Technical Report CS2019-12

Scottipy: An In-Depth Playlist Creator/Editor Using the Spotify API

John Scott

Submitted to the Faculty of
The Department of Computer Science

Project Director: Dr. Aravind Mohan
Second Reader: Dr. Gregory Kapfhammer

Allegheny College
2019

I hereby recognize and pledge to fulfill my responsibilities as defined in the Honor Code, and to maintain the integrity of both myself and the college community as a whole.

John Scott

JOHN SCOTT. Scottipy: An In-Depth Playlist Creator/Editor Using the Spotify API.
(Under the direction of Dr. Aravind Mohan.)

ABSTRACT

Spotify collects hundreds of data points for every song in their massive catalog of music. Information on features of the song, as well as meta-data about the artist, can paint a near complete picture that can be analyzed and compared to other songs. Despite this breadth of data, music discovery on Spotify is lacking in many ways. Other than a single automatic and static weekly playlist, there is no easy way to find new music related to a user's taste and preferences. This thesis paper explores the creation of a web-based playlist creator and editor, dubbed "Scottipy", using Spotipy, a Python wrapper for the official Spotify API. This project seeks to extend the current boundaries of music personalization.

Acknowledgements

If I had to complete this on my own, I wouldn't have been able to. I first have to thank my parents, as well of the rest of my family, for supporting me even when I doubted myself. Thanks to Dr. Mohan and Dr. Kapfhammer, and all of my professors and peers in the Allegheny Computer Science Department. Shouts out to my friends and roommates, you kept me going. Special thanks to Jon Schaeffer and Zac Shaffer, for the 11th hour bug squashing.

Contents

List of Figures

Chapter 1

Introduction

The Spotify API allows public access to an ever-growing list of end points for every song in their 40 million and growing music catalog. Despite this vast wealth of information, Spotify users are not allowed to select tracks based on this data. This chapter lists the driving motivations for this project, analyzes other ventures into Spotify API data, and outlines the goals and vision for this application.

1.1 Motivation

The personal motivations for this project stemmed from a series of criteria that I wanted my thesis to fulfil, before choosing a specific project to pursue. The requirements included:

- Inclusion of music - As an active musician at Allegheny College, as well as a music theory minor, it almost went without saying that I wanted to include my biggest passion in a senior research project.
- Practicality - I sought to create a project that, at the very least, could likely have continued real-world utilization by myself (and hopefully others) after the conclusion of my academic career.
- Opportunity to learn - Being not well-versed in working with existing APIs, as well as the area of both front-end and back-end web development, I saw this assignment as an opportunity to explore areas I was not able to through college courses

Apart from these self-defined criteria, after settling on the specific topic of playlist creation, the Spotify API itself served as motivation. My simultaneous amazement of the power that Spotify holds, as well as frustration for the lack of use and support of the breadth of data generated, was enough to begin creating a tool to remedy this shortfall in music discovery.

Spotify currently controls 36% of the global music streaming market [2]. With a catalog of well over 40 million songs, there is an abundance of music to fit any listeners' taste. Despite this, Spotify has no easy way to use the information Spotify collects to

sort and choose songs, other than a selection of premade playlists for diferent genres and keywords. Artist pages include tabs categorizing related artists, but there is no way to quickly generate a list of songs that takes advantage of the track information Spotify possesses. The Spotify API can allow for this and much more, but curated music discovery is limited to one weekly playlist, called "Discover Weekly", consisting of 30 songs tailor made for each Spotify user. As accurate as this playlist is, it merely scratches the surface of what the data that Spotify possesses can achieve. With so much more data on every song in Spotify's library already available, the motivation for this project is simple: allowing Spotify customers to find new music at any time, with any selected specifications, puts the power of discovery in their hands. Finding new songs is always enjoyable, and this project can create that feeling at a user's disposal.

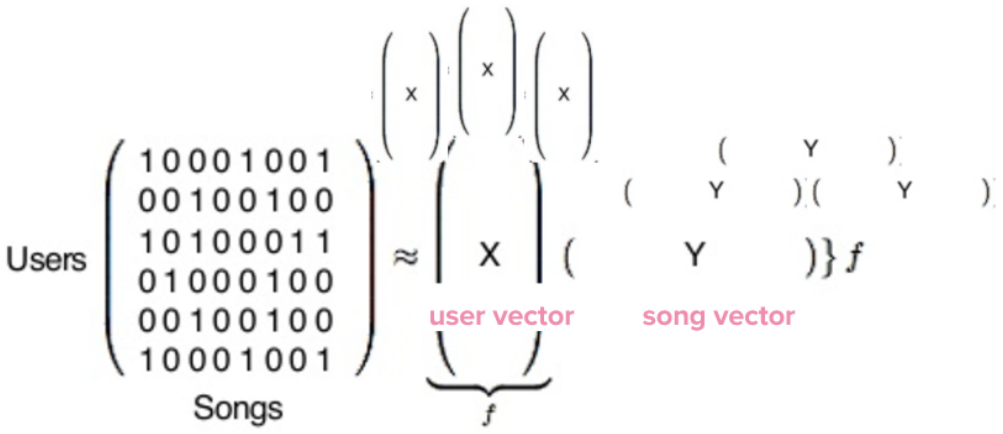
1.2 Current State of the Art

In 2014, Spotify acquired the music intelligence company "The Echo Nest" [18]. The company collects hundreds of data points on songs, such as values for the key, tempo, song duration, time signature, and other features. They have also created formulas to calculate more abstract features, including energy and "danceability". These values are composites of simpler points; energy is based on overall loudness of a song, as well as segment duration, which refers to loops and repeated sections of songs. The danceability formula has not been released, but is a combination of beat strength, tempo and tempo stability, and other undisclosed factors. Spotify gaining control of the company gave them access to this database, and a year later the Discover Weekly playlist was released [5].

The Discover Weekly playlist is so successful because of the way it is tailored to Spotify customers. The first deciding portion is pulled from the user's "taste profile", aggregated from songs a user listens to, or saves to their library. The profile is grouped into clusters, such as this example for reporter Nikhil Sonnad [19]:

Fine-grained genres, such as "chamber pop", or "stomp & holler", are taken from methods from The Echo Nest, and are used as a first level of personalization. The cluster-based approach serves to eliminate many songs that users will not be interested in, but also allow multiple wide-spanning genres that may have no relation with each other.

The other prong of the Discover Weekly method is collaborative filtering [4]. For Spotify, this technique works by comparing users who have played many of the same tracks in the same genre. Spotify creates a giant matrix to compare the songs, with each row corresponding to one of the 140 million users, and a column for each song in their catalog [12].



The user vectors, represented by columns in the matrix, are compared with each other to find their closest matches, determined by similar patterns of interactions with the same tracks. Songs which appear in user's A's vector, but not in user B's, are recommended to user B. The end result is a playlist of music which feels handpicked by a close friend, but is actually the result of a cleverly crafted algorithm. Currently, the Discover Weekly playlist is the gold standard for personalized music discovery. Millions of users receive a list of songs they are likely to enjoy, every week without fail.

1.3 Goals of the Project

As this project is an academic thesis, and not just a simple weekend hobby, my overarching objective is to demonstrate what I have been able to learn at Allegheny College, as a capstone to end my academics. More specifically to this tool, I wish to prove my ability to meld my passion of music, with my major of computer science. In this process, I wish to research utilization of established, robust APIs, to create an application that shows principles which can be brought with me into the professional industry. Upon beginning this thesis project, I have just elementary experience on working with APIs, and even less knowledge of web development. With these short-

comings in mind, my primary goal in completing this thesis is to develop a working ability in these areas as such:

- Spotify API - be able to call information from Spotify's database, in a way which allows music listeners greater control over tracks they listen to.
- Web Development - Explore front-end and back-end web development strategies, as well as the use of framework and bootstrap tools, to package my tool in a more logical and accessible way for public use.

Concerning goals within the scope of the tool, the objective of this project is to create a web-based application which allows music fans of all levels to create fine-tuned playlists, based on any amount of available criteria. As previously discussed, the Discover Weekly playlist is the premier tool for finding song tracks unrecognized to the user. However, the tool has its own set of drawbacks. Since both methods use Spotify customer behavior to create parameters, non-subscribers will not be able to use the feature at all, and less active or new subscribers will not have a baseline for the technique to begin from. In addition, the playlist only refreshes once a week, giving thirty new tracks. There is no way to manually reset your playlist, or get more new songs. The project I outline would be web-based, using the data that Spotify already possesses. Using the Spotify API, I will create a custom playlist which could be played in the page, or saved to a user's profile. The input for the tool would be any song in the Spotify catalog, or alternatively genres or certain keywords. Keywords would be words that appear in popular playlist titles, and tracks would be selected from them. In addition to this user input, there will also be options to adjust ranges of features that are inherent to every song, such as overall song popularity, tempo, energy, amount of vocals present, and other characteristics. These metrics allow fans to fine tune the resulting playlist, ensuring they will be able to find enjoyable tracks, making finding new songs an easy task for any music fan.

Considering this project heavily involves something I am passionate about, I foresee myself adding features, and increasing the robustness of this application, for years to come down the road. The Spotify API is constantly adding features and increasing available information, so the final state of this project is apt to change after the conclusion of the academic year. Because of this possibility, it is important that I do not lose track of the scope and time available. Especially for areas I have less experience with, success may be defined as having a working "proof of concept", and a fully functional web-app may have to be completed at a later time. My final goal is to enjoy the process of creating a working product, which should be aided by the subject matter.

1.4 Thesis Statement

This thesis project is an application stemming heavily from research into the Spotify API, as well as various strategies utilized for hosting the playlist creator and editor

methods in a web-app. The resulting product will be designed to allow any user to edit and create Spotify playlists, taking advantage of the end points of tracks in the Spotify catalog. The target audience is music fans of any level of theory knowledge, so making the application both usable and accessible for anyone is a major design concern. Research into the Spotify API will directly impact the design and output of the project.

1.5 Thesis Outline

Chapter 2 outlines the idea of Music Information Retrieval, as well as how the field has been used in existing Spotify projects. Chapter 3 gives a framework of how the Spotify API is structured, as well as outlines of Spotipy and Flask. Chapter 4 outlines the initial design of the project, including specific code examples to better explain the product. Chapter 5 lists future improvements that are desired, as well as new conceptual features.

Chapter 2

Related Work

This chapter gives a description of what music information retrieval (MIR) is, as well as how the data has been used by Spotify services, as well as third parties using the Spotify API. Other applications using the Spotify API are also outlined, and compared to Scottipy.

2.1 Music Information Retrieval

One of the first overviews on MIR [6] uses seven pieces of information to analyze each song. These areas are pitch, tempo, harmony, timbre, as well as editorial information, lyrics, and bibliographic information. The pitch of a song includes notes, intervals between notes, as well as key. The temporal facet included the speed of a song, time signature, and pitch and harmonic duration. Harmony refers to two or more pitches at the same time creating a polyphony of notes. This area also includes chords and chord progressions. The timbral area includes tone color, such as the difference in sound quality between a flute and a violin. Orchestration, or instrument selection, is the most important part of timbre. The editorial section is mostly performance instructions on music, such as notes for dynamics and articulations. The textual facet is the lyrics of a piece. The lyrics viewed alone are usually a good measure of the mood of a song, though they may not be enough to find a desired melody. The paper mentions an example of this phenomenon between "God Save the Queen" and "My Country 'tis of Thee", as two songs with the same melody but different lyrics. The bibliographical facet has little to do with the actual form of the song, but more meta information about who made the song. This can include:

- song title
- composer
- arranger
- editor
- performer

and other descriptions about a song, rather than traits garnered from the song. Although these seven facets include all of the information needed about a song, for comparison purposes a more nuanced look at individual areas within these groupings would be needed. The paper also includes looks into systems that use MIR, which can be grouped into either analytic/production, or locating systems. Analytic and production services focus mostly on the complete digital representation of music, with a deemphasis on bibliographical information. Projects like this are used to create comprehensive theoretic analyses of songs. Locating systems are used to quickly retrieve existing works, such as finding songs based on user criteria and where they can be played.

2.2 MIR in Spotify

The Spotify API offers many different features for songs, which include "danceability", energy, key, loudness, mode, "speechiness", "acousticness", "instrumentalness", duration, time signature, "liveness", valence, and tempo [20]. In addition to the data points mentioned earlier, Spotify uses many others from The Echo Nest to paint a portrait of a musical composition. Danceability, valence, energy, and tempo are all different ways of measuring the mood of a track, typically songs with a happier mood will have higher values. Loudness, speechiness, and instrumentalness are all properties of songs, showing how much of a given element is present. Liveness and acousticness show the context of a song, so if it is a live recording or an acoustic version.

2.3 Related Works

The Spotify Developer Showcase [8] is a collection of submitted projects that take advantage of various facets of the Spotify API. Since the API collects information about user listening habits, as well as song data, many projects focus on using existing user data. One such project is Klarafy [3], which takes information from a user's playlists, to find classical music that they are more likely to enjoy. Klarafy's documentation states,

> Klarafy starts from the relatively plausible premise that someone who loves loud, fierce metal, is more likely to enjoy a loud, fierce piece by Wagner than soft, delicate piano music. Klarafy seeks out the affinities, similarities or connections between your favourite music and classical music. To arrive at this 'translation', Klarafy scans your playlist for three criteria: - your favourite music genre (most common genre); - the musical mood that prevails in your playlist; - points of departure determined in advance, such as specific instruments, voice types, etc.

Klarafy aggregates data from user playlists, and links that data to a predetermined set of composers and their works. There is no way to get a new artist if your

taste doesn't fit the composer, however, and getting a new emotion would practically involve resetting your Spotify profile.

musicScape [1] is another example relying on user data, which takes recently played tracks for a user, to generate a minimalist landscape from audio features of the songs. The color and landscape will change, based on track features from recently played music.

scottjohn0 's musicScape

Playlist Souffle [9] is another playlist editor, but it works by swapping out songs in a playlist, and replacing them with tracks either from the same album, or by the same artist. This tool can be useful to find lesser-known songs, while still having a connection to songs the user enjoys.

The above image is a "souffle" seeded from Spotify's "Today's Top Hits" playlist [23]. The songs have all been changed out, but there is no other manipulation that

can be done. Because of this, the tool is limited as a "one-and-done", that can only be so effective at finding new music.

Since the spirit of this project is to allow the general public to use the strength of Spotify's library, and not just established users, projects that do not rely on user profile data can be more comparable to Scottipy. One of these is MagicPlaylist [14], which is also a web-based playlist creator. The site is much more limited in its usage of Spotify song data, however; a look through the GitHub source page [15] shows that resulting playlists are based on one aspect of Spotify artist pages, which is the related artist feature. Because of this, creating multiple playlists using songs from the same artist as a seed will give near identical results. In addition to this drawback, there is no way to edit the resulting list of songs. In tandem, these design decisions create a rather static experience, which leaves a lot to be desired in terms of creating an enjoyable collection of music.

Although there are many services using various parts of Spotify's API, no publicly showcased projects can create playlists based on user chosen criteria, and make a dynamic result. In the next chapters, I outline how Scottipy seeks to create a program that meets these challenges.

Chapter 3

Method of Approach

This chapter outlines how the process of building the project will work. It also shows how the Spotify API works, and includes an outline of Spotipy and Flask.

3.1 User Inputs

The existing Spotify API allows for searching for tracks, albums, artists, and playlists, which makes it easy for the user to enter a song name and quickly find it. For example, here is a query for the term "dark fantasy":

```
curl -X "GET" "https://api.spotify.com/v1/search?q=dark%20fantasy&
   type=track" -H "Accept: application/json"
-H "Content-Type: application/json"
```

The result is the Spotify link to the track, as well as information about the popularity, and listing on its album.

```
"href": "https://api.spotify.com/v1/tracks/7yNK27ZTpHew0c55VvIJgm",
        "id": "7yNK27ZTpHew0c55VvIJgm",
        "is_local": false,
        "name": "Dark Fantasy",
        "popularity": 66,
        "preview_url": null,
        "track_number": 1,
        "type": "track",
        "uri": "spotify:track:7yNK27ZTpHew0c55VvIJgm"
```

From the previous step, information about the song can be found by searching for its audio features:

```
curl -X "GET" "https://api.spotify.com/v1/audio-features/7
   yNK27ZTpHew0c55VvIJgm"
-H "Accept: application/json" -H "Content-Type: application/json"
```

This query results in an easy to use list of characteristics and their matching values:


```
{
  "danceability": 0.59,
  "energy": 0.587,
  "key": 5,
  "loudness": -5.919,
  "mode": 1,
  "speechiness": 0.0457,
  "acousticness": 0.274,
  "instrumentalness": 0,
  "liveness": 0.167,
  "valence": 0.367,
  "tempo": 88.015,
  "type": "audio_features",
  "id": "7yNK27ZTpHew0c55VvIJgm",
  "uri": "spotify:track:7yNK27ZTpHew0c55VvIJgm",
  "track_href": "https://api.spotify.com/v1/tracks/7
     yNK27ZTpHew0c55VvIJgm",
  "analysis_url": "https://api.spotify.com/v1/audio-analysis/7
     yNK27ZTpHew0c55VvIJgm",
  "duration_ms": 280787,
  "time_signature": 4
}
```


Each of the listed features can be compared to the rest of the Spotify library to find similar tracks. In addition to tracks, keywords such as moods or events would also be accepted as inputs. Instead of searching for tracks in that case, the Spotify API allows us to find popular playlists with the keywords in the title. In this example, the term "sleep" is queried, with no other context: //

```
curl -X "GET" "https://api.spotify.com/v1/search?q=sleep&type=
   playlist" -H "Accept:
application/json" -H "Content-Type: application/json"
```

This search finds the playlist called "Sleep", curated by Spotify, as the first result:


```
{
  "playlists": {
    "href": "https://api.spotify.com/v1/search?query=sleep&type=
       playlist&market=US&offset=0&limit=20",
    "items": [
      {
        "collaborative": false,
        "external_urls": {
          "spotify": "https://open.spotify.com/playlist/37
             i9dQZF1DWZd79rJ6a7lp"
        },
        "href": "https://api.spotify.com/v1/playlists/37
           i9dQZF1DWZd79rJ6a7lp",
        "id": "37i9dQZF1DWZd79rJ6a7lp",
```

```
      "images": [
        {
          "height": 300,
          "url": "https://i.scdn.co/image/
              f499b0497289a6432dcccfb6de5f57164739d525",
          "width": 300
        }
      ],
      "name": "Sleep",
      "owner": {
        "display_name": "Spotify",
        "external_urls": {
          "spotify": "https://open.spotify.com/user/spotify"
        },
        "href": "https://api.spotify.com/v1/users/spotify",
        "id": "spotify",
        "type": "user",
        "uri": "spotify:user:spotify"
      },
      "primary_color": null,
      "public": null,
      "snapshot_id": "MTU0MDkxNDI1OCwwMDAwMDBhNDAwMDAw
                  MTY2YzExYTAyNTcwMDAwMDE2MmYyYjBlOGQ4",
      "tracks": {
        "href": "https://api.spotify.com/v1/playlists/37
            i9dQZF1DWZd79rJ6a7lp/tracks",
        "total": 152
      },
      "type": "playlist",
      "uri": "spotify:user:spotify:playlist:37i9dQZF1DWZd79rJ6a7lp
          "
    },
```


Popular tracks from those would be added to the playlist created by this application. Thus, the basis of the model can find related tracks from a user entering either a song name, or even just an emotion that they would like to relate music to.

3.2 Spotipy

Spotipy is a *Python* wrapper for the Spotify API, developed by The Echo Nest team [13]. Spotipy gives full access to the data available from the Spotify platform. Spotify's native resource identifiers, URL links, and IDs for artists, tracks, playlists, and albums are all supported. Spotipy allows users to quickly gather information such as recently played songs, saved tracks, playlists, and albums, top played artists and songs, and even currently playing tracks. There are also methods for featured playlists across Spotify, new releases, and even recommendations from seed artists, genres, or tracks.

The track end points are listed below, with their types, value ranges, and descriptions.

- Acousticness - float - 0.0 to 1.0 - Lack of electric sounds/instruments
- Danceability - float - 0.0 to 1.0 - Combination of tempo, rhythm stability, and beat strength
- Energy - float - 0.0 to 1.0 - Intensity, dynamic range, loudness, timbre, entropy
- Instrumentalness - float - 0.0 to 1.0 - Lack of vocals
- Liveness - float - 0.0 to 1.0 - If a track was performed live
- Loudness - float - -60 to 0 - Overall volume
- Speechiness - float - 0.0 to 1.0 - Presence of spoken words
- Tempo - float - 0 to 320 - Beats per minute of a track
- Valence - float - 0.0 to 1.0 - Musical positiveness of a track

Spotipy was chosen for this project simply for ease of use, and my own familiarity with *Python*, instead of using *JavaScript*. This also allowed coding to be quickly checked in the terminal, which in the testing process was easier for me than using *JavaScript*. From the beginning of the research project to the present, I have not seen any advantage in using the native API instead of Spotipy. From accessing the database, to general ease of use, Spotipy had no visible limitations.

3.3 Flask

The flipside of coding in *Python* rather than *JavaScript*, was the increased difficulty in creating a web app. Luckily, the Flask framework covered every need of this project. Flask's emphasis on being a "micro framework" means that

> By default, Flask does not include a database abstraction layer, form validation or anything else where different libraries already exist that can handle that. Instead, Flask supports extensions to add such functionality to your application as if it was implemented in Flask itself. Numerous extensions provide database integration, form validation, upload handling, various open authentication technologies, and more [24].

Since the project does not currently require any advanced or complicated techniques, a simpler framework like Flask can easily and smoothly handle the requirements. In addition to these benefits, Flask has a strong online presence, and extensive documentation. This became even more useful than I first realized, as it helped me to create the HTML templates with more guidance than many other, perhaps more robust, frameworks. The command line interface also included above average debugging, which became quite useful in the learning process, even when my issues were in the HTML templates, rather than the *Python* source code. As this program grows

in the future, I will surely explore using other frameworks, if Flask becomes limiting in features or speed. Django has become a likely choice, as I had the opportunity to work with it in class this semester.

Chapter 4

Implementation

This chapter outlines the design of the project. The packages and resources outlined in the previous chapter are referenced in how they were used in the project design. Many of the methods from the source code are outlined and explained, in order of how they may be accessed by a user.

4.1 Main Menu

Due to time constraints of the thesis, much of this project still currently exists as *Python* scripts, which execute in the terminal. As such, to start the project, the user must execute in the terminal.

```
$ python3 main.py
```

The *main.py* serves only as a gateway to the two current scripts, *recent.py* and *edit.py*. The available input options are "recent", "edit", and "exit". The entire script is encompassed in a while loop, such that an incorrect input simply prompt the user for another input. The function scripts are called using:

```
exec(open("./file.py").read())
```

4.2 Playlist Editor

edit.py is used to edit a user's existing playlists by filtering them using the Spotipy API data. As outlined in the previous chapter, this project uses the Spotipy wrapper for the official API, written in JavaScript. The constructor function (Figure 4.1) is used for user authentication; the `CLIENT_ID`, `CLIENT_SECRET`, `REDIRECT_URI`, and SCOPE variables are all needed for any projects that use the Spotify API.

```
  def __init__(self):
    self.CLIENT_ID = SPOTIPY_CLIENT_ID
    self.CLIENT_SECRET = SPOTIPY_CLIENT_SECRET
    self.REDIRECT_URI = "http://localhost:5000"
    self.SCOPE = "playlist-read-private playlist-modify-private
    playlist-read-collaborative
    playlist-modify-public"
    self.sp = self.getUser() #Creates Spotify instance
    self.id = self.sp.me()["id"] #Gets ID of authenticating user
```

Figure 4.1: `edit.py` constructor

```
  def getPlaylist(self):
    #This function gets all playlists from the user.
    results = self.sp.current_user_playlists()
    for i, item in enumerate(results["items"]):
      print ("{number} {name}".format(number=i, name=item["name"]))
```

Figure 4.2: `edit.py getPlaylist` function

Once the user is authenticated, the *getPlaylist* function (Figure 4.2) gets the name of every playlist the user has created or followed on Spotify. It also assigns a numeric value to each one.

The user then inputs the corresponding number to the playlist they desire to filter. On the selection, a series of graphs are generated, illustrating the distribution of the editable end points of songs in the playlist. When the user closes these plots, they are prompted to enter the low and high values of the end points of the tracks, in the *getLimits* function. Leaving the input empty will automatically choose the lowest and highest possible values for each set, so if a certain criteria doesn't matter, no songs will be excluded by it.

The *getSongs* function is then triggered, which returns the song IDs for every track in the playlist. Once the track IDs are gathered, the song features are returned using the *getFeatures* function. Thanks to Spotipy, this takes only a single line of code. (Figure 4.3)

```
def getFeatures(self, track):
  #This function retrieves audio features from Spotify
  features = self.sp.audio_features(track)
  return features
```

Figure 4.3: `edit.py getFeatures` function

It is at this point that the API end points are utilized, in the *sortSongs* function. In a series of nested *if* statements, each desired range is checked for each value of the songs. If they all return true, the song "passes", and is added to the generated playlist. The *createPlaylist* function takes the passing songs, and creates a playlist

using two Spotipy methods: *user_playlist_create* and *user_playlist_add_tracks*. At this point, plots with an identical format as generated from the initial playlist are once again created, in order to concretely visualize the new changes. The final line to execute opens the user's playlists in a web page, using the *open_new_tab* method from the *webbrowser* package. This puts the user one click away from the newly generated playlist.

4.3 Web Interface

The *recent.py* file is a venture into creating a web portal, using the Flask framework to facilitate the use of *Python*, as outlined in the previous chapter. Initialized similarly to *edit.py*, the authorization process is nearly identical, but with more robustness. Upon either creation or discovery of the authentication token, the Flask app will first return the last 50 tracks the user has saved to their profile, using the *current_user_saved_tracks* method. Unfortunately, the Spotify API limits rates to 50 per call, but an if statement paired with a dynamic offset in the *getSongList* function allows this number to be increased, albeit with a very miniscule cooldown time.

```
if callNo == 0:
        offset = 0
    else:
        offset = callNo*min(playlistLength, 50)

    results = sp.current_user_saved_tracks(limit=min(playlistLength,
        50), offset=offset)
```

Figure 4.4: Dynamic Offset

The app then opens an HTML template. The user is presented with a form, in which they may choose to create a Spotify playlist, with options for the number of songs, a limit to the number of songs by a single artist, and whether the playlist should be shuffled or not. (Figure 4.5)

These values are passed into a thread in the *run* function, which then loads the *running.html* page.

The *running.html* is mainly a placeholder, but does include a progress bar corresponding to the creation of the playlist (Figure 4.6).

In order to limit the maximum number of songs from a single artist, the code needed the ability to generate r random, sorted integers from [0, n). This method was taken from an example by Raymond Hettinger [11] (Figure 4.7).

After removing the excess songs, the playlist is created and populated, using a similar method to *edit.py*. On generation of the playlist, the user is redirected to the *success.html* template, which has options to view the created playlist on Spotify's web player, and to go back to the input form.

```
<div>
    <form action="/run" method="POST">
    Number of songs: <br>
    <input type="number" name="playlistLength" value="50" min=20>
    <br><br>
    Max songs per artist: <br>
    <input type="number" name="maxSongs" value="3" min=1>
    <br><br>
    <input type="checkbox" name="shuffle" value="True">Shuffle Songs
       ?
    <br><br>
    <input type="submit" value="Create Playlist!" autofocus>
    </form>
</div>
```

Figure 4.5: `runRecentlyadded.html` input form

```
<div>
    <h3>Generating playlist. Please wait...</h3>
    <progress value= {{ progress }} max="100"></progress>
</div>
```

Figure 4.6: `running.html` progress bar

```
def sample(n, r):
    #taken from http://code.activestate.com/recipes/272884-random-
       samples-without-replacement/
    #Generate r randomly chosen, sorted integers from [0,n)
    rand = random.random
    pop = n
    for samp in range(r, 0, -1):
        cumprob = 1.0
        x = rand()
        while x < cumprob:
            cumprob -= cumprob * samp / pop
            pop -= 1
        yield n-pop-1
```

Figure 4.7: Hettinger's random generator

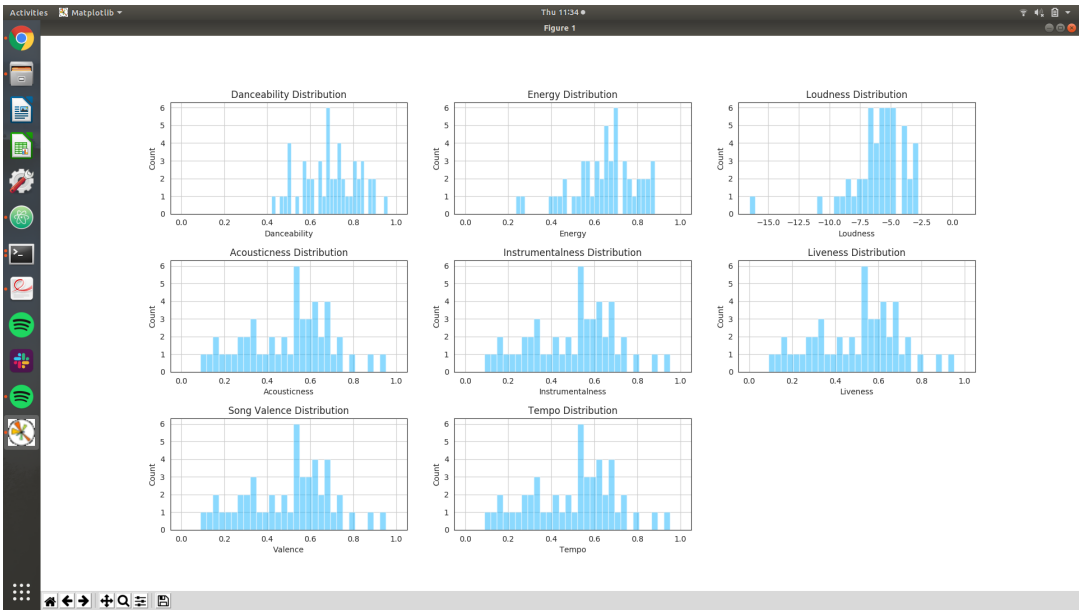
4.4 Result Testing

In order to empirically check the effectiveness of the playlist editing, the Matplotlib library for Python [16] was used to generate graphs showing the values of the audio features, before and after the editing process. In addition to this, the "pandas" library [25] was used to create a dataframe with the track data.

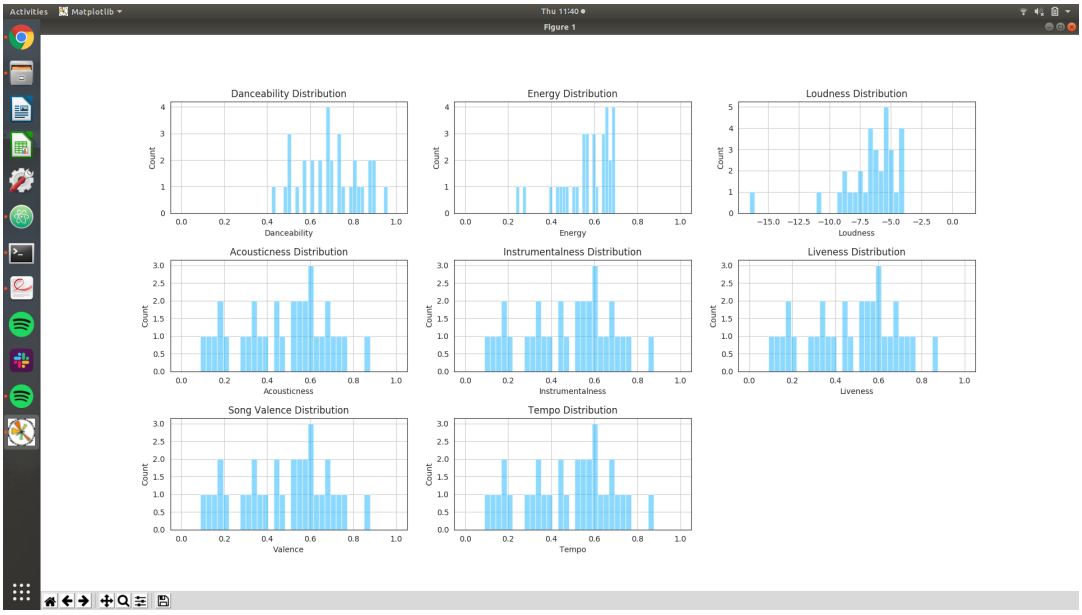
```
ax1 = fig2.add_subplot(331)
ax1.set_xlabel('Danceability')
ax1.set_ylabel('Count')
ax1.set_title('Danceability Distribution')
pos_dance.hist(alpha= 0.5, bins=30)
ax2 = fig2.add_subplot(331)
neg_dance.hist(alpha= 0.5, bins=30)
```

Figure 4.8: Plot Template

As seen above, eight graphs are generated, one for each available end point. The graphs show the distribution of the end point values, as well as the amount of tracks that have each value.



The above graphs are generated from Spotify's "Today's Top Hits" playlist [8], which consists of the current 50 most popular songs in Spotify's catalog. The values are reflective of current pop music, with a heavy skew towards high energy music, which is also danceable. Interestingly, valence is fairly evenly distributed, suggesting that some of the tracks subject matters are sadder and more negative, but are still able to sound energetic and danceable.



These resulting graphs were aggregated from a playlist generated from "Today's Top Hits" [23], with the only applied requirement being energy less than 0.7. Of course, the energy of the new playlist abruptly stops at .7, but the interesting part is how several other end points are affected by this change. Loudness distribution is limited to below -3.5 dB, which suggests that loudness is a major segment of the formula used to calculate energy.

4.5 Playlist Creation Efficiency

```
if danceL <= songF["danceability"] <= danceH:
  if energyL <= songF["energy"] <= energyH:
    if loudL <= songF["loudness"] <= loudH:
      if acousticL <= songF["acousticness"] <= acousticH:
        if instrumentL <= songF["instrumentalness"] <= instrumentH:
          if livenessL <= songF["liveness"] <= livenessH:
            if valenceL <= songF["valence"] <= valenceH:
              if tempoL <= songF["tempo"] <= tempoH:
                return True
```

Figure 4.9: The sortSongs function

Due to the design of the function used to pick out songs that match the user criteria, the playlist generation is the most time consuming element of the program, most notably when applying to long playlists. Because of this, the execution of each end point was measured, using the "time" Python library. This was an attempt to see if the different endpoints took longer to sort than others, and if the sorting process execution time was affected at all by the user defined range of values. The Spotify-curated "Today's Top Hits" playlist [23] was used to check each value. The

first trials involved changing the ranges one end point at a time, which resulted in a negligble amount of difference in execution time for every variable. The next testing method was to attempt shrinking the range of every variable simultaneously, compared to leaving every range completely open. These changes also failed to create any noticeable difference in execution time. This is likely due to the calls to the Spotify database, which store the end points as identical types. Unless a more efficient way to rewrite this method's code is found, the time to generate a playlist likely will not be lowered. The execution time is limited by the necessary API calls, which limits time optimization.

4.6 Feasibility

Due to the intrinsically subjective nature of music, graphical representations which show that the playlist filtering is correctly working is not enough to ensure that the resulting songs form an enjoyable listening experience. To this end, the best testing method would involve human trials. Currently, I am the only person to somewhat formally verify the correctness of the results, but there are several avenues to allow more user feedback, which will likely be pursued after the web portal is more complete. The first would be to simply host the app on a web server, with a feedback form available for all users. Questions relating to the user experience, and more specific inquiries about satisfaction and accuracy of results would be valuable in tuning the metrics for future use. Something as simple as a Google form could be used, with questions including:

- Did the application work correctly?
- Was the creator easy to use?
- How much, if any, music theory background do you have?
- Were you satisfied with the resulting songs?
- Were there any unwanted songs in the new playlist?

This feedback would be useful to gather another, outside perspective from my own, hopefully making the project more accessible to any user. Another available resource is the Spotify Developer Showcase [8], which features user developed projects built using the Spotify API. Submitting Scottipy to the showcase would be another effective way to increase the project's user base. These will be used after completing the web portion of the project.

Chapter 5

Discussion and Future Work

This chapter outlines how the current state of the project compares to the original proposal, including shortcomings and changes in the vision. It finishes with a reflection on the application.

5.1 Future Work

The vision I have for the final version of this application is far beyond what I was able to accomplish in one semester. I would like to eventually create a fully functional web app, with a variety of options in the hopes of using as much of the Spotify API as possible.

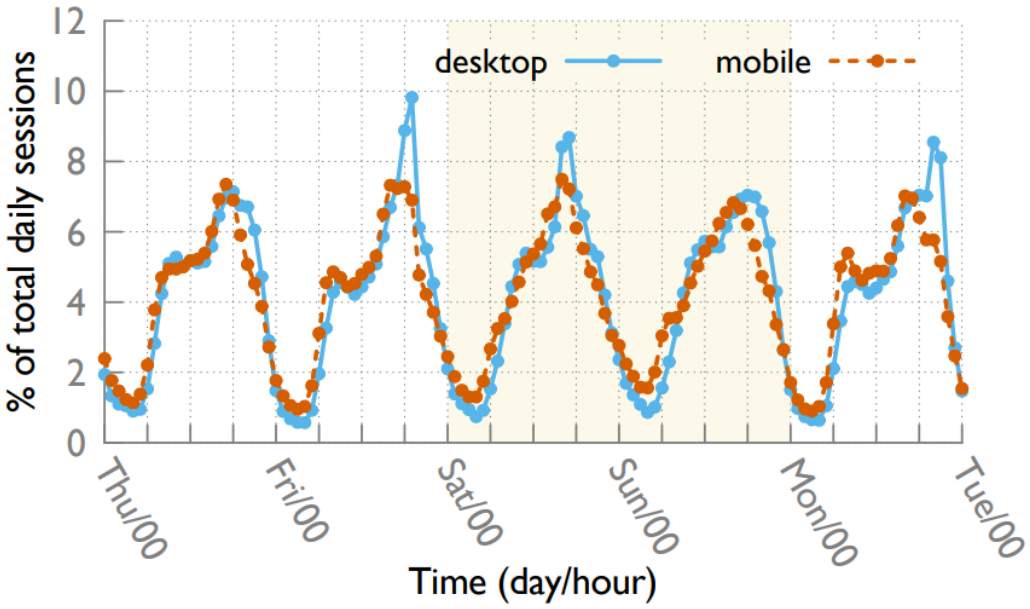
Due to the time constraints associated with this thesis project, the current release is lacking many features that I expect the final application to include. The biggest compromise was in the web page user interface, as I did not have the time to configure every file to work with the Flask framework. Despite this, I am encouraged that my first attempt works correctly, which should serve to make future work easier, or at the least more familiar. If this project is to be used by the general public, it is a necessity to be hosted on a website, with a user interface that is both easy to understand and use. For this reason, creating an entirely web-based application is the most pressing change to work on in the future.

Due to the structure of this project, it is conducive to add new features, without having to weaken the integrity of the overall project. As such, the next feature I would like to add is to trawl playlists for songs, by requesting a keyword from the user. This would allow users to just type in a word, and receive songs that other users have put in playlists related to the term. Since playlists are a major part of Spotify, and contain songs chosen by users, instead of algorithms alone, this is a powerful resource, which arrives pre-sorted. Therefore, choosing songs from playlists will likely add a level of accuracy, and likeliness of enjoyment, that is difficult to replicate by automation alone.

Another feature which I consider very important, especially for public use, is mobile support. Again due to time, I did not have the opportunity to look into how to develop for mobile, but I did learn that Flask can be used for mobile app back-end.

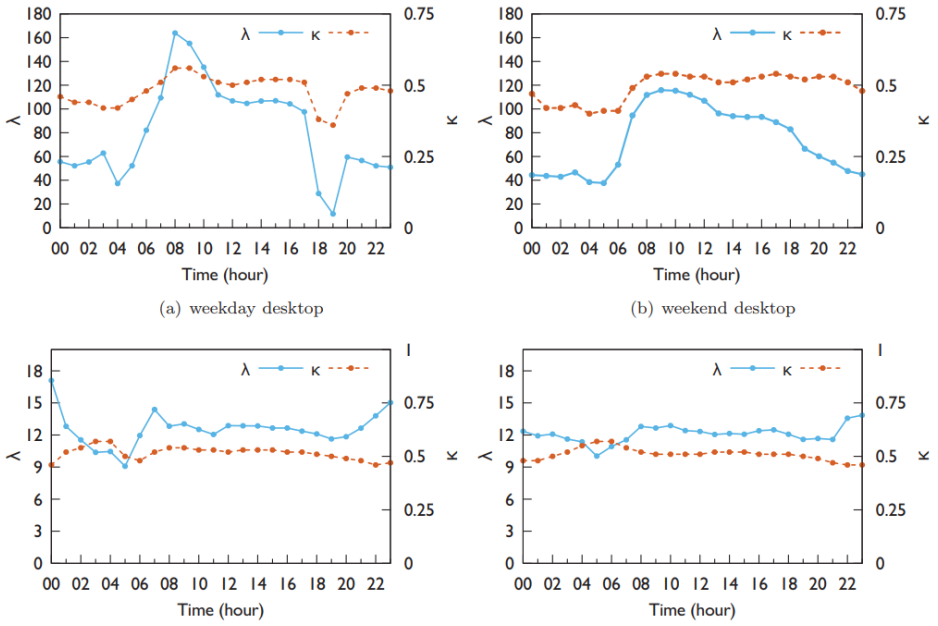
This is also encouraging for the final stage of this project. Like much of the current internet, Spotify relies heavily on mobile usage. This is an area I would like Scottipy to be able to access and address, but will require more work.

A more conceptual change to investigate is to investigate Spotify user behavior, in order to match this project with how users treat Spotify. In a paper focused on this user behavior, user trends, and differences between desktop and mobile usage were found [27]. For one, the research team found that desktop and mobile sessions peaked at different times through the day, and also in different amounts; where destop sessions tended to peak higher than mobile, in terms of percentage of total Spotify users.



These tendencies of mobile users preceding desktop sessions were attributed by the researchers to the number of commuting mobile users. This information may be useful for Scottipy's mobile support, as it would suggest that the mobile view should be very simple, while remaining robust in terms of available adjustable options.

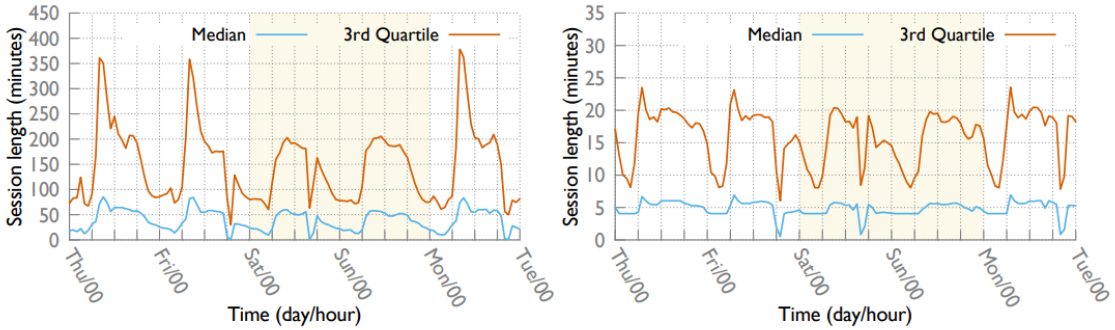
The research team also investigated the average session length of users, both mobile and desktop. The team used Weibull distribution parameters to measure the trends, where λ and κ are respectively the shape and scale of the parameters.



(a) weekday desktop (b) weekend desktop

(c) weekday mobile (d) weekend mobile

The graphs above reveal much information about listening habits. In graph (a), which is for desktop users during weekdays, session length peaks in the morning, and steadily decreases until the evening. This suggests that most destop users are playing songs on Spotify as background music during the workday.



(a) desktop sessions (b) mobile sessions

The above graphs, using the same data, show that on average, mobile sessions are shorter than their desktop companions. In addition, the morning peak is not as pronounced, and there is more overall variation. Since these graphs suggest a difference in listening habits between mobile and desktop, reflecting these practices in Scottipy is an intriguing task. Since desktop sessions tend to be longer, it may be fruitful to increase the default number of songs generated for the platform. On mobile, it would be worthwhile to investigate tailoring the list to shorter playlists, perhaps by involving song popularity, or other efforts that could simultaneously lower the amount of music, but make each track more meaningfully chosen by the data metrics Spotify gives.

5.2 Conclusion

The data that The Echo Nest and Spotify have aggregated makes finding new music the easiest that it has ever been. Despite this, music discovery on the Spotify platform is currently limited to just one weekly static playlist. The project I have outlined will allow music listeners to harness the data that Spotify posesses, in the hopes of finding unfamiliar but enjoyable music. This project aims to make music discovery a more easy and rewarding task than currently possible.

Appendix A

Python Code: `main.py`

```
#!/usr/bin/python

#Author: John Scott
#GitHub: @scottj0
#This work is mine unless otherwise cited.

#This file is the menu for the project.

print ("Welcome to the Scottipy Playlist Generator!")

print("You may choose to create a recently added playlist, or edit
   an existing one!")

while True:
    print ("What would you like to create?")

    choice = input("Enter your choice: ")

    if choice == "recent":
        exec(open("./recent.py").read()) #executes "recent.py"
           script

    elif choice == "edit":
        exec(open("./edit.py").read()) #executes "edit.py"

    elif choice == "exit":
        print ("Goodbye!")
        break

    else:
        print ("Not an option.")
```

Appendix B

Python Code: recent.py

```
#! /usr/bin/env python

#Author: John Scott
#GitHub: @scottj0
#This work is mine unless otherwise cited.

#This file creates a playlist from recently added music.
#It also serves as a proof of concept for a web app, and Spotipy/
    HTML integration.

from flask import Flask, request, render_template, redirect, url_for
import spotipy
from spotipy import oauth2
from webbrowser import open_new_tab
import random
import datetime
from threading import Thread
from queue import Queue

#create the flask application
app = Flask(__name__)

#get the spotify api keys from key.py
from key import SPOTIPY_CLIENT_ID, SPOTIPY_CLIENT_SECRET

#don't change these!
SPOTIPY_REDIRECT_URI = 'http://localhost:5000/'
SCOPE = 'user-library-read user-read-recently-played playlist-modify
    -public user-top-read'
CACHE = '.spotipyoauthcache'

sp_oauth = oauth2.SpotifyOAuth( SPOTIPY_CLIENT_ID,
    SPOTIPY_CLIENT_SECRET, SPOTIPY_REDIRECT_URI, scope=SCOPE,
    cache_path=CACHE)

#global spotify object forward instatiation (defined in index())
sp = None
```

```
#global multiprocessing variables
q = Queue()
t = None

@app.route('/',methods=["POST","GET"])
def index():
    #Authorization workflow borrowed heavily from github user
       perelin
    #original found here: https://github.com/perelin/
       spotipy_oauth_demo
    global sp
    access_token = ""

    token_info = sp_oauth.get_cached_token()

    #is there a cached token?
    if token_info:
        print ("Found cached token!")
        access_token = token_info['access_token']

    #try to get a new access token
    else:
        url = request.url
        code = sp_oauth.parse_response_code(url)
        if code:
            print ("Found Spotify auth code in Request URL! Trying
               to get valid access token...")
            token_info = sp_oauth.get_access_token(code)
            access_token = token_info['access_token']

    #if we have everything we need, so create the spotify object
    if access_token:
        print ("Access token available! Creating spotify object")
        sp = spotipy.Spotify(access_token)
        return redirect(url_for("runRecentlyAdded"))

    #need to prompt user to log in
    else:
        auth_url = getSPOauthURI()
        return render_template("index.html", auth_url=auth_url)


def getSPOauthURI():
    #helper for creating a oauth url
    auth_url = sp_oauth.get_authorize_url()
    return auth_url


@app.route("/runRecentlyAdded")
def runRecentlyAdded():
    #this is primarily just a form page
    return render_template("runRecentlyAdded.html")
```

```
@app.route("/run", methods=['POST', 'GET'])
def run():
    global t, q

    #has the thread already been created?
    if t is not None:
        #update the progress bar
        if t.is_alive():
            progress = q.get()
            return render_template("running.html", progress=progress
                )

        #everything is done, so success! Reset the thread and queue
        else:
            t.join()
            t = None
            q.queue.clear()
            return redirect(url_for("success"))

    #no existing thread. Create worker thread...
    else:
        #get info out of form from runRecentlyAdded
        playlistLength = int(request.form.get("playlistLength"))
        maxSongs = int(request.form.get("maxSongs"))
        shuffle = bool(request.form.get("shuffle"))
        #create and start thread
        t = Thread(target=createNewlyAddedPlaylist, args=(q,
            playlistLength, maxSongs, shuffle))
        t.start()
        return render_template("running.html", progress=1)



@app.route("/success")
def success():
    return render_template("success.html")

def filterPlaylists(songs, artistDict, maxSongsPerArtist):
    songList = []

        #loop through items
        #populate dict with artist -> [songs]
    for song in songs:
        artistName = song['artists'][0]['name']
        if artistName in artistDict.keys():
            #(make sure we don't have duplicate songs)
            if song not in artistDict[artistName]:
                artistDict[artistName].append(song)
        else:
            artistDict[artistName] = [song]
```

```
        #for each artist, check the number of songs, randomly choose
            maxSongsPerArtist to add to list
    for artist in artistDict:
        artistSongList = artistDict[artist]
        numSongs = len(artistSongList)
        if numSongs < maxSongsPerArtist:
            #less than max, so put in all songs
            songList += artistSongList
        else:
            #randomly select using sample function
            selection = list(sample(numSongs,maxSongsPerArtist))
            songList += list(artistSongList[i] for i in selection)

    #return the list
    return songList


def getSongList(playlistLength, callNo=0):
    #returns a list of track items, in order of most recently added
    #https://developer.spotify.com/web-api/object-model/#track-
       object-full

    #note: can only get maximum of 50 at a time

    global sp

    #keep track of the number of times we call this, so we can do
       the offset correctly
    if callNo == 0:
        offset = 0
    else:
        offset = callNo*min(playlistLength, 50)

    results = sp.current_user_saved_tracks(limit=min(playlistLength,
        50), offset=offset)

    songsList = []
    #get rid of some of the extra info. Only have song objects
    for song in results['items']:
        songsList.append(song['track'])

    return songsList


def sample(n, r):
    #taken from http://code.activestate.com/recipes/272884-random-
       samples-without-replacement/
    #Generate r randomly chosen, sorted integers from [0,n)
    rand = random.random
    pop = n
    for samp in range(r, 0, -1):
        cumprob = 1.0
```

```
        x = rand()
        while x < cumprob:
            cumprob -= cumprob * samp / pop
            pop -= 1
        yield n-pop-1


def createNewlyAddedPlaylist(queue, playlistLength=50,
   maxSongsPerArtist=4, shuffle=False):
    global sp

    #populate the queue with a starting point
    queue.put(1)

    #get current user's id
    result = sp.current_user()
    userID = result['id']

    #get the current user's playlists (assumes less than 50)
    result = sp.current_user_playlists()
    playlists = result['items']

    #remove the old version of the playlist, if exists
    for playlist in playlists:
        if 'Recently Added [auto]' in playlist['name']:
            sp.user_playlist_unfollow(userID,playlist['id'])

    #create new playlist
    date = datetime.datetime.now()
    result = sp.user_playlist_create(userID, "Recently Added *auto*
       (" + date.strftime("%m/%d/%Y") + ")")
    newPlaylistID = result['id']

    #get most recent songs
    songs = getSongList(playlistLength)

    #filter the playlist to remove duplicate songs by artist (pass
       in empty dict)
    artistDict = {}
    songs = filterPlaylists(songs, artistDict, maxSongsPerArtist)

    #check to make sure at least playlistLength
    i = 1
    while len(songs) < playlistLength:
        #need more so ask spotify for more songs, filter (using dict
           ), then append to list
        songs += getSongList(playlistLength, callNo=i)
        songs = filterPlaylists(songs, artistDict, maxSongsPerArtist
           )
        i += 1
        #update progress to the main process
        queue.put(int((len(songs)/50)*100))
```

```
    #truncate list
    songs = songs[0:playlistLength]

    #add all songs in list to playlist
    songIDs = []
    for song in songs:
        songIDs.append(song['id'])

    #shuffle the playlist if desired
    if shuffle:
        random.shuffle(songIDs)

    #commit songs to the playlist
    sp.user_playlist_add_tracks(userID, newPlaylistID, songIDs)

    #end signal
    queue.put(-1)


if __name__ == "__main__":
    #open up a new webpage tab for the localhost server
    open_new_tab("http://localhost:5000")

    #run the flask server
    app.run()
```

Appendix C

Python Code: `edit.py`

```
#Author: John Scott
#GitHub: @scottj0
#This work is mine unless otherwise cited.

#This file contains the code used to sort existing playlists, by
   user defined criteria.
#The available conditions are taken from Spotify database end points
    publicly available.
#Graphs of the end points are also generated.

import spotipy
import spotipy.util as util
import random
from webbrowser import open_new_tab
from key import SPOTIPY_CLIENT_ID, SPOTIPY_CLIENT_SECRET #gets
   secret user keys from key.py
import base64 #this is needed to decode the playlist names from
   bytes to strings

import pandas as pd #Dataframe, Series
import numpy as np

from matplotlib import pyplot as plt
import seaborn as sns

import graphviz
import pydotplus
import io

import time

from scipy import misc

from sklearn.metrics import accuracy_score

a='cm9vdA=='
b=base64.b64decode(a).decode('utf-8')
```

```
red_blue = ['#19B5FE', '#EF4836']
palette = sns.color_palette(red_blue)
sns.set_palette(palette)
sns.set_style('white')

class User():
  def __init__(self):
    self.CLIENT_ID = SPOTIPY_CLIENT_ID
    self.CLIENT_SECRET = SPOTIPY_CLIENT_SECRET
    self.REDIRECT_URI = "http://localhost:5000"
    self.SCOPE = "playlist-read-private playlist-modify-private
       playlist-read-collaborative playlist-modify-public" #Allows
       program to access/edit the user's private and public
       playlists
    self.sp = self.getUser() #Creates Spotify instance
    self.id = self.sp.me()["id"] #Gets ID of authenticating user

  def getUser(self):
    #This function is required to authorize the application
    token = self.getUserToken()
    sp = spotipy.Spotify(auth=token)
    sp.trace = False
    return sp

  def getFeatures(self, track):
    #This function retrieves audio features from Spotify
    features = self.sp.audio_features(track)
    return features

  def getPlaylist(self):
    #This function gets all playlists from the user.
    results = self.sp.current_user_playlists()
    for i, item in enumerate(results["items"]):
      print ("{number} {name}".format(number=i, name=item["name"]))
         #Prints out the name of each playlist and a corresponding
         number

    choice = input("Please choose a playlist number: ")
    return results["items"][int(choice)]["id"]

  def getSongs(self, playlist_id):
    #This function gets the track IDs from the songs in the selected
        playlist.
    #It also generates graphs showing the features of the playlists'
        songs.
    results = self.sp.user_playlist_tracks(self.id,playlist_id)
    tracks = results["items"]
    song_ids = []
    while results["next"]:
      results = self.sp.next(results)
      tracks.extend(results["items"])
    for song in tracks:
```

```
    song_ids.append(song["track"]["id"])

features = []
j = 0
for i in range(0,len(song_ids),50):
    audio_features = self.sp.audio_features(song_ids[i:i+50])
    for track in audio_features:
        features.append(track)
        track = tracks[j]
        j= j+1
        #features[-1]['trackPopularity'] = track['track']['
            popularity']
        #features[-1]['artistPopularity'] = self.sp.artist(track['
            track']['artists'][0]['id'])['popularity']
        features[-1]['target'] = 1
j = 0

trainingData = pd.DataFrame(features)
trainingData.head()

#train, test = train_test_split(trainingData, test_size = 0.15)
#print("Training size: {}, Test size: {}".format(len(train),len(
    test)))

red_blue = ['#19B5FE', '#EF4836']
palette = sns.color_palette(red_blue)
sns.set_palette(palette)
sns.set_style('white')

pos_dance = trainingData[trainingData['target'] == 1]['
    danceability']
neg_dance = trainingData[trainingData['target'] == 0]['
    danceability']
pos_energy = trainingData[trainingData['target'] == 1]['energy']
neg_energy = trainingData[trainingData['target'] == 0]['energy']
pos_loudness = trainingData[trainingData['target'] == 1]['
    loudness']
neg_loudness = trainingData[trainingData['target'] == 0]['
    loudness']
pos_acousticness = trainingData[trainingData['target'] == 1]['
    acousticness']
neg_acousticness = trainingData[trainingData['target'] == 0]['
    acousticness']
pos_instrumentalness = trainingData[trainingData['target'] ==
    1]['instrumentalness']
neg_instrumentalness = trainingData[trainingData['target'] ==
    0]['instrumentalness']
pos_liveness = trainingData[trainingData['target'] == 1]['
    liveness']
neg_liveness = trainingData[trainingData['target'] == 0]['
    liveness']
```

```
pos_valence = trainingData[trainingData['target'] == 1]['valence
    ']
neg_valence = trainingData[trainingData['target'] == 0]['valence
    ']
pos_tempo = trainingData[trainingData['target'] == 1]['tempo']
neg_tempo = trainingData[trainingData['target'] == 0]['tempo']


fig2 = plt.figure(figsize=(15,13))
plt.subplots_adjust(hspace=0.42)
#Danceability
ax1 = fig2.add_subplot(331)
ax1.set_xlabel('Danceability')
ax1.set_ylabel('Count')
ax1.set_title('Danceability Distribution')
pos_dance.hist(alpha= 0.5, bins=30)
ax2 = fig2.add_subplot(331)
neg_dance.hist(alpha= 0.5, bins=30)

#Energy
ax3 = fig2.add_subplot(332)
ax3.set_xlabel('Energy')
ax3.set_ylabel('Count')
ax3.set_title('Energy Distribution')
pos_energy.hist(alpha= 0.5, bins=30)
ax4 = fig2.add_subplot(332)
neg_energy.hist(alpha= 0.5, bins=30)

#Loudness
ax5 = fig2.add_subplot(333)
ax5.set_xlabel('Loudness')
ax5.set_ylabel('Count')
ax5.set_title('Loudness Distribution')
pos_loudness.hist(alpha= 0.5, bins=30)
ax6 = fig2.add_subplot(333)
neg_loudness.hist(alpha= 0.5, bins=30)

#Acousticness
ax7 = fig2.add_subplot(334)
ax7.set_xlabel('Acousticness')
ax7.set_ylabel('Count')
ax7.set_title('Acousticness Distribution')
pos_valence.hist(alpha= 0.5, bins=30)
ax8 = fig2.add_subplot(334)
neg_valence.hist(alpha= 0.5, bins=30)

#Instrumentalness
ax9 = fig2.add_subplot(335)
ax9.set_xlabel('Instrumentalness')
ax9.set_ylabel('Count')
ax9.set_title('Instrumentalness Distribution')
pos_valence.hist(alpha= 0.5, bins=30)
```

```
    ax10 = fig2.add_subplot(335)
    neg_valence.hist(alpha= 0.5, bins=30)

    #Liveness
    ax11 = fig2.add_subplot(336)
    ax11.set_xlabel('Liveness')
    ax11.set_ylabel('Count')
    ax11.set_title('Liveness Distribution')
    pos_valence.hist(alpha= 0.5, bins=30)
    ax12 = fig2.add_subplot(336)
    neg_valence.hist(alpha= 0.5, bins=30)


    #Valence
    ax13 = fig2.add_subplot(337)
    ax13.set_xlabel('Valence')
    ax13.set_ylabel('Count')
    ax13.set_title('Song Valence Distribution')
    pos_valence.hist(alpha= 0.5, bins=30)
    ax14 = fig2.add_subplot(337)
    neg_valence.hist(alpha= 0.5, bins=30)

    #Tempo
    ax15 = fig2.add_subplot(338)
    ax15.set_xlabel('Tempo')
    ax15.set_ylabel('Count')
    ax15.set_title('Tempo Distribution')
    pos_valence.hist(alpha= 0.5, bins=30)
    ax16 = fig2.add_subplot(338)
    neg_valence.hist(alpha= 0.5, bins=30)

    plt.show()


    return song_ids


def getUserToken(self):
    #This function is for user authentication
    name = "scottjohn0"
    token = util.prompt_for_user_token(username=name,scope=self.
        SCOPE, client_id=self.CLIENT_ID, client_secret=self.
        CLIENT_SECRET, redirect_uri=self.REDIRECT_URI)
    return token

def sortSongs(self, songF, danceL, danceH, energyL, energyH, loudL
    , loudH, acousticL, acousticH,
    instrumentL, instrumentH, livenessL, livenessH, valenceL,
        valenceH, tempoL, tempoH):
    #This function returns true if the required song end points are
        met, adding 'true' to the list
    if danceL <= songF["danceability"] <= danceH:
```

```
      if energyL <= songF["energy"] <= energyH:
        if loudL <= songF["loudness"] <= loudH:
          if acousticL <= songF["acousticness"] <= acousticH:
            if instrumentL <= songF["instrumentalness"] <=
               instrumentH:
              if livenessL <= songF["liveness"] <= livenessH:
                if valenceL <= songF["valence"] <= valenceH:
                  if tempoL <= songF["tempo"] <= tempoH:
                    return True
  def getLimits(self):
    #This function allows the user to set limits on tracks. No
       response takes the lowest or highest in the range.
    danceL = float(input("Danceability minimum (how suitable track
       is for dancing 0.0-1.0): ") or "0")
    danceH = float(input("Danceability maximum: ") or "1")
    energyL = float(input("Energy minimum (intensity, or speed of a
       track 0.0-1.0): ") or "0")
    energyH = float(input("Energy maximum: ") or "1")
    loudL = float(input("Loudness minimum (Overall loudness of a
       track in decibels -60-0): ") or "-60")
    loudH = float(input("Loudness maximum: ") or "0")
    acousticL = float(input("Acousticness minimum (measure of
       whether a track is acoustic 0.0-1): ") or "0")
    acousticH = float(input("Acousticness maximum: ") or "1")
    instrumentL = float(input("Instrumentalness minimum (Predicts
       whether track contains no vocals 0.0-1.0): ") or "0")
    instrumentH = float(input("Instrumentalness maximum: ") or "1")
    livenessL = float(input("Liveness minimum (Detects presence of
       audience 0.0-1.0): ") or "0")
    livenessH = float(input("Liveness maximum: ") or "1")
    valenceL = float(input("Valence minimum (Positivity measurement
       0.0-1.0): ") or "0")
    valenceH = float(input("Valence maximum: ") or "1")
    tempoL = float(input("Tempo minimum: ") or "0")
    tempoH = float(input("Tempo maximum: ") or "300")
    name = input("Please name your playlist: ") #allows user to name
        the new playlist
    return [danceL, danceH, energyL, energyH, loudL, loudH,
       acousticL, acousticH, instrumentL, instrumentH, livenessL,
       livenessH, valenceL, valenceH, tempoL, tempoH, name]

  def createPlaylist(self, title, tracks):
    #Makes the playlist
    playlist = self.sp.user_playlist_create(self.id, title, False)
    for track in tracks:
      self.sp.user_playlist_add_tracks(self.id, playlist['id'], [
         track])

    features = []
    j = 0
    for i in range(0,len(tracks),50):
      new_tracks = self.sp.audio_features(tracks[i:i+50])
```

```
  for track in new_tracks:
    features.append(track)
    track = tracks[j]
    j= j+1
    features[-1]['target'] = 1
j = 0

trainingData = pd.DataFrame(features)
trainingData.head()

red_blue = ['#19B5FE', '#EF4836']
palette = sns.color_palette(red_blue)
sns.set_palette(palette)
sns.set_style('white')

pos_dance = trainingData[trainingData['target'] == 1]['
   danceability']
neg_dance = trainingData[trainingData['target'] == 0]['
   danceability']
pos_energy = trainingData[trainingData['target'] == 1]['energy']
neg_energy = trainingData[trainingData['target'] == 0]['energy']
pos_loudness = trainingData[trainingData['target'] == 1]['
   loudness']
neg_loudness = trainingData[trainingData['target'] == 0]['
   loudness']
pos_acousticness = trainingData[trainingData['target'] == 1]['
   acousticness']
neg_acousticness = trainingData[trainingData['target'] == 0]['
   acousticness']
pos_instrumentalness = trainingData[trainingData['target'] ==
   1]['instrumentalness']
neg_instrumentalness = trainingData[trainingData['target'] ==
   0]['instrumentalness']
pos_liveness = trainingData[trainingData['target'] == 1]['
   liveness']
neg_liveness = trainingData[trainingData['target'] == 0]['
   liveness']
pos_valence = trainingData[trainingData['target'] == 1]['valence
   ']
neg_valence = trainingData[trainingData['target'] == 0]['valence
   ']
pos_tempo = trainingData[trainingData['target'] == 1]['tempo']
neg_tempo = trainingData[trainingData['target'] == 0]['tempo']


fig = plt.figure(figsize=(15,15))
plt.subplots_adjust(hspace=0.42)
#Danceability
ax1 = fig.add_subplot(331)
ax1.set_xlabel('Danceability')
ax1.set_ylabel('Count')
ax1.set_title('Danceability Distribution')
```

```
pos_dance.hist(alpha= 0.5, bins=30)
ax2 = fig.add_subplot(331)
neg_dance.hist(alpha= 0.5, bins=30)

#Energy
ax3 = fig.add_subplot(332)
ax3.set_xlabel('Energy')
ax3.set_ylabel('Count')
ax3.set_title('Energy Distribution')
pos_energy.hist(alpha= 0.5, bins=30)
ax4 = fig.add_subplot(332)
neg_energy.hist(alpha= 0.5, bins=30)

#Loudness
ax5 = fig.add_subplot(333)
ax5.set_xlabel('Loudness')
ax5.set_ylabel('Count')
ax5.set_title('Loudness Distribution')
pos_loudness.hist(alpha= 0.5, bins=30)
ax6 = fig.add_subplot(333)
neg_loudness.hist(alpha= 0.5, bins=30)

#Acousticness
ax7 = fig.add_subplot(334)
ax7.set_xlabel('Acousticness')
ax7.set_ylabel('Count')
ax7.set_title('Acousticness Distribution')
pos_valence.hist(alpha= 0.5, bins=30)
ax8 = fig.add_subplot(334)
neg_valence.hist(alpha= 0.5, bins=30)

#Instrumentalness
ax9 = fig.add_subplot(335)
ax9.set_xlabel('Instrumentalness')
ax9.set_ylabel('Count')
ax9.set_title('Instrumentalness Distribution')
pos_valence.hist(alpha= 0.5, bins=30)
ax10 = fig.add_subplot(335)
neg_valence.hist(alpha= 0.5, bins=30)

#Liveness
ax11 = fig.add_subplot(336)
ax11.set_xlabel('Liveness')
ax11.set_ylabel('Count')
ax11.set_title('Liveness Distribution')
pos_valence.hist(alpha= 0.5, bins=30)
ax12 = fig.add_subplot(336)
neg_valence.hist(alpha= 0.5, bins=30)


#Valence
ax13 = fig.add_subplot(337)
```

```
    ax13.set_xlabel('Valence')
    ax13.set_ylabel('Count')
    ax13.set_title('Song Valence Distribution')
    pos_valence.hist(alpha= 0.5, bins=30)
    ax14 = fig.add_subplot(337)
    neg_valence.hist(alpha= 0.5, bins=30)

    #Tempo
    ax15 = fig.add_subplot(338)
    ax15.set_xlabel('Tempo')
    ax15.set_ylabel('Count')
    ax15.set_title('Tempo Distribution')
    pos_valence.hist(alpha= 0.5, bins=30)
    ax16 = fig.add_subplot(338)
    neg_valence.hist(alpha= 0.5, bins=30)

    plt.show()

    open_new_tab("https://open.spotify.com/collection/playlists") #
       uses web browser to view playlist

  def main(self):
    playlist = self.getPlaylist()
    songs = self.getSongs(playlist)
    newPlaylist = []
    pref = self.getLimits()
    for song_id in songs:
      song = self.getFeatures([song_id])
      if self.sortSongs(song[0], pref[0], pref[1], pref[2], pref[3],
          pref[4], pref[5], pref[6], pref[7], pref[8], pref[9], pref
         [10], pref[11], pref[12], pref[13], pref[14], pref[15]):
        newPlaylist.append(song[0]['id'])

    self.createPlaylist(pref[16], newPlaylist)

if __name__ == "__main__":
  SpotifyUser = User()
  SpotifyUser.main()
```

Appendix D

HTML Templates

D.1 index.html

```
<!DOCTYPE html>
<html>
<body style="background-color:#1ED760;">
<head>
  <center>
  <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/
     bootstrap/4.3.1/css/bootstrap.min.css" integrity="sha384-
     ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/
     iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous" </link>
    {% block head %}
    <title>Recently Added{% block title %}{% endblock %}</title>
  </center>
    {% endblock %}
</head>
<body>
  <center>
    <h1>Recently Added</h1>
    <div></div>

    {% block body %}
    <div></div>
    {% endblock %}
  </center>
</body>
</html>
```

D.2 runRecentlyAdded.html

```
{% extends "index.html" %}
{% block title %}-Recently Added{% endblock %}
{% block head %}{{super()}}{% endblock %}

{% block body %}
<div>
    <form action="/run" method="POST">
    Number of songs: <br>
    <input type="number" name="playlistLength" value="50" min=20>
    <br><br>

    Max songs per artist: <br>
    <input type="number" name="maxSongs" value="3" min=1>
    <br><br>

    <input type="checkbox" name="shuffle" value="True">Shuffle Songs
       ?
    <br><br>

    <input type="submit" value="Create Playlist!" autofocus>
    </form>
</div>

{% endblock %}
```

D.3 running.html

```
{% extends "index.html" %}
{% block title %}- Generating{% endblock %}
{% block head %}{{super()}}
<meta http-equiv="refresh" content="1" >
{% endblock %}



{% block body %}
<div>
    <h3>Generating playlist. Please wait...</h3>
    <progress value= {{ progress }} max="100"></progress>
</div>

{% endblock %}
```

D.4 success.html

```
{% extends "index.html" %}
{% block title %}- Running{% endblock %}
{% block head %}{{super()}}{% endblock %}



{% block body %}
<center>
<h2>Playlist Generated!</h2>
<a href="https://open.spotify.com/collection/playlists/" target="
   _blank">View playlist</a>
<div>
<a href="http://localhost:5000/runRecentlyAdded">Back to menu</a>
</div>
</center>
{% endblock %}
```

Bibliography

[1] Stefan Aleksik. musicscape, 2018. https://github.com/StefanAleksik/musicScape.

[2] Music Industry Blog. Mid-year 2018 streaming market shares. https://musicindustryblog.wordpress.com/2018/09/13/mid-year-2018-streaming-market-shares/.

[3] Bart De Bock. Klarafy, 2017. http://klarafy.klara.be/en/?utm_source=spotify.developer.showcase&utm_medium=referral.

[4] Sophia Ciocca. How does spotify know you so well?, 2017. https://medium.com/s/story/spotifys-discover-weekly-how-machine-learning-finds-your-new-music-19a41ab76e

[5] Josh Constine. Inside the spotify-echo nest skunkworks, 2015. https://techcrunch.com/2014/10/19/the-sonic-mad-scientists/.

[6] J. Stephen Downie. Music information retrieval. *Annual Review of Information Science and Technology 37: 295-340*, 2003.

[7] FAMOUS. Klarafy documentation, 2017. http://klarafy.klara.be/en/about/.

[8] Spotify for Developers. Developer showcase, 2019. https://developer.spotify.com/community/showcase/.

[9] Zach Hammer. Playlist souffle, 2018. https://playlistsouffle.com/playlists.

[10] Alex Heath. Spotify is getting unbelievably good at picking music heres an inside look at how, 2015. https://www.businessinsider.com/inside-spotify-and-the-future-of-music-streaming.

[11] Raymond Hettinger. Random samples without replacement (python recipe), 2004. http://code.activestate.com/recipes/272884-random-samples-without-replacement/.

[12] Chris Johnson. From idea to execution: Spotify's discover weekly, 2015. https://www.slideshare.net/MrChrisJohnson/from-idea-to-execution-spotifys-discover-weekly/31-1_0_0_0_1.

[13] Paul Lamere. Spotipy documentation, 2017. https://spotipy.readthedocs.io/en/latest/.

[14] Joel Lovera. Magicplaylist, 2015. https://magicplaylist.co/#/?_k=x76vtc.

[15] Joel Lovera. Magicplaylist documentation/source code, 2015. https://github.com/loverajoel/magicplaylist.

[16] Matplotlib. User's guide, 2019. https://matplotlib.org/users/index.html.

[17] The Echo Nest. The echo nest. theechonest.com.

[18] The Echo Nest. The echo nest joins spotify. http://blog.echonest.com/post/78749300941/the-echo-nest-joins-spotify.

[19] Adam Pasick. The magic that makes spotify's discover weekly playlists so damn good. 2015. https://qz.com/571007/.

[20] Spotify. Audio features and analysis, 2018. https://developer.spotify.com/discover/.

[21] Spotify. Spotify company info, 2018. https://newsroom.spotify.com/company-info/.

[22] Spotify. Web playback sdk, 2018. https://developer.spotify.com/documentation/web-playback-sdk/.

[23] Spotify. Today's top hits, 2019. https://open.spotify.com/user/spotify/playlist/37i9dQZF1DXcBWIGoYBM5M?si=6oaz5a0SRzu6II1LltFrEA.

[24] Pallets Team. Flask documentation, 2018. http://flask.pocoo.org/docs/1.0/.

[25] PyData Development Team. Pandas documentation, 2019. https://pandas.pydata.org/pandas-docs/stable/getting_started/overview.html.

[26] H. Zamani, M. Schedl, P. Lamere, and C.-W. Chen. An Analysis of Approaches Taken in the ACM RecSys Challenge 2018 for Automatic Music Playlist Continuation. *ArXiv e-prints*, October 2018.

[27] B. Zhang, G. Kreitz, M. Isaksson, J. Ubillos, G. Urdaneta, J. A. Pouwelse, and D. Epema. Understanding user behavior in spotify. In *2013 Proceedings IEEE INFOCOM*, pages 220–224, April 2013.