Leslie Aucapina

CIS 4130 Big Data Technologies

leslie.aucapina@baruchmail.cuny.edu

Milestone 1: Proposal

The dataset I plan to use for my semester project is from Kaggle and the is Genius Song Lyrics. The URL is
https://www.kaggle.com/datasets/carlosgdcj/genius-song-lyrics-with-language-information

The dataset includes 11 columns with attributes such as title, genre, artist, year, views, features, lyrics, ID, and language. My project's objective is to predict a song's genre based on its lyrics. The columns that will be mostly used are the genre and lyrics as well as artist, and year. Due to my variables of genre and lyrics being qualitative, I plan to accomplish this goal by encoding my categorical data of genre to numbers depending on the types of genres already listed. I plan to use a supervised learning technique to predict the genre such as logistic regression. It would be beneficial to search for commonalities between songs lyrically and compare their genres therefore listeners can have a better understanding of their music taste and the data scientists can anticipate the audience or which genre listeners would appeal to certain songs. This impacts music streaming companies by appealing to consumers' personal music tastes, creating custom playlists, and resulting in consumers returning to the streaming service.

Milestone 2: Data Acquisition

The dataset used for this project is found on Kaggle at https://www.kaggle.com/datasets/carlosgdcj/genius-song-lyrics-with-language-information. After creating an account on Kaggle and enabling the API token, a kaggle directory is made which the downloaded kaggle.json is moved into. Then, a Python environment is created, activated, and changed to be the directory. The Kaggle command line interface is also installed. (Appendix A) To install the Kaggle dataset into the virtual machine, the API command is found on the Kaggle URL site, copied, and inputted in the command line, it is written as follows:

```
kaggle datasets download -d
carlosgdcj/genius-song-lyrics-with-language-information
```

Once the file is downloaded the genius song lyrics with the language information file must be unzipped by installing a Python tool with the command:

```
sudo apt install zip
```

It is then unzipped with the command:

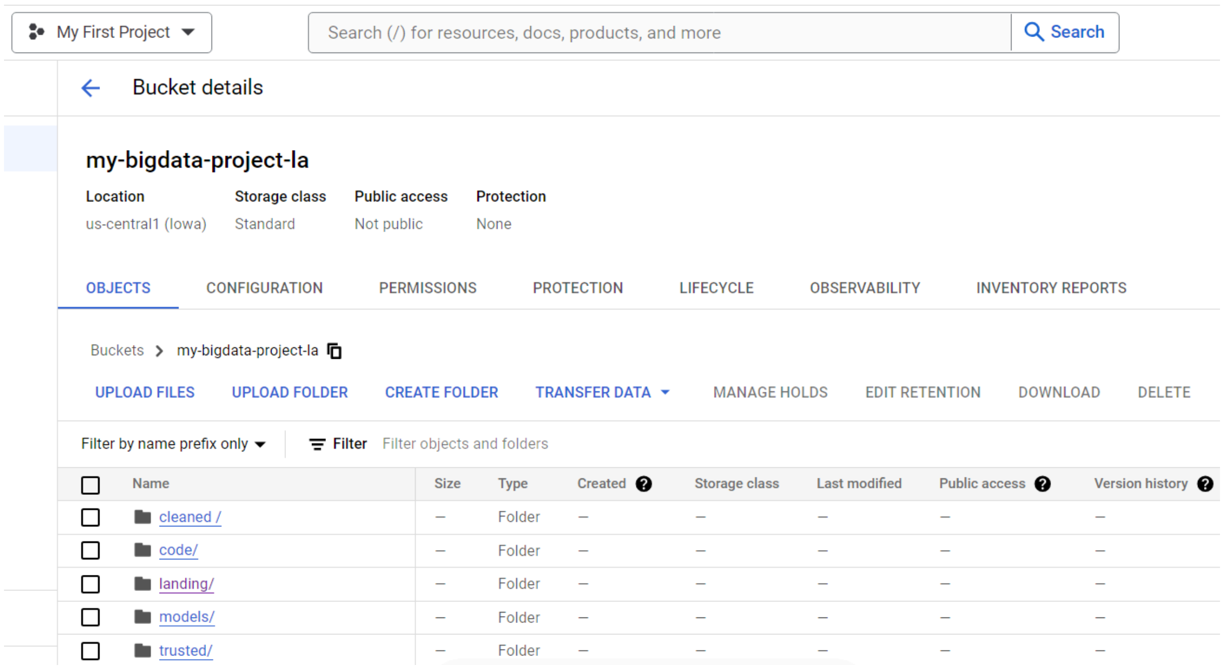
```
unzip genius-song-lyrics-with-language-information.zip
```

After unzipping the data, I created a bucket titled my-bigdata-project-la on Google Storage (Appendix A) and copied the data files to the /landing folder in Google Cloud Storage. The additional folders labeled cleaned, code, models, and trusted were made on the Cloud Console.

```
gcloud storage cp song_lyrics.csv
gs://my-bigdata-project-la/landing/
```

Results for the data in a bucket in Google Cloud Storage

On the Cloud Storage

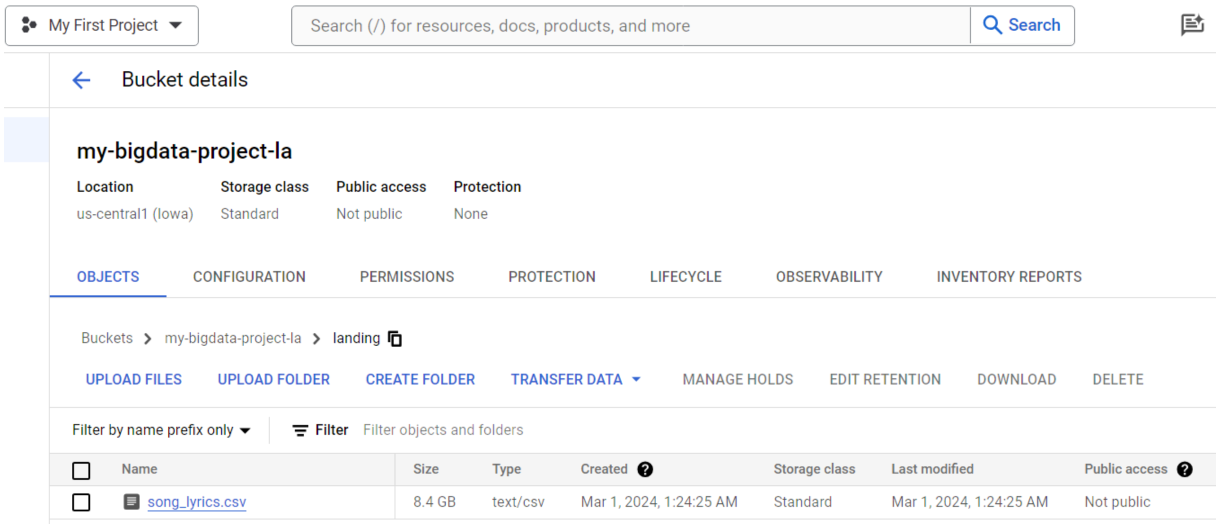


my-bigdata-project-la

Location	Storage class	Public access	Protection
us-central1 (Iowa)	Standard	Not public	None

Name	Size	Type	Created	Storage class	Last modified	Public access	Version history
cleaned /	—	Folder	—	—	—	—	—
code/	—	Folder	—	—	—	—	—
landing/	—	Folder	—	—	—	—	—
models/	—	Folder	—	—	—	—	—
trusted/	—	Folder	—	—	—	—	—

(Figure A)



my-bigdata-project-la

Location	Storage class	Public access	Protection
us-central1 (Iowa)	Standard	Not public	None

Name	Size	Type	Created	Storage class	Last modified	Public access
song_lyrics.csv	8.4 GB	text/csv	Mar 1, 2024, 1:24:25 AM	Standard	Mar 1, 2024, 1:24:25 AM	Not public

(Figure B)

Command:

```
gcloud storage ls -l gs://my-bigdata-project-la/landing
```

Output:

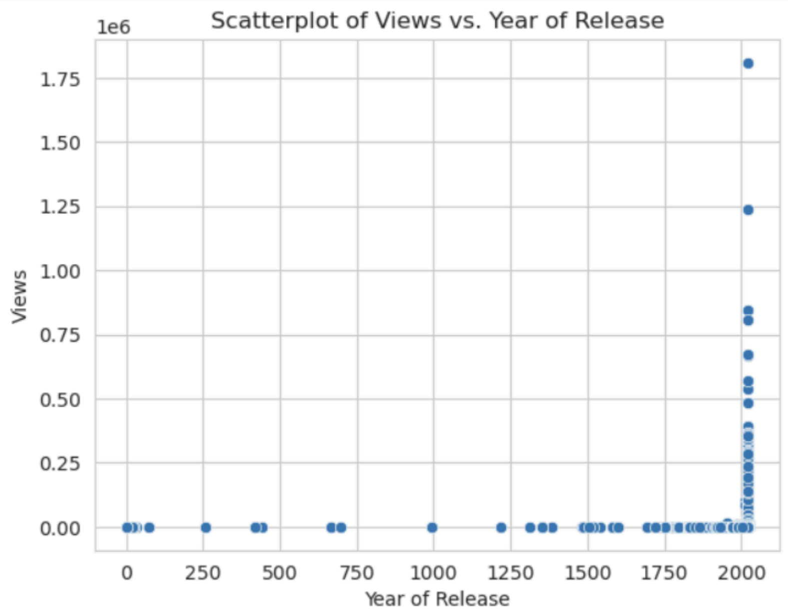
```
9070394868  2024-03-01T06:24:25Z

gs://my-bigdata-project-la/landing/song_lyrics.csv

TOTAL: 1 objects, 9070394868 bytes (8.45GiB)
```

Milestone 3: Exploratory Data Analysis and Data Cleaning

Exploratory Data Analysis

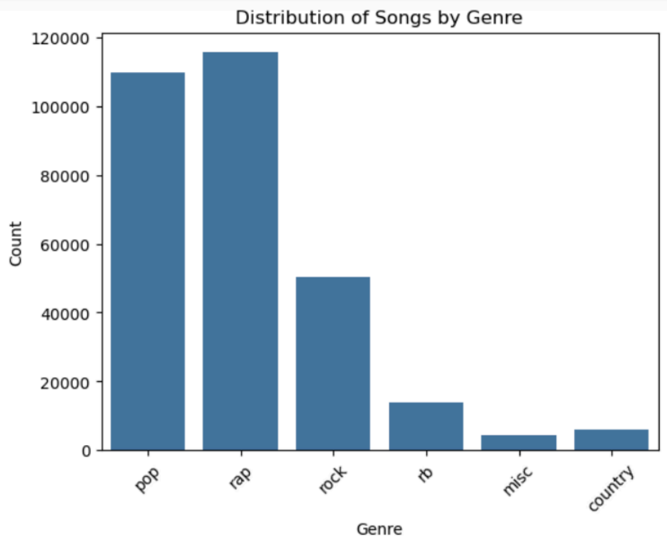
The exploratory data analysis was executed with Spark to count the number of observations which was 233,187,720. Of these records, there were various columns with null values such as 4,309 in the title column, 167,040,873 in “tag” or later to be named “genre.” There were 21,897,485 null values in “year”, 209,264,309 in “artist”, 221,462,443 null values in views, and 225868101 in features. Furthermore, there were duplicates of records with a total of 26,326,207.

When using Spark to produce descriptive statistics of the data it showed non-English characters and some “NaN” which would later be cleaned in Spark. Therefore a Dataproc cluster was created and Python module was used to conduct further exploratory data analysis on the raw data to read in batches of 300,000. Each batch conducted EDA and produced the average, min, max, and standard deviation. The columns with integer data types were year, views, and id. The minimum for “year” was 1 and the maximum was 2023. The views had a minimum of 0 and a maximum of 23,351,415.00 with the mean of views for batches of datasets ranging from 700 to 5,000. This large range for “year” and “views” shows the column needs to be cleaned. This discrepancy is best viewed in the scattered plot when there is no change from zero until the 1930’s. (Figure C).



(Figure C)

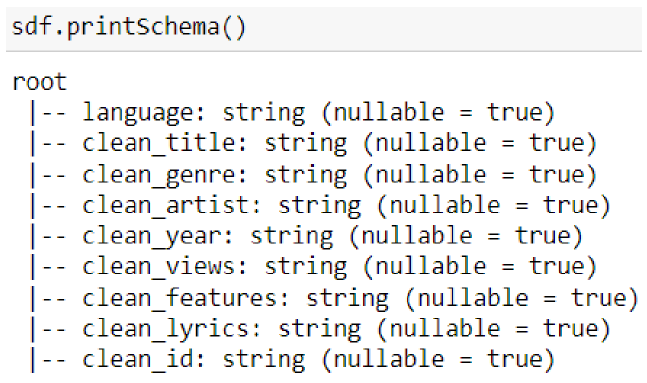
The distribution of songs is best illustrated with the counterplot (Figure D) used to visualize the frequency of categorical variables, such as genres. It showed that the genres most seen in the dataset are rap, ranking highest, and pop music in second. The other categorical genres are rock, RB, misc, and country.



(Figure D)

Data Cleaning

Spark Dataframe was used to read and clean the data “song_lyrics.csv”. The schema was first observed to view the fields and structure type and then a summary of the statistics for fields year, id, view, genre, and artists revealed non-English characters therefore, a function to strip out any non-ASCII characters was created, and turned into a user-defined function. This resulted in new fields containing only ASCII characters and the original columns were dropped in addition to language_cld3, language_ft. This left the following spark data frame schema:



```
sdf.printSchema()
```

```
root
 |-- language: string (nullable = true)
 |-- clean_title: string (nullable = true)
 |-- clean_genre: string (nullable = true)
 |-- clean_artist: string (nullable = true)
 |-- clean_year: string (nullable = true)
 |-- clean_views: string (nullable = true)
 |-- clean_features: string (nullable = true)
 |-- clean_lyrics: string (nullable = true)
 |-- clean_id: string (nullable = true)
```

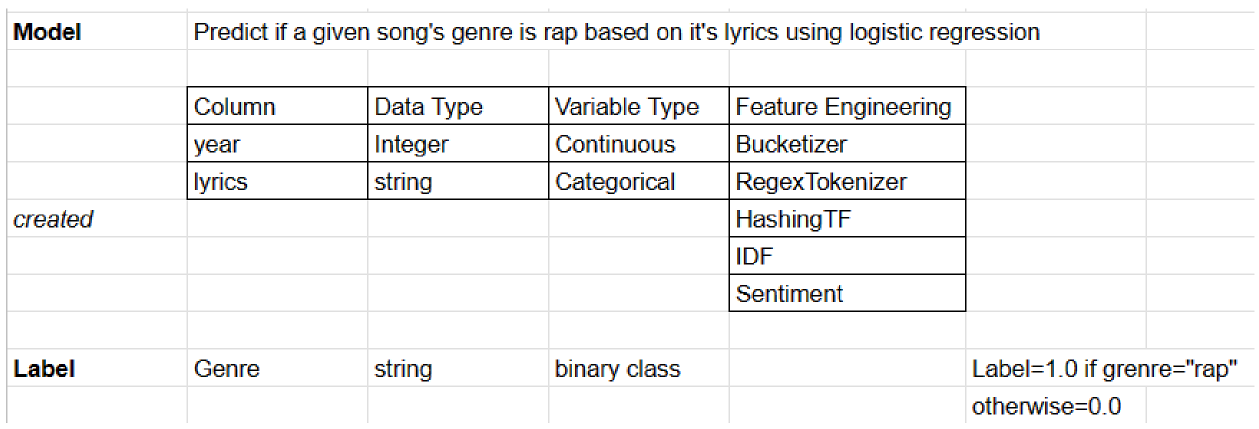
(Figure E)

The schema was edited to rename columns so as not to include “clean_”. StructFields for id, year, and views were changed to IntegerType(). Records were dropped based on their genre and lyrics being null in addition to non-numerical IDs. The data frame was reordered to show the id column first and renamed all the record’s id to be unique. Duplicate records were found and dropped. Finally, the spark data frame was converted into a parquet file and stored in the “clean” bucket on Google Storage.

Challenges that can be foreseen for feature engineering are sorting the record’s lyrics as some may be longer than others and a few are in different languages. The language column is kept in preparation for sorting based on English when applicable and columns with null lyrics were dropped from the data frame as they would not be capable of contributing to the ML model for predicting the genre based on lyrics. Having many words for the lyrics will be challenging to utilize thus specific words will have to be extracted and compared among records to predict the genre. The years have a mixed range but could be useful for observing when certain genres were more published.

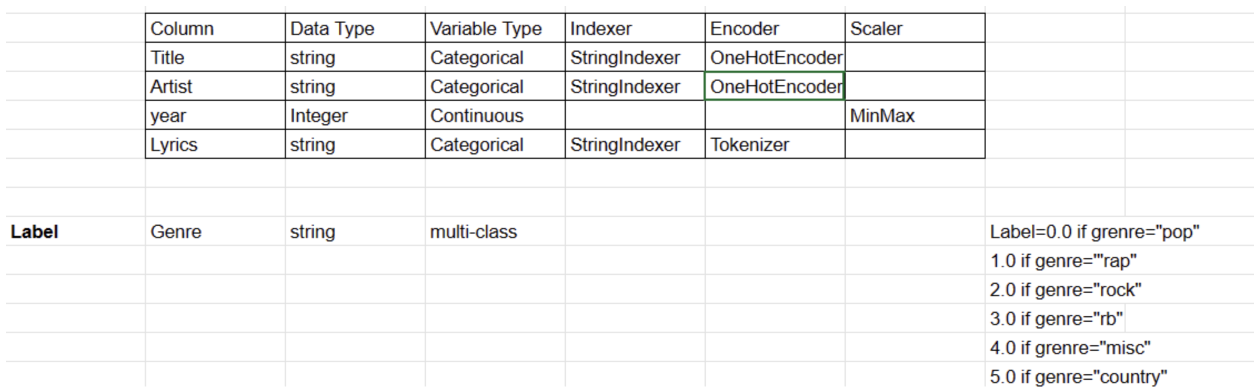
Milestone 4: Feature Engineering and Modeling

Feature Engineering



Model	Predict if a given song's genre is rap based on it's lyrics using logistic regression				
	Column	Data Type	Variable Type	Feature Engineering	
	year	Integer	Continuous	Bucketizer	
	lyrics	string	Categorical	RegexTokenizer	
created				HashingTF	
				IDF	
				Sentiment	
Label	Genre	string	binary class		Label=1.0 if grenre="rap"
					otherwise=0.0

(Figure F)



	Column	Data Type	Variable Type	Indexer	Encoder	Scaler	
	Title	string	Categorical	StringIndexer	OneHotEncoder		
	Artist	string	Categorical	StringIndexer	OneHotEncoder		
	year	Integer	Continuous			MinMax	
	Lyrics	string	Categorical	StringIndexer	Tokenizer		
Label	Genre	string	multi-class				Label=0.0 if grenre="pop"
							1.0 if genre="'rap"
							2.0 if genre="rock"
							3.0 if genre="rb"
							4.0 if grenre="misc"
							5.0 if genre="country"

(Figure G)

References of Libraries

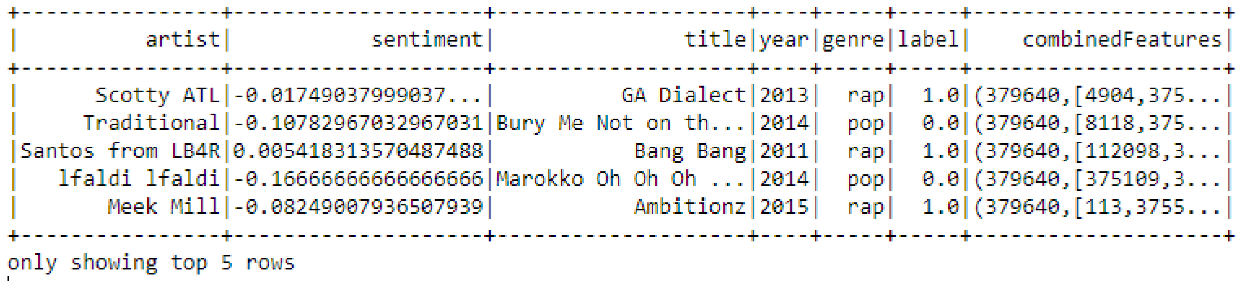
- `!pip install textblob`
- `from textblob import TextBlob`
- `from pyspark.ml import Pipeline`
- `from pyspark.ml.feature import OneHotEncoder`
- `import matplotlib.pyplot as plt`
- `import seaborn as sns`
- `from pyspark.sql.functions import col, isnan, when, count, udf, to_date, year, month, date_format, size, split`
- `from pyspark.ml.stat import Correlation`
- `from pyspark.ml.feature import VectorAssembler`
- `from pyspark.ml.feature import StringIndexer`

- `from pyspark.ml.feature import Tokenizer, RegexTokenizer`
- `from pyspark.ml.feature import HashingTF, IDF`
- `from pyspark.sql.functions import monotonically_increasing_id`

The feature engineering was originally established in the table presented above (Figure G) with the original plan to use “Title, artist, year, and lyrics” to predict multi-class label genre. However, after indexing, encoding, and performing the transformation it was learned that the artist and title were too specific and unique to the song therefore would not be helpful for the prediction. The label was changed to predict one genre to improve the model and expand in the future.

To perform sentiment analysis, the module TextBlob was used and was a major challenge for performing the feature engineering. It would result in errors with the message `“Module: ‘Textblob’ not found”` and therefore a separate Python env was created in the virtual machine instance to install textblob packages within a dependencies.zip. A SparkContext was then created and linked. (Appendix D) The variable “Year” was bucketized to reflect the decades as a feature. The lyrics were engineered using RegexTokenizer, HashingTF, IDF, and Sentiment Analysis. (Figure F) Of this engineering, the variables “idf_features” and “sentiment” were created and assembled with “yearBucket." A pipeline was created and transformed with the genre being the label and a binary class of 1.0 for the rap genre and otherwise 0.0

The transformed features are as shown:



```
+----------------+--------------------+--------------------+----+-----+-----+--------------------+
|          artist|           sentiment|               title|year|genre|label|    combinedFeatures|
+----------------+--------------------+--------------------+----+-----+-----+--------------------+
|      Scotty ATL|-0.01749037999037...|          GA Dialect|2013|  rap|  1.0|(379640,[4904,375...|
|     Traditional|-0.10782967032967031|Bury Me Not on th...|2014|  pop|  0.0|(379640,[8118,375...|
|Santos from LB4R|0.005418313570487488|           Bang Bang|2011|  rap|  1.0|(379640,[112098,3...|
|   lfaldi lfaldi|-0.16666666666666666|Marokko Oh Oh Oh ...|2014|  pop|  0.0|(379640,[375109,3...|
|       Meek Mill|-0.08249007936507939|           Ambitionz|2015|  rap|  1.0|(379640,[113,3755...|
+----------------+--------------------+--------------------+----+-----+-----+--------------------+
only showing top 5 rows
```

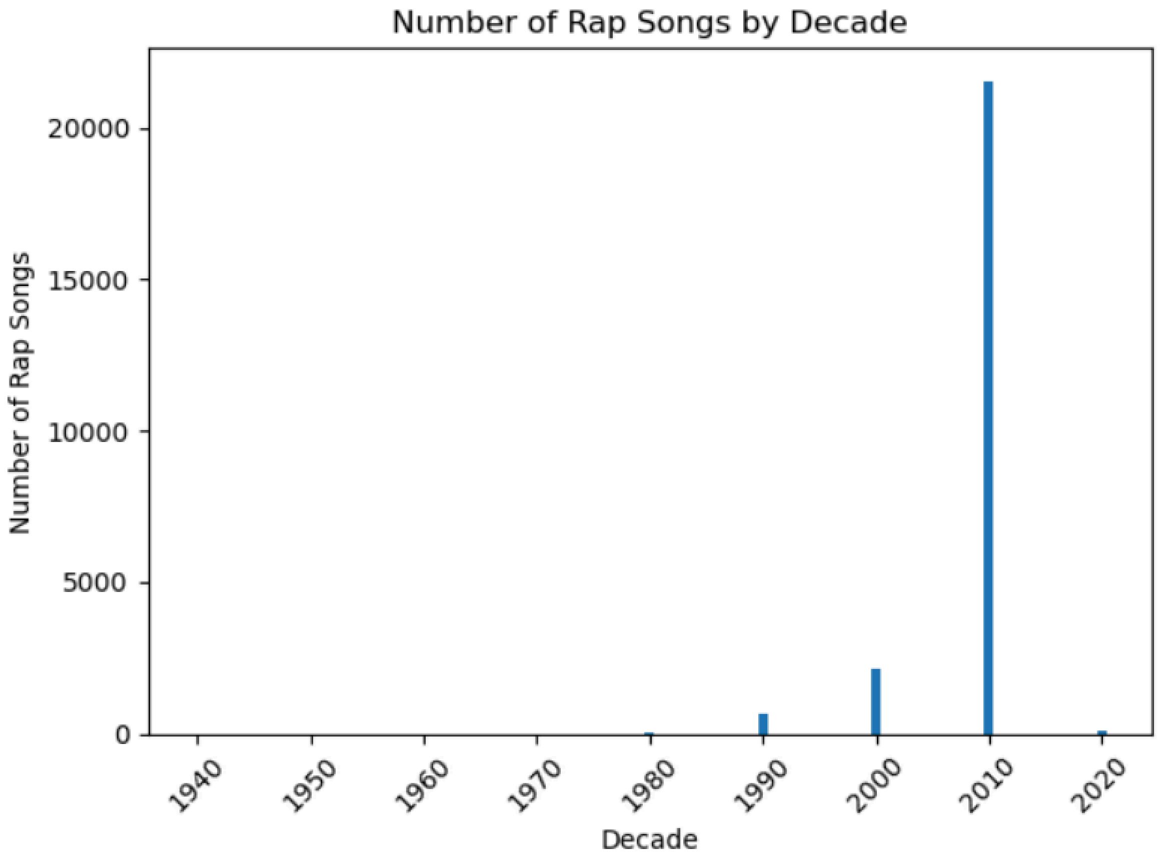
(Figure H)

Modeling

The modeling was performed by splitting the data into 70% training and 30% test sets. Logistic regression was used as the label, rap genre, was being predicted. A model was performed on a 10% sample size because of how large the dataset was. The results for accuracy, precision, recall, and F1 score are as follows respectively: (0.11328500553727826, 0.1318033828317319, 0.04818892334134643, 0.07057482896297777)

The accuracy was very low with ~11% and ~13% of positive outcomes currently identified. This is a result of the small sample size. Lastly, the data was validated by creating a BinaryClassificationEvaluator and CrossValidator. (Appendix D)

Milestone 5: Visualizations



(Figure I)

Visualizations were performed in Python using Matplotlib and Seaborn. The visuals were created using the transformed spark data frame. In the first figure a bar graph was created to show when the rap genre is most prominent and is shown to start appearing in the 1990's and skyrocket in the 2010's. Multiple trials of the model would be needed to create a better visual understanding of the genre's presence in the 2020s.

Milestone 6: Summary and Conclusions

The goal of this project was to predict a song's genre based on the song's lyrics and additional features. Many challenges were present including the large dataset of over a million records with full-length songs for lyrics. This resulted in slow updates when cleaning the data and when reading the cleaned parquet. Cleaning had to be done again with filtering and clearing nulls before feature engineering. The following milestone shows the variables artist and title being too unique of a trait for the song to add value to the model. Therefore the only features used were years, and those derived of the lyrics such as its frequency using tokenization, HF hashing, and IDF. Deriving the sentiment of the lyrics proved to be the hardest hurdle as the main package TextBlob kept disconnecting when performing the analysis. This proved massive delays in the transformation and modeling. This hurdle is still a work of progress however assistance to run the code from another account, sentiment was able to be added as a feature. The massive amount of data proved to be difficult to clean and perform models on because samples had to be used first and this is shown as a disadvantage in the modeling because it was not accurate and will very likely need to be worked on in the future. Overall, this project continues to provide updates on building a more comprehensive model and is very useful for predicting music listener's song preferences. This can be predicted by features used such as the sentiment of the lyrics based on the word's frequency, the year a song was released, and hopefully in the future also taking into account the artist. Music streamers can adapt this in recommending songs or curating playlists with similar songs the consumer is already listening to thus ensuring they return to the streaming service.

For updates on this genre predictor model and closer looks at the scripts used, the following GitHub link is provided: LeslieAucapina/GenrePredictor (github.com)

Appendix A

Creating a Python environment:

```
python3 -m venv pythondev
```

Changing to the directory and activating the virtual environment:

```
cd pythondev
source bin/activate
```

Kaggle command line interface installed:

```
pip3 install kaggle
```

Downloading the Kaggle file to the virtual machine learning:

```
kaggle datasets download -d
carlosgdcj/genius-song-lyrics-with-language-information
```

Showing the data file in the virtual machine

Command:

```
ls -l
```

Output:

```
total 12044624
drwxr-xr-x 2 aucapinaleslie aucapinaleslie       4096 Feb 29
21:24 bin
-rw-r--r-- 1 aucapinaleslie aucapinaleslie 3263274583 Jan 11
2023 genius-song-lyrics-with-language-information.zip
drwxr-xr-x 3 aucapinaleslie aucapinaleslie       4096 Feb 29
21:22 include
drwxr-xr-x 3 aucapinaleslie aucapinaleslie       4096 Feb 29
21:22 lib
```

```
lrwxrwxrwx 1 aucapinaleslie aucapinaleslie          3 Feb 29
21:22 lib64 -> lib
-rw-r--r-- 1 aucapinaleslie aucapinaleslie        169 Mar  1
20:15 pyvenv.cfg
-rw-r--r-- 1 aucapinaleslie aucapinaleslie 9070394868 Jan 11
2023 song_lyrics.csv
```

Creating storage buckets on Google Storage

```
gcloud storage buckets create gs://my-bigdata-project-la
--project=dazzling-ego-415514 --default-storage-class=STANDARD
--location=us-central1 --uniform-bucket-level-access
```

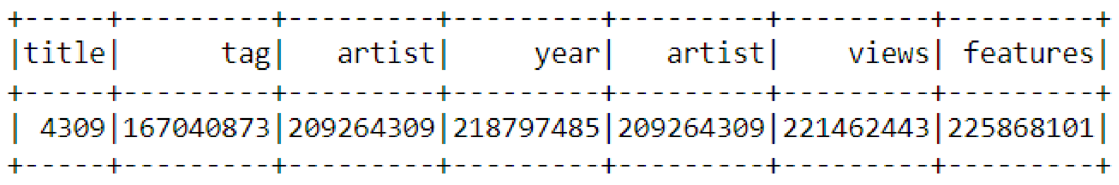
If access is denied then use the following command and follow the given instructions to proceed. Once authorized, input the previous command.

```
gcloud auth login
```

Appendix B

Using Spark to count observations, nulls in columns, and duplicates

```
Spark

# Import some functions we will use later
from pyspark.sql.functions import col, isnan, isnull, when, count,
udf

# Set the logging level for ERRORs only.
sc.setLogLevel("ERROR")

bucket = 'data-project-la/landing/'
filename = 'song_lyrics.csv'
file_path = "gs://my-bigdata-project-la/landing/"
# Create a Spark Dataframe from the file on GCS
sdf = spark.read.csv(file_path, sep=',', header=True,
inferSchema=True)

#Number of observations
sdf.count()

Output: 233187820


# Check to see if some of the columns have NULL values

sdf.select([count(when(isnull(c), c)).alias(c) for c in ["title",
"tag","artist","year","artist","views","features"] ]).show()
```



```
+-----+---------+---------+---------+---------+---------+---------+
|title|      tag|   artist|     year|   artist|    views| features|
+-----+---------+---------+---------+---------+---------+---------+
| 4309|167040873|209264309|218797485|209264309|221462443|225868101|
+-----+---------+---------+---------+---------+---------+---------+
```

```
duplicate_count = sdf.groupBy(sdf.columns).count().filter("count >
1").count()

print(f"Number of duplicate records: {duplicate_count}")
```

```
Output: Number of duplicate records: 26326207
```

DataProc cluster and Python module to define perform_EDA and read the CSV file from Google Cloud Storage

```
# Import pandas library
import pandas as pd
# Set Pandas options to always display floats with a decimal point
# (not scientific notation)
pd.set_option('display.float_format', '{:.2f}'.format)
pd.set_option('display.width', 1000)

import seaborn as sns

def perform_EDA(df, filename):
    """
    perform_EDA(df : pd.DataFrame, filename : str)
    Accepts a dataframe and a text filename as inputs.
    Runs some basic statistics on the data and outputs to console.

    :param df: The Pandas dataframe to explore
    :param filename: The name of the data file
    :return: A dictionary containing the statistics
    """
    eda_results = {}

    eda_results[f"{filename} Number of records"] = df.shape[0]
    eda_results[f"{filename} Number of duplicate records"] = len(df)
- len(df.drop_duplicates())
    eda_results[f"{filename} Info"] = df.info()
    eda_results[f"{filename} Describe"] = df.describe()
    eda_results[f"{filename} Columns with null values"] =
df.columns[df.isnull().any()].tolist()
    eda_results[f"{filename} Number of Rows with null values"] =
df.isnull().any(axis=1).sum()
    eda_results[f"{filename} Integer data type columns"] =
df.select_dtypes(include='int64').columns.tolist()
    eda_results[f"{filename} Float data type columns"] =
df.select_dtypes(include='float64').columns.tolist()

    return eda_results
```

```
all_eda_results = {}

filepath = "gs://my-bigdata-project-la/landing/"
filename_list = ['song_lyrics.csv']
column_names = ['title', 'genre', 'artist', 'year', 'views',
'featured Artists', 'lyrics', 'id',
'language_cld3','language_ft','language']

for filename in filename_list:
    # Read in amazon reviews. Reminder: Tab-separated values files
    print(f"Working on file: {filename}")
    skip = 0
    nrows = 300000
    # Total number of rows to read
    total_rows_to_read = 5100000
    # Read in the first set of rows using Python Pandas
    df_chunk = pd.read_csv(f"{filepath}{filename}", sep=',',
skiprows=skip, nrows=nrows, names=column_names, encoding='utf-8')
    # Increment the skip
    skip = skip + nrows
    while (skip < total_rows_to_read):
        print(f"Reading in {nrows} records starting at {skip}")
        df_chunk = pd.read_csv(f"{filepath}{filename}", sep=',',
skiprows=skip, nrows=nrows, names=column_names, encoding='utf-8')
        skip = skip + nrows
        eda_results = perform_EDA(df_chunk, filename)
        all_eda_results[f"{filename}_chunk{skip // nrows}"] =
eda_results

for chunk, results in all_eda_results.items():
    print(f"EDA results for {chunk}:")
    print(results)
```

OUTPUT for each batch:

```
EDA results for song_lyrics.csv_chunk2:
{'song_lyrics.csv Number of records': 300000, 'song_lyrics.csv Number of duplicate records': 0,
'song_lyrics.csv Info': None, 'song_lyrics.csv Describe':          year     views       id
count 300000.00  300000.00 300000.00
mean    2002.66    3901.06 700960.31
std      41.98   53456.77 147908.87
min        1.00       0.00 423169.00
25%     2000.00      52.00 556393.00
50%     2010.00     190.00 743128.50
75%     2014.00     813.00 825545.25
max     2023.00 9291775.00 917855.00, 'song_lyrics.csv Columns with null values': ['title', 'language_cld3',
'language_ft', 'language'], 'song_lyrics.csv Number of Rows with null values': 8425, 'song_lyrics.csv Integer
data type columns': ['year', 'views', 'id'], 'song_lyrics.csv Float data type columns': []}
EDA results for song_lyrics.csv_chunk3:
{'song_lyrics.csv Number of records': 300000, 'song_lyrics.csv Number of duplicate records': 0,
'song_lyrics.csv Info': None, 'song_lyrics.csv Describe':            year      views        id
count 300000.00  300000.00  300000.00
mean    2001.34     756.41 1062781.70
std       14.30    6698.74   90799.51
min       79.00       0.00  904738.00
25%     1996.00      52.00  984146.75
50%     2004.00     135.00 1062757.50
75%     2011.00     397.00 1141352.25
max     2022.00 1326363.00 1249380.00, 'song_lyrics.csv Columns with null values': ['title', 'language_cld3',
'language_ft', 'language'], 'song_lyrics.csv Number of Rows with null values': 7656, 'song_lyrics.csv Integer
data type columns': ['year', 'views', 'id'], 'song_lyrics.csv Float data type columns': []}
EDA results for song_lyrics.csv_chunk4:
{'song_lyrics.csv Number of records': 300000, 'song_lyrics.csv Number of duplicate records': 0,
'song_lyrics.csv Info': None, 'song_lyrics.csv Describe':            year     views        id
count 300000.00 300000.00  300000.00
mean    2001.34    700.07 1378302.37
std       15.05   5122.96   91492.32
min        1.00      0.00 1220062.00
25%     1996.00     51.00 1298983.75
50%     2004.00    133.00 1378212.50
75%     2011.00    385.00 1457529.25
max     2022.00 766624.00 1546593.00, 'song_lyrics.csv Columns with null values': ['title', 'language_cld3',
'language_ft', 'language'], 'song_lyrics.csv Number of Rows with null values': 7523, 'song_lyrics.csv Integer
data type columns': ['year', 'views', 'id'], 'song_lyrics.csv Float data type columns': []}
```

```
EDA results for song_lyrics.csv_chunk5:
{'song_lyrics.csv Number of records': 300000, 'song_lyrics.csv Number of duplicate records': 0,
'song_lyrics.csv Info': None, 'song_lyrics.csv Describe':          year     views       id
count 300000.00  300000.00  300000.00
mean    2003.16    1091.54 1699340.77
std       16.93   15323.51   97519.16
min        1.00       0.00 1536749.00
25%     1998.00      50.00 1616013.75
50%     2006.00     137.00 1695346.50
75%     2014.00     425.00 1775982.25
max     2023.00 3948378.00 1951617.00, 'song_lyrics.csv Columns with null values': ['title', 'language_cld3',
'language_ft', 'language'], 'song_lyrics.csv Number of Rows with null values': 8783, 'song_lyrics.csv Integer
data type columns': ['year', 'views', 'id'], 'song_lyrics.csv Float data type columns': []}
EDA results for song_lyrics.csv_chunk6:
{'song_lyrics.csv Number of records': 300000, 'song_lyrics.csv Number of duplicate records': 0,
'song_lyrics.csv Info': None, 'song_lyrics.csv Describe':          year     views       id
count 300000.00  300000.00  300000.00
mean    2010.34    4618.14 2183707.43
std       36.36   61252.03  165824.45
min        1.00       0.00 1932730.00
25%     2011.00      67.00 2045385.75
50%     2015.00     229.00 2165926.50
75%     2015.00     927.00 2329440.25
max     2023.00 7871555.00 2486487.00, 'song_lyrics.csv Columns with null values': ['title', 'language_cld3',
'language_ft', 'language'], 'song_lyrics.csv Number of Rows with null values': 13050, 'song_lyrics.csv Integer
data type columns': ['year', 'views', 'id'], 'song_lyrics.csv Float data type columns': []}
EDA results for song_lyrics.csv_chunk7:
{'song_lyrics.csv Number of records': 300000, 'song_lyrics.csv Number of duplicate records': 0,
'song_lyrics.csv Info': None, 'song_lyrics.csv Describe':          year      views       id
count 300000.00   300000.00  300000.00
mean    2012.23     7300.80 2925058.35
std       43.53    96161.19  172442.99
min        1.00        0.00 2460722.00
25%     2015.00       64.00 2860719.25
```

```
50%     2016.00      280.00 2948222.50
75%     2017.00     1528.00 3051807.25
max     2022.00 23351415.00 3155145.00, 'song_lyrics.csv Columns with null values': ['title', 'language_cld3',
'language_ft', 'language'], 'song_lyrics.csv Number of Rows with null values': 12088, 'song_lyrics.csv Integer
data type columns': ['year', 'views', 'id'], 'song_lyrics.csv Float data type columns': []}
EDA results for song_lyrics.csv_chunk8:
{'song_lyrics.csv Number of records': 300000, 'song_lyrics.csv Number of duplicate records': 0,
'song_lyrics.csv Info': None, 'song_lyrics.csv Describe':           year     views       id
count 300000.00  300000.00  300000.00
mean    2013.15    4909.23 3373235.22
std       43.31   61090.67  142877.64
min        1.00       0.00 3147351.00
25%     2015.00      50.00 3249789.50
50%     2017.00     181.00 3351164.50
75%     2017.00     951.00 3509899.25
max     2023.00 9107059.00 3630281.00, 'song_lyrics.csv Columns with null values': ['title', 'language_cld3',
'language_ft', 'language'], 'song_lyrics.csv Number of Rows with null values': 13016, 'song_lyrics.csv Integer
data type columns': ['year', 'views', 'id'], 'song_lyrics.csv Float data type columns': []}
EDA results for song_lyrics.csv_chunk9:
{'song_lyrics.csv Number of records': 300000, 'song_lyrics.csv Number of duplicate records': 0,
'song_lyrics.csv Info': None, 'song_lyrics.csv Describe':           year     views       id
count 300000.00  300000.00  300000.00
mean    2013.39    4044.10 3839414.45
std       43.50   51719.60  125385.90
min        1.00       0.00 3621275.00
25%     2015.00      37.00 3731061.75
50%     2018.00     116.00 3839801.50
75%     2018.00     598.00 3946950.25
max     2023.00 9078511.00 4065161.00, 'song_lyrics.csv Columns with null values': ['title', 'language_cld3',
'language_ft', 'language'], 'song_lyrics.csv Number of Rows with null values': 14084, 'song_lyrics.csv Integer
data type columns': ['year', 'views', 'id'], 'song_lyrics.csv Float data type columns': []}
EDA results for song_lyrics.csv_chunk10:
{'song_lyrics.csv Number of records': 300000, 'song_lyrics.csv Number of duplicate records': 0,
'song_lyrics.csv Info': None, 'song_lyrics.csv Describe':           year     views       id
```

```
count 300000.00  300000.00  300000.00
mean    2013.31    2527.42 4291266.82
std       48.12   36872.24  137769.49
min        1.00       0.00 4057545.00
25%     2016.00      26.00 4171247.75
50%     2018.00      82.00 4289514.50
75%     2019.00     423.00 4412124.25
max     2023.00 9072131.00 4569264.00, 'song_lyrics.csv Columns with null values': ['title', 'language_cld3',
'language_ft', 'language'], 'song_lyrics.csv Number of Rows with null values': 16057, 'song_lyrics.csv Integer
data type columns': ['year', 'views', 'id'], 'song_lyrics.csv Float data type columns': []}
EDA results for song_lyrics.csv_chunk11:
{'song_lyrics.csv Number of records': 300000, 'song_lyrics.csv Number of duplicate records': 0,
'song_lyrics.csv Info': None, 'song_lyrics.csv Describe':           year      views         id
count 300000.00  300000.00  300000.00
mean    2013.60    2449.48 4763272.56
std       46.84   26540.11  134579.68
min        1.00       0.00 4531572.00
25%     2017.00      18.00 4646373.50
50%     2019.00      62.00 4762417.50
75%     2019.00     348.00 4883154.25
max     2024.00 2882193.00 5011344.00, 'song_lyrics.csv Columns with null values': ['title', 'language_cld3',
'language_ft', 'language'], 'song_lyrics.csv Number of Rows with null values': 16645, 'song_lyrics.csv Integer
data type columns': ['year', 'views', 'id'], 'song_lyrics.csv Float data type columns': []}
EDA results for song_lyrics.csv_chunk12:
{'song_lyrics.csv Number of records': 300000, 'song_lyrics.csv Number of duplicate records': 0,
'song_lyrics.csv Info': None, 'song_lyrics.csv Describe':           year      views         id
count 300000.00  300000.00  300000.00
mean    2014.41    1868.08 5217636.46
std       48.78   26452.79  129167.65
min        1.00       0.00 4994294.00
25%     2017.00      13.00 5105171.50
50%     2019.00      43.00 5217647.00
75%     2020.00     239.00 5329371.25
```

```
max     2023.00 6129369.00 5444314.00, 'song_lyrics.csv Columns with null values': ['title', 'language_cld3',
'language_ft', 'language'], 'song_lyrics.csv Number of Rows with null values': 15287, 'song_lyrics.csv Integer
data type columns': ['year', 'views', 'id'], 'song_lyrics.csv Float data type columns': []}
EDA results for song_lyrics.csv_chunk13:
{'song_lyrics.csv Number of records': 300000, 'song_lyrics.csv Number of duplicate records': 0,
'song_lyrics.csv Info': None, 'song_lyrics.csv Describe':           year      views        id
count 300000.00   300000.00  300000.00
mean    2015.08     1472.75 5667915.00
std       46.05    37574.09  131644.14
min        1.00        0.00 5441887.00
25%     2018.00       11.00 5553038.75
50%     2020.00       33.00 5666321.50
75%     2020.00      170.00 5782582.00
max     2100.00 16003444.00 5919651.00, 'song_lyrics.csv Columns with null values': ['title', 'language_cld3',
'language_ft', 'language'], 'song_lyrics.csv Number of Rows with null values': 16955, 'song_lyrics.csv Integer
data type columns': ['year', 'views', 'id'], 'song_lyrics.csv Float data type columns': []}
EDA results for song_lyrics.csv_chunk14:
{'song_lyrics.csv Number of records': 300000, 'song_lyrics.csv Number of duplicate records': 0,
'song_lyrics.csv Info': None, 'song_lyrics.csv Describe':           year     views        id
count 300000.00  300000.00  300000.00
mean    2015.75    1268.89 6132294.85
std       42.42   17645.85  137798.94
min        1.00       0.00 5897785.00
25%     2019.00      11.00 6010213.50
50%     2020.00      30.00 6133277.50
75%     2020.00     151.00 6251918.50
max     2023.00 2691880.00 6374580.00, 'song_lyrics.csv Columns with null values': ['title', 'language_cld3',
'language_ft', 'language'], 'song_lyrics.csv Number of Rows with null values': 16579, 'song_lyrics.csv Integer
data type columns': ['year', 'views', 'id'], 'song_lyrics.csv Float data type columns': []}
EDA results for song_lyrics.csv_chunk15:
{'song_lyrics.csv Number of records': 300000, 'song_lyrics.csv Number of duplicate records': 0,
'song_lyrics.csv Info': None, 'song_lyrics.csv Describe':           year     views        id
count 300000.00  300000.00  300000.00
mean    2015.64     974.54 6602314.59
```

```
std      46.25   18505.01  131684.26
min       1.00      0.00 6372066.00
25%     2019.00      8.00 6489302.75
50%     2021.00     23.00 6602506.50
75%     2021.00    112.00 6715957.25
max     2023.00 3282336.00 6853497.00, 'song_lyrics.csv Columns with null values': ['title', 'language_cld3',
'language_ft', 'language'], 'song_lyrics.csv Number of Rows with null values': 17618, 'song_lyrics.csv Integer
data type columns': ['year', 'views', 'id'], 'song_lyrics.csv Float data type columns': []}
EDA results for song_lyrics.csv_chunk16:
{'song_lyrics.csv Number of records': 300000, 'song_lyrics.csv Number of duplicate records': 0,
'song_lyrics.csv Info': None, 'song_lyrics.csv Describe':          year      views        id
count 300000.00  300000.00  300000.00
mean    2015.81     879.49 7069227.39
std      57.89   14057.24  138714.52
min       1.00      0.00 6831656.00
25%     2020.00      7.00 6948065.75
50%     2021.00     20.00 7067646.00
75%     2021.00     96.00 7190270.75
max     2023.00 2713579.00 7344068.00, 'song_lyrics.csv Columns with null values': ['title', 'language_cld3',
'language_ft', 'language'], 'song_lyrics.csv Number of Rows with null values': 17774, 'song_lyrics.csv Integer
data type columns': ['year', 'views', 'id'], 'song_lyrics.csv Float data type columns': []}
EDA results for song_lyrics.csv_chunk17:
{'song_lyrics.csv Number of records': 300000, 'song_lyrics.csv Number of duplicate records': 0,
'song_lyrics.csv Info': None, 'song_lyrics.csv Describe':          year      views        id
count 300000.00  300000.00  300000.00
mean    2016.59     477.18 7557021.47
std      46.20    7523.03  146156.23
min       1.00      0.00 7309805.00
25%     2020.00      5.00 7428909.75
50%     2021.00     11.00 7554461.50
75%     2022.00     43.00 7683028.25
max     2024.00 1808788.00 7827129.00, 'song_lyrics.csv Columns with null values': ['title', 'language_cld3',
'language_ft', 'language'], 'song_lyrics.csv Number of Rows with null values': 17390, 'song_lyrics.csv Integer
data type columns': ['year', 'views', 'id'], 'song_lyrics.csv Float data type columns': []}
```

Scatter Plot

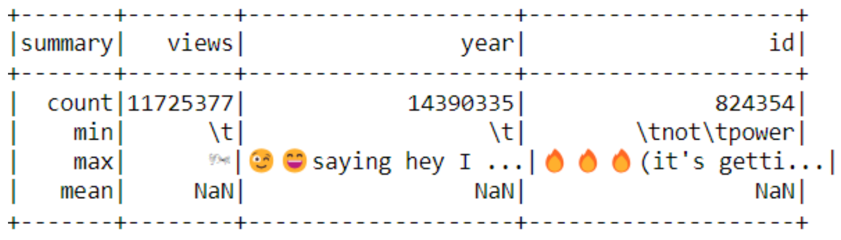
```
import seaborn as sns
import matplotlib.pyplot as plt

# Assuming df_chunk contains your dataframe with columns 'year' and
'views'

# Set the style of seaborn
sns.set_style("whitegrid")

# Create the scatterplot
sns.scatterplot(data=df_chunk, x='year', y='views')

# Set labels and title
plt.xlabel('Year of Release')
plt.ylabel('Views')
plt.title('Scatterplot of Views vs. Year of Release')

# Show the plot
plt.show()
```

Counterplot

```
import seaborn as sns
import matplotlib.pyplot as plt

# Assuming your dataframe is named 'df' and you want to plot the
'Genre' column
sns.countplot(data=df_chunk, x='genre')
plt.title('Distribution of Songs by Genre')
plt.xlabel('Genre')
plt.ylabel('Count')
plt.xticks(rotation=45)  # Rotate x-axis labels for better
readability if needed
plt.show()
```

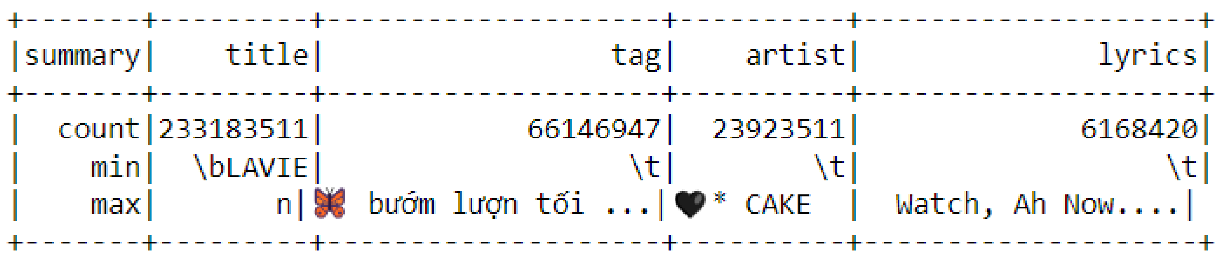
Appendix C

```
spark

# Import some functions we will use later
from pyspark.sql.functions import col, isnan, isnull, when, count,
udf
from pyspark.sql import SparkSession
from pyspark.sql.types import StructField, StructType, StringType,
IntegerType, FloatType

# Set the logging level for ERRORs only.
sc.setLogLevel("ERROR")

bucket = 'data-project-la/landing/'
filename = 'song_lyrics.csv'
file_path = "gs://my-bigdata-project-la/landing/"
# Create a Spark Dataframe from the file on GCS
sdf = spark.read.csv(file_path, sep=',', header=True,
inferSchema=True)


# Look at statistics for some specific columns
sdf.select("views", "year", "id").summary("count", "min", "max",
"mean").show()
```



```
+-------+--------+--------------------+--------------------+
|summary|   views|                year|                  id|
+-------+--------+--------------------+--------------------+
|  count|11725377|            14390335|              824354|
|    min|      \t|                  \t|        \tnot\tpower|
|    max|[illegible]|😉😄saying hey I ...|🔥🔥🔥(it's getti...|
|   mean|     NaN|                 NaN|                 NaN|
+-------+--------+--------------------+--------------------+
```

```
# Look at the Review headline and Review Body
sdf.select("title", "tag", "artist", "lyrics").summary("count",
"min", "max").show()
```

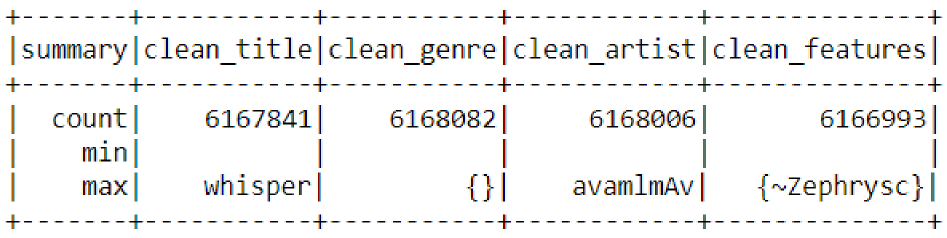


```
+-------+---------+--------------------+----------+--------------------+
|summary|    title|                 tag|    artist|              lyrics|
+-------+---------+--------------------+----------+--------------------+
|  count|233183511|            66146947|  23923511|             6168420|
|    min|  \bLAVIE|                  \t|        \t|                  \t|
|    max|        n|🦋 bướm lượn tối ...|🖤* CAKE   |  Watch, Ah Now....|
+-------+---------+--------------------+----------+--------------------+
```

```
# Define a function to strip out any non-ascii characters
def ascii_only(mystring):
  if mystring:
    return mystring.encode('ascii', 'ignore').decode('ascii')
  else:
    return None

# Turn this function into a User-Defined Function (UDF)
ascii_udf = udf(ascii_only)

sdf = sdf.withColumn("clean_title", ascii_udf('title'))
sdf = sdf.withColumn("clean_genre", ascii_udf('tag'))
sdf = sdf.withColumn("clean_artist", ascii_udf('artist'))
sdf = sdf.withColumn("clean_year", ascii_udf('year'))
sdf = sdf.withColumn("clean_views", ascii_udf('views'))
sdf = sdf.withColumn("clean_features", ascii_udf('features'))
sdf = sdf.withColumn("clean_lyrics", ascii_udf('lyrics'))
sdf = sdf.withColumn("clean_id", ascii_udf('id'))


# Drop columns "column1" and "column2"
sdf = sdf.drop("title","tag","artist","year","views","features",
"lyrics", "id","language_cld3","language_ft")
```

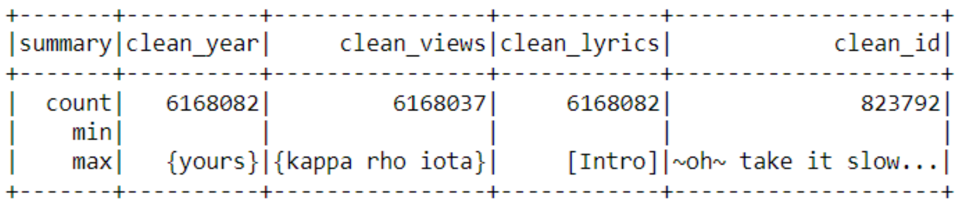
```
#Re-check the cleaned headline and body
```

```
sdf.select("clean_title",
"clean_genre","clean_artist","clean_features").summary("count",
"min", "max").show()
```



```
+-------+-----------+-----------+------------+--------------+
|summary|clean_title|clean_genre|clean_artist|clean_features|
+-------+-----------+-----------+------------+--------------+
|  count|    6167841|    6168082|     6168006|       6166993|
|    min|           |           |            |              |
|    max|    whisper|         {}|    avamlmAv|    {~Zephrysc}|
+-------+-----------+-----------+------------+--------------+
```

```
sdf.select("clean_year",
"clean_views","clean_lyrics","clean_id").summary("count", "min",
"max").show()
```



```
+-------+----------+----------------+------------+--------------------+
|summary|clean_year|     clean_views|clean_lyrics|            clean_id|
+-------+----------+----------------+------------+--------------------+
|  count|   6168082|         6168037|     6168082|              823792|
|    min|          |                |            |                    |
|    max|   {yours}|{kappa rho iota}|     [Intro]|~oh~ take it slow...|
+-------+----------+----------------+------------+--------------------+
```

```
# Filter out records with non-numeric values in the id column
sdf = sdf.filter(col("id").rlike("^[0-9]+$"))

#count duplicates
# Group by the columns you suspect have duplicates and count the
occurrences
duplicate_counts = sdf.groupBy(["clean_title", "clean_genre",
"clean_artist", "clean_year", "clean_views", "clean_features",
"clean_lyrics", "clean_id"]).count()

# Filter out records where count is greater than 1 to identify
duplicates
duplicates = duplicate_counts.filter(col("count") > 1)

# Count the total number of duplicate records
total_duplicates = duplicates.count()

# Show the total number of duplicate records
print("Total number of duplicates:", total_duplicates)

# Show the duplicate records
```

```
sdf= sdf.dropDuplicates()

# Define the desired order of columns
desired_order = ["clean_id", "clean_title", "clean_genre",
"clean_artist", "clean_year", "clean_views", "clean_features",
"clean_lyrics","language"]

# Select columns in the desired order
sdf = sdf.select(desired_order)

from pyspark.sql.types import StructField, StructType, StringType,
IntegerType

# Define the new schema with modified column names and types
new_schema = StructType([
    StructField("id", IntegerType(), nullable=True),
    StructField("title", StringType(), nullable=True),
    StructField("genre", StringType(), nullable=True),
    StructField("artist", StringType(), nullable=True),
    StructField("year", IntegerType(), nullable=True),
    StructField("views", IntegerType(), nullable=True),
    StructField("features", StringType(), nullable=True),
    StructField("lyrics", StringType(), nullable=True),
    StructField("language", StringType(), nullable=True)

])
sdf = sdf.toDF(*[field.name for field in new_schema.fields])

sdf = sdf.orderBy("id")

from pyspark.sql.functions import monotonically_increasing_id

# Add a new column with monotonically increasing IDs
sdf = sdf.withColumn("new_id", (monotonically_increasing_id() +
1).cast("int"))

# Drop the old "id" column and rename the new "new_id" column to "id"
sdf = sdf.drop("id").withColumnRenamed("new_id", "id")

# Drop Nulls
# Drop some of the records where the certain columns are empty (null
or nan)
sdf = sdf.na.drop(subset=["genre", "lyrics"])
# Save the cleaned data in a new file. Use Parquet file format.
# Options:
https://spark.apache.org/docs/latest/sql-data-sources-parquet.html
output_file_path=
f"gs://{'my-bigdata-project-la/cleaned'}/cleaned_song_lyrics.parquet"
sdf.write.mode("overwrite").parquet(output_file_path)
```

Appendix D

Feature engineering

In a virtual machine Instance

```
python3 -m venv myenv
source myenv/bin/activate
mkdir dependencies
cd dependencies
pip download textblob -d .

zip -r dependencies.zip .
```

In Jupyter Notebook within DataProc Cluster

```
Spark

%pip install textblob

from pyspark.sql import SparkSession

# Create SparkSession
spark = SparkSession.builder.getOrCreate()

# Add the dependencies zip file to the SparkContext
spark.sparkContext.addPyFile("/home/aucapinaleslie/dependencies/depen
dencies.zip")

!source /home/aucapinaleslie/myenv/bin/activate

from textblob import TextBlob

try:
    # Create a TextBlob object
    blob = TextBlob("This is a test sentence")
    # Print the sentiment
    print("Sentiment:", blob.sentiment)
```

```
except ImportError:
    print("Failed to import TextBlob")

try:
    # Create a TextBlob object
    blob = TextBlob("This is a test sentence")
    # Print the sentiment
    print("Sentiment:", blob.sentiment)
except ImportError:
    print("Failed to import TextBlob")

from pyspark.sql.functions import col
from pyspark.sql.functions import *
from pyspark.ml.feature import StringIndexer, OneHotEncoder,
VectorAssembler
from pyspark.ml import Pipeline
# Import the logistic regression model
from pyspark.ml.classification import LogisticRegression,
LogisticRegressionModel
# Import the evaluation module
from pyspark.ml.evaluation import *
# Import the model tuning module
from pyspark.ml.tuning import *
import numpy as np


filepath =
'gs://my-project-bucket-lyrics/cleaned/song_lyrics_cleaned.parquet'
sdf = spark.read.parquet(filepath )


sdf = sdf.filter( sdf.year > 1900).filter( sdf.year < 2025)

sdf.select('year').show()

sdf = sdf.withColumn("year", col("year").cast("int"))

sdf.printSchema()

# Drop specific columns
columns_to_drop = ['views', 'features', 'language', 'language_cld3',
'language_ft']
sdf = sdf.drop(*columns_to_drop)

# Rename the 'tag' column to 'genre'
```

```
sdf = sdf.withColumnRenamed("tag", "genre")

sdf.printSchema()


# Drop rows with null values in the "year" column
sdf = sdf.dropna(subset=["year","artist","title","lyrics","genre"])

sdf.select("year","artist","title","lyrics","genre").show()

sdf_features = sdf

from pyspark.ml.feature import RegexTokenizer, HashingTF, IDF
from pyspark.sql.functions import monotonically_increasing_id



# Create a RegexTokenizer instance
regexTokenizer = RegexTokenizer(inputCol="lyrics", outputCol="words",
pattern="\\w+", gaps=False)

# Apply the tokenizer to the lyrics DataFrame
sdf_features = regexTokenizer.transform(sdf_features)

# Create an instance of HashingTF
hashingTF = HashingTF(numFeatures=4096, inputCol="words",
outputCol="word_features")

# Apply the HashingTF transformation to your DataFrame containing
tokenized words
sdf_features = hashingTF.transform(sdf_features)

# # Show the result
#
term_freq_sdf.select(['words','word_features']).show(truncate=False)

# Create an instance of IDF, importance of word based on which most
repeated
idf = IDF(inputCol='word_features', outputCol="idf_features",
minDocFreq=1)
```

```
#Create the idf_features
# Fit IDF model to your DataFrame, this idf is just another feature
sdf_features = idf.fit(sdf_features).transform(sdf_features)

sdf_features.select('lyrics', 'idf_features').show(10)

#Output:
+--------------------+--------------------+
|              lyrics|        idf_features|
+--------------------+--------------------+
|[Verse 1 – Scotty...|(4096,[16,87,116,...|
|"O bury me not on...|(4096,[36,55,87,1...|
|[Verse 1 - Santos...|(4096,[23,43,77,1...|
|[Produced by: Fra...|(4096,[25,28,54,1...|
|[Chorus]\nMy ambi...|(4096,[32,120,122...|
|[Intro]\nBaby, we...|(4096,[81,87,106,...|
|ateos - audio scr...|(4096,[30,32,50,7...|
|[Verse 1]\nI buil...|(4096,[81,87,125,...|
|[Intro: Fizzy]\nI...|(4096,[32,77,87,1...|
|Mister Speaker, o...|(4096,[10,31,63,7...|
+--------------------+--------------------+
only showing top 10 rows


#Create an instance of sentiment

from pyspark.sql.functions import udf
from pyspark.sql.types import DoubleType
from textblob import TextBlob
import numpy

# Define a function to perform sentiment analysis on TF-IDF vectors
def sentiment_analysis(words_string):
    # Convert TF-IDF vector back to text
    # text = " ".join(tfidf_vector) if tfidf_vector else ""
    # Perform sentiment analysis using TextBlob
    # RH: Needs to work on a STRING not a vector
    sentiment = TextBlob(words_string).sentiment.polarity
    return sentiment
```

```
# Define a UDF for the sentiment analysis function
sentiment_analysis_udf = udf(sentiment_analysis, DoubleType())

# Apply sentiment analysis to the TF-IDF vectors
sdf_features = sdf_features.withColumn("sentiment",
sentiment_analysis_udf("lyrics"))

sdf_features.printSchema()

sdf_features.select('lyrics', 'idf_features', 'sentiment').show(10)
```

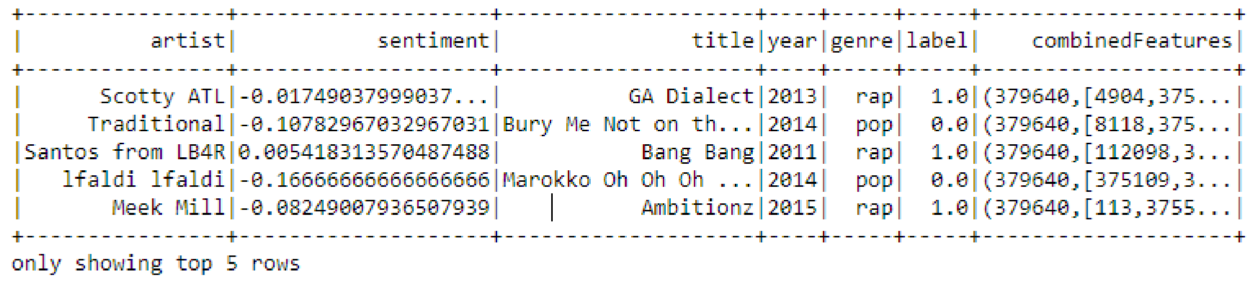
```
+--------------------+--------------------+--------------------+
|              lyrics|        idf_features|           sentiment|
+--------------------+--------------------+--------------------+
|[Verse 1 – Scotty...|(4096,[16,87,116,...|-0.01749037999037...|
|"O bury me not on...|(4096,[36,55,87,1...|-0.10782967032967031|
|[Verse 1 - Santos...|(4096,[23,43,77,1...|0.005418313570487488|
|[Produced by: Fra...|(4096,[25,28,54,1...|-0.16666666666666666|
|[Chorus]\nMy ambi...|(4096,[32,120,122...|-0.08249007936507939|
|[Intro]\nBaby, we...|(4096,[81,87,106,...| 0.35790249433106575|
|ateos - audio scr...|(4096,[30,32,50,7...|0.025307291666666676|
|[Verse 1]\nI buil...|(4096,[81,87,125,...| 0.37946938775510203|
|[Intro: Fizzy]\nI...|(4096,[32,77,87,1...| 0.11273638642059697|
|Mister Speaker, o...|(4096,[10,31,63,7...|  0.0817458062770563|
+--------------------+--------------------+--------------------+
only showing top 10 rows
```

```
from pyspark.ml.feature import VectorAssembler, MinMaxScaler
from pyspark.sql.functions import col, when
from pyspark.ml.feature import Bucketizer

# Define splits for bucketizing years
year_splits = [-float("inf"), 1950.0, 1970.0, 1990.0, 2010.0,
float("inf")]

# Create a Bucketizer instance for year
## RH: This was a bug - it was reading from the original 'sdf' so it
woud erase all of the features
## sdf_features = sdf.withColumn('yearBucket',
sdf_features = sdf_features.withColumn('yearBucket',
               when(sdf.year >= 2010, 5)
              .when(sdf.year >= 2000, 4)
              .when(sdf.year >= 1990, 3)
              .when(sdf.year >= 1980, 2)
              .when(sdf.year >= 1970, 1)
              .otherwise(0))

sdf_features

from pyspark.ml import Pipeline
from pyspark.ml.feature import OneHotEncoder, StringIndexer,
VectorAssembler
from pyspark.sql.functions import col

# Create an indexer for just the artist string-based columns.

##  RH  indexer = StringIndexer(inputCols=["artist", "title"],
outputCols=["artistIndex", "titleIndex"])
indexer = StringIndexer(inputCols=["artist"],
outputCols=["artistIndex"])

# sdf_features = indexer.fit(sdf_features).transform(sdf_features)
```

# Creating the pipeline and transforming data

```
song_pipe = Pipeline(stages=[indexer, encoder, assembler])

# Call .fit to transform the data
transformed_sdf = song_pipe.fit(sdf_features).transform(sdf_features)

# Predict if this song is rap or not.
transformed_sdf = transformed_sdf.withColumn("label",
when(transformed_sdf["genre"] == "rap", 1.0).otherwise(0.0))

# Show the DataFrame with labels
transformed_sdf.select("genre", "label").show()

+-----+-----+
|genre|label|
+-----+-----+
|  rap|  1.0|
|  pop|  0.0|
|  rap|  1.0|
|  pop|  0.0|
|  rap|  1.0|
|   rb|  0.0|
| misc|  0.0|
|  pop|  0.0|
|  rap|  1.0|
| misc|  0.0|
| misc|  0.0|
|  rap|  1.0|
| misc|  0.0|
|  rap|  1.0|
|  rap|  1.0|
|  rap|  1.0|
|  rap|  1.0|
|  rap|  1.0|
|  rap|  1.0|
| misc|  0.0|
+-----+-----+
only showing top 20 rows
```

```
transformed_sdf.select('artist','sentiment','title','year','genre','l
abel','combinedFeatures').show(5)
```



```
+----------------+--------------------+--------------------+----+-----+-----+--------------------+
|          artist|           sentiment|               title|year|genre|label|    combinedFeatures|
+----------------+--------------------+--------------------+----+-----+-----+--------------------+
|      Scotty ATL|-0.01749037999037...|          GA Dialect|2013|  rap|  1.0|(379640,[4904,375...|
|     Traditional|-0.10782967032967031|Bury Me Not on th...|2014|  pop|  0.0|(379640,[8118,375...|
|Santos from LB4R|0.005418313570487488|           Bang Bang|2011|  rap|  1.0|(379640,[112098,3...|
|   lfaldi lfaldi|-0.16666666666666666|Marokko Oh Oh Oh ...|2014|  pop|  0.0|(379640,[375109,3...|
|       Meek Mill|-0.08249007936507939|           Ambitionz|2015|  rap|  1.0|(379640,[113,3755...|
+----------------+--------------------+--------------------+----+-----+-----+--------------------+
only showing top 5 rows
```

```
# Index the 'genre' column
indexer_genre = StringIndexer(inputCol="genre",
outputCol="GenreIndex")
indexed_sdf = indexer_genre.fit(sdf).transform(sdf)
```

```
# Create label based on index of genre
label_sdf = indexed_sdf.withColumn("label",
                  when(indexed_sdf["GenreIndex"] == 0, "pop")
                 .when(indexed_sdf["GenreIndex"] == 1, "rap")
                 .when(indexed_sdf["GenreIndex"] == 2, "rock")
                 .when(indexed_sdf["GenreIndex"] == 3, "rb")
                 .when(indexed_sdf["GenreIndex"] == 4, "misc")
                 .when(indexed_sdf["GenreIndex"] == 5, "country")
                 .otherwise("unknown"))

# Show the DataFrame with labels
label_sdf.select("genre", "GenreIndex", "label").show()


from pyspark.ml.feature import VectorAssembler, MinMaxScaler
from pyspark.sql.functions import col
from pyspark.ml.feature import Bucketizer

# Define splits for bucketizing years
year_splits = [-float("inf"), 1950, 1970, 1990, 2010, float("inf")]

# Create a Bucketizer instance for year
bucketizer = Bucketizer(splits=year_splits, inputCol="year",
outputCol="yearBucket")

# Apply the bucketizer to your DataFrame
sdf = bucketizer.transform(sdf)

from pyspark.ml import Pipeline
from pyspark.ml.feature import OneHotEncoder, StringIndexer,
VectorAssembler
from pyspark.sql.functions import col

# Create an indexer for the three string-based columns.
indexer = StringIndexer(inputCols=["artist", "title"],
outputCols=["artistIndex", "titleIndex"])

# Create an encoder for the three indexes and the year bucket column.
encoder = OneHotEncoder(inputCols=["artistIndex", "titleIndex",
"yearBucket"],
                        outputCols=["artistVector", "titleVector",
"yearVector"],
                        dropLast=False)
```

```
# Create an assembler for the individual feature vectors and the year
vector.
assembler = VectorAssembler(inputCols=["artistVector", "titleVector",
"yearVector"], outputCol="features")

# Create the pipeline
song_pipe = Pipeline(stages=[indexer, encoder, assembler])

# Call .fit to transform the data
transformed_sdf = song_pipe.fit(sdf).transform(sdf)

# Review the transformed features
transformed_sdf.select("artist", "title", "year",
"features").show(30, truncate=False)
```

--

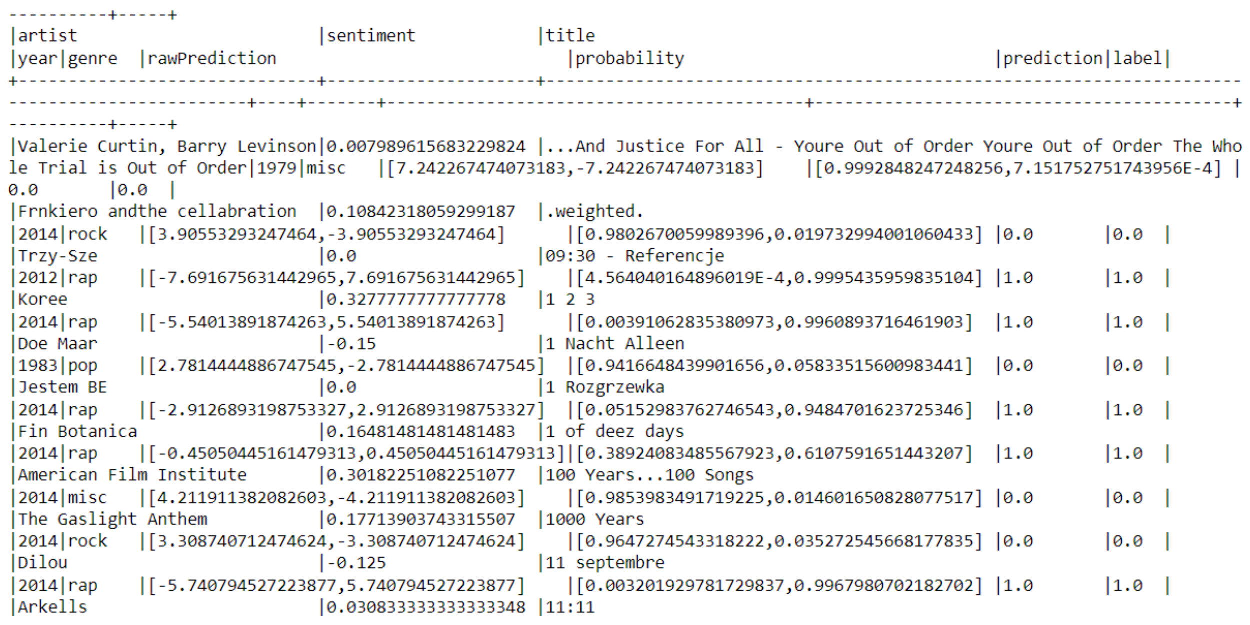
```
# Assuming you have a DataFrame named sdf with a column named
"lyrics"
# Add a unique identifier column
lyrics_sdf = sdf.withColumn("id", monotonically_increasing_id())

# Create a RegexTokenizer instance
regexTokenizer = RegexTokenizer(inputCol="lyrics", outputCol="words",
pattern="\\w+", gaps=False)

# Apply the tokenizer to the lyrics DataFrame
words_sdf = regexTokenizer.transform(lyrics_sdf)

# Show the result
words_sdf.select("id", "lyrics", "words").show(truncate=False)

# Create an instance of HashingTF
hashingTF = HashingTF(numFeatures=4096, inputCol="words",
outputCol="word_features")

# Apply the HashingTF transformation to your DataFrame containing
tokenized words
term_freq_sdf = hashingTF.transform(words_sdf)
```

```
# Show the result
term_freq_sdf.select(['words','word_features']).show(truncate=False)

# Create an instance of IDF
idf = IDF(inputCol='word_features', outputCol="features",
minDocFreq=1)

# Fit IDF model to your DataFrame
idfModel = idf.fit(term_freq_sdf)

# Transform the DataFrame using the IDF model
scaled_sdf = idfModel.transform(term_freq_sdf)

# Show the result
scaled_sdf.select("id", "features").show(truncate=False)


# Define a function to perform sentiment analysis on some text
def sentiment_analysis(some_text):
    sentiment = TextBlob(some_text).sentiment.polarity
    return sentiment

# Turn our function into a UDF
sentiment_analysis_udf = udf(sentiment_analysis, DoubleType())

# Apply sentiment analysis to the lyrics DataFrame
sentiment_sdf = words_sdf.withColumn("sentiment",
sentiment_analysis_udf("lyrics"))

# Show the result
sentiment_sdf.select("id", "lyrics",
"sentiment").show(truncate=False)
```

Modeling

```
# Now you can use the combinedFeatures to predict the label
transformed_sdf = transformed_sdf.sample(False, 0.10)


# Create a LogisticRegression Estimator
lr = LogisticRegression(featuresCol="combinedFeatures",
labelCol="label")
```

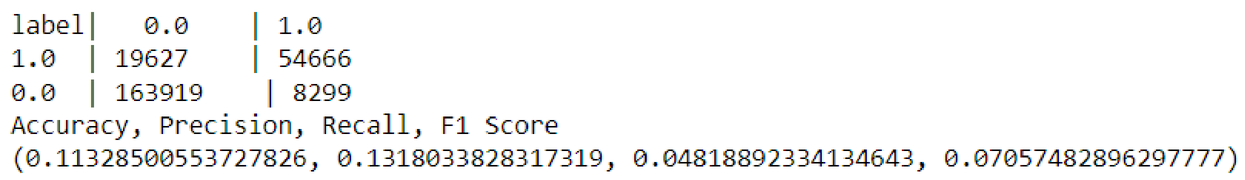
```
lr_model = lr.fit(trainingData)
lr_model = lr.fit(trainingData)
Output:
Intercept:  -6.918475386580401


# Test the model on the testData
test_results = lr_model.transform(testData)

# Show the test results
test_results.select('artist','sentiment','title','year','genre',
'rawPrediction', 'probability', 'prediction','label').show(20,
truncate=False)
```



```
----------+-----+
|artist                        |sentiment           |title
|year|genre  |rawPrediction                            |probability                                   |prediction|label|
+------------------------------+--------------------+------------------------------------------------------------------
-----------------------+----+-------+-----------------------------------------+-----------------------------------------+
----------+-----+
|Valerie Curtin, Barry Levinson|0.007989615683229824 |...And Justice For All - Youre Out of Order Youre Out of Order The Who
le Trial is Out of Order|1979|misc   |[7.242267474073183,-7.242267474073183]    |[0.9992848247248256,7.151752751743956E-4] |
0.0       |0.0  |
|Frnkiero andthe cellabration  |0.10842318059299187  |.weighted.
|2014|rock   |[3.90553293247464,-3.90553293247464]      |[0.9802670059989396,0.019732994001060433] |0.0       |0.0  |
|Trzy-Sze                      |0.0                  |09:30 - Referencje
|2012|rap    |[-7.691675631442965,7.691675631442965]    |[4.564040164896019E-4,0.9995435959835104] |1.0       |1.0  |
|Koree                         |0.3277777777777778   |1 2 3
|2014|rap    |[-5.54013891874263,5.54013891874263]      |[0.00391062835380973,0.9960893716461903]  |1.0       |1.0  |
|Doe Maar                      |-0.15                |1 Nacht Alleen
|1983|pop    |[2.7814444886747545,-2.7814444886747545]  |[0.9416648439901656,0.05833515600983441]  |0.0       |0.0  |
|Jestem BE                     |0.0                  |1 Rozgrzewka
|2014|rap    |[-2.9126893198753327,2.9126893198753327]  |[0.05152983762746543,0.9484701623725346]  |1.0       |1.0  |
|Fin Botanica                  |0.16481481481481483  |1 of deez days
|2014|rap    |[-0.45050445161479313,0.45050445161479313]|[0.38924083485567923,0.6107591651443207]  |1.0       |1.0  |
|American Film Institute       |0.30182251082251077  |100 Years...100 Songs
|2014|misc   |[4.211911382082603,-4.211911382082603]    |[0.9853983491719225,0.014601650828077517] |0.0       |0.0  |
|The Gaslight Anthem           |0.17713903743315507  |1000 Years
|2014|rock   |[3.308740712474624,-3.308740712474624]    |[0.9647274543318222,0.035272545668177835] |0.0       |0.0  |
|Dilou                         |-0.125               |11 septembre
|2014|rap    |[-5.740794527223877,5.740794527223877]    |[0.003201929781729837,0.9967980702182702] |1.0       |1.0  |
|Arkells                       |0.030833333333333348 |11:11
```

```
# Show the confusion matrix
#
test_results.groupby('label').pivot('prediction').count().sort('label
').show()

confusion_matrix =
test_results.groupby('label').pivot('prediction').count().fillna(0).c
ollect()

print("label|   0.0    | 1.0")
print( confusion_matrix[0][0]," |", confusion_matrix[0][1],"
|",confusion_matrix[0][2])
```

```
print( confusion_matrix[1][0]," |", confusion_matrix[1][1],"
|",confusion_matrix[1][2])

def calculate_recall_precision(confusion_matrix):
    tn = confusion_matrix[0][1]  # True Negative
    fp = confusion_matrix[0][2]  # False Positive
    fn = confusion_matrix[1][1]  # False Negative
    tp = confusion_matrix[1][2]  # True Positive
    precision = tp / ( tp + fp )
    recall = tp / ( tp + fn )
    accuracy = ( tp + tn ) / ( tp + tn + fp + fn )
    f1_score = 2 * ( ( precision * recall ) / ( precision + recall )
)
    return accuracy, precision, recall, f1_score

print("Accuracy, Precision, Recall, F1 Score")
print( calculate_recall_precision(confusion_matrix) )
```



```
label|   0.0    | 1.0
1.0  | 19627    | 54666
0.0  | 163919    | 8299
Accuracy, Precision, Recall, F1 Score
(0.11328500553727826, 0.1318033828317319, 0.04818892334134643, 0.07057482896297777)
```

```
model_folder =
"gs://my-bigdata-project-la/models/logistic_regression_model"
lr_model.save(model_folder)
print(f"Model saved to {model_folder}")
```

Validation

```
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.ml.classification import LogisticRegression

# Assuming lr is your LogisticRegression estimator and trainingData
is your training dataset

# Sample a fraction of your training data
sampled_trainingData = trainingData.sample(fraction=0.1, seed=42)

# Create a BinaryClassificationEvaluator with AUC-ROC as the metric
evaluator = BinaryClassificationEvaluator(metricName='areaUnderROC')
```

```
# Define a smaller parameter grid with fewer combinations
paramGrid = ParamGridBuilder() \
    .addGrid(lr.regParam, [0.1, 0.01]) \
    .addGrid(lr.elasticNetParam, [0.0, 0.5]) \
    .build()

# Create a CrossValidator with 3 folds
cv = CrossValidator(estimator=lr,
                    estimatorParamMaps=paramGrid,
                    evaluator=evaluator,
                    numFolds=3)

# Fit CrossValidator on sampled training data to find the best model
cv_model = cv.fit(sampled_trainingData)

# Show the average performance metrics over the folds
print("Average AUC-ROC across all folds: ", cv_model.avgMetrics)
```

Appendix E

Visualizations

Bar Graph (Figure I)

```
import matplotlib.pyplot as plt
from pyspark.sql.functions import col, floor
from pyspark.sql import SparkSession

# Assuming SparkSession is already created
spark = SparkSession.builder \
    .appName("RapSongsByDecadeVisualization") \
    .getOrCreate()

# Example modification for your dataset

# Calculate decade from year
transformed_sdf = transformed_sdf.withColumn("decade",
floor(col("year") / 10) * 10)

# Filter for rap songs
rap_songs_df = transformed_sdf.where(col("genre") == "rap")

# Group by decade and count the number of rap songs
rap_songs_by_decade =
rap_songs_df.groupby("decade").count().sort("decade").toPandas()

# Matplotlib visualization
fig = plt.figure(facecolor='white')
plt.bar(rap_songs_by_decade['decade'], rap_songs_by_decade['count'])
plt.xlabel("Decade")
plt.ylabel("Number of Rap Songs")
plt.title("Number of Rap Songs by Decade")
plt.xticks(rotation=45)
fig.tight_layout()
plt.savefig("rap_songs_by_decade_matplotlib.png")
plt.show()
```