Content-based Music recommendation system using Neo4j

(A step by step guide of the project)

A little background on the dataset(original raw dataset Spotify):

#	Column	Non-Null Count	Dtype
0	genre	232725	object
1	artist_name	232725	object
2	track_name	232725	object
3	track_id	232725	object
4	popularity	232725	int64
5	acousticness	232725	float64
6	danceability	232725	float64
7	duration_ms	232725	int64
8	energy	232725	float64
9	instrumentalness	232725	float64
10	key	232725	object
11	liveness	232725	float64
12	loudness	232725	float64
13	mode	232725	object
14	speechiness	232725	float64
15	tempo	232725	float64
16	time_signature	232725	object
17	valence	232725	float64

- Genre: The category or type of music. For example, "Rock," "Pop," or "Jazz."
- Artist Name: The name of the musician or group that created the music.
- Track Name: The title of the song or piece of music.
- Track ID: A unique identifier assigned to each track. It's like a serial number that distinguishes one track from another.
- Popularity: A measure of how popular a track is, often based on factors like the number of listens or downloads.
- Acousticness: A measure of how acoustic or non-electronic a track is. High values indicate more acoustic sounds.
- Danceability: A measure of how suitable a track is for dancing based on its rhythm.
- Duration (ms): The length of the track in milliseconds (thousandths of a second).
- Energy: A measure of the intensity and activity of a track.
- Instrumentalness: Indicates whether a track is instrumental (no vocals). Higher values mean more instrumental.
- Key: The key in which the track is composed, representing the musical pitch.
- Liveness: A measure of whether the track sounds like a live performance.
- Loudness: The volume of the track.
- Mode: Indicates whether the track is in a major or minor key.

- Speechiness: A measure of the presence of spoken words in a track.
- Tempo: The speed or pace of a track, measured in beats per minute.
- Time Signature: Specifies the number of beats in a measure and the type of note that receives one beat.
- Valence: A measure of the musical positiveness of a track. Higher values suggest more positive or happy tracks.

****Approaches to Node Relationship Establishment: A Comparative Analysis****

In the context of building a recommendation system for music tracks, we have explored two distinct approaches, each emphasizing different aspects of data representation and retrieval efficiency.

1. Genre and Artist-Based Relationships:

Our initial strategy involved establishing explicit relationships between nodes (songs) based on genre and artist associations. This method began with the creation of genre and artist nodes, followed by connecting individual tracks to their respective genres and artists. Subsequently, the database was queried to extract recommended songs based on genre and artist relationships. This approach prioritizes the inherent organizational structure of music, emphasizing genre and artist affiliations.

2. Similarity-Based Relationships using Machine Learning:

The second approach adopted a machine learning algorithm to infer similarity-based relationships between tracks. Leveraging a sampled dataset derived from the extensive Spotify dataset, we utilized a graph data science library to compute Euclidean similarity scores between tracks. Subsequently, the database was queried to retrieve recommended songs based on these similarity scores. This approach focuses on leveraging machine learning to establish implicit connections between songs, potentially uncovering less evident relationships.

Data Preprocessing:

Prior to establishing relationships, a dataset comprising 19,432 rows (Spotify_ft.csv) was sampled from the original Spotify dataset, which consists of 176,774 rows. The selection of relevant columns was performed, and the data was appropriately scaled using Python to ensure uniformity and comparability.

Programming Languages Used:

The coding implementations were carried out using Python for data preprocessing and Cypher Query Language (CQL) for interacting with the Neo4j graph database. A separate Python file (.pynb) will be provided to showcase the Python code in detail.

This comprehensive methodology allows for a nuanced exploration of the strengths and considerations associated with both relational and similarity-based approaches within the context of music recommendation systems.

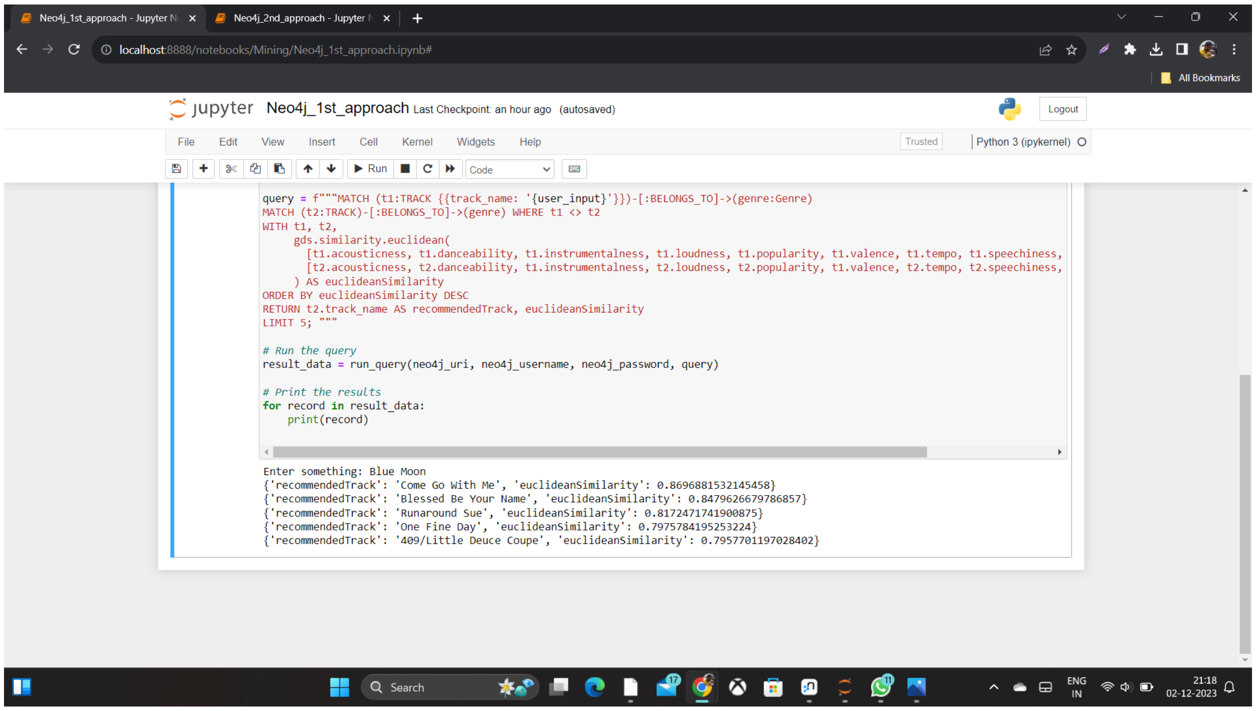
The 1st approach: [SAMPLE_1]
1. To create the nodes with relationships, run the below query,

```
// Create Genre nodes
LOAD CSV WITH HEADERS FROM "file:///Spotify_ft.csv" AS row
MERGE (g:Genre {name: row.genre});

// Create Artist nodes
LOAD CSV WITH HEADERS FROM "file:///Spotify_ft.csv" AS row
MERGE (a:Artist {name: row.artist_name});

// Create TRACK nodes and relationships
LOAD CSV WITH HEADERS FROM "file:///Spotify_ft.csv" AS row
MERGE (t:TRACK {track_id: row.track_id, track_name: row.track_name, popularity:
toFloat(row.popularity), acousticness: toFloat(row.acousticness), danceability:
toFloat(row.danceability), energy: toFloat(row.energy), instrumentalness:
toFloat(row.instrumentalness), liveness: toFloat(row.liveness), loudness: toFloat(row.loudness),
speechiness: toFloat(row.speechiness), tempo: toFloat(row.tempo), valence:
toFloat(row.valence)});

// Create relationships
LOAD CSV WITH HEADERS FROM "file:///Spotify_ft.csv" AS row
MATCH (t:TRACK {track_id: row.track_id})
MATCH (g:Genre {name: row.genre})
MERGE (t)-[:BELONGS_TO]->(g);

LOAD CSV WITH HEADERS FROM "file:///Spotify_ft.csv" AS row
MATCH (a:Artist {name: row.artist_name})
MATCH (t:TRACK {track_id: row.track_id})
MERGE (a)-[:PERFORMS]->(t);
```

In case of large data, we cannot load all the rows at once. So we upload via batches using apoc.periodic.iterate, as shown below (an example):

```
CALL apoc.periodic.iterate(
 'LOAD CSV WITH HEADERS FROM "file:///SpotifyFeatures.csv" AS row RETURN row',
 '
 MERGE (t:Track {track_id: row.track_id})
 SET t.name = row.track_name,
     t.popularity = toInteger(row.popularity),
     t.acousticness = toFloat(row.acousticness),
     t.danceability = toFloat(row.danceability),
     t.duration_ms = toInteger(row.duration_ms),
     t.energy = toFloat(row.energy),
     t.instrumentalness = toFloat(row.instrumentalness),
     t.key = row.key,
     t.liveness = toFloat(row.liveness),
     t.loudness = toFloat(row.loudness),
     t.mode = row.mode,
     t.speechiness = toFloat(row.speechiness),
     t.tempo = toFloat(row.tempo),
     t.time_signature = row.time_signature,
     t.valence = toFloat(row.valence)
 MERGE (a:Artist {artist_name: row.artist_name})
 MERGE (g:Genre {name: row.genre})
 MERGE (a)-[:PERFORMS]->(t)
 MERGE (t)-[:BELONGS_TO]->(g)
 ',
 { batchSize: 1000, parallel: false }
);
```

2. Run the python file with any song name from the dataset.
We’ll get the result like this:

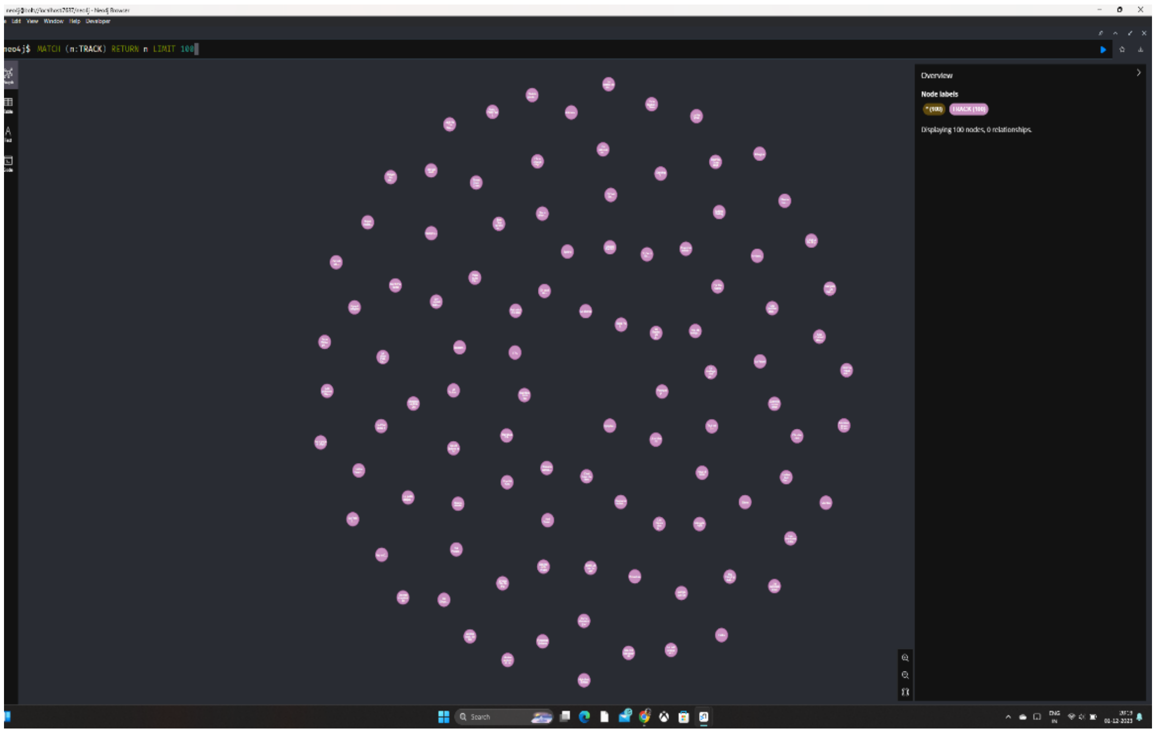

```
query = f"""MATCH (t1:TRACK {{track_name: '{user_input}'}})-[:BELONGS_TO]->(genre:Genre)
MATCH (t2:TRACK)-[:BELONGS_TO]->(genre) WHERE t1 <> t2
WITH t1, t2,
     gds.similarity.euclidean(
       [t1.acousticness, t1.danceability, t1.instrumentalness, t1.loudness, t1.popularity, t1.valence, t1.tempo, t1.speechiness,
       [t2.acousticness, t2.danceability, t1.instrumentalness, t2.loudness, t2.popularity, t1.valence, t2.tempo, t2.speechiness,
     ) AS euclideanSimilarity
ORDER BY euclideanSimilarity DESC
RETURN t2.track_name AS recommendedTrack, euclideanSimilarity
LIMIT 5; """

# Run the query
result_data = run_query(neo4j_uri, neo4j_username, neo4j_password, query)

# Print the results
for record in result_data:
    print(record)
```

```
Enter something: Blue Moon
{'recommendedTrack': 'Come Go With Me', 'euclideanSimilarity': 0.8696881532145458}
{'recommendedTrack': 'Blessed Be Your Name', 'euclideanSimilarity': 0.8479626679786857}
{'recommendedTrack': 'Runaround Sue', 'euclideanSimilarity': 0.8172471741900875}
{'recommendedTrack': 'One Fine Day', 'euclideanSimilarity': 0.7975784195253224}
{'recommendedTrack': '409/Little Deuce Coupe', 'euclideanSimilarity': 0.7957701197028402}
```

The 2nd approach: [SAMPLE]

1. Load the nodes directly without creating a relationship, all the tracks with features

```
LOAD CSV WITH HEADERS FROM "file:///Spotify_ft.csv" AS row
MERGE (t:TRACK {track_name: row.track_name})
SET t.artist_name = row.artist_name,
   t.genre = row.genre,
   t.popularity = toFloat(row.popularity),
   t.acousticness = toFloat(row.acousticness),
   t.danceability = toFloat(row.danceability),
   t.duration_ms = toInteger(row.duration_ms),
   t.energy = toFloat(row.energy),
   t.instrumentalness = toFloat(row.instrumentalness),
   t.liveness = toFloat(row.liveness),
   t.loudness = toFloat(row.loudness),
   t.speechiness = toFloat(row.speechiness),
   t.tempo = toFloat(row.tempo),
   t.valence = toFloat(row.valence)
RETURN t;
```



```
neo4j$ MATCH (n:TRACK) RETURN n LIMIT 100
```

Displaying 100 nodes, 0 relationships.

2. Create an in-memory graph called “track_graph”, which will be added to the catalog. We will load all nodes with the label TRACK into the in-memory graph. Since we don’t have any relationships in our graph yet, we will use a * wildcard as a placeholder.

```
CALL gds.graph.project('track_graph',
'TRACK',
'*',
{nodeProperties:["popularity", "acousticness", "danceability", "energy",
"instrumentalness","liveness","loudness","speechiness","tempo","valence"]})
```

3. The K-Nearest Neighbors algorithm will create relationships in our graph from each node to its closest neighbors according to the Euclidean similarity formula. The topK parameter tells the algorithm to create a relationship called IS_SIMILAR from each node to its 15 nearest neighbors. Before running the algorithm in write mode, we’ll run it in stats mode and examine the distribution of similarity scores.

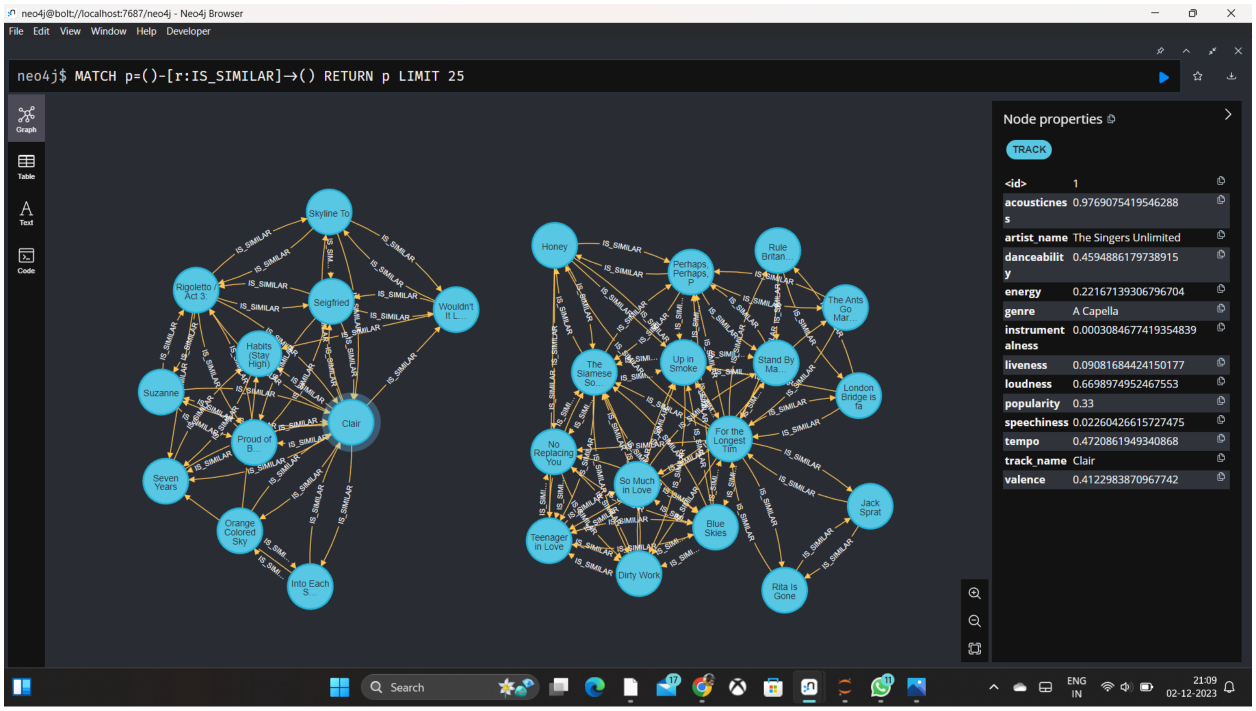
```
CALL gds.knn.stats("track_graph",
  {
    nodeProperties: [
"popularity",
"acousticness",
"danceability",
"energy",
"instrumentalness",
"liveness",
"loudness",
"speechiness",
"tempo",
"valence"
    ],
     topK: 15,
     sampleRate: 1,
     randomSeed: 42,
     concurrency: 1
  }
) YIELD similarityDistribution
RETURN similarityDistribution;
```

4. When we update the in-memory graph with similarity relationships, we will use the similarityThreshold parameter to exclude relationships with similarity scores in the bottom 1%.

```
CALL gds.knn.mutate("track_graph",
            {nodeProperties: [
"popularity",
"acousticness",
"danceability",
"energy",
"instrumentalness",
"liveness",
"loudness",
"speechiness",
"tempo",
"valence"
            ],
            topK: 15,
            mutateRelationshipType: "IS_SIMILAR",
            mutateProperty: "similarity",
            similarityCutoff: 0.92132568359375,
            sampleRate:1,
            randomSeed:42,
            concurrency:1}
           )
```

5. Running KNN in mutate mode created the IS_SIMILAR relationships in our in-memory graph projection. Now write them to the Neo4j graph so that they can be queried with Cypher and visualized in Bloom.

```
CALL gds.graph.writeRelationship(
   "track_graph",
   "IS_SIMILAR",
   "similarity"
)
```



neo4j@bolt://localhost:7687/neo4j - Neo4j Browser

```
neo4j$ MATCH p=()-[r:IS_SIMILAR]→() RETURN p LIMIT 25
```

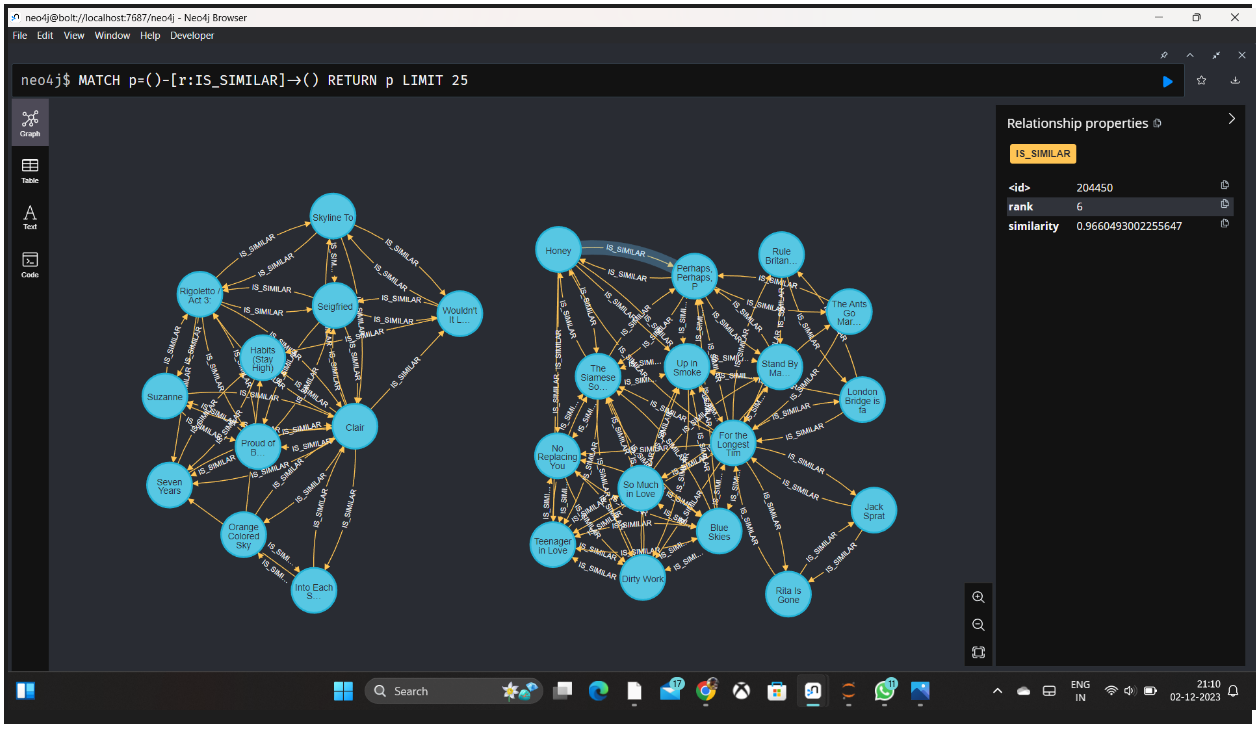
Node properties

TRACK

<id>	1
acousticness	0.9769075419546288
artist_name	The Singers Unlimited
danceability	0.4594886179738915
energy	0.22167139306796704
genre	A Capella
instrumentalness	0.0003084677419354839
liveness	0.09081684424150177
loudness	0.6698974952467553
popularity	0.33
speechiness	0.02260426615727475
tempo	0.4720861949340868
track_name	Clair
valence	0.4122983870967742

6. Then we added the rank property to the IS_SIMILAR relationships so that we could filter the top-ranked relationships for each node.

```
MATCH (m:TRACK)-[s:IS_SIMILAR]->()
WITH m, s ORDER BY s.similarity DESC
WITH m, collect(s) as similarities, range(0, 11) AS ranks
UNWIND ranks AS rank
WITH rank, similarities[rank] AS rel
SET rel.rank = rank + 1
```

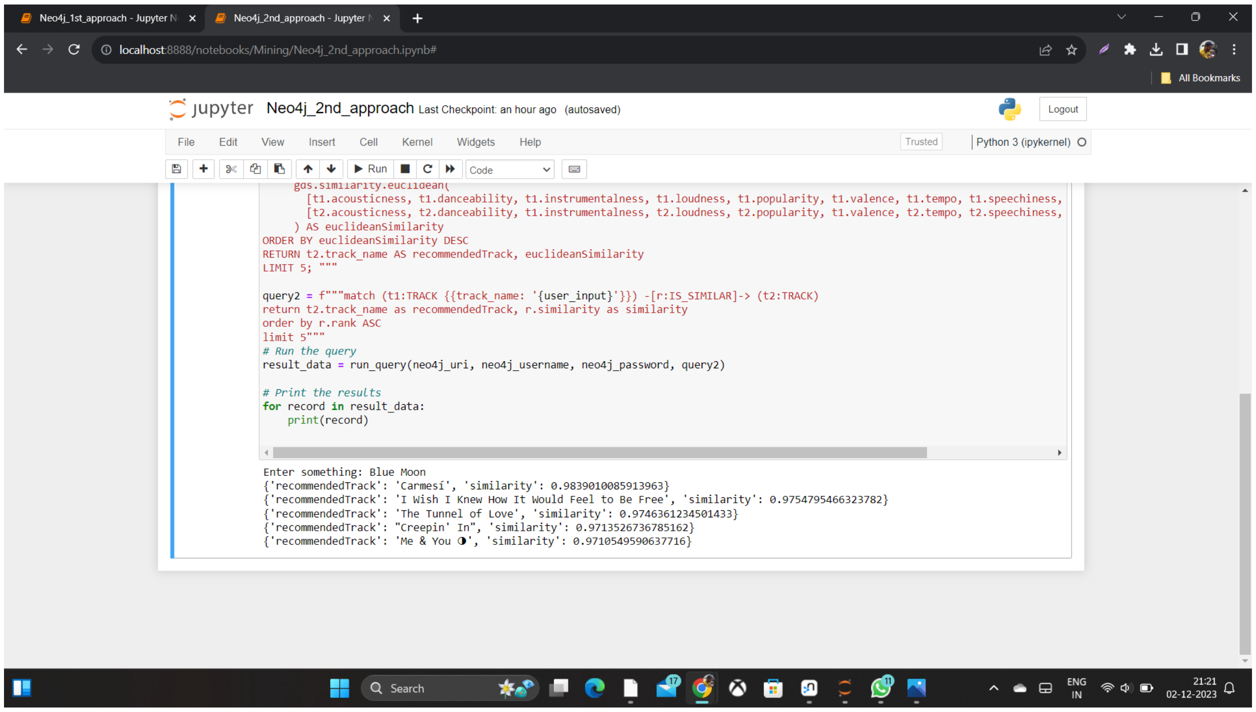

```
neo4j$ MATCH p=()-[r:IS_SIMILAR]→() RETURN p LIMIT 25
```

Relationship properties

IS_SIMILAR

```
<id>        204450
rank        6
similarity  0.9660493002255647
```

Then After this, we queried using rank as a criteria and returned the recommended ones.


```
    gds.similarity.euclidean(
      [t1.acousticness, t1.danceability, t1.instrumentalness, t1.loudness, t1.popularity, t1.valence, t1.tempo, t1.speechiness,
      [t2.acousticness, t2.danceability, t1.instrumentalness, t2.loudness, t2.popularity, t1.valence, t2.tempo, t2.speechiness,
    ) AS euclideanSimilarity
ORDER BY euclideanSimilarity DESC
RETURN t2.track_name AS recommendedTrack, euclideanSimilarity
LIMIT 5; """

query2 = f"""match (t1:TRACK {{track_name: '{user_input}'}}) -[r:IS_SIMILAR]-> (t2:TRACK)
return t2.track_name as recommendedTrack, r.similarity as similarity
order by r.rank ASC
limit 5"""
# Run the query
result_data = run_query(neo4j_uri, neo4j_username, neo4j_password, query2)

# Print the results
for record in result_data:
    print(record)
```

```
Enter something: Blue Moon
{'recommendedTrack': 'Carmesí', 'similarity': 0.9839010085913963}
{'recommendedTrack': 'I Wish I Knew How It Would Feel to Be Free', 'similarity': 0.9754795466323782}
{'recommendedTrack': 'The Tunnel of Love', 'similarity': 0.9746361234501433}
{'recommendedTrack': "Creepin' In", 'similarity': 0.9713526736785162}
{'recommendedTrack': 'Me & You ◑', 'similarity': 0.9710549590637716}
```