Personalized music search based on graph embedding

Christian Esswein

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 about users, tracks, artists, genres, tags, . Using graph embedding techniques a low-dimensional vector representation is learned and provide 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.
The goal of this thesis is to use graph embedding techniques for creating a latent representation of the Spotify music dataset. This low-dimensional vector embedding is used to provide personalized recommendations based on arbitrary search queries. After providing the first search results, the user should be assisted while exploring the results and in the refinement of his search terms. Instead of a list, the learned vector representation should be exploited to generate 3D representations of the suggested items.

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[LKH16]. For example Spotify[1] with currently 140 million active users has over 30 millions songs to offer (as of June 2017) [abo]. 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. Similar songs to / similar to this artist / genre / tags.

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. The ecosystem of music can be represented as an heterogeneous graph using all available data about users, tracks, artists, genres, tags,

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

. Using graph embedding techniques a low-dimensional vector representation is learned and provide 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.

The goal of this thesis is 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 results and in the refinement of his search terms. Traditional list based aggregations of search results can only model a one dimensional similarity 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.

When using naive approaches with simple attribute matching complete annotated metadata is required. Otherwise song can not appear when querying for tag it has not assigned to. Especially when it comes to genres and tags it is unfeasible to manually annotate every entity. Desired search engine needs to learn representation. tags in training phase but while predicting new connections have to be drawn.

2 Related work

music search [CTLY16]

3D interface [LE07]

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 $f : v_i \to y_i \in \mathbb{R}^d \ \forall i \in [n]$ such that $d \ll |V|$ and the function f preserves some proximity measure defined on graph G "[GF17]. The final result is therefore a vector for 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 base methods are interesting (time complexity of $O(|V|)$). 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*[MCCD13]. Both *Deepwalk*[PARS14] and *node2vec*[GL16] are making use in their reference implementations of *Word2vec* 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 user-track relations produced by playing history, 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 (eg through connecting with other services, see 5), a fast method is desirable. A naive approach could simply recreate the whole embedding, but this is not scalable for bigger sets.

The 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 computational model and later restore it for further learning and additional 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.

In order 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 new graph structure can be added with much less effort than relearning the complete model. Unfortunately even small changes can influence the whole embedding and create a complete new vector space. The scalability is therefore questionable because for big datasets the new model 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 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 intension 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 describe 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.

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 general recommendations are retrieved with the same system.

4.1 Search refinement

Searching for music is not a single action process where a user formulates query intension 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 be 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 an vector it can partially constructed as a combination of multiple search term vectors. This means that any of 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 intension. 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, much more advanced interfaces are possible. Using dimension reducing methods 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. 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 is refined while exploring indirectly.

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.

Through advantages in browser technology just in time visualizations of 3D scenes can be created directly on websites without complex precomputations. 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 an retrieved items are represented in a list or 3D view. Besides the offline evaluation this makes it possible to test the implementation in a real world scenario.

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

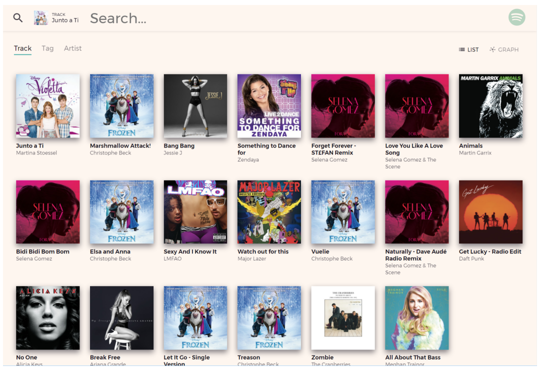

Fig. 1. Web client search suggestions

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. 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.

For sure 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 was 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 somehow needs to allow the users to estimate the results without being biased through available options or the test environment. A/B test 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 [KJ17], 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 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* 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[PZS17]. 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.

Table 1 and 2 show which and how many elements and connections are contained in the graph.

Table 1. Node count by type:

Playlists	24,359
Users	xxx
Tracks	872,521
Artists	110,377
Tags	xxx

Table 2. Edge count by types (undirected):

Playlist-User	21,323
User-Tracks	xxx
Track-Artists	671,903
Track-Tags	83,789

As expected, the more structural data is available the better the results perform, both for the *playlist evaluation* and *track recommendation evaluation.*

6.2 Playlist evaluation

For using the playlists as ground truth, known terms need to be extracted from the name as a first step. match with known terms and construct query. Using this terms a query can be constructed including the user. To evaluate the performance the IR measurements precision@k and recall@k are used.

problems: many names are noisy and do not describe content in any sense. results are compared with random predictor.

–¿ Query expansion

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

6.3 Track recommendation evaluation

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. Using the test data, a new graph is generated and embedded. To compare the results, baseline scores are computed using *MyMediaLite* [GRFST11].

7 Conclusion

This work presented an approach to use graph embedding techniques to create a low dimensional vector space of music data. This embedding was used to create query-bases music recommendations and evaluated against playlist track predictions. Combined with a 3D representation of the result item it improved the way how user find and explorer new music. This approach 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

[abo] *About Spotify*, https://press.spotify.com/us/about/, accessed: 2017-11-02.

[CTLY16] C.-M. Chen, M.-F. Tsai, Y.-C. Lin and Y.-H. Yang: *Query-based music recommendations via preference embedding*, *Proceedings of the 10th ACM Conference on Recommender Systems*, ACM, 2016, pages 79–82.

[GF17] P. Goyal and E. Ferrara: *Graph Embedding Techniques, Applications, and Performance: A Survey*, arXiv preprint arXiv:1705.02801.

[GL16] A. Grover and J. Leskovec: *node2vec: Scalable feature learning for networks*, *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2016, pages 855–864.

[GRFST11] Z. Gantner, S. Rendle, C. Freudenthaler and L. Schmidt-Thieme: *MyMediaLite: A Free Recommender System Library*, *Proceedings of the 5th ACM Conference on Recommender Systems (RecSys 2011)*, 2011.

[KJ17] I. Kamehkhosh and D. Jannach: *User Perception of Next-Track Music Recommendations*, *Proceedings of the 25th Conference on User Modeling, Adaptation and Personalization*, ACM, 2017, pages 113–121.

[LE07] P. Lamere and D. Eck: *Using 3D Visualizations to Explore and Discover Music.*, *ISMIR*, 2007, pages 173–174.

[LKH16] J. H. Lee, Y.-S. Kim and C. Hubbles: *A Look at the Cloud from Both Sides Now: An Analysis of Cloud Music Service Usage.*, *ISMIR*, 2016, pages 299–305.

[MCCD13] T. Mikolov, K. Chen, G. Corrado and J. Dean: *Efficient estimation of word representations in vector space*, arXiv preprint arXiv:1301.3781.

[PARS14] B. Perozzi, R. Al-Rfou and S. Skiena: *Deepwalk: Online learning of social representations*, *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2014, pages 701–710.

[PZS17] M. Pichl, E. Zangerle and G. Specht: *Improving Context-Aware Music Recommender Systems: Beyond the Pre-filtering Approach*, *Proceedings of the 2017 ACM on International Conference on Multimedia Retrieval*, ACM, 2017, pages 201–208.

A Implementation Details

Evaluation: in Pyhton Graph is generated, embedded

ElasticSearch for autocomplete im client and query extraction for playlist evaluation

MongoDb as a document store for metadata in order to enrich search results. User data is also stored here

Client written in TypeScript to have static type checking and advanced autocomplete suggestions in editor React with Redux ThreeJs to render 3D scene

explain spotify connector: having two microservices besides API to not block resources with long running tasks for further requests. Crawler: Watches DB for new tracks through music library sync and makes sure that all neccessary metadata is available. For new tracks spotifiy track data and artists are crawled, then tags are retrieved from Last.fm.

Embedder: Waits until crawlers are finished and then extends existing graph with new data. The changes are collected and then existing model is retrained to retrieve new embedding which then replaces existing.