Acoustics Multiple User Influenced Song Recommendation System

Jon Tedesco * Rob Burke † Adrian Kreher ‡ Brian Gladden §

Department of Computer Science
University of Illinois at Urbana-Champaign

Abstract

Acoustics is a networked jukebox, developed by the UIUC ACM chapter, that is ideally geared towards an environment where a community democratically chooses music. It is highly configurable and flexible, but relies heavily on user participation to function correctly, and as such, when user participation dwindles for some period of time, songs are selected randomly [Kreher 2011]. Although this allows music to continue to be played when users stop voting, we can make a more intelligent song selection system. Through our project, we have implemented a substantial extension to Acoustics that allows autonomous selection and recommendation of music based on the musical tastes of users in the room, even if they are not actively voting for songs. To do this, we implement a content-based music filter that analyzes user vote histories to determine musical tastes of each user, and chooses songs that best fit the shared musical tastes of users in the room.

Keywords: content-based filtering, Acoustics, music similarity, Last.fm, music analysis, ACM, UIUC

1 Introduction

Acoustics is typically used by a handful of people, roughly 2-4 users at any given time. It runs as a web application, and as users work in the ACM office, they intermittently place votes for songs that they, as an individual, would like to hear. Based on user votes, *Acoustics* maintains a queue of songs, but if users stop voting, and the song queue empties, Acoustics randomly selects the next music to be played.

This random algorithm does not reflect user interests, and does not take advantage of information that we have about each user. To motivate our project, we consider this problem, and ask how we may use Acoustics data to determine common musical tastes of users. Likewise, Acoustics already maintains vote histories for users, so how can we take advantage of this data to improve the autonomous song selection functionality in Acoustics? Our project aims to answer these questions, and studies vote history that we have for each user, determining the musical tastes of each user by using tag information from the Last.fm API [Last.fm]. Our application uses this information to analyze common musical tastes of users in the room, and suggests songs that best fit the musical tastes of a given set of users.

*e-mail: tedesco1@illinois.edu
†e-mail: burke5@illinois.edu
‡e-mail: akreher2@illinois.edu
§e-mail: gladden1@illinois.edu

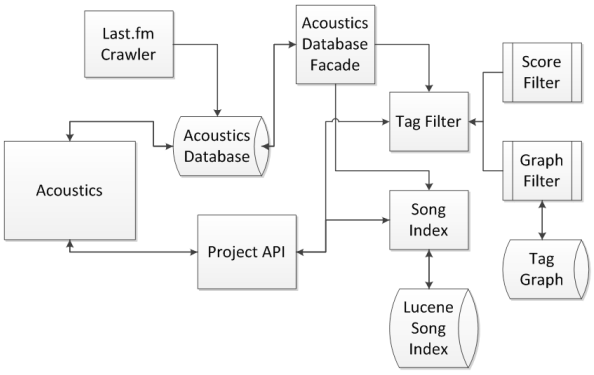
2 Architecture

2.1 Data Flow

When a query is submitted to our application, there are three conceptual steps that occur in processing the top-level query, which represents a set of users:

1. Receive API request to provide best songs, given a set of users
2. Find the set of high-level tags that best represent the musical interests of the group of users
3. Search for songs from the song index using these tags as a query

Below, we show a diagram of the high-level layout of our project, to better describe how each component interacts:



Our project essentially consisted of four major components:

1. Last.fm crawler

   Runs separately from the core Java code of our project, polls the Acoustics database for new songs, albums, and artists, and fetches tag information from Last.fm

2. Project API

   Provides a simple interface to our project from the core Acoustics code, and allows our project to be run separately from Acoustics

3. Tag filter

   Retrieves a ranked list of high-level musical tags that best represent the musical interests of a set of users

4. Song index

   Uses this list of tags as a query to generate a

pseudo-randomized playlist of songs to suggest to Acoustics

The role and implementation of each component is described in detail below.

3 Last.fm Crawler

This component of the project was run separately from the primary filtering components of the project, but was no less critical for our implementation. This was integrated into Acoustics, written in Perl, and crawled the Last.fm API for existing song, album, and artist data in the Acoustics database.

To initially gather the data required for our implementation, the crawler used 8 computers to fetch data in parallel from Last.fm for over 24 hours. Now, the entire 42,000+ song collection of Acoustics has been fully tagged with 200,000+ high-level music tags from Last.fm. The crawler works around constraints of the Last.FM API service, including the rate limits imposed. The crawler is capable of running on multiple machines concurrently, and this concurrency support is provided by using the database as a backing store.

The crawler provided several technical challenges that we dealt with throughout the duration of our project. The metadata from Last.fm was guaranteed to be neither high quality nor complete. In fact, some songs from Acoustics had no associated metadata in Last.fm, while many had excessive, and oftentimes useless, or even duplicate metadata associated with them. To address this, we cleaned the crawled data by removing excessively long or short tags and removed duplicate tags. Likewise, the Last.fm API restrictions put a rather heavy burden on crawling, as it allows only 5 requests per computer per second, and we needed to crawl 42,000+ songs while keeping a light load on the Acoustics server. To solve this problem, we allowed the crawler to be distributed across several machines.

4 Project API

This component of the project allows our filtering components of the project, written in Java, to be accessed asynchronously from the Acoustics system. The primary Acoustics application and our filtering project can be launched separately, and Acoustics uses this API to fetch song recommendations for users thought to be listening to the jukebox. Specifically, the API receives a list of netids, which represent users, and returns a list of songs that would best fit their musical interests as a group. While we plan to implement a more robust API that uses RPC, our current API simply listens for incoming connections on a specified port, and when a conneciton is made, returns results over the socket.

5 Tag Filter

This component represented the bulk of the implementation, and comprised of two alternative filtering methods, which we hope to combine for higher quality results. A *tag filter* in our project is defined by a generic interface that implements a `findBestRepresentativeTags` function, so that tag filters can be arbitrarily combined using their output from this function. This function receives a list of users, and using the vote history for each user, and tag information for Acoustics songs, returns a ranked list of musical tags that best represent the common musical interests of these users.

In our implementation, we created two tag filters; first, a traditional score-based filtering system, and second, a more aggressive graph-based filtering system. In each filter, we currently assume that song, album, and artist information is static, so that when our application receives a query, it does not check for new song, album, or artist information. However, since users may likely be voting and using our application simultaneously, we consider user history data to be much more volatile, and gather user history on demand from the Acoustics database. Since this requires relatively little data for a small subset of users, we believe this approach strikes a balance between speed and quality of performance.

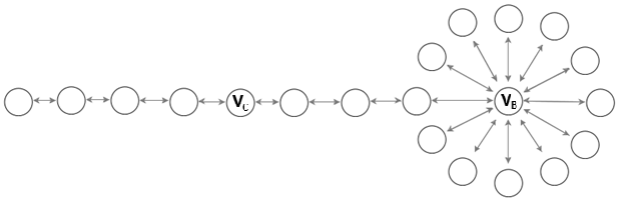
5.1 Score-Based Filtering

Our score-based tag filter uses an intuitive and traditional approach to determine tags for user's musical interests. Essentially, it considers the most recent votes for a user, and tags accumulate a score as we iterate through each song in a user's history. This score accumulator also weights the relative quantity of votes from each user, so that scores are not skewed by users with an exceptionally high total number of votes. This ensures that each user's musical interests are considered equally, and allows us to judge the musical interests of every user, even those for whom we have little information. As a side effect, this penalizes users with diverse musical tastes, which is consistent with the basic assumption that a user with diverse musical taste is likely to accept a wide variety of music. We believe this penalty is reasonable, and should not significantly affect the results. Likewise, this implementation is computationally efficient, requiring only one pass through the tags of each user, but with our optimizations, nonetheless yields reasonable results for queries.

5.2 Graph-Based Filtering

For our graph-based tag filter, we use a weighted, undirected graph to represent the relationship between users and tags. Specifically, each user and music tag represents a vertex of a graph, where edges represent a user's interest in a tag, connecting a tag vertex to a user vertex. Tags and users thus form independent partite sets, yielding a bipartite graph structure. We use the calculation of a *barycenter* to approximate the best tags in this graph.

The *barycenter* is essentially a calculation of the centers of the graph that yields a more accurate representation of a graph's centers than the traditional method, and mimics the standard calculation of the center of mass of a structure [Wikipedia 2011a]. Specifically, the traditional *center* of a graph is defined as the vertex whose longest path to any other vertex is minimum in the graph, so that distance in the graph is the primary measure of centers [Wikipedia 2011c]. On the other hand, the barycenter represents the vertex for which the sum of path lengths to all other vertices is minimized. For instance, consider the following graph diagram shown below:



While the traditional calculation of a graph center would yield the vertex located in the center of the graph, V_c, the barycenter calculation would yield the vertex V_b as the center of the graph. Conceptually then, we can think of the center as weighting the maximum distance over the total distances, where as the barycenter weights the total distances over maximum distance, and thus yields a more intuitively correct calculation of the center of a graph.

To use this calculation, for each vertex, we sum up the path lengths to all other vertices, and give a vertex this weight. The set of vertices with the lowest such weight then correspond to the best representative tags for the set of users. This calculation allows clusters of tags to form naturally, and we can improve this weighting scheme by assigning non-uniform edge weights to the graph. In our implementation, we assign different weights for tags that correspond to songs, artists, and albums. Although this demands a large amount of computation, we use the Floyd-Warshall shortest paths algorithm to find all shortest paths in the graph in $O(n^3)$ time [Wikipedia 2011b]. This is feasible given a small subset of the data, and cleaned Last.fm tags for vertices.

In our current implementation, we exclusively use the score-based filtering method for computational efficiency, but hope to incorporate this graph-based filtering implementation once we have more thoroughly cleaned and clustered the Last.fm tags.

6 Song Index

Once the best representative tags has been determined for a set of users, it is sent to a Lucene index of songs. In our Lucene index, we use song documents as objects, where their attributes consist of associated tags from Acoustics. Lucene uses this ranked list of tags as a query for songs, and returns a set of song suggestions. While these results match the tags given, they are not necessarily ranked, but rather represent a randomized subset of relevant songs, so that successive queries using the same user yield different song suggestions each time. This is done to ensure that extended use of our application does not yield redundant song suggestions.

7 Resources

Our project leverages several existing resources for each component of the project, which we have referenced throughout the paper. First, we make use of the Last.fm API, which allows us to fetch high-level tags for songs, artists, and albums [Last.fm]. This information was used heavily to determine the musical tastes of users in our system. Likewise, we make use of Lucene, an open-source Java retrieval toolkit, which use to search for songs given a query in terms of high-level music tags [Apache 2011]. Likewise, we make use of tools like MySQL, to persist Acoustics data, and JGraphT to implement a graph-based scoring algorithm for music tags [Oracle 2011][JGraphT 2011].

7.1 MySQL

Acoustics's data is persisted via a MySQL database, which we extended with crawled tag information for songs. When our extension to Acoustics is initialized, we gather all song data and their associated tags, which we assume changes infrequently. However, since users are continuosly voting and thus changing their musical tastes in Acoustics, we retrieve user information on demand from the Acoustics database. This way, we ensure that information on users' musical interests is accurate, even when changing rapidly. However, this two-stage fetch process ensures that our application also retrieves results quickly.

7.2 Last.fm

The Last.fm API was initially crawled using a Perl CPAN module to fetch tag data for each song in the Acoustics database [CPAN 2011]. This component was run independently from our core Java project, so that it can carefully monitor changes in the Acoustics database, and asynchronously fetch data from Last.fm while our extension executes independently.

7.3 JGraphT

The graph component of our project was built using a Java library called JGraphT, which abstracted the details of a graph implementation from our application, and allowed us to represent a weighted, undirected, bipartite graph with built-in graph algorithms such as the Floyd-Warshall Shortest Paths algorithm [Wikipedia 2011b].

7.4 Lucene

The Lucene component of our project indexes songs based on song tags, so that we can easily retrieve songs for a given list of high-level musical tags. Once the tag filtering component of our project determines the best representative set of tags for the current users, the tags are joined to form a query to Lucene, and Lucene is used to retrieve the songs, specifically song ids and song titles in our implementation, that best match this set of tags. We make use of Lucene's built-in query parser to delegate higher weights to earlier query terms from our concatenated tag query.

8 Evalution

In order to evaluate our project, we gathered feedback from the Acoustics user base via the Acoustics website. We added a socket-based interface to the Java application, and allows the website to return suggested songs to the user for evaluation. Specifically, for each test, we submitted a list of thirty songs to a set of users, fifteen of which were suggested by our application and fifteen of which were chosen randomly from the database, as Acoustics currently operates. These songs were shuffled and then displayed to the group of users, so that there was no visible indication as to which which song was selected via which method. The group of users then decided which songs were agreeable for all members of the group, and marked each song as either relevant or nonrelevant. The survey recorded the number of selected suggested songs, selected random songs, and number of voters.

9 Results

The results we gathered, shown in Figure 1, affirm that our project works as we expected. From our 47 trials run over 3 days, we displayed 1410 songs to users in total. Of these 1410 songs, user groups approved of 472, 115 of which were selected randomly, and 357 of which were selected by our project. Our project's suggestions improved over random suggestions by a factor of 3.10, and of the songs our project suggested to users, 50.64% were approved of by all members of the group. From our results, we can see that we have room to improve our suggestions, but as a whole, our suggestions were far more accurate than the random suggestion heuristic currently implemented in Acoustics. In the table shown in Figure 1, the first column displays the number of songs that our project suggested that the group of users found accurate, the second column represents the number of randomly selected songs that users found accurate, and the third column represents the number of users in the group for that trial.

Suggestions	**Random Selections**	**Users**
13	7	2
8	3	2
3	0	2
7	4	2
5	1	2
11	7	2
8	7	3
5	1	3
11	3	3
4	0	2
12	0	2
3	1	2
5	1	3
10	1	3
0	1	3
12	3	2
2	2	2
7	2	2
5	2	4
2	0	4
13	4	4
3	1	4
10	1	2
9	0	2
8	2	3
10	3	3
12	4	3
4	3	2
12	5	2
10	1	3
7	1	3
14	1	3
10	8	4
11	6	4
6	5	4
10	4	4
3	1	2
0	1	2
7	2	2
6	6	2
4	2	4
11	2	4
9	2	4
5	1	2
13	0	2
4	0	2
13	3	2
357	115	**Total**

Figure 1: *Experimental results from Acoustics users*

10 Conclusion

Through our project, we addressed Acoustics's problematic automated music selection system by designing and implementing a content-based music filtering recommendation system based on groups of users. Our addition to Acoustics, which consists of four major components, utilizes a two-stage process based on musical tastes of users to suggest a set of songs for a each group of users.

First, our crawler gathers high-level Last.fm music tags for each song, artist, and album. This crawler is implemented as an add-on to Acoustics that polls for new data in the database. Next, our API receives a query in the form of a set of users, gathering the tag and song data from Acoustics when our application first launches. Then, when a query is received, it retrieves a sorted list of tags that best represent the common musical interests of the users requested, and looks up a set of songs for these tags using our Lucene song index. Our index is pseudo-randomized, so that successive calls to our application yield various, equally relevant sets of song suggestions.

Over the course of three days, we evaluated our application's effectiveness using nearly fifty trials, each involving two to four users, and found that our application improves upon the current random song selection of Acoustics by a factor of three. We hope to improve the accuracy and performance of this project, and plan to integrate it into Acoustics in the near future.

11 Future Work

Although our application appears to be working well, we have thought of several ways in which it may be improved in the future.

11.1 Song Index Improvements

Although Lucene has extensive customization capabilities, we have not taken advantage of these in our current implementation. Queries to Lucene now are done by simply concatenating an ordered list of music tags, and while Lucene interprets the relative weights of these tags appropriately

using the default query parser, we can use a more elaborate querying scheme to improve the interaction between the tag filter and song index components.

11.2 Query Building

As of now, we have created a very simple interface to test different methods of filtering. While we have seen substantial improvements in the performance of our techniques using manual inspections of filter results, delving into a statistical approach to this analysis would yield higher quality results.

11.3 Tag Filter Improvements

Likewise, we have several ideas for improving the tag filter component of our project. Our project does not take advantage of both the graph-based tag filter and the score-based tag filter, but in the future, we hope to combine these tag filters to improve the quality of our suggestions, and specifically, of our calculation of a set of user's common musical tastes. Each of these methods of filtering has interesting implications, and could be combined for better performance.

11.4 Tag Data Improvements

For the tag data for our songs, we can try to cluster tags by similarity before analyzing common musical tastes. This would alleviate one of our major problems with the Last.fm API, in that many of the tags we retrieved, for songs in particular, were inaccurate or vague. Performing an additional cleaning and clustering stage would help reduce the number of tags we process and improve their quality, which would yield more accurate results, and faster computation, particularly with the graph-based filter.

11.5 Long-Term

In terms of long term improvements, we could expand our information about Acoustics users by combining voting data from Acoustics with data from user social sites such as Facebook. Likewise, we could expand our information about songs by retrieving supplemental data from sites such as Musicbraiz, Wikipedia, or Pandora [MusicBrainz][Wikipedia][Pandora]. This could be useful for filtering based on song lyrics, studios, musical eras, or any additional information about the songs.

Division of Labor

11.6 Jon

Jon initially focused on the Java data structures and architecture of the project. Likewise, he designed and implemented the graph-based tag filter, created the more traditional score-based tag filter, created the tag filter system to interchange filters, and implemented the randomized functionality of the Lucene song index. Jon also created the poster for the in-class presentations, and wrote research paper, given some snippets from other group members.

11.7 Rob

Rob focused on moving data into our program, building the data structures that encapsulate users and songs, and the architecture of the project. This involved implementing the DAO pattern to abstract out database interactions from the core Java implementation. Rob also designed the Lucene interface, which involved encapsulating all of Lucene dependent code into a class, and abstracted its API to a simple 'runQuery' method, that takes in a query of ordered tags, and returns a list of relevant songs.

11.8 Adrian

Adrian focused initially on the crawler implementation. In addition to providing knowledge of Acoustics, he helped with knowledge of the Last.FM data. He worked on optimization of the recommendation system, and provided knowledge of individual's music tastes to ensure accuracy. Adrian also primarily wrote the survey application, which was used to determine the accuracy of our recommendations.

11.9 Brian

Brian worked on improving the accuracy of the retrieval component, primarily dealing with Lucene. This work included increasing song variety by preventing duplicate artists. He worked on improving tag parsing and handling of tags representing multiple genres. Brian helped test and implement the survey application. He also came up with groups of people to survey, administered the survey, and exported the data from the database.

Acknowledgments

We would like to thank the ACM members who participated in our user study, as well as professor ChengXiang Zhai for his guidance and suggestions with our project and filtering strategies.

References

Apache. 2011. Lucene. http://lucene.apache.org/java/docs/features.html, May 2011.

CPAN. 2011. Cpan. http://www.cpan.org/, May 2011.

JGraphT. 2011. Jgrapht. http://www.jgrapht.org/, May 2011.

Kreher, A. 2011. Acoustics wiki. https://github.com/avuserow/amp/wiki/, May 2011.

Last.fm. Last.fm api. http://www.last.fm/api.

MusicBrainz. Musicbrainz. http://musicbrainz.org/.

Oracle. 2011. Mysql. http://www.mysql.com/, May 2011.

Pandora. Pandora internet radio. http://www.pandora.com/.

Wikipedia. Wikipedia. http://en.wikipedia.org/.

Wikipedia. 2011. Center of mass. http://en.wikipedia.org/wiki/Center_of_mass, May 2011.

Wikipedia. 2011. Floydwarshall algorithm. http://en.wikipedia.org/wiki/Floyd-Warshall_algorithm/, May 2011.

Wikipedia. 2011. Graph center. http://en.wikipedia.org/wiki/Graph_center, May 2011.