20, slightly lower than the training RMSE, which is to be expected given it is a generalisation to another unseen data. The fact that RMSE hasn't increase much suggest that the algorithm didn't overfit the data. However, the adjusted R-squared stayed relatively flat at 12.2%.

Comparing to the baseline model, this is a big improvement in adjusted R-squared by a factor of 6, even though the RMSE only fall marginally. However, the gradient descent algorithm performs similarly to the elastic net one. A potential extension is to consider fitting the gredient descent algorithm on a penalised regression. In addition, using different `start_beta` is also a consideration since the algorithm will converge to different final set of coefficients.

6.5) Random forest

For our high dimensional model we chose to implement random forests. Initially we wanted to tune all three hyperparameters, however this proved to be too long computationally to process. Instead we tuned the number of predictors sampled at each node (`mtry`) as we thought this would have the most impact, and then separately tuned the other variables with the tuned `mtry`. Most values gave almost identical metrics, so we kept the initial values as they were the same to three decimal places.

6.5.1) Random Forests with Grouped Genres

The first random forest model below is with the data that has the wider genre categories. The code below shows how we tuned the `mtry` hyperparameter using `workflows` and `recipes` from `tidymodels`. We also chose to do only 2 cross validation folds as the computation time was quite long, however in earlier tries we did 5 folds, and the results were negligibly different.

```r
# Recipe for random forest with larger genre categories
rf_recipe <- recipe(popularity ~., data=data_train) %>%
  prep()
```

```r
#Train mtry
rf_spec_1 <- rand_forest(mtry = tune(), trees = 300, min_n = 50) %>%
  set_engine('randomForest') %>%
  set_mode('regression')

rf_grid_1 <- grid_regular(finalize(mtry(),data_train), levels = 5)

rf_cv <- vfold_cv(data_train, 2)

rf_wf_1 <- workflow() %>%
  add_model(rf_spec_1) %>%
  add_recipe(rf_recipe)

rf_res_1 <- rf_wf_1 %>%
  tune_grid(resamples = rf_cv, grid = rf_grid_1)

# Print the result
rf_res_1$.metrics[[1]]
```

We see that `mtry` $= 7$ gives the best values for RMSE and rsq.

After tuning the parameters, we fit the final model on the whole training set and tested on the test data.

```r
# Model on full training data
rf_spec_1_final <- rand_forest(mtry = 7,trees = 300,min_n = 50,) %>%
  set_engine('randomForest') %>%
  set_mode('regression')

rf_wf_1_fin <- workflow() %>%
  add_model(rf_spec_1_final) %>%
  add_recipe(rf_recipe)

rf_model_1 <- fit(rf_wf_1_fin, data_train)
rf1_predict <- augment(rf_model_1, data_test)
```

```r
rf1_predict <- read.csv('https://raw.githubusercontent.com/pparkkk/ST310_Spotify_Project/main/rf1_predi
                        header = TRUE)
```

Then, we manually calculate the performance metric for comparability.

```r
RMSE_rf1 <- sqrt(mean((rf1_predict$popularity - rf1_predict$.pred)^2))
TSS_rf1 <- sum((rf1_predict$popularity - mean(rf1_predict$popularity))^2)
SSR_rf1 <- sum((rf1_predict$.pred - mean(rf1_predict$popularity))^2)
RSQ_rf1 <- SSR_rf1 / TSS_rf1
n_rf1 <- nrow(data_test)
K_rf1 <- ncol(data_test) - 1
adRSQ_rf1 <- 1 - (1 - RSQ_rf1) * (n_rf1 - 1) / (n_rf1 - K_rf1)

print("Random Forest: ")
cat("RMSE:", RMSE_rf1, "\n")
cat("R-squared:", RSQ_rf1, "\n")
cat("Adjusted R-squared:", adRSQ_rf1, "\n")
```

Interpretation: The random forest outperform all of the previous linear models, explaining 19% of the variation in popularity. This supports the hypothesis that the relationship between popularity and songs characteristics is nonlinear, and potentially not strong ie. popularity is being influenced by the artist name and their track record.

Tidymodels however in it's efforts to standardize the process meant that we were unable to retrieve the importance metrics associated with randomForest, so here we've rerun the model to gather the importance metrics, and then saved as a csv to avoid rerunning.

```
# Random forest without tidymodels to gather importance metrics
rf_imp <- randomForest(data_train[,-1], data_train$popularity, ntree = 300, mtry = 7, min_n = 50, impor
rf_1_importance <- importance(rf_imp)

# Order by importance
rf_1_importance[order(rf_1_importance$IncNodePurity, decreasing = TRUE), ]
```

Interpretation: Since higher value of `%IncMSE` is associated with more important predictors, it is somewhat surprising that `duration_ms` and `energy` are the most important ones, given that they have very low magnitude in all of the above linear models. Nevertheless, `instrumentalness`, `acousticnes speechiness` and `classical` are still deemed important.

6.5.2) Random Forests with Original Genres

We then wanted to try a higher dimension random forest which uses the original `track_genre` values as predictors. To do this, we utilised the `step_dummy` function in the recipe which allowed us to convert the 114 different variables to 113 dummy variables.

It's worth noting that this model required a different version of the original dataset. The dataset used above had no repeated `track_id` values, and compiled the varying data from `track_genre` into binary variables of wider genres. As here we are not using wider genres and there are repeats of tracks being listed with different genres, we were not able to make `track_id` unique. Instead this data just drops entries that have the same `track_id`, `track_genre` and `artist`. This mainly included tracks that were repeated due to being released on different albums.

```
data2_raw <- read.csv('https://raw.githubusercontent.com/pparkkk/ST310_Spotify_Project/main/original_ge
data2 <- select(data2_raw, -X, -artists, -album_name, -track_name)
data_split2 <- initial_split(data2)
data_train2 <- training(data_split2)
data_test2 <- testing(data_split2)
```

Then, we repeat the same process with the new dataset.

```
rf_recipe2 <- recipe(popularity ~., data=data_train2) %>%
  step_dummy(track_genre) %>%
  update_role(track_id, new_role = "ID") %>%
  prep()

rf_spec2 <- rand_forest(mtry = tune(), trees = 300, min_n = 50,) %>%
  set_engine('randomForest') %>%
  set_mode('regression')

rf_grid2 <- grid_regular(finalize(mtry(), data_train2), levels = 5)
```

```r
rf_cv2 <- vfold_cv(data_train2, 2)

rf_wf2 <- workflow() %>%
  add_model(rf_spec2) %>%
  add_recipe(rf_recipe2)

rf_res2 <- rf_wf2 %>%
  tune_grid(resamples = rf_cv2, grid = rf_grid2)

# Print the result
rf_res2$.metrics[[1]]
```

Now, we can see `mtry` = 17 has the best results, so this is what we shall use in the final model. Intuitively, this makes sense as we now have much more predictors available.

Next, we refit the best model and use it to predict on the test set.

```r
rf_spec2_fin <- rand_forest(mtry = 17,trees = 300,min_n = 50) %>%
  set_engine('randomForest') %>%
  set_mode('regression')

rf_wf2_fin <- workflow() %>%
  add_model(rf_spec2_fin) %>%
  add_recipe(rf_recipe2)

rf_model_2 <- fit(rf_wf2_fin, data_train2)
rf2_predict <- augment(rf_model_2, data_test2)
```

Below are the manual calculations of the larger random forest's predictions. We can see a substantial improvement from the random forest with wider genre categories.

```r
RMSE_rf2 <- sqrt(mean((rf2_predict$popularity - rf2_predict$.pred)^2))
TSS_rf2 <- sum((rf2_predict$popularity - mean(rf2_predict$popularity))^2)
SSR_rf2 <- sum((rf2_predict$.pred - mean(rf2_predict$popularity))^2)
RSQ_rf2 <- SSR_rf2 / TSS_rf2
n_rf2 <- nrow(data_test2)
K_rf2 <- ncol(data_test2)
adRSQ_rf2 <- 1 - (1 - RSQ_rf2) * (n_rf2-1) / (n_rf2 - K_rf2)

print("Random Forest: ")
cat("RMSE:", RMSE_rf2, "\n")
cat("R-squared:", RSQ_rf2, "\n")
cat("Adjusted R-squared:", adRSQ_rf2, "\n")
```

Interpretation: The adjusted R-squared has jumped to 28% with RMSE dropping to 16. It is no surprising that the full random forest model performs even better, with more information being supplied and the ability to deal with any interactions between predictors.

In order to gather the importance metrics like we did on the previous random forest, we first have to create dummy variables for the `track_genre` values. To do this, we've used `dummy_cols` from the `fastDummies` package.

```r
# Pre-process the data
library(fastDummies)
data3 <- dummy_cols(data2, select_columns = 'track_genre', remove_first_dummy = TRUE)
data_split3 <- initial_split(data3)
data_train3 <- training(data_split3)
data_test3 <- testing(data_split3)
```

We then fitted the models with the same hyperparameters as above.

```r
# Fit the model
rf_2_imp <- randomForest(data_train3[,-1:-2], data_train3$popularity,
                         ntree = 300, mtry = 17, min_n = 50, importance = TRUE)
rf_2_importance <- importance(rf_2_imp)

# Order by importance based on %IncMSE
rf_2_importance[order(rf_2_importance$X.IncMSE, decreasing = TRUE), ]
```

Interpretation: With the full data, we see that certain genres are very important. In fact, all of the top 5 predictors are all different genres of a song, ranking higher than all of the predictors that were "important" in other models. Whilst `acousticness` just made it to the top 10, the significance of `instrumentalness` and `speechiness` have now faded, with `valence`, `energy` and `duration_ms` replacing them.

7) Conclusion

In summary, random forest as a method outperforms linear models. This is not surprising given the non-linearity nature of the data and its high-dimensionality, especially with the full dataset. By grouping the `track_genre` into 11 categories, we lost some valuable information.

Despite a meaningful improvement in prediction accuracy, the model still performs poorly, explaining less than a third of the variation in `popularity`. As a potential area for future research, we may consider incorporating the name of the artist using natural language processing to investigate whether they are affecting the popularity of the song or not.