Generating Music Playlists with Hierarchical Clustering and Q-Learning

James King and Vaiva Imbrasaitė

Computer Laboratory
University of Cambridge
JamesK@cantab.net, Vaiva.Imbrasaite@cl.cam.ac.uk

Abstract. Automatically generating playlists of music is an interesting area of research at present, with many online services now offering "radio channels" which attempt to play through sets of tracks a user is likely to enjoy. However, these tend to act as recommendation services, introducing a user to new music they might wish to listen to. Far less effort has gone into researching tools which learn an individual user's tastes across their existing library of music and attempt to produce playlists fitting to their current mood. This paper describes a system that uses reinforcement learning over hierarchically-clustered sets of songs to learn a user's listening preferences. Features extracted from the audio are also used as part of this process, allowing the software to create cohesive lists of tracks on demand or to simply play continuously from a given starting track. This new system is shown to perform well in a small user study, greatly reducing the relative number of songs that a user skips.

Keywords: Music playlist generation, reinforcement learning, hierarchical clustering, user study.

1 Introduction

We listen to music in a variety of ways. Many people listen to individual albums one at a time, some prefer to listen to tracks in a random order, whilst others opt to create playlists by hand. Each of these options suits different individuals better than others, but all of them come with drawbacks. With the growing popularity of digital music, the motivation for more intelligent *automatic playlist generation* is also growing, as people's music collections become unmanageable.

Many existing solutions to this problem make use of large online databases, or rely heavily on tagged audio files. In this paper we show a solution that removes any dependence on these types of data sources or any external services to achieve a completely personal and independent music player.

The player monitors the user's actions continuously using reinforcement learning and updates its matrices based on user behaviour. Using implicit user behaviour only, our player is able to learn user preferences providing users with a better music listening experience. We show this by executing both a quantitative user study as well as some more qualitative tests.

A. Hanbury et al. (Eds.): ECIR 2015, LNCS 9022, pp. 315–326, 2015.

2 Background

The increasing use of online music streaming services in recent years has seen a surge in research investigating music recommendation and playlisting. Here we focus solely on Automatic Playlist Generation (APG), and while this shares many traits with recommendation, there are some important differences. Music recommendation systems assume access to external data and new songs, while APG systems should be expected to run with only local data. Irregularities in local music collections, along with the lack of external data, make this a hard problem. Furthermore, recommendation algorithms focus on discovery, whereas APG tends to be more concerned with coherence within a set of ordered tracks.

2.1 Commercial Services

Vast online meta-data resources are being increasingly utilised by a wide range of services [2]. Of particular note is The Echo Nest[1], which beyond simple song meta-data provides an array of similarity, personalisation and learning tools for its extensive database. Some interesting 'acoustic attributes' are extracted from audio, such as 'danceability', 'energy', 'speechiness' and 'liveness'. Unfortunately, as it is a commercial project, most of the implementation details used are hidden.

Spotify[2] is a well known audio player that provides a radio feature which creates automatic playlists. However, Spotify is an online service, and currently only learns song similarities across its database of users. It can generate random playlists, but only based on a user's favourites. The primary technique Spotify uses is *collaborative filtering* [8].

iTunes Genius[3], another popular playback tool, is known to use latent factor analysis [10] to extract song recommendations from huge data sets of listening patterns from other users.

2.2 Existing Research

Existing solutions to APG tend to focus on two techniques: Collaborative Filtering (CF) [7,17], and Content-Based (CB) approaches [16], as well as hybrids of the two [21,3]. CF has seen extensive use of online marketplaces, and follows the reasoning that if a user likes A (in our case, a music track), and many other users like both A and B, then we can recommend B to the user. Well known issues with this are dense grouping of tracks by the same artist, and the cold-start problem in which new tracks cannot easily be introduced. CB methods instead look directly at audio data to recommend similar tracks to a user, but this clearly relies on the assumption that audio similarity is a key factor in what a user likes, when many other factors are also involved.

To overcome the problems above, these approaches have often been fused with other data, such as social media information and sensory data (spatiotemporal)

[1] `http://the.echonest.com`
[2] `http://www.spotify.com`
[3] `http://www.apple.com/uk/itunes/features`

on mobile devices [5,19]. These context-aware methods seek to choose tracks which fit with a current situation or mood. Closely tied with this, there has also been a more limited investigation into what a 'good' playlist actually is [9,14], and how we might evaluate whether a given APG tool creates good playlists.

The approach we have taken is to use CB methods to inform an unsupervised machine learning algorithm (Q-learning), which over time can learn from implicit user feedback to generate playlists which are personalised to different listeners. We do not use CF or other methods which require external data. A number of other machine learning approaches have been applied to this problem [6,4], and implicit user feedback methods have been tried before [15], but to our best knowledge the Q-learning with clustering approach taken here is novel.

One final related work has been the creation of automatic 'DJ's [11]. Perhaps most notable of these is the Microsoft Research AutoDJ project. This uses Gaussian Process Regression [18] to learn priors for selecting new tracks. It also looks carefully at the issues surrounding similarity measures, which are a related issue we do not focus on in this paper.

3 Audio Analysis and Clustering

3.1 Feature Extraction

Given the requirement for the generator to not use meta-data, it is necessary to gather information about files from the audio data itself. Much work has been done into extracting meaningful statistics about signals, and many of these apply directly to audio signals in the context of music.

We divide the 16KHz signal into windows of 512 samples and extract 14 features using the jAudio [12] library: spectral centroid, spectral roll-off, spectral flux, compactness, spectral variability, root mean square of the power, fraction of low energy windows, zero crossings rate, strongest beat, beat sum, strength of the strongest beat, Mel-Frequency Cepstrum Coefficients (MFCC), Linear Predictive Coding (LPC) and the statistical method of moments. All values are mean averaged across all windows, forming an array of 39 values in total.

3.2 Clustering

It may not be obvious why we need to cluster the songs before going further—after all, the aim is to create playlists from individual songs and learn the probabilities of transitioning between them. As we are aiming to learn the relationship between individual songs, the number of values we would have to learn for all the transitions would be impractically large for all but the smallest of music libraries (n^2 transition probabilities for a library with n songs). So, the solution is to first cluster the songs, and then learn transitions between these clusters. Provided the clusters are small enough, this will be accurate enough that users cannot tell the difference, and it will reduce the number of learning values to K^2, where K is the number of clusters.

Hierarchical Clustering. Although k-means clustering of songs is a good starting point for tackling the playlist learning problem, it has one obvious deficiency. Namely, there is no universal method for choosing K without generating an infeasible number of clusters, or clusters that are simply too big. If K is too small, we tend towards the K = 1 case in which the clusters contain so many songs that knowing which cluster to choose provides no useful information. As K grows, the number of transition probabilities grows, and so we go back to the original problem where the matrices are too large to learn.

The solution we propose is therefore to use *hierarchical* clusters, in which a tree of clusters is constructed using k-means clustering, and Q-learning (explained below) is performed on nodes at multiple levels. This keeps the benefits of clustering without introducing large transition matrices or large groups of songs to choose between. In fact, this reduces the space complexity from $\mathcal{O}(n^2)$ to just $\mathcal{O}(n)$.

Space Complexity. Consider an arbitrary cluster tree. Let n be the number of songs clustered, and m be the limit set on the number of clusters per node, so that $m = \mathrm{K}$ in terms of the k-means clustering algorithm. In an ideal tree m will also be the branching factor making the tree balanced. Finally, let h be the height of the tree so that a 1-node tree has height 0, the $1 + m$ node tree has height 1, and so on.

Now, we have that $m^h = n$ for a balanced tree, since the branching factor is m. Each node has a transition matrix in this model, so we are interested in the total number of nodes. At level 0 there is $m^0 = 1$ node, at level 1—$m^1 = m$ nodes, at level 2—m^2 nodes, and at the bottom level $m^h = n$ nodes. However, the bottom level nodes are the individual tracks, so we only need to consider up to level $h - 1$ which has m^{h-1} nodes.

If we sum the total number of nodes, we get $S = \sum_{x=0}^{h-1} m^x$, and this is a simple geometric series $S = \frac{m^h - 1}{m-1} = \frac{n-1}{m-1}$. As each node's transition matrix has m^2 entries, the total number of values is therefore the product, $m^2 \frac{n-1}{m-1}$, which is $\mathcal{O}(mn)$. Since m is a small constant, this tends to $\mathcal{O}(n)$, compared with the $\mathcal{O}(n^2)$ complexity we get without clustering.

4 Learning from User Behaviour

4.1 Reinforcement Learning

Markov Decision Processes [1] (MDPs), which are an extension of Markov Chains, are used in situations where a decision maker has some control over choices with random outcomes. An MDP is a discrete-time stochastic control process, in which the agent is in some state s at time t, and must choose an action a which will move the process to a new state s' and give the agent a reward $\mathcal{R}(s, a)$. Since all previous states are ignored under these conditions, MDPs possess the *Markov property*. In addition to the set of states S, an initial state s_0 and a set of actions A, the agent also requires:

- A transition function $\mathcal{S}\colon S \times A \to S$ which gives the probability that action a causes a state transition $s \to s'$
- A reward function $\mathcal{R}\colon S \times A \to \mathbb{R}$ giving the reward (a real number) of choosing action a in state s

In our case the set of states S corresponds to the set of clusters, and not the individual songs. Furthermore, since the clusters are hierarchical, we will need one model per node in the tree, with S being only the node's child clusters. In order to make the choice on the next track, or to update policies based on feedback, we will need to walk the tree and update multiple nodes along the path (the exact solution is described below). Note that this is not the same as hierarchical reinforcement learning, where a problem is solved by abstracting it to multiple levels.

4.2 Q-Learning

Q-learning [20] is a model-free reinforcement learning technique which learns a Q-function that gives the expected utility of taking a particular action a. It is model-free because the agent has no knowledge of the transition function $\mathcal{S}$ or the reward function $\mathcal{R}$. In the music player, the agent does actually know $\mathcal{S}$, since this is just a deterministic function in which a_1 means "transition to s_1", a_2—"transition to s_2", and so on. However, $\mathcal{R}$ is clearly unknown because the rewards are assigned by the user. This is known as active learning, since the agent is learning what to do in each state rather than simply looking for some goal state. Of course, in the music player there is no goal state since it is the combination and path through the states which matters, and playback may continue indefinitely.

Q-learning works with a value iteration update formula:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t(s_t, a_t) \times \Big[R_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)\Big]$$

In our case it can be simplified as follows, with transition matrix P:

$$P'_{r,c} = P_{r,c} + \alpha_t \times \Big[R_t + \gamma \max_a P_{c,a} - P_{r,c}\Big] \tag{1}$$

where r is the matrix row, c the column, and t is the time of the update. The α and R in this formula, though indexed by t for clarity, are functions which may depend on other parameters including the current song choice.

The intuition behind this formula is that when a transition from state r to c receives some reward R_t, we essentially want to change the respective entry in Q by the amount R_t. However, since the goal is really to optimise the long-term gain, there is also a factor based on the maximum attainable reward in the next state c. This is weighted by the *discount factor* γ, and then the whole update is weighted by the *learning rate* factor denoted α_t. The purpose of α_t is to prevent the Q value being set immediately equal to the reward, but instead smooth updates so that both the reward and the current value are taken into account.

So the reward is multiplied by α_t and the old value by $1-\alpha_t$, which explains why α_t is typically fairly low (less than 0.1), since with updates occurring frequently it is desirable to place more weight on the accumulated past updates than on a single new reward.

The value of γ, such that $0 \leq \gamma \leq 1$, is a constant which sets a trade-off for how much immediate rewards are valued compared to future rewards. A value of 0 makes the agent *myopic* and leads to a greedy algorithm since it only considers current rewards, whereas $\gamma = 1$ will make it aim for a higher long-term utility.

Q-learning algorithms often iterate within 'episodes' until local convergence is reached. This method doesn't apply well to the music player scenario, so instead there is just one update per user action. This reflects that rather than having a single clear goal for which an optimum path must be found, we are continually trying to find good states and may continue indefinitely. So there are no absorbing states in the MDP, and its transition graph may contain loops.

Optimal Policies. The Q-learning algorithm is designed to converge to an optimal policy, where a policy is simply the assignment of weights to possible actions for each state (taken from P). The optimal policy here is the 'discounted cumulative reward' which maximises $r_{tot} = \sum_{i=0}^{\infty} \gamma r_{t+i}$, where t is the start time, and we use policy π for infinite time after starting in some state s_t. The r_t are rewards at time t. It is important to notice that this is only defined for a given γ, since clearly a greedy algorithm would have a different sense of optimal reward to a long-term optimiser. However, the definition is unconditional on α, which only affects how quickly and effectively it can learn this optimal policy.

4.3 Calculating Rewards

Since the learning agent can only adapt itself based on the rewards obtained from the user, the design of these rewards is vital to the success of the agent. We want the user feedback to be implicit—avoiding features such as voting buttons whose usage inevitably drops over time and leads to inconsistencies. However, the user's normal behaviour provides plenty of clues about suitable rewards, providing they are reasonably active whilst listening. The design chosen for the reward system for music playback is outlined below. All values are chosen so that $-1 < r < 1$, and generally linear functions are sufficient to interpolate between the two extremes.

Track Skipped or Finished. The basic measure of implicit reward is listening time, an assertion which Chi et al. [6] have provided evidence to support. So, when a track plays all the way through without interruption from the user, a positive reward to it from the prior track is established. In our system we take it to be the maximum reward $r = 1.0$.

The converse of this is how soon a track was skipped, which allows a negative reward to be assigned. It has been shown that even a fairly simple heuristic based on this principle can be effective [15]. With this reward, the earlier a track

is skipped, the greater the negative weight on the reward assigned. This leads us to $r = -1.0$ as the greatest negative reward for a skip after 0 seconds, and $r = 1.0$ as the greatest positive reward for not skipping at all.

So a track finishing is just a special case of this general rule, and we interpolate linearly between the two reward extremes based on how far through the track we got to before the skip.

There is another issue raised by skipping a track early (here defined as within 10s): we should ignore it in all the updates regarding the previous state. So, if the following track then completes, the associated positive reward will 'jump' over the skipped track and relate the song before and the song after the skipped track. In other words, whenever a track is skipped quickly, the previous song should be treated as the current track when future rewards are calculated.

Playlist Rewards. This is a separate class of reward from those mentioned previously, as it does not result from specific user behaviours, and can never be triggered by an action on behalf of the agent. However, a user's existing playlists offer a huge insight into their listening preferences. A small reward should link every song in the playlist (as the songs are meant to be listened to at roughly the same time), with a larger reward for consecutive songs to emphasise the importance of a specific ordering. In practice, the small rewards should only be applied for smaller playlists since the $\mathcal{O}(n^2)$ time requirement can be constraining, and the reward itself becomes negligible.

4.4 Learning with Hierarchical Clusters

As we have mentioned above, to perform Q-learning across a hierarchical tree with many levels, a single update may need to trigger Q-learning updates at several different nodes. Each node is represented by a Q-matrix except for the bottom level, where nodes are leaves holding individual songs.

Firstly, we need to find the lowest common ancestor (LCA) between two nodes. We then define the level multiplier as an exponential function based around the maximum cluster count K of any node, $multiplier = K^{-(LCA-level)}$. This takes the value 1 at the LCA, and rapidly approaches 0 elsewhere with speed dependent on K. Note that this multiplies the learning rate, and not the reward, which would lead to inconsistencies with the updates. The intuition behind this multiplier is that a node shouldn't learn as fast if it is updated more frequently. We want the net rate of learning to be roughly constant across all nodes. The root cluster is involved in every update, so it will be weighted with the smallest multiplier.

After every action all the clusters starting from the root and down to the LCA get an update that is weighted by the *multiplier*. The updates also involve re-normalising the affected matrices, since it is required that every row sums to 1 when we come to base probabilities off these values.

4.5 Track Choices from Hierarchical Clusters

Track choices are made using the HEURISTIC function, and a method for choosing the next cluster to explore from a particular point in the tree. These vital two components will be described separately in the following two sections. The basic idea is to traverse down the tree choosing the next cluster from each node, until a point is reached where using the heuristic is either required or sensible. If no choice can be found, the algorithm repeats itself one level higher in the tree, until a choice of song is made.

In order to achieve a good playlist generator we need to shift gradually from exploration to exploitation. We want to explore the different clusters in the early stages of training to learn what the user's preferences are. We define the overall randomness probability λ and initialise it with a large value. We then gradually lower it as the agent learns more about the user. We can also give the user some control over the randomness, and define the actual degree of randomness used in the system as a combination of these two.

Cluster Choice. The algorithm for choosing the cluster traces the probabilities down the cluster tree, picking a cluster at each level. The choice of a cluster depends on the randomness setting used—if the randomness is set to 0, then the cluster with the highest probability is chosen; if the randomness is set to 1, then all the clusters have an equal chance of being selected.

Heuristic Function. Once a cluster is chosen (which happens when a node with leaves as children is reached, or the cluster probability matrix is no longer applicable at this depth in the tree), the heuristic function picks a song to play. The algorithm works by establishing several features which vote for the viability of choosing different songs: the distance from the current track (the closer the better), a list of the immediate listening history (the further the better), and the overall 'preferred' tracks. All three of these are scaled to avoid any one from causing another voter to be ignored, and combined to provide the final vote. An extra setting prevents the current track from being immediately repeated by the heuristic. Then the song with the highest vote gets picked as the next song.

4.6 System Parameters

One of the most important parameters in our system is K, the number of clusters enforced by the k-means algorithm. We chose K to be 6, as this means a hierarchy for a typical library will have around 4–5 levels, and each matrix will have only 36 elements. Furthermore, 6 is roughly the right magnitude to represent the different number of genres at the top of the hierarchy.

The parameters for the Q-learning agent are also vital. The learning rate α was chosen as a balance between a high, oscillation-prone value, and a lower value which may never converge over the course of a user study. Generally smaller values are a wiser choice, so we used $\alpha = 0.05$ for the user study.

We used a very low discount factor, γ, since each state is equally important in such a system—there is no golden end state like we might find with a robot exploring to reach a goal, so it shouldn't heavily optimise for the future.

The initial condition used in the learning may also be quite important. We decided to initialise all matrices to the identity matrix, with a utility of 1 for returning back to the same cluster (itself) again. This was a safe option because the NEXT–CLUSTER algorithm incorporates a degree of randomness irrespective of the user setting.

5 Evaluation

5.1 Methodology

We chose to evaluate our playlist generator in two distinct ways to try to demonstrate its merits. Firstly, a user study was conducted with a small sample of active users to gather both qualitative and quantitative data about the generator's performance. Secondly, a series of set experiments were performed to demonstrate that the player can learn user's preferences and can be trained to select or avoid certain types of song depending on the user's listening history.

Some related work has gone into other methods for evaluating music playlist generators [13]—a surprisingly difficult problem. People do not agree on what defines a 'good playlist', beyond a set of fairly basic assumptions. We cannot just enumerate all good playlists, because there are an intractable number of possible song selection and ordering options for all but the most trivial of music collections. Many APG papers nonetheless attempt to devise metrics which will assign fair scores to attempts by different playlist generators. But without a standard method used across all research in the field, these inevitably self-optimise for the solution in question.

5.2 User Study — Data

For our evaluation study, we recruited 20 participants which we separated into two groups: the control group (5 participants) and the experimental group (15 participants). The participants were asked to use the player for 28 days (the average listening time was 41 minutes per day). The player had 3 modes: manual playlist creation, shuffle mode and the Smart Playlist mode. For the experimental group, the Smart Playlist was generated using our algorithm (and was the most frequently used mode by the participants), and all three modes were used for learning. For the control group, the Smart Playlist's behaviour was identical to shuffle mode. The participants were not aware of which group they belonged to.

The user behaviour was tracked throughout the user study and the data was collected and reported in the form of the following features: total listening time and the number of tracks played, the time and the number of tracks listened to in the three modes we provided, the total number of skips and the number of early skips, the number of jumps and the number of queued songs, the total number of song searches, and, finally, the size of the stored library.

Whilst many interesting conclusions can be drawn from the data we collected, the most important one for the evaluation of our method is the skip count. Tracking the total number of counts, one the other hand, can lead to incorrect conclusions. The absolute skip count values would be higher on more active days, without the *proportion* of songs skipped necessarily changing. We are therefore interested in the relative number of skips—normalising the total number of skips by the total time the user has spent listening to music. Figure 1 plots the relative skip counts in both the experimental and control groups.

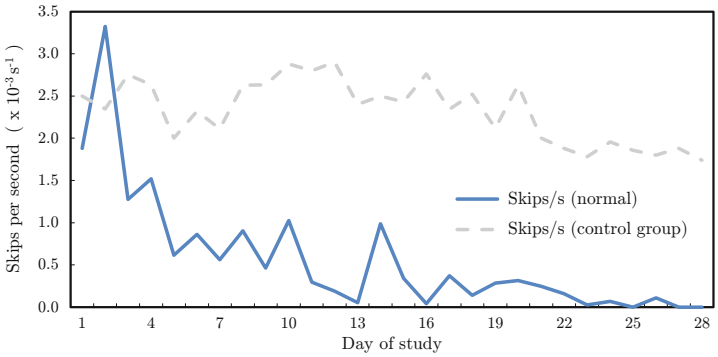

Fig. 1. Skip count results from the user study, normalised by listening time

This graph shows a very convincing trend—one we were hoping for, but did not expect to be as clear. The drop in the relative number of skips is very sharp even after only a couple of days of use, while the relative number of skips for the control group remains stable throughout the duration of the study.

5.3 User Study — Survey

The qualitative side of the user study data was provided by a survey sent out at the end of the study to the participants in the experimental group. The responses were generally very positive. Users seemed to be either fairly active or somewhat dormant in using the player, with no middle ground. Most users had libraries of 100–1000 songs, which aligns well with the cluster size setting we had chosen. The player was described as very easy to use, and users spent most of the time in Smart Play mode. Static playlists were not heavily used, although a number of users claimed this as one of their usual listening practices. The participants were also pleased with the player's utility in rediscovering old music.

5.4 Testing Learning Capabilities

In order to test the learning capabilities of the playlist generator we artificially devised a library with 3 clusters corresponding to 3 different genres. The agent could then be trained by repeatedly creating playlists with tracks chosen from

the different clusters as required. For instance, selecting one track from each cluster in order was shown to result in a Q-matrix with high transition probabilities between those adjacent tracks in the playlist. Throughout this process, the randomness was set near to 0 to enable the exploitation phase of the algorithm.

A similar method was used to demonstrate that the agent can be trained to choose particular types of songs more often, such as a specific musical style or a set of the user's favourite tracks. In this case the learning process was simulated by the skipping of certain tracks during playback with a high learning rate set. The converse process was used to demonstrate that tracks can also be learned to be chosen less frequently, such as for a style the user dislikes.

6 Conclusions

In this paper we have described a novel method for generating music playlists that relies entirely on the analysis of music and can be used offline. It requires no explicit user input, yet still learns user's preferences and is able to generate a playlist that is personalised and adapted to user needs. The playlist generator is also especially suited for large libraries as the use of hierarchical clustering enables it to scale extremely well, while still being able to learn user's preferences.

In addition to that, we have executed a user study to evaluate our algorithm and using the objective skip count metric showed that the algorithm behaves as intended and that it outperforms a baseline shuffle mode. We are making the player publicly available for download at `www.james.eu.org/musicplayer`.

7 Discussion and Future Work

While the system we have proposed shows potential to improve the users' experience when listening to generated playlists, a perfect solution that achieves a zero skip count is obviously impossible—people are always unpredictable to some degree when deciding what music they would like to listen to.

A system that requires no explicit user input is even more difficult to achieve. The reward system for the Q-learning assumes that whenever a song finishes playing, the user must have enjoyed that song. Of course, if a user happens to leave the room or becomes distracted, then this assumption breaks down entirely, so an active listener is required. Possible future extensions include:

- Choosing songs by considering the feature difference between the end of one song and start of the next one would ease the playlist into less similar clusters during exploration more gently and improve coherence.
- Experimentation with different feature weightings. At present, all audio features are given equal weight, but it has not been shown that this is the ideal approach. For instance, the speed of a track may well be a more salient feature than its spectral flux.
- Weighting of user actions by 'user activeness'. So if a user very rarely intervenes in playback, then when they do skip a track this is more significant.

References

1. Bellman, R.: A Markovian Decision Process. Journal of Mathematics and Mechanics 6 (1957)
2. Bertin-Mahieux, T., Ellis, D.P., Whitman, B., Lamere, P.: The Million Song Dataset. ISMIR 12 (2011)
3. Bu, J., Tan, S., Chen, C., Wang, C., Wu, H., Zhang, L., He, X.: Music Recommendation by Unified Hypergraph: Combining Social Media Information and Music Content. In: Proc. ICMR, pp. 391–400, New York, USA (2010)
4. Chen, S., Moore, J.L., Turnbull, D., Joachims, T.: Playlist Prediction via Metric Embedding. In: Proc. 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 714–722 (2012)
5. Cheng, Z., Shen, J.: Just-for-Me: An Adaptive Personalization System for Location-Aware Social Music Recommendation. In: Proc. ICMR (2014)
6. Chi, C.-Y., Lai, J.-Y., Tsai, R.T.-H., Jen Hsu, J.Y.: A Reinforcement Learning Approach to Emotion-based Automatic Playlist Generation. In: International Conference on Technologies and Applications of Artificial Intelligence, vol. 12, pp. 60–65 (2010)
7. Schafer, J., et al.: Collaborative filtering recommender systems. The Adaptive Web, 291–324 (2007)
8. Hu, Y., Koren, Y., Volinsky, C.: Collaborative Filtering for Implicit Feedback Datasets. In: ICDM, pp. 263–272 (2008)
9. Jannach, D., Kamehkhosh, I., Bonnin, G.: Analyzing the Characteristics of Shared Playlists for Music Recommendation. In: Proceedings of the 6th Workshop on Recommender Systems and the Social Web (2014)
10. Koren, Y.: Factorization Meets the Neighborhood: A Multifaceted Collaborative Filtering Model. In: KDD (2008)
11. Liebman, E., Stone, P.: DJ-MC: A Reinforcement-Learning Agent for Music Playlist Recommendation. AAMAS 13 (2014)
12. McEnnis, D., Fujinaga, I., McKay, C., DePalle, P.: JAudio: A feature extraction library. In: ISMIR (2005)
13. McFee, B., Lanckriet, G.: The Natural Language of Playlists. In: Proc. ISMIR, Miami, FL, USA (October 2011)
14. McFee, B., Lanckriet, G.: Hypergraph Models of Playlist Dialects. In: Proc. ISMIR, Porto, Portugal (October 2012)
15. Pampalk, E., Pohle, T., Widmer, G.: Dynamic Playlist Generation Based on Skipping Behavior. In: Proc. ICMIR, pp. 634–637 (2005)
16. Pazzani, M.: Content-based recommendation systems. The Adaptive Web, 325–341 (2007)
17. Platt, J.C.: Fast embedding of sparse music similarity graphs. In: NIPS (2003)
18. Platt, J.C., Burges, C.J.C., Swenson, S., Weare, C., Zheng, A.: Learning a Gaussian Process Prior for Automatically Generating Music Playlists. Microsoft Corporation (2001)
19. Schedl, M., Breitschopf, G., Ionescu, B.: Mobile Music Genius: Reggae at the Beach, Metal on a Friday Night? In: Proc. ICMR (2014)
20. Watkins, C.: Learning from Delayed Rewards. PhD thesis, King's College, University of Cambridge (1989)
21. Yoshii, K., Goto, M., Komatani, K., Ogata, T., Okuno, H.G.: An Efficient Hybrid Music Recommender System Using an Incrementally Trainable Probabilistic Generative Model. Trans. Audio, Speech and Lang. Proc. 16(2), 435–447 (2008)