CSP554—Big Data Technologies
Final Project Report

NAME: MANSI PATIL

MAILID: mpatil6@hawk.iit.edu

PROJECT TOPIC:

SONG/ARTIST RECOMMENDATION SYSTEM USING SPARK STREAMING

PROBLEM STATEMENT:

My aim is to develop a Song/Artist Recommendation System using Spark Streaming in order to improve the user experience. Users will find new artists and songs that suit their musical interests due to the system's evaluation of their listening history and personal recommendations

ABSTRACT:

The Song/Artist Recommendation System using Spark Streaming aims to enhance user experience on music streaming platforms by providing personalized recommendations in real time. By leveraging Apache Spark, PySpark, PySpark SQL, FindSpark, and MLlib, the system effectively ingests, processes, and analyzes user listening data to identify their preferences and generate tailored recommendations.

Development Summary:

Installation: pyspark, findspark, matplotlib-venn

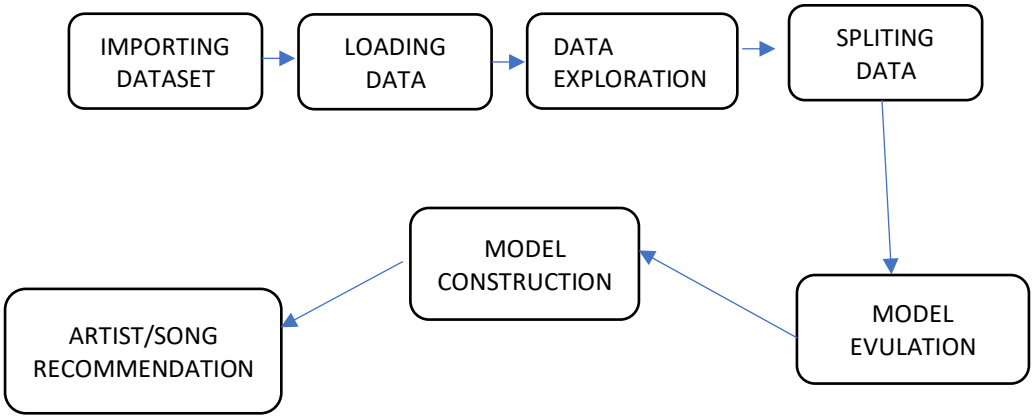
The project followed a structured approach to develop the Song/Artist Recommendation System using Spark Streaming:
Imported Dataset, Loading Data, Data Exploration, Data Splitting, Artist/Song Evaluation, Model Construction, Recommendation Generation.

Source:

GitHub Repository:

https://github.com/mansi2804/CSP554-BiGDATATECHNOLOGY_MANSIPATIL_FINALPROJECT

ARCHITECTURE:



Objectives:

The project aimed to achieve the following objectives:

- Enhanced User Experience: Provide personalized song and artist recommendations to improve user satisfaction and engagement.
- Current Trend Analysis: Identify and present the most popular songs, artists, and trends within the music community.
- Scalable and Efficient System: Develop a scalable and efficient system capable of handling large datasets.

Next Steps:

Future development will focus on:

- Recommendation Accuracy: Enhance recommendation accuracy by investigating alternative recommendation algorithms and incorporating diverse data sources.
- Continuous Monitoring: Continuously monitor the system's performance and user feedback to identify areas for improvement.

OVERVIEW:

Solution Outline:

The Song/Artist Recommendation System using Spark Streaming is designed to provide real-time, personalized music recommendations to users of music streaming platforms. The system utilizes the distributed data processing capabilities of Apache Spark Streaming to continuously ingest, preprocess, and analyze user listening data and preferences. By employing collaborative filtering algorithms, the system identifies similar users and recommends songs and artists based on their listening patterns.

Relevant Literature:

Existing literature underscores the significance of collaborative filtering algorithms in recommender systems, with studies like Ricci et al. (2015) emphasizing their prevalence and effectiveness. McFee et al. (2012) specifically address challenges and opportunities in music recommendation systems, highlighting the importance of incorporating audio processing and music feature recognition. The literature also discusses the efficiency of distributed stream processing frameworks such as Apache Spark (Zaharia et al., 2010) and the need for real-time data processing in dynamic recommender systems, as explored by Gedik et al. (2014).

Proposed System:

The proposed system integrates insights from the literature review, aiming to provide an interactive, scalable, and effective Song/Artist Recommendation System. PySpark is employed for efficient data manipulation, collaborative filtering algorithms from MLlib are utilized for accurate predictions, and Spark Streaming is integrated to enable real-time data processing. The system's architecture includes components for data preprocessing, model training, and recommendation generation. By incorporating music feature recognition and audio processing technologies, the proposed system endeavors to generate personalized playlists aligned with users' unique tastes.

The project's innovation lies in its ability to adapt to the evolving landscape of music consumption, offering a solution that combines traditional collaborative filtering approaches with real-time responsiveness through Spark Streaming.

Design:

The system employs PySpark and PySpark SQL for data manipulation and Spark for distributed stream processing. Collaborative filtering algorithms from MLlib are used to analyze user preferences and generate personalized recommendations. The technology stack includes Apache Spark, PySpark, PySpark SQL and findspark.

Software Components:

The Song/Artist Recommendation System using Spark Streaming utilizes a layered architecture comprising various software components:

- Data Ingestion Layer: Responsible for collecting real-time streaming data.
- Data Preprocessing Layer: Handles data cleaning, filtering, and transformation to prepare it for recommendation generation.
- Recommendation Engine Layer: Utilizes MLlib's collaborative filtering algorithms to generate personalized recommendations.

Interfaces:

The system defines interfaces between components to ensure seamless data flow and interaction:

- Data Ingestion Interface: Defines the format and protocol for data ingestion from streaming sources.
- Data Preprocessing Interface: Specifies the data transformation and preparation steps for recommendation generation.
- Recommendation Engine Interface: Defines the input data format and the expected output format for recommendations.

Datasets:

Some publicly available song data from audioscrobbler. However, I modified the original data files so that the code will run in a reasonable time on a single machine. The reduced data files have been suffixed with _small.txt and contains only the information relevant to the top 50 most prolific users (highest artist play counts).
The original data file user_artist_data.txt contained about 141,000 unique users, and 1.6 million unique artists. About 24.2 million users' plays of artists are recorded, along with their count.

Note that when plays are scribbled, the client application submits the name of the artist being played. This name could be misspelled or nonstandard, and this may only be detected later. For example, "The Smiths", "Smiths, The", and "the smiths" may appear as distinct artist IDs in the data set, even though they clearly refer to the same artist. So, the data set includes artist_alias.txt, which maps artist IDs that are known misspellings or variants to the canonical ID of that artist.
The artist_data.txt file then provides a map from the canonical artist ID to the name of the artist.

- user_artist_data.txt contains
  3 columns: userid
  artistid
  playcount

- artist_data.txt contains
  2 columns: artistid
  artist_name

- artist_alias.txt
  2 columns: badid, goodid

IMPLEMENTATION:

Necessary Package Imports:

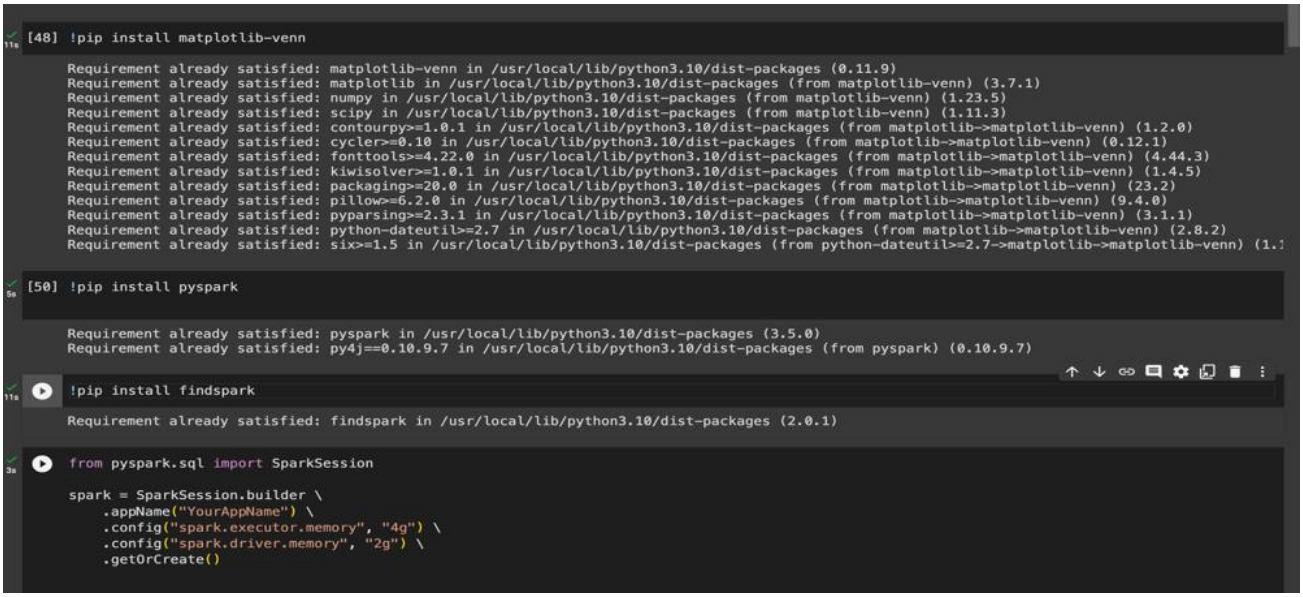
pyspark:

pyspark is the Python library for Apache Spark. It provides a Python API for interacting with Spark, enabling you to write Spark applications using Python. This library allows you to leverage the distributed processing capabilities of Spark while working in a Python environment.
With pyspark, we can create SparkContexts, work with Resilient Distributed Datasets (RDDs), and use Spark's SQL and machine learning libraries from Python.

findspark:

findspark is a Python library that allows you to easily locate the Spark installation on your system and makes it available in your Python environment. It is particularly useful when working in a local development environment or on a machine where Spark is not installed using spark-submit or other configurations.
We can use findspark to set the SPARK_HOME and PYTHONPATH environment variables in your Python script or Jupyter notebook, enabling to import and use PySpark without the need for additional configuration.

CODE AND OUTPUT:



Loading data:

Load the three datasets into RDDs and name them artistData, artistAlias, and userArtistData. View the README, or the files themselves, to see how this data is formated. Some of the files have tab delimeters while some have space delimiters. userArtistData RDD contains only the canonical artist IDs.

CODE:

```
# Import test files from location into RDD variables
artistData = spark.textFile('/content/artist_data_small.txt').map(lambda s:(int(s.split("\t")[0]),s.split("\t")[1]))
artistAlias = spark.textFile('/content/artist_alias_small.txt')
userArtistData = spark.textFile('/content/user_artist_data_small.txt')
```

Data Exploration:

Finding the three users with the highest number of total play counts (sum of all counters) and print the user ID, the total play count, and the mean play count (average number of times a user played an artist).

CODE:

```
# Assuming userArtistData is a tuple of three elements (userid, artistid, playcount)
# If it's different, adjust the code accordingly

# Mapping userArtistData
userArtistData = userArtistData.map(lambda s: (int(s[0]), int(s[1]), int(s[2])))

# Create a dictionary of the 'artistAlias' dataset
artistAliasDictionary = {}
dataValue = artistAlias.map(lambda s: (int(s.split("\t")[0]), int(s.split("\t")[1])))
for temp in dataValue.collect():
    artistAliasDictionary[temp[0]] = temp[1]

# If artistid exists, replace with artistsid from artistAlias, else retain original
userArtistData = userArtistData.map(lambda x: (x[0], artistAliasDictionary.get(x[1], x[1]), x[2]))

# Create an RDD consisting of 'userid' and 'playcount' objects of the original tuple
userSum = userArtistData.map(lambda x: (x[0], x[2]))

# Count instances by key and store in broadcast variable
playCount1 = userSum.reduceByKey(lambda a, b: a + b)
playCount2 = userSum.map(lambda x: (x[0], 1)).reduceByKey(lambda a, b: a + b)
playSumAndCount = playCount1.leftOuterJoin(playCount2)

# Compute and display users with the highest playcount along with their mean playcount across artists
playSumAndCount = playSumAndCount.map(lambda x: (x[0], x[1][0], int(x[1][0] / x[1][1] if x[1][1] is not None and x[1][1] != 0 else 0)))

# Display top three users
TopThree = playSumAndCount.top(3, key=lambda x: x[1])
for i in TopThree:
    print('User ' + str(i[0]) + ' has a total play count of ' + str(i[1]) + ' and a mean play count of ' + str(i[2]) + '.')
```

OUTPUT:

```
User 1059637 has a total play count of 674412 and a mean play count of 1878.
User 2064012 has a total play count of 548427 and a mean play count of 9455.
User 2069337 has a total play count of 393515 and a mean play count of 1519.
```

Splitting Data for Testing:

Used the randomsplit function to divide the data (userArtistData) into:
A training set, trainData, that will be used to train the model. This set should constitute 40% of the data.
A validation set, validationData, used to perform parameter tuning. This set should constitute 40% of the data.
A test set, testData, used for a final evaluation of the model. This set should constitute 20% of the data.
Use a random seed value of 13. Since these datasets will be repeatedly used, you will probably want to persist them in memory using the CACHE function.

CODE:

```
trainData, validationData, testData = userArtistData.randomSplit((0.4,0.4,0.2),seed=13)
trainData.cache()
validationData.cache()
testData.cache()

# Display the first 3 records of each dataset followed by the total count of records for each datasets
print(trainData.take(3))
print(validationData.take(3))
print(testData.take(3))
print(trainData.count())
print(validationData.count())
print(testData.count())
```

OUTPUT:

```
['1059637 1000049 1', '1059637 1000056 1', '1059637 1000114 2']
['1059637 1000010 238', '1059637 1000062 11', '1059637 1000123 2']
['1059637 1000094 1', '1059637 1000112 423', '1059637 1000113 5']
19761
19862
9858
```

Model Evaluation:

```
def modelEval(model, dataset):

    # All artists in the 'userArtistData' dataset

    AllArtists = spark.parallelize(set(userArtistData.map(lambda x:x[1]).collect()))

    # Set of all users in the current (Validation/Testing) dataset

    AllUsers = spark.parallelize(set(dataset.map(lambda x:x[0]).collect()))

    # Create a dictionary of (key, values) for current (Validation/Testing) dataset

    ValidationAndTestingDictionary ={}
    for temp in AllUsers.collect():
        tempFilter = dataset.filter(lambda x:x[0] == temp).collect()
        for item in tempFilter:
            if temp in ValidationAndTestingDictionary:
                ValidationAndTestingDictionary[temp].append(item[1])
            else:
                ValidationAndTestingDictionary[temp] = [item[1]]


    # Create a dictionary of (key, values) for training dataset

    TrainingDictionary = {}
    for temp in AllUsers.collect():
        tempFilter = trainData.filter(lambda x:x[0] == temp).collect()
        for item in tempFilter:
            if temp in TrainingDictionary:
                TrainingDictionary[temp].append(item[1])
            else:
                TrainingDictionary[temp] = [item[1]]


    # For each user, calculate the prediction score i.e. similarity between predicted and actual artists

    PredictionScore = 0.00
    for temp in AllUsers.collect():
        ArtistPrediction =  AllArtists.map(lambda x:(temp,x))
```

Model Construction:

Build the best model possibly using the validation set of data and the modeleval function. Although, there are a few parameters we could optimize, just try a few different values for the rank parameter (leave everything else at its default value, except make seed=345). Loop through the values [2, 10, 20] and figure out which one produces the highest scored based on model evaluation function.

CODE:

```
rankList = [2,10,20]
for rank in rankList:
    model = ALS.trainImplicit(trainData, rank , seed=345)
    modelEval(model,validationData)
```

OUPUT:

```
The model score for rank 2 is ~3.7677998998468976
The model score for rank 10 is ~4.882709016203582
The model score for rank 20 is ~4.333792013806091
```

Artist Recommendations:

Using the best model above, predict the top 5 artists for user 1059637 using the recommendproduct function. Map the results (integer IDs) into the real artist name using artistAlias.

CODE:

```
bestModel = ALS.trainImplicit(trainData, rank=10, seed=345)
modelEval(bestModel, testData)
```

OUTPUT:

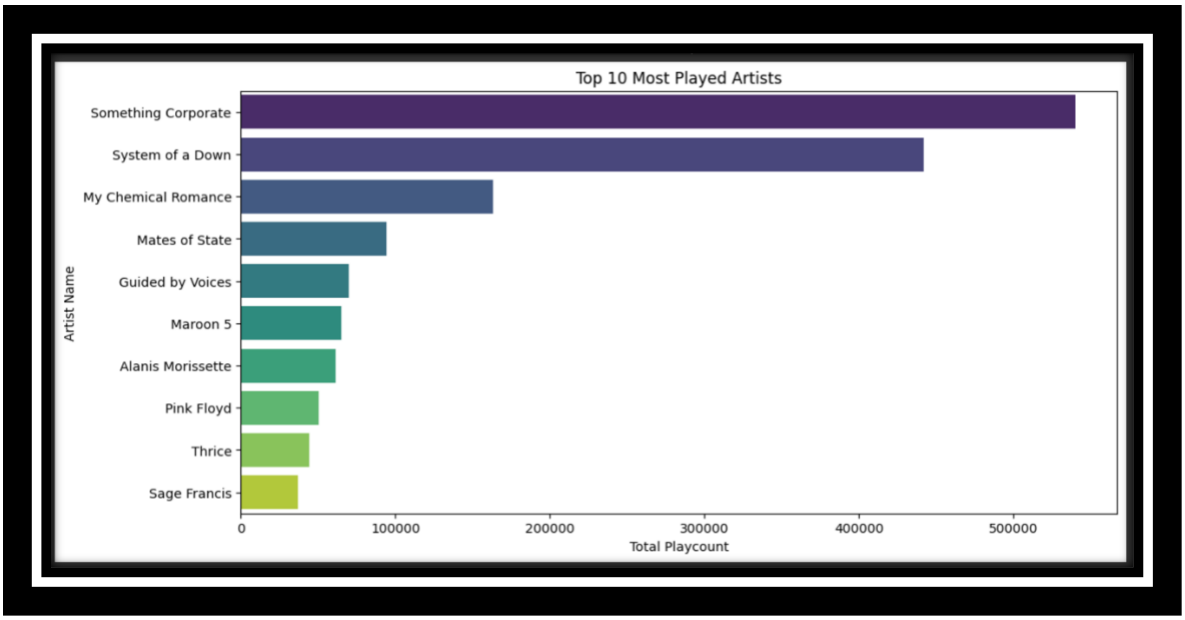
```
The model score for rank 10 is ~3.2229560504848322
```

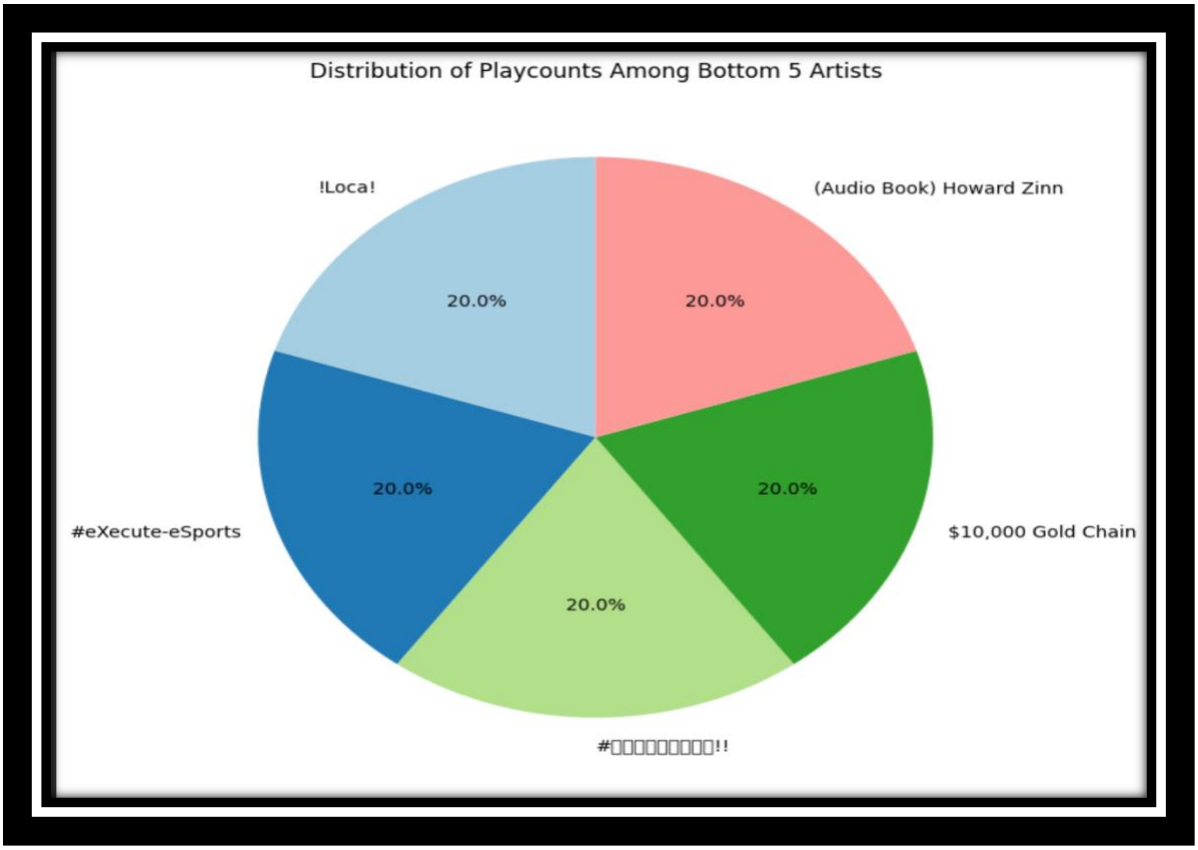
CODE AND OUTPUT:

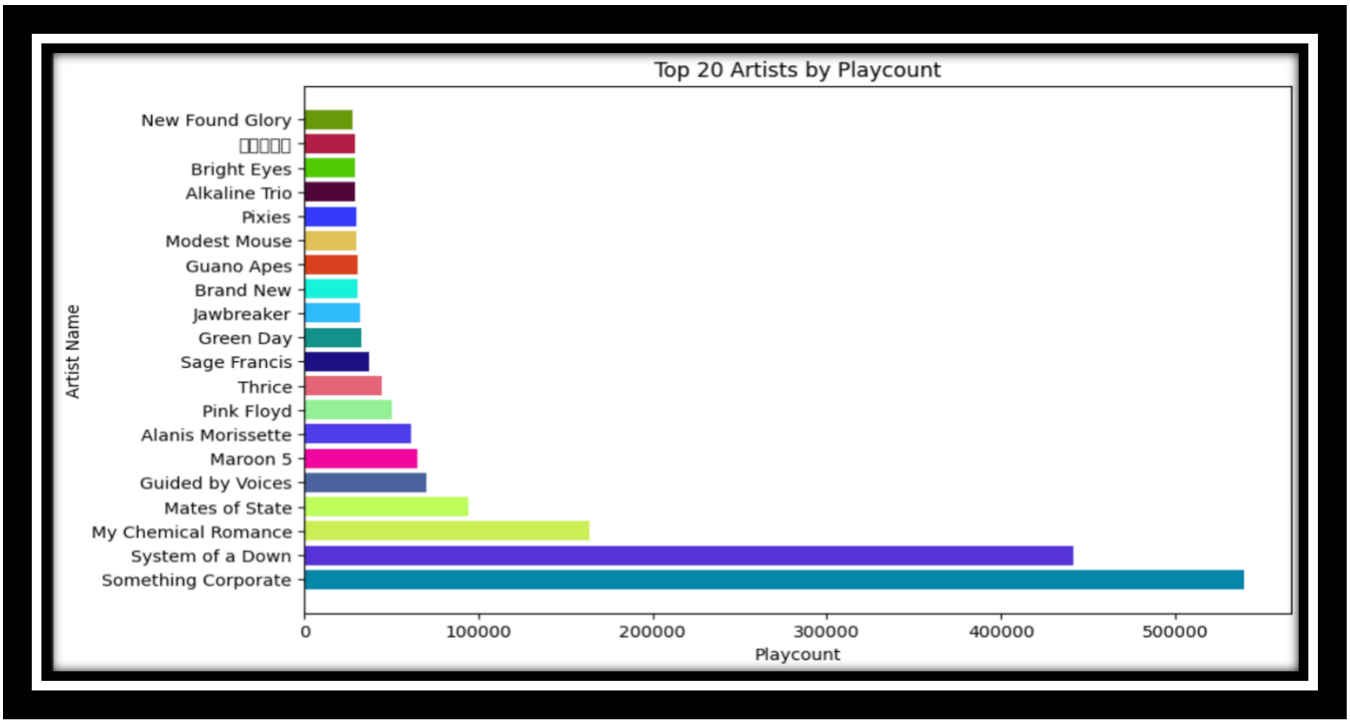
```
# Find the top 5 artists for a particular user and list their names
# YOUR CODE GOES HERE

TopFive = bestModel.recommendProducts(1059637,5)
for item in range(0,5):
    print("Artist "+str(item)+": "+artistData.filter(lambda x:x[0] == TopFive[item][1]).collect()[0][1])
```

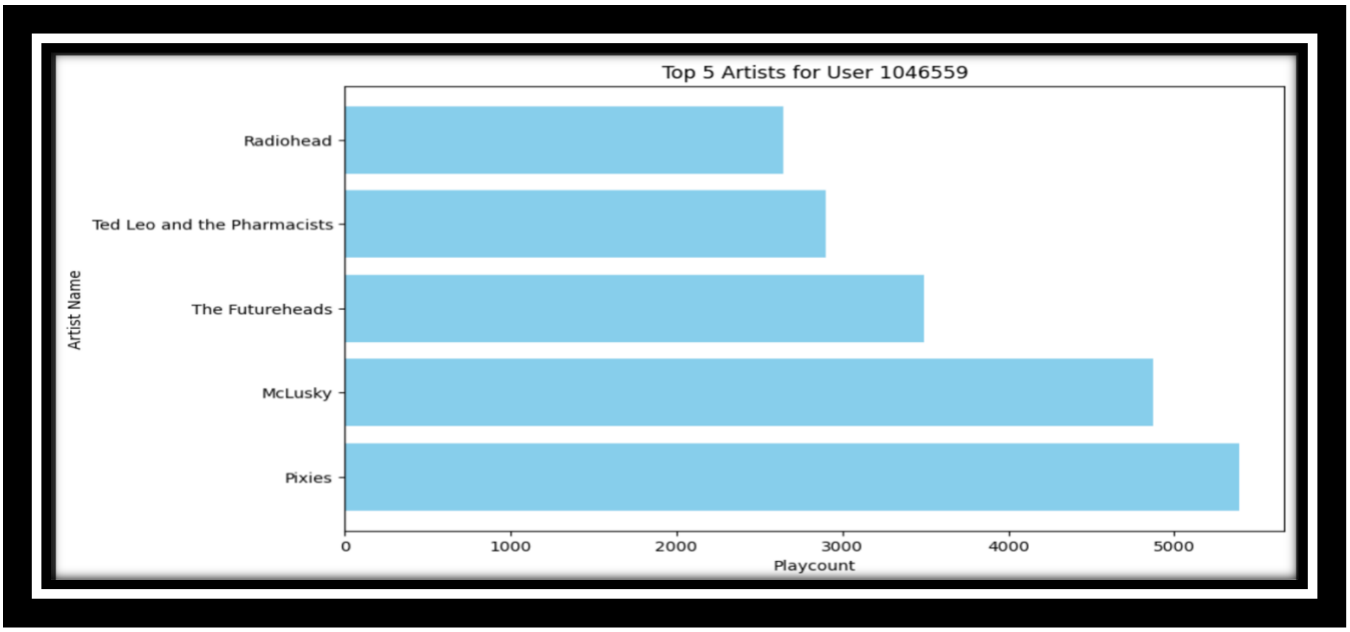
```
Artist 0: Something Corporate
Artist 1: My Chemical Romance
Artist 2: Rage Against the Machine
Artist 3: blink-182
Artist 4: Evanescence
```

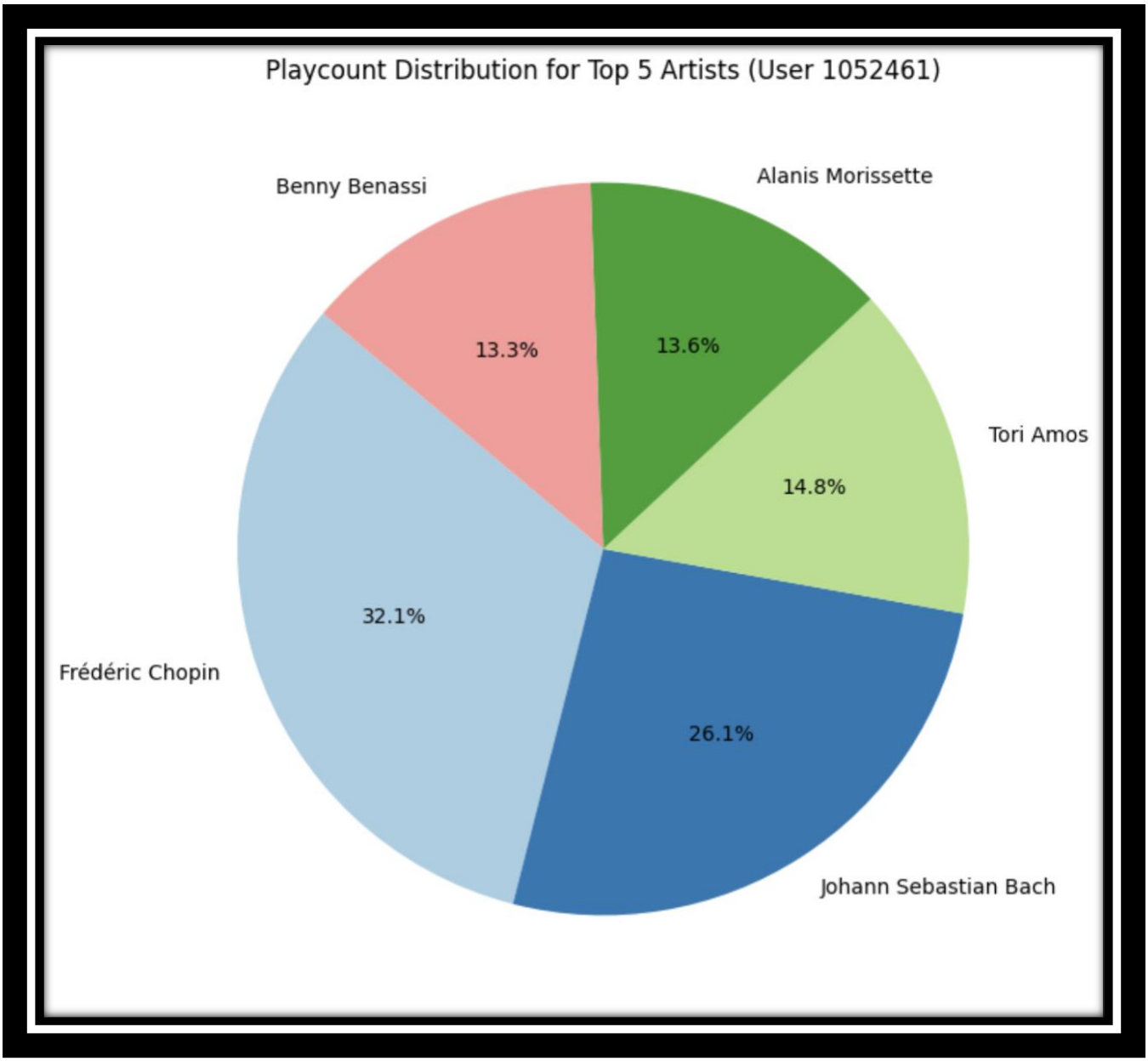
Visualization:


Top 10 Most Played Artists
Something Corporate
System of a Down
My Chemical Romance
Mates of State
Guided by Voices
Maroon 5
Alanis Morissette
Pink Floyd
Thrice
Sage Francis
Artist Name
0
100000
200000
300000
400000
500000
Total Playcount


Distribution of Playcounts Among Bottom 5 Artists
!Loca!
(Audio Book) Howard Zinn
20.0%
20.0%
20.0%
20.0%
20.0%
#eXecute-eSports
$10,000 Gold Chain
#□□□□□□□□□!!


Top 20 Artists by Playcount
New Found Glory
□□□□□
Bright Eyes
Alkaline Trio
Pixies
Modest Mouse
Guano Apes
Brand New
Jawbreaker
Green Day
Sage Francis
Thrice
Pink Floyd
Alanis Morissette
Maroon 5
Guided by Voices
Mates of State
My Chemical Romance
System of a Down
Something Corporate
Artist Name
0
100000
200000
300000
400000
500000
Playcount


Top 5 Artists for User 1046559
Radiohead
Ted Leo and the Pharmacists
The Futureheads
McLusky
Pixies
Artist Name
0
1000
2000
3000
4000
5000
Playcount


Playcount Distribution for Top 5 Artists (User 1052461)
Benny Benassi
Alanis Morissette
13.3%
13.6%
Tori Amos
14.8%
32.1%
Frédéric Chopin
26.1%
Johann Sebastian Bach

Conclusion:

The Song/Artist Recommendation System powered by Spark Streaming is a solid solution for improving user experience on music streaming services.
The system, which uses Apache Spark, PySpark, and MLlib, processes enormous amounts of data in real time, offering personalised song and artist suggestions based on individual listening histories. Its layered design enables fast data flow, and the addition of collaborative filtering algorithms enhances suggestion accuracy.
Beyond individual tastes, the initiative provides insights into current music trends, adding to a comprehensive user engagement strategy.
I recognizes the importance of scalability, emphasising the system's efficiency in dealing with large datasets, and outlines future steps, such as continuous monitoring and improved recommendation accuracy, demonstrating a forward-thinking approach to the dynamic field of recommendation systems.

The implementation of the system includes extensive data investigation, model creation, and assessment, exhibiting a strong mastery of machine learning techniques.
Matplotlib-Venn visualizations improve the presentation of insights, making the project more accessible and useful.

REFERENCE:

- https://www.analyticsvidhya.com/blog/2021/06/spotify-recommendation-system-using-pyspark-and-kafka-streaming/
- https://induraj2020.medium.com/recommendation-system-using-pyspark-kafka-and-spark-streaming-3eb36b3c3df0
- https://spark.apache.org/docs/1.5.2/api/python/pyspark.mllib.html#pyspark.mllib.recommendation.MatrixFactorizationModel.recommendProducts
- Adiyansjaha , Alexander A S Gunawana,, Derwin Suhartono –"Music Recommender System Based on Genre using Convolutional Recurrent Neural Networks"
- AnuPrabha P S, HarsithaN, Vaishnavi K, Dr.P.Velvadivu and Dr.M.Sujithra – "Music Recommender System Based on Collaborative Filtering"
- Anand Neil Arnold, Vairamuthu S. – "Music Recommendation using Collaborative Filtering and Deep Learning" ISSN: 2278-3075, Volume-8 Issue-7, May, 2019
- Ferdos Fessahaye, Luis Perez, Tiffany Zhan, Raymond Zhang, Calais Fossier, Robyn Markarian, Carter Chiu, Justin Zhan, Laxmi Gewali – "A Novel Music Recommendation System Using Deep Learning"