Automatic Playlist Continuation: A Comparison of Different Graph-Based Methods

Drew Davis
University of Michigan - CSE
Ann Arbor, Michigan
drewdavi@umich.edu

Jungho Bang
University of Michigan - CSE
Ann Arbor, Michigan
bjungho@umich.edu

Siddharth Venkatesan
University of Michigan - ECE
Ann Arbor, Michigan
siddvenk@umich.edu

ABSTRACT

Music Recommender Systems are receiving more attention in recent years because of the rapid growth of online music streaming services. While these systems are capable of making good recommendations, current implementations are typically naive in that they do not consider much, if any, information outside of user-item interactions. It is reasonable to assume that an improved recommender system would incorporate additional information to make better recommendations. In this paper we focus on a sub-problem of music recommendation: automatic playlist continuation. Much of the recent research in automatic playlist continuation explores the use of graphical models to make predictions. In this paper we consider three different graph-based methods for the task of automatic playlist continuation: random walk with restarts on a homogeneous graph, personalized page rank on a higher-order network, and random walk with restarts on a hypergraph. We show that both higher order networks and hypergraphs can be used to make predictions for playlists, and explain why an improved method should incorporate the strengths of both of these models.

1 INTRODUCTION

Automatic Playlist Continuation (APC) is the process of automatically constructing sequences of tracks to continue a playlist from the tracks already present in a playlist. Playlists represent collections of tracks that are consumed sequentially and typically represent some classification unique to the creator (i.e. favorite jazz songs, study music, or birthday songs). Quality playlist track recommendations can significantly impact the quality of a user's listening experience by introducing them to new tracks that fit the playlist classification. For the semester-long project, our group will participate in the RecSys Challenge 2018 competition [1]. The goal of this challenge is to achieve automatic playlist continuation by predicting the missing tracks to complete playlists from which a number of tracks have been withheld. Specifically, we have the following problem statement: *Given a partial playlist with K seed tracks and relevant metadata, predict candidate continuation tracks ranked in decreasing order of relevance.*

[1] http://www.recsyschallenge.com/2018/

EECS 598, 2018, University of Michigan

Much of the previous research in playlist continuation focuses on building models based on random walks with restarts (RWR) on homogeneous graphs [5] [7]. These attempts build a graph *G(X,E,W)*, where the vertices *X* are tracks, the edges *E* are connections between tracks, and *W* are the weights of the edges. Playlists are then generated by following a random walk using a teleport set, which is composed of the tracks in a partial playlist. The random walk consists of following a random trajectory through the graph where the probability of transitioning to another vertex is proportional to its edge weight. More recently, there has been research conducted on network analysis employing the use of higher-order networks [11] and hypergraphs [6] [1] [9]. Higher-order networks use a similar graph structure to homogeneous graphs, except that they directly support the modeling of higher-order relationships in the graph. Instead of assuming first-order relationships between tracks, higher-order networks consider the sequence of several tracks preceding and/or following a given track in a playlist. This model inherently assumes that the sequence of tracks inside of a playlist is significant and latent information can be derived from the ordering of tracks. Hypergraphs are extensions of homogeneous graphs where hyperedges can connect more than two vertices. Unlike homogeneous graphs, which can only express pairwise relationships, hypergraphs can represent more general concepts of connectivity through n-way relationships. Hyperedges are used to represent playlists or features, and the resulting hypergraph can be used to perform a modified RWR.

After surveying the current field of music recommendation systems, it was not clear which of the described methods was superior to the others. RWR on homogeneous graphs is simple, well documented, highly available, and has been used successfully in many fields. Higher-order networks may be able to extract complex relationships between sequences of tracks that humans do not naturally interpret and are fairly well studied. Hypergraphs are the least well-studied and have the most limited code availability; however, they more naturally represent groups without placing high importance on sequence. Given that there is no 'golden standard' algorithm for APC, we explore these three different methods in hopes of uncovering how the different approaches compare for playlist continuation. In the following sections, we summarize our data sources, introduce our proposed approaches, describe our experiments and results, and discuss related work.

2 DATA

In partnership with the 2018 ACM RecSys Challenge, Spotify has put out the Million Playlist Dataset (MPD). Each of the one million playlists in the dataset has been individually created and curated by actual Spotify users. Every playlist lists the sequence of tracks it

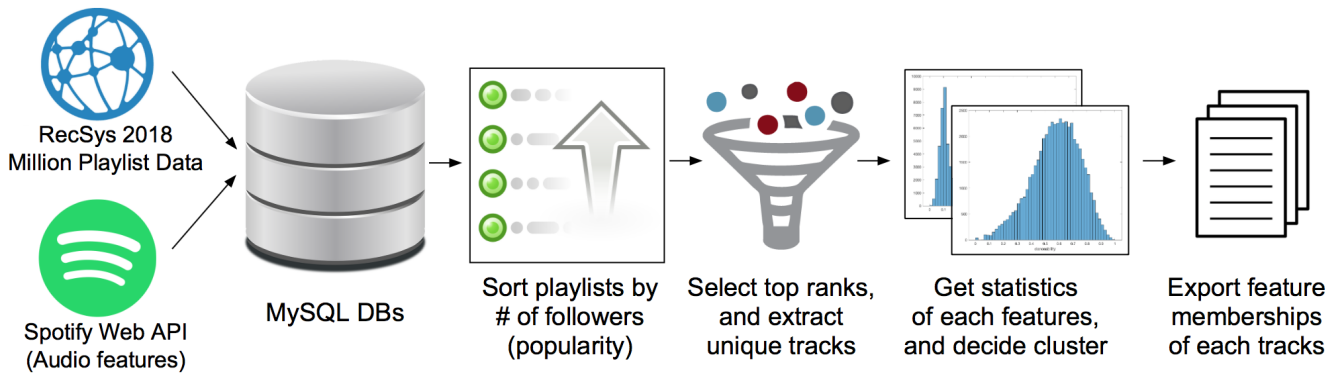

Figure 1: Data collection and processing

contains as well as basic metadata such as playlist name, number of followers, duration of all tracks, etc.

We ran analysis using Hadoop to get statistical information about the MPD. One interesting find from the statistics is that tracks are very densely connected between playlists. Even though there are 66 million total tracks across all 1M playlists, the number of unique tracks is just 2M. This means that on average, each playlist only has 2 unique tracks on and its remaining 64 tracks are shared with other playlists. For example, the top 5 most frequent tracks appear in about 40k different playlists.

Due to the massive size of the MPD, we needed to have an initial, appropriately sized dataset to test our methods on. If we only use a small portion of the MPD, however, we can't guarantee a similar distribution of playlist statistics to the full data set. As you can see in Table 1, other than the number of listings, all other metrics are different from the ratio of the overall data. Because of the varied behavior between small subsets of the larger dataset, we decided that it would be necessary to take multiple approaches with the data.

2.1 Synthetic Data

Our first approach to procuring a manageable set of data was to generate synthetic data. We created a playlist synthesis algorithm that generates a small data set that shows similar behavior to the overall 1 million dataset. The synthesis program takes input parameters like number of genres, number of artists, range of albums per artists, range of tracks per album, number of playlists, and range of tracks per playlist.

The synthesis algorithm is based on following assumptions:

	Overall MPD	1K prefix
Playlists	1M	1K
Track listings	66.3M	67.5K
Unique tracks	2.3M	34.4K
Unique albums	0.7M	19.2K
Unique artists	0.3M	1.0K

Table 1: Statistics of MPD using Hadoop

- An artist is not likely to have her albums in too many diverging genres.
- Some of the tracks appear more frequently in playlists than others: They are expected to follow power law.
- Tracks in the same genre will share similar audio features. In other words, audio features are based on the genre of the track.
- Most of the tracks in one playlist are likely to have similar genre.

To realize these assumptions, we used a weighted probability distribution where our intended feature value has 66% to 90% chance of being associated with a track. This randomness allows us to have more realistic and appropriate data in training and evaluation.

The result of our synthesis model shows similar behavior like the MPD. Contrary to the 1K prefix, our synthesized graph has high degree of overlap among tracks over playlists. For example, the data we used in our experiment has 2.7 unique tracks per playlist, which means that they are as closely connected as the MPD.

In the experiments, however, we found that the synthetic data is not suitable for learning importance of features, represented by weights, on a hypergraph. Our assumption on the relationship between each audio feature and its genre results in redundant features. Even though our model can predict the best genre and audio features to recommend, the intersection of these hyperedges is very large. This high degree of overlap between hyperedges causes the hypergraph model to rank tracks almost equally, yielding no useful results. Also, since our synthesis algorithm treats each playlist as a set without order, the synthetic data is inherently inappropriate for the higher-order network.

2.2 Real World Data

The MPD consists of 1000 JSON files, where each file contains the data of 1000 playlists, and the overall size is over 30GB. To efficiently query data from this huge dataset, we decided to use DBMS to store and load the data. We contacted Advanced Research Computing - Technology Services, where the flux machines are hosted, to get access to database-ready machines. However, we lost contact with them and could not get access to their database. Therefore, we had to set up our own instances of MySQL daemon on flux and

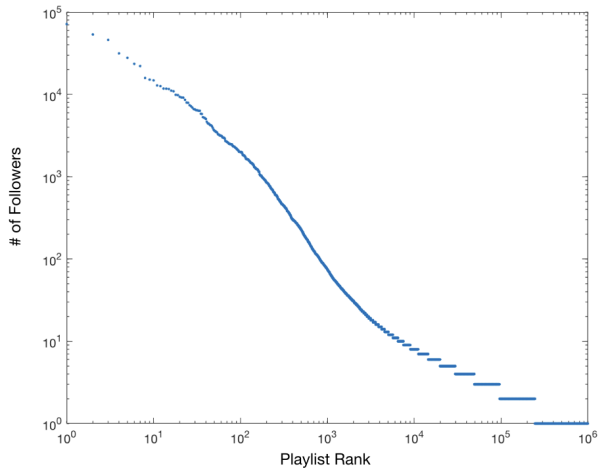

Figure 2: Number of followers of playlists over ranking

the CAEN machines. Installing MySQL without admin privileges was very laborious, but it enabled us to perform complex queries effectively.

We implemented a DB adapter to parse the original MPD, determine the relationships, and insert them into a database using Ruby on Rails. Rails' Active Record connects ruby classes to database tables to establish a persistence layer and reduces the need to write raw SQL. Also, by using ruby, we could directly make requests to the Spotify API, parse the responses for track features, and store them to the database with in one program.

Figure 1 explains how we collected and processed data. First, we got the list of unique track URIs(Uniform Resource Identifier) using Hadoop. Then, we processed the MPD and queried the Spotify API in two different machines. One machine analyzed listing relationships between playlists and tracks, while the other machine made requests to the Spotify API to get audio features for all 2 million tracks. Since Spotify API has certain limitations on the number of requests per time, we needed delicate control of timing when accessing Spotify. Finally, we merged these two databases and obtained playlist data associated with tracks where each track is associated with related audio features.

Storing data in the database allows us to create subsets of playlists in a variety of methods. In the MPD, each playlist includes information about how many followers the playlist has, which can indicate the popularity of a playlist. Figure 2 suggests that the number of followers of playlists follows a Power Law distribution. We therefore originally assumed that popular playlists would serve as better playlists to initially test on. Under this assumption, our first approach was to sort the playlists by popularity, and then evaluate our models on the popular playlists. However, by using the most popular playlists as our training and testing data we inherently introduce bias into our system. To combat this, we also evaluate our models on randomly selected playlists. The performance comparison of the two different playlist sets is discussed in Section 4.

	Unique tracks	per playlist
1M (overall)	2.3M	2.3
Popular 100K	792.5K	7.9
Popular 10K	248.6K	24.9
Popular 1K	51.4K	51.4
Random 100K	679.8K	6.8
Random 10K	170.3K	17.0
Random 1K	35.7K	35.7

Table 2: Statistics of MPD subsets by sorting

3 PROPOSED METHOD

3.1 Random Walk with Restart

Random walk with restart (RWR) is one of the fundamental methods for understanding graphs today. It has been implemented to great success in many different applications, as partly discussed in [10] and [4]. The basic idea behind RWR is that, given a bias set, randomly walking across edges in the graph will naturally make contact with the vertices that are more relevant to the bias set most often. More formally, at every step of the random walk an edge connected to the current location vertex is randomly selected based on its weight. The bias set acts as a teleport set, and at each step there is a restart probability (c) that the position of the walk is reset to one of the vertices inside of the teleport set. Vertices are scored based on how frequently the random walk reached them.

Specifically for our application, the graph was set up so that each track was represented as a vertex and an edge was created between two tracks for every instance where they both appeared on the same playlist. This structure favors songs that appear on shared playlists multiple times over songs that only appear together once. To predict songs, a partial test set playlist is provided as the teleport set. The random walk scores of all vertices inside the graph with the provided teleport set are saved and ranked. The top ranked scores are used as predictions of what is in the hidden part of the set.

3.2 Higher-order Network

Traditional network representations only account for first-order dependencies (Markov property), whereas higher-order networks (HON) preserve higher-order dependencies that exist implicitly in the structure of a network [11]. Compared to a conventional first-order network, HON is more expressive in representing the sequences of vertices that compose a unique state. This is accomplished by internally representing each state as a set of vertices, where each different vertex represent a unique sequence of dependencies leading up to that state. A simple example of this structure is shown in 3, where a single state is represented by the two vertices seen inside of the gray box.

Playlists inherently have an ordering, intentionally or otherwise. A playlist can be seen as a directed chain over the tracks. If the ordering of tracks is significant information, then finding a pattern in ordering will guide us in the extension of playlists. On the other hand, the ordering could be irrelevant, in which case the benefits of HONs would be mitigated.

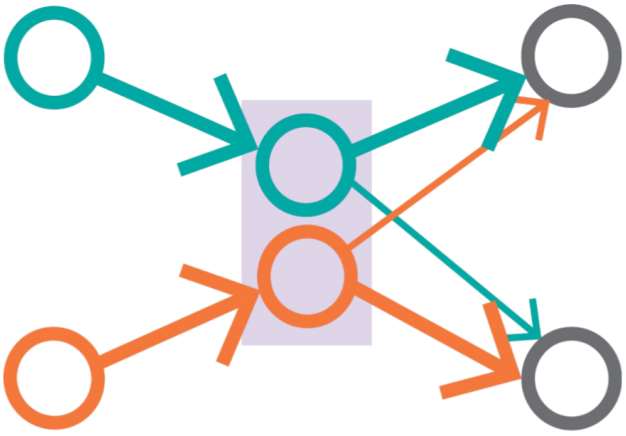

Figure 3: Example of Higher-order Network Graph Structure. Source [11]

A brief overview of the method used to represent playlists as a HON follows. For a more in-depth explanation, refer to [11]. To construct the HON, we provided the playlists as a directed sequence of tracks. From this, rules are extracted that represent significant dependencies in the tracks of the data. These rules are generated incrementally by increasing the order of dependencies and will reveal the typical order of dependencies found in the data. After the rules are generated, they can be used to generate a graphical representation of the dependencies. This is accomplished by representing a vertex with an n-order dependency as n vertices. The in-edges and out-edges for higher-order vertices have to be carefully constructed since they do not naturally exist in the input data, but must represent the same information. The outputted HON acts just like a standard graph and enables use of conventional network analysis techniques. A Personalized PageRank analysis is then performed on the outputted HON. PageRank is discussed in [4]. A partial test set playlist is used as the preference set. The rankings of all vertices, as calculated by PageRank is stored and the top ranked tracks are used as predictions of what is inside the test set.

Before we performed the evaluation, we checked the highest order that was found in the HON result. With our Top 10K popular playlist subset data, we didn't get any dependencies higher than third-order. Second-order dependencies were the longest, and they accounted for 41% of all of the dependencies found. These low order dependencies hint at the fact that the sequence ordering of tracks inside of a playlist may not be as informative as one would hope, when using HONs.

3.3 Hypergraph Random Walk

A hypergraph is a generalization of a graph in which homogeneous vertices are grouped by hyperedges, where a hyperedge is a subset of vertices of arbitrary size. The distinction between a traditional graph and a hypergraph is shown in Fig. 4. Hyperedges on a graph can be used to represent any grouping of vertices (i.e. vertices with a specific shared feature).

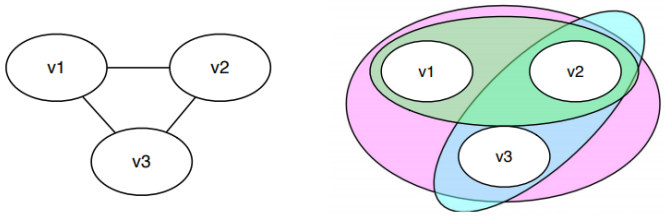

Figure 4: A graph representation of three distinct groupings of vertices. Shown as a traditional graph with standard edges (left), and hypergraph with hyperedge groupings (right). Source [3]

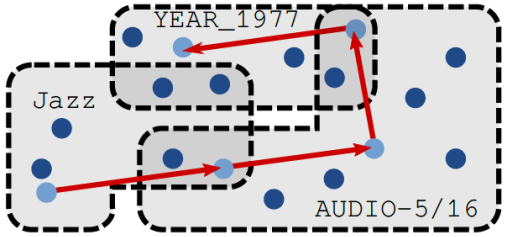

Figure 5: An example of a random walk in a hypergraph. Edges are the grey regions surrounded by dotted black lines, and each edge contains multiple vertices (songs). Source [6]

Our approach is heavily influenced by the work presented in [3], [6], and [9]. To understand our proposed approach, we start with modeling user behavior when constructing playlists. Given a full library of songs X, a user first narrows down to a subset of the entire music library $X_s \subset X$. For example, if a user is creating a rock playlist it is reasonable to assume that they are only considering a subset of the entire library of songs when constructing their rock playlist. Once the user has narrowed down to a subset of songs X_s, there is an equal probability of selecting a seed vertex (song) x_0 from this subset. To continue the playlist, we assume that the user first selects a subset of songs containing x_o, and then selects a transition $x_t \rightarrow x_{t+1}$ within this subset.

This model of user behavior is conducive to representing the APC problem in terms of a random walk on a Hypergraph. Hypergraphs are generalizations of undirected graphs that allow an edge (hyperedge) $e \in E$ to be an arbitrary subset of vertices rather than just two vertices. Thus, it makes sense to think of hyperedges as sets rather than pairwise relationships. Hyperedges can be general sets such as *Country Songs*, or specific sets such as *Songs in Drake's album Views*. Essentially, hyperedges represent sets of songs related in some manner, so we can use hyperedges to represent different relationships between songs. Some examples of possible relationships that can be encoded as subsets of songs are songs with the same artist, songs in the same genre, acoustically similar songs, songs in playlists with a specific label like "Rock", etc. An example of a hypergraph with hyperedges is shown in Figure 5.

We now formalize our model, with inspiration from [6]. Let $H = (X, E)$ be a hypergraph over the vertices (songs) X, and edges $E \subseteq 2^X$. In the training phase, we are to construct the hyperedges. This reduces to deciding which features to represent in the hypergraph. There are three feature classes we represent in our hypergraph model through hyperedges: common album, audio similarity, and training playlists. Common album hyperedges group songs together that come from the same album. Audio similarity hyperedges group songs together based on various levels of audio similarity. To construct these hyperedges, we ran K-means clustering on the audio feature vectors for each song in the data set. Using values of 4, 16, and 64 for K, we construct hyperedges that describe various levels of similarity between songs. Finally, for the playlists in the training set, we include hyperedges representing these playlists in the model. Now that we have constructed the hyper graph model $H = (X, E)$, we can run RWR using the test playlists as teleport sets. Random walks on hypergraphs are similar to random walks on homogeneous graphs, and we follow the procedure outline in [1]. Given a starting vertex x_0, we first randomly jump to any hyperedge that contains x_0. From this hyperedge, we then randomly pick a song x_t as the next song in the sequence. We continue this process, allowing for restarts with probability 0.15, where a restart is a jump to any song in the visible set. For our experiments, we divide each test playlist into visible (first half of the playlist) and hidden (second half of the playlist) sets. Using the visible set as our teleport set in the RWR, we obtain vertex rankings, and evaluate the model using the hidden set and vertex rankings.

3.4 Evaluation Criteria

The primary metric we will use to evaluate the performance of our system is R-Precision. R-Precision measures the number of retrieved relevant tracks R divided by the number of known relevant tracks G:

$$R - Precision = \frac{|G \cap R_{1:|G|}|}{|G|}$$

A perfect R-Precision score is 1.0, which indicates that every hidden track was returned in the top ranked vertices. A score of 0.0 indicates that no hidden tracks were returned in the top ranked vertices. In order to understand the performance of our system, we only consider the top 500 ranked vertices, regardless of the data set size. This is the same procedure that will be used to evaluate submissions to the RecSys Challenge.

4 EXPERIMENTS

We conducted experiments on each of the three previously mentioned methods. For each method, we evaluated performance on the 1k and 10k random playlist sets. In each experiment, we use an 80-20 split for training and testing respectively. For each playlist in the test set, we use a 50-50 split at the midpoint to separate the seed tracks from the hidden tracks. To evaluate each method, we compute the R-Precision score based on how many of the tracks from the hidden set appear within the top 500 ranked tracks, averaged across all test playlists.

	Average Precision
Synthetic	0.500
Popular	0.064
Random	0.011

Table 3: Average Precision of Hypergraph RWR on Different 1,000 Playlist Datasets

4.1 Dataset Selection

Due to the limited computational resources available, it was not feasible to evaluate the methods on the full one million playlists. Instead, as discussed in Section 2, we generated several smaller sets of data.

The first set of data was a synthetic dataset consisting of 1000 playlists, generated to replicate the ratios of tracks, albums, artists, and genres seen in the full MPD. However, it was discovered the features of the synthetically generated data were overly-correlated and created a dataset that was trivial to learn. The empirical results demonstrating this are shown in 3. The hypergraph RWR method achieves an average precision of 50% on this dataset, whereas on the 1k random data set its average precision is only 1.1%. This extreme disparity in performance was a clear indicator that the synthetic dataset was not representative of the real world data we were attempting to optimize for, so we will not discuss results on the synthetic data sets further.

The next sets of data were generated by retrieving the most popular playlists from the MPD and learning/testing with those. A 1,000 playlist set and a 10,000 playlist set were created. The intuition was that more popular datasets are accepted by a larger number of people and therefore contain more significant information for learning. However, it was realized that using the most popular playlist for training and testing introduced a bias into the data and was showing in artificially high precision, relative to the randomly selected data, as shown in 3. Therefore, it was also decided that the evaluations of methods should not be performed on the most popular playlist sets.

The final sets of data were created by randomly selecting playlists from the MPD for learning and testing. A 1,000 playlist set, 10,000 playlist set, and 100,000 playlist set were randomly retrieved from the 1,000,000 playlists available. This method of selection avoided intentional bias and still allowed a decent representation of the composition of the MPD. Between the different playlist set sizes, it was decided that evaluations should be performed on the 10,000 playlist set because it allowed the most accurate representation of the full MPD while still being computationally reasonable with the resources that we had access to.

4.2 Homogeneous Graph Random Walk

The homogeneous graph random walk with restarts method was intended to be our baseline. It is the simplest and most well studied of the three methods. Therefore, we anticipated mediocre performance that would set the bar for the other methods. The experimental setup was as follows. First, we constructed the graph for each data set using the method outlined previously in section 3.1. We then

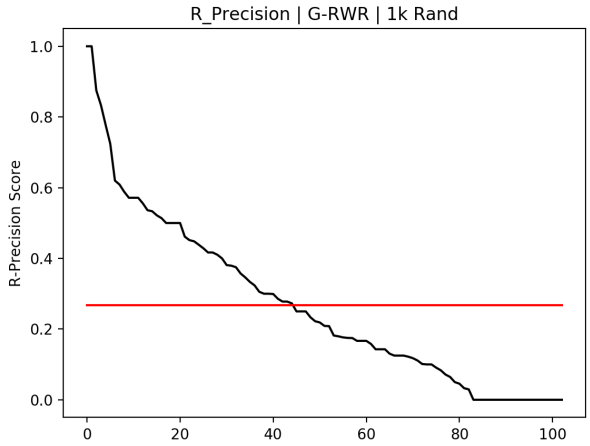

Figure 6: R-Precision for Homogeneous Graph on 1k random set. Mean in red

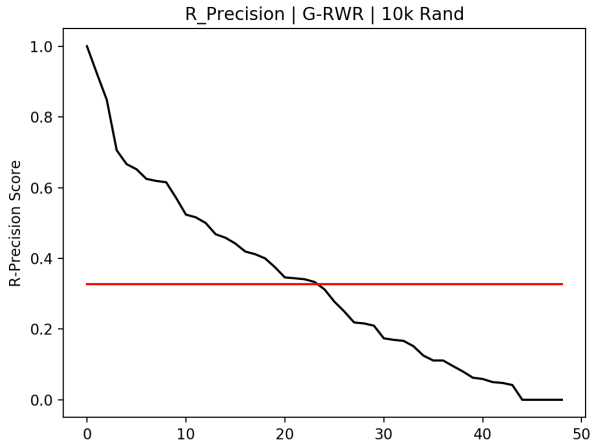

Figure 7: R-Precision for Homogeneous Graph on 10k random set. Mean in red

used the PEGASUS[2] graph mining system to run RWR with each data set. We faced significant computation time challenges when running this system on the flux servers at the University of Michigan, so we were only able to generate partial results. As an example, running the complete 10k random playlist set is estimated to take roughly 250 hours. Figures 6 and 7 show our performance for the homogeneous graph model on the 1k and 10k data sets. On the 1k and 10k data sets, we obtained an average R-Precision value of 0.2672 and 0.3272 respectively. It is extremely important to note that these averages are based on a small set of the total testing playlist set due to slow computation time per playlist. Thus, while these results seem surprisingly good, it is hard to compare these results with the results from the other models because they do not reflect the true performance over the entire testing set.

4.3 HON Personalized Page Rank

The HON Personalized Page Rank method was used to determine whether good recommendations can be made using only the structural and sequential information of playlists. The experimental setup was as follows: First we constructed an input graph for both data sets according to the procedure outlined in section 3.2. Next, we run Personalized Page rank with a preference set for each each of the test playlists in the given data set. Our implementation leverages the NetworkX[3] graphical analysis library to perform Personalized Page Rank, and we run this method on the Horton server at the University of Michigan. Computation of the 10k random dataset took roughly 20 hours to complete. The results for the 1k and 10k data sets are shown in figures 8 and 9 respectively. On the 1k and 10k data sets we obtained average R-Precision values of 0.3513 and 0.2978 respectively.

[2]http://www.cs.cmu.edu/ pegasus/
[3]https://networkx.github.io/

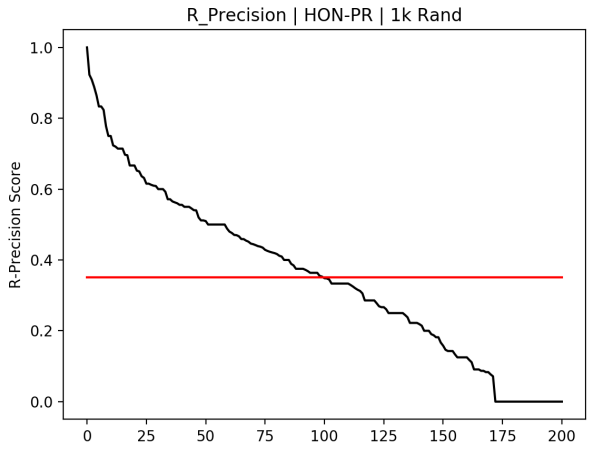

Figure 8: R-Precision for HON on 1k random set. Mean in red

4.4 Hypergraph Random Walk

The hypergraph random walk method is the primary method we explored for the task of APC. The experimental setup was as follows: First we construct the hyperedge to vertex mappings for albums, audio features, and training playlists for the data set in consideration. Next, we combine all the hyperedge-vertex mappings into a single hypergraph file. This is fed as input to our hypergraph RWR system. Our hypergraph RWR system is built upon the MESH library from [3], which is a scalable and efficient library built on top of Apache Spark for hypergraph analysis. MESH represents a hypergraph internally as a bipartite graph, where the hyperedges are transformed into hyperedge vertices. This internal representation is then used to perform a RWR in a distributed manner where one

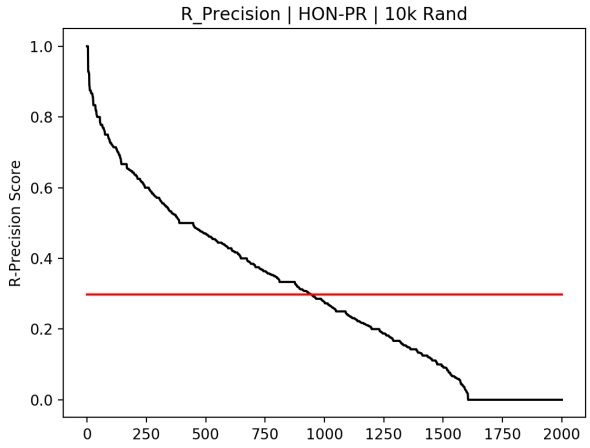

Figure 9: R-Precision for HON on 10k random set. Mean in red

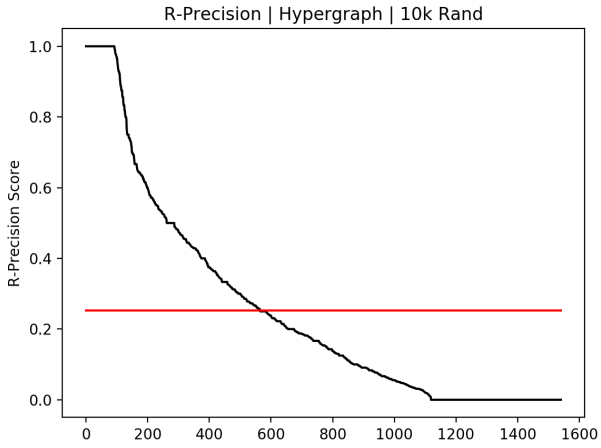

Figure 11: R-Precision for Hypergraph on 10k random set. Mean in red

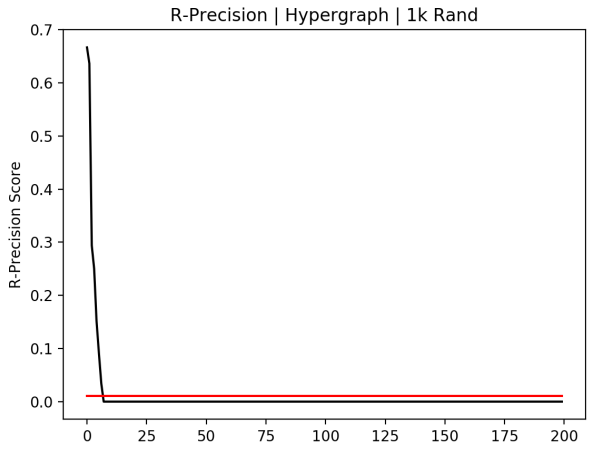

Figure 10: R-Precision for Hypergraph on 1k random set. Mean in red

step makes a jump from a song vertex to an edge vertex, and then from the edge vertex to a song vertex. Computation of the 10k random dataset took roughly 40 hours to complete on the University of Michigan's Flux servers. Figures 10 and 11 show our performance for the hypergraph model on both data sets. On the 1k and 10k data sets, we obtained an average R-Precision value of 0.0106 and 0.2526 respectively

4.5 Comparison

There are a few important results we would like to point out when comparing the three models with each other. A summary of our results can be found in table 4. First, we are surprised that the homogeneous graph model performed as well as it did. However, these results are obtained over a much smaller fraction of the test playlists (50% and 2.5% for 1k and 10k respectively) due to the high computation time of using PEGASUS on flux. Therefore we focus on the comparison between the HON and hypergraph models.

The results from the HON model are surprising because the model has a higher average R-Precision score for the smaller 1k dataset versus the larger 10k data set. We think this is because the 1k dataset represents 0.1% of the entire MPD dataset, so we would expect higher variance in performance among different subsets of 1k playlists, and for the performance to be highly dependent on the songs and playlists in the data set. A full exploration of different 1k and 10k data sets would be needed to examine this further.

With the hypergraph model, we see a significant difference in the performance when using different sized data sets. The hypergraph model using the 1k data set performs the worse out of all the experiments. The performance improves significantly, more than twenty times better, when using a much larger data set. We believe that this is due to the fact that with the smaller data set, the hyperedges contain fewer nodes that are shared with other hyperedges. With higher levels of overlap between hyperedges, there are more options for the random walker to take at a given step. In the smaller data set, it is much easier for the random walker to get "stuck" in a hyperedge if there are relatively few nodes also contained in another hyperedge. Intuitively, the less overlap there is between hyperedges, the more likely the random walker is to stay within a hyperedge.

	1k random	10K random
G-RWR	0.2672	0.3272
HON-PR	0.3513	0.2978
HG-RWR	0.0106	0.2526

Table 4: Summary of Results for 3 Comparison Models

Comparing the results of the HON model and the hypergraph model using the 10k data set, we see that the HON model performs slightly better. This is a surprising find for us given our preconceived notion of sequence being unimportant in playlist creation. The results from both HON trials suggest that sequence is a relevant factor when constructing playlists. However, results from the hypergraph model suggest that more nuanced notions of similarity, like acoustic similarity and album membership, can also be used to provide recommendations of a similar quality to the ones generated by the HON model. Given these findings, we believe that one way to create an improved graph-based approach would combine the strengths of HONs and hypergraphs. From the HON experiments we learned that common sequences of 3 songs are useful pieces of information. We could model these sequences as hyperedges in the hypergraph model, which would allow us to combine the useful information learned by the HON, with the flexible nature of representing relationships in hypergraphs. Additionally, our current hypergraph model only utilizes album information, acoustic information, and playlist-track information. There are more hyperedge classes we could introduce such as common artist, genre, time-period, producer, and nearly every other common feature of the songs. With a more complex hypergraph that models more notions of similarity, we believe that we could improve the average R-Precision value of the hypergraph model.

5 RELATED WORK

5.1 Music Recommendation

The authors of [8] provide a survey of the current field of music recommender systems. They explain that music recommendation systems have experienced a surge of interest in the past several years. Functional automatic recommendation systems are in place today that help users discover new music; however, the current systems focus almost exclusively on user-item interactions. Further, the authors list the grand challenges of music recommendation systems today, which includes automatic playlist continuation. It is clear that there is significant room for improvement in the field, which has led Spotify to put out their RecSys challenge.

5.2 Random Walk with Restart

The random walk with restart method is a fundamental method for understanding graphs and how different vertices relate to each other. It is well studied, documented, and has a variety of off-the-shelf code available for use. The basic algorithm and uses for RWR are discussed in [10] and [4].

5.3 Higher-order Networks

Higher-order Networks are a relatively new system in graph mining, with [11] being published less than two years ago. We did not come across any publications where HON had been used specifically for playlist recommendation. However, the concept of interpreting the sequence of songs inside of a playlist has been studied in previous works such as [2], where the context of a user is inferred from the sequence of tracks and used to recommend the next track. The authors show that inferring strong sequential patterns inside of a playlist can improve accuracy and recall of their results. These results show, that at least on some playlists, it is possible to infer meaningful information from the ordering of tracks.

5.4 Hypergraphs

Hypergraphs are not as recent a development as HON; however, they are not nearly as well studied as homogeneous graph RWR. Traditional network analysis methods cannot be natively applied to hypergraphs, however, use of RWR with hypergraphs has been used in several papers, including [3], [6], and [9]. In fact, [6] applied the use of hypergraph RWR to playlist generation. The authors of the paper focused on implementing different features as hyperedges and showing that tracks can be grouped based off a weighted RWR across a hypergraph. This method was our original inspiration to bring hypergraphs to the APC problem.

6 CONCLUSIONS

To wrap up, there are a few key things we would like to emphasize about our experiments. First, the sequential nature of playlists is more important than we originally believed. The results from the HON model indicate that learning important sequences, specifically sequences of 3 songs, can help make song recommendations for playlists. Second, the results from the hypergraph model indicate that information beyond user-item relationships can also be used to make song recommendations. Both of these models performed comparably on the 10k data set, but neither model performed exceptionally. Thus, we believe there is room for improvement by adding additional features to the hypergraph model, including important HON sequences.

There are also some lessons we learned from the implementation aspect of this project. Although we discovered a few different hypergraph models in our initial research, there are limited options when it comes to implementing hypergraph algorithms. We spent more time than we would have liked testing and adapting different hypergraph libraries and implementations for our project. It became very apparent to the group that even if a paper is published, its public code may not be highly functional (or documented). Additionally, we learned the importance of understanding reasonable expectations. More than once our results showed average precisions in the 90%s. Each time it was tempting to feel that we had solved the problem and had the perfect solution, when in reality the over-performance was just due to a bug in the code.

7 DIVISION OF WORK

Paper reading

All group members have read the relevant papers mentioned in this report, as well as additional papers related to random walk, higher-order networks, hypergraphs, and broader graph learning methods.

Implementation and experimentation

- DD: RWR code baseline implementation/evaluation, HON code modification/evaluation, (failed) implementation of hypergraph code from [6]
- JB: Data collection and organizing, Data preprocessing for HON and HG, Implement/Modify code for HON testing, HON evaluation

- SV: Hypergraph implementation and experiments, K-means clustering, plotting and evaluation results

Report writing

- DD: Introduction, Proposed Method, Experiments, Related Work
- JB: Data, Proposed Method
- SV: Abstract, Proposed Method, Experiments, Related work

REFERENCES

[1] Jiajun Bu, Shulong Tan, Chun Chen, Can Wang, Hao Wu, Lijun Zhang, and Xiaofei He. 2010. Music Recommendation by Unified Hypergraph: Combining Social Media Information and Music Content. In *Proceedings of the 18th ACM International Conference on Multimedia (MM '10)*. ACM, New York, NY, USA, 391–400. https://doi.org/10.1145/1873951.1874005

[2] Negar Hariri, Bamshad Mobasher, and Robin Burke. 2012. Context-aware music recommendation based on latenttopic sequential patterns. In *Proceedings of the sixth ACM conference on Recommender systems*. ACM, 131–138.

[3] Benjamin Heintz, Shivangi Singh, Corey Tesdahl, and Abhishek Chandra. 2016. MESH: A Flexible Distributed Hypergraph Processing System. (2016).

[4] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. 2014. *Mining of massive datasets*. Cambridge university press.

[5] Beth Logan. 2002. Content-Based Playlist Generation: Exploratory Experiments Content-Based Playlist Generation: Exploratory Experiments.

[6] Brian Mcfee and Gert Lanckriet. 2018. HYPERGRAPH MODELS OF PLAYLIST DIALECTS. (03 2018).

[7] R. Ragno, C. J. C. Burges, and C. Herley. 2005. Inferring Similarity Between Music Objects with Application to Playlist Generation. In *Proceedings of the 7th ACM SIGMM International Workshop on Multimedia Information Retrieval (MIR '05)*. ACM, New York, NY, USA, 73–80. https://doi.org/10.1145/1101826.1101840

[8] Markus Schedl, Hamed Zamani, Ching-Wei Chen, Yashar Deldjoo, and Mehdi Elahi. 2017. Current Challenges and Visions in Music Recommender Systems Research. *arXiv preprint arXiv:1710.03208* (2017).

[9] A. Theodoridis, C. Kotropoulos, and Y. Panagakis. 2013. Music recommendation using hypergraphs and group sparsity. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. 56–60. https://doi.org/10.1109/ICASSP.2013.6637608

[10] Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. 2006. Fast random walk with restart and its applications. (2006).

[11] Jian Xu, Thanuka L. Wickramarathne, and Nitesh V. Chawla. 2016. Representing higher-order dependencies in networks. *Science Advances* 2, 5 (2016). https://doi.org/10.1126/sciadv.1600028 arXiv:http://advances.sciencemag.org/content/2/5/e1600028.full.pdf