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

Due to the rise of music streaming platforms, huge collections of music are now available to users on various devices. Within these collections, users aim to find and explore songs based on certain criteria reflecting their current and context-specific preferences. Currently, users are limited to either using search facilities or relying on recommender systems that suggest suitable tracks or artists. Using search facilities requires the user to have some idea about the targeted music and to formulate a query that accurately describes this music, whereas recommender systems are traditionally geared towards long-term shifts of user preferences in contrast to ad-hoc and interactive preference elicitation. To bridge this gap, we propose geMsearch, an approach for personalized, explorative music search based on graph embedding techniques. As the ecosystem of a music collection can be represented as a heterogeneous graph containing nodes describing e.g., tracks, artists, genres or users, we employ graph embedding techniques to learn low-dimensional vector representations for all nodes within the graph. This allows for efficient approximate querying of the collection and, more importantly, for employing visualization strategies that allow the user to explore the music collection in a 3D-space. Based on a dataset with over 1,5m graph nodes, we show that the performance of our recommendations is comparable to standard matrix factorization techniques and that query-based results can be created.

KEYWORDS

Recommender Systems, Graph Embedding, Personalization

1 INTRODUCTION

In recent years, music streaming platforms have become a central means for listening to music as these allow users to access huge collections of music. This evolution has also influenced the way users search and explore music. For instance, the streaming platform Spotify currently serves 140 million active users and provides a collection of more than 30 million songs[1] (as of June 2017). Consequently, the primary objective for users has shifted from retrieving specific songs to finding and ultimately exploring songs that match certain criteria reflecting the user's current preferences and context [7, 11].

[1]http://press.spotify.com/us/about

Currently, two paradigms allow users to explore large music collections: search and recommender systems. Utilizing naive search approaches based on simple attribute matching requires the collection data to be fully annotated with metadata. When relying on keyword search facilities, the user is required to have some idea of his/her current preferences and must be able to formulate a query that actually describes these preferences well. More advanced search facilities are based on content similarities of items (aka "find similar artists or songs") and are rarely personalized. Especially data sparsity and the lacking ability for comparing heterogeneous items (tracks, artists, albums, etc.) makes it hard for such systems to succeed. In contrast, recommender systems propose items that might be suitable for the user (based on some collaborative filtering approach or more complex models. While recommender systems do not require the user to be able to formulate his/her current preferences, the user also is not able to directly influence recommendations by stating e.g., a starting point for his/her explorative search for music matching his/her current preferences (except for feedback mechanisms like relevance feedback and explicit ratings that influence the user model in the long term).

Only very few approaches like, e.g., the one proposed by Chen et al. [1] allow the user to specify his/her current needs and preferences in an abstract manner, where the returned results are jointly based on the query (the user's current information need) and the user's personal music preferences. However, there is still a substantial lack of systems which combine flexible search mechanisms with user interfaces that provide dynamic, exploration-driven visualization strategies for large collections of music.

Therefore, we propose the geMsearch system to bridge this gap in explorative music search. In particular, we propose to use graph embedding techniques for computing latent representations of items contained in the graph, such as tracks, users, artists, genres or acoustic features of tracks. Using such graph embedding techniques [18], a low-dimensional latent vector representation is learned for every node. These firstly allow to create advanced search facilities as search queries can be encoded in the same vector space. As a result, not only exact results can be retrieved, but also similar items and hence, exploiting previously unknown similarities between heterogeneous items that can be utilized to retrieve diverse search results. Secondly, the obtained vector representations can be exploited for advanced visualization paradigms enabling explorative music search beyond traditional list-based aggregations of search results that only provide a one-dimensional view of the retrieved items.

Real world applications usually generate new data during runtime which requires to update previously learned models. To optimize this process and avoid the recreation of models from scratch, we present an extension to the Deepwalk algorithm which allows to extend initially learned embeddings with new graph structures such as vertices and edges on the fly.

To evaluate the proposed system a Spotify dataset containing 852,293 tracks is used to perform user-track recommendation and to predict the contained tracks of playlist based on its title.

The remainder of this paper is structured as follows. In Section 2, we describe related work and graph embedding. Section 3 presents geMsearch a system for personalized music search. The visualization prototype for explorative music search is proposed in Section 4. Then Section 5 describes the evaluation setup which results are discussed in Section 6. We conclude the paper in Section 7 by summing up key aspects and detailing future work.

2 RELATED WORK AND BACKGROUND

In this section we first recap graph embedding in general and then describe related work from two fields: query-based recommender systems and approaches which focus on user interfaces for the exploration of new music.

Graph embedding techniques aim to transform graph structures into a low dimensional vector space. More formally, given a graph $G = (V, E)$ with vertices V and edges E a graph embedding is a mapping $f : v_i \to y_i \in \mathbb{R}^d \; \forall i \in [n]$ such that each node in the initial graph is mapped to a vector representation. This resulting vector space has a dimension $d \ll |V|$ and the function f preserves some proximity measure defined on graph G [4]. 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 $O(|V|)$ [4]. The short random walks over edges are used to generate sentences which reflect the graph structure. Using these walks as training data, a representation for each node (word) is learned. In the field of natural language processing this method is known as word embedding and especially popular with *Word2vec* [13]. Both *Deepwalk* [15] and *node2vec* [5] are making use of *Word2vec* in their algorithms and 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 (cf. 3.3).

Recently, graph embedding techniques have also been introduced to the field of music information retrieval. Chen et al. [1] utilize graph embeddings for realizing a query-based music recommender approach that is similar to the one presented in this paper. The main difference is that the music graph was modeled by Chen 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. However, item similarities are only constructed through collaborative filtering without content relationships because music items themselves are not connected in the initial graph.

Chung et al. [2] utilize a pure text-based music retrieval on the same dataset we are using to predict the content of playlists with their title. A common latent representation of words and songs is learned with unsupervised learning based on the co-occurrence of tracks and words in playlist titles. However, in this model only tracks are included, proximities are based on playlists and the construction of queries is very limited.

For the task of building visualizations for music exploration, there are a number of relevant approaches, mostly based on proximity-preserving dimension reduction techniques.

The Islands of Music interface [14] incorporates rythm descriptors and employs self-organizing maps for visualizing music collections based on the metaphor of geographic maps in two-dimensional space. One highly relevant extension of these maps is a browsable 3D landscape by Knees et al. [9], where tracks are clustered based on content features. Hamasaki and Goto [6] propose Songrium, a collection of visualization and exploration approaches. These include the "Music Star Map", a visualization of songs in a graph, where placement of songs is based on audio similarity. Also, Lamere et al. [10] presented a 3D interface (Search Inside the Music) based on Multidimensional scaling techniques to visualize similarities between tracks, where each item is represented as a single colored item in the 3D space. Similarly, the Music Box visualization approach relies on Principal Component Analyses to visualize tracks, where song similarity is used to distribute tracks on a plane. The visualization proposed in this work differs from these approaches in the fact that we base the visualization on latent representations of items within a heterogeneous graph that includes tracks, artists, albums, genres, etc. Due to the applied graph embedding techniques, proximities within the graph visualization are not restricted to similarities between items of the same type (e.g., tracks) or similarities based on a single set of features (e.g., audio features), but rather capture the similarity of items of any type in the latent feature space.

3 GEMSEARCH: PERSONALIZED MUSIC SEARCH

In the following two sections, we present the geMsearch system, a first prototype for personalized explorative music search based on latent representations of nodes of the musical ecosystem[2]. *geMsearch* stands for **g**raph **em**bedding based music **search** and consists of two main components: the graph embedding and retrieval engine that computes latent representations of items and query results, and the client providing a search and visualization interface which is described in Section 4.

3.1 Graph Embedding and Retrieval Engine

A music corpus can be modeled as 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 latent representations for each item can be learned. Higher order proximities are preserved and hence, distances in the embedding represent similarities between items.

[2]The prototype can be accessed at http://dbis-graphembeddings.uibk.ac.at

This is very powerful because previously missing metadata is compensated, and also heterogeneous items can be compared.

A common task for music discovery is to retrieve similar items for one item which is given as an example. Here, simply the nearest neighbors of this seeding item within the embedding must be computed. Each node of the initial graph could possibly be used as query and consequently also tags or artists for example serve as seeding items. After retrieving similar results, a simple post filtering allows to restrict the item type to only return tracks for example. Moreover, multiple items can be combined to construct complex queries which allows to express the desired outcome in a fine granularity. To evaluate a given query on the embedding, only a vector for the nearest neighbor search is required. This means that the latent representation of each query term has to be combined and could possibly be used as positive or negative example with different weights of impact.

If user feedback is available, e.g., through historic track listening behavior or positive feedback on items, the user and relationships to items can be included into the graph. This improves the available graph structure and therefore may improve the embedding quality (through collaborative filtering) and additionally makes it possible to model a user preference on queries. After embedding the user itself has a latent representation in the same vector space with all other items. Under the assumption that proximities have been preserved, his/her consumed and preferred items are positioned nearby this vector. Constructing a query with the user and no additional seed items retrieves general recommendations. For each user-initiated search query, the user's latent representation is added and hence, long-term preferences partly influence the outcome. To limit this effect to a certain margin, the user's vector has to be downscaled.

Equation 1 formalizes how to create a search vector q which is evaluated to retrieve nearest neighbors on embedding f_θ. The short-term query intension may be expressed by multiple graph nodes $x_0, x_1, ..., x_n$ and is influenced by general preferences of user u. Each item (query nodes and user) is transformed with the embedding to retrieve the latent representations. The final vector is then produced by the weighted mean ($\alpha_{x_0} + ... + \alpha_{x_n} + \alpha_u = 1$).

$$q := \underbrace{\alpha_{x_0} * f_\theta(v_{x_0}) + ... + \alpha_{x_n} * f_\theta(v_{x_n})}_{query\ intension} + \underbrace{\alpha_u * f_\theta(v_u)}_{user\ preference} \qquad (1)$$

For the creation of the graph underlying our approach, we rely on the Spotify playlist dataset by Pichl et al. [16], containing 852,293 tracks crawled from public Spotify playlists. Because from this dataset derived embeddings only reflect proximities which are represented by the graph we also added user created semantic information. For this we crawled Last.fm tags[3] for the contained tracks. They enrich the available item descriptors which users can use in their search queries. The resulting dataset is represented as a graph containing undirected edges between the following item types: user–track, track–tag, track–album, album–artist and artist–genre. For the computation of latent representations of nodes via graph embedding, we rely on the popular Deepwalk algorithm [15], where we learn representations for all nodes in a 128-dimensional vector space. The resulting latent representations provides means for flexibly computing similarities between heterogeneous items such as tracks, users or artists.

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

3.2 Search refinement

geMsearch allows users to interactively explore the music space to find new music. Therefore, a starting position for browsing through the items has to be determined by eliciting the user's current musical preferences. As can be seen in the top left corner of Figure 1, a text input field (with autocompletion support) allows to select multiple items from the dataset to construct a query that reflects the user's current preferences. Here, the search query for artist "Jimi Hendrix" may return similar and suitable artists, tracks or tags. In addition, the search result can further be restricted by adding further search terms. In Figure 1, the tag "guitar" is entered and combined with the first term.

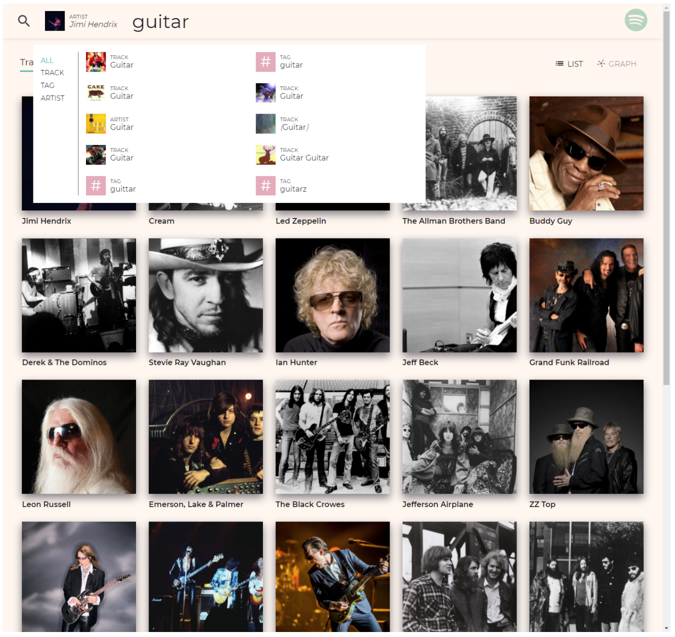


Figure 1: geMsearch query bar with autocomplete and list results.

Searching for music is not a single action process where a user formulates his information needs 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 are also supported with suggested terms for adjustments. The user should feel like navigating through a virtual result space instead of jumping to unconnected places after manually modifying requirements. In the underlying latent vector space, any of the proposed items can be used to further extend the query and hence, refine the search to match current preferences more precisely. This means that any of the proposed items which are retrieved after the initial query can be used to further extend the query and refine the search.

For example, suppose that you inspected your first results of Figure 1 and your search intention also matches "Jeff Beck". Then you can either consume songs directly from this artist or adding him to the current query. This will not limit new results to "Jeff Beck" but will return items which are similar to both artists. Using this technique, the user has not actively adapt the query through reformulating it and still gets more precise results.

3.3 Model extension

In real world scenarios, the dataset represents a dynamic system, constantly changing because new songs or users are added. 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. To reflect this changes in the recommendations, this adaption has to be applied to the model. Especially if initial data is available for new users, e.g. through connecting with other services, a fast method is desirable to alleviate the cold start problem such that the system can be used right away. A naive approach could simply recreate the whole embedding from scratch, but this is not scalable for bigger sets. To solve this issue, we present an extension to the Deepwalk [15] algorithm which can include additional graph structure after creating an initial model.

Random walk based embedding techniques are mostly online algorithms [4][5] which can consume new walks during training as they are produced. This property is very powerful in general because for huge datasets, not all walks have to be generated in advance or even kept in memory. However, the probability distribution of random walks in *node2vec* is not uniform and precomputed before walk generation based on the graph structure. This implies that neither edges nor vertices can be added or removed after the initial computation because it would invalidate previous transition probabilities.

On the contrary, *DeepWalk* uses a uniform distribution over random walks and therefore allows graph structure extensions in theory. The presented algorithm and reference implementation does not offer this functionality but uses *Word2vec* of Gensim [17] to compute the actual embedding. Using *Word2vec* in the backend makes it possible to store the current internal skip-gram model, later restore it for further learning with new sentences (walks) and additionally allows to extend the current vocabulary (vertices) during runtime. Combined the desired options are available to partially extend the existing graph and retrieve new embeddings.

To extend an existing embedding two datasets are required: added vertices and new random walks. To retrieve this sets, first the initial graph is extended by the new graph structure while the system keeps track of modifications. In the next step, random walks are generated to reflect the added data in the same way as the initial walks. Here only walks which contain new edges or vertices have to be created. 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 the existing *Word2vec* model can be loaded and extended with the new nodes which initializes the embedded vectors with random values for those indices. Finally, learning is continued with the new random walks to retrieve an extended embedding which includes latent representations for added nodes and potential adapted vectors for existing nodes.

With this method, graph and embedding can be updated with lower effort than relearning the complete model. However, 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 may impact all items contained in the graph which invalidates all created indexes. Furthermore, only graph structure can be extended but not modified or removed.

geMsearch uses this live model updates to alleviate the cold start problem for user profiles by connecting with existing Spotify accounts. The official Spotify API supports the OAuth protocol with different scopes, allowing access to, e.g., personal playlists, playing history or saved tracks. To create a personal preference profile, *geMsearch* retrieves the user's saved tracks as they may serve as a strong indicator for preference. After a user has connected with his/her account, the user's music library is loaded and compared with the current contents of the underlying graph. For tracks, artists, etc. that are not yet contained in the underlying graph, we gather the missing metadata from Spotify and user-curated tags describing these items from Last.fm. After the data is collected, the graph is extended with this new information to generate additional random walks. In the next step, the existing Deepwalk model is expanded and learned with the presented algorithm to compute and obtain a new latent representation which finally replaces the existing one.

4 VISUALIZATION

The most common visualization for both recommendation and search results is to display a list of items ordered by the predicted relevance of the individual items for the user. This limits users to only observing the sequential order of items and hence, a one-dimensional view agnostic to distances between consecutive items. With a latent feature space underlying the system (obtained through, e.g., graph embedding techniques), similarities between arbitrary items can be expressed which permits developing more advanced interfaces. Through recent advances in browser technology, like the availability of native WebGl, just-in-time visualizations of 3D scenes can be created directly on websites without complex precomputations or add-ons.

Using dimension reduction methods, the computed high-dimensional latent representations can be reduced to three dimensions, allowing to directly visualize items while preserving proximity. Here, we utilize principal component analysis to reduce the 128-dimensional representation of items to a three dimensional space. Instead of displaying a list of items, the recommended items can now be visualized in a 3D scene. Each track, artist or album can be positioned using its three-dimensional representation and can hence be displayed as an interactive 3D object. The positions and resulting distances reflect the relationships and proximities between items within the music collection. Beside the traditional list view for search results, the gemSearch client visualizes the surrounding items in a 3D WebGl scene as depicted in Figure 2. Using such an interface does not only allow to express distance between items, but, more importantly, it allows the user to explore and browse

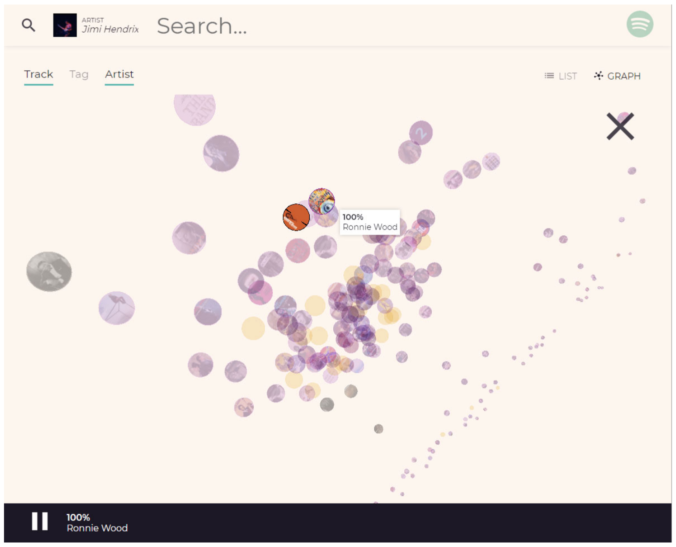


Figure 2: Web client 3D view and player bar.

through the result space interactively. Mouse gestures allow for exploring the virtual space and while navigating, additional items are lazy-loaded into the scene.

The user may first use a keyword search to express his/her current preferences (cf. section on Graph Embedding and Retrieval Engine). Based on these criteria, the first search results are retrieved and displayed in a 3D space. Besides the active manipulation of the search query which was described in Section 3.2, the 3D scene provides an even more effective process of implicit refinement. The most relevant search results are positioned around the center of the screen. When exploring additional items further away, the user has to opt for a direction in which to continue exploring. After inspecting items based on their album covers or through listening to music samples for tracks at the new position, the navigation direction can be refined. If the user detects suitable items, the direction is correct; otherwise the user will navigate in a different direction. This choice of directions and moving within the virtual result space directly translates to (implicit) query refinement.

It is crucial to simplify the inspection of single items such that huge collections of music are explorable in reasonable time. We use album covers as textures for 3D objects describing track and album items and hence, also allow for visually inspecting node textures as this has shown to be an efficient means for judging the relevance of albums and tracks [12]. To provide detailed information about selected items (e.g., artists of a given track, genres, etc.), information from the underlying graph is retrieved and displayed. Furthermore, the provide music samples for each track that allow users to immediately consume newly discovered tracks.

As similar items are located in close proximity to one another in the resulting space, distance-based clustering techniques can be applied to represent accumulations of items as annotated clusters. This allows users to decide whether a set of items might be of interest by looking at the characteristics of the cluster and not having to inspect the individual items contained in the cluster. However, zooming in into a cluster to inspect the individual contained items is still possible. Figure 2 shows how clusters of similar items are represented as single orange circles. On click, the contained items are shown while all other elements are faded with transparency to enhance the contrast. As items within a cluster are positioned nearby, the scene is zoomed in without scaling the circle sizes to avoid overlapping elements.

5 EXPERIMENTS

Estimating the quality of the presented approach is one of the key challenges because there exists 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 user 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 [8], these evaluations are comparable to user studies. 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 be easily obtained from public platforms. They contain multiple song tracks sharing some common characteristics and are usually labeled with a short textual description to describe its content. In addition, they are associated to the user who has chosen the collection. In a broader sense the title of playlists can therefore be seen as queries and the contained tracks as the desired personalized results.

As there is no direct test dataset for geMsearch, we relied on playlists to evaluate two aspects of the system. One aspect is the general embedding quality in respect to personal preferences. Here we transform the playlists to historic listening data and predict one hidden track per user. This evaluation is further described in Section 5.2. To incorporate the query mechanism with seeding elements we use the playlist title and contained tracks as ground truth. As then explained in Section 5.3, the *playlist evaluation* tries to predict the tracks based on user context and playlist name.

5.1 Dataset and graph generation

The initial dataset was constructed from crawled Spotify playlists by Pichl et al. [16]. This set consists of playlists with hashed user id and multiple tracks with artist and audio features. To enrich the available query terms and gather additional graph structure, we extended the dataset with socially curated tags on tracks crawled from *Last.fm* [4] and artist genres from Spotify.

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

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

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

Because both the playlists and *Last.fm* tags are unfiltered user produced data, preprocessing was necessary. The playlist evaluation has to extract query terms from playlist titles. Hence, all playlists without at least one valid term were removed. We defined the smallest meaningful term to consist of either at least three alphanumeric characters or two numbers (e.g. the term "80" could possible label tracks which were produced in the 1980s). Assuming that a playlist with less than four tracks represents only an incomplete list and therefore no processable information, this data was removed.

To match same tags on different tracks, the tag names are transformed to lowercase and special characters are removed. All tags which did not match the length requirement for query terms or with less than five user assignments on *Last.fm* are discarded.

Album and track titles from Spotify always satisfied those restrictions.

In total 1,571,467 vertices and 13,054,447 are contained in the resulting graph after preprocessing which is further listed in Table 1 and 2. Within the dataset, the mode of the number of tracks in playlists is 12.

5.2 Track recommendation evaluation

In geMsearch, queries are constructed with seeding elements combined with user context. Without seeds, the user node alone can be used to query for recommendations which are only based on his long-term preferences. To perform a classic evaluation on user track recommendations, we use the playlists to construct historic track listening data, which is split randomly into a training (80%) and test set (20%) per user with more than 10 tracks. Using the training data, a new graph is generated and embedded. Then for each user, recommendations which are relevant and new to the user are calculated and compared with tracks in the test set of this user. To retrieve those tracks, first the user serves as query to retrieve nearest neighbors of its latent representation in the embedding. Then these items are iterated in ascending order by their distance to filter all other items which are not tracks and to remove known samples from the user's training set. This is continued until the required amount of k elements is found and finally the results are returned.

To measure the performance precision@10 and recall@10 are computed and compared against three baseline scores using *MyMediaLite* [3]. 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 neighborâĂŹs tracks.

5.3 Playlist evaluation

To predict tracks of playlists based on their playlist title, two steps are required. First, a query must be constructed from the title and then this query can be used to retrieve relevant recommendations. For example, suppose the playlist title "Tommy's best of 80s rock classics" which may contains a personal best-of collection of rock songs which were produced in the 1980s. Analyzing the title multiple different queries are possible. As the embedding contains tags, the transformation could extract the tags "80s", "rock", "classics" and "rock classics". But there may also exist an album with the name "80s rock classics" which would potentially result in different recommended tracks. For this task of query term extraction, the full-text search capabilities of *Elasticsearch*[5] are used. Having such high-level full text search, fuzzy matching and proximity queries contribute to match terms even if they are not syntactical equivalent.

Before running the evaluation, the total set of playlists is randomly split into a training (80%) and a test set. Each user-playlist relation in the training set is used to create user-track edges in the graph which models the user preferences. With all available metadata, the graph is completed and then the embedded is computed using Deepwalk. In the evaluation phase, the title for each playlist together with the user context is combined to recommend potential relevant tracks. Those results are matched against the actual playlist tracks. For the query extraction, terms in the title must be matched against known item names. In the training phase, all graph nodes except users and playlists are inserted into the Elasticsearch database and are then available by their title as query terms. Different techniques to extract and then combine multiple items for a final search vector are evaluated. The following list explains these methods and illustrates the possible extracted query on the example playlist title "Tommy's best of 80s rock classics":

one query term The first item which is returned by the search for the playlist title is used as single query term.
Produces query: [tag:"rock classic"]

one query term with user Same as "one query term" but the user is added as query term which is scaled with $\alpha_u = 0.3$ to limit the influence of long-term preferences.

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

Produces query: [tag:"rock classic" ∗ 0.7 + user:"playlist user id" ∗ 0.3]

first two query terms The first two results (query extension) from the text-search for the playlist title are used to construct the query.
Produces query: [tag:"rock classic" ∗ 0.5 + tag:"80s" ∗ 0.5]

user Only the user is used as query term.
Produces query: [user:"playlist user id"]

random Random tracks are returned
Produces query: [random item]

To evaluate the performance, information retrieval measurements precision@k and recall@k are used. In addition, the metric leastOneHit@k is calculated which represents the percentage of playlists which had at least one valid hit at k proposed tracks. For baseline comparison, a random track recommender returns k random items for each playlist.

6 RESULTS AND DISCUSSION

As it can be seen in Table 3, geMsearch can be used to produce personalized track recommendations which perform better than non-personalized methods and are comparable to standard matrix factorization techniques. Because the test-listening data of this evaluation was constructed with playlist tracks, it confirms that the content of playlists are influenced by personal preferences.

Table 3: Track recommendation results

Recommender	Precision@10	Recall@10
UserKNN	0.10468	0.0201
geMsearch	0.09468	0.0090
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, methods relying on seeding items perform better than the track recommendations which are only based on user preferences. The best scores for precision and recall are achieved when only one query term without user context is used . The experiments can not benchmark the query extraction and item retrieval separately. Tests on more complex query creations did not succeed because playlist titles are rather short and mostly only one query term is extracted. Therefore, not the full capacity of the geMsearch query facilities are used in this evaluation.

As expected, more structural data provided through the graph creates better embedding and therefore more meaningful results. Table 5 contains the performance measured with precision@10 on running the playlist evaluation with the extraction method "one query term". Each execution was performed on different subsets of the full dataset which contained only the specified item types. Especially tags assigned to tracks and artist genres improved the precision scores for the playlist track prediction up to 60%.

Table 5: Playlist recommendation performance on graphs with less structural data

Subset with item types	Precision@10
TODO: insert real values	
Tracks, Artists, User	0,10
Tracks, Artists, User, Tags	0,10
Tracks, Artists, User, Albums	0,10
Tracks, Artists, User, Genres	0,10
Tracks, Artists, User, Tags, Albums	0,10
Tracks, Artists, User, Tags, Albums, Genres	0,10

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 [2]. In contrast, the proposed system *geMsearch* is designed to discover new music and should therefore retrieve different sets of recommendations for each user. To reflect this desired property in the test data, the playlists are split into two disjunctive sets based on whether they contain tracks from multiple artists or not.

The results for these two datasets, listed in Table 4, show that for playlists with tracks from only one artist the overall performance is much better but also the user as part of the query does not improve the results. It seems that users assign more meaningful titles for such lists (e.g. they keep the album / artist name). This makes it easier to predict the content but also user preferences are less important. On the contrary are playlists with a diversity of artists. All recommender methods are less efficient on this dataset except the recommendations which are only based on the user context. Using one single seeding element is still the best approach for short results lists but as soon as more items should be retrieved (higher k) the long-term preferences improves the scores (precision@10 and recall10 values for "one query term with user" are highest).

Analyzing the percentage of playlists with at least one hit in Figure 3 confirms that personalized strategies are only suitable for longer result sets. In order to predict one single item within playlists, it is more efficient to propose a "classic" example for the extracted query because it is more likely to be contained.

geMsearch produces meaningful personalized recommendations, but the effect in combination with queries is currently not as strong as expected. It is not possible to explain whether the current embedding strategy or query computation lacks this personalization or if the evaluation with playlist data can not reflect the desired outcome. Especially the combination of flexible search facilities and 3D visualizations which is one of the key strengthen of our work could not contribute in this evaluation.

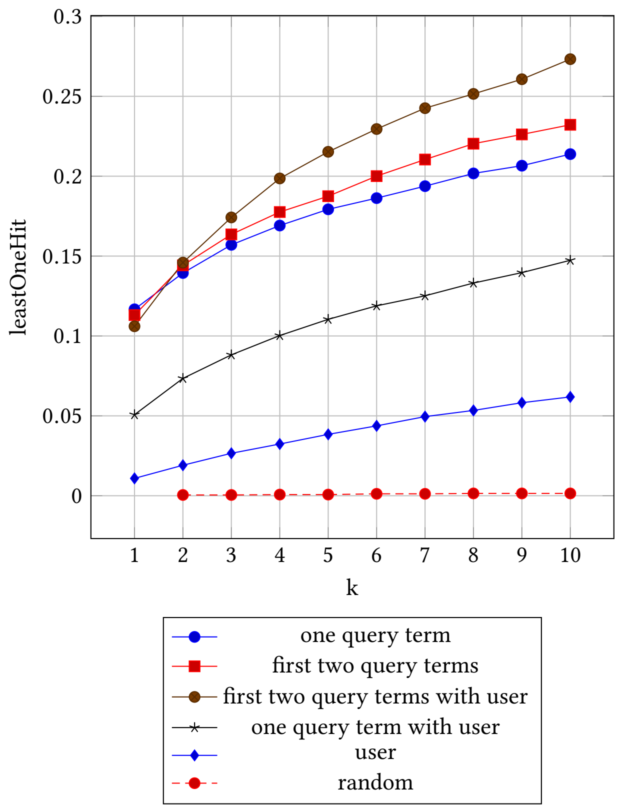
7 CONCLUSION

This work presented an approach to use graph embedding techniques to create a low dimensional vector space of music data. This

Table 4: Playlist recommendation results

Recommender Method	Precision@1	Precision@10	Recall@1	Recall@10
Full dataset				
one query term	0,11669	0,10014	0,00854	0,07117
first two query terms	0,11307	0,10101	0,00814	0,07140
one query term with user	0,08988	0,07637	0,00269	0,01433
user	0,01087	0,00937	0,00024	0,00208
random	0,00000	0,00014	0,00000	0,00002
Playlists with only one artist				
one query term	0,55280	0,46765	0,04258	0,34096
one query term with user	0,53981	0,45963	0,04140	0,33506
first two query terms	0,50706	0,45082	0,03852	0,32804
user	0,00169	0,00147	0,00018	0,00106
random	0,00000	0,00014	0,00000	0,00002
Playlists with multiple artists				
one query term	0,10456	0,07758	0,00495	0,03391
one query term with user	0,09897	0,07896	0,00464	0,03408
first two query terms	0,08477	0,06880	0,00391	0,02920
user	0,01979	0,01863	0,00038	0,00320
random	0,00000	0,00014	0,00000	0,00002

Figure 3: Percentage of playlists with at least one hit



embedding is used to create query-based music recommendations and evaluated against playlist track predictions. Combined with a 3D representation of the result items it improves the way how user find and explore new music. We believe that the proposed 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 to improve the embedding itself and the query mechanism. Weighted which are for example possible in node2vec[5] seems to be a promising approach to improve the embedded proximities in early tests. With them it could even be possible to include audio features as graph nodes in order to introduce audio similarities. However, this would make model extensions more difficult. A user study is eligible to further evaluate the performance on multi term queries and to understand how users may use such flexible search systems.

REFERENCES

[1] Chih-Ming Chen, Ming-Feng Tsai, Yu-Ching Lin, and Yi-Hsuan Yang. 2016. Query-based Music Recommendations via Preference Embedding. In *Proc. of the 10th ACM Conf. on Recommender Systems (RecSys '16).* 79–82.

[2] Chia-Hao Chung, Yian Chen, and Homer Chen. 2017. Exploiting Playlists for Representation of Songs and Words for Text-Based Music Retrieval. In *Proc. of the 18th Intl. Society for Music Information Retrieval Conf.*

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

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

[5] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proc. of the 22nd ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining.* ACM, 855–864.

[6] Masahiro Hamasaki and Masataka Goto. [n. d.]. Songrium: A Music Browsing Assistance Service Based on Visualization of Massive Open Collaboration Within Music Content Creation Community. In *Proc. of the 9th Intl. Symposium on Open Collaboration.*

[7] Mohsen Kamalzadeh, Dominikus Baur, and Torsten Möller. [n. d.]. A survey on music listening and management behaviours. In *Proc. of the 12th Intl. Society for Music Information Retrieval Conf.*
[8] Iman Kamehkhosh and Dietmar Jannach. 2017. User Perception of Next-Track Music Recommendations. In *Proc. of the 25th Conf. on User Modeling, Adaptation and Personalization*. ACM, 113–121.
[9] Peter Knees, Markus Schedl, Tim Pohle, and Gerhard Widmer. 2006. An innovative three-dimensional user interface for exploring music collections enriched. In *Proc. of the 14th ACM Intl. Conf. on Multimedia*. ACM, 17–24.
[10] Paul Lamere and Douglas Eck. 2007. Using 3D Visualizations to Explore and Discover Music. In *Proc. of the 7th Intl. Society for Music Information Retrieval Conf.* 173–174.
[11] 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 *Proc. of the 17th Intl. Society for Music Information Retrieval Conf.* 299–305.
[12] 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.
[13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).
[14] Elias Pampalk. 2001. *Islands of music: Analysis, organization, and visualization of music archives*. Master's thesis. Technical University Vienna.
[15] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *Proc. of the 20th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*. ACM, 701–710.
[16] Martin Pichl, Eva Zangerle, and Günther Specht. 2017. Improving Context-Aware Music Recommender Systems: Beyond the Pre-filtering Approach. In *Proc. of the 7th Conf. on Multimedia Retrieval (ICMR)*. ACM, 201–208.
[17] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. ELRA, Valletta, Malta, 45–50. http://is.muni.cz/publication/884893/en.
[18] Shuicheng Yan, Dong Xu, Benyu Zhang, Hong-Jiang Zhang, Qiang Yang, and Stephen Lin. 2007. Graph embedding and extensions: A general framework for dimensionality reduction. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 29, 1 (2007), 40–51.

A IMPLEMENTATION DETAILS

The implementation can be structured into two main components. As it can be seen in figure 4 the data management, graph embedding and recommendation computation is implemented as a Python application. With a REST interface these services are exposed and decoupled from the second component, the web client which provides a user interface.

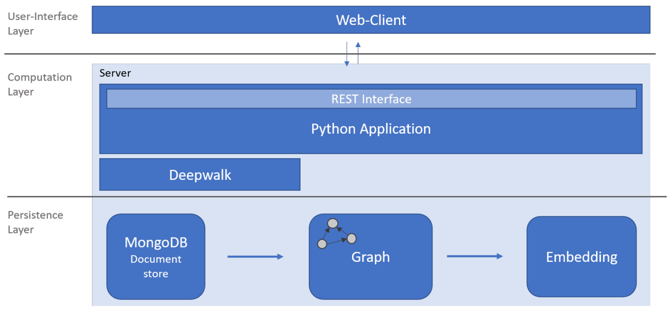


Figure 4: Main architecture overview

A.1 Python application

The actual recommender for computing and storing the required data is done in a Python application. I have chosen Python because many packages for data processing and machine learning exist. 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 [16]. 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 statistical 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 unsigned integer id. These 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 single nodes in the graph and allows to store the data as arrays for fast and efficient access.

Each item which has 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. Also, the client makes use of this index service to provide an autocomplete function for users while formulating queries.

A.3 Computation of recommendations

The interface for recommendations takes a list of object ids as input and returns items ordered by the similarity to this input. Any item of the graph can be used as seeding element. Optional the weights are assignable for each item to specify positive, negative and exceptional items. By default, each item is weighted equally as positive seeding element. To create personalized results, the user can either be specified as additional seeding item or as optional parameter where defaults weights are used ($\alpha_u = 0.3$).

To create a query vector the latent representation for each object ids are fetched and combined as described in equation 1. Then the cosinus distance is computed to each item within the embedding to retrieve nearest neighbors. Ordered by distance to the search vector (similarity) this node indices are traversed and resolved with a lookup to the actual item object. In this step optional element type filters (e.g. restrict results to tracks) remove unwanted items. During this traversal the pagination is applied and stopped as soon as the necessary number of items are found. Finally, metadata like album covers and preview URLs are fetched from the document store before the final set returned.

A.4 Webclient

The implemented Webclient makes it possible for users to formulate queries and explore 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 is done via REST interface to retrieve search results and metadata. Hereby both components are independent and additional or different clients could possibly be 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 writing 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 transferred completely to the client. Therefore, only the most accurate query results are returned 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. To limit the required computational and memory resources the maximum amount of displayed elements is limited. After a fixed threshold previous items are disposed when new items are added to the scene.

A.5 Spotify Connector

As described in Section 3.3, 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 userâĂŹs 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 to 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, the track data 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 inserted into the existing Word2Vec model to retrieve a new embedding which then replaces the existing one.

The client polls for updates on these services and notifies the user as soon as all tasks are done. On subsequent requests the user context can be used to compute personal query results.