connect.fm Final Report

Javon Colvin-Hatchett, Daniel Shao, Ryan Tatton, Ethan Voss, David Ye
12 May 2021

Overview

Problem

With over 60 million songs, 345 million active users, and 155 million premium members, Spotify is the top music streaming service in the world (as of March 2021)[1]. It offers rich features on both desktop and mobile platforms for browsing, creating, and sharing playlists, but an investing social experience is lacking. A user is limited to either sharing links to songs, playlists, and artists, as well as to view what song or playlist their friends are currently listening to. Consumers are increasingly looking for novel digital experiences that provide unique avenues of entertainment. Though Spotify offers some recommendation features, there is a prime opportunity to offer innovative listening experiences that provide Spotify users with new ways to both enjoy and discover music.

Solution

We intend to create a recommendation system that takes into account a person's approximate distance to other users, their tastes, and gives a song that will both be popular to the general population of the area and is something that the user will enjoy listening to. It will be formatted in a continuous playlist, with updates at normal intervals.

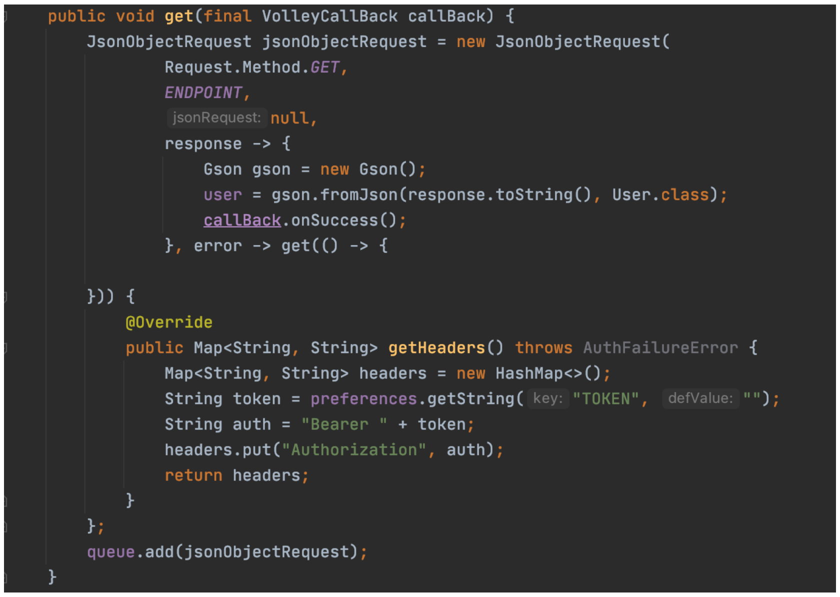
Goals

User Information

Retrieving User Information

User information is stored within Spotify's RESTful API, which requires a specific set of strings for authorization and proper querying of the API: the client id, the redirect uri, and the set of scopes the application would like to access. This information is then sent to Spotify's account service manager, which logs in the user and authorizes the application to act upon the user behalf in the ways described by the scopes. The application then requests access and refresh tokens. The access token is the main key that will be used for interactions with the web API while the refresh token is used once the access token has expired. Once we have this token set, we can now make HTTP calls to the Spotify API.

[1] https://en.wikipedia.org/wiki/Comparison_of_on-demand_music_streaming_services, based on the number of active users and paying users (as of 2020).



```
public void get(final VolleyCallBack callBack) {
    JsonObjectRequest jsonObjectRequest = new JsonObjectRequest(
            Request.Method.GET,
            ENDPOINT,
             jsonRequest: null,
            response -> {
                Gson gson = new Gson();
                user = gson.fromJson(response.toString(), User.class);
                callBack.onSuccess();
            }, error -> get(() -> {

    })) {
        @Override
        public Map<String, String> getHeaders() throws AuthFailureError {
            Map<String, String> headers = new HashMap<>();
            String token = preferences.getString( key: "TOKEN",  defValue: "");
            String auth = "Bearer " + token;
            headers.put("Authorization", auth);
            return headers;
        }
    };
    queue.add(jsonObjectRequest);
}
```

Figure 1: Example of information call and parse from Spotify API

When requesting information about the user from the Spotify API, it is given in JavaScript Object Notation (JSON) format. This is a standard file format within data transmitting which stores objects in key value pairs, arrays, and serializable values. Since it is an industry standard, it is very easy to work with this format. Java 8 has built in methods to both request and parse JSON format. In order to make a network call which will result in a JSON response, a *JsonObjectRequest* is used. It's constructor takes six parameters: the endpoint which the request is calling to, a Body object which will store all server-side information needed, a network request type (GET, PUT, POST, HEAD, etc.), a lambda function in the case of a successful request, a lambda function in the case of a failed network request, and a modifier for the header of the request. In this case of getting user information, the endpoint is Spotify's predefined endpoint for getting anything about any user, there is no body needed to retrieve information, the request type is GET, the response lambda will be explained later, the error lambda returns a Log of the network code, and the header is modified to include the access key.

When a response is received, the JSON must be parsed into its respective data structure. In the case of the user, there is a User item which stores the birth date, country, display name, email, and id of the user as Spotify has stored it. In order to transfer this information from a JSON request to a usable data structure, Java 8 has a data structure called Gson which transfers the given information into the data structure by relating the fields of the data structure to the keys of the JSON response. This cuts down on the time it would take to individually parse all of the information given about the user drastically.

One main issue with using JsonObjectRequest is that it is enacted asynchronously. While this is typically useful, due to the nature of how the information is being used, the data must have reached the device and been parsed before the next stage of the application can develop.Thus, in order to use the data, a VolleyCallBack interface was created. This interface contains a single method, *onSuccess(),* which gives the option to manipulate the data at the point in which we call this method. This is done through a lambda function where all manipulation is done within a different class. In the case of this application, this manipulation is storing an instance in the device's preferences, as well as storing/updating DataStore's column containing this information.

Storing And Processing User Information

Information gathered through Spotify's REST API is then stored using Amazon Web Services' DataStore service, which stores data on the user's device and in the AWS cloud, and syncs the data using AWS AppSync. Data like the user's listening history, preferences, location, and playlists are all stored and synced with the AWS cloud to be processed by our AWS-hosted algorithm. The data model for our application is in the image below.

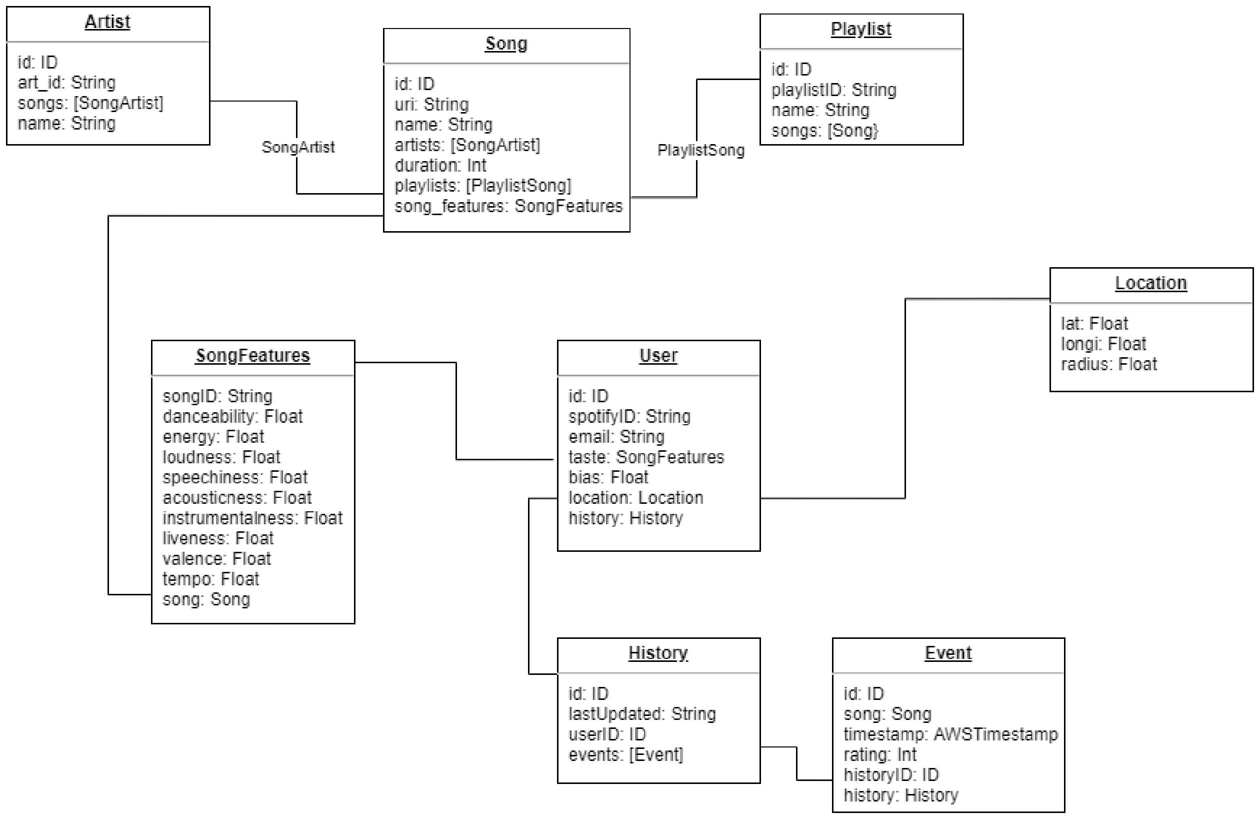


Figure 2: connect.fm data model

This data is then processed through a variety of AWS services in order to generate recommendations to the user. Figure 3 describes the generic data flow of connect.fm, while Figure 4 describes the specific AWS backend structure.

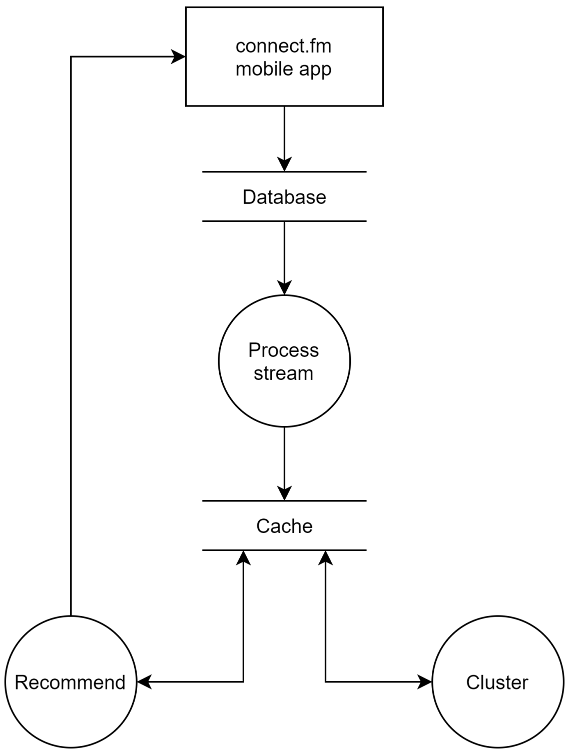


Figure 3: connect.fm data flow diagram. A rectangle represents an input/output. A circle represents a function. Two parallel lines represent a data store.

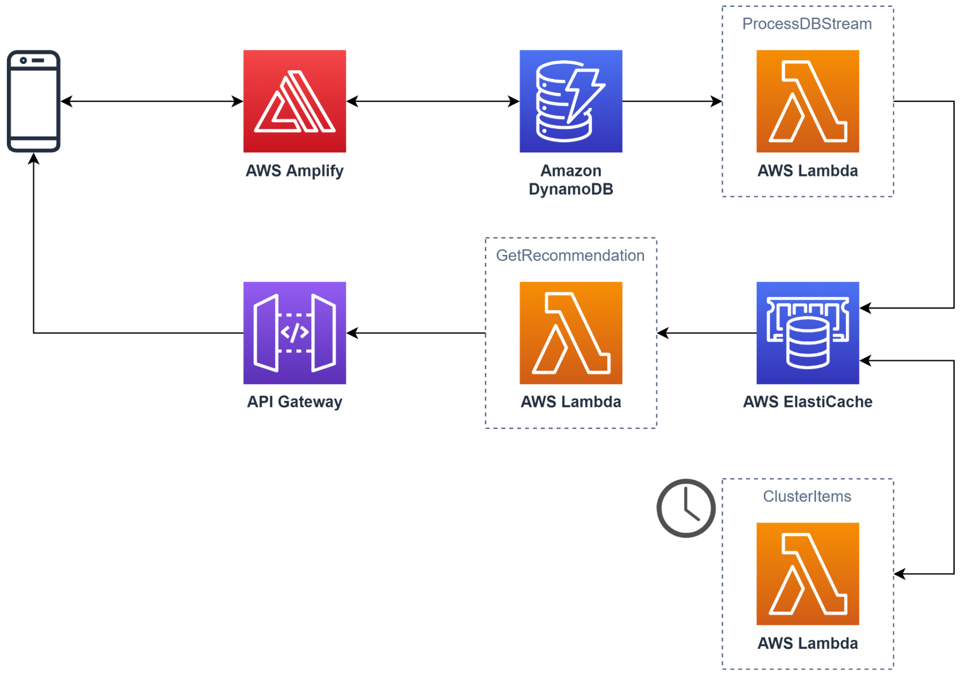


Figure 4: connect.fm AWS backend architecture.

AWS Amplify provides the gateway to other AWS services to process data through. Amplify connects our application with DynamoDB through AWS DataStore and AppSync. This data is then processed through AWS Lambda in our ProcessDB stream instance, which streamlines the data. The data is then loaded in AWS ElastiCache, along with our cluster information from the ClusterItems instance of AWS Lambda, in order to provide faster access to data for our recommendation algorithm. Finally, the data is then processed through the GetRecommendation instance of AWS Lambda and through our API Gateway to provide our application with our final recommendations.

Maintaining Privacy of User Information

In addition to user data from the Spotify API, we also need user location data. However, this information is sensitive, as it is easily abused. So, we take measures to maintain the privacy of our users by securing this information at all times.

User location data is gathered from Android's Location SDK through the FusedLocationProviderClient which provides a simple way to periodically retrieve location data on a background thread. It requires two permissions: ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION. As the latter permission gives granularity that could be considered sensitive, we anonymize the data by converting it to a geohash before sending it to Amplify DataStore. This ensures that sensitive data is never transmitted off the device, and is never stored anywhere in our databases.

In addition to the active measure of using geohashes, we take several other precautions to maintain user privacy. For example, we minimize the instances of geolocation data in transit, even when anonymized as a geohash. This is accomplished by storing the current geohash on a background thread, and only sending a location update to DataStore when the calculated geohash changes. Additionally, due to the fact that location permissions are runtime permissions, connect.fm can only request location updates from the system when the app is in the foreground. This ensures that we never can or will covertly harvest data without your approval. Our integration of AWS services also affords us the privacy protections built into their systems: AWS services encrypt data that passes through their systems both in transit and at rest, which affords additional protections to all user data that passes through these services.

Song Recommendation

We generate song recommendations from a custom-built algorithm developed by team members Daniel Shao, Ryan Tatton, and Alexander Sfakianos after a literature review of existing recommendations systems. The algorithm consists of a clustering phase to group similar songs into clusters, and a sampling phase which uses the tastes of users and their neighbors to identify appropriate song clusters. We discuss the details of the recommendation system, as a feature, later in the report.

Clustering

The immense selection of 60 million songs in the Spotify library poses a substantial computational challenge for identifying suitable song recommendations. A naive approach for identifying the best-suited song recommendation would require an exhaustive search of the entire library for the best song, each time we wanted to recommend a song. This is obviously infeasible. To reduce complexity, we organize songs into clusters, where similar songs are grouped into similar clusters. We follow the assumption of content-based recommendation systems that if a user likes one song in a cluster, they are likely to enjoy all songs in the cluster. Notice that since we represent users and songs in the same vector space, we can extend this assumption to recommending songs similar to the user. This vastly reduces the running time and storage complexity since we need to only sample from the entire cluster. That is, rather than scaling linearly in the number of songs, it only scales in the number of clusters, which is less than the number of songs by several orders of magnitude.

After a survey of potential clustering algorithms, we settled on the Dirichlet process mixture model (DPMM), one of the most popular probabilistic clustering methods for non-parametric clustering. This model fulfills the three requirements of being non-parametric, computationally efficient at high dimensions, and probabilistic. Intuitively, the Dirichlet mixture models function as a distribution of distributions. Under the DPMM, a Dirichlet probability distribution is defined for a set of probability measures, where each probability measure is itself a probability distribution over the sample space. In the context of our song space, each probability measure defines the likelihood that a song belongs to a given cluster, and the dirichlet distribution defines the likelihood of each probability measure to describe each song cluster.

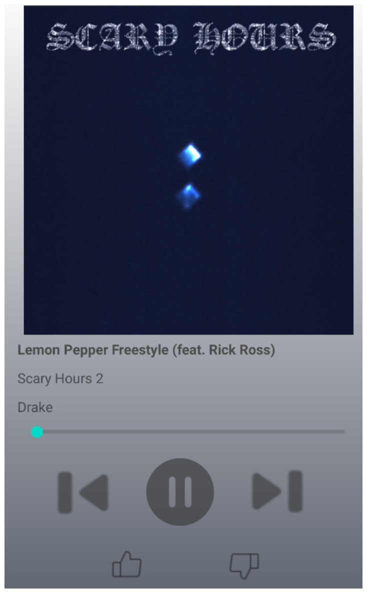


Figure 5: connect.fm Playback Fragment

Playback

Playback works in a similar way to how accessing user information works. In this case, the network request contains the endpoint relating to the state playback needs to be in ({state} is directed to https://api.spotfy.com/v1/me/player/{state}),parameters storing the id of the device that the music is to be played on and the point within the song to begin playing if needed, the request type PUT, empty cases for the response case, handlers for the error case, and a header modifier that includes the access key for the user. The response parameter does not need to be addressed since the only thing that will return is a network response code.

Features

Liking and Disliking Songs

Status: *Complete*
Based on if the like or dislike button is pressed, the method *setStatus()* within the spotifyFramework.Song class. Song status is determined on a scale of one to three; One is the state in which a user dislikes the song, two is the state in which no decision has been made, and three is the state in which the user likes the song.

Find Nearby Users

Status: *Complete*
A core feature of connect.fm is its ability to immerse you in your surroundings by recommending you music from the people around you. A HandlerThread queries for user location, storing the resulting geohash and sending it to DataStore. Whenever this geohash changes, it will send the new geohash. Our algorithm prioritizes users in the same or adjacent geohashes, selecting music from other nearby users to build a playlist.

When retrieving nearby users for recommendation, we utilize the Redis geospatial API[2] which allows us to efficiently and easily update user location and retrieve nearby users.

Recommend Songs to User from People Nearby

Status: *Complete*
The recommender system is fully functional as a feature, but is not yet integrated into the application since DataStore is not yet fully developed. The technical paper discussing the theory and some preliminary results of the recommender system is attached to the end of this report.

User Playback of Recommended Songs

Status: *Incomplete*
While playback has been fully implemented and tested with a set of Spotify playlists, the method in which the application receives songs to play has not been implemented. We plan on having this implemented before the presentation date.

Adjustable Radius Slider

Status: *Complete with Modification*
No longer a slider, but rather a box in which the user can input the exact distance from themselves they would like to use (distance is in miles). Also added a function to set the bias to a specific amount, with zero being just the user's taste being taken into account and one hundred being taking into account purely the people around the user.

[2] https://redislabs.com/redis-best-practices/indexing-patterns/geospatial/

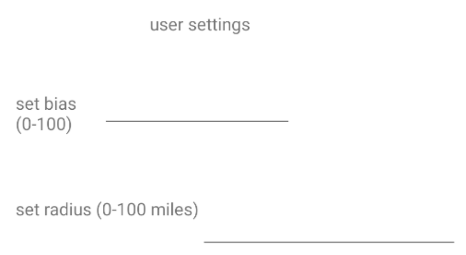


Figure 6: Adjustable radius and bias settings in app

Addition of recommended songs to personal list of saved songs on Spotify

Status: *Incomplete*
This system is dependent on some frontend development not completely understood, but will more than likely be implemented before the final presentation.

Testing

Testing of frontend and all Java methods was done by hand due to the cost of having a Spotify premium account throughout the semester as well as to preserve the privacy of all members' personal accounts.

The recommender system was implemented in its entirety prior to testing. Static code analysis was provided by the PyCharm IDE. Once implemented, Redis was set up locally. Simulated user and song data was used to iteratively debug and get the system into a working state. Obviously, this is not a very robust approach to testing because the functionality is limited to the extent that the simulated data covers the code's functionality. However, for the purposes of this project, this approach allowed us to implement as fast as possible.

Usage Cases

The final product is a streamlined app experience that allows the user to login with their Spotify account, give permission for their location to be shared with the application, and receive personalized recommendations based on their location and what people around them are listening to.

To demonstrate the user usage case, consider taking a vacation to a different island, wanting to fully immerse themselves in the culture, especially in the music. The user then logs into our application on their Android phone, on an island in the middle of the Carribean. The application then pulls up a radio that fully integrates the user's own tastes into the locals' tastes, giving the user a totally new experience, integrating the users' own tastes with the locals' music.

The user can then totally immerse themselves in the local culture and music that is most similar to their own music tastes. The user can now get playback of each song that is recommended to them based on their music tastes, and influence the recommendations based on their ratings of the songs that are recommended to them.

Individual Report: Ethan Voss

Project Responsibilities

I was responsible for the project's storage solutions, both frontend and backend. I was responsible for creating the data model and integrating it into our frontend application and integrating it into our other backend solutions in AWS.

Achieving Objectives

I used AWS's Amplify DataStore as our storage solution. DataStore provides a streamlined way to generate a data model, generate code to help with implementation in the frontend, and connect the storage to the other backend data processing services.

Through Amplify DataStore, a lot of integration code was generated in order to complete some objectives by the Amplify Command Line Interface (CLI). That code was then used to integrate our storage solutions into our frontend. The other backend solutions (including multiple instances of AWS Lambda and AWS ElastiCache) were then coordinated using the Amplify CLI.

Issues

Staying true to our timeframe was the biggest issue I had with implementing my part of the project. I especially got behind when I had the COVID-19 virus for two weeks during the semester, which completely threw off the plans we had for completing the project on time. My lack of time management really threw off the project, and limited what we could get done for the final project. My contribution to the project was a very substantial piece in getting everything to work together, but when everything got derailed by my absence because of illness, we didn't have a good contingency plan to deal with it.

Aside from my time management, some issues that I faced included getting the data model to accurately represent the data in our project, and learning the basics of AWS Amplify in order to properly set up the backend. The data model and Amplify were interesting to figure out, since I had never used any of the technologies used before.

Lessons Learned

The biggest lesson I learned this semester was about communicating with my team better about how I was doing and if I needed help completing the objectives I was tasked with. I definitely lacked in keeping up with the progress of my other teammates, but also did not fully inform them of my predicaments with adhering to the timeframes that our group had initially set out. If I had fully communicated with them throughout the semester, our team would have been much further along with our implementation of our features.

The other big lesson I learned was to rely on my teammates more, since they are also very capable with the software applications and integrations that I am. I continually find myself relying on my own knowledge, rather than that of others in order to see myself through projects that my teammates are more than capable of doing as well.

Insights

Software development is both an individual task and a group task, and if one is unable to complete a task, the entire software group shouldn’t suffer because of a single member’s ineptitude. The group should function without a single members’ contribution, but this isn’t a pass to not do something. The group should be mobile and fluid, and help cover other people’s losses, even if it might damage the rest of the group.

Software development is also really fun. Demonstrating that you can tackle a problem within a certain set of boundaries is very fun.

Individual Report: Javon Colvin-Hatchett

Project Responsibilities

I had two primary responsibilities within the creation of connect.fm: frontend design and Spotify API network calling and handling.

Achieving Objectives

For the frontend design, the objective was to make a front end design that was attractive to the majority of the team. I began drafting frontend sketches for some of the ideas I thought would be good to both use and distinguish ourselves from other media players. I first landed on a nostalgic theme, which meant to play on the "radio" aspect of our name and our playback methodology (pulling more songs to play while other songs were currently being played). I used red and grey as the theme colors and found a custom font that was reminiscent of radio station billboards in the 70s and 80s. This came as a controversial idea, as some people thought it looked clunky and old. The next time, I decided to make a more modern theme, more reminiscent of Spotify. I used a grey-to-white gradient as the theme color, and used the default font given by the Android suite. This was more acceptable but was still seen as a little bit window heavy and overburdened with features. I stripped some of the features while streamlining others into a bottom navigation board (with some assistance from Daniel Shao). This ended up being our vision for what the application would be. I got to work on creating what would be the frontend of the application. This would include making recyclerviews for the recently listened to section of the application after logging in, A login page which the user can authenticate their Spotify account, a playback page that allows for the user to skip, replay, go back to, and scrub through, the music they are currently listening to, and a splash page with animation of our custom logo made by Ethan Voss.

For the Spotify API, the objective was to develop as many network call functions as possible to make the algorithm design as plug-and-play as possible. In this, there are three main classes, marked with the name "Service" at the end of their class name. These services are network calls to different endpoints within the Spotify API (UserService works with the user based endpoints, SongService works with the song based endpoints, etc.). Depending on the function of the method within the class, the endpoint and the response will be different. Thus, it was also my responsibility to properly parse and handle these different endpoints and responses. These responses would be stored by the needed values into a data structure, both locally and in an external database.

Issues

The main issue I had within the development process was timing. We had a schedule that was pretty tight, and I wanted to stick to it. While researching, it seemed very realistic for us to make all of our team set deadlines. About halfway through the semester though, Ethan got COVID-19. This basically derailed our entire timetable, since all of the sections are very much so tied to one another. We've tried to work through this, but it's becoming increasingly likely that we finish this semester without a minimum viable product. This is not a slight to Ethan at all, but more of a lesson I had to internalize about being flexible in timelines.

Another issue I had was learning how to properly design frontend designs. When working on other applications, I was primarily a backend developer. In this project though, I was going to work on the frontend along with David Ye, who has done front end development before. When David had other responsibilities that took the majority of his time though, I became the sole developer. I ended up learning a lot of xml and how Android activities and fragments work. Over time, I became relatively comfortable with the framework, but it was definitely a learning curve.

Lessons Learned

I have to be cognizant of others' schedules. When I have worked in groups before, everyone is working on the same things during that time. For this project though, people were working on different projects to combine into one giant completed project. It made looking at other people's progress difficult and personally stressful. I had to learn to just focus on what my jobs are and wait for others to finish their development process.

Insights

Independent work is very freeing, but also very frustrating. There's a lot of stuff you have to learn on your own, and it rarely works properly the first time you try it. Then, fixing those issues takes forever since you don't have anyone to consult about your issues that has a concrete solution relatively quickly.

Individual Report: Ryan Tatton

Project Responsibilities

My contributions to this project entail:

1. being one of the unofficial leads on the project,
2. designing the connect.fm AWS architecture, and
3. implementing the recommender system (excluding clustering and stream processing)

With respect to (1), Javon and I have served as the two “unofficial” leads of the project. I use quotes because we organized the app development by *feature* (not *role*), but Javon and I have contributed significant amounts of work both in source code and completing deliverables, particularly the progress reports. During the first couple of weeks of the semester, after forming our team, I took on the additional responsibility of organizing meeting times and keeping meeting minutes. However, this lessened as the semester progressed, particularly once we began implementation.

My work done for (2) has been a considerable focus of mine this semester as the recommender system, including clustering, was also the topic of my project for a graphical probabilistic models course. As part of this component of the application, I worked with Daniel Shao and Alexander Sfakianos (not in CSDS 395) to collect and read over 40 research papers on recommender systems and clustering. Once we settled on the approach, I took on the significant responsibility of implementing the recommendation component, which assumes songs have been clustered and user data from the mobile application has been processed and stored in ElastiCache. Tangential to this effort was designing a database schema that would allow us to access user and song information that is relevant to recommendation in a scalable manner.

Achieving Objectives

As stated in Section 1, I was responsible for implementing the recommender system. Ultimately, I wrote approximately 1300 lines of working code, of which I am incredibly proud. I was able to fully satisfy my responsibilities for this project.

Issues

After initially implementing the recommender system, I realized that the original version of our algorithm was not scalable. To compute probabilities over a collection of values, you must normalize them. This normalization step scales linearly in the number of members in the collection. Assuming we do not allow the user to skip the currently playing song, we conservatively have about 1 minute to retrieve another recommendation. However, when locally testing with 100 thousand songs, the recommendation took about 3 minutes.

I initially tried to resolve this by implementing a “fuzzy” matching component that was intended to increase the number of times we could use the cached normalization values. This, unfortunately, did not significantly or consistently help since we still end up with long runtimes whenever we do not have the values cached.

The final solution was to sample a subset of songs from the clusters to approximate the probability values. While it only allows a subset of the songs to be recommended per recommendation, it scales infinitely with the number of users and songs since we set an upper bound on the number of neighboring users and songs to sample. Locally tested with sampling 1000 songs and 100 nearest users, it takes about 3 seconds to recommend a song, which is sufficient for our purposes. Note that this running time does not depend on the total number of songs or users.

Lessons Learned

Given my contributions in Section 1, I learned several valuable lessons, including

1. how to adapt a current approach to be scalable,
2. what current approaches exist for recommender systems, and
3. how to manage a long-term project schedule to satisfy my responsibilities.

Insights

Recommender systems are pervasive in entertainment services. The approaches used vary considerably and have associated trade-offs. Designing the AWS architecture of connect.fm made me aware of the data flow. While I did not work much on the application itself, I am now aware of the expansive capabilities of AWS Amplify. With regard to our recommender system design, it was incredibly rewarding developing it from an idea to working code, and hope to submit the paper I helped author on it over the coming year.

Individual Report: Daniel Shao

Project Responsibilities

I had two primary roles in this project: algorithmic design and implementation of music recommendation, and a user-settings page.

For algorithmic design, we began by performing a literature review of 40 total papers on existing recommendation systems. I took extensive notes on my component, which was 15 total papers. The full connect.fm team met 3 times a week, and the algorithmic design team met once a week.

I had a close hand in the design of the algorithm, which ultimately had a clustering step and a sampling step. I took the lead on the clustering algorithm, performing a private literature review of potential clustering algorithms to find an algorithm which fits our needs. The criteria I had when searching were as follows:

Non-parametric in the number of clusters: Due to the immense size and high feature dimensionality of our clusters, it is difficult to qualitatively estimate the optimal number of clusters from the data. Consequently, we search for a method which implicitly optimizes the cluster count based on the data's structure. Although this optimization adds a constant cost to running time, its benefit in the quality of our clusters is well-worth the cost.

Computationally efficient in high dimensions: Due to the immense number of entries and non-trivial dimensionality of our feature set, we require a clustering method which remains feasible at high dimensionality. This eliminates a large subset of clustering algorithms, including hierarchical clustering, k-means, and many density-based approaches.

Probabilistic: We desire a probability density function-based approach in order to capture the variance in a user's music taste. A person's music taste varies each day with mood and environment, such that if we were to choose songs that perfectly match the user's music taste, then music would become repetitive. Furthermore, variety is a key component of the listening experience. If songs with the exact same cadence, rhythm, and beat were repeatedly recommended in a single listening session, the music would become predictable and boring. Consequently, non-probabilistic clustering methods such as k-means are undesirable, as the songs in a cluster would lack variance.

I ultimately settled on dirichlet mixture models, which satisfied all our requirements. After narrowing down our model, I learned how to create a dirichlet mixture model with tensorflow, and I performed the clustering and saved the results in a JSON file for the song sampling step.

I additionally created a user settings page for our android application which allows the user to set their radius preference and algorithmic bias value. These values are saved locally and also sent to our AWS Datastore server to be used during song recommendation.

Achieving Objectives

Both objectives of clustering and user settings were completed properly.

Issues

The dataset of 60,000,000 songs was too large for dirichlet mixture models to cluster within the bounds of my RAM as well as google colab pro’s available ram of 27 GB. To reduce runtime, we instead clustered 10,000 songs as a proof of concept.

Lessons Learned

A thorough literature review is key to developing a proper algorithm, regardless of initial ideas. Even though we didn’t use any existing recommendation systems,the process of reading about existing algorithms served as important inspiration for our ultimate algorithm.

Insights

Unsupervised learning is highly beneficial, but also extremely challenging, for the process of item clustering. This is because we have no control over which features the algorithm considers important, and the clustering algorithm must identify meaningful features on its own.

Additionally, recommendation system performance is difficult to evaluate systematically, because there is no ground-truth for whether a single recommendation was effective or not. The ideal would be to provide the recommendations to a large enough number of users, and evaluate the ratio of likes to dislikes. However this is difficult to do in an experimental, objective setting.

Individual Report: David Ye

Project Responsibilities

Initially, project responsibilities were not defined very strictly. Javon, Ethan, and myself were assigned to the app implementation, while Daniel and Ryan worked on the machine learning algorithmic component. While I initially assisted Javon with some of the planning and mockups for UI design, my responsibilities transitioned to retrieving geolocation data from the device and sending it to DataStore.

Achieving Objectives

To receive location data, two goals needed to be accomplished. First, permissions needed to be granted by the device and the user to access location data. Second, I needed to write some method of regularly receiving data and sending it to the database when necessary. When achieving these two goals, user privacy had to be maintained the entire time.

The types of permissions needed from Android were for location data, which (from Android Marshmallow onwards) is classified as a sensitive permission with security risks: a runtime permission. As such, the permission could not be retrieved at compile-time, and had to be actively given by the user. In previous android projects, I've experienced significant difficulty with Android permissions for various reasons, so in this project I used the EasyPermissions library to drastically streamline this process.

Location data was retrieved using the FusedLocationProviderClient, which is designed to periodically request location data. It also is easily compatible with background threads, so I created a HandlerThread to periodically check that the user's location hasn't significantly changed. It does this by converting a Location object into a 5-digit geohash, via the Android-Geohash library. This anonymizes a precise GPS location into a 5x5 km box. The HandlerThread stores this geohash, and sends an updated geohash to the database if it ever changes. This ensures user privacy while minimizing unnecessary transmission of data.

Issues

As privacy was one of our priorities, we initially wanted to bypass the ACCESS_FINE_LOCATION permission entirely, as we thought it would be risky to even access fine location data from the user. However, due to the constraints of the libraries we used to retrieve location information, the permission ended up being necessary. To counteract this, we needed to implement privacy solutions elsewhere (as mentioned above in the group report) such as geohashing and strictly limiting the flow of user information out of the device.

Another issue in this process was the fact that Android has many classes that achieve similar goals, but some may be deprecated or inappropriate for a certain use case for reasons that are not obvious. For example, Android provides LocationListener, LocationManager, and FusedLocationProviderClient classes to get location data. Additionally, many online sources have outdated, conflicting, or simply incorrect information on how to correctly use these classes. A compounding issue was that we tested with Android emulators, on which emulated location is known to be unreliable.

These issues were resolved with exhaustive debugging, and I eventually settled on using FusedLocationProviderClient, using the requestLocationUpdates() method. While this class also has getLastLocation(), this method was continually returning outdated coordinates. Additionally, I found that this method required the ACCESS_FINE_LOCATION permission.

Lessons Learned

I gained some experience in dealing with large, sprawling languages in which there is no canonical approach to a problem. Because Android is constantly modifying and replacing features, it takes some dedication to determine the best way to accomplish a task.

I also learned that group projects often require a measure of patience, as some of my functionality was dependent on code from my team members being in place. Because the rollout of my code was dependent on theirs being in place, I wrote placeholder code that was near completion, with only a few changes necessary once my team members finished their tasks.

Insights

Group projects with hardworking and skillful team members is an engaging and beneficial experience. In past projects, some of my team members have been unreliable or unable to deliver work to a reasonable standard, which can often lead to very stressful situations. However, when all team members are proficient in their trade, we can quickly brainstorm to tackle issues together that would take far longer to solve individually.

Context-Aware, Scalable, Probabilistic Music Recommendation with Sampling and Clustering

Alexander Sfakianos alexander.sfakianos@case.edu
Department of Computer and Data Sciences
Case Western Reserve Univeristy
Cleveland, OH 44106-1712, USA

Daniel Shao daniel.shao@case.edu
Department of Computer and Data Sciences
Case Western Reserve Univeristy
Cleveland, OH 44106-1712, USA

Ryan Tatton ryan.tatton@case.edu
Department of Computer and Data Sciences
Case Western Reserve Univeristy
Cleveland, OH 44106-1712, USA

Abstract

Many modern streaming services seek to provide a continuous stream of content across a vast selection by utilizing recommendation systems. Recommendation systems seek to recommend new content based on various contexts, such as individual user preferences and platform-wide trends. We seek to create our own recommendation system, *connect.fm*, operating on top of the Spotify API to provide an alternative source for music recommendation. We utilize a unique component referred to as *stations* to establish connections between users based on their geographic location. Stations provide an alternative source of recommendations that works alongside more traditional recommender approaches. In addition to our stations, we interpret several contexts (user ratings, history, nearby users, and bias specified by the user) to group users. We generate a clustering of songs in our search space to facilitate an online learning environment. New recommendations are suggested through ancestral sampling in which our system samples over users, clusters, and songs.

Keywords: Context awareness, Clustering, Probabilistic graphical model, Recommendation system

1. Introduction

The objective of a *recommendation system* is to provide "optimal" items to a user. The method used to provide these items is unique to the recommendation system. The field of recommendation systems is diverse and rapidly developing with the accelerating usage of the Internet and personalized services. Recommendation may be based on a variety of factors, including user attributes, item attributes, item-item relationships, user-item relationships, user-user relationships, and contextual data.

Recommendation systems are ubiquitous in music streaming services. Streaming services, such as Spotify or Pandora, provide platforms that recommend media based on a the taste profile of a user. Both services attempt to learn based on the listening history of a user. While these services may incorporate collaborative filtering into their recommendation system, they do not offer the means to dynamically recommend based on nearby users. Rather than basing a recommendation entirely on a the taste profile of a user, we seek to integrate the taste profile of nearby users to supplement recommendation process.

We chose to develop a recommendation system based on Spotify's library of music and song features. With over 60 million songs, 280 million active users, and 144 million premium members, Spotify is the top music streaming service in the world. It offers rich features on both desktop and mobile platforms for browsing, creating, and sharing playlists. Consequently, we believe a recommendation system built from Spotify's enormous library and vibrant community will provide a novel and meaningful means to discover new music within the context of one's community.

connect.fm is an intelligent music streaming mobile application built on top of the Spotify API. A central concept in connect.fm is a *station*: a personalized, context-aware music recommendation stream that utilizes the music preferences of the listener, as well as those of nearby users. By considering the music tastes of others, we provide Spotify users a new way to discover music and feel a closer connection with their community. This is a generalization over Spotify's current recommendation system, Discover Weekly, which provides a curated playlist based on an individual's listening history. Our system is distinct in that it considers the listening history of users and those in their geographical location, can incorporate user feedback to develop more precise understandings of user taste, and utilizes probabilistic clustering efficiently recommend songs to users. Ultimately, the recommendation system of connect.fm should be (1) probabilistic, to address the uncertainty of recommendation; and (2) online, to allow for efficient and continual improvements to recommendations.

The remainder of the paper is organized as follows. Section 2 provides a discussion of the literature reviewed for this paper. Section 3 explains the methodology, scalability, and implementation of our recommendation system. Section 4 evaluates the ability and efficiency of our recommendation system. Section 5 discusses future directions for our project. Section 6 is offers a summary of our work. Lastly, Section 7 outlines the contributions of each author.

2. Related Works

In this section, we provide an overview over existing clustering methods and recommendation systems, as well as commentary on the feasibility of these methods for our specific goal of generating song recommendations.

2.1 Recommendation systems

Recommendation systems have been approached with various methods in the past. In this section, we will discuss some of the recommendation systems designs that influenced our approach. An intuitive approach to recommendation systems might be to observe trends in the userspace. For example, if there exist two very similar users u_1 and u_2, we can recommend u_1's most popular items to u_2. The assumption here is that just because u_1 and u_2 are similar, they will have similar tastes. The method for grouping users is key in this approach. (Renaud-Deputter et al., 2013) explores this method for building a recommendation system. Our approach doesn't seek to group users in such a way, but we can compare users with their taste profiles and geographic location. It's also worth noting that our recommendation system uses a similarity measure when grouping items rather than users.

A couple of the recommendation systems attempt to relate a user to the user space for additional context when producing recommendations. Social networks may be used to relate a user to many other users. This can be helpful when deriving the probability that a user will like a recommended item based on users that they are similar to within the social network as discussed in (Chaney et al., 2015). Alternatively, users may be grouped into social networks based on a personalized Point of Interest (POI) approach (Kulkarni and Rodd, 2020). This was demonstrated with an example of the locations being different buildings and the items being what you might purchase at the building (e.g. a restaurant might lead the recommendation system to recommend various foods). We draw from both of these social network approaches with our usage of β. A user has the option to choose how much impact the userspace affects their recommendations by specifying β.

2.2 Clustering

Clustering is commonly applied in recommendation systems to create organization within a massive catalog of items. Consider Amazon's immense asssortment of products: each product is grouped into a category, such as cooking supplies, games, electronics, etc. Consumers who frequently shop for electronics will receive recommendations of items categorized as electronics (Linden et al., 2003). Evidently, the categorization of each item in a catalog is key to providing tailored recommendations while circumventing the need to exhaustively search the entire catalogue for suitable recommendations.

There exist a variety of methods for unsupervised clustering outlined by (Xu and Wunsch). *Hierarchical clustering* produces a dendrogram where the leaves are the data points and the interior nodes are cutoff criteria. Hierarchical clustering is intuitive and visualizing its output is easy to interpret, but it performs poorly with high dimensional datasets. *Partitional clustering* defines the number of clusters beforehand, and group objects into K clusters based on a similarity or distance metric. This method was also insufficiently efficient for high dimensional data. *Mixture-based clustering* assume each data point is generated from a unique probability distribution, where each probability distribution is a cluster. Different methods can be derived from different density functions (e.g., Gaussian vs t-distribution), or just have different parameters. These models, however, assume that the data can be well modeled by the chosen distribution. *Graph-based clustering* create a graph where each vertex is an item. Weighted edges represent the distance between two items. Numerous concepts in graph theory, including mincuts and minimum spanning trees, can then be applied to find suitable clusters. These methods are highly interpretable, but suffer from large reductions in running time as dimensionality increases. Lastly, we observe *fuzzy clustering*. Under this formulation, members can belong to multiple clusters. This provides an interesting insight, in which users with markedly different tastes may have some songs which they both enjoy. Under a fuzzy clustering approach, those shared songs can exist in multiple clusters with high likelihood. We settle on the Dirichlet process mixture model (DPMM). These models are computationally efficient, which is essential to perform clustering of a large, high-dimensional dataset. Furthermore, the probabilistic nature of these models reflects variance in user preference.

3. Methodology

3.1 Notation Convention

A non-bold, lowercase x denotes a scalar, function, or a generic variable. A non-bold, uppercase X denotes a random variable or constant. A bold, lowercase $\mathbf{x}$ denotes a vector or sequence of scalars. A bold, uppercase $\mathbf{X}$ denotes a matrix or column-wise sequence of non-scalars. A calligraphic, uppercase $\mathcal{X}$ denotes a set or distribution. We denote a single indexed value as either $x_i \in \mathbf{x}$ or $\mathbf{x}_i \in \mathbf{X}$. For indexed sequences, we use the notation $x_{i:j} \in \mathbf{x}$ or $\mathbf{x}_{i:j} \in \mathbf{X}$ where the elements at indices i, j are included. Note that $x_{i:j}$ is a vector of dimension $(j-i) \times 1$ and $\mathbf{x}_{i:j}$ is a matrix of dimension $D \times (j-i)$ where D is the column dimension. Finally, a blackboard, uppercase $\mathbb{X}$ denotes a mathematical field.

3.2 Concepts

Before discussing the sampling and clustering components of our approach, we must define some terminology and concepts. A *song* $\mathbf{s} \in \mathcal{S}$ is a D-dimensional vector of features, where $\mathcal{S}$ is the set of all songs that we can recommend. These features may be explicitly provided or implicitly learned. Additionally, they may be continuous, discrete, or some arbitrary combination thereof. We assume only continuous features. Namely, we use a subset of the available features provided by Spotify (2021):

$$\mathbf{s} = \begin{bmatrix} \textit{danceability} \\ \textit{energy} \\ \textit{loudness} \\ \textit{speechiness} \\ \textit{accousticness} \\ \textit{instrumentalness} \\ \textit{liveness} \\ \textit{valence} \\ \textit{tempo} \end{bmatrix} \in [0,1]^9 \tag{1}$$

All but loudness and tempo are already normalized between 0 and 1. For both loudness (-60 to 0 dB) and tempo (1 to 300[1] bpm), we apply min-max normalization,

$$\tilde{x} = \frac{x - x_{\min}}{x_{\max} - x_{\min}} \tag{2}$$

The music *taste* $\mathbf{t}$ of a user is a function over their listening history,

$$\mathbf{t}_t = f(\mathbf{s}_{1:t-1}) \tag{3}$$

We use the simple yet, computationally-efficient approach of *exponential smoothing*,

$$\mathbf{t}_t = \alpha \mathbf{s}_t + (1-\alpha)\mathbf{t}_{t-1} \tag{4}$$

1. The choice of 300 bpm as the upper limit of tempo is arbitrary.

where $\alpha \in [0,1]$ is the smoothing factor. To incorporate the time at which a user listens to a song,

$$\alpha = 1 - e^{-\Delta t/\tau} \tag{5}$$

where τ is time-constant that we can use to vary the decay, and Δt is the elapsed duration between $\mathbf{t}_{t-1}$ and $\mathbf{s}_t$. Since Equation (4) is iterative, we can update the representation of in an online fashion and only maintain the single vector.

A user can give each song recommendation a *rating* $r \in \{1,2,3\}$, where 1 indicates the user listened to the song and rated it negatively; 2 indicates the user either listened to the song and did not rate it or has not listened to the song; and 3 indicates the user listened to the song and rated it positively. Additionally, we associate with each rating a *timestamp* $t \in \mathbb{N}_+$ to account for its temporal relevancy, assuming that older ratings are less indicative of current user preference. We denote the timestamped rating history of a user as $\mathcal{R}$.

While our recommendation system draws upon music taste of other nearby users, we recognize that a user may want to control the degree to which we recommend based on the taste of other users. Perhaps the user *only* wants recommendations based on their taste at one moment, but suddenly decides to explore the local music preferences of those around them. Standard recommendation systems typically do not provide the user the ability to actively engage with the recommendation process. However, we believe that offering such control can improve user experience. We achieve this functionality in our recommendation system with *recommendation bias* $\beta \in [0,1]$. When $\beta = 1$, we only recommend based on the taste and ratings of the user. Conversely, $\beta = 0$ causes recommendations to only consider the taste and ratings of nearby users. If $\beta \in (0,1)$, we linearly interpolate the preferences of the user and a nearby user.

Lastly, central to our approach is the *geolocation* g of a user, which may take the form of a latitude-longitude pair or a geohash (Morton, 1966). To summarize, we represent a user u as a 4-tuple,

$$u = (\mathbf{t}, \mathcal{R}, \beta, g) \tag{6}$$

With these concepts in mind, we can now discuss the details of our recommendation system.

3.3 Sampling

We conceptualize the recommendation process as sampling, where our objective is to efficiently sample from the joint distribution

$$p(\mathbf{s}, u, v, C, \beta) = p(\mathbf{s}|u, v, C, \beta)p(C|u, v, \beta)p(v|u)p(u) \tag{7}$$

where C is a cluster of songs and v is a nearby user or "neighbor." We discuss our approach to clustering songs in Section 3.4. We depict Equation (7) as a directed graphical model in Figure 1. To generate recommendations, we apply *ancestral sampling* (Bishop, 2006). That is, we first sample a neighbor $v \in \mathcal{N}$, then a cluster C, and finally a song $\mathbf{s} \in C$. In the degenerate case that a user u does not have any neighbors (i.e., $|\mathcal{N}| = 0$), then we simple let the user be their own neighbor, $u = v$.

Algorithm 1 defines our recommendation process. Because we represent user taste and songs as D-dimensional real-valued vectors, care should be taken when selecting an appropriate distance metric, particular as D becomes large (Houle et al., 2010). Notice that

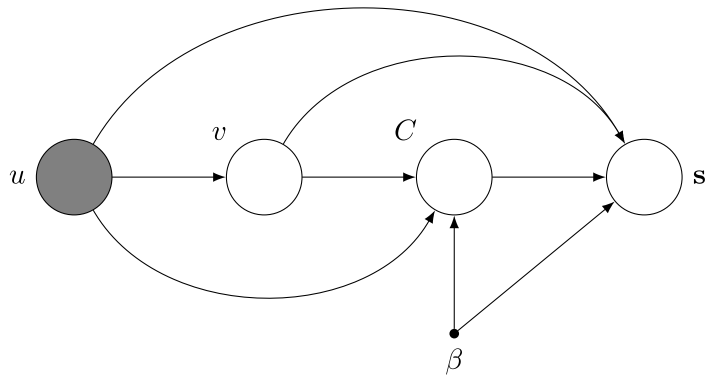


Figure 1: Directed graphical model of the connect.fm recommendation system. We denote a latent (observed) random variable with a white (ref. gray) circle and a deterministic parameter with a point.

Equation (9) allows us to transform any symmetric, non-negative metric or distance function $d : X \times X \to \mathbb{R}_+$ into a normalized similarity metric $s : X \times X \to [0, 1]$ (Schubert, 2017). This is desirable, because we wish to associate *higher* probability with *similar* users, songs, and clusters. In normalizing, we sample based on *relative* similarity to a user.

When sampling a cluster, and subsequently a song, we utilize an *adjusted rating* $\tilde{r} \in (0, 3]$ to integrate the concepts explained in Section 3.2. An adjusted rating is defined by Equation (12) and comprises of *similarity scaling* and *rating scaling*. We define the former by Equation (13), which accounts for the similarity between a song and the taste of the user. In other words, it indicates the overall "compatibility" of a song as a recommendation. Equation (14) defines the latter, integrating the ratings given by the user and their neighbor to a specific song. As $|\mathcal{S}|$ grows, it is likely that most songs will either not be recommended to the user or not receive an explicit rating. This is often the case in collaborative filtering techniques. Thus, this term will often evaluate to 2. For each rating, we apply *time scaling* using Equation (15). The assumption with such a weighting is that recommendation relevance is inversely proportional to the duration between the present and the time at which the user rated the song. The timestamp associated with the rating is the most recent occurrence of recommending the song. This is still the case if a user does not rate a recommendation since we assign all songs not given an explicit rating by the user a neutral rating of 2. Notice that since an adjusted rating cannot be 0, we avoid zero-probability clusters, which alleviates the need to apply smoothing over the sampling distribution. Altogether, an adjusted rating allows us to encapsulate several factors that influence how probable a song is to be recommended.

3.4 Clustering

The immense selection of 60 million songs in the Spotify library poses a substantial computational challenge for identifying suitable song recommendations. A naive approach for

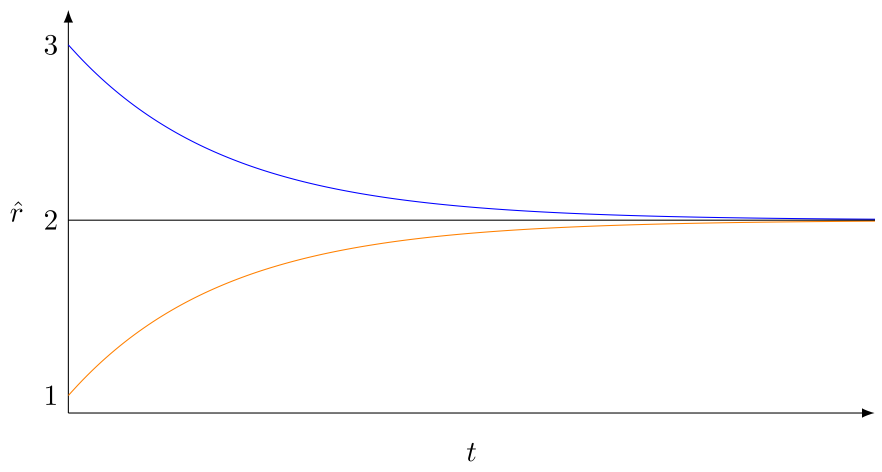


Figure 2: Time scaling. We apply assume exponential convergence toward a neutral rating if a user either positively or negatively rates a recommendation.

identifying the best-suited song recommendation would require an exhaustive search of the entire library for the best song, each time we wanted to recommend a song. This is obviously infeasible. To reduce complexity, we organize songs into clusters, where similar songs are grouped into similar clusters. We follow the assumption of content-based recommendation system that similar items to those user has liked are good recommendations. Notice that since we represent users and songs in the same vector space, we can extend this assumption to recommending songs similar to the user. This vastly reduces the running time and storage complexity since we need only sample from the entire cluster. For a user, the running time reduces from $O(|\mathcal{S}|)$ to $O(|\mathcal{C}|)$ where $|\mathcal{C}| << |\mathcal{S}|$.

When examining prospective clustering algorithms to employ for song clustering and user clustering, we considered the following requirements and desirable characteristics.

- *Non-parametric in the number of clusters.* Due to the immense size and high feature dimensionality of our clusters, it is difficult to qualitatively estimate the optimal number of clusters from the data. Consequently, we search for a method which implicitly optimizes the cluster count based on the data's structure. Although this optimization adds a constant cost to running time, its benefit in the quality of our clusters is well-worth the cost.

- *Computationally efficient in high dimensions.* Due to the immense number of entries and non-trivial dimensionality of our feature set, we require a clustering method which remains feasible at high dimensionality. This eliminates a large subset of clustering algorithms, including hierarchical clustering, k-means, and many density-based approaches.

- *Probabilistic.* We desire a probability density function-based approach in order to capture the variance in a user's music taste. A person's music taste varies each day with mood and environment, such that if we were to choose songs that perfectly

Algorithm 1: connect.fm Recommendation System

1. Retrieve nearby users $\mathcal{N}$ within a distance R from user u such that

$$\mathcal{N} = \{v | d(g_u, g_v) \leq R, v \in \mathcal{U} \setminus \{u\}\} \tag{8}$$

where d is a suitable distance metric, and

$$s(x, y) = \frac{1}{1 + d(x, y)} \tag{9}$$

2. Sample a nearby user $v \in \mathcal{N}$ with probability

$$p(v|u) = \frac{s(\mathbf{t}_u, \mathbf{t}_v)}{\sum_{v \in \mathcal{N}} s(\mathbf{t}_u, \mathbf{t}_v)} \tag{10}$$

3. Sample a cluster $C \in \mathcal{C}$ with probability

$$p(C|u, v, \beta) = \frac{\sum_{\mathbf{s} \in C} \tilde{r}(u, v, \mathbf{s}, \beta)}{\sum_{C' \in \mathcal{C}} \sum_{\mathbf{s} \in C'} \tilde{r}(u, v, \mathbf{s}, \beta)} \tag{11}$$

where $\tilde{r}$ is the adjusted rating of song $\mathbf{s}$,

$$\tilde{r}(u, v, \mathbf{s}, \beta) = f(u, v, \mathbf{s}, \beta) h(u, v, \mathbf{s}, \beta) \tag{12}$$

$$f(u, v, \mathbf{s}, \beta) = \beta s(\mathbf{t}_u, \mathbf{s}) + (1 - \beta) s(\mathbf{t}_v, \mathbf{s}) \tag{13}$$

$$h(u, v, \mathbf{s}, \beta) = \beta \hat{r}(u, \mathbf{s}) + (1 - \beta) \hat{r}(v, \mathbf{s}) \tag{14}$$

$$\hat{r}(u, \mathbf{s}) = \operatorname{sign}(r_{u,\mathbf{s}} - 2) \cdot e^{-\Delta t_{u,\mathbf{s}}} + 2 \tag{15}$$

and $\beta \in [0, 1]$ is recommendation bias.

4. Sample a recommended song $\mathbf{s} \in C$ with probability

$$p(\mathbf{s}|u, v, C, \beta) = \frac{\tilde{r}(u, v, \mathbf{s}, \beta)}{\sum_{\mathbf{s} \in C} \tilde{r}(u, v, \mathbf{s}, \beta)} \tag{16}$$

match the user's music taste, then music would become repetitive. Furthermore, variety is a key component of the listening experience. If songs with the exact same cadence, rhythm, and beat were repeatedly recommended in a single listening session, the music would become predictable and boring. Consequently, non-probabilistic clustering methods such as k-means are undesirable, as the songs in a cluster would lack variance.

After a survey of potential clustering algorithms, we settled on the *Dirichlet process mixture model* (DPMM), one of the most popular probabilistic clustering methods for non-parametric clustering. This model fulfills the three requirements of being non-parametric, computationally efficient at high dimensions, and probabilistic. Intuitively, the Dirichlet mixture models function as a distribution of distributions. Under the DPMM, a Dirichlet probability distribution is defined for a set of probability measures, where each probability measure is itself a probability distribution over the sample space. In the context of our song space, each probability measure defines the likelihood that a song belongs to a given cluster, and the dirichlet distribution defines the likelihood of each probability measure to describe each song cluster.

We utilize a DPMM of Gaussian distributions with a symmetric Dirichlet prior. For a mixture of K Gaussian distributions and N samples, the model can be depicted as

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \prod_{i=1}^{N} \text{GMM}(\mathbf{x}_i) \tag{17}$$

where

$$\text{GMM}(\mathbf{x}_i) = \sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_i | \boldsymbol{\mu}_j, \boldsymbol{\sigma}_j) \tag{18}$$

$$\begin{aligned}
\mathbf{x}_i &\sim \mathcal{N}(\boldsymbol{\mu}_{\mathbf{z}_i}, \boldsymbol{\sigma}_{\mathbf{z}_i}) \\
\mathbf{z}_i &\sim \text{Cat}(\boldsymbol{\pi}) \text{ s.t. } \boldsymbol{\pi} = \{\pi_1, \cdots, \pi_K\} \\
\boldsymbol{\pi} &\sim \text{Dir}(\boldsymbol{\alpha}) \text{ s.t. } \boldsymbol{\alpha} = \{\alpha_1, \ldots, \alpha_K\} \\
\boldsymbol{\alpha} &\sim \text{Inv-Gamma}(\mathbf{1}, \mathbf{1}) \\
\boldsymbol{\mu}_j &\sim \mathcal{N}(\mathbf{0}, \mathbf{1}) \\
\boldsymbol{\sigma}_j &\sim \text{Inv-Gamma}(\mathbf{1}, \mathbf{1})
\end{aligned}$$

Just like in a Gaussian mixture model (GMM), we want to assign points to clusters according to the probability that each point belongs in a cluster. Formally, we assign x_i to the jth cluster through z_i, which is the inferred index of a cluster. Although we initially specify a number of clusters, the final number of clusters is inferred through optimization.

We use a multivariate Gaussian distribution as a mixture component and let $K = 60$. We optimize our model with *preconditioned stochastic gradient Langevin dynamics* (pSGLD), which is ideally suited for optimizing a large dataset through mini-batch gradient descent. pSGLD is defined as follows. Let $\boldsymbol{\theta} = \{\boldsymbol{\pi}, \boldsymbol{\alpha}, \boldsymbol{\mu}_j, \boldsymbol{\sigma}_j\}$ denote all model parameters. Then we can update the parameters of the ith iteration with a mini-batch size M,

$$\Delta\boldsymbol{\theta}^{(i)} \sim \frac{\epsilon^{(i)}}{2} \left\{ g(\boldsymbol{\theta}^{(i)}) \mathcal{L}(\mathbf{x}^{(i)}, \boldsymbol{\theta}^{(i)}) + \sum_{\boldsymbol{\theta}} \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)}) \right\} + \sqrt{g(\boldsymbol{\theta}^{(i)})} \mathcal{N}(\mathbf{0}, \epsilon^{(i)} \mathbf{1}) \tag{19}$$

$$\mathcal{L}(\mathbf{x}^{(i)}, \boldsymbol{\theta}^{(i)}) = \nabla_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta}^{(i)}) + \frac{N}{M} \sum_{k=1}^{M} \nabla_{\boldsymbol{\theta}} \log \text{GMM}(\mathbf{x}_k^{(i)}) \tag{20}$$

where $\epsilon^{(i)}$ is the learning rate for the ith iteration, $\log p(\boldsymbol{\theta}^{(i)})$ is the sum of log prior distributions over $\boldsymbol{\theta}$, and $g(\boldsymbol{\theta}^{(i)})$ is used to scale the importance of each iteration.

3.5 Complexity and Scalability

As a real-time music recommendation system, our approach must be consistently efficient with respect to both time and space. Moreover, its performance must be scalable to millions of songs and users. These requirements pose a significant challenge for any recommendation system, but are critical to address for the practical application.

3.5.1 Data Access

A key component of our implementation architecture is the Redis in-memory database (see Section 3.6.2), which provides both versatile data structures and unparalleled performance. Table 1 provides the time complexities of the Redis commands we use. We discuss details regarding implementation using Redis in Section 3.6.2.

Command	Description	Time complexity
`MGET`	Retrieve the values of the associated keys.	$O(N)$
`MSET`	Set the values of the associated keys.	$O(N)$
`DEL`	Remove the key-value pairs.	$O(NM)$
`SADD`	Add elements to the set.	$O(N)$
`SSCAN`	Iterate over all members of the set.	$O(M)$
`SRANDMEMBER`	Retrieve random members from the set.	$O(N)$
`GEOPOS`	Retrieve the location of members.	$O(N)$
`GEOADD`	Set the location of new or existing members.	$O(\log N)$
`GEOSEARCH`	Retrieve the members within a given radius.	$O(N + \log M)$

Table 1: Redis commands. Unless specified otherwise, N is the number of keys specified in the command. For `DEL`, if the associated value of a key is a collection, then it takes $O(M)$ time to delete them. Otherwise, the running time reduces to $O(1)$. For `SSCAN`, M is the set cardinality. For `GEOSEARCH`, "N is the number of elements in the grid-aligned bounding box area around the shape provided as the filter and M is the number of items inside the shape" (Redis Labs). If a limit of $K \in \mathbb{N}_+$ is set on the number of items, then the K nearest of them are returned.

3.5.2 Sampling

The number of neighbors we retrieve in Equation (8) is a function of the radius R, user location g_u, and total number of users $|\mathcal{U}|$. Intuitively, we will retrieve a larger number of neighbors if (1) the user u sets R to a larger value, (2) the user is located in a highly populated region, (3) the number of users increases, or any combination thereof. This affects the running time of Equation (10), which scales linearly in the number of neighbors, $\Theta(|\mathcal{N}|)$. Since we do not know *a priori* how many neighbors we will retrieve, it is prudent to limit the number of neighbors to $N_{\max} << |\mathcal{N}|$. With this modification, the running time becomes $O(N_{\max})$ since it is possible that only $N < N_{\max}$ are within a distance R from the user. In this way, we trade-off flexibility in recommendation for stability and time.

After sampling a neighbor, we then sample a cluster. Because we sum over all songs to compute the normalization term, Equation (11) is the most computationally expensive step in the recommendation process and has a running time of $\Theta(|\mathcal{S}|)$. This complexity becomes prohibitive, even for a modest number of songs. To resolve this problem, we propose sampling at most $K_{\max} << |\mathcal{S}|$ from all clusters such that we sample

$$K_i = \max\left\{|C_i|, \left\lceil \frac{K_{\max}}{|\mathcal{C}|} \right\rceil\right\} \tag{21}$$

from the ith cluster. We can show the validity of this approach by letting $X, \widehat{X}$ be the true and approximate values of the numerator in Equation (11), respectively. Moreover, let $Z, \widehat{Z}$ be the true and approximate normalization terms, respectively. Then, the percent error ε of an approximate cluster score is defined as

$$\varepsilon = \frac{Z}{X}\left|\frac{X}{Z} - \frac{\widehat{X}}{\widehat{Z}}\right| = \frac{Z\left|\widehat{X}Z - X\widehat{Z}\right|}{XZ\widehat{Z}} = \frac{\left|\widehat{X}Z - X\widehat{Z}\right|}{X\widehat{Z}} \tag{22}$$

First, note that we may sample all $|C_i|$ songs and still have non-zero error since $Z, \widehat{Z}$ depend on all clusters. Second, observe that $\widehat{X}, \widehat{Z}$ are relatively fixed for a given $K_{\max}$, depending on the variance of the song feature vectors in the cluster. Thus, as $|\mathcal{S}|$ increases, X, Z will dominate. Let us assume that each cluster is equiprobable, $X = \frac{Z}{|\mathcal{C}|}$. Then

$$\varepsilon = \frac{|\mathcal{C}|}{Z\widehat{Z}}\left|\widehat{X}Z - \frac{Z\widehat{Z}}{|\mathcal{C}|}\right| = \left|\frac{|\mathcal{C}|\widehat{X}}{\widehat{Z}} - 1\right| \tag{23}$$

Thus, unless the true probability deviates significantly from the uniform distribution, $X \propto p(C|u, v, \beta)$ will be reasonably approximated. Practically, the cluster probabilities will not be equiprobable and of equal size, so varying degrees of under- and over-estimation will occur.

Once we sample a cluster, the last step in our recommendation process is to sample a song. We again sample K_i songs in accordance with Equation (21) and use the result from sampling a cluster to validate the approximate probabilities of the sampled songs. It is evident that the running time of this step is $\Theta(K_i)$. While limiting the number of sampled songs to K_i, we repeat this process for each recommendation, and so each recommendation will be based a different K_i-subset of songs. Further, so long as their is sufficient storage to store user data, song features, and cluster assignments, the time complexity of our sampling approach is *fixed*, regardless of $|\mathcal{S}|$ or $|\mathcal{U}|$.

3.5.3 Clustering

A simple, brute-force method for song recommendation would be to exhaustively search the song space $\mathcal{S}$ for songs with characteristics matching the user's taste. This method, though it runs in $O(K)$ time for a single user, is still computationally expensive due to Spotify's enormous music library of over 60 million songs. For the problem of $|\mathcal{U}|$ users requesting song recommendations, the $O(|\mathcal{U}| \cdot K)$ problem quickly scales to become intractable for even a small increase in users. Evidently, the processing of 60 million songs is the bottleneck. One option to hasten the song recommendations for multiple users would be to sort the music library into clusters $\mathcal{C}$, where each cluster C consists of songs with similar characteristics.

Given $\mathcal{C}$, we can simply identify which cluster is most likely to be positively rated given u's music tastes, resulting in us recommending some $\mathbf{s} \in C$. Consequently, we would only need to cluster the music library once to produce $\mathcal{C}$. Using $\mathcal{C}$, we identify the most fitting clusters $\{C_1, ...C_i\} \subseteq \mathcal{C}$ for each u. After clustering, this would run in $O(|\mathcal{C}| \cdot |\mathcal{U}|)$ time, $|\mathcal{C}| << K$. If we let $f(\mathcal{S})$ be the time required to complete clustering, then the final running time would be $O(f(\mathcal{S}) + |\mathcal{C}| \cdot K)$. This is an improvement over the original $O(|\mathcal{U}| \cdot K)$ provided that $f(\mathcal{S}) < |\mathcal{U}| \cdot K$. The running time for Dirichlet mixture modeling is based on the number of max iterations parameter, so we can simply set $f(K)$ to be less than $|\mathcal{U}| \cdot K$ to achieve an improvement in running time.

3.6 Implementation

As part of the connect.fm mobile application, the Python source code for this project can be found in the `backend` directory of (Tatton et al., 2021).

3.6.1 Architecture

We describe the data flow of connect.fm in Figure 3. While we use Amazon Web Services (AWS) for the implementation, we keep the architecture generic here.

3.6.2 Data Storage

The connect.fm application sends its data directly to AWS' DynamoDB. DynamoDB is a key-value database that is hosted in multiple regions to provide low-latency data access (Services). Low-latency data access is especially important in publishing and accessing data through our app since the speeds are less dependent on user location, allowing users to be served data as quickly as possible. We use DynamoDB to host User and Song data. An AWS Lambda function interprets the update stream whenever data is added, removed, or altered in DynamoDB and manipulates the data to be accessible on our Redis platform that is hosted on AWS Elasticache.

From the official documentation, "Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache, and message broker. Redis provides data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes, and streams" (Redis Labs). Redis' in-memory component allows for sub-millisecond read and write performance. While DynamoDB is sufficient for serving data to users, Redis allows us to make a large amount of queries to the items that we need to perform clustering and sampling, as decribed in Sections 3.5.2 and 3.5.3. We utilize the geospatial indexing Redis data structure to trivially implement the geolocation-based queries for connect.fm. Descriptions and time complexities of the commands we use are provided in Table 1.

The general key schema we follow when caching data is `type:id` where `type` allows us iterate over the same attribute across users. The name of an object is defined by the `id` component. If the identifier requires multiple components, we separate each with a period. For example, we access the taste of user 123 with the key `taste:123`. The schema is completely "flat" which allows us to scale with the number of songs and users, as discussed in 3.5.2. We store clusters as sets, enables us to retrieve random members and avoid elements.

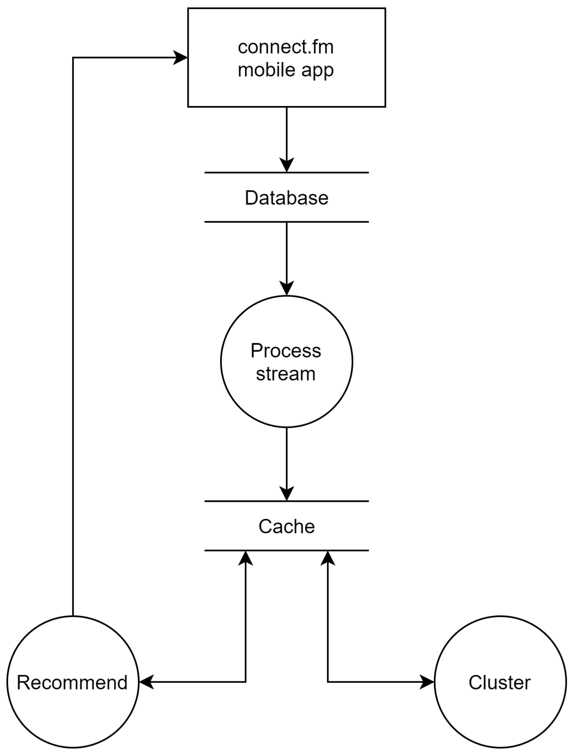


Figure 3: connect.fm dataflow. A rectangle, two parallel lines, and circle represent input and output, a data store, and a function, respectively. User data flows from the connect.fm mobile application to the database. The changes to the database form a stream, which we process and store in the cache for efficient access. We periodically cluster over the songs present in the cache. Lastly, the connect.fm mobile application calls the recommendation function, which accesses all cluster and user data in the cache to compute music recommendations.

To further improve access performance, we utilize the Redis pipeline functionality, which allows for an arbitrary sequence of commands to be sent once over the network, avoiding the overhead of executing each of them individually.

3.6.3 Sampling

The recommendation system is organized into two primary modules: `model.py` and `recommend.py`. The former is intended to for data modeling (i.e., of a user) and abstracting access to the data store. We encapsulate custom queries to Redis with the data access object `RecommendDB`. This has the signficant benefit of allowing implementation modifications without affecting the interface it maintains with other aspects of the system. The latter module contains the `Recommender` class, which is called with `Recommender.recommend()`, which expects as input the user identifier and returns the identifier of a song. For efficiency, we extensively use the `numpy` data structure and its numerous methods to improve performance and simplify implementation. We also use `numpy` to sample songs and clusters.

3.6.4 Clustering

Dirichlet Mixture Model clustering was implemented with Tensorflow version 1.13.1 and Tensorflow-probability version 0.6. Stochastic Gradient Langevin Dynamics was implemented with Tensorflow-probability's optimizer library.

4. Results

4.1 Take-Home Lessons

We learned about quite a few components in recommender approaches. Many of the papers we read in preparation illustrated very different approaches. While some utilize machine learning, a few approaches utilized probability. We modeled our idea on the probability approaches but were also interested in the learning approaches. In implementing our own approach, we found that there are various softwares available that can provide a strong platform for building large-scale data applications. More of what we took away from this project is discussed in the following paragraphs.

In addition to our clustering, we used t-distributed Stochastic Neighbor Embedding (t-SNE) from the `sklearn` Python package to visualize our clusters. Figure 4 illustrates t-SNE plots for our initial clustering of 10,000 songs. The points themselves are the songs with reduced dimensionality via the t-SNE algorithm. Different colors represent different clusters that each song belonged to. There were 43 total clusters, so points that share a color but are not near each other are in different clusters.

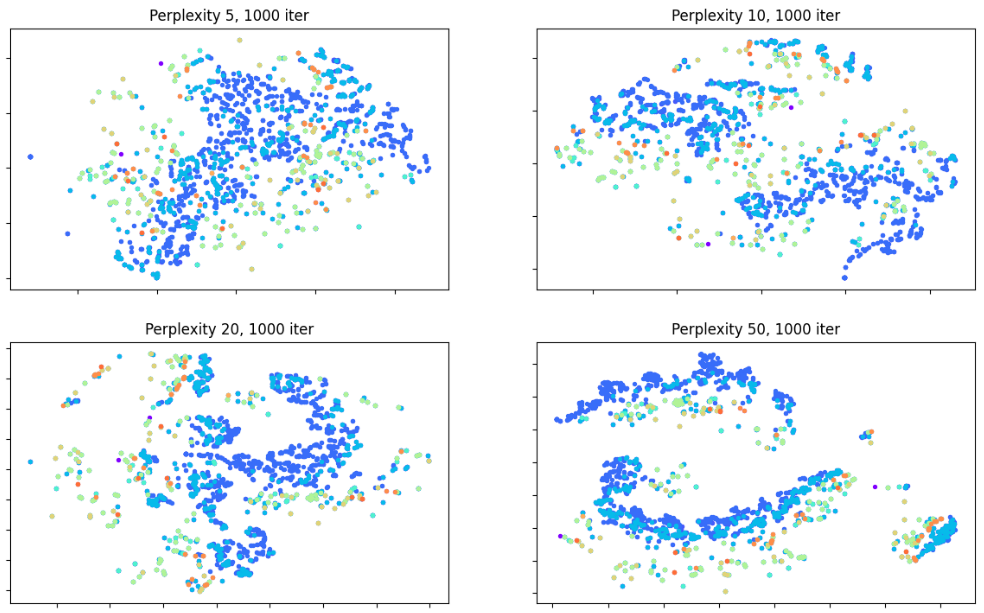


Figure 4: t-SNE plots on our clustering of 10,000 songs

4.2 Success of Our Approach

One the methods we employed to evaluate if our approach would work was querying our recommendation system for a single user with a varying bias. We generated 10,000 users with randomly-assigned attributes (taste, bias, location, radius) sampled from a uniform distribution. Of these 10,000 users, 1,000 were chosen. For each user, we tested 10 β's (each multiple of 0.10 from 0 to 1) to observe the changes in sampling. To evaluate the relevance of the songs, we define the change in taste to be $\Delta\mathbf{t}$:

$$\Delta\mathbf{t} = \sum_{\mathbf{s}\in\mathcal{S}} \sum_{f\in\mathbf{s}} \frac{|\mathbf{s}_f - u_f|}{|\mathcal{S}|} \tag{24}$$

$\Delta\mathbf{t}$ measures the average difference between a recommended song's features and the corresponding feature in the user's taste profile. The hope is that $\Delta\mathbf{t}$ decreases with an increase in β since We tried the same test with 2 and 5 song samples, but neither procedure was able to refute the null hypothesis with a 95% confidence interval.

4.3 Improvements

It would be ideal to see that our method can more consistently recommend songs that tend to decrease in $\Delta\mathbf{t}$ when β is increased.We observed a mean of $\Delta\mathbf{t} = 6.3 \pm 1.7$ when sampling from a range of β's per user. For reference, at $\beta = 0$, we observed a mean of 6.3 ± 1.7. The average time to recommend of 0.99 ± 0.07s. Further improvements would likely require larger changes as described in Section 5.

4.4 Learnings

Prior to implementation of this project, we set out to learn all that we could regarding traditional recommender approaches. As a whole, we read over 40 journal articles including a few surveys to build our understanding of recommendation systems. Using this foundation, we constructed a few ideas as to how we might build our own system. After narrowing down our ideas, we embarked on implementing a novel idea to tie into the connect.fm system. This transformation involved developing the equations to back our approach and continued research on what structure would best support our idea. We focused on scalability and incorporated geographic location with respect to other users, resulting in our choice of backend technologies. AWS services, in general, are built for scalability. Redis, as described in 3.6.2, provides a fantastic framework for accessing our data quickly, and has a built-in service for managing geolocation. Tensorflow was key in building our clustering algorithm.

4.5 Further Work

There were numerous adaptable decisions made for sampling, learning, and clustering. These choices and alternatives are further described in Section 5.

5. Future Directions

There are several possibilities for extending our work. First and foremost is integrating a learning component into our recommendation system. While we encapsulate many contexts into the computed song and cluster probabilities, the system does not have a mechanism to improve its recommendation performance. Going by generic distance functions, we could learn a semantically meaningful metric over the space of users and songs. On caching time and space complexity, experimenting with alternative Redis schemas could help reduce the cost associated with persisting the cache on AWS.

We are lastly interested in employing different clustering methods. One promising alternate clustering algorithm which suits our data is ρ-approximate Density-Based Spatial Clustering of Applications with Noise (ρ DBSCAN), which applies density based clustering, requires only one hyperparameter with clear heuristics for deciding an optimal value, and has an average running time of $O(n \log n)$. DBSCAN satisfies our requirements, and also contains an added benefit of identify clusters of arbitrary shape. In contrast, traditional clustering algorithms typically search for spherical clusters. Furthermore, DBSCAN is built to consider the notion of noise, which aligns well with the inconsistent nature of music taste. Two songs with identical song characteristics may evoke completely different reactions from users due to factors such as variance in the mood and environment of the user.

6. Conclusion

Our recommendation system focuses on a novel approach to music recommendation. Throughout our implementation, we focus on keeping recommendation relatively fast. Our recommendation system has not been shown to produce new recommendations with respect to β, though our output data has shown that reducing the bias contributed by neighboring users creates slightly more relevant song recommendations. We have, however, shown that we can quickly generate new song recommendations that take into account the taste profile of a given user, and we can easily group users by geographic location. In general, most of the recommended songs were within a reasonable range of the user's taste profile, but we will continue to look a more consistent contribution from the β term. We also hope to integrate a learning element into our model so that our recommendation system improves as we record more user interactions. We also hope to integrate DBSCAN to better cluster our data.

7. Individual Contributions

For Daniel and Ryan, this project also relates to their senior capstone on implementing the mobile application, connect.fm. While Alex is not officially involved in the capstone project, he has helped implement some of the AWS architecture of the recommendation system backend. Specifically for this project, the following is a description of individual contributions from each member. All members contributed equally toward reviewing the research literature.

7.1 Alexander Sfakianos

- Implemented the evaluation code of our recommendation system under the `evaluation` package on the connect.fm GitHub repository.

- Implemented the processing of incoming data from connect.fm users under the `dbprocessing` package on the connect.fm GitHub repository.

7.2 Daniel Shao

- Implemented the clustering model under the `clustering` package on the connect.fm GitHub repository.

7.3 Ryan Tatton

- Unofficial team lead for the project, helping ensure its succesful completion.
- Formalized the recommendation system algorithm and data representation.
- Implemented the recommendation system provided under the `recommendation` package on the connect.fm GitHub repository.

References

C. M. Bishop. *Pattern Recoginiton and Machine Learning.* Springer International Publishing, 1 edition, 2006. ISBN 978-0-387-31073-2.

Allison J.B. Chaney, David M. Blei, and Tina Eliassi-Rad. A probabilistic model for using social networks in personalized item recommendation. In *Proceedings of the 9th ACM Conference on Recommender Systems*, RecSys '15, page 43–50, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336925. doi: 10.1145/2792838.2800193. URL `https://doi.org/10.1145/2792838.2800193`.

M. Houle, H. Kriegel, P. Kröger, E. Schubert, and A Zimek. Can shared-neighbor distances defeat the curse of dimensionality? In *Scientific and Statistical Database Management, SSDBM 2010*, volume 6187, pages 482–500, 2010. ISBN 3642138179. doi: 10.1007/978-3-642-13818-8_34.

Saurabh Kulkarni and Sunil F. Rodd. Context Aware Recommendation Systems: A review of the state of the art techniques. *Computer Science Review*, 37:100255, August 2020. ISSN 1574-0137. doi: 10.1016/j.cosrev.2020.100255. URL `https://www.sciencedirect.com/science/article/pii/S1574013719301406`.

G. Linden, B. Smith, and J. York. Amazon.com recommendations: item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1):76–80, 2003. doi: 10.1109/MIC.2003.1167344.

G.M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Ottawa, Ontario, Canada, March 1966. IBM Ltd. URL `https://dominoweb.draco.res.ibm.com/reports/Morton1966.pdf`.

Redis Labs. Redis. URL `https://redis.io/`.

Simon Renaud-Deputter, Tengke Xiong, and Shengrui Wang. Combining collaborative filtering and clustering for implicit recommender system. In *2013 IEEE 27th International*

Conference on Advanced Information Networking and Applications (AINA), pages 748–755, 2013. doi: 10.1109/AINA.2013.65.

E. Schubert. Knowledge discovery in databases [PDF document], 2017. URL `https://dbs.ifi.uni-heidelberg.de/files/Team/eschubert/lectures/KDDClusterAnalysis17-screen.pdf`.

Amazon Web Services. Amazon DynamoDB. URL `https://aws.amazon.com/dynamodb/`.

Spotify. Spotify web api reference, 2021. URL `https://developer.spotify.com/documentation/web-api/reference/#category-tracks`.

R. Tatton, J. Colvin-Hatchett, D. Shao, A. Sfakianos, E. Voss, and D. Ye. connectfm, 2021. URL `https://github.com/connectfm/cfm/tree/main/backend`.

R. Xu and D. Wunsch. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678. doi: 10.1109/TNN.2005.845141.