A Spotify Recommendation System with Knowledge Graphs

Table of contents:

1. Scenario

1.1. Domain of knowledge graph application

Spotify is a music service that hosts over 80 million tracks on its platform. This includes songs, podcasts, interviews and much more. Over 11 million artists and creators have contributed and more tracks continue to be added every single day [1].

As a user faced with so much choice, which song should I listen to next?

This is the problem that this paper aims to solve. It creates a knowledge graph of a subset of Spotify artists, tracks, and genres, and uses a combination of GraphSAGE embeddings and kNN similarity scores to yield recommendations of the most similar song, artist, or genre to one that the user already likes.

1.2. The service that the knowledge graph provides

The knowledge graph provides recommendations to the user based on the input of their favourite song, artist, or genre. In doing so, it can increase general user satisfaction and the engagement of the user with the platform. This service can be useful when a user is building a new playlist of songs and requires some suggestions to complete it further. Another use case is when the user is playing music in 'discovery' mode, in which the next song is selected automatically.

The key advantages of this knowledge graph are that it is highly scalable and that it is possible to keep adding new nodes without having to retrain the model. Indeed, variations on a recommendation system with GraphSAGE at web-scale can be found in real life applications by companies like Pinterest [2] and UberEats [3]. Section 2.2 discusses the advantages of GraphSAGE in more detail.

2. KG Construction

2.1. Dataset

There are two ways to obtain the data necessary to build this knowledge graph. The first is to use the Spotify API through the `spotipy` python package [4]. This method ensures a high quality of data, but comes with certain volume limitations (e.g. a maximum of 100 tracks can be queried at once). The second method is to use a pre-existing dataset found online, such as the Spotify Dataset 1921-2020 from Kaggle [5]. This dataset is of somewhat lower quality, however, as it contains many empty and incomplete entries. Moreover, there are limitations to the technologies used to store the knowledge graph (i.e. Neo4J Sandbox and Colab notebooks, detailed in section 2.2). Training a model on the entire Kaggle dataset resulted in time-outs and crashes of the notebook and Sandbox.

To avoid losing too much time and after having tried various options, I narrowed down the dataset as follows. First, I pulled the data from a personal playlist of 500 tracks using the Spotify API. Meanwhile, I cleaned the Kaggle dataset to remove duplicates and empty and incomplete entries. I then made a list of all unique artists in the playlist, and pulled all of their tracks from the cleaned Kaggle dataset. This resulted in a dataset of 264 artists, 273 genres and 8674 tracks, or 9211 nodes in total. That volume was manageable for the free tiers of both Neo4J and Colab.

The dataset consists of two files: tracks_subset.csv and artists_subset.csv. These can be publicly accessed through Github Gist [6]. Both files include a number of variables to identify and quantify tracks and artists. The artists_subset.csv file, for example, contains a unique ID for an artist, their name, their number of followers and popularity score when the Kaggle dataset was pulled (April 2021), and the genres that Spotify has classified the artist with. Each artist has their own row and features. An example row from the csv-file can be found in Appendix 7.1.

The tracks_subset.csv file contains general information about a song, such as its name and unique ID, the artist's name and unique ID, the release date, the popularity score of the track, and whether or not the track contains explicit language. In addition to this, Spotify has made 13 audio features available which score each track. These audio features are acousticness, danceability, duration in microsecond, energy, instrumentalness, key, liveness, loudness, mode, speechiness, tempo, a time signature, and valence. An example row from the csv-file can be found in Appendix 7.2 and the exact definitions of the track features are included in Appendix 7.3.

Both the tracks_subset.csv and artists_subset.csv files contain the name and unique ID of the artist, so that the data can be joined together. The dataset exclusively contains songs (as opposed to podcasts, interviews, etc), so the terms 'track' and 'song' are used interchangeably in this paper. A detailed overview of which csv files contain which variables is included in Appendix 7.4. The following csv files are attached to the assignment: the full artists.csv and tracks.csv files from Kaggle, the created subsets artists_subset.csv and tracks_subset.csv, and the file with all tracks of the 500-track playlist: oxfordmain_playlist.csv.

2.2. Technologies used to store and process the knowledge graph

The knowledge graph is created by means of a Colab notebook that connects to a Neo4J Sandbox. Colab notebooks are known for being versatile and well documented tools. For the purposes of this assignment, the notebook is enhanced with six packages: the `os` package and `google.colab.drive` function for the general set-up of a working directory; `sqlite3` to allow for the use of SQL in the notebook [7]; `pandas` to save data in data frames for easy access and manipulation; `spotipy` for the import of data directly from a Spotify playlist [4][8]; and `neo4j` to run commands in the Neo4J Sandbox [9][10]. The Kaggle dataset is imported directly from Kaggle to the working directory. This does not require a separate package, but relies on the use of an API token from the Kaggle account [11].

A Neo4J Sandbox is a cloud-instance of a Neo4J database. The advantage of these sandboxes is that they can be linked directly to a Colab notebook, so that the knowledge graph

can be created from within the same development environment. Neo4J Sandboxes expire after 3 days (with a possible extension of 7 days), but can easily be recreated.

Neo4J is a graph database system that has highly performant read and write scalability without compromising on data integrity [12]. Unlike 'traditional' relational databases, it does not store NULL values, which helps to make it exceptionally fast to query [13]. The system is reliable, easy to use and learn, and is supported by a large online community and robust documentation. It furthermore supports two algorithms that will be used in the ML Representation and Logic Representation sections of this assignment: GraphSAGE and the kNN algorithm. The former will be used to generate embeddings (i.e. fixed length vector representations for each node) and the latter is used to generate similarity scores from the embeddings. This allows us to make recommendations as part of a k-Nearest Neighbors query. The specific reasons why GraphSAGE and kNN were chosen is detailed in sections 3 and 4 respectively.

2.3. Design and description of the construction of the knowledge graph

The available data (tracks_subset.csv and artists_subset.csv) allows us to create a knowledge graph with three types of nodes ('Artist', 'Track', and 'Genre') and two types of edges ('MADE' and 'IN_GENRE'). Artists and tracks can be connected by a 'MADE'-edge if an artist has created a track. Collaborating artists all have a connection to the same track, but no direct connection between each other (this could be added, as discussed in section 5.2). Each artist can be connected to multiple tracks. Furthermore, artists and genres can be connected by an 'IN_GENRE'-edge if the artist is classified as belonging to a given genre. An artist can be connected to multiple genres. Figure 1 shows an example of this in practice.

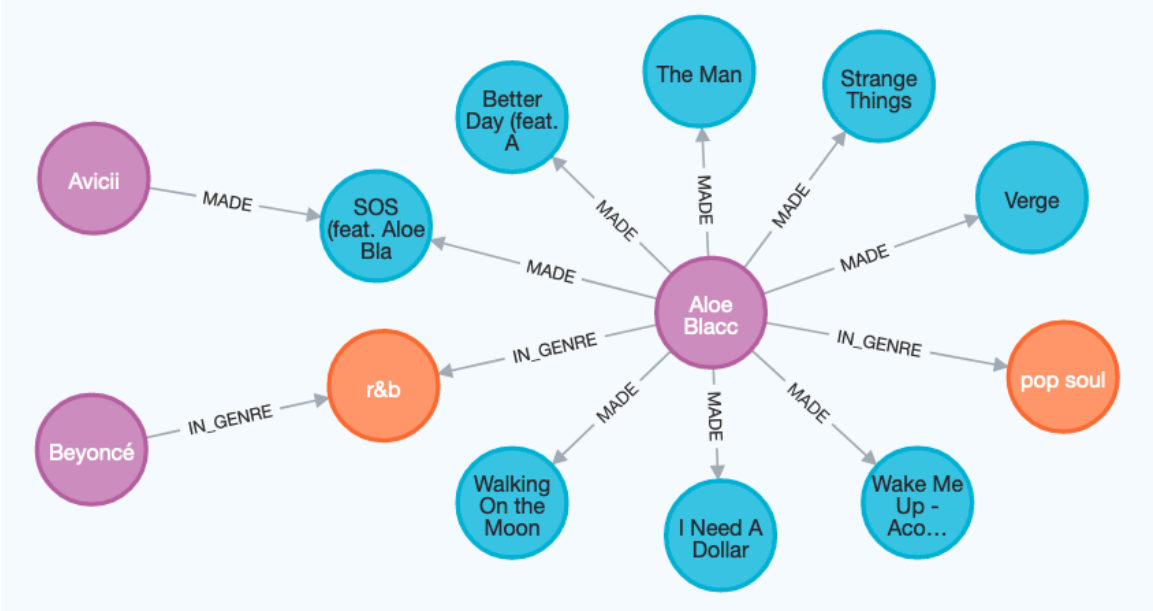

Figure 1: Example of the nodes and edges in practice (screenshot from Neo4J)

To create this knowledge graph, I use a Neo4J connection helper class to establish the connection between the Colab notebook and the Neo4J Sandbox as described in [10]. I then run four queries in the Neo4J query language, Cypher:

1. Setting up constraints to only create a new node if another node with the same name or ID does not exist already. This avoids duplicate nodes;
2. Loading artists_subset.csv, creating the Artist and Genre nodes, and the 'IN_GENRE' edges between them;
3. Loading tracks_subset.csv, creating the Track nodes;
4. Creating the 'MADE' edges between the tracks and the artists, if the track contains the artist's ID in its data.

The result is a knowledge graph of 264 Artist nodes, 273 Genre nodes and 8674 Track nodes, or 9211 nodes in total. There are 999 'IN_GENRE' edges and 8777 'MADE' edges. The full code can be found in the attached python file.

It should be noted that this graph is not a representative sample of all 80 million tracks and artists on Spotify. Since it is based on a personal playlist of 500 tracks, it is biassed towards a specific type of music (i.e. rock and pop from Western countries). Moreover, not all artists and songs are included in the Kaggle dataset, which means that some artists will be underrepresented in the knowledge graph. An artist like Damien Leith is included, for example, but is not connected to any tracks. In addition, older music is more likely to be missing from the graph, because the data quality of those songs was lower overall. Older release dates were more likely to be missing or incomplete, and the songs were therefore removed from the final dataset.

3. ML-based Representation

3.1. Design and description of the ML-based representation

GraphSAGE is a Graph Neural Network algorithm and general inductive framework. It leverages node feature information to efficiently generate node embeddings for previously unseen data [14]. It does so by sampling the neighbourhood of a node and learning a function to calculate the embedding [15]. This function can subsequently be used to generate embeddings for newly introduced nodes as well. As a result, the algorithm is more efficient than Graph Convolutional Networks (GCNs), which require retraining of the model any time that a new node is added to the network [16][17].[1]

[1] Indeed, the majority of GCNs do not scale well over large graphs and/or are designed for whole-graph classification [18]. GraphSAGE embeddings, on the other hand, are highly scalable thanks to the neighbourhood sampling and the learned function. The GraphSAGE framework is nevertheless closely related to GCNs; the authors describe it as essentially an extension of the GCN framework to the inductive setting. Their approach is closely related to Kipf et al [14][19].

GraphSAGE takes the attributes of nodes into account, in contrast to algorithms like node2vec that only consider the graph structure [20]. This is useful for graphs that contain information beyond just the topology of the graph, such as meta data and other features. In GraphSAGE, the graph's topology can be inferred from the embeddings even if they're not directly included [21][22][23].

There are three reasons why I have selected GraphSAGE for the application to the Spotify recommendation system. First, the Spotify use case involves the continuous addition of new nodes, because new music is released all the time. It is computationally inefficient to retrain the model every time an embedding for a new node has to be created. Instead, this embedding can be induced from a previously trained GraphSAGE model. Second, GraphSAGE's consideration of node features suits the application to music recommendations very well. The Spotify dataset contains ample numerical data that scores each song and this information can be accounted for by a GraphSAGE model. Finally, the GraphSAGE framework is highly scalable due to its sampling methodology [23], which is critical for an application of the likes of Spotify. As noted before, variations of GraphSAGE have been proven to be useful for web-scale implementations such as Pinterest [2] and UberEats [3]. It has also been mentioned in Spotify's own research, as their own model builds on neural networks with sampling methodology [36].

To train a GraphSAGE model on the knowledge graph in Neo4J, I first create an in-memory projection of the graph to identify the node types, edge types and properties that have to be taken into account by the model [15][24]. Properties can only be considered if they are a float or integer [25]. This includes the following features from the dataset:

- Artist-nodes: followers, artist_popularity
- Track-nodes: track_popularity, explicit, acousticness, danceability, duration_ms, energy, instrumentalness, key, liveness, loudness, mode, speechiness, tempo, time_signature, valence
- Genre-nodes: none

Various non-numerical features are not included in this list and are therefore not used to train the model. Section 3.3 addresses which features were excluded and why.

Once the in-memory projection is created, the model can be trained and stored in the model catalog using the gds.beta.graphSage.train command. A number of parameters can be configured to determine how the model is trained: the full list can be accessed in the documentation [15]. Unfortunately, the technical limitations of the Colab and Neo4J Sandbox combination do not allow for much room to manoeuvre. GraphSAGE's default settings already optimise for efficiency, and even small adjustments tend to lead to a time-out or crashing of the notebook. As such, it was not possible to optimise the model fully and I stuck with most of the default settings instead. I discuss the changes that I would have made if the technology had allowed for it in more detail in section 3.3. The selected settings to train the model are as follows:

- Embedding dimension: 10 (default: 64)
- Aggregator: Mean (default)

- Activation function: Sigmoid (default)
- Sample size: [25, 10] (default)
- Epochs: 20 (default: 1)
- Projected feature dimension: 17

3.2. Design and description of the evolution of the knowledge graph

Once the model has been trained and stored in the model catalog, it can be invoked to create embeddings through the gds.beta.graphSage.stream command. The following is an excerpt from the output of this command, returning the node ID and 10-dimensional embedding:

```
[<Record nodeId=0 embedding=[0.9979694116789383, 8.088802753152299e-07,
0.037470342002825084, 0.00011811975325478923, 0.00011921565948875587,
3.204195700968575e-06, 0.004826923597565604, 0.0367718014700094,
0.03574240467016771, 0.00011976173926599215]>,
 <Record nodeId=1 embedding=[0.9979694116789383, 8.088802753152299e-07,
0.037470342002825084, 0.00011811975325478923, 0.00011921565948875587,
3.204195700968575e-06, 0.004826923597565604, 0.0367718014700094,
0.03574240467016771, 0.00011976173926599215]>,
```

These embeddings can be added to the in-memory graph projection and the Neo4J graph with the mutate- and write-commands in GraphSAGE. This updates the knowledge graph with a new feature for each node, containing the full embedding information. Figure 2 shows an example of an updated node in Neo4J, which resulted after running the write-command.

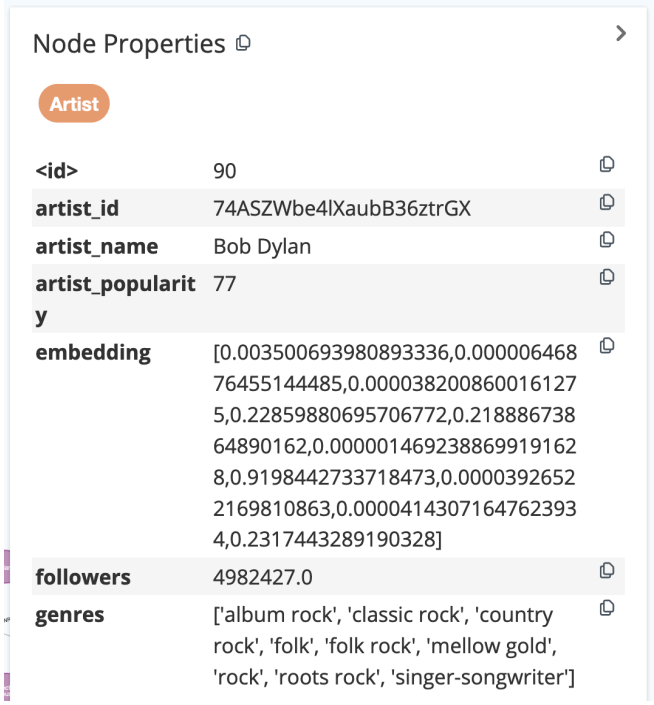

Figure 2: The newly added embedding for the Artist-node Bob Dylan (screenshot from Neo4J)

3.3. Context and limitations of this approach

As mentioned in section 3.1, there are a few limitations to this approach: 1) several features from the knowledge graph are not included when training the model, and 2) technical limitations did not allow for further tweaking and optimization of the model.

With regard to the missing features, the model has not taken into account the names and IDs of artists and tracks, genre names, and the release date of tracks. These features are not numerical (float or integer) and could therefore not be included to train the model. There are ways to convert non-numerical data to numerical data through one hot encoding, but since these variables have a high cardinality (i.e. many unique categories), the dimensionality of the transformed vector would become unmanageable [26]. Moreover, variables like the track name might not have much added value for the embedding: I deemed it unlikely that a user likes two songs simply because they have a similar name. The only exception to this is the release date: that variable could potentially be included if it were converted to microseconds. For the purposes of time, I decided not to pursue this. The knowledge graph embeddings could potentially be improved if this data had been present.

As for the technical issues when training the model, the Colab notebook and Neo4J Sandbox were not capable of processing my preferred settings without timing out or crashing. Below, I discuss the most important parameters that I would have adjusted if I had not been limited by these issues.

- Embedding dimension: this parameter determines the size of the node embedding. It is set to 64 by default, but since the present graph is relatively small (9211 nodes), it is possible to reduce this to gain on memory and computation time. A common rule of thumb is to set the size of the embedding to the 4th root of the number of nodes [27]. That would result in 9211 ** 0.25 = 9.7966 ≈ 10 dimensions. For the purposes of this assignment, I set the dimensions to 10, but I would have increased it if it had been possible.
- Aggregator: the aggregator defines how a node's embedding and the embeddings from its sampled neighbours are combined. The default is the Mean aggregator, but I would have preferred to use the Pool aggregator. The authors of the GraphSAGE algorithm have demonstrated that the latter yields better results [14].
- Activation function: this function maps the input of a neuron to a value from 0 to 1. Neo4J GraphSAGE supports Sigmoid (the default) and Leaky ReLu. The former is known to squeeze information and suffers from the vanishing gradient problem; the latter does not have this issue [28]. I would have picked Leaky ReLu if it had not caused more frequent time-outs.
- Sample size: the default sample size is [25, 10], which means that the algorithm considers 25 neighbours in the first layer and 10 neighbours in the second layer. I would have liked to test different numbers of layers and neighbours, but had to stick to the default settings. Nevertheless, when testing their model, Hamilton et al [14] found

diminishing returns when increasing the number of layers and the size of the neighbourhoods sampled[2], so the default settings are likely to be optimal already.

- Epochs: the default setting for the number of epochs is 1, but I increased this to 20. New neighbours are sampled for each layer in every epoch, so increasing the total number helps to avoid overfitting the model to specific neighbourhoods. Ideally I would have experimented with a higher number of epochs, like 50, but the notebook timed out with anything above 20.
- Projected feature dimension: this parameter is only used in multi-label GraphSAGE, i.e. for graphs with more than one node-type [15]. There are 17 properties that need to be taken into account when training the model, so that is the number I assigned here. I never adjusted this parameter to improve the performance of the model, as I preferred all data to be considered.

Technical limitations aside, GraphSAGE remains a solid framework to train the model with. It scales well to larger graphs thanks to its sampling methodology and inductive nature. The Neo4J GraphSAGE documentation even suggests training the model on a smaller subgraph to save time, as the trained model can be inductively applied to predict embeddings on a *full* graph [15]. Nevertheless, the overhead of subsampling is likely to outweigh the benefit on graphs that are as small as mine. The algorithm was originally intended for use on graphs with 100.000 nodes or more [23].

One final limitation of my approach is that the subset of data remains biassed and that it is not a representative sample of the full Spotify dataset. The model might therefore suffer from overfitting on this specific subset of artists and tracks. This is a trade-off that was necessary to ensure a high quality and sufficient amount of data, as described in section 2.1.

4. Logic-based Representation

4.1. Design and description of the logic-based representation

To arrive at a logic-based representation, I first create similarity scores with the kNN algorithm. These scores can later be used to filter the knowledge graph with specific queries.

The kNN (or k-Nearest Neighbours) algorithm compares the given properties of a node (such as an embedding) with the other nodes in the graph. In doing so, it calculates a distance between the node pairs and assigns a similarity score. The nodes for which the properties are most similar are the k-nearest neighbours [29]. For the Spotify knowledge graph, this similarity can be used to give recommendations to the user as to what they can listen to next.

[2] They state that "[f]or the GraphSAGE variants, we found that setting K = 2 provided a consistent boost in accuracy of around 10-15%, on average, compared to K = 1; however, increasing K beyond 2 gave marginal returns in performance (0-5%) while increasing the runtime by a prohibitively large factor of 10-100×, depending on the neighborhood sample size." [14, p.8]. They found similar diminishing returns for sampling large neighbourhoods.

I selected the kNN algorithm because Neo4J's implementation is highly efficient and scales quasi-linearly [30]. It is based on the NN-Descent algorithm by Dong et al [31], which incorporates the assumption that the neighbours of the neighbours of a node are most likely to be the most similar. The algorithm therefore compares each node to only a limited sample of neighbours, as opposed to comparing it to every other node in the knowledge graph. By adding in parallelization techniques under the hood, the Neo4J implementation of the algorithm avoids scaling in a quadratic way and can be used on larger datasets [32].

The embeddings from which I calculate the similarity scores consist of an array of floating-point numbers. For such data, Neo4J offers three similarity functions as a part of its kNN implementation: cosine similarity, Pearson correlation score, and Euclidean similarity [30]. Cosine similarity is invariant to scaling and suitable for datasets with high variability in the input data. An example could be the comparison of texts of different length and other Natural Language Processing applications. Pearson correlation scores are invariant to scaling as well and are well-suited to compute the correlation between jointly distributed random variables.

Since the knowledge graph embeddings consist of a constant number of data points within predefined ranges and would therefore not benefit from the invariance to scaling, I select Euclidean similarity to calculate the similarity scores in the knowledge graph. Running the gds.alpha.knn.filtered.stream command in Neo4J matches every track to its most similar song as follows:

```
<Record Track1='Billie Jean' Artist1='Michael Jackson'
Track2='Back In NYC' Artist2='Genesis' similarity=1.0>,
<Record Track1='"40" - Remastered 2008' Artist1='U2'
Track2='Brown Sugar - 2009 Mix' Artist2='The Rolling
Stones' similarity=1.0>,
<Record Track1='25 To Life' Artist1='Eminem'
Track2='Teenage Dream' Artist2='T. Rex' similarity=1.0>,
```

These similarities can subsequently be used to write new relationships between the track-nodes of the knowledge graph, as will be described in section 4.2.

Once the relationships have been formed, the knowledge graph can be queried to get recommendations and filter the results. For example, one can run the following query to get songs that have a maximum similarity score when compared to Coldplay's track 'Paradise':

```
MATCH (t:Track)-[s:SIMILAR]->(u:Track)
WHERE t.track_name = "Paradise" AND t.artist_name =
"Coldplay" AND s.similarity_score = 1.0
RETURN t
```

This query returns the node for 'Paradise', which then reveals the subgraph in Figure 3 in Neo4J. According to the knowledge graph, 'Paradise' is similar to another Coldplay song ('A Rush of Blood to the Head') and to various songs from other artists (including Calvin Harris, Ed Sheeran, the Rolling Stones and the Black Eyed Peas).

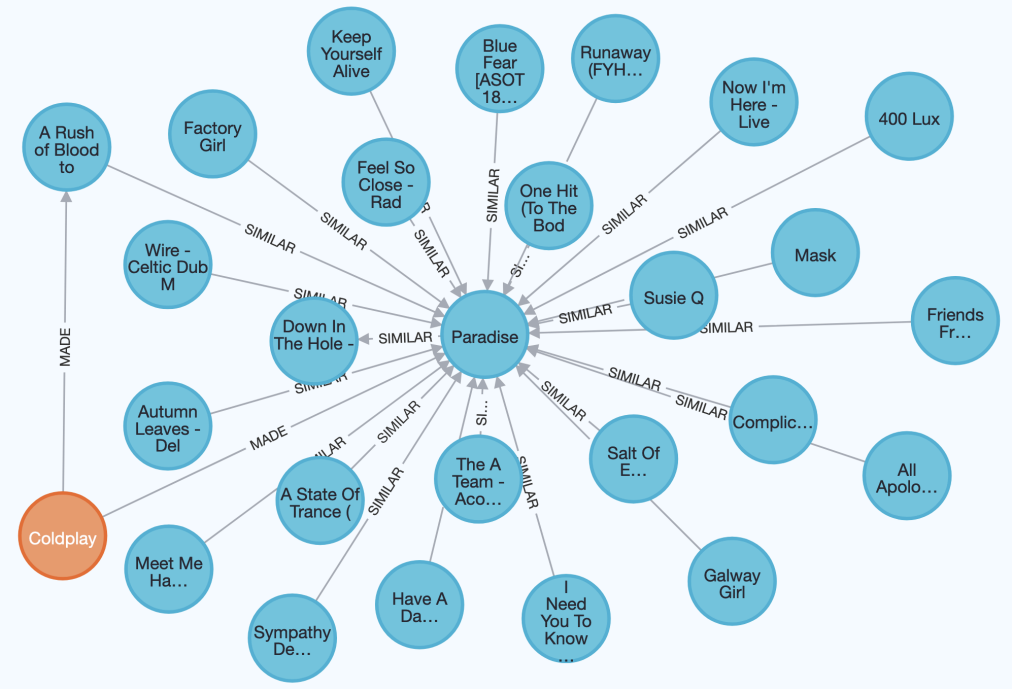

Figure 3: Recommendations related to Coldplay's track 'Paradise'

4.2. Design and description of the evolution of the knowledge graph

The knowledge graph is updated by adding new 'SIMILAR'-edges that contain the similarity score as a property. These edges are created through the gds.alpha.knn.filtered.write command, which compares all nodes in the graph (9211 in total) and adds one relationship for every track (8674 in total). The command can be adjusted to include connections between different node types and to add more connections between the nodes. Figure 4 shows a close-up of two newly formed connections between tracks.

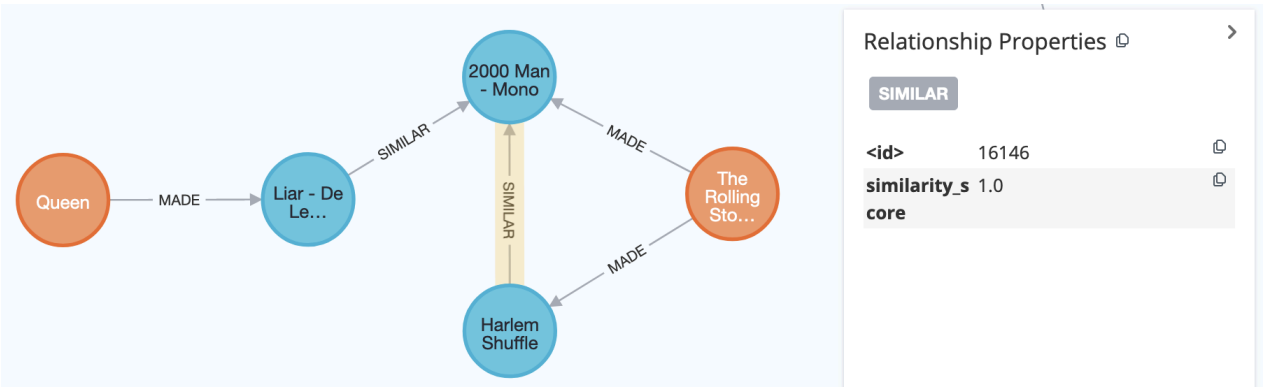

Figure 4: New relationships between two songs of the Rolling Stones, and between a Queen song and a Rolling Stones song. Similarity score = 1.0

Since the Filtered kNN implementation in Neo4J is still in alpha, it is currently not possible to create edges with certain restrictions [33][34]. For example, one cannot limit the

creation of new ‘SIMILAR’-edges based on the similarity score, even though it might not be logical to create a connection between nodes if the similarity score is only 0.1. As such, I ran the following query to remove all ‘SIMILAR’-edges with a score below the threshold of 0.75. This command removed a total of 8 edges.

```
MATCH (t:Track)-[s:SIMILAR]->(u:Track)
WHERE s.similarity_score < 0.75
DELETE s
```

4.3. Context and limitations of this approach

The limitation of this approach is that the kNN algorithm relies entirely on the GraphSAGE embeddings to calculate similarity scores. Following the ‘garbage in, garbage out’-principle, the resulting recommendations may be of low quality if the GraphSAGE embeddings were of low quality as well.

As for the present knowledge graph, the similarity scores indeed seem to be of low quality. When I explored them in more detail, I found that only 28 out of 8674 scores were lower than 0.9 (i.e. 0.3%). This indicates that the embeddings were too similar and that they would likely have benefited from a higher dimension than 10. Unfortunately, it was not possible to increase this parameter due to technical limitations as has been discussed in section 3.3.

A side-effect of this low quality is that some tracks have multiple ‘top recommendations’. After all, there are multiple songs that have a similarity score of 1.0 when compared to those tracks, like a shared first place. The song for which the ‘SIMILAR’-edge was created was randomly picked by the Filtered kNN algorithm in Neo4J, but this does not mean that that song was indeed the best choice.

Nevertheless, Neo4J’s implementation of the kNN algorithm remains an efficient and scalable choice to calculate similarity scores, as long as the quality of input is high. Since the algorithm relies on a sample of all possible neighbours, it can be used on larger graphs as well. In that sense its methodology is similar to that of GraphSAGE.

5. Reflection

5.1. Outcome of the service

This paper set out to answer the question which song a user should listen to next, given that they are faced with the choice from over 80 million tracks on Spotify. The knowledge graph addressed this issue by providing recommendations based on GraphSAGE embeddings and kNN similarity scores.

Although the knowledge graph was successful in generating recommendations, it is difficult to quantify the *quality* of these suggestions. Music taste is highly subjective, so what could be an excellent recommendation for one user might miss the mark completely for

another. Moreover, the quality of the recommendations is highly dependent on the quality of the embeddings and similarity scores, as discussed in the previous section.

The best way to enhance the knowledge graph is to include user-related data. Indeed, all metrics that are currently included in the model are data points that describe the content itself, but they do not say much about the way in which a user listens to this content. For example, it does not consider how often a user has played a song in the past, and how recent those plays were. It also ignores factors like the time of day at which the recommendation is requested: users might be looking for calmer music during work hours and more energetic music on Saturday evening [35].

In addition to user-specific data, the knowledge graph could be improved with group-related information through techniques like collaborative filtering (CF) [cf. 32]. If two users listen to the same songs and user A adds some new music to their playlist, then user B might enjoy that music as well. Spotify calls this “playlist co-occurrence” and has stated that they use such data in their recommendation models [36][37][38]. Since the platform has over 700 million users, this data will be a valuable addition to generate music recommendations.

Finally, the knowledge graph can be further improved by running NLP-models against web-crawled data to better understand how people describe music and artists. In addition to the sonic analysis of how the song sounds, this method adds a social dimension of how the song is perceived. Spotify already uses this in their recommendation systems, as it improves their representation learning significantly [35][36][39].

Nevertheless, the present knowledge graph is a strong starting point for representation learning and a recommendation system for Spotify music. The use of GraphSAGE and the kNN algorithm make it a scalable and efficient system that allows for the continuous addition of new nodes to the graph. The quality of the embeddings could be improved with the help of more technical resources, but the foundations are solidly in place.

5.2. Connections between different forms of AI used

The approach in this paper relied on a combination of GraphSAGE (a representation learning algorithm that generates embeddings) and the kNN algorithm (a classification algorithm that generates similarity scores). The two algorithms are closely connected in the pipeline, as the kNN algorithm takes the GraphSAGE embeddings as input. This means that the quality of the kNN similarity scores is strongly dependent on GraphSAGE’s performance as well, as discussed in section 4.3.

The interaction between the two algorithms could be improved by addressing the technical limitations that throttled the training of the GraphSAGE model. By generating GraphSAGE embeddings with a higher number of dimensions and of a higher quality overall, the downstream similarity scores would become more refined. The resulting recommendations would likely be of higher quality.

In addition, it would be better if the model were trained on a larger subset of the Spotify dataset. I reduced the dataset to 9211 nodes due to the same technical limitations as mentioned above, but the resulting dataset was not quite representative of all 80 million Spotify tracks (or even of the 600K Kaggle dataset). To avoid overfitting on the specific subset

that I selected, it would be better to take a larger subset of data (e.g. 100.000 nodes or more). Both GraphSAGE and the kNN algorithm should be able to handle such larger volumes without issue.

Although the knowledge graph relies on the ML-based creation of 'SIMILAR'-edges, some edges could also be created through logical representation. For example, if two artists collaborate on a given song, that could mean that their other tracks are similar to each other too. Similarly, one could add 'KNOWS'-edges between artists who have worked together in the past, as this might be indicative of music being created in a specific genre. This way, a user can discover new, related music without having to rely on kNN similarity scores.

6. References

[1] Spotify, 'About Spotify', *Spotify*, 2022. https://newsroom.spotify.com/company-info/ .

[2] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, 'Graph Convolutional Neural Networks for Web-Scale Recommender Systems', in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Jul. 2018, pp. 974–983. doi: 10.1145/3219819.3219890.

[3] Uber, 'Food Discovery with Uber Eats: Using Graph Learning to Power Recommendations', *Uber Blog*, Dec. 04, 2019. https://www.uber.com/en-IN/blog/uber-eats-graph-learning/ .

[4] Spotipy, 'Spotipy 2.0 documentation', 2022. https://spotipy.readthedocs.io/en/2.9.0/# .

[5] Y. E. Ay, 'Spotify Dataset 1921-2020, 600k+ Tracks', 2021. https://www.kaggle.com/datasets/yamaerenay/spotify-dataset-19212020-600k-tracks .

[6] Github Gist, 'Artists_subset and Tracks_subset CSV files', *Gist*, 2023. https://gist.github.com/natromno/70167d69974959af031234a1f674860a .

[7] M. S. Leo, 'Have a SQL Interview Coming Up? Ace It Using Google Colab!', *Medium*, Aug. 03, 2021. https://towardsdatascience.com/have-a-sql-interview-coming-up-ace-it-using-google-colab-6d3c0ffb29dc

[8] S. Jones, 'Extracting Your Fav Playlist Info with Spotify's API', Mar. 04, 2021. https://www.linkedin.com/pulse/extracting-your-fav-playlist-info-spotifys-api-samantha-jones.

[9] *Part 1: Bite-Sized Neo4j for Data Scientists - Connect from Jupyter to a Neo4j Sandbox*, (Aug. 06, 2021). [Online Video]. Available: https://www.youtube.com/watch?v=Niys6g6NFfw

[10] C. J. Sullivan, 'Create a graph database in Neo4j using Python', *Medium*, Feb. 15, 2021. https://towardsdatascience.com/create-a-graph-database-in-neo4j-using-python-4172d40f89c4 .

[11] M. Gupta, 'How to fetch Kaggle Datasets into Google Colab', *Analytics Vidhya*, Nov. 07, 2020. https://medium.com/analytics-vidhya/how-to-fetch-kaggle-datasets-into-google-colab-ea682569851a

[12] I. Robinson, J. Webber, and E. Eifrem, *O'Reilly's Graph Databases*. 2015. Available: https://neo4j.com/lp/book-graph-databases/

[13] Neo4J Graph Data Platform, 'White Paper: Powering Real-Time Recommendations with Graph Database Technology', *Neo4j Graph Data Platform*, 2021. https://neo4j.com/whitepapers/recommendations-graph-database-business/

[14] W. L. Hamilton, R. Ying, and J. Leskovec, 'Inductive Representation Learning on Large Graphs'. arXiv, Sep. 10, 2018. doi: 10.48550/arXiv.1706.02216.

[15] Neo4J Graph Data Platform, 'GraphSAGE - Neo4j Graph Data Science', *Neo4j Graph Data Platform*, 2022. https://neo4j.com/docs/graph-data-science/2.2/machine-learning/node- embeddings/graph-sage/

[16] A. Ruberts, 'GraphSAGE for Classification in Python', *Well Enough*, May 04, 2021. https://antonsruberts.github.io/graph/graphsage/

[17] S. Wu, F. Sun, W. Zhang, X. Xie, and B. Cui, 'Graph Neural Networks in Recommender Systems: A Survey'. arXiv, Apr. 02, 2022. doi: 10.48550/arXiv.2011.02260.

[18] W. L. Hamilton, R. Ying, and J. Leskovec, 'Representation Learning on Graphs: Methods and Applications'. arXiv, Apr. 10, 2018. doi: 10.48550/arXiv.1709.05584.

[19] *Intro to graph neural networks (ML Tech Talks - Tensorflow)*, (Jun. 17, 2021). [Online Video]. Available: https://www.youtube.com/watch?v=8owQBFAHw7E

[20] Stellargraph, 'Node representation learning with GraphSAGE and UnsupervisedSampler — StellarGraph 1.2.1 documentation', 2022. https://stellargraph.readthedocs.io/en/stable/demos/embeddings/graphsage-unsupervised-sampler-embeddings.html

[21] *Graph SAGE - Inductive Representation Learning on Large Graphs | GNN Paper Explained*, (Jan. 03, 2021). [Online Video]. Available: https://www.youtube.com/watch?v=vinQCnizqDA

[22] Neo4J Graph Data Platform, 'Graph Embeddings - Developer Guides', *Neo4j Graph Data Platform*, 2023. https://neo4j.com/developer/graph-data-science/graph-embeddings/ .

[23] W. L. Hamilton, 'williamleif/GraphSAGE'. Jan. 10, 2023. [Online]. Available: https://github.com/williamleif/GraphSAGE

[24] Neo4j Graph Data Platform, 'Projecting graphs using native projections - Neo4j Graph Data Science', *Neo4j Graph Data Platform*, 2022. https://neo4j.com/docs/graph-data-science/2.2/management-ops/projections/graph-project/ .

[25] Neo4J Community, 'Gds.graph.create doesn't support String for nodeProperties data type', Dec. 08, 2020. https://community.neo4j.com/t5/neo4j-graph-platform/gds-graph-create-doesn-t-support-string-for-node properties-data/m-p/33317#M17693 .

[26] V. Lakshmanan, S. Robinson, and M. Munn, *Machine Learning Design Patterns: Solutions to Common Challenges in Data Preparation, Model Building, and MLOps*, 1st edition. O'Reilly Media, 2020.

[27] Google Developers Blog, 'Introducing TensorFlow Feature Columns', Nov. 20, 2017. https://developers.googleblog.com/2017/11/introducing-tensorflow-feature-columns.html .

[28] P. Covington, J. Adams, and E. Sargin, 'Deep Neural Networks for YouTube Recommendations', presented at the RecSys '16, Sep. 2016.

[29] D. Subramanian, 'A Simple Introduction to K-Nearest Neighbors Algorithm', *Medium*, Jul. 12, 2021. https://towardsdatascience.com/a-simple-introduction-to-k-nearest-neighbors-algorithm- b3519ed98e .

[30] Neo4J Graph Data Platform, 'K-Nearest Neighbors', *Neo4j Graph Data Platform*, 2022. https://neo4j.com/docs/graph-data-science/2.2/algorithms/knn/ .

[31] W. Dong, C. Moses, and K. Li, 'Efficient k-nearest neighbor graph construction for generic similarity measures', in *Proceedings of the 20th international conference on World wide web*, New York, NY, USA, Mar. 2011, pp. 577–586. doi: 10.1145/1963405.1963487.

[32] Z. Blumenfeld, 'Exploring Practical Recommendation Engines In Neo4j', *Medium*, Apr. 12, 2022. https://towardsdatascience.com/exploring-practical-recommendation-engines-in-neo4j-ff09fe767782 .

[33] Neo4J Graph Data Platform, 'Filtered Node Similarity - Neo4j Graph Data Science', *Neo4j Graph Data Platform*, 2022. https://neo4j.com/docs/graph-data-science/2.2/algorithms/alpha/ filtered-node-similarity/ .

[34] Neo4J Community, 'Graph Data Science: Filtered Node Similarity', Aug. 24, 2022. https://community.neo4j.com/t5/neo4j-graph-platform/graph-data-science-filtered-node-similarity/m-p/59479#M35389 .

[35] D. Pastukhov, 'How Spotify's Algorithm Works? A Complete Guide to Spotify Recommendation System [2022] | Music Tomorrow Blog', Feb. 09, 2022. https://www.music-tomorrow.com/blog/how-spotify-recommendation-system-works-a-complete-guide-2022 .

[36] A. Saravanou, F. Tomasi, R. Mehrotra, and M. Lalmas - Spotify, 'Multi-Task Learning of Graph-based Inductive Representations of Music Content', in *Proceedings of the 22nd International Society for Music Information Retrieval Conference*, Online, Nov. 07, 2021, pp. 602–609. doi: 10.5281/zenodo.5624379.

[37] L. Mok, S. F. Way, L. Maystre, and A. Anderson, 'The Dynamics of Exploration on Spotify', *Spotify Research*, Jun. 2022. https://research.atspotify.com/publications/the-dynamics-of- exploration-on-spotify/

[38] J. Wirfs-Brock, S. Mennicken, and J. Thom, 'Giving Voice to Silent Data: Designing with Personal Music Listening History', *Spotify Research*, May 15, 2020. https://research.atspotify.com/2020/05/giving-voice-to-silent-data-designing-with-personal-music-listening-history/

[39] A. Wang, A. Pappu, and H. Cramer, 'Representation of Music Creators on Wikipedia, Differences in Gender and Genre', *Spotify Research*, May 2021. https://research.atspotify.com/publications/representation-of-music-creators-on-wikipedia-differences-in-gender-and-genre/ .

[40] Spotify for Developers, 'Web API Reference', 2022. https://developer.spotify.com/documentation/web-api/reference/ (accessed Dec. 18, 2022).

7. Appendix

7.1. Artists_subset.csv example row

artists_id	artists	genres	followers	popularity
1dfeR4HaWDbWqFHLkxsg1d	Queen	['classic rock', 'glam rock', 'rock']	33483326.0	89

7.2. Tracks_subset.csv example row

track_id	track_name	artist_id	artist	release_date	popularity	explicit	...
5vdp5UmvTsnMEMESIF2Ym7	Another One Bites The Dust - Remastered 2011	1dfeR4HaWDbWqFHLkxsg1d	Queen	1980-06-27	82	False	

...	acousticness	danceability	duration_ms	energy	instrumentalness	key	...
	0.112	0.933	214653	0.528	0.312	5	

...	liveness	loudness	mode	speechiness	tempo	time_signature	valence
	0.163	-6.472	0	0.161	109.967	4	0.754

7.3. Definitions of Spotify's audio features

Source: [40]

Features	Type	Definition
acousticness	number<float>	"A confidence measure from 0.0 to 1.0 of whether the track is acoustic. 1.0 represents high confidence the track is acoustic. >= 0 <= 1"
danceability	number<float>	"Danceability describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity. A value of 0.0 is least danceable and 1.0 is most danceable."
duration_ms	integer	"The duration of the track in milliseconds."

energy	number<float>	“Energy is a measure from 0.0 to 1.0 and represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy. For example, death metal has high energy, while a Bach prelude scores low on the scale. Perceptual features contributing to this attribute include dynamic range, perceived loudness, timbre, onset rate, and general entropy.”
explicit	boolean	“Whether or not the track has explicit lyrics (true = yes it does; false = no it does not OR unknown).”
instrumentalness	number<float>	“Predicts whether a track contains no vocals. "Ooh" and "aah" sounds are treated as instrumental in this context. Rap or spoken word tracks are clearly "vocal". The closer the instrumentalness value is to 1.0, the greater likelihood the track contains no vocal content. Values above 0.5 are intended to represent instrumental tracks, but confidence is higher as the value approaches 1.0.”
key	integer	“The key the track is in. Integers map to pitches using standard Pitch Class notation. E.g. 0 = C, 1 = C♯/D♭, 2 = D, and so on. If no key was detected, the value is -1. >= -1 <= 11”
liveness	number<float>	“Detects the presence of an audience in the recording. Higher liveness values represent an increased probability that the track was performed live. A value above 0.8 provides strong likelihood that the track is live.”
loudness	number<float>	“The overall loudness of a track in decibels (dB). Loudness values are averaged across the entire track and are useful for comparing relative loudness of tracks. Loudness is the quality of a sound that is the primary psychological correlate of physical strength (amplitude). Values typically range between -60 and 0 db.”
mode	integer	“Mode indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived. Major is represented by 1 and minor is 0.”
popularity (track)	integer	“The popularity of a track is a value between 0 and 100, with 100 being the most popular. The popularity is calculated by algorithm and is based, in the most part, on the total number of plays the track has had and how recent those plays are. Generally speaking, songs that are being played a lot now will have a higher popularity than songs that were played a lot in the past. Duplicate tracks (e.g. the same track from a single and an album) are rated independently. Artist and album popularity is derived mathematically from track popularity. Note: the popularity value may lag actual popularity by a few days: the value is not updated in real time.”
popularity (artist)	integer	“The popularity of the artist. The value will be between 0 and 100, with 100 being the most popular. The artist's popularity is calculated from the popularity of all the artist's tracks.”
release_date	string	“The date the album was first released”

speechiness	number<float>	“Speechiness detects the presence of spoken words in a track. The more exclusively speech-like the recording (e.g. talk show, audio book, poetry), the closer to 1.0 the attribute value. Values above 0.66 describe tracks that are probably made entirely of spoken words. Values between 0.33 and 0.66 describe tracks that may contain both music and speech, either in sections or layered, including such cases as rap music. Values below 0.33 most likely represent music and other non-speech-like tracks.”
tempo	number<float>	“The overall estimated tempo of a track in beats per minute (BPM). In musical terminology, tempo is the speed or pace of a given piece and derives directly from the average beat duration.”
time_signature	integer	“An estimated time signature. The time signature (meter) is a notational convention to specify how many beats are in each bar (or measure). The time signature ranges from 3 to 7 indicating time signatures of "3/4", to "7/4". >= 3 <= 7”
valence	number<float>	“A measure from 0.0 to 1.0 describing the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry). >= 0 <= 1”

7.4. Overview of variables in each csv file

	tracks_subset	artists_subset
track_id	X	
track_name	X	
artists_id	X	X
artist_name	X	X
release_date	X	
popularity (track)	X	
explicit	X	
genres		X
followers		X
popularity (artist)		X

Spotify audio features		
acousticness	X	
danceability	X	
duration_ms	X	
energy	X	
instrumentalness	X	
key	X	
liveness	X	
loudness	X	
mode	X	
speechiness	X	
tempo	X	
time_signature	X	
valence	X	