AI Project Report: Spotify Song Recommender

Project type 9: Exploration of own approach for an application

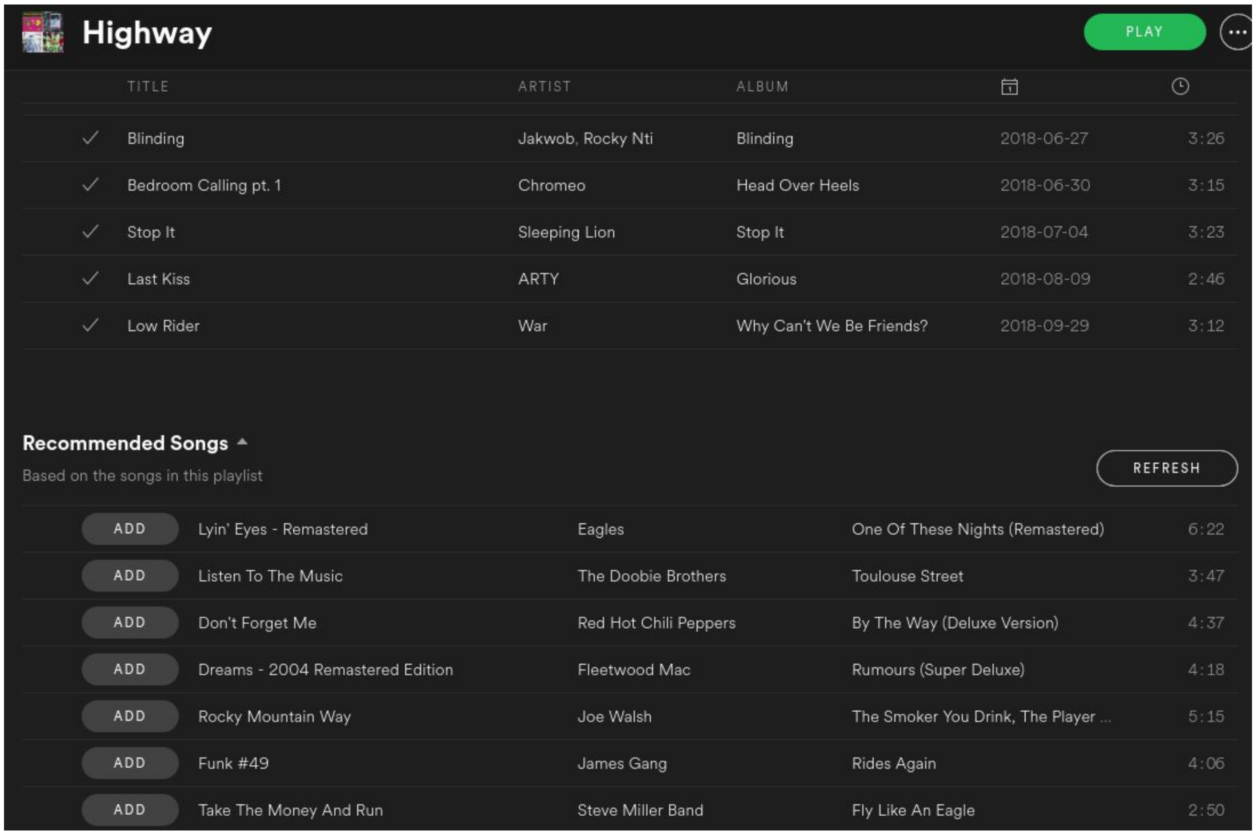
Written by
Caleb St-Denis
Group #15
Student #7256131

For the course
CSI4106 - Introduction to Artificial Intelligence

University of Ottawa
School of Electrical Engineering and Computer Science
December 12, 2018

1 Context

Spotify is a music streaming service with millions of monthly active users [1]. The service offers the ability to make playlists of songs. The playlist interface includes a feature that recommends songs based on the ones in the playlist.



	TITLE	ARTIST	ALBUM	📅	🕒
✓	Blinding	Jakwob, Rocky Nti	Blinding	2018-06-27	3:26
✓	Bedroom Calling pt. 1	Chromeo	Head Over Heels	2018-06-30	3:15
✓	Stop It	Sleeping Lion	Stop It	2018-07-04	3:23
✓	Last Kiss	ARTY	Glorious	2018-08-09	2:46
✓	Low Rider	War	Why Can't We Be Friends?	2018-09-29	3:12

Recommended Songs

Based on the songs in this playlist

ADD	Lyin' Eyes - Remastered	Eagles	One Of These Nights (Remastered)	6:22
ADD	Listen To The Music	The Doobie Brothers	Toulouse Street	3:47
ADD	Don't Forget Me	Red Hot Chili Peppers	By The Way (Deluxe Version)	4:37
ADD	Dreams - 2004 Remastered Edition	Fleetwood Mac	Rumours (Super Deluxe)	4:18
ADD	Rocky Mountain Way	Joe Walsh	The Smoker You Drink, The Player ...	5:15
ADD	Funk #49	James Gang	Rides Again	4:06
ADD	Take The Money And Run	Steve Miller Band	Fly Like An Eagle	2:50

Figure 1.1 - Spotify playlist interface, featuring a song recommender system

The “Recommended Songs” feature is an application of artificial intelligence known as a **recommender system**. Recommender systems can be viewed as a subset of learning systems. A recommender system learns from the likes and/or dislikes of its users to suggest new items that the users may like.

This project was an attempt to build an alternative song recommender for Spotify playlists. For brevity, the alternative song recommender is referred to in this report as “the Application.”

2 Description of the Application

The interface for the Application was implemented as a standalone Android prototype built with React Native. The recommender algorithm itself was implemented in Python using scikit-learn. The recommender communicates with the Android client via a GraphQL API served over Flask.

The code for the Application, as well as instructions for running its components, can be found at https://github.com/calebstdenis/playlist-maker. The instructions for running the Application are also included in Appendix A.

2.1 Interface

The Android prototype has three screens.

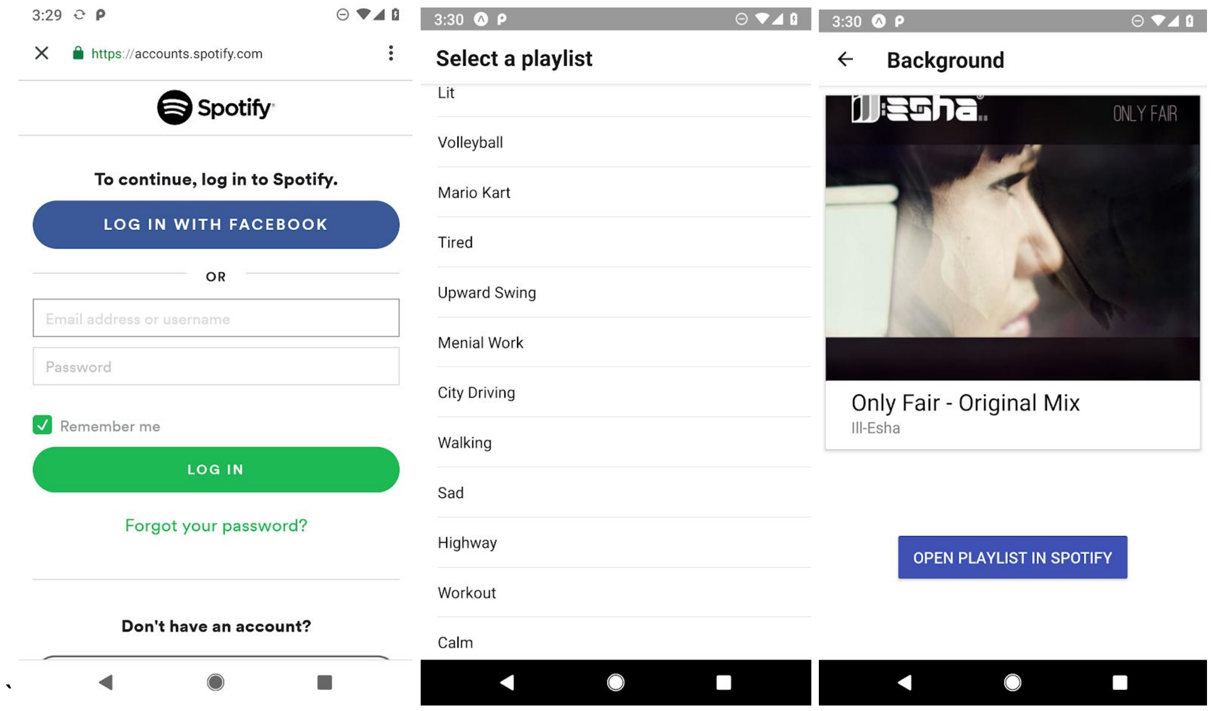


Figure 2.1 - The Application's GUI

The first prompts the user for their Spotify credentials in order to access their data. The second prompts the user to choose a playlist to receive recommendations for.

The third screen presents the song recommendations one at a time. The user can like or dislike the recommendation by swiping the presented card to the left or right, respectively. The Application learns iteratively from the user's feedback.

In theory, when a recommendation is “liked”, it should be added to the respective playlist. This functionality was not implemented in the project prototype given its irrelevance to the AI portion of the project.

2.2 Architecture

The UML package diagram in Figure 2.2 provides an overview of the Application’s structure. It describes the relationship between its modules. The diagram maps directly to the directory structure of the Application. That is, “container” blocks correspond to folders in the code, whereas “leaf” blocks correspond to .js or .py source files.

Note that only the recommender and evaluation modules, marked in yellow in the diagram, apply concepts covered in the AI course. The others are support modules that were needed to provide useful inputs to the recommender and evaluation modules. The implementation of the recommender and evaluation modules are explained in sections 4 and 5, respectively.

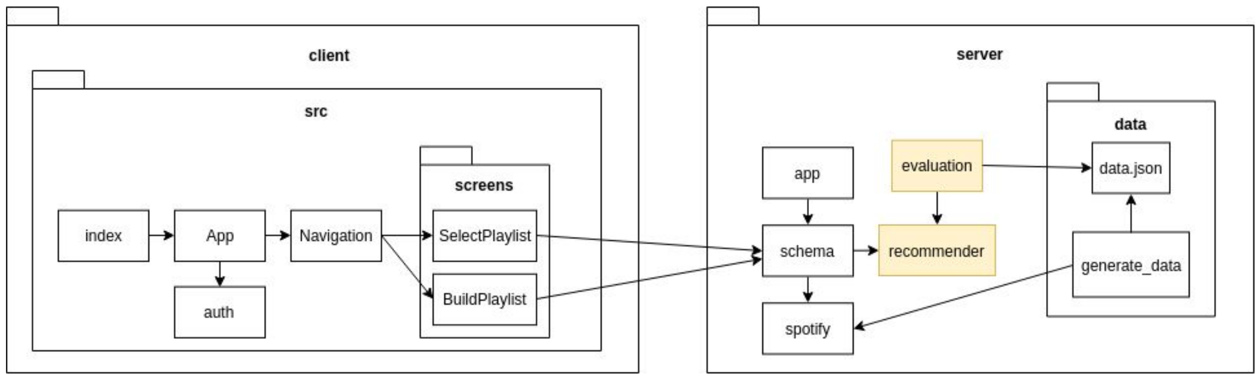


Figure 2.2 - Package diagram of the Application which maps to its code directory structure

2.2.1 Module descriptions

client:

- **index** bootstraps the UI and launches App.
- **App** is the root component of the UI. It ties together the various components of the UI.
- **auth** obtains a Spotify user token, which is needed to access a user’s Spotify data.
- **Navigation** handles the back-and-forth navigation between application **screens**
    - **SelectPlaylist** renders the screen where users select a playlist
    - **BuildPlaylist** displays the recommender’s suggestions and allows the user to like or dislike them

server:

- **app** bootstraps the Flask server.
- **schema** defines the server’s GraphQL API, which allows the client to:

    - select a playlist and begin the recommendation process
    - provide feedback to the recommender module by liking or disliking recommendations

- **spotify** retrieves user data from Spotify and transforms it into the format used by the Android client as well as the recommender.

- **recommender** implements the recommendation algorithm described in section 3.

- **evaluation** does a performance analysis of the recommender relative to a baseline. This module is described in more detail in section 4.

- **data.json** is an export of my Spotify data, consisting of my song library and few of my playlists. The data is formatted so that it can be directly used as input to the recommender. This allows the evaluation module to test the recommender in isolation without having to use all the other modules in the system.

- **generate_data** is a script that generates data.json. It takes as input a Spotify user token and a list of playlist IDs.

3 Recommendation algorithm

The recommendation algorithm is implemented in the class **NaivePUClassifier** in the file **server/recommender.py** (the recommender module in Figure 2.2)**.**

The recommendation algorithm works as follows:

1. Take as inputs:
    - a list of songs representing a Spotify playlist
    - a list of songs representing a user's song library

2. Use as the recommendation pool all the songs in the user's library, minus the songs in the playlist.

3. Use a support vector machine to train a model for binary classification of songs: does a given song belong in the playlist or not? The songs in the playlist are labeled with the positive class (belongs in the playlist). The songs in the recommendation pool are labeled with the negative class (does not belong in the playlist). Both sets of songs are used to train the model.

4. Remove the song from recommendation pool with the highest probability of belonging in the positive class. Recommend this song.

5. Allow the user to like or dislike the recommendation. Label the song as positive or negative accordingly.

6. Retrain the model as in step 3, including the new annotation.
7. Repeat steps 4-7.

The following sections explain the rationale behind the chosen algorithm and explains some of the underlying AI concepts.

3.1 Collaborative filtering vs. content-based filtering

The two most common approaches to recommender systems are *collaborative filtering* and *content-based* filtering. The approaches rely on different inputs [2, p. 16].

Collaborative filtering models rely on the annotations provided by many users. The users' data collaboratively determines similarity between items. If a Spotify user named Bob likes similar songs to another user named Alice, then a collaborative filtering algorithm could detect that similarity, and reasonably recommend other songs from Alice's library to Bob. In the context of the Application, the training inputs to the recommender would be the playlist we're recommending songs for (let's call it playlist X) as well as other playlists Spotify users have made. The task of the recommender would be to find playlists that have many of the same songs as playlist X, then recommend songs from those playlists.

Content-based filtering, on the other hand, looks only at the annotations of the user to whom we are recommending items. Instead of determining similarity from other users' annotations, the content-based approach looks at the attributes of the items from the one user we're recommending to. The algorithm recommends items with similar attributes or features. In terms of the Application, the training inputs to the recommender would be the songs in the playlist we've selected. Spotify provides a set of features for each song, which include "accousticness", "danceability", "energy", among others (see [3]). Suppose a playlist has songs that have high "danceability" values. In this case, the recommender should recommend other songs that are also high in "danceability".

To build the Application with a collaborative filtering approach would require a large bank of Spotify playlists. Spotify has released such datasets in the past, but they are not available to the public.[4] Thus, the collaborative approach is ruled out.

With the content-based approach, the problem can be framed as a binary classification problem similar to the ones seen in class. We have a training set (the songs in the playlist) and a test set (other songs in the user's library) where each instance has a set of features (the song attributes) that can be used to determine its class (whether it belongs in the playlist or not). The Application recommends the songs with the highest probability of belonging in the playlist (i.e. the highest probability of being in the positive class).

3.2 Total bias towards user's song library

Personally, I only make playlists of songs that I already know. Therefore, I built the Application such that it only recommends songs that are in the user's library. This is a departure from Spotify's recommender, which will recommend any song in its offering whether or not the user likes it or has ever heard it.

Limiting the recommendation pool to the user's song library reduces the size of the pool from millions of instances to hundreds. In doing so, I am applying a learner bias based on my domain knowledge, as seen in class.

3.3 One recommendation at a time

The Application must be able to generate recommendations for playlists with only a few songs. In these cases, the training set is very small, thus the recommender is highly prone to overfitting.

To mitigate this, it is essential that the learner improves as quickly as possible. The Application presents only one recommendation at a time, making the user choose or discard it before being presented another. This way, before each new recommendation, the classification is re-trained with the new information from the previous recommendation, and is thus should be able to improve quickly.

In future iterations, the Application's recommender could be enhanced with active learning in order to make it learn more quickly from a small dataset. Active learning differs from traditional machine learning (or passive learning) in that the learner decides what data should be annotated. The goal is to increase the quality of the training data, reducing the need for quantity [5, p. 4]. How the learner chooses its training instances is called a *querying strategy*. One query strategy is to select the instance that the learner is the least confident about. In the context of the Application, this would mean choosing the song whose posterior probability of belonging in the playlist is closest to 50% [5, p. 12].

With active learning, the Application would start by recommending songs that it's uncertain about before it starts recommending songs that are the most likely to belong in the playlist. One would have to define the criteria for when the Application *stops* active learning. Active learning must stop after a certain number of iterations so that the Application can start making "real" recommendations.

3.4 Classification method

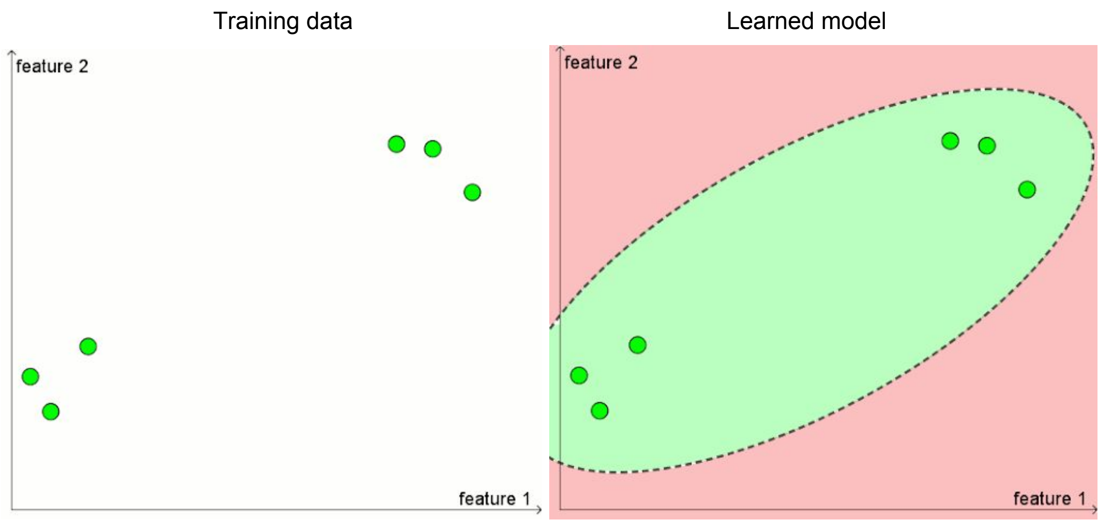
In theory, any machine learning classifier could be used to build a recommender. For use cases with less than 100,000 training samples, scikit learn recommends nearest-neighbor classification (kNN) and support vector classification (SVC) [6]. SVC was chosen over kNN given that the former generally performs better for a small number of data points in a high dimensional space [7].

What is support vector classification? Briefly stated, the n-dimensional feature space of the training dataset (where n is the number of features) is separated in two by a hyperplane such that its margin (the distance from any of the data points) is maximal. If the data is not linearly separable in n-dimensional space, the dimensionality of the data can be increased using a kernel function to make separation easier. The hyperplane is defined by the data points closest to its margin. These data points are called the support vectors [8][9, p. 389]. SVC is effective with small data samples because only the support vectors are used to construct the hyperplane [10].

3.5 Positive-unlabeled learning

We have seen that the Application's recommender system can be framed as a binary classification problem. There is an issue: all our training data belongs in the positive class. When making the very first recommendation, the Application knows only about what songs belong in the playlist and knows nothing about what songs explicitly *do not* belong in the playlist - not until the user begins to dislike recommendations. In fact, the concept of a negative sample is unnatural in many real-world domains [12].

The Application uses a positive-unlabeled learning strategy to attempt to mitigate this issue. Positive-unlabeled learning is a subparadigm of semi-supervised learning [12]. In semi-supervised learning, the model is trained with a small amount of annotated data and a large amount of unlabeled data. This makes the prediction model less prone to overfitting and inaccuracy. This concept is illustrated in figure 3.1.



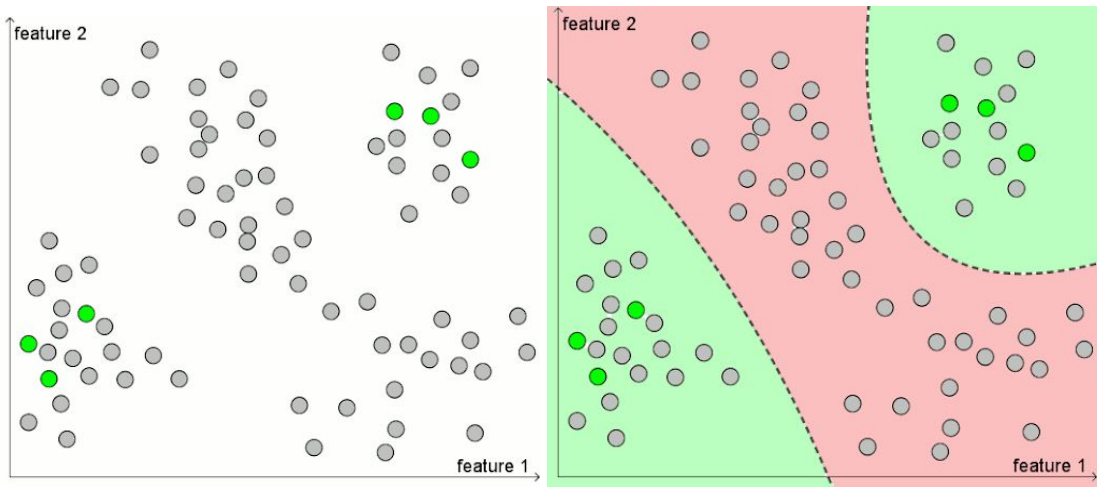


Figure 3.1 - An illustration of learning only on the positive labeled samples (top) versus learning on positive and unlabeled samples (bottom) [12]

The Application uses a simple (if naive) approach to positive-unlabeled learning where the unlabeled data is used in the training set as if it all belonged to the negative class (as seen in [11]). A more sophisticated version of this approach would involve bagging, as seen in class. With this approach, we would “bag” randomly selected subsets of the unlabeled data along with the positive data. We would then perform ensemble learning on the classifiers trained from each bag (as seen in [13]).

4 Evaluation

The recommender evaluation script is implemented in the file **server/evaluation.py.** Instructions for running the script are included in Appendix A.

4.1 Methodology

The recommender algorithm was ranked against a baseline, which recommends songs in whatever order they are retrieved from Spotify. The ranking metric for the recommender is calculated as follows:

1. Take as inputs:
    - a list of songs representing a Spotify playlist
2. Randomly select a third of the songs in the playlist, hide them from the training set, and add them to the recommendation pool.
3. Count how many recommendations are made before all the missing songs have been recommended.
4. Repeat step 2-3 20 times and calculate the mean score.

A higher score indicate poorer performance. It means that the recommender makes more irrelevant recommendations.

Typically, mean average precision (MAP) is used to evaluate recommenders where the items have binary relevance [14, p. 147][15]. Given a list of x recommendations, the recommender is ranked based on the number of relevant recommendations, and their position in the list (higher is better) [14 p. 147][16]. The Application, however, only recommends a single item at a time. The MAP scores would have to be recomputed with each iteration. For simplicity, the scoring system described above was used instead.

4.2 Results

The recommender in the Application was evaluated with my top 4 playlists. The maximum score is 411 (the total number of songs in the recommendation pool).

Playlist Name	Baseline score	Recommender score
Calm	159.75	397.9
Run	176.3	381.85
Highway	133.05	180
Sad	152.4	163.5

4.3 Discussion

The results show that the recommender performs significantly worse than its baseline overall. For playlists with a larger number of songs, the recommender almost goes through all the songs in the recommendation pool (nearly reaching the maximum score), almost as if it is actively *avoiding* recommending relevant songs.

It is possible that the attributes used in the current version of the Application are insufficient for making recommendations. Otherwise, the hyperparameters on the classifier may be improperly adjusted. A more sophisticated PU learning strategy, as discussed in section 3.5, may also improve results.

5 Conclusion

Spotify's song recommender can be framed as a binary classification problem similar to the kind we have seen in class. In real-world binary classification problems, it is often the case that the only annotations available in the training set are for the positive class. In these situations, one can apply positive-unlabeled (PU) learning strategies to improve model predictions. Evaluation revealed that the simplistic PU strategy used in the Application was not sufficient to improve the recommender over the baseline. Future versions of the application should use a more sophisticated PU approach, possibly involving ensemble learning.

Over the course of the semester, we learned how to build and use various classifiers. However, this project has taught me something we should not take for granted: choosing a classifier in the first place is a whole problem in itself.

6 References

[1] Spotify. *Spotify Technology S.A. Announces Financial Results for Third Quarter 2018*. November 1, 2018. URL: https://investors.spotify.com/financials/press-release-details/2018/Spotify-Technology-SA-Announces-Financial-Results-for-Third-Quarter-2018/default.aspx (visited on 12/11/2018)

[2] Charu C. Aggarwal. *Recommender Systems: The Textbook*. Springer, 2016.

[3] Spotify. *Get Audio Features for a Track.* URL: https://developer.spotify.com/documentation/web-api/reference/tracks/get-audio-features/ (visited on 12/11/2018)

[4] Ching-Wei Chen. *Introducing The Million Playlist Dataset and RecSys Challenge 2018*. Spotify. URL: https://labs.spotify.com/2018/05/30/introducing-the-million-playlist-dataset-and-recsys-challenge-2018/ (visited on 12/11/2018)

[5] Burr Settles. *Active Learning Literature Survey*. University of Wisconsin–Madison. URL: http://burrsettles.com/pub/settles.activelearning.pdf (visited on 12/11/2018)

[6] Scikit-learn. *Choosing the right estimator.* URL: https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html (visited on 12/11/2018)

[7] Luis Argerich, Data Science Professor at the University of Buenos Aires. *What is better, k-nearest neighbors algorithm (k-NN) or Support Vector Machine (SVM) classifier? Which algorithm is mostly used practically? Which algorithm guarantees reliable detection in unpredictable situations?* Quora. URL: https://www.quora.com/What-is-better-k-nearest-neighbors-algorithm-k-NN-or-Support-Vector-Machine-SVM-classifier-Which-algorithm-is-mostly-used-practically-Which-algorithm-guarantees-reliable-detection-in-unpredictable-situations (visited on 12/11/2018)

[8] OpenCV. *Introduction to Support Vector Machines*. URL: https://docs.opencv.org/2.4/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html (visited on 12/11/2018)

[9] Johan Suykens et al. Advances in Learning Theory: Methods, Models, and Applications, *NATO Science Series III: Computer & Systems Sciences*, vol. 190. IOS Press Amsterdam, 2003.

[10] Lijun Zhao. *Which classifier is the better in case of small data samples?* ResearchGate. URL: https://www.researchgate.net/post/Which_classifier_is_the_better_in_case_of_small_data_samples (visited on 12/12/2018)

[11] Charles Elkan and Keith Noto. 2008. *Learning classifiers from only positive and unlabeled data*. In Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '08). ACM. pp. 213-220.

[12] Roy Wright, *Positive-unlabeled learning*. URL: https://roywright.me/2017/11/16/positive-unlabeled-learning/ (visited on 12/12/2018)

[13] Fantine Mordelet and Jean-Philippe Vert. *A bagging SVM to learn from positive and unlabeled examples*. Pattern Recognition Letters (2013). URL: http://dx.doi.org/10.1016/j.patrec.2013.06.010

[14] Christopher D. Manning, Prabhakar Raghavan, Hinric Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008. URL: https://www.math.unipd.it/~aiolli/corsi/0910/IR/irbookprint.pdf

[15] FastML. 2015. *Evaluating Recommender Systems*. URL: http://fastml.com/evaluating-recommender-systems/ (visited 12/12/2018)

[16] FastML. 2012. *What you wanted to know about Mean Average Precision*. URL: http://fastml.com/what-you-wanted-to-know-about-mean-average-precision/ (visited 12/12/2018)

Appendix A: Getting started

Prerequisites

- Python 3

Installation

First, clone the repo:

```
git clone https://github.com/calebstdenis/playlist-maker.git
```

Then install the depencies:

```
cd playlist-maker/server
pip install pipenv
pipenv install
```

Running the evaluation script

```
pipenv shell
python evaluation.py
```

The script takes approximately 10 minutes to complete on my machine.

Using the Android client

Prerequisities

- adb (comes with Android Studio)
- An Android device with USB debugging enabled
- Expo Client installed on the Android device (available on Google Play)

Running the app client

Start the server

```
cd server
pipenv install
pipenv shell
python app.py
```

Connect the the Android device to the server machine via USB, then run the following command on the server machine:

```
adb reverse tcp:5000 tcp:5000
```

Finally, open the following app in Expo on the Android device: https://expo.io/@calebstdenis/spotify-playlist-maker

Running the app client locally

With Node.js installed and the Android device connected:

```
cd client
npm install
npm start
```

Select "Run on Android device" in the web interface that appears.