```
#!pip install pyspark
```

```
Collecting pyspark
  Downloading pyspark-3.5.1.tar.gz (317.0 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 317.0/317.0 MB 3.9 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: py4j==0.10.9.7 in /usr/local/lib/python3.10/dist-packages (from pyspark) (0.10.9.7)
Building wheels for collected packages: pyspark
  Building wheel for pyspark (setup.py) ... done
  Created wheel for pyspark: filename=pyspark-3.5.1-py2.py3-none-any.whl size=317488493 sha256=a8fad28035e732e4cd3556eaa6d4b
  Stored in directory: /root/.cache/pip/wheels/80/1d/60/2c256ed38dddce2fdd93be545214a63e02fbd8d74fb0b7f3a6
Successfully built pyspark
Installing collected packages: pyspark
Successfully installed pyspark-3.5.1
```

Mount Shared Drive

Mount a google shared drive in order to develop ETL for model developmentation

```
import pandas as pd
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

Create Spark Session

Create a spark session called spotify which will handle Spark functionalities.

```
from pyspark.sql import SparkSession

# Create Spark Enviroment
ss = SparkSession.builder.appName("spotify").getOrCreate()
```

Sample Playlist Data

The dataset filled with a million playlists consists of 33.54 GB of json data files. On a cluster of servers, it should not be an issue, however, we are limited to a pro version of Google Colab which cannot handle the large dataset. We, therefore, need to resample the dataset to obtain a smaller working size. The sample size chosen was 5 files consisting of the portion of the playlist within the dataset. This, however, is a more manageable data size for our computational power.

```
import os
import random
import re
import shutil

#============= Sample Data ================
def random_sample_from_folder(folder_path, sample_size=10):
    """
    Select a random sample of files from the specified folder.

    :param folder_path: Path to the folder from which to sample files.
    :param sample_size: Number of files to sample.
    :return: A list of paths to the randomly sampled files.
    """
    # List all files in the folder
    all_files = [os.path.join(folder_path, f) for f in os.listdir(folder_path) if os.path.isfile(os.path.join(folder_path, f))]

    # Determine the sample size (can't be larger than the number of files)
    actual_sample_size = min(sample_size, len(all_files))

    # Randomly sample files
    sampled_files = random.sample(all_files, actual_sample_size)

    return sampled_files

# Create Sample
folder_path = "/content/gdrive/Shareddrives/737_Project/spotify_million_playlist_dataset/data/"
sample_size = 5  # Adjust as needed
sampled_files = random_sample_from_folder(folder_path, sample_size)


#============= Abstract File Name ================
# Regular expression pattern to match the file name at the end of the path
pattern = r'/([^/]+\.json)$'

# Initialize a list to hold the matched file names
matched_file_names = []

# Iterate over the file paths and match the pattern
for file_path in sampled_files:
    match = re.search(pattern, file_path)
    if match:
        matched_file_names.append(match.group(1))  # Append the matched file name

# Print all matched file names
#for file_name in matched_file_names:
#    print(file_name)

#print(len(sampled_files))

#============= Move Data To Sampled Path ================
source_folder = '/content/gdrive/Shareddrives/737_Project/spotify_million_playlist_dataset/data/'
destination_folder = '/content/gdrive/Shareddrives/737_Project/spotify_million_playlist_dataset/sampled_data'

# Create New File
if not os.path.exists(destination_folder):
    os.makedirs(destination_folder)

# Move each file to the destination folder
for file in matched_file_names:
  # Construct the full file path
  source_file_path = os.path.join(source_folder, file)
  destination_file_path = os.path.join(destination_folder, file)

  # Move the file
  shutil.move(source_file_path, destination_file_path)
```

```
5
```

Load sampled data in to Pyspark

This loads our data files within a sampled folder and reads it to a spark data frame. It can handle json file as seen in the "View of sampled data files by track_uri" within the next block

```
# Folder Path of Playlist
folder_path = "/content/gdrive/Shareddrives/737_Project/spotify_million_playlist_dataset/sampled_data/*"

df_playlist = ss.read.option("multiline", "true").json(folder_path)
#df = ss.read.option("mergeSchema", "true").json(folder_path)
```

View of sampled data files by track_uri

Here we can see that the files have been properly loaded as we can witness pid and track_uri. Pid is the playlist id and the track_uri is the track identifier. This is for future implementations to build out song recommendations for playlists. The current implementation only has recommendations based on a single song.

```
from pyspark.sql.functions import explode, col
from pyspark.sql.functions import lit
tracks_df = df_playlist.select(explode("playlists").alias("playlist")) \
              .select("playlist.pid", explode("playlist.tracks").alias("track")) \
              .select("pid", "track.track_uri")

#tracks_df = tracks_df.withColumn("rating", lit(1))
tracks_df.show()
```

```
+------+--------------------+
|   pid|           track_uri|
+------+--------------------+
|150000|spotify:track:5CB...|
|150000|spotify:track:4Sn...|
|150000|spotify:track:50Y...|
|150000|spotify:track:636...|
|150000|spotify:track:2ay...|
|150000|spotify:track:5IM...|
|150000|spotify:track:5C9...|
|150000|spotify:track:4tD...|
|150000|spotify:track:15W...|
|150000|spotify:track:2kD...|
|150000|spotify:track:3yt...|
|150000|spotify:track:50K...|
|150000|spotify:track:0TY...|
|150000|spotify:track:0fy...|
|150000|spotify:track:2bh...|
|150001|spotify:track:76V...|
|150001|spotify:track:6eF...|
|150001|spotify:track:6Jn...|
|150001|spotify:track:0CK...|
|150001|spotify:track:7dC...|
+------+--------------------+
only showing top 20 rows
```

Recommend Users who like a track to an Artist

Read Music Data

Load the data into spark and clean out columns that won't be need in model developmentation.

```
from pyspark.sql.types import StructType, StructField, IntegerType, StringType, FloatType, BooleanType

# Load dataset
spotify_song_df = ss.read.csv("/content/gdrive/Shareddrives/737_Project/spotify_million_playlist_dataset/spotify_data.csv", heac
col_drop = ['_c0','key', 'duration_ms', 'time_signature'] #['Unnamed: 0','artist_name', 'track_name', 'key', 'duration_ms', 'tim
spotify_song_df = spotify_song_df.drop(*col_drop)
spotify_song_df.printSchema()
```

```
root
 |-- track_id: string (nullable = true)
 |-- artists: string (nullable = true)
 |-- album_name: string (nullable = true)
 |-- track_name: string (nullable = true)
 |-- popularity: string (nullable = true)
```

```
 |-- explicit: string (nullable = true)
 |-- danceability: string (nullable = true)
 |-- energy: string (nullable = true)
 |-- loudness: string (nullable = true)
 |-- mode: string (nullable = true)
 |-- speechiness: string (nullable = true)
 |-- acousticness: string (nullable = true)
 |-- instrumentalness: double (nullable = true)
 |-- liveness: string (nullable = true)
 |-- valence: string (nullable = true)
 |-- tempo: double (nullable = true)
 |-- track_genre: string (nullable = true)
```

```
spotify_song_df.show(truncate = False)
```

```
---------------+------------------------------------+------------------------------------------------------+----------------
.d              |artists                             |album_name                                            |track_name
---------------+------------------------------------+------------------------------------------------------+----------------
iRyPMVoIQDJUgSV|Gen Hoshino                         |Comedy                                                |Comedy
1i3p13qLCt0Ki3A|Ben Woodward                        |Ghost (Acoustic)                                      |Ghost - Acoustic
s7jYXzM8EGcbK5b|Ingrid Michaelson;ZAYN              |To Begin Again                                        |To Begin Again
G4xtTiEg7opyCyx|Kina Grannis                        |Crazy Rich Asians (Original Motion Picture Soundtrack)|Can't Help Falli
imiIP26QG5WcN2K|Chord Overstreet                    |Hold On                                               |Hold On
KtVTNfFiBU9I7dc|Tyrone Wells                        |Days I Will Remember                                  |Days I Will Remer
mXdKIAM7WUoEb7N|A Great Big World;Christina Aguilera|Is There Anybody Out There?                           |Say Something
mMH3G43AXT1y7pA|Jason Mraz                          |We Sing. We Dance. We Steal Things.                   |I'm Yours
nAGrvD03AWnz3Q8|Jason Mraz;Colbie Caillat           |We Sing. We Dance. We Steal Things.                   |Lucky
Lp2AzqokyEdwEw2|Ross Copperman                      |Hunger                                                |Hunger
kRvGxdhdGdAH7EJ|Zack Tabudlo                        |Episode                                               |Give Me Your For
BqJiVL5IAE9jRyl|Jason Mraz                          |Love Is a Four Letter Word                            |I Won't Give Up
l35d7gQfeNteBwp|Dan Berk                            |Solo                                                  |Solo
1rTkEHDjp95F2OO|Anna Hamilton                       |Bad Liar                                              |Bad Liar
N82ZRhz9jqzgrb3|Chord Overstreet;Deepend            |Hold On (Remix)                                       |Hold On - Remix
K9QxnGjdXb55NiG|Landon Pigg                         |The Boy Who Never                                     |Falling in Love
fjixSUld14qUezm|Andrew Foy;Renee Foy                |ily (i love you baby)                                 |ily (i love you
coNyat1secaH0OD|Andrew Foy;Renee Foy                |At My Worst                                           |At My Worst
uEC3ruGJg4SMMN6|Jason Mraz;Colbie Caillat           |We Sing. We Dance. We Steal Things.                   |Lucky
FJ4Q4Id4EjtbXlC|Boyce Avenue;Bea Miller             |Cover Sessions, Vol. 4                                |Photograph
---------------+------------------------------------+------------------------------------------------------+----------------
wing top 20 rows
```

ALS Model For Artist Recommendation Based on Popularity

The ALS model develops a recommendation system for a user who likes a song to explore other artists that are of the same popularity. The model takes into consideration the track_id, the artist_id, and popularity column to offer the top 10 recommended artist for a user who likes a song.

```
from pyspark.ml.feature import StringIndexer, IndexToString, OneHotEncoder, VectorAssembler
from pyspark.ml import Pipeline
from pyspark.ml.recommendation import ALS
from pyspark.ml.evaluation import RegressionEvaluator

# Convert columns to data types
df = spotify_song_df \
    .withColumn('popularity', spotify_song_df['popularity'].cast('int')) \
    .withColumn('explicit', spotify_song_df['explicit'].cast('boolean')) \
    .withColumn('danceability', spotify_song_df['danceability'].cast('double')) \
    .withColumn('energy', spotify_song_df['energy'].cast('double')) \
    .withColumn('loudness', spotify_song_df['loudness'].cast('double')) \
    .withColumn('mode', spotify_song_df['mode'].cast('int')) \
    .withColumn('speechiness', spotify_song_df['speechiness'].cast('double')) \
    .withColumn('acousticness', spotify_song_df['acousticness'].cast('double')) \
    .withColumn('instrumentalness', spotify_song_df['instrumentalness'].cast('double')) \
    .withColumn('liveness', spotify_song_df['liveness'].cast('double')) \
    .withColumn('valence', spotify_song_df['valence'].cast('double')) \
    .withColumn('tempo', spotify_song_df['tempo'].cast('double'))

# Drop rows with NaN values in any column
spotify_song_df = spotify_song_df.dropna(how='any')

# StringIndexer to convert string fields to indices which will be used by ALS
indexers = [StringIndexer(inputCol=column, outputCol=column+"_index").fit(df) for column in list(set(df.columns)-set(['popularit

# Index the genre column
genre_indexer = StringIndexer(inputCol='track_genre', outputCol='track_genre_index')#.fit(df)

# One-hot encode the indexed genre column
genre_encoder = OneHotEncoder(inputCol='track_genre_index', outputCol='track_genre_vec')

assembler=VectorAssembler(inputCols=['track_id', 'artists', 'album_name', 'track_name', 'popularity',\
                                     'explicit', 'danceability', 'energy', 'loudness', 'mode', 'speechiness',\
                                     'acousticness', 'instrumentalness', 'liveness', 'valence', 'track_genre_vec'], outputCol='f

# Define the ALS algorithm
als = ALS(
    userCol="track_id_index",
    itemCol="artists_index",
    ratingCol="popularity",
    nonnegative=True,
    implicitPrefs=False,
    coldStartStrategy="drop"
)

# Now, add these to your pipeline
pipeline_stages = indexers + [genre_indexer, genre_encoder, als]
pipeline = Pipeline(stages=pipeline_stages)

# Split the data into training and test sets
(training_data, test_data) = df.randomSplit([0.8, 0.2])

# Fit the model
model = pipeline.fit(training_data)

# Make predictions
predictions = model.transform(test_data)

# Evaluate the model by computing the RMSE on the test data
evaluator = RegressionEvaluator(metricName="rmse", labelCol="popularity", predictionCol="prediction")

rmse = evaluator.evaluate(predictions)

print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

```
Root Mean Squared Error (RMSE) on test data = 2.04544
```

This portion recommends a user 10 artists based on songs they like.

```
# Get 10 artist to recommendation for a track
user_recs = model.stages[-1].recommendForAllUsers(10)

user_recs.show(truncate=False)
```

```
+--------------+-----------------------------------------------------------------------------------------------------------
|track_id_index|recommendations
+--------------+-----------------------------------------------------------------------------------------------------------
|26            |[{29461, 136.20064}, {17011, 114.20156}, {18244, 113.6354}, {24863, 112.77322}, {12008, 106.92717}, {7920, 1
|27            |[{99, 0.0}, {98, 0.0}, {97, 0.0}, {96, 0.0}, {95, 0.0}, {94, 0.0}, {93, 0.0}, {92, 0.0}, {91, 0.0}, {90, 0.0
|28            |[{9134, 100.34088}, {10074, 98.84116}, {27974, 97.190216}, {30972, 95.44558}, {27411, 93.40673}, {23027, 90.
|31            |[{5053, 108.15918}, {3788, 104.418076}, {25341, 101.096504}, {9634, 100.35293}, {4003, 98.2284}, {5735, 96.4
|34            |[{5053, 117.77333}, {3788, 113.69968}, {25341, 110.08286}, {9634, 109.273186}, {4003, 106.959816}, {5735, 10
|44            |[{99, 0.0}, {98, 0.0}, {97, 0.0}, {96, 0.0}, {95, 0.0}, {94, 0.0}, {93, 0.0}, {92, 0.0}, {91, 0.0}, {90, 0.0
|53            |[{99, 0.0}, {98, 0.0}, {97, 0.0}, {96, 0.0}, {95, 0.0}, {94, 0.0}, {93, 0.0}, {92, 0.0}, {91, 0.0}, {90, 0.0
|65            |[{23088, 14.280181}, {5052, 13.922445}, {30797, 13.549765}, {23716, 13.401429}, {29804, 13.358647}, {29461,
|76            |[{13329, 78.078186}, {27316, 77.44498}, {10676, 75.35704}, {3865, 72.996315}, {6580, 71.56297}, {5771, 71.50
|78            |[{16410, 74.35415}, {24864, 66.8762}, {8580, 66.67165}, {7307, 64.094955}, {17388, 62.098248}, {30970, 61.52
|81            |[{29461, 164.05984}, {17011, 137.56097}, {18244, 136.879}, {24863, 135.84045}, {12008, 128.79865}, {7920, 12
|85            |[{99, 0.0}, {98, 0.0}, {97, 0.0}, {96, 0.0}, {95, 0.0}, {94, 0.0}, {93, 0.0}, {92, 0.0}, {91, 0.0}, {90, 0.0
|101           |[{6234, 94.50512}, {16366, 87.20385}, {16001, 84.42688}, {19243, 83.72504}, {16280, 83.58065}, {1561, 83.436
|103           |[{99, 0.0}, {98, 0.0}, {97, 0.0}, {96, 0.0}, {95, 0.0}, {94, 0.0}, {93, 0.0}, {92, 0.0}, {91, 0.0}, {90, 0.0
|108           |[{99, 0.0}, {98, 0.0}, {97, 0.0}, {96, 0.0}, {95, 0.0}, {94, 0.0}, {93, 0.0}, {92, 0.0}, {91, 0.0}, {90, 0.0
|115           |[{99, 0.0}, {98, 0.0}, {97, 0.0}, {96, 0.0}, {95, 0.0}, {94, 0.0}, {93, 0.0}, {92, 0.0}, {91, 0.0}, {90, 0.0
|126           |[{99, 0.0}, {98, 0.0}, {97, 0.0}, {96, 0.0}, {95, 0.0}, {94, 0.0}, {93, 0.0}, {92, 0.0}, {91, 0.0}, {90, 0.0
|133           |[{99, 0.0}, {98, 0.0}, {97, 0.0}, {96, 0.0}, {95, 0.0}, {94, 0.0}, {93, 0.0}, {92, 0.0}, {91, 0.0}, {90, 0.0
|137           |[{29461, 164.05984}, {17011, 137.56097}, {18244, 136.879}, {24863, 135.84045}, {12008, 128.79865}, {7920, 12
|148           |[{99, 0.0}, {98, 0.0}, {97, 0.0}, {96, 0.0}, {95, 0.0}, {94, 0.0}, {93, 0.0}, {92, 0.0}, {91, 0.0}, {90, 0.0
+--------------+-----------------------------------------------------------------------------------------------------------
only showing top 20 rows
```

Song Recomender

The song recommender recommends songs to a user based on a song that they like. The model is built using cosine similarity which takes into account variables on aspects of the song to recommend similar ones. The model takes a song name and recommends 10 other songs that the user might like.

```
from pyspark.ml.feature import VectorAssembler, StringIndexer, OneHotEncoder, Normalizer
from pyspark.ml import Pipeline

# Drop NaNs
spotify_song_df = spotify_song_df.dropna(how='any')

# Convert columns to data types
spotify_song_df = spotify_song_df \
    .withColumn('popularity', spotify_song_df['popularity'].cast('int')) \
    .withColumn('explicit', spotify_song_df['explicit'].cast('boolean')) \
    .withColumn('danceability', spotify_song_df['danceability'].cast('double')) \
    .withColumn('energy', spotify_song_df['energy'].cast('double')) \
    .withColumn('loudness', spotify_song_df['loudness'].cast('double')) \
    .withColumn('mode', spotify_song_df['mode'].cast('int')) \
    .withColumn('speechiness', spotify_song_df['speechiness'].cast('double')) \
    .withColumn('acousticness', spotify_song_df['acousticness'].cast('double')) \
    .withColumn('instrumentalness', spotify_song_df['instrumentalness'].cast('double')) \
    .withColumn('liveness', spotify_song_df['liveness'].cast('double')) \
    .withColumn('valence', spotify_song_df['valence'].cast('double')) \
    .withColumn('tempo', spotify_song_df['tempo'].cast('double'))

# Example of handling categorical variables
features = ['popularity', 'danceability', 'energy', 'loudness', 'mode', 'speechiness',
            'acousticness', 'instrumentalness', 'liveness', 'valence', 'tempo']

assembler_audio = VectorAssembler(inputCols=features, outputCol="audio_vec")

normalizer = Normalizer(inputCol="audio_vec", outputCol="normFeatures")

stringIndexer = StringIndexer(inputCol="track_genre", outputCol="genreIndexed")
encoder = OneHotEncoder(inputCols=["genreIndexed"], outputCols=["genreVec"])

# Assuming other features are numerical and already in the desired format
assembler = VectorAssembler(inputCols=["normFeatures", "genreVec"], outputCol="features")

pipeline = Pipeline(stages=[assembler_audio, normalizer, stringIndexer, encoder, assembler])
pipelineModel = pipeline.fit(spotify_song_df)
df_transformed = pipelineModel.transform(spotify_song_df)
```

```
df_transformed.show(truncate=False)
```

```
--------------------------------------------------+------------+----------------+-----------------------------------------
                                                  |genreIndexed|genreVec        |features
--------------------------------------------------+------------+----------------+-----------------------------------------
5245735512779771,0.7679808798280549]              |97.0        |(113,[97],[1.0])|(124,[0,1,2,3,5,6,7,8,9,10,108],[0.637676
5931002454,0.002764393570075794,0.8022849938262292]  |97.0     |(113,[97],[1.0])|(124,[0,1,2,3,4,5,6,7,8,9,10,108],[0.5694
53013726232395,0.7970420312564265]                |97.0        |(113,[97],[1.0])|(124,[0,1,2,3,4,5,6,8,9,10,108],[0.595181
039925472E-4,7.295997626585927E-4,0.9272549710879207]|97.0     |(113,[97],[1.0])|(124,[0,1,2,3,4,5,6,7,8,9,10,108],[0.3622
52746553164,0.8236709262653021]                   |97.0        |(113,[97],[1.0])|(124,[0,1,2,3,4,5,6,8,9,10,108],[0.563081
5829743383356447,0.8579789147243977]              |97.0        |(113,[97],[1.0])|(124,[0,1,2,3,4,5,6,8,9,10,108],[0.507695
368905538E-4,4.789025993661266E-4,0.8844611091352135]|97.0     |(113,[97],[1.0])|(124,[0,1,2,3,4,5,6,7,8,9,10,108],[0.4632
04161074576565257,0.88224131752569]               |97.0        |(113,[97],[1.0])|(124,[0,1,2,3,4,5,6,8,9,10,108],[0.467536
00446231128999991,0.8677027669559167]             |97.0        |(113,[97],[1.0])|(124,[0,1,2,3,4,5,6,8,9,10,108],[0.493588
32554E-4,0.0020206501186201442,0.8134044576990345]  |97.0      |(113,[97],[1.0])|(124,[0,1,2,3,4,5,6,7,8,9,10,108],[0.5773
5761461562928,0.8018161090280541]                 |97.0        |(113,[97],[1.0])|(124,[0,1,2,3,4,5,6,8,9,10,108],[0.593908
55783673381E-4,0.8862050960264252]                |97.0        |(113,[97],[1.0])|(124,[0,1,2,3,4,5,6,8,9,10,108],[0.458361
0.920255946618441]                                |97.0        |(113,[97],[1.0])|(124,[0,1,2,3,5,6,8,9,10,108],[0.38518689
001951089737383 6735,0.813137652609235]           |97.0        |(113,[97],[1.0])|(124,[0,1,2,3,4,5,6,8,9,10,108],[0.578792
540146862E-4,0.002919085477163931,0.9051729550428433]|97.0     |(113,[97],[1.0])|(124,[0,1,2,3,4,5,6,7,8,9,10,108],[0.4223
416635567,0.002334490699213771,0.8186117238835449]  |97.0      |(113,[97],[1.0])|(124,[0,1,2,3,4,5,6,7,8,9,10,108],[0.5689
55818911745,0.0033146713992235695,0.8819427857731233]|97.0     |(113,[97],[1.0])|(124,[0,1,2,3,4,5,6,7,8,9,10,108],[0.4483
05636809091615388,0.8497142611454308]             |97.0        |(113,[97],[1.0])|(124,[0,1,2,3,5,6,7,8,9,10,108],[0.499815
549368256869812,0.8846311177872647]               |97.0        |(113,[97],[1.0])|(124,[0,1,2,3,4,5,6,8,9,10,108],[0.462417
25287626764338403,0.8477323474233768]             |97.0        |(113,[97],[1.0])|(124,[0,1,2,3,4,5,6,8,9,10,108],[0.526171
--------------------------------------------------+------------+----------------+-----------------------------------------
```

Data cleaning process to convert a sparse vector to an array and then to a list. Needed to obtain a cosine similarity score as we will need to take the dot product of the vectors.

```
from pyspark.sql.types import ArrayType, DoubleType
from pyspark.sql.functions import udf

# UDF to convert a sparse vector to a dense vector
def sparse_to_dense(sparse_vector):
    return sparse_vector.toArray().tolist()

# Register the UDF with the appropriate return type
sparse_to_array_udf = udf(sparse_to_dense, ArrayType(DoubleType()))

# Apply the UDF to the normalized feature column
df_transformed = df_transformed.withColumn("features", sparse_to_array_udf("features"))


df_transformed.show(truncate=False)
```

```
------------------------------------------------+------------+----------------+-----------------------------------------
                                                |genreIndexed|genreVec        |features
------------------------------------------------+------------+----------------+-----------------------------------------
45735512779771,0.7679808798280549]              |97.0        |(113,[97],[1.0])|[0.6376764929131795, 0.005905059030264512,
31002454,0.002764393570075794,0.8022849938262292]  |97.0     |(113,[97],[1.0])|[0.5694443683676729, 0.004348484267534956,
013726232395,0.7970420312564265]                |97.0        |(113,[97],[1.0])|[0.5951815199603877, 0.004573500100748242,
09925472E-4,7.295997626585927E-4,0.9272549710879207]|97.0    |(113,[97],[1.0])|[0.3622488332081125, 0.00135715760046983, 3
746553164,0.8236709262653021]                   |97.0        |(113,[97],[1.0])|[0.5630811090859847, 0.00424370884652608, 0
29743383356447,0.8579789147243977]              |97.0        |(113,[97],[1.0])|[0.5076953697217327, 0.0060223174891129655,
8905538E-4,4.789025993661266E-4,0.8844611091352135]|97.0     |(113,[97],[1.0])|[0.46325218762213555, 0.002547887031921745,
161074576565257,0.88224131752569]               |97.0        |(113,[97],[1.0])|[0.46753646927699516, 0.0041084767237715945
446231128999991,0.8677027669559167]             |97.0        |(113,[97],[1.0])|[0.49358899171897364, 0.004168825943572412,
554E-4,0.0020206501186201442,0.8134044576990345]  |97.0      |(113,[97],[1.0])|[0.5773286053200412, 0.004556772206276039,
51461562928,0.8018161090280541]                 |97.0        |(113,[97],[1.0])|[0.5939081334074972, 0.005032167562790551,
783673381E-4,0.8862050960264252]                |97.0        |(113,[97],[1.0])|[0.45836133026867854, 0.00320852931188075,
920255946618441]                                |97.0        |(113,[97],[1.0])|[0.38518689911102383, 0.0036222383397171275
19510897373836735,0.813137652609235]            |97.0        |(113,[97],[1.0])|[0.5787921708985061, 0.006450732098239801,
0146862E-4,0.002919085477163931,0.9051729550428433]|97.0     |(113,[97],[1.0])|[0.4223999656878039, 0.005694856680255214,
6635567,0.002334490699213771,0.8186117238835449]  |97.0      |(113,[97],[1.0])|[0.5689094981277257, 0.004796495596283756,
318911745,0.0033146713992235695,0.8819427857731233]|97.0     |(113,[97],[1.0])|[0.44836134868724614, 0.0056525555745213524
536809091615388,0.8497142611454308]             |97.0        |(113,[97],[1.0])|[0.49981558447821167, 0.0073583961048181165
9368256869812,0.8846311177872647]               |97.0        |(113,[97],[1.0])|[0.462417102342522, 0.00425015719064818, 0.
287626764338403,0.8477323474233768]             |97.0        |(113,[97],[1.0])|[0.5261711159039357, 0.005630816270195849 5,
------------------------------------------------+------------+----------------+-----------------------------------------
```

Cosine similarity score to determine the similarity between two songs.

```
from pyspark.sql.functions import udf
from pyspark.ml.linalg import Vectors, VectorUDT
from pyspark.sql.types import DoubleType
import numpy as np

# UDF to calculate the dot product of two vectors
def cosine_similarity(list_a, list_b):
    # Convert lists to numpy arrays
    np_a = np.array(list_a)
    np_b = np.array(list_b)
    # Calculate dot product using numpy
    #return float(np.dot(np_a, np_b)) #/ (np.linalg.norm(np_a) * np.linalg.norm(np_b))

    # Calculate dot product of features
    dot_product = np.dot(np_a, np_b)
    dot_product = float(dot_product)

    # Calculate the normalization of list a
    norm_vec1 = np.linalg.norm(list_a)
    norm_vec1 = float(norm_vec1)

    # Calculate the normalization of list b
    norm_vec2 = np.linalg.norm(list_b)
    norm_vec2 = float(norm_vec2)

    # Calculate the cosine similarity of all songs
    cosine_similarity = dot_product / (norm_vec1 * norm_vec2) if norm_vec1 != 0 and norm_vec2 != 0 else 0.0
    return cosine_similarity

cosine_similarity_udf = udf(cosine_similarity, DoubleType())
```

Recommend the top 10 songs based on a song track.

```
from pyspark.sql.functions import col, lit
import pyspark.sql.functions as F

# Assuming `song_id` is a unique identifier for each song
def recommend_songs(song_id, df):
    # Filter out the target song
    target_song = df.filter(df.track_id == song_id).select("features").collect()[0][0]
    #return target_song

    # Add a column for the target song's features
    df_with_similarity = df.withColumn("targetFeatures", F.array([F.lit(x) for x in target_song]))
    #df_with_similarity = df.withColumn("targetFeatures", lit(target_song))

    # Calculate cosine similarity between target song and all songs
    df_with_similarity = df_with_similarity.withColumn("similarity", cosine_similarity_udf("features", "targetFeatures"))

    # Sort by similarity and fetch top 10 songs
    df_filtered = df_with_similarity.filter(df_with_similarity.track_id != song_id)
    top_songs = df_filtered.sort(col("similarity").desc()).limit(10)

    return top_songs

# Example usage
song_id = '6DCZcSspjsKoFjzjrWoCdn'
top_recommendations = recommend_songs(song_id, df_transformed)
top_recommendations
top_recommendations.show(truncate=False)
```

```
-----------------------------------------------------------------------------------------------------------------------------

-----------------------------------------------------------------------------------------------------------------------------
).0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
).0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
).0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
).0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
).0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
).0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
).0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
).0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
).0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
).0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
-----------------------------------------------------------------------------------------------------------------------------
```

```
df_transformed.filter(df_transformed.artists == 'Drake').show(truncate=False)
```

```
+----------------------+-------+----------+----------+----------+--------+------------+------+--------+----+-----------+----
|track_id              |artists|album_name|track_name|popularity|explicit|danceability|energy|loudness|mode|speechiness|acou
+----------------------+-------+----------+----------+----------+--------+------------+------+--------+----+-----------+----
|6DCZcSspjsKoFjzjrWoCdn|Drake  |Scorpion  |God's Plan|84        |true    |0.754       |0.449 |-9.211  |1   |0.109      |0.03
+----------------------+-------+----------+----------+----------+--------+------------+------+--------+----+-----------+----
```

Find similar songs

The function takes in a query and returns the data we want about a song to search for.

```
def search_songs_in_df(query):
    # Filter DataFrame where song_name contains the query string (case-insensitive)
    filtered_df = df_transformed.filter(df_transformed.track_name.contains(query)).select("track_id", "track_name", 'artists', '
    return filtered_df
```

```
import ipywidgets as widgets
from IPython.display import display, clear_output

# Input text and button for the search
text = widgets.Text(value='', placeholder='Type a song name', description='Search:', disabled=False)
button = widgets.Button(description="Search")

# Dropdown for selecting a song from search results
dropdown = widgets.Dropdown(options=['Select a song'], description='Results:', disabled=False)

# Output widget for displaying the track ID
output = widgets.Output()

display(text, button, dropdown, output)


def on_button_clicked(b):
    with output:
        clear_output()
        query = text.value
        if query:
            results_df = search_songs_in_df(query).collect()
            # Update dropdown options with search results
            dropdown.options = [(row.track_name, row.artists, row.album_name, row.track_id) for row in results_df]
            dropdown.disabled = False if results_df else True
        else:
            print("Please enter a query.")

def on_dropdown_change(change):
    with output:
        if change['name'] == 'value' and change['new']:
            clear_output()
            print(f"Track Name: {change['new'][0]}")
            print(f"Artist: {change['new'][1]}")
            print(f"Album: {change['new'][2]}")
            print(f"Track ID: {change['new'][3]}")

            song_id = change['new'][3]
            top_recommendations = recommend_songs(song_id, df_transformed)
            top_recommendations.select("track_name", "artists", "album_name").show(truncate=False)
```