# recsforme: A media recommendation web app*

Lucas Kitaev

April 30, 2021

## Introduction

I started working on this in 2017 as my high school senior project. Over the years, I have found myself coming back and making improvements with the new skills I have learned. The website allows users to search for music, TV shows, and movies (these are referred to as "groups"). Users can vote on a group by either "liking" or "disliking" it. Users can view their list of likes and dislikes, and generate recommendations based on the groups they have liked and disliked. The backend of the application is built on the since-renamed Java EE platform for Java 8. The frontend of the application uses a Model-View-ViewModel (MVVM) architecture provided by Knockout and styling is done through Bootstrap.

## The Problem

Turns out the naïve recommendation engine implementation I wrote wasn't the most efficient! To generate recommendations for a user, the algorithm would essentially compare that user to all other users, and rank them based on similarity. The recommendations would then be taken from the users with the highest similarity. You can only imagine this approach would quickly become slow as the number of users increased. Given that this website is mostly for my own academic interest rather than to actually provide a service to real people, efficency is not much of a concern, but I was looking to see if machine learning could help speed up the process.

*https://github.com/bobsmith947/recsforme

# The Solution

Since Java doesn't have the best support for machine learning toolkits, I figured that I should stick with Python to implement the new recommendation engine. The proposed solution is a Python script can be periodically run to generate recommendations for all users. The script can communicate with the backend Postgres database using Psycopg.

## Data

Group information is taken from the MusicBrainz Database. As the name suggests, this only covers music groups, likes artists and albums. The website previously used an alternative API similar to IMDb to get information on movies and TV shows, however that has since been removed for the purposes of this project. After importing the MusicBrainz sample dataset, I created a list of users and randomly assigned groups as either likes or dislikes of that user. Obviously, randomly picking things to like and dislike may not accurately represent the taste of most real people, but given the absence of real people to use, this will serve as the training data. No feature selection or extraction methods were used.

## Model

The chosen model is the `SimilarityWeightedAveraging` class from the `fancyimpute` package for matrix completion. This model seemed like a good choice because it expects sparse input, which is reasonable since most users will only ever add a small percentage of the total number of groups to their lists. The input to the model is a $n \times m$ incomplete matrix $A$ where $n$ is the number of users and $m$ is the number of groups. $A_{ij} = 1$ if user $i$ likes group $j$, and $A_{ij} = -1$ if user $i$ dislikes group $j$, otherwise $A_{ij} =$ `NaN`. In order to complete the matrix, the model actually users a similar approach to the algorithm I wrote myself; it calculates similarity between rows (users) and uses those similarities to calculate missing entries. The model is run twice: once for artists and once for albums. Using the completed matrices, at most 10 of the highest scoring groups of both types is added to each user's recommendations. Scores are real numbers between $-1$ and 1; groups must have a score of at least 0 to be added.

## Evaluation

Since it's impossible to know the true completed matrix, and obscuring entries isn't particularly valuable for randomly generated data, there is no scoring metric I'm interested in using to evaluate the model. Instead, I'm more interested in the computational performance of the model. For testing this, I created 1,000 users each with 100 artists and 100 albums on their lists. I ran it on my laptop with a 2 GHz CPU to get an idea of the lower end of performance; only one CPU core was utilized; memory usage peaked at 1.4 GB; 13,430 recommendations were generated in total. The verbose output and execution time (using `time(1)`) is shown on the next page.

```
Album recommendations:
[SimilarityWeightedAveraging] Creating dictionary from matrix  with shape (1000, 18595)
[SimilarityWeightedAveraging] # rows = 1000
[SimilarityWeightedAveraging] # columns = 18513
[SimilarityWeightedAveraging] Computed 61416 similarities between rows

Artist recommendations:
[SimilarityWeightedAveraging] Creating dictionary from matrix  with shape (1000, 184055)
[SimilarityWeightedAveraging] # rows = 1000
[SimilarityWeightedAveraging] # columns = 77367
[SimilarityWeightedAveraging] Computed 1205 similarities between rows

real    22m49.871s
user    20m35.516s
sys     0m13.719s
```

# Conclusion

The implementation of this machine learning system has gone relatively well with no major challenges. There are no particular ethical implications (outside of storing usernames/passwords, which is not included in the scope of this report). The next step for the project will be setting up the website to get recommendations from the database. This will allow me to do further testing by manually creating users that have more representative tastes, and seeing if I can get reasonable recommendations for those users. I can also do more performance tuning, and potentially implement multithreading to allow utilization of multiple CPU cores.