Personalized music search based on graph embedding

Christian Esswein
Databases and Information Systems
Department of Computer Science
University of Innsbruck
christian.esswein@student.uibk.ac.at

ABSTRACT

While exploring new music, users are typically limited to recommender systems which are proposing items either based on their listening history or on content similarities. Combining both methods models a "query-based recommendation" which enables users to filter content based on their preferences. The ecosystem of music can be represented as an heterogeneous graph using all available data like tracks, artists, genres, tags and users. Using graph embedding techniques a low-dimensional vector representation is learned and provides a simple method to calculate similarities. Search queries, either single terms or combinations of items in the music graph, can be encoded using the same vector space. Therefore, not only exact results are found, but also similar items.

This work presents an approach to create a music search with Spotify data where new users can connect with their existing accounts. The retrieved results are not only presented in a list but instead the learned vector representation is exploited to generate 3D representations.

KEYWORDS

Recommender Systems, Graph Embedding, Personalization

ACM Reference Format:
Christian Esswein. 2017. Personalized music search based on graph embedding. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In recent years music streaming platforms are getting more popular which enables users to access huge collections of music. With this trend it has also changed how users search and explore music [10]. For example Spotify[1] with currently 140 million active users has over 30 millions songs to offer (as of June 2017) [1]. As a consequence of this huge collection of songs, the primary objective for user is not to find specific songs anymore but to find songs matching seeding criteria. While exploring new music, users are typically limited to recommender systems which are proposing items either based on their listening history or on content similarities (aka find similar artists, songs). Combining both methods models a "query-based recommendation" which enables users to filter content based on their preferences.

When using naive approaches with simple attribute matching queries, complete annotated metadata is required. Otherwise songs can not appear when querying for tags which they are not assigned to. Especially when it comes to genres and tags for tracks it is unfeasible to manually annotate every entity. A desired search engine needs to learn the representation with some tags in the training phase but while predicting new connections have to be drawn. The ecosystem of music can be represented as an heterogeneous graph using all available data like tracks, artists, genres, tags and users. Using graph embedding techniques a low-dimensional vector representation is learned and provides simple method to calculate similarities. Search queries, either single terms or combinations of items in the music graph, can be encoded using the same vector space. As a result, not only exact results are found, but also similar items.

This work presents an approach to embed the Spotify dataset into a latent representation to enable recommendations based on queries. After providing the first search results, the user should be assisted while exploring the remaining items and in the refinement of his search terms. Traditional list based aggregations of search results can only model a one dimensional view for the items. Instead of a list, the low-dimensional vector representation should be exploited to generate 3D representations of the suggested items. Combined with the graph representation, query suggestions can be provided.

2 RELATED WORK

In [2] a very similar approach is presented where query-based music recommendations are created with embedded graphs. The main difference is that the music graph was modeled as a bipartite graph with users in one and all other items in the other set. This allows to create next track recommendations based on recent seed tracks but does not model a full qualified search because music items themselves are not connected.

To evaluate the performance of my query recommendations, playlist names are used to predict their tracks. On the same dataset [3] presented a text-based method to embed and query for tracks. Here similarities between items are only based on co-occurring songs.

Alternative representation and exploration models for music are mainly complex and require additional hardware. [9] presented a 3D interface to visualize similarities between tracks. Each item is represented as a single item but no clusters are used and album covers are only involved in separate visualizations. The music space

[1] http://www.spotify.com

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

is mapped on a virtual landscape in [8] which makes it possible to observe audio based similarities and clusters but the interface is rather complex.

3 GRAPH EMBEDDING

Graph embedding techniques aiming to transform graph structures into low a dimensional vector space. More formally, given a graph $G = (V, E)$ with vertices V and edges E "a graph embedding is a mapping $f : v_i \rightarrow y_i \in \mathbb{R}^d \; \forall i \in [n]$ such that $d \ll |V|$ and the function f preserves some proximity measure defined on graph G" [5]. The final result is therefore a vector representation of each node in the initial graph. Having this coherent search space makes it much easier to calculate higher-order proximities between heterogeneous nodes. Similar items can be retrieved using nearest-neighboring searches.

Existing embedding algorithms can in general be categorized into factorization based, random walk based and deep walking based methods. Concerning time complexity and preserved higher order proximities, mainly random walk based methods are interesting. Others methods either only embed similarities between connected nodes or their runtime is dependent on the number of edges in contrast to $O(|V|)$ [5]. Short random walks over edges are used to generate sentences which reflect the graph structure. Using this walks as training data, a representation for each word is learned. In the field of natural language processing this method is known as word embedding and especially popular with *Word2vec* [12]. Both *Deepwalk* [13] and *node2vec* [6] are making use of *Word2vec* in their reference implementations to embed arbitrary graph structures with random walks. For this work *Deepwalk* was chosen because despite that *node2vec* retrieves better embeddings in theory, it is not possible to extend the graph structure after its initial creation as explained in the next section.

3.1 Model extension

In real world scenarios, the dataset is not fixed and during runtime, more data is collected which needs to be included into the model. New data can either consists of new edges, e.g. new listening events of users during the use of the system, but also of new vertices when tracks or users are added. Especially if new users do not suffer from the cold start problem because initial data is available (e.g. through connecting with other services, see 5), a fast method is desirable such that the system can be used right away. A naive approach could simply recreate the whole embedding, but this is not scalable for bigger sets.

Random walk based embedding techniques are mostly online algorithms which can consume new walks as they are produced. This property is very powerful in general because for huge datasets not all walks have to be generated in advanced or even kept in memory. Unfortunately, both reference implementations for *node2vec* and *DeepWalk* do not provide interfaces to store the internal state, retrieve intermediate embeddings or continue learning. Even worse, such interfaces could only improve node proximities performance through adding edges. In order to be able to include new vertices, the internal data structures must be extended. For *node2vec*, such an extension is not possible because the probability distribution of random walks is not uniform and precomputed before walk generation. For doing this, the whole graph has to be known and no graph structure can be added afterwards because it would invalidate previous transition probabilities.

However, *DeepWalk* uses a uniform distribution over random walks and as previously noted, uses *Word2vec* to compute the actual embedding. Using *Word2vec* in the backend makes it possible to store the current internal computational model and later restore it for further learning and additionally also allows to extend the current vocabulary, in this case adding new nodes, during runtime. Together the desired options are available to partially extend the existing graph and retrieve the new embedding.

To extend an existing embedding, first, new vertices and edges have to be collected and appended to the existing graph. Then new random walks are generated, but only over added vertices and edges. This new training data has only a small size compared to the initial data set and is proportional to the number of added structure. Then finally the existing *Word2vec* model can be loaded, extended with new nodes, and learning is continued with new walks.

With this method, graph and embedding can be updated with much less effort than relearning the complete model. Even small changes can influence the whole embedding and create a new vector space. The scalability is therefore questionable because for big datasets the new embedding has to be updated completely which may invalidate all created indexes. Furthermore only graph structure can be extended but not modified or removed.

4 PERSONALIZED QUERIES

The increasing availability of huge music datasets through streaming platforms requires more sophisticated search methods to access desired tracks. Especially on streaming platforms, users expect to filter and consume music with predefined seeds like genres, context based scenarios or queries to find similar items of "x". This content based filters are in contrast to personal recommendations which are only based on historic listening behavior. To combine those two concepts and to allow users to specify arbitrary search intention, a flexible query system is required.

The music corpus can be modeled in a heterogeneous graph with tracks, artists, album and tags as nodes and relationships as edges. Using graph embedding techniques as described in the previous section, a vector space with preserved proximities is created. Each item from the graph is included in the final embedding and can therefore be used as query term. Nearest neighbors represent similar items with decreasing confidence on higher distances. Because every query is formulated with vectors, also the combination of different search terms is possible. The retrieved items have mixed item types which means that not only tracks are returned but also artists for example. This is very powerful because neither user nor system have to decide at first hand the item type but can also easily apply filters. Another benefit is, that similarity measures between different item types are possible.

It is crucial to include as many graph structure as possible to produce meaningful outputs. Tracks together with connected artists for example are clearly too sparse, mainly because no edges between different tracks or artists exist. Including tags for tracks and

genres for artists does solve two problems. On the one hand side, it models similarities between different items and therefore improves the embedding. On the other hand, it enriches the possible terms which can be used by users to construct queries.

Until now the user itself was not included into the system. If user feedback is available, e.g. through historic track listening behavior or positive feedback on items, this data can be included into the graph. This improves the available graph structure and therefore may improve the embedding quality and additional makes it possible to model a user preference on queries. For each user initiated search query, the user context is added and influences results with personal recommendations. Using the user without additional seed items even retrieves general recommendations with the same system.

4.1 Search refinement

Searching for music is not a single action process where a user formulates query intention and then consumes the results. The search can be seen as a process where query refinements are always part of it. Therefore it is necessary that users are not only able to extend queries but also are supported with suggested terms. The user should feel like navigating through a virtual result space instead of jumping to unconnected places after manually modifying requirements. Having a search query which is represented as a vector, it can partially constructed as a combination of multiple search term vectors. This means that any of the proposed items can be used to further extend the query and refine the search.

For example suppose that you first search for a music genre you are interested in and then find an artists in the results which matches your search intention. Adding this artist to the current query will not limit new results to the added artists but will return items which are similar to the genre and artist. Using this technique the user has not actively adapt the query through reformulating it and still gets more precisely results.

4.2 Result representation and exploration

The most common visualization for recommendations is a list of items which is ordered by relevance for the user. Only the ordering can be observed and therefore it represents a very limited one dimensional view where even distance measures between consecutive items are not visible. To create at least a sense of vision for distances to the most significant items, it is common to spread the results among multiple result pages. When having an embedding where similarities between arbitrary items can be expressed using vectors, more advanced interfaces are possible. Using dimension reducing methods like the Principal component analysis (PCA), the embedding can be compressed to three dimensions. Instead of using a list, the retrieved recommendations items can then easily visualized in a 3D scene. Each track, artist or album has a corresponding position vector and is representable as interactive 3D object. Using such an interface does not only express distance measure between items but also allows the user to explore the result space more easily. Less accurate items are positioned around the center hit and the direction in three dimension can be chosen to explorer additional items. This means that the query itself is refined while exploring the result space indirectly and without active user involvement.

Similar items are positioned very nearby and when applying distance based clustering, accumulations of items can be represented as single but annotated clusters. For exploration, only one item has to be inspected for deciding whether the whole cluster is in interest or not. Postfiltering to remove variants of very similar items is then not necessary anymore while all items are still available. Additionally this also simplifies the visualization because more items can be displayed on less space without introducing noise.

It is crucial to simplify the inspection of single items such that big collections are explorable in reasonable time. Album covers as textures for 3D objects improve the vision because known items can be recognized and the discovery or desired tracks is more efficient [11]. To provide detail information about selected items (e.g. artist name for track, genres), directly connected neighbors can be retrieved from the initial graph structure. Finally music samples have to be provided to rate and consume discovered tracks.

Through advantages in browser technology just in time visualizations of 3D scenes can be created directly on websites without complex precomputations or add-ons. Also users are nowadays more familiar with interactive and multidimensional interfaces. Enhanced result views could therefore may replace simple lists.

5 GEMSEARCH

geMsearch is a prototype which implements the proposed method for personalized music search. An embedded music corpus is searchable through a web client and retrieved items are represented in a list or 3D view which can be seen in figure 1 and 2. Besides the offline evaluation this makes it possible to test the implementation in a real world scenario.

Supported with autocomplete suggestion the user can select any item of the graph to formulate single or combined queries. To provide recommendations, this selection is send to the API server instance and evaluated on the embedded graph. User filters can further restrict the search results for certain element types, like tracks, tags or artists. Before sending results to the client, additional metadata like artist names for tracks and album covers are queried from a simple document store and added to the response. Because of its popularity and easy to use API the data highly depends on the Spotify API. This ensures a high availability of metadata as well as album covers and makes it possible to play short sound samples for each track.

To overcome the cold start problem for user profiles and making the client usable, users can connect with their Spotify account. The official Spotify API supports the OAuth protocol with different scopes which makes it possible to access personal playlists, playing history and saved tracks. To create a personal preference profile, *geMsearch* only accesses the saved tracks because it requires much less preprocessing and computational effort than the whole playing history which is potential very big and contains much more noise. After a user has connected with his account, the music library is loaded and compared with known tracks in the current database.

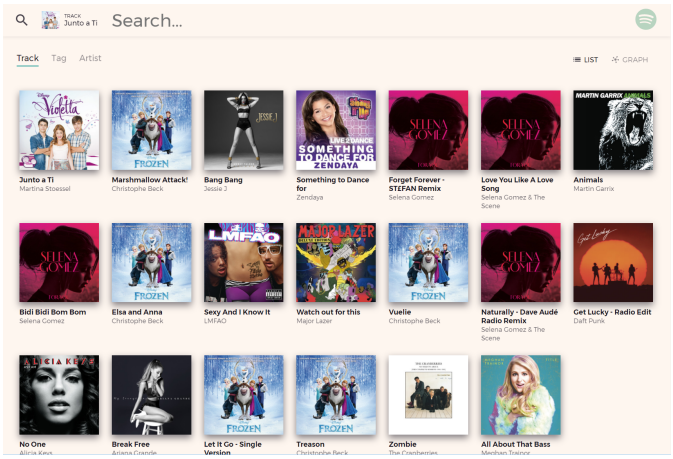

Figure 1: Web client list results

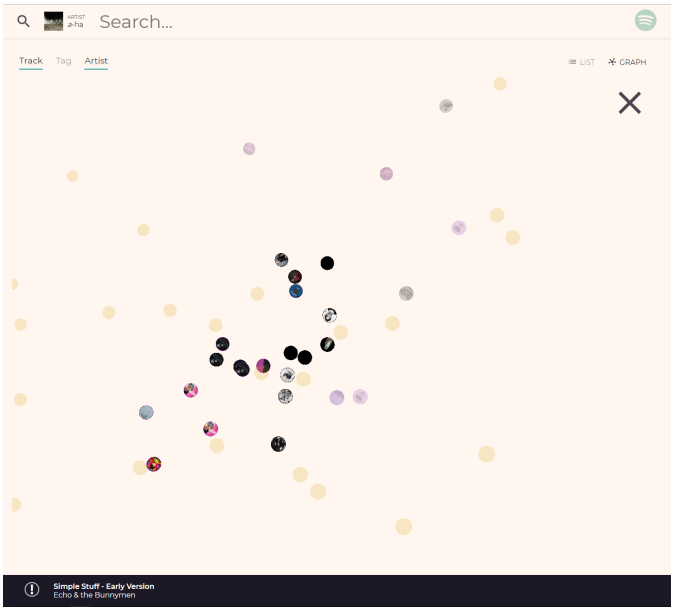

Figure 2: Web client 3D view and player bar

Missing songs are crawled to include metadata from Spotify about the track and artist, as well as user curated tags from Last.fm. After this data is collected, the graph can be extended and included into the existing embedding.

This approach can only recommend and present known items in the database which clearly does not reflect all tracks from Spotify. The initial model is pretrained and evaluated with playlists as discussed in the next section and with every user this dataset is extended and improved.

6 EXPERIMENTS

Estimating the quality of the presented approach is one of the key challenges because there exist no dataset which directly maps search queries with user context to results. Even having a working prototype client to test the system on real users, representative user studies require fairly big and diverse users bases. The test cases need to allow the users to estimate the results without being biased through available options or the test environment. A/B tests can model fair and solid results but require scopes in terms of number of users and participation which are only available on commercial platforms. Therefore, the common approach in research is to make use of crawled playlists which were manually created by users. As presented in [7], this evaluations are comparable to user studies but require much less effort. Additionally, offline experiments can be easily repeated with adapted implementations and input parameters which helps during implementation and optimization to observe and benchmark the outcome.

Manually curated playlists can easily obtained, they have an user context and are usually labeled with a name. In a broader sense the contained tracks can therefore be seen as the desired output for queries by the name, related to the user. This is why the *playlist evaluation*, further described in section 6.2, uses the playlists as ground truth and tries to predict the tracks based on user context and playlist name. To evaluate the embedding quality in respect to personal preferences the *track recommendation evaluation* in section 6.3 compares user track recommendations to baseline methods.

6.1 Dataset and graph generation

The initial dataset was constructed from a crawled Spotify playlists by DBIS at the University of Innsbruck [14]. This data consists of playlists with an user context and contained tracks with artist and audio features. In order to enrich the available query terms and gather more graph structure, social curated track tags were crawled from *Last.fm* [2] and artist genres from Spotify.

Because both the playlists and the *Last.fm* tags where unfiltered user produced data, preprocessing was necessary. All playlists with less than four tracks or without an alphanumeric character in the title are removed. To match same tags on different tracks, the names are transformed to lowercase and special characters are removed. Tags with less than four characters, or without more than five user assignments on *Last.fm* are also cleaned.

Table 1 and 2 show which and how many elements and connections are contained in the graph after preprocessing. Within the dataset 12 tracks are most common for playlists.

Table 1: Node count by type:

Type	Count
Playlists	21,336
Users	1,180
Tracks	852,293
Artists	110,377
Tags	395,587
Albums	189,174
Genres	1,520
Total nodes	1,571,467

[2] https://www.last.fm/api/show/track.getTags

Table 2: Edge count by types (undirected):

Type	Count
Playlist-User	21,323
User-Tracks	1,662,605
Track-Album	852,293
Track-Artists	1,027,918
Track-Tags	9,341,603
Artist-Genre	148,705
Total edges	13,054,447

6.2 Playlist evaluation

To predict playlist tracks based on the playlist title, two steps are required. First a query must be constructed from the title and then this query can be used to retrieve recommendations. For the query extraction, terms in the title must be matched against known item titles. To do so, the full-text search capabilities of *Elasticsearch*[3] are used. In the training phase all graph nodes except users and playlists are inserted into the database and then available by their title as query terms.
To evaluate the performance, information retrieval measurements precision@k and recall@k are used. For baseline comparison also a random track recommender is used.

6.3 Track recommendation evaluation

In the presented model, queries are constructed using seeding elements combined with user context. Without seeds, the user alone can be used to query for recommendations. To perform a classic evaluation on user track recommendations, the playlists were used to construct historic track listening data, which is split into a training and test set per user. Using the training data, a new graph is generated and embedded. Then the users of the test set are used as queries to retrieve nearest neighbors in the embedding and compared with test tracks. To measure the performance precision@10 and recall@10 are computed and compared against baseline scores using *MyMediaLite* [4]. Without personal context, the *Random* method returns random items and the *MostPopular* predicts tracks with the most overall listening counts. Whereas *UserKNN* uses user-based collaborative filtering to predict k-nearest neighbors tracks.

7 RESULTS AND DISCUSSION

As it can be seen in table 3, the embedding can be used to produce personalized track recommendations which perform better than non-personalized methods. Because this test data was constructed with the playlists, it confirms that playlists can be seen in a user context.

[3] www.elastic.co/products/elasticsearch

Table 3: Track recommendation results:

Recommender	Precision@10	Recall@10
UserKNN	0.10468	0.0201
geMsearch	0.08827	0.003
MostPopular	0.01872	0.00175
Random	0.00017	0.00001

The results of the playlist recommender are listed in table 4. Compared with the random track recommender, the performance does clearly outstand. Furthermore they are better than the track recommendations without seeding items.

Analyzing playlists without hits makes it clear that many playlist names are noisy and do not describe the contained tracks which makes it hard to predict the content. Furthermore about 48% of the playlists contain tracks only from one artist. A nearer inspection shows, that often playlists are used to store albums or best-of collections of artists. As a consequence pure text-based methods on the same dataset can produce better results [3]. In contrast, the here presented approach *geMsearch* is designed to discover new music. Therefore the retrieved list of tracks is composed by a diversity of artists. Even the search results for a given artist does not guarantee to contain songs of the same artist in the top results. To reflect this desired outcome in the test data, the playlists are split into two disjunctive sets based on whether they contain tracks from multiple artists or not.

Table 4: Playlist recommendation results:

Recommender	Precision@1	Precision@10	Recall@10	Precision
UserKNN	0.10468	0.0201		
geMsearch	0.08827	0.003		
MostPopular	0.01872	0.00175		
Random	0.00017	0.00001		

As expected, more structural data provided through the graph creates better embedding and therefore more meaningful results. Especially tags assigned to tracks improved the precision scores for the playlist track prediction by XX%.

8 CONCLUSION

This work presented an approach to use graph embedding techniques to create a low dimensional vector space of music data. This embedding is used to create query-bases music recommendations and evaluated against playlist track predictions. Combined with a 3D representation of the result item it improves the way how user find and explorer new music. This method is not limited to music and may be also used in different domains where application data can be represented as graph but metadata for single items is sparse.

There is still potential for future work in order to improve the embedding itself and also the query mechanism. Weighted graphs seemed to be a promising approach to improve the embedded proximities in early tests. It could also be possible to even include audio features as graph nodes.

REFERENCES

[1] [n. d.]. About Spotify. https://press.spotify.com/us/about/. ([n. d.]). Accessed: 2017-11-02.

[2] Chih-Ming Chen, Ming-Feng Tsai, Yu-Ching Lin, and Yi-Hsuan Yang. 2016. Query-based music recommendations via preference embedding. In *Proceedings of the 10th ACM Conference on Recommender Systems*. ACM, 79–82.

[3] Chia-Hao Chung, Yian Chen, and Homer Chen. [n. d.]. EXPLOITING PLAYLISTS FOR REPRESENTATION OF SONGS AND WORDS FOR TEXT-BASED MUSIC RETRIEVAL. ([n. d.]).

[4] Zeno Gantner, Steffen Rendle, Christoph Freudenthaler, and Lars Schmidt-Thieme. 2011. MyMediaLite: A Free Recommender System Library. In *Proceedings of the 5th ACM Conference on Recommender Systems (RecSys 2011)*.

[5] Palash Goyal and Emilio Ferrara. 2017. Graph Embedding Techniques, Applications, and Performance: A Survey. *arXiv preprint arXiv:1705.02801* (2017).

[6] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 855–864.

[7] Iman Kamehkhosh and Dietmar Jannach. 2017. User Perception of Next-Track Music Recommendations. In *Proceedings of the 25th Conference on User Modeling, Adaptation and Personalization*. ACM, 113–121.

[8] Peter Knees, Markus Schedl, Tim Pohle, and Gerhard Widmer. 2007. Exploring music collections in virtual landscapes. *IEEE multimedia* 14, 3 (2007).

[9] Paul Lamere and Douglas Eck. 2007. Using 3D Visualizations to Explore and Discover Music.. In *ISMIR*. 173–174.

[10] Jin Ha Lee, Yea-Seul Kim, and Chris Hubbles. 2016. A Look at the Cloud from Both Sides Now: An Analysis of Cloud Music Service Usage.. In *ISMIR*. 299–305.

[11] Janis Libeks and Douglas Turnbull. 2011. You can judge an artist by an album cover: Using images for music annotation. *IEEE MultiMedia* 18, 4 (2011), 30–37.

[12] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[13] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 701–710.

[14] Martin Pichl, Eva Zangerle, and Günther Specht. 2017. Improving Context-Aware Music Recommender Systems: Beyond the Pre-filtering Approach. In *Proceedings of the 2017 ACM on International Conference on Multimedia Retrieval*. ACM, 201–208.

A IMPLEMENTATION DETAILS

The implementation can be structured into two main components. The data management, graph embedding and recommendation computation is implemented as a Python application. With a REST interface this services are exposed and decoupled from the second component, the web client.

A.1 Python application

The actual recommender is implemented in Python because many packages for data processing and machine learning exists. In addition also the reference implementation of *Deepwalk* and the tool *Word2vec* are written in Python. To achieve reasonable runtime performance, big data structures are stored and accessed with the Python package *numpy* and matrix computations are performed with *scipy* which both rely on native implementations.

A.2 Data management

The main data source for geMsearch is the Spotify API where DBIS has already crawled a big dataset [14]. Because this data is stored as JSON, the NoSQL document store *MongoDb* is used to store all crawled data. This makes it easy to create data subsets for testing, performing statistic analyses and retrieving metadata to enrich the search results. Synchronized user music libraries are also stored here. For embedding and evaluation required data is extracted and stored as CSV files in an intermediate step. This makes is easier to process or split data and repeat experiments with same sub-data inputs. After potential training-test splits are applied, the music graph is constructed. When adding new data to the graph a mapping for item ids is applied which makes sure that each node is identified by a unique integer id. This continuous ids are required for the graph embedding algorithms as input and later to transform an embedding index back to the original item. Additionally they make sure that same items, e.g. text equal tags, are represented as a single node in the graph.

Each item which have a name except playlists and users are inserted into a *Elasticsearch* full-text index. For the evaluation based on playlists, this service is used to extract query terms from the playlist title. The client supports users while formulating queries with an autocomplete function which is also provided by the Elasticsearch index.

A.3 Webclient

The implemented Webclient makes it possible for users to formulate queries and explorer recommendations. Without the need for an installation or additional setup, the web application has many advantages over traditional desktop applications. *TypeScript*, a programming language superset of JavaScript, was chosen for the implementation because it provides static type checking during compilation. Created and maintained by Microsoft it also enables strong autocomplete suggestions in their editor *VS Code* which improves the development process.
The JavaScript framework *React* helps to maintain the client state and having a virtual DOM enables to program on a more abstract level as no direct DOM manipulations have to be applied. The whole client is a standalone application and communication with

the Python API to retrieve search results and metadata is done via REST interface. Hereby both components are independent and additional or different clients could be possibly introduced.

Beside the list view, results can also be explored in a 3D scene where each item is represented as single interactive object. For performance reasons this scene is rendered in a canvas element using WebGL which is hardware accelerated on most devices. WebGL has currently many crossbrowser issues in different browsers and requires to write shader codes for simple visualizations. The JavaScript library *ThreeJS* fixes this issues with a common API and simplifies the development with many utility methods.

For exploring the search results, users can modify the scene camera position using their mouse and navigate through the 3D space. The whole embedding is too big as it could be transfered complete to the client. Therefore only the most accurate results are returned for queries and additional items are loaded step by step. After each position change, the camera direction is unprojected to get the focused 3D position. Having the new center, additional elements can be queried and added to the existing scene.

A.4 Spotify Connector

As described in section 5, a Spotify account can be used to get personalized recommendations. When a user connects, access to the username and the personal music library is granted. The OAuth protocols allows to retrieve this data as a third party application without knowing the users login credentials. Only a token is transmitted which authorizes API request for a limited time.

After the user has connected, it is checked if he is already known in the system. For new users, the token is send to the server in order to synchronize the music library. In the database this user data is only identified by the hash of the username and prevents backtracking of personal information.

Besides the API there are two microservices on the server which execute long running tasks and prevent to block resources for further requests:

- The **Crawler service** watches the database for new tracks which are inserted through the music library synchronization. It makes sure that all necessary metadata is available. For new items, track and artists are crawled from Spotify and tags for tracks are retrieved from Last.fm.
- The **Embedder services** waits until crawlers are finished and then extends the existing graph with new the data. This task is executed periodically and therefore may embed multiple users at once. Changes are collected and then the existing model is retrained to retrieve a new embedding which then replaces the existing one.