Genre Prediction
SENG 550 Final Report
Dec 15th, 2023

Tony Vo (30091585)
Chace Nielson (30045674)
Xian Wei Low (30113016)
Chad Holst (30105378)

Table Of Contents

Introduction and Motivation

In this project, we will be using PySpark and machine learning to handle a music lyric database. The database information is the Genius Song Lyrics obtained through the Kaggle[1] with each record containing lyrics and other text information related to a song. Using the data gathered from this dataset, we will be applying sentiment analysis to the lyrics, titles, and artists of rap and pop songs. The sentiment scores will be the basis for creating machine learning models to predict if a song genre is pop or rap.

Sentiment analysis looks at a word and gives it a score using a preset dictionary of words and scores. Since each word can have a sentiment score a line of text can have a total sentiment based on the total of each word or an average sentiment score for each word. Ultimately the sentiment scores will allow us to determine whether the piece of text is more negative or positive. In this project, the sentiment scores will be calculated using AFINN[5] which is another dataset that has 2,476 stem words that have been rated for their sentiment based on a scale between -5 (negative) and 5 (positive). When applying the data from the AFINN dataset to the lyrics and title, we will be able to find out whether a song will be positive or negative based on the value the lyrics and titles got. Six metrics will be created based on sentiments. The first three are the average sentiment scores for lyrics, artist and song title. The second three are the total sentiment scores for lyrics, artist and song title. These metrics, combined with the year and number of views will make up the features of the dataset.

Machine learning will use the substantial dataset and eight distinct features, including total sentiment and average sentiment for lyrics, title, and artist, as well as year and number of views. Using supervised learning with genre labels, the model aims to discern patterns and construct a predictive framework. Rap and pop songs were chosen as the classification labels because we believed that rap would have a more negative sentiment score when compared to pop music. Since we believe that these two genres will produce a notable difference between their sentiment scores, the model should produce higher accuracy. By focusing on rap and pop genres, and anticipating a substantial difference in sentiment scores, the model is expected to achieve higher accuracy.

The Machine learning section will use four models. These include Logistic Regression with LBFGS, Decision Tree, Random Forest and Gradient Boosted Trees(GBT). While they are not all strictly classification models they are each capable of predicting a binary output. Using the eight features will ideally determine whether a song belongs to the rap or pop genre. The primary goal of the project is to explore the feasibility of predicting a song's genre from only the text information found in its lyrics title and artist and their corresponding sentiment scores.

Project Goals

In this project, we have multiple goals that we will have to complete so that our overall motivation for the creation of this project can be fulfilled. Firstly, we will have to collect our data from the Kaggle database[1]. We will also reduce the data set by removing other songs with genres other than rap and pop and non-English songs. Next, we will have to set up a Google Colab[3] which will be our main tool for the creation of the project. This will require us to set up any libraries that we use and set up any environment variables that are used throughout the project.

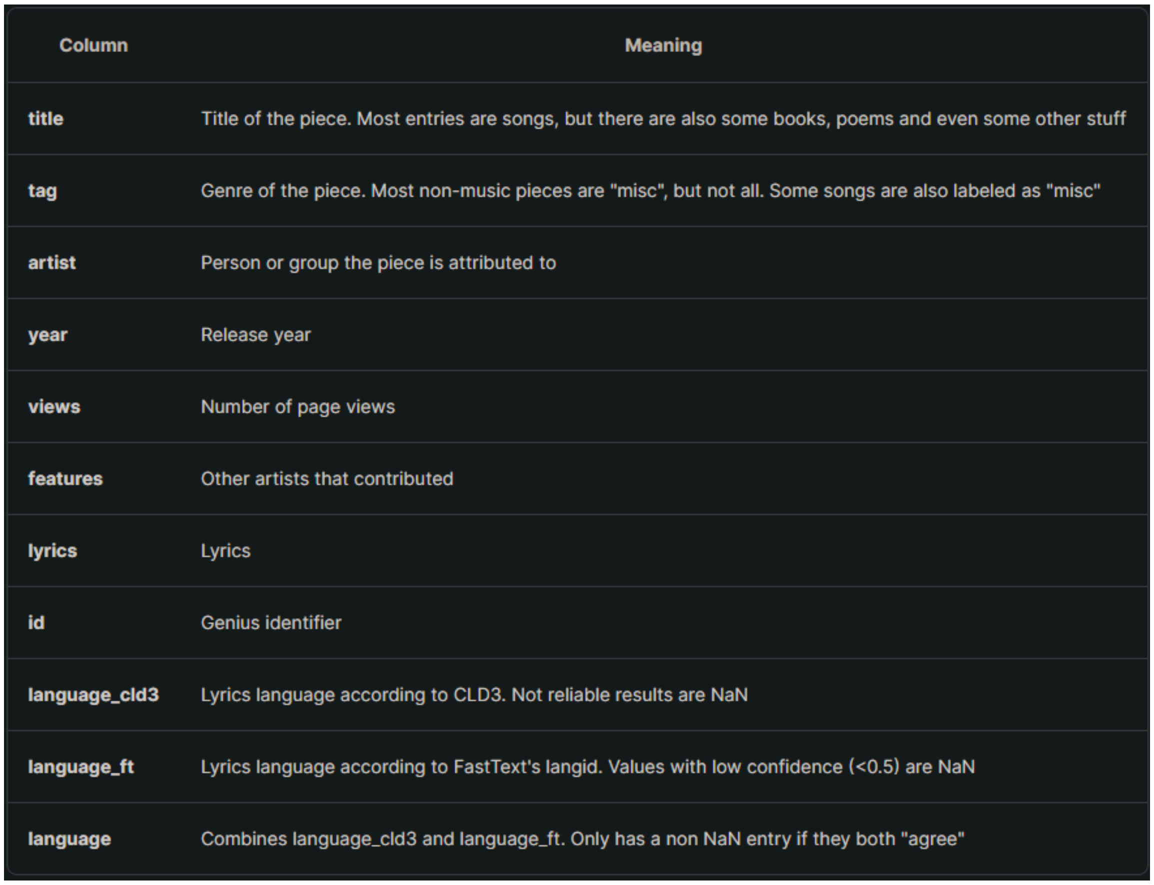
Apache Spark[2] will be used to handle the large amount of data. Next, we will filter the data. This will consist of deleting any unnecessary punctuations, new line characters, invalid entries, etc. Now that all the data is cleaned up, we will have to perform data transformations where the sentiment analysis will be computed. The sentiment analysis scores will be set up as the features for machine learning with the genre as the label (rap and pop). To produce a model that we are satisfied with, we will figure out which features set and model create the highest accuracy.

This system will help us to determine the reliability of predicting a song's genre from only the text information found in its lyrics title and artist and their corresponding sentiment scores.

Data Collection

To gather the data used in this project, we will be using datasets that have been gathered by Kaggle[1] which uses the songs posted on Genius from the years 2022 and below. This will be our main source of data and we will be using it throughout the whole project. To get this information, we had to manually download the CSV file that was posted on Kaggle which had all the information that was needed. Since we are using Google Colab[3], we posted the CSV file inside Google Drive. In this file, each record will have the columns shown in the picture below. The file will contain a total of 3,093,218 songs and will be brought down to a workable size described later. Each song will contain these values shown in Fig 1.

Next, we had to download the dictionary for our sentiment analysis. This was done by downloading the AFINN zip file on the IMM[5] website which has the dictionary inside it. This dictionary will provide us with all the words and scores that are tied to it on a scale between -5 and 5. How this information is formatted in the text document will be in the form of a piece of text and a tab between them. Then a new line will be separating the different records. The file will hold 2,476 records. We will also have to place this inside the Google Drive so we will have access to it during this project.

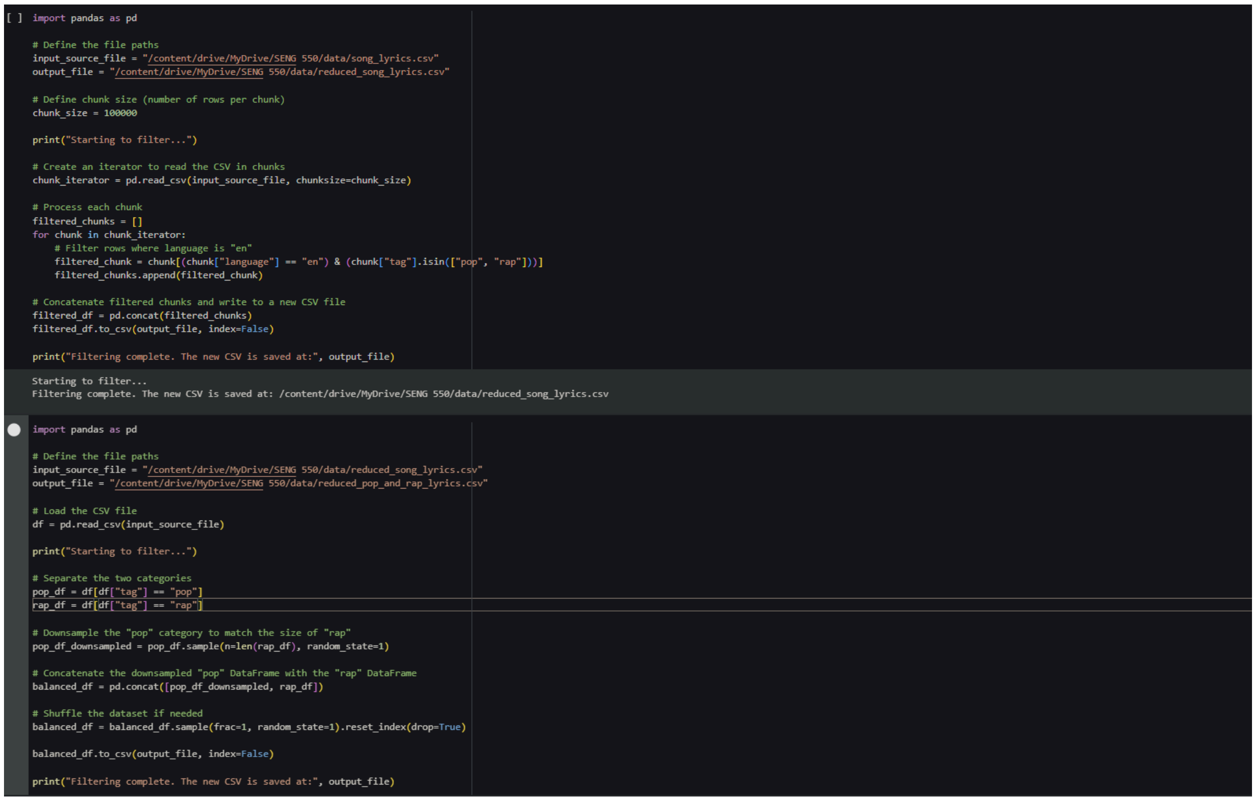


Column	Meaning
title	Title of the piece. Most entries are songs, but there are also some books, poems and even some other stuff
tag	Genre of the piece. Most non-music pieces are "misc", but not all. Some songs are also labeled as "misc"
artist	Person or group the piece is attributed to
year	Release year
views	Number of page views
features	Other artists that contributed
lyrics	Lyrics
id	Genius identifier
language_cld3	Lyrics language according to CLD3. Not reliable results are NaN
language_ft	Lyrics language according to FastText's langid. Values with low confidence (<0.5) are NaN
language	Combines language_cld3 and language_ft. Only has a non NaN entry if they both "agree"

Figure 1. A list of data columns in the song dataset

Now that we have all the information needed inside the Google Drive, we will need to mount the Google Drive onto the Google Colab so that we will have access to the downloaded files. For later use, we will need to use rap and pop songs only since we will be feeding this information to machine learning to predict whether a song is a rap or pop song using the features provided as labels. Using the code below, the output will provide us with the full CSV file that has only the rap and pop songs.

After receiving the CSV file from the code in Fig 2, we will now have access to the CSV file with only pop and rap songs. We will have to cut down the amount of data even more after this. To do this, we will randomly choose 10,000 songs which will be placed into a new CSV file with all 10,000 songs inside it. When we use a data size with more than 10,000 songs, the machine learning used in this project will crash and will not function as intended. Once we bring the data down to 10,000 songs, we will be ready to store all the information in a variable. This variable will be our RDD which will be used throughout the whole project. Using the code below, we will first define the directories for the files that we have stored within Google Drive. Once we have defined the directory, we can start to store the data in a variable for later use. After we set up the correct directory path, we will be able to set up the spark RDD(Fig 3) using the data frame that contains the data.

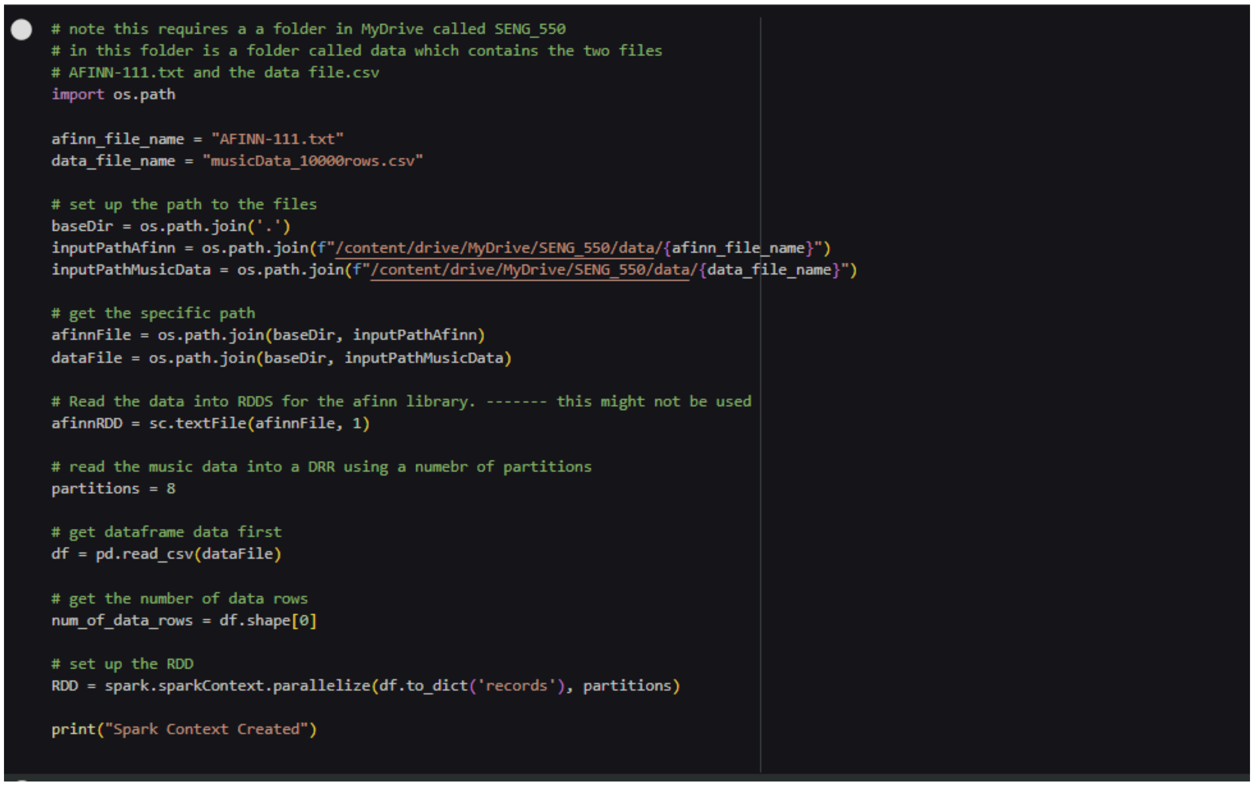


```
import pandas as pd

# Define the file paths
input_source_file = "/content/drive/MyDrive/SENG 550/data/song_lyrics.csv"
output_file = "/content/drive/MyDrive/SENG 550/data/reduced_song_lyrics.csv"

# Define chunk size (number of rows per chunk)
chunk_size = 100000

print("Starting to filter...")

# Create an iterator to read the CSV in chunks
chunk_iterator = pd.read_csv(input_source_file, chunksize=chunk_size)

# Process each chunk
filtered_chunks = []
for chunk in chunk_iterator:
    # Filter rows where language is "en"
    filtered_chunk = chunk[(chunk["language"] == "en") & (chunk["tag"].isin(["pop", "rap"]))]
    filtered_chunks.append(filtered_chunk)

# Concatenate filtered chunks and write to a new CSV file
filtered_df = pd.concat(filtered_chunks)
filtered_df.to_csv(output_file, index=False)

print("Filtering complete. The new CSV is saved at:", output_file)
```

```
Starting to filter...
Filtering complete. The new CSV is saved at: /content/drive/MyDrive/SENG 550/data/reduced_song_lyrics.csv
```

```
import pandas as pd

# Define the file paths
input_source_file = "/content/drive/MyDrive/SENG 550/data/reduced_song_lyrics.csv"
output_file = "/content/drive/MyDrive/SENG 550/data/reduced_pop_and_rap_lyrics.csv"

# Load the CSV file
df = pd.read_csv(input_source_file)

print("Starting to filter...")

# Separate the two categories
pop_df = df[df["tag"] == "pop"]
rap_df = df[df["tag"] == "rap"]

# Downsample the "pop" category to match the size of "rap"
pop_df_downsampled = pop_df.sample(n=len(rap_df), random_state=1)

# Concatenate the downsampled "pop" DataFrame with the "rap" DataFrame
balanced_df = pd.concat([pop_df_downsampled, rap_df])

# Shuffle the dataset if needed
balanced_df = balanced_df.sample(frac=1, random_state=1).reset_index(drop=True)

balanced_df.to_csv(output_file, index=False)

print("Filtering complete. The new CSV is saved at:", output_file)
```

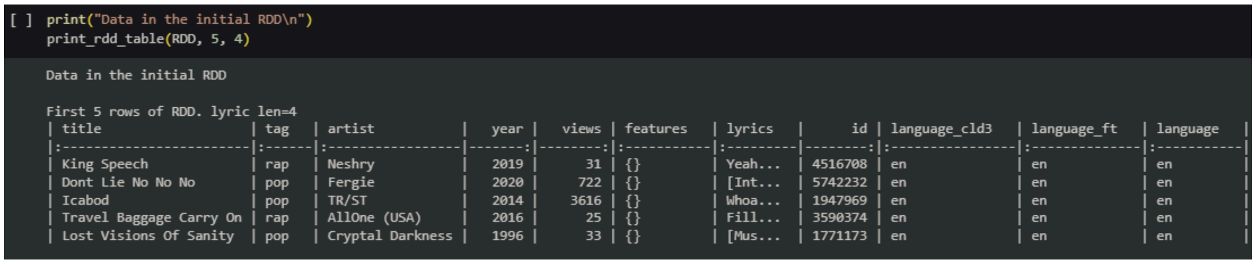
Figure 2. Code to reduce the dataset to rap and pop songs



```
# note this requires a a folder in MyDrive called SENG_550
# in this folder is a folder called data which contains the two files
# AFINN-111.txt and the data file.csv
import os.path

afinn_file_name = "AFINN-111.txt"
data_file_name = "musicData_10000rows.csv"

# set up the path to the files
baseDir = os.path.join('.')
inputPathAfinn = os.path.join(f"/content/drive/MyDrive/SENG_550/data/{afinn_file_name}")
inputPathMusicData = os.path.join(f"/content/drive/MyDrive/SENG_550/data/{data_file_name}")

# get the specific path
afinnFile = os.path.join(baseDir, inputPathAfinn)
dataFile = os.path.join(baseDir, inputPathMusicData)

# Read the data into RDDS for the afinn library. ------- this might not be used
afinnRDD = sc.textFile(afinnFile, 1)

# read the music data into a DRR using a numebr of partitions
partitions = 8

# get dataframe data first
df = pd.read_csv(dataFile)

# get the number of data rows
num_of_data_rows = df.shape[0]

# set up the RDD
RDD = spark.sparkContext.parallelize(df.to_dict('records'), partitions)

print("Spark Context Created")
```

Figure 3. Code to create a spark context and an RDD containing the song data.

Now that we have placed all the data onto a variable named RDD, we can view the data inside by using the code in Fig 4.



```
[ ] print("Data in the initial RDD\n")
    print_rdd_table(RDD, 5, 4)
```

```
Data in the initial RDD

First 5 rows of RDD. lyric len=4
| title                   | tag  | artist           |   year |    views | features   | lyrics    |       id | language_cld3   | language_ft   | language   |
|:------------------------|:-----|:-----------------|-------:|---------:|:-----------|:----------|---------:|:----------------|:--------------|:-----------|
| King Speech             | rap  | Neshry           |   2019 |       31 | {}         | Yeah...   |  4516708 | en              | en            | en         |
| Dont Lie No No No       | pop  | Fergie           |   2020 |      722 | {}         | [Int...   |  5742232 | en              | en            | en         |
| Icabod                  | pop  | TR/ST            |   2014 |     3616 | {}         | Whoa...   |  1947969 | en              | en            | en         |
| Travel Baggage Carry On | rap  | AllOne (USA)     |   2016 |       25 | {}         | Fill...   |  3590374 | en              | en            | en         |
| Lost Visions Of Sanity  | pop  | Cryptal Darkness |   1996 |       33 | {}         | [Mus...   |  1771173 | en              | en            | en         |
```

Figure 4. A small section of the data that is now in the RDD formatted as a table

Notebook Setup

The project utilized CSV files, Google Drive and a Google Colab[3] notebook. The final results are found in a Google Colab notebook.

Library Setup

A Google Drive folder was created to store a CSV containing all the song data. The folder also contained a text file of the afinn sentiment scores. These files are utilized by a Google Colab project.

The Google Colab project is initially set up with the following libraries and tools:

1) Mount the Google Drive
2) Install OpenJDK 8
3) Download Spark and Hadoop
4) Install findSpark
5) Install libraries
    a) pandas
    b) numpy
    c) nltk
    d) afinn
6) Create environment variables
    a) JAVA_HOME
    b) SPARK_HOME

Spark Setup

The data is also read into the notebook as a dataframe. Normally the CSV data would be read directly into a spark RDD, however, the lyrics section of the data contains data with many '\n' characters. For example:

"Yeah\n\nNeshry\n\nUh\n\n{Verse 1}\n\nListen\n\nOnly thing bigger than me is my ego…"

This created a problem with reading the data as the '\n' would send the data to a new line and interfere with the RDD setup.

The dataframe was read and then parallelized into the spark context. It was put into an RDD using the following code.

```
partitions = 8
df = pd.read_csv(dataFile) # dataFile represents the path to song data CSV
RDD = spark.sparkContext.parallelize(df.to_dict('records'), partitions)
```

Data Filtering

The data needs to be processed before the machine learning models can use it. This is done to convert the data into a suitable format.

Each element of the RDD is structured as a dictionary. This is for ease of use while cleaning the data. The following is an example of a single element in the RDD. Please note the lyrics have been reduced for readability. The RDD displayed is the one initially created when parallelizing the dataframe.

```
{
  'title': 'King Speech',
  'tag': 'rap',
  'artist': 'Neshry',
  'year': 2019,
  'views': 31,
  'features': '{}',
  'lyrics': "Yeah\n\nNeshry\n\nUh\n\n{Verse 1}\n\nListen\n\nOnly thing
          bigger than me is my ego…"
  'id': 5742232,
  'language_cld3': 'en',
  'language_ft': 'en',
  'language': 'en'
}
```

Remove Missing Data

Some songs in the database are missing data. While there are few of these songs they are outliers and need to be removed. The main features will be created using the following data:

1) Title
2) Artist
3) Lyrics
4) Year
5) Views

The following code creates an RDD that only contains rows that have the above columns.

```
filteredRDD = labelledRDD.filter( lambda row:
                                        'artist' in row and
                                        'title' in row and
                                        'lyrics' in row and
                                        "year" in row and
                                        "views" in row
                                       )
```

Remove Punctuation

The main goal is to create sentiment scores for text. The text needs to be cleaned before these scores are generated. This filtering includes:

1) Removing punctuation
2) Removing hyphens from hyphenated words
3) Remove new lines (‘\n’)
4) Remove verse and chorus indicators

Below is the function responsible for cleaning the text.

```
def clean_text(text):
  text = re.sub(r'\{.*?\}', '', text)# Remove text in curly braces {}
  text = re.sub(r'\[.*?\]', '', text)# Remove text in square brackets []
  text = text.replace('-', ' ')  # replace - with a space
  translator = str.maketrans("", "", string.punctuation)#createtranslator
  text = text.replace('\n', ' ') # remove new lines
  text = text.translate(translator)# Use translator to remove punctuation
  return text # return text
```

The RRD was transferred to clean the title, artist and lyrics.

```
cleanedRDD = filteredRDD.map(lambda row: {
  **row,
  'lyrics': clean_text(row['lyrics']),
  'artist': clean_text(row['artist']),
  'title': clean_text(row['title'])
})
```

Since the sentiment analysis would still work even if the text was not cleaned, trials were done on the original text. They found that the logistic regression model had close to 50% accuracy highlighting the importance of cleaning the text beforehand.

Data Transformations

The transformations were done to create numeric values that would serve as inputs to the machine learning models. Sentiment scores were set up using two variations of the calculation. This was done to provide many options for the feature inputs.

Setup Labels

The goal is to predict if a song's genre is rap or pop. Therefore, the tag/genre category needed to be converted to a numeric value using the following code (rap is 0 and pop is 1):

```
labelledRDD = RDD.map(lambda row: {**row, 'label': 0 if row['tag'] == 'rap' else 1})
```

This created a key representing the label entitled 'label'.

Create Sentiment Scores

The sentiment analysis is a way of converting the text to a numeric value. It was decided to use two methods of sentiment scores that can be used as features. These two methods were both performed on the title, artist and lyrics. This generated six scores in total.

Total Sentiment

This is a cumulative score of each word in a string of text. It is calculated by adding the sentiment for every word together. Since some sentiments are negative and some are positive, a song may have a neutral total score. Adding all the scores together meant that long songs could have a large positive or negative score. Short songs had less potential for large positive or negative scores.

The function uses the afinn library to calculate the total sentiment score

```
from afinn import Afinn
afinn = Afinn()
afinn.score(test)
```

Average Sentiment

This is the average sentiment score of a string of words. It is meant to represent the average sentiment of each word the string carries. The following function calculates it.

```
def get_avg_afinn_score(text):
  words = re.compile(r"\W+").split(text.lower())
  sentiments = list(map(lambda word: afinn_dict.get(word, 0), words))
  if sentiments:
      sentiment = float(sum(sentiments))/math.sqrt(len(sentiments))
  else:
      sentiment = 0
  return sentiment
```

Sentiment Code

The following code creates a new RDD with additional columns for sentiment score and average sentiment score. This created 6 new keys for sentiment scores based on total and average sentiment scores.

```
numericRDD = cleanedRDD.map(lambda row: {
   **row,
   'title_sentiment': afinn.score(row['title']),
   'lyrics_sentiment': afinn.score(row['lyrics']),
   'artist_sentiment': afinn.score(row['artist']),

   'title_sentiment_avg': get_avg_afinn_score(row['title']),
   'lyrics_sentiment_avg': get_avg_afinn_score(row['lyrics']),
   'artist_sentiment_avg': get_avg_afinn_score(row['artist'])
})
```

Labelled Points

The machine learning models require specific input. The RDD before this point had each element as a dictionary. They needed to be converted to labelled points before being used. The following code created a tuple in the RDD that represented the label and all the features. It should be noted that features could be changed at this point. This will be discussed further.

```
ML_RDD = numericRDD.map(lambda row:
 (
   # -----Label------
   row['label'],

   # ----Features----
   row["title_sentiment"],
   row['lyrics_sentiment'],
   row['artist_sentiment'],

   row['title_sentiment_avg'],
   row['lyrics_sentiment_avg'],
   row['artist_sentiment_avg'],

   row['year'],
   row['views'],
 )
)
```

The output of the ML_RDD is the following:

```
(0, 0.0, -10.0, 0.0, -0.5986843400892496, 2019, 31)
(1, -3.0, -106.0, 0.0, -5.416346981260518, 2020, 722)
```

This was then converted to labelled point format where the first element is a label and the second is an array of features(label, [feature 1, feature 2, …]). The code below shows the process and the contents of the RDD after the conversion:

```
labeledPointsRDD = ML_RDD.map(lambda row: LabeledPoint(row[0], row[1:]))
```

Outputs

```
(0.0,[0.0,-10.0,0.0,-0.5986843400892496,2019.0,31.0])
(1.0,[-3.0,-106.0,0.0,-5.416346981260518,2020.0,722.0])
```

Training and Testing Data

Finally, the two RDDs for training data and testing data were created. This was done in a random split:

```
training_data,testing_data=labeledPointsRDD.randomSplit([0.8,0.2],seed=42)
```

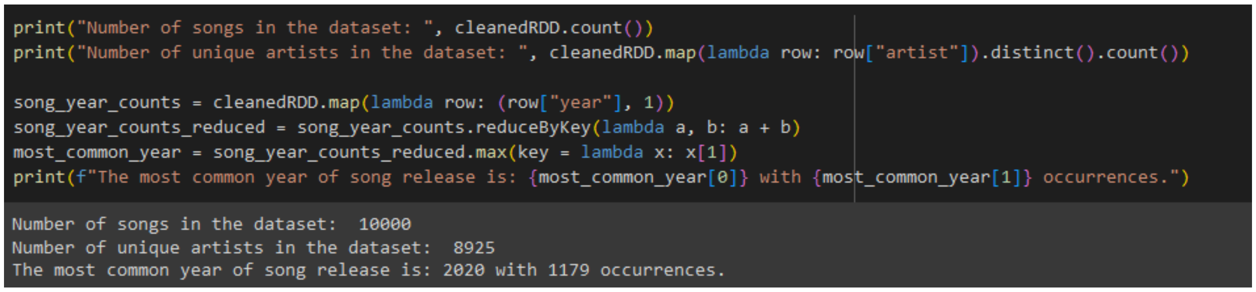
After these transformations the labelled point data is ready for the machine learning models. It contains the label and the array of features for each song.

Exploratory Data Analysis

After the data was filtered and transformed, Exploratory Data Analysis was done to provide an overview of the data before creating any machine learning models.

The first few data analysis statements are not visualizations, but simple printed text, as they are the most ideal way to display this data. This includes the number of songs in the dataset, the number of unique artists, and the most common year of song release.

The first statement (number of songs) is a sanity check, as we have intentionally filtered the dataset to 10,000 songs to ensure that the execution of the notebook does not timeout.

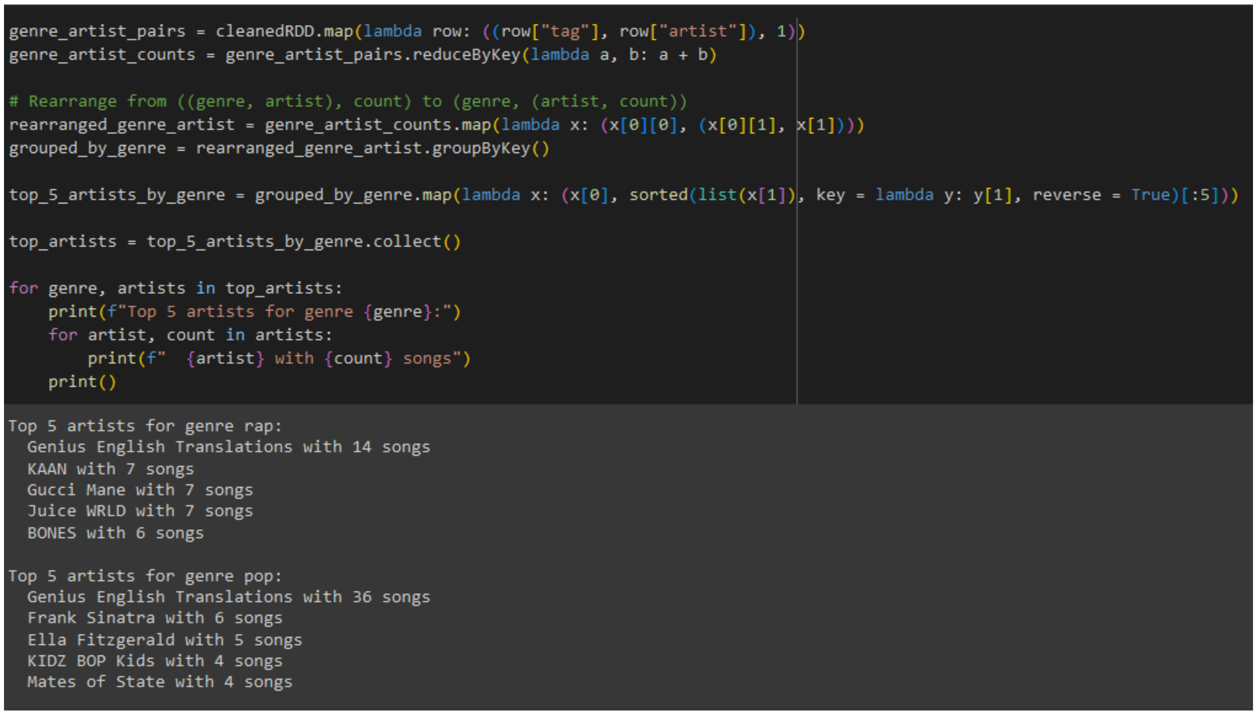


```python
print("Number of songs in the dataset: ", cleanedRDD.count())
print("Number of unique artists in the dataset: ", cleanedRDD.map(lambda row: row["artist"]).distinct().count())

song_year_counts = cleanedRDD.map(lambda row: (row["year"], 1))
song_year_counts_reduced = song_year_counts.reduceByKey(lambda a, b: a + b)
most_common_year = song_year_counts_reduced.max(key = lambda x: x[1])
print(f"The most common year of song release is: {most_common_year[0]} with {most_common_year[1]} occurrences.")
```

```
Number of songs in the dataset:  10000
Number of unique artists in the dataset:  8925
The most common year of song release is: 2020 with 1179 occurrences.
```

Figure 5. Counts for unique artists and common release year.

Then, the top 5 artists were displayed from each genre(Fig 6). It is important to note that the 10,000 songs are not indicative of the entire dataset, and as such, the song count of the top 5 artists is limited to this truncated dataset.

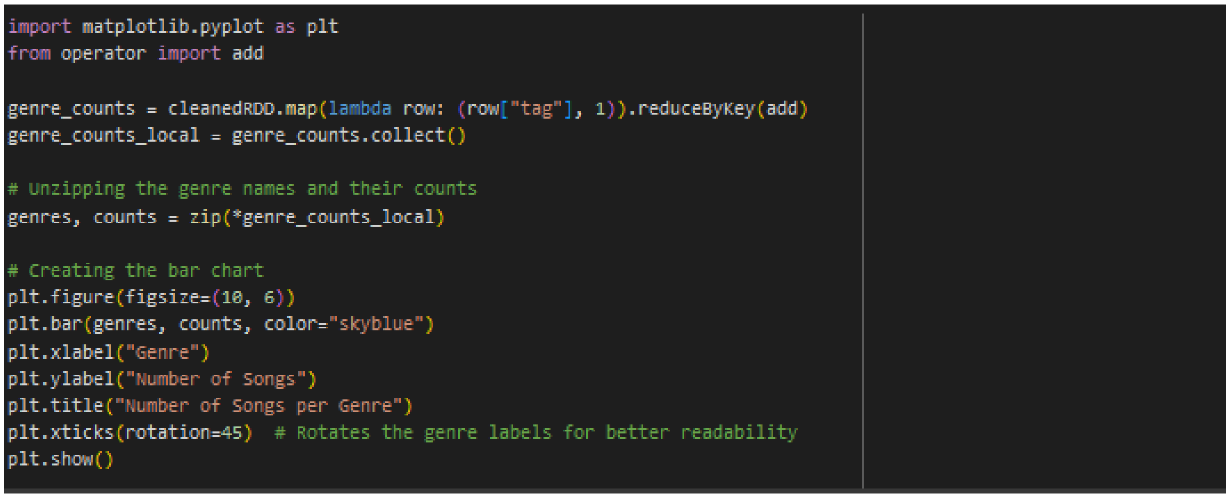


```
genre_artist_pairs = cleanedRDD.map(lambda row: ((row["tag"], row["artist"]), 1))
genre_artist_counts = genre_artist_pairs.reduceByKey(lambda a, b: a + b)

# Rearrange from ((genre, artist), count) to (genre, (artist, count))
rearranged_genre_artist = genre_artist_counts.map(lambda x: (x[0][0], (x[0][1], x[1])))
grouped_by_genre = rearranged_genre_artist.groupByKey()

top_5_artists_by_genre = grouped_by_genre.map(lambda x: (x[0], sorted(list(x[1]), key = lambda y: y[1], reverse = True)[:5]))

top_artists = top_5_artists_by_genre.collect()

for genre, artists in top_artists:
    print(f"Top 5 artists for genre {genre}:")
    for artist, count in artists:
        print(f"  {artist} with {count} songs")
    print()
```

```
Top 5 artists for genre rap:
  Genius English Translations with 14 songs
  KAAN with 7 songs
  Gucci Mane with 7 songs
  Juice WRLD with 7 songs
  BONES with 6 songs

Top 5 artists for genre pop:
  Genius English Translations with 36 songs
  Frank Sinatra with 6 songs
  Ella Fitzgerald with 5 songs
  KIDZ BOP Kids with 4 songs
  Mates of State with 4 songs
```

Figure 6. Top 5 artists for rap and pop songs

Next, a bar graph is used to visualize the number of pop versus rap songs(Fig 7.2). During the data processing, we intentionally chose an approximately equal number of pop and rap songs for this dataset to ensure that we have a balanced dataset. This visualization confirms an approximately equal number of pop and rap songs.



```
import matplotlib.pyplot as plt
from operator import add

genre_counts = cleanedRDD.map(lambda row: (row["tag"], 1)).reduceByKey(add)
genre_counts_local = genre_counts.collect()

# Unzipping the genre names and their counts
genres, counts = zip(*genre_counts_local)

# Creating the bar chart
plt.figure(figsize=(10, 6))
plt.bar(genres, counts, color="skyblue")
plt.xlabel("Genre")
plt.ylabel("Number of Songs")
plt.title("Number of Songs per Genre")
plt.xticks(rotation=45)  # Rotates the genre labels for better readability
plt.show()
```

Figure 7.1. Code to generate the number of songs for each genre

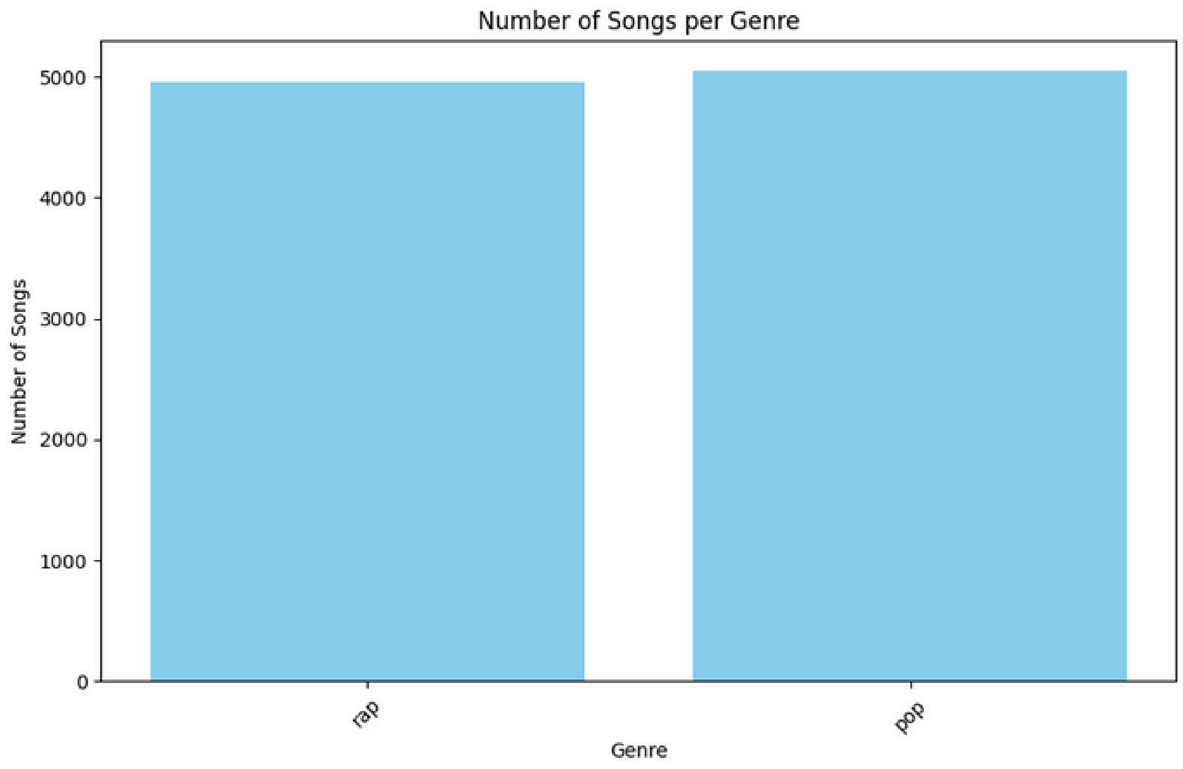
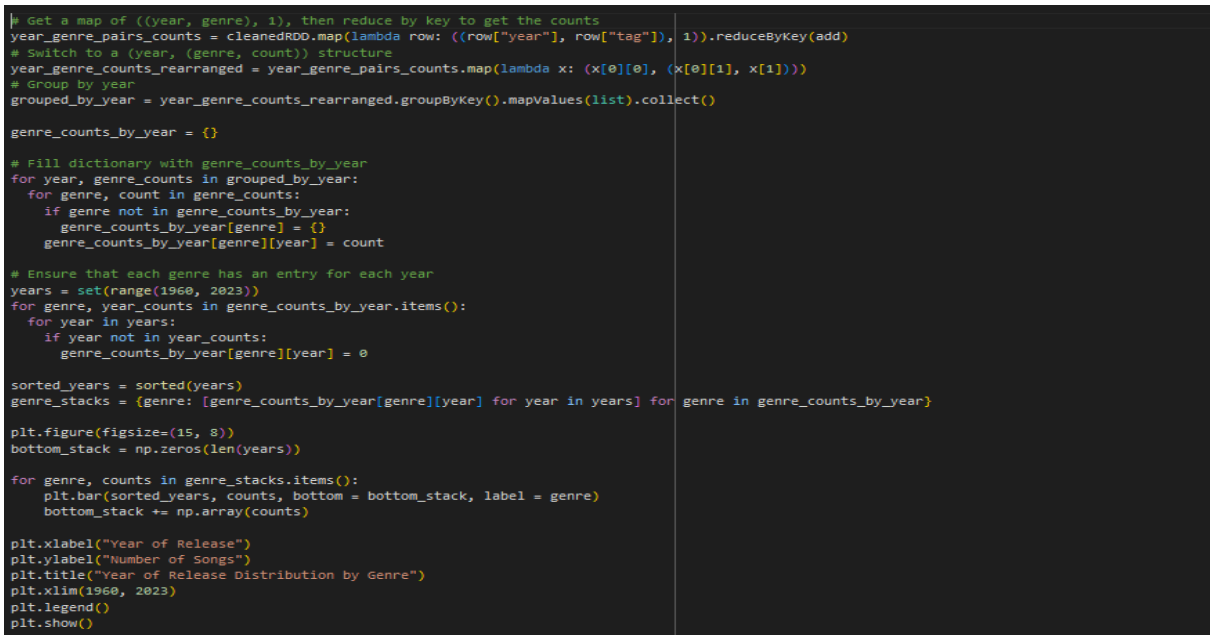


Figure 7.2. Number of songs for each genre

The following visualization depicts the year of song release by the genre(Fig 8.2). The visualization shows that both pop and rap songs skew towards the 2010 to 2020 period. As such, this minimizes the effect of one decade being more prominent than the other, which may cause the machine learning models to be biased.



```
# Get a map of ((year, genre), 1), then reduce by key to get the counts
year_genre_pairs_counts = cleanedRDD.map(lambda row: ((row["year"], row["tag"]), 1)).reduceByKey(add)
# Switch to a (year, (genre, count)) structure
year_genre_counts_rearranged = year_genre_pairs_counts.map(lambda x: (x[0][0], (x[0][1], x[1])))
# Group by year
grouped_by_year = year_genre_counts_rearranged.groupByKey().mapValues(list).collect()

genre_counts_by_year = {}

# Fill dictionary with genre_counts_by_year
for year, genre_counts in grouped_by_year:
  for genre, count in genre_counts:
    if genre not in genre_counts_by_year:
      genre_counts_by_year[genre] = {}
    genre_counts_by_year[genre][year] = count

# Ensure that each genre has an entry for each year
years = set(range(1960, 2023))
for genre, year_counts in genre_counts_by_year.items():
  for year in years:
    if year not in year_counts:
      genre_counts_by_year[genre][year] = 0

sorted_years = sorted(years)
genre_stacks = {genre: [genre_counts_by_year[genre][year] for year in years] for genre in genre_counts_by_year}

plt.figure(figsize=(15, 8))
bottom_stack = np.zeros(len(years))

for genre, counts in genre_stacks.items():
    plt.bar(sorted_years, counts, bottom = bottom_stack, label = genre)
    bottom_stack += np.array(counts)

plt.xlabel("Year of Release")
plt.ylabel("Number of Songs")
plt.title("Year of Release Distribution by Genre")
plt.xlim(1960, 2023)
plt.legend()
plt.show()
```

Figure 8.1. Code to generate the year released distribution by genre

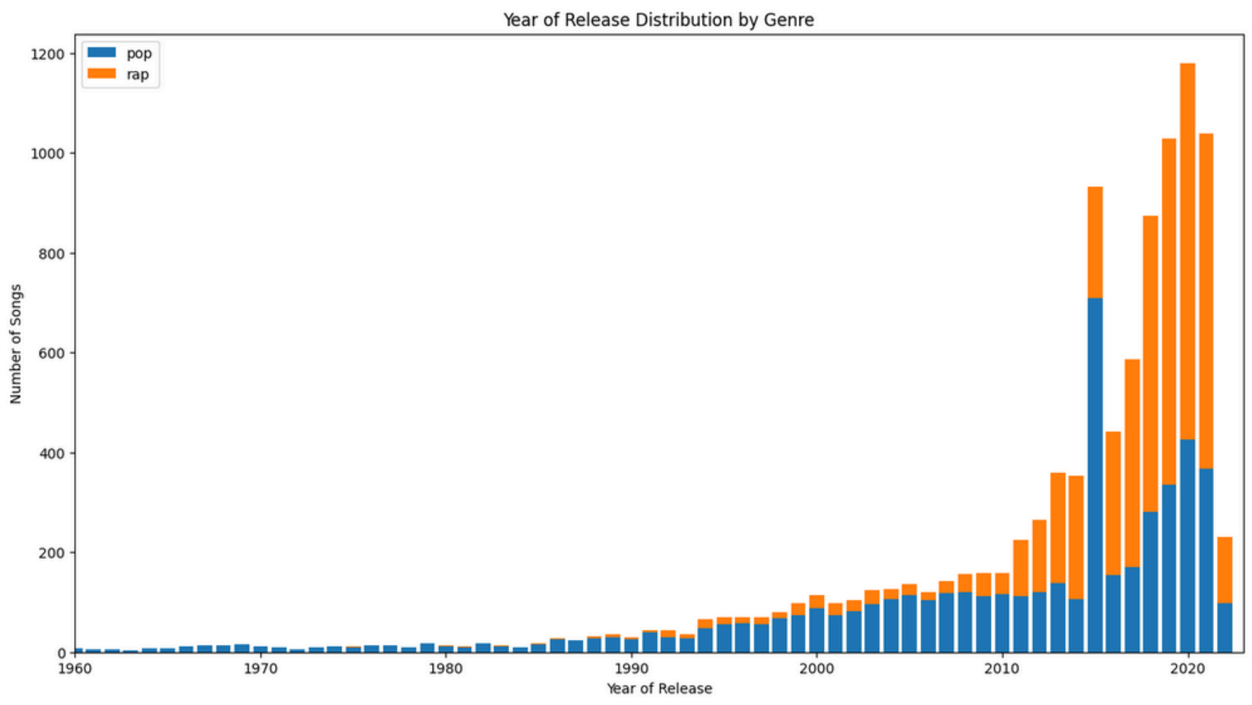
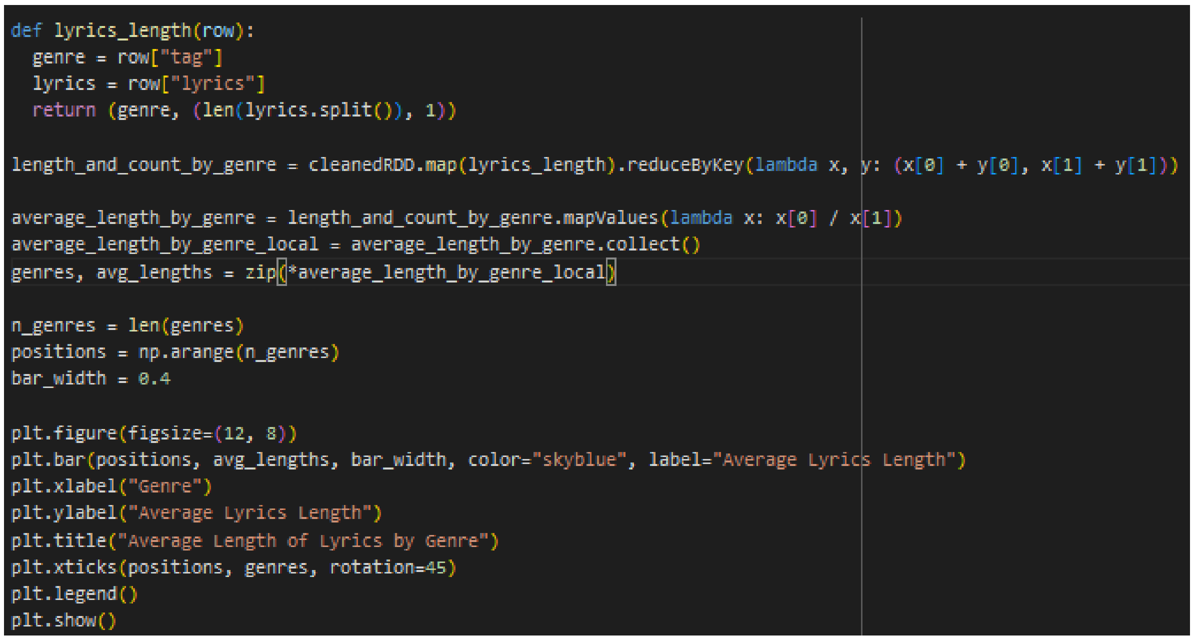


Figure 8.2. Year released distribution by genre

The fifth type of exploratory data analysis that was conducted was plotting the average length of lyrics by genre (Fig 9.2). The bar chart indicates that rap songs tend to have longer lyric lengths than pop songs. This may be attributed to the faster rate of speech within rap songs. However, we did not make an effort to ensure parity between the length of the lyrics, as this would affect the song's sentiment score. It also may have contributed to the ML being able to differentiate.



```
def lyrics_length(row):
  genre = row["tag"]
  lyrics = row["lyrics"]
  return (genre, (len(lyrics.split()), 1))

length_and_count_by_genre = cleanedRDD.map(lyrics_length).reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))

average_length_by_genre = length_and_count_by_genre.mapValues(lambda x: x[0] / x[1])
average_length_by_genre_local = average_length_by_genre.collect()
genres, avg_lengths = zip(*average_length_by_genre_local)

n_genres = len(genres)
positions = np.arange(n_genres)
bar_width = 0.4

plt.figure(figsize=(12, 8))
plt.bar(positions, avg_lengths, bar_width, color="skyblue", label="Average Lyrics Length")
plt.xlabel("Genre")
plt.ylabel("Average Lyrics Length")
plt.title("Average Length of Lyrics by Genre")
plt.xticks(positions, genres, rotation=45)
plt.legend()
plt.show()
```

Figure 9.1. Code to generate average length of lyrics by genre

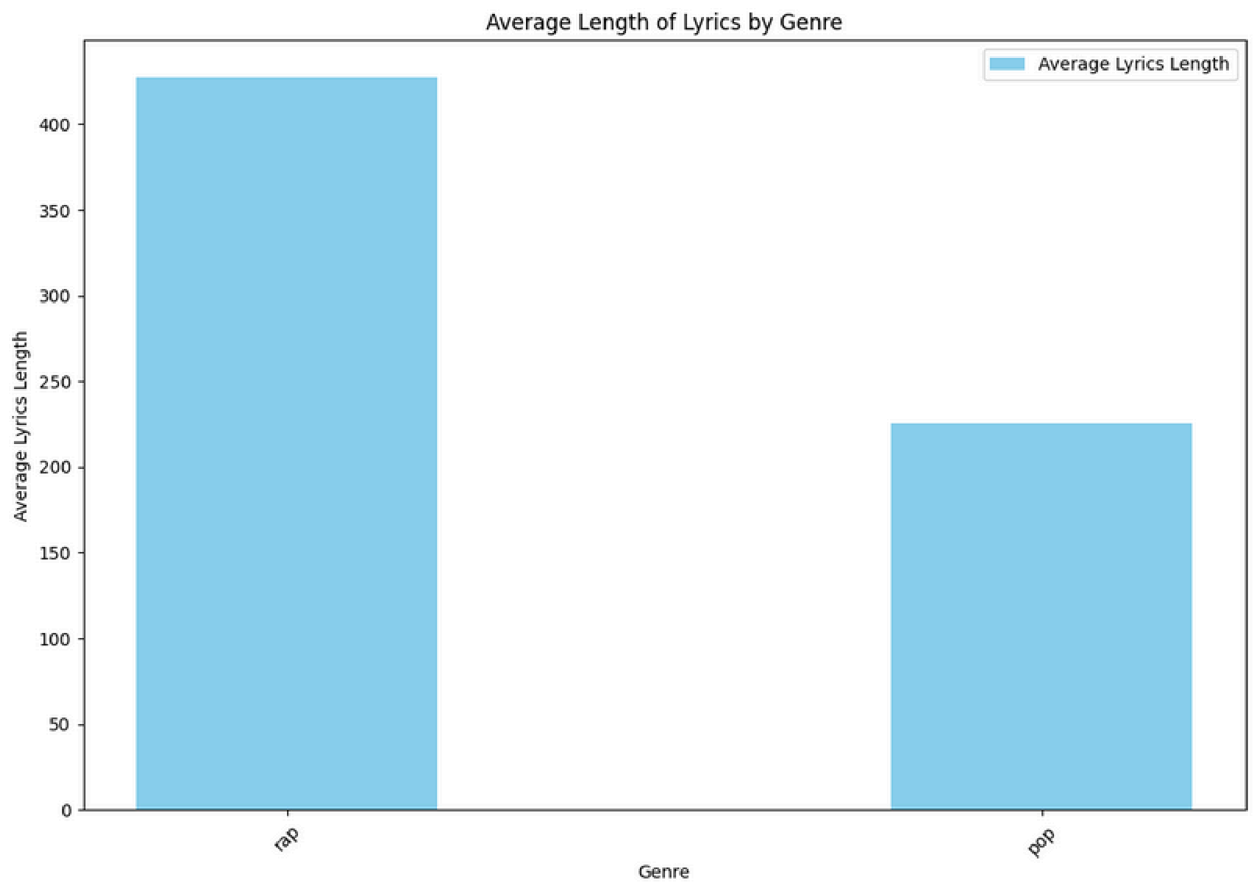
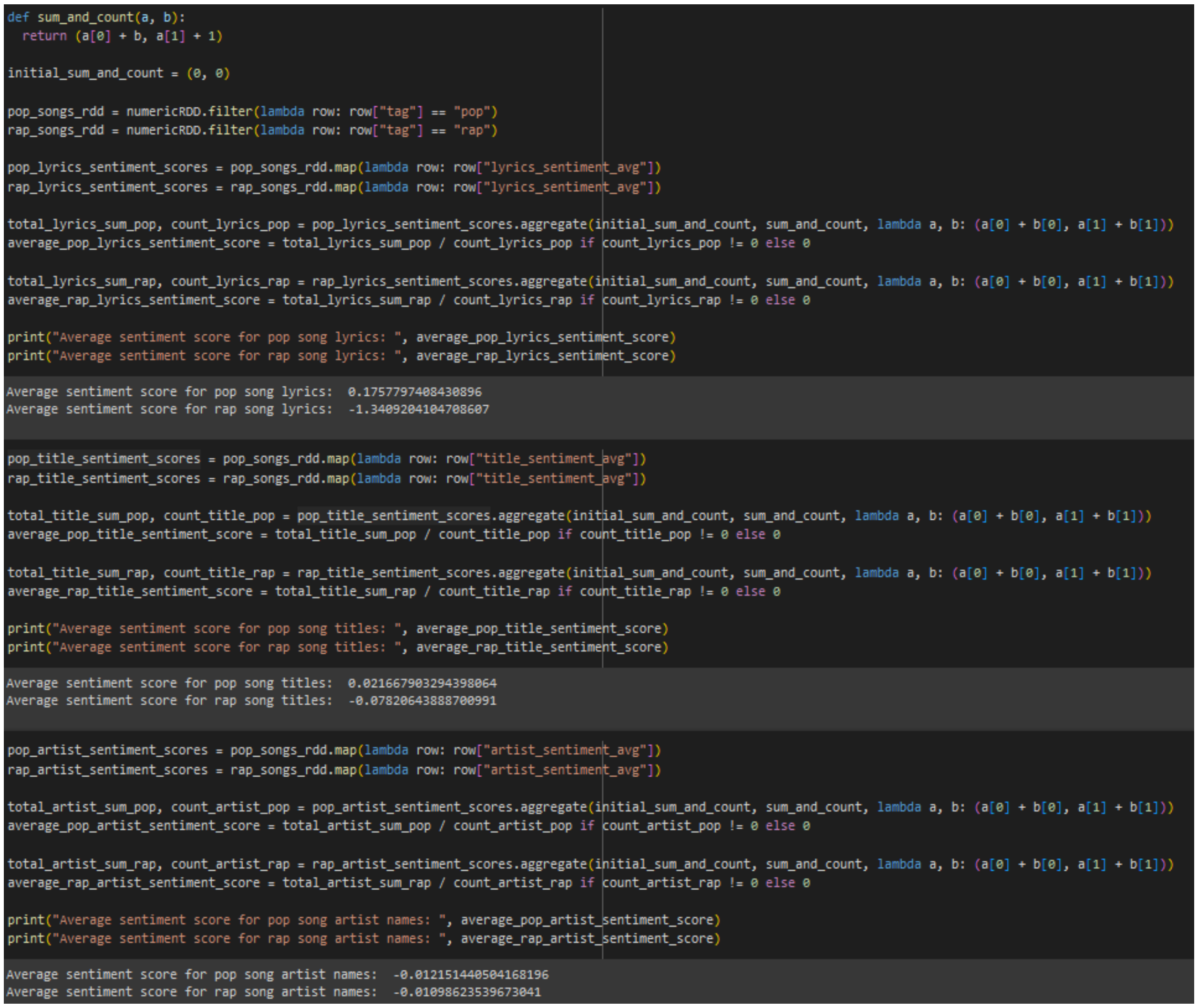


Figure 9.2. Average length of lyrics by genre

Finally, the last type of exploratory data analysis explores the average score of the sentiment analysis by genre through the AFINN score (Fig 10). Since we have the sentiment analysis scores for the lyrics, titles, and artist names, we will look at each one, separated by the genre. The average sentiment scores for the lyrics had a positive value for pop songs (~0.176) and a negative value for rap songs (~-1.341). This trend also followed for the song titles (pop: ~0.0217, rap:~ -0.0782). Interestingly, this does not hold for the artist names, as the average sentiment score for pop and rap songs were both negative (pop: ~-0.0122, rap: ~-0.0110). However, this could be attributed to the fact that many of the words in an artist's name are not found in the AFINN sentiment dictionary, so these words would have a score of 0. As such, the sentiment score is calculated based on the words in an artist's name that are in the AFINN dictionary.



```
def sum_and_count(a, b):
  return (a[0] + b, a[1] + 1)

initial_sum_and_count = (0, 0)

pop_songs_rdd = numericRDD.filter(lambda row: row["tag"] == "pop")
rap_songs_rdd = numericRDD.filter(lambda row: row["tag"] == "rap")

pop_lyrics_sentiment_scores = pop_songs_rdd.map(lambda row: row["lyrics_sentiment_avg"])
rap_lyrics_sentiment_scores = rap_songs_rdd.map(lambda row: row["lyrics_sentiment_avg"])

total_lyrics_sum_pop, count_lyrics_pop = pop_lyrics_sentiment_scores.aggregate(initial_sum_and_count, sum_and_count, lambda a, b: (a[0] + b[0], a[1] + b[1]))
average_pop_lyrics_sentiment_score = total_lyrics_sum_pop / count_lyrics_pop if count_lyrics_pop != 0 else 0

total_lyrics_sum_rap, count_lyrics_rap = rap_lyrics_sentiment_scores.aggregate(initial_sum_and_count, sum_and_count, lambda a, b: (a[0] + b[0], a[1] + b[1]))
average_rap_lyrics_sentiment_score = total_lyrics_sum_rap / count_lyrics_rap if count_lyrics_rap != 0 else 0

print("Average sentiment score for pop song lyrics: ", average_pop_lyrics_sentiment_score)
print("Average sentiment score for rap song lyrics: ", average_rap_lyrics_sentiment_score)
```

```
Average sentiment score for pop song lyrics:  0.1757797408430896
Average sentiment score for rap song lyrics:  -1.3409204104708607
```

```
pop_title_sentiment_scores = pop_songs_rdd.map(lambda row: row["title_sentiment_avg"])
rap_title_sentiment_scores = rap_songs_rdd.map(lambda row: row["title_sentiment_avg"])

total_title_sum_pop, count_title_pop = pop_title_sentiment_scores.aggregate(initial_sum_and_count, sum_and_count, lambda a, b: (a[0] + b[0], a[1] + b[1]))
average_pop_title_sentiment_score = total_title_sum_pop / count_title_pop if count_title_pop != 0 else 0

total_title_sum_rap, count_title_rap = rap_title_sentiment_scores.aggregate(initial_sum_and_count, sum_and_count, lambda a, b: (a[0] + b[0], a[1] + b[1]))
average_rap_title_sentiment_score = total_title_sum_rap / count_title_rap if count_title_rap != 0 else 0

print("Average sentiment score for pop song titles: ", average_pop_title_sentiment_score)
print("Average sentiment score for rap song titles: ", average_rap_title_sentiment_score)
```

```
Average sentiment score for pop song titles:  0.021667903294398064
Average sentiment score for rap song titles:  -0.07820643888700991
```

```
pop_artist_sentiment_scores = pop_songs_rdd.map(lambda row: row["artist_sentiment_avg"])
rap_artist_sentiment_scores = rap_songs_rdd.map(lambda row: row["artist_sentiment_avg"])

total_artist_sum_pop, count_artist_pop = pop_artist_sentiment_scores.aggregate(initial_sum_and_count, sum_and_count, lambda a, b: (a[0] + b[0], a[1] + b[1]))
average_pop_artist_sentiment_score = total_artist_sum_pop / count_artist_pop if count_artist_pop != 0 else 0

total_artist_sum_rap, count_artist_rap = rap_artist_sentiment_scores.aggregate(initial_sum_and_count, sum_and_count, lambda a, b: (a[0] + b[0], a[1] + b[1]))
average_rap_artist_sentiment_score = total_artist_sum_rap / count_artist_rap if count_artist_rap != 0 else 0

print("Average sentiment score for pop song artist names: ", average_pop_artist_sentiment_score)
print("Average sentiment score for rap song artist names: ", average_rap_artist_sentiment_score)
```

```
Average sentiment score for pop song artist names:  -0.012151440504168196
Average sentiment score for rap song artist names:  -0.01098623539673041
```

Figure 10. Average sentiment scores by genre

Modelling Building

As we are trying to predict if songs are either pop or rap, it is logical for us to use models that are focused on classification. For this project, we decided to use four different classification models to assess their accuracy:

1. Logistic Regression with the L-BFGS optimization algorithm
    - A machine learning model that can be used in both classification and predictive analytics. Logistic regression is designed to calculate the likelihood of occurrence for a specific event using a set of independent variables from a dataset. The predictive variable it predicts will range from 0 to 1 [6].
    - In addition to the logistic regression, the Limited-memory BFGS (L-BFGS) optimization algorithm was used. This algorithm approximates the objective function locally as a quadratic without evaluating the second partial derivatives of

the objective function. Using this algorithm tends to result in a faster convergence when compared to other first-order optimizations [7].

2. Decision Tree
    - Similar to the Logistic Regression model, the Decision Tree model is also used for classification and regression tasks. The name "Decision Tree" comes from its hierarchical and tree-like structure, which has a root node, branches, internal nodes, and leaf nodes. In the Decision Tree model, a "divide and conquer" strategy is used through a greedy search to find optimal split points within a tree. This process is repeated in a top-down, recursive manner until most of the records have been classified under specific class labels [8].
3. Random Forest
    - A machine learning model that builds upon the Decision Tree model by using multiple decision trees. This is done to minimize bias and overfitting through ensembling, which results in more accurate results, especially when the individual trees are uncorrelated with each other. As such, the Random Forest model aims to create an uncorrelated forest of decision trees through feature randomness and bagging (i.e., selecting a random sample of data with replacement) [9].
4. Gradient Boosted Trees
    - The Gradient Boosted Trees is another type of model that builds upon the Decision Tree model. In a Gradient Boosted Tree model, weaker prediction models (from decision trees) are used to develop better models by trying to predict the error left over from the previous model. Many weaker models are combined to create a strong learning model. Since the tree from the previous model is connected in sequence, this model starts with slower learning rates but has better accuracy rates [10].

We had also initially chosen to use the Naïve Bayes model. However, we discovered that negative values from our dataset (such as our sentiment scores) cannot be passed into the Naïve Bayes model as input. As such, we decided to not use Naïve Bayes in this project.

There are several considerations that we would like to note for the development of these models:

- As noted in the "Exploratory Data Analysis" section, rap songs tend to have a longer lyric length (in terms of total number of words) than pop songs. This may affect the models. However, we have decided not to truncate any of the songs' lyrics since this would affect the sentiment analysis process.
- Compared to the lyrics, the length of a song's title and artist name are not very long (as they are at most a few words long). As such, the sentiment from the song title and artist are not the best at determining a song's sentiment compared to the lyrics.
- Furthermore, the AFINN dictionary only contains 2,477 words [4], so there are many words in a song that are not present in the AFINN dictionary. These words are then

assigned a sentiment score of 0. As Such, we cannot truly determine a song's sentiment score without further developing the sentiment analysis dictionary. This is incredibly time-consuming and out-of-scope for this project, so using AFINN is the best approximation.

Furthermore, as a result of our data processing and transformation, we had a total of 9 potential features from the dataset that could be selected in the machine learning models:

- Title Total Sentiment: the sum of the AFINN score from a song's title
- Lyrics Total Sentiment: the sum of the AFINN score from a song's lyrics
- Artist Total Sentiment: the sum of the AFINN score from the artist's name
- Title Average Sentiment: the average of the AFINN score from a song's title
- Lyrics Average Sentiment: the average of the AFINN score from a song's lyrics
- Artist Average Sentiment: the average of the AFINN score from the artist's name
- Year Released: the year in which a song was released which provides temporal data
- Number of Views: the number of views on the website Genius.com (the source in which the song's lyrics came from and indicates popularity)

In the following code, we could select or deselect features to train the model on. Depending on the model type, not all of the features are used.

```
ML_RDD = numericRDD.map( lambda row:
  (
     # -----Label------
     row['label'],

     # ----Features----
     row["title_sentiment"],
     row['lyrics_sentiment'],
     row['artist_sentiment'],

     row['title_sentiment_avg'],
     row['lyrics_sentiment_avg'],
     row['artist_sentiment_avg'],

     row['year'],
     row['views'],
  )
)
```

Results and Discussion

Our goal in this project was to discover which model outperforms the others in terms of various metrics; as well as, which feature set supports this model. We used a balanced dataset (a similar number of pop and rap songs), meaning that accuracy is an important metric for evaluating model performance. This will compensate for the potential of bias as we cannot have high accuracy for a majority class. Accuracy was an initial metric that guided us in comparing each model to our baseline model. The following are examples of the accuracies of each model and sample data points within the current notebook, utilizing all 8 features:

Logistic Regression with LBFGS
Model Accuracy 67.11%
Testing Data Points:

- Point 0 | Real:0 | Predicted:1
- Point 1 | Real:1 | Predicted:0
- Point 2 | Real:1 | Predicted:1
- Point 3 | Real:0 | Predicted:1
- Point 4 | Real:1 | Predicted:0

Decision Tree
Model Accuracy 74.07%
Testing Data Points:

- Point 0 | Real:0 | Predicted:0.0
- Point 1 | Real:1 | Predicted:0.0
- Point 2 | Real:1 | Predicted:0.0
- Point 3 | Real:0 | Predicted:0.0
- Point 4 | Real:1 | Predicted:1.0

Random Forest
Model Accuracy: 75.15%
Testing Data Points:

- Point 0 | Real:0 | Predicted:0.0
- Point 1 | Real:1 | Predicted:0.0
- Point 2 | Real:1 | Predicted:1.0
- Point 3 | Real:0 | Predicted:0.0
- Point 4 | Real:1 | Predicted:1.0

GBT
Model Accuracy: 74.36%
Testing Data Points:

- Point 0 | Real:0 | Predicted:0.0
- Point 1 | Real:1 | Predicted:0.0
- Point 2 | Real:1 | Predicted:1.0
- Point 3 | Real:0 | Predicted:0.0
- Point 4 | Real:1 | Predicted:1.0

In summary, our baseline model, Logistic Regression with LBFGS, provides an accuracy of 67.11%; however, the other tree-based models (Decision Tree, Random Forest, and Gradient Boosted Trees) outperform it. It is worth noting that Random Forest has the highest accuracy at

75.15%. Further analysis and tuning may be required to improve the performance of this model and our baseline model.

Our main goal for this project was to find a model and feature set combination that would optimize accuracy for a binary genre classification model. In doing so, we could improve our baseline model or explore other models that may outperform it. Considering we had limited features for binary genre prediction, we tried to find the most meaningful features for any given model. Many combinations exist for the 8 features; we ran 10 trials due to time. We accumulated the results in a table where we highlighted the highest accuracy for each feature set from the test data in yellow. We also marked the highest accuracy for each model by bolding the text of the model type. We acknowledged that the random factor involved in the Random Forest model was accounted for using 2 runs to ensure similar results. Through experimentation, we found that the highest accuracy we could achieve was **75.54%** using the Random Forest model in feature *set 2*. Table 1 describes this process:

Feature Set, Model, and Accuracy Table

Set	Features	Model	Accuracy(%)	Average Accuracy (%)
1	~~Title: Total Sentiment Score~~ Lyrics: Total Sentiment Score Artist: Total Sentiment Score ~~Title: Average Word Sentiment~~ Lyrics: Average Word Sentiment Artist: Average Word Sentiment Year Released Number of Views	Logistic Regression with LBFGS	68.14	73.11
		Decision Tree	74.22	
		Random Forest 1	73.58	
		Random Forest 2	75.15	
		Gradient-Boosted Trees (GBT)	74.46	
2	Title: Total Sentiment Score Lyrics: Total Sentiment Score Artist: Total Sentiment Score Title: Average Word Sentiment Lyrics: Average Word Sentiment Artist: Average Word Sentiment Year Released Number of Views	Logistic Regression with LBFGS	67.11	73.02
		Decision Tree	74.07	
		Random Forest 1	74.02	
		Random Forest 2	**75.54**	
		Gradient-Boosted Trees (GBT)	**74.36**	
3	~~Title: Total Sentiment Score~~ Lyrics: Total Sentiment Score ~~Artist: Total Sentiment Score~~ ~~Title: Average Word Sentiment~~	Logistic Regression with LBFGS	63.04	69.16
		Decision Tree	70.88	
		Random Forest 1	70.93	

	Lyrics: Average Word Sentiment ~~Artist: Average Word Sentiment~~ ~~Year Released~~ ~~Number of Views~~	Random Forest 2	71.08	
		Gradient-Boosted Trees (GBT)	69.85	
4	~~Title: Total Sentiment Score~~ Lyrics: Total Sentiment Score ~~Artist: Total Sentiment Score~~ ~~Title: Average Word Sentiment~~ Lyrics: Average Word Sentiment ~~Artist: Average Word Sentiment~~ Year Released Number of Views	Logistic Regression with LBFGS	67.84	72.36
		Decision Tree	74.31	
		Random Forest 1	74.85	
		Random Forest 2	74.02	
		Gradient-Boosted Trees (GBT)	70.80	
5	~~Title: Total Sentiment Score~~ ~~Lyrics: Total Sentiment Score~~ ~~Artist: Total Sentiment Score~~ Title: Average Word Sentiment Lyrics: Average Word Sentiment Artist: Average Word Sentiment Year Released Number of Views	Logistic Regression with LBFGS	67.60	72.49
		Decision Tree	73.48	
		Random Forest 1	74.17	
		Random Forest 2	74.26	
		Gradient-Boosted Trees (GBT)	72.94	
6	Title: Total Sentiment Score Lyrics: Total Sentiment Score Artist: Total Sentiment Score ~~Title: Average Word Sentiment~~ ~~Lyrics: Average Word Sentiment~~ ~~Artist: Average Word Sentiment~~ Year Released Number of Views	**Logistic Regression with LBFGS**	**70.20**	**73.24**
		Decision Tree	**74.51**	
		Random Forest 1	73.14	
		Random Forest 2	74.50	
		Gradient-Boosted Trees (GBT)	73.87	
7	Title: Total Sentiment Score Lyrics: Total Sentiment Score Artist: Total Sentiment Score ~~Title: Average Word Sentiment~~ Lyrics: Average Word Sentiment ~~Artist: Average Word Sentiment~~ Year Released Number of Views	Logistic Regression with LBFGS	67.70	72.91
		Decision Tree	74.31	
		Random Forest 1	74.17	
		Random Forest 2	74.21	
		Gradient-Boosted Trees (GBT)	74.17	
8	Title: Total Sentiment Score Lyrics: Total Sentiment Score	Logistic Regression with LBFGS	67.70	**73.24**
		Decision Tree	74.31	

	Artist: Total Sentiment Score ~~Title: Average Word Sentiment~~ Lyrics: Average Word Sentiment Artist: Average Word Sentiment Year Released Number of Views	Random Forest 1	74.71	
		Random Forest 2	75.15	
		Gradient-Boosted Trees (GBT)	74.31	
9	Title: Total Sentiment Score ~~Lyrics: Total Sentiment Score~~ Artist: Total Sentiment Score Title: Average Word Sentiment ~~Lyrics: Average Word Sentiment~~ Artist: Average Word Sentiment Year Released Number of Views	Logistic Regression with LBFGS	50.93	66.34
		Decision Tree	70.34	
		Random Forest 1	70.34	
		Random Forest 2	69.65	
		Gradient-Boosted Trees (GBT)	70.44	
10	Title: Total Sentiment Score Lyrics: Total Sentiment Score ~~Artist: Total Sentiment Score~~ Title: Average Word Sentiment Lyrics: Average Word Sentiment ~~Artist: Average Word Sentiment~~ Year Released Number of Views	Logistic Regression with LBFGS	66.91	72.64
		Decision Tree	74.17	
		Random Forest 1	74.22	
		Random Forest 2	73.73	
		Gradient-Boosted Trees (GBT)	74.17	

Table 1. Table showing the accuracies of various combinations of model and feature set. The bolded model has the highest accuracy of that model. The yellow highlight has the highest accuracy for the feature set. Random Forest models 1 and 2 are the same model but since random forest will produce various accuracies the two models show the variation in accuracy for the random forest model. Given 8 features there exists a total of 255 possible combinations. Only 10 of these options are shown in the table.

As suggested in the table above, *set 6* contained our most optimized baseline model where Logistic Regression with LBFGS had an accuracy of **70.20%**. This feature set was promising as our combined models had an average accuracy of **73.24%**. This is expected considering the features that are used are split of sentiment scores (3 features) and non-sentimental scores (2 features). Considering that sentiment scores are solely emotional context, there is more weight placed on areas that could provide more predictive power (views and year released). However, *set 8* contradicts this notion as only the *Title: Average Word Sentiment* feature is removed and each set is tied as the highest ranking in terms of average model accuracy. It follows that in second place is *set 1* with an average accuracy of **73.11%.** Once again, the *Title: Average Word Sentiment* feature is removed which allows us to conclude that such a feature may not facilitate the accuracy of our baseline model.

Next, we gained insights into the models with the highest accuracy where *set 2* maximized accuracy for two of our models: Random Forest at **75.54%** and GBT at **74.36%.** This set also revealed adequate results for our other models as the average accuracy of all models was **73.02%**. However, this set used all features which negatively impacted our baseline model as it had an accuracy of **67.11%**, supporting the idea that certain features have a positive or negative impact on specific models. We decided to include this feature set in our notebook as it included our best-performing model, Random Forest. It also included our second best-performing model, GBT, where both models outperformed our baseline model in terms of accuracy. Such model types also performed well using other metrics using these features as described below.

Area Under the Receiver Operating Characteristic curve (AUC-ROC) is a common metric to evaluate binary classification models. We can compare the models to a random classifier and perfect classifier to see how our classifier models use this feature set. For further analysis, we computed AUC-ROC on the testing data (Fig 11) to see how well the model differentiates between the two classes (pop and rap):

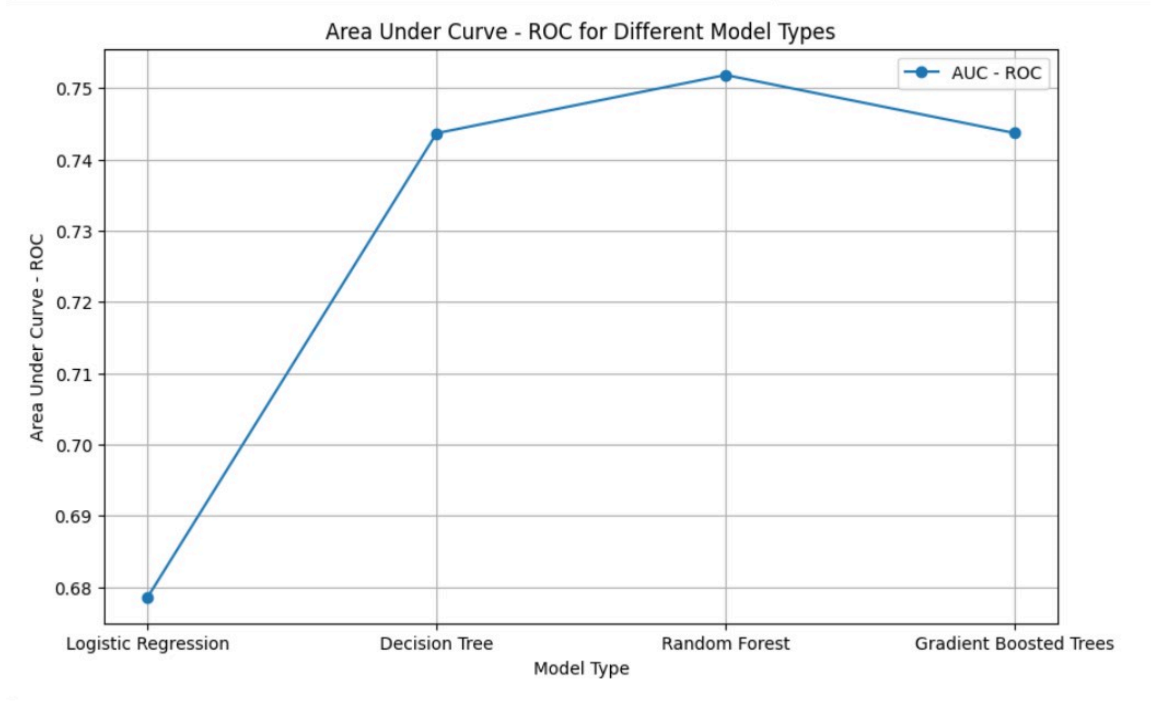


Figure 11. Area Under Curve - ROC Model Comparison

Considering this is on the testing data, we can infer that the Random Forest model is the most optimized for discriminating between rap and pop songs on unseen data. Such a metric allows us to gain insight into the various classification thresholds to elicit the impact of False Negatives (FN) and False Positives (FP). Considering a random classifier has an AUC value of 0.5 and a perfect classifier has an AUC value of 1.0, we are satisfied to see Random Forest have an AUC value of approximately **0.752**. Once again, it proves it is our most generalizable and best-performing model. Next, we notice that GBT is the second-ranking model again with an AUC value of approximately **0.743**. Such results allow us to infer that these models are not simply guessing a positive or negative class because they are halfway in between a random and perfect classifier AUC value. Unfortunately, our baseline model is the lowest-ranking model with an AUC value of **0.678**.

We also evaluated the models using recall, precision, and F1 scores which can be viewed within the notebook. The results of the metrics display consistent performance for Random Forest and GBT. Such trends show that these two models are our top-performing models for most metrics, excluding precision. Precision is the only metric where the Decision Tree model was the highest-ranking model and only had slightly more precision than Random Forest and GBT.

Conclusion

Through an iterative process that involved trial and error, we wanted to achieve the following desired outcomes:

1. Improve our logistic regression baseline model
2. Discover other models that may outperform our baseline model
3. Determine which set of features supports our highest-performing model

Various factors impacted our results such as the dependence on sentiment scores, data richness, a limited amount of data, and a lack of model variety. First, we want to acknowledge that using AFINN does have its limitations. Such a dataset only contains 2,477 words [4] and is one of the simpler sentiment analysis methods. It follows that computing the average for lyrics that have a larger word count may vary greater than the average for lyrics with a lesser word count. It is also worth noting that many neutral words (words with a sentiment score of 0), could also be included, which increases the denominator (number of words), influencing the averages. Such scores can cause too much variability to depend on and we require more diverse features with richness.

We could compensate for the effect that sentiment score has on our results by gathering more relevant data and putting less weight on such scores. We are confident that our metrics could have been improved by gathering rich data such as the following:

1. Audio/Musical features (tempo, rhythm, timbre, pitch, beat structure, etc.)
2. Linguistic features (rhyme density, metaphors, similes, etc.)
3. Genre-specific features (common genre words, cultural references, etc.)

Another thing that impacted our results is the use of similar tree-based models (i.e., Decision Tree, Random Forest, and Gradient Boosted Trees) which may not provide enough model variety. Such models may have similar mechanisms/algorithms that influence the evaluations. Each tree-based model had similar metrics when compared to our baseline model, the sole model that is not tree-based.

Through our experimentation and considering such factors, we still achieved all of the desired outcomes. We tried different features that improved the accuracy of our baseline model, maximizing its accuracy at **70.20%**. We also found other models that outperformed our baseline model in most metrics using feature *set 2*. It follows that Random Forest is the model that we would choose to tune and improve upon for a binary genre classification model as it had the best performance when compared against other models in most metrics. We would also consider GBT as this model is a close second in ranking for most metrics, excluding precision.

References

[1] Genius Song Lyrics with Language Information, "Kaggle”. Available: (https://www.kaggle.com/datasets/carlosgdcj/genius-song-lyrics-with-language-information/.

[2] Apache Software Foundation, "Apache Spark". Available: https://spark.apache.org/.

[3] Google, "Google Colab". Available: https://colab.research.google.com/.

[4] Finn Årup Nielsen, "AFINN: A New Word List for Sentiment Analysis," Technical University of Denmark,. Available: https://www2.imm.dtu.dk/pubdb/pubs/6010-full.html.

[5] AFINN Python Library, “PyPI”. Available: https://pypi.org/project/afinn/.

[6] “What is logistic regression?,” IBM, https://www.ibm.com/topics/logistic-regression (accessed Dec. 15, 2023).

[7] “Advanced topics,” Advanced topics - Spark 2.3.0 Documentation, https://spark.apache.org/docs/2.3.0/ml-advanced.html (accessed Dec. 15, 2023).

[8] “What is a decision tree,” IBM, https://www.ibm.com/topics/decision-trees (accessed Dec. 15, 2023).

[9] “What is Random Forest?,” IBM, https://www.ibm.com/topics/random-forest (accessed Dec. 15, 2023).

[10] Gaurav, “An introduction to gradient boosting decision trees,” Machine Learning Plus, https://www.machinelearningplus.com/machine-learning/an-introduction-to-gradient-boosting-decision-trees/ (accessed Dec. 15, 2023).