ANIMUSIC - Anime Song Recommender

FINAL REPORT - COMS 6156

Navinashok Swaminathan Uni : NS3886 ns3886@columbia.edu

Synopsis

Animusic is a Chrome extension that recommends anime songs on YouTube. Animusic avails the wealth of structured metadata available publicly on anime-songs at websites like AniDB, Anilist, and MyAnimeList for free. It is designed for a low-resource environment like start-ups, which do not have a large dataset of user-item interaction history. Hence, methods like collaborative filtering that require a large user base or computationally expensive audio analysis-based methods are avoided. We instead investigate the effectiveness of item-based models - transformer-based semantic models (MPNet, MiniLM), traditional vector space model (TF-IDF), and graph-based model (Node2Vec) that are trained only on metadata.

Research Questions

1) How engaged are people with each of these models comparatively?

To answer this question, we explore the average like rate and watch ratio of each of the models.

2) How well do the models adapt over time with feedback?

To answer this question, we explore how, with every playlist fetch, the like rate and watch ratio change per model.

3) How feasible is this recommendation system for a start-up-like environment?

To answer this, we look into the time and cost involved in building such a recommendation system across all steps in the methodology.

Novelty

We explore how item-based recommendation models perform in the real world, since most mainstream recommendation systems do not explicitly use these models. This project will provide a sense of how well start-up-like environments can do without much user data.

Furthermore, we separate the recommendation system from the streaming platform.

Recommendation systems today are integrated into the platforms directly, such as YouTube and Spotify. This project explores how the 2 can be separated, and a recommendation system can be built separately.

One application of the world this opens up is think of browsers - what if, say, browsers like Firefox have a feed consisting of Spotify music you like, YouTube music you like, Pinterest pins, scenic images, all on one page based on user interaction data on these websites. This would be much more user-friendly - an all-in-one place for feed recommendation systems.

This project sets up the base for recommendation systems that are not tied up to platforms. We aggregate information from multiple sources, and we have links to YouTube videos. Then we navigate between YouTube videos, but if needed, we could simply replace some of the YouTube video links with Soundcloud links, and all we would have to do is add columns for Soundcloud links. The models wouldn't need an overhaul. So there is a separation between the underlying streaming platform and the recommendation system that is exhibited by this project.

Dataset, Methodology & Metrics

The top 1500 ranked anime in MyAnimeList is fetched through RESTAPI. This is done in batches with sleep intervals between each call to comply with rate limits. For each anime, selenium was used to scrape the MAL Page to fetch MAL_Title_JP, MAL_Title_En, as well as the AniDb link under the resources section, as these aren't available through the API.
Using the AniDB link, tags for the anime, as well as the list of songs in the anime, and metadata for each song, are scraped. Cleaning is done to avoid duplicate anime song copies.

Using the song title and MAL_Title_JP, a YouTube search query to fetch results is constructed. Then ads and shorts are filtered out to get the first relevant video's link. If there are no relevant results, the same is tried with MAL_Title_EN instead. The YouTube page of the first relevant video is then scraped to fetch likes and view count.

After cleaning the data, we had **14,885** songs for **1,235** anime. Then we train the models offline.

In TF-IDF, all the text columns make up the document per song. We then configure TfidfVectorzer with stop_words='english' to remove words with lesser semantic meaning (such as 'a', 'the', etc). The fitted vectorizer object, tfidf matrix (containing tf_idf vector for each song), and list of video_ids corresponding to each row in the matrix are stored in .pkl format.

A network graph is created with textual columns making up the nodes and numeric columns the edges. A node is created for each unique song, artist, and other textual columns, and popularity and views make up edges with default weights set to 1.0. The Node2Vec library is then used to perform random walks and learn embeddings for each node based on its neighbours. The graph, as well as the fitted Node2vec model containing learned embedding vectors for each node, are stored in .pkl format.

We then use sentence transformers, which are deep neural networks that generate dense embeddings that capture semantics. In this case, we use MPNet (all-mpnet-base-v2) and MiniLM-L6(all-MiniLM-L6-v2) sentence transformers. The list of video IDs as well as the embeddings are stored in .pkl format for both models. The embeddings are 384-dimensional vectors for MiniLM and 768 dimensions for MPNet.

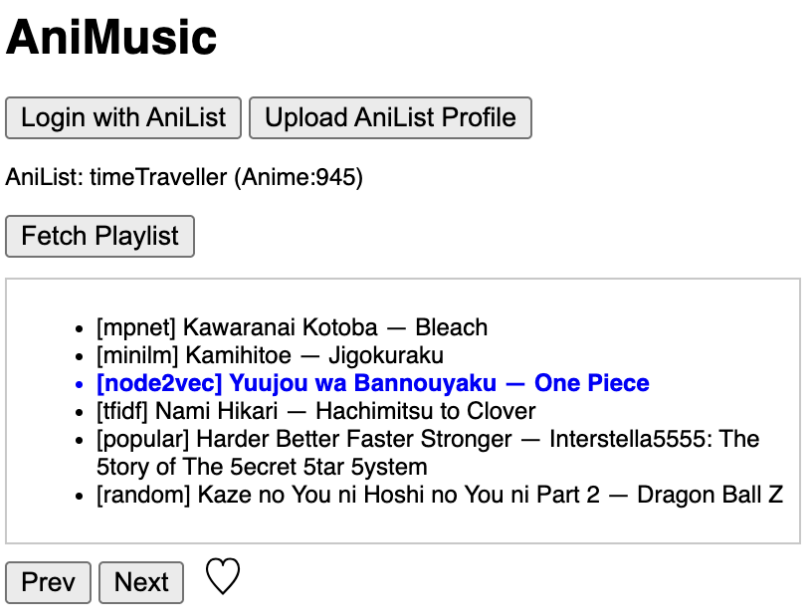
The backend is built using FastAPI, and the containerized backend, along with the pkl files is deployed on Railway. Similarly, the PostgreSQL database is also deployed on Railway. SQLAlchemy is used as the Object Relational Mapper to interact with the PostgreSQL database. The FeedBack table stores anilist_username, song_id, the model that recommended the song, watch time, total video time, as well as whether the user liked the song or not. This is updated for every user-song interaction. The UserAnime table stores Anilist usernames and anime IDs of shows from their anime list.

The endpoints consist of POST /upload_anilist_profile, which receives an anilist username and anime IDs. Existing entries for the user (if they exist) are deleted, and the new entries are added.

POST /feedback receives an object comprising anilist_username, song_id, the model that recommended the song, watch time, total video time, as well as whether the user liked the song or not. If feedback already exists for the song, it is updated, otherwise, a new feedback is created. A point to note is that here watch_time is maximum progress user has made in the video (say if on first watch they watched first 2 minutes, then they switch to some other song, the come back and watch the remaining 1 minute to watch 3 minutes of the song, then watch_time is set to 3 minutes, even if user later replays and provides feedback for the song(say at 1 minute mark) the previous max watch time of 3 minute is still retained). So the watch time is max over all interactions with that song.

GET /playlist takes anilist_username as a parameter, and fetches their watched anime_ids as well as past feedback from the database, then passes it to the recommender module. A ‘skip set’ of all songs already watched is created. These songs won’t be shown to the user again. Favourability towards an anime for a user is computed (every like/ watch_time>50%, is considered favoured), and when the counter is negative the anime is blacklisted. These 2 are put in place to increase randomization of the models. So, from the list of all possible candidate videos, songs belonging to skip_set or blacklisted_anime are removed. (These operations are reverted if the resultant candidate set ends up becoming empty.) A user profile is a list of songs they liked or watched more than 50 percent of. To address cold start issues, songs belonging to anime from their anime list are picked at random and considered to be previously liked by the user. A combined text of all textual features for all songs belonging to the liked set is created. An embedding vector is computed, which is the mean of the embedding vectors of all the songs from the liked list. The respective Recommender methods of all the models are then called, which will give the video ID of the next video to recommend. The video_items of format YouTube URL for the video, anime name, song name, and model that recommended it for all the models

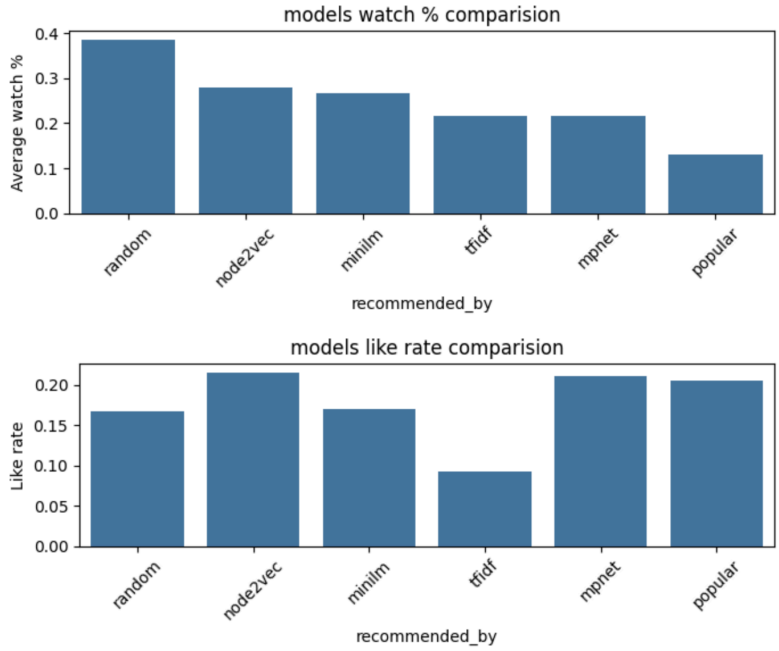
are returned. We also return the next popular video the user hasn't watched yet (labelled as popularity) and a random video tagged as random.



The frontend Chrome extension consists of an interface like the one above. Users click login with Anilist, which opens a pop-up to log in with their Anilist profile. Following which users click upload anilist profile, which calls POST /upload_anilist_profile in the backend. Users can fetch a playlist, which calls GET /playlist and opens a page of the first song. Users can click like on a song they are currently listening to. Users can click prev/next to navigate between songs. On clicking prev/next user's watch time, the current video's ID and total video length are sent to POST /feedback.
The app was open-sourced with a link to github repo of the extension, a post was made in Product Hunt with a video demo on how users can enable developer mode and load unpacked to enable extensions on local machines. The post was updated with links to the Chrome extension after it went live on Chrome following a 1-week review process by Chrome. The published extension is available at https://chromewebstore.google.com/detail/animusic/bpnflfbnnekkledjfcdnkaoanfkflpho. In total i had around **20** users including from both before the product went live on Chrome as well as after.

Results and Findings

Analysis was done to compare the average watch% of recommended videos across models. Similarly, the like rate was also compared. The Node2vec model had the highest like rate, and TF-IDF had the lowest like rate. Random videos had the most watch time, followed by node2vec, with popularity having the least watch time.



Furthermore, the average watch time and like rate were compared for each exposure number. Exposure number n is basically the nth song that the user watches, which is recommended by the model. This is done to compare how the models perform over time with more and more feedback. We also include a more detailed like rate vs exposure number, along with grey bars indicating how many users actually reached an exposure number for a particular model. Figures in Appendix (A.1).

The detailed graph gives a more complete picture. We see that a lot of people skipped random songs, that is, they would click to fetch the next playlist even before getting to the random song. And this especially gets worse with time. Although the earlier bar chart seemed to show more watch time on average per person for random songs, the fact remains that not a lot of people even watched the random songs in the first place. They skipped even before getting to it.

Another trend we notice is that for MPNet and Node2Vec (let's focus on the first 13 exposures, as they had the most users), we see a general downward trend before an upward trend for the like rate. This seems to suggest that the initial cold start recommendations are better than the early feedback-tuned recommendations, but with time and more feedback, the feedback-oriented model performs better than cold start.
Overall, MPNet had the most adaptability - in general, with more feedback, it improved a lot more compared to other models. Closely followed by node2vec.

Answers to research questions

1) How engaged are people with each of these models comparatively?

Like rate and watch % show that people are engaged with the Node2Vec model the most. Followed by transformer-based models.

2) How well do the models adapt over time with feedback?

MpNet adapts best with exposure. Followed by Node2Vec.

3) How feasible is this recommendation system for a start-up-like environment?

From an investment in terms of time, the most amount of time goes into data collection. The operations would be done in batches, making requests up to the daily limit. The collection of data alone took 2-3 weeks. The model training was all completed in minutes. Publishing the Chrome extension took a week-long review process.

In terms of money, no money was used for data collection or model training. All of this was done locally without a GPU. But for hosting the PostgreSQL DB, as well as the backend, it costs around 15 USD. Furthermore, a developer account to publish Chrome extensions costed 5 USD. So, in all, this is extremely cost-effective for a start-up environment. However, more money would have to be spent if they wish to speed up the data collection process or advertise the product to get more users.

Resources/External Materials used

Myanimelist (https://myanimelist.net/), Anidb (https://anidb.net/), Youtube(https://www.youtube.com/), Anilist (https://anilist.co/) APIs were used/were scraped.

FastAPI web framework (https://fastapi.tiangolo.com/), Python libraries such as Sentence-Transformers, Node2Vec, numpy, pandas, scikit-learn, and SQLAlchemy.

Javascript, Chrome Extension Apis (https://developer.chrome.com/docs/extensions/reference/api)(for accessing local storage, scripting, etc)

Docker for containerization(https://www.docker.com/), Railway for backend and db deployment (https://railway.com/), GitHub (https://github.com/) for code hosting.

Demo

The demo was presented in class, link to slides : (https://docs.google.com/presentation/d/1kiz9IVu3cXZ5d9ZbDA1HbJAiYL1jbY8PVK-3uNBdvBk/edit?usp=sharing) covered a video demo of navigating through the chrome extension as well as, the findings and answers to 2 of the research questions (didn't cover feasibility in demo), as well as links to promotion product hunt link and published chrome and edge extension links.

Value and Availability to the user community

People from the anime community can find this extension in Chrome to discover new anime music. Also, since this is open-sourced, they can improve on this to make a full-fledged product containing millions of songs.

Self-Evaluation

Problems Encountered/Didn't go as planned

Getting a user base is a lot more difficult than I expected. Whenever I posted in Discord/reddit/relevant anime communities, my posts/comments were immediately removed, as self-promotion is heavily frowned upon(even if it's entirely free). This was quite a disappointing experience, and hence I pivoted to ProductHunt, where a lot of other folks post their new products/side projects, etc.

Also, the api rate limits were a lot more restrictive than I expected. I couldn't collect information for more than 50 anime in a batch for AniDB, etc.

What I learned/accomplished

While I was already familiar with backend development and ML, Chrome extension development was a new field for me, and I have gained confidence in building Chrome extensions. I had also learnt to build feedback-oriented models by modifying user profile vectors. I have never published to the app store/chrome extension, so the 1 week review period, as well as filling out information regarding permissions, privacy, etc, was also a new experience for me.

Considerations from the revised Proposal and Progress Report

In the proposal, I had discussed using Chrome extension rating as a metric, but as mentioned in the problems encountered section, I didn't get the traffic I expected, and hence have no user ratings/reviews. Another change was that I deployed on the railway instead of AWS. The fast-stretch goal was not targeted.

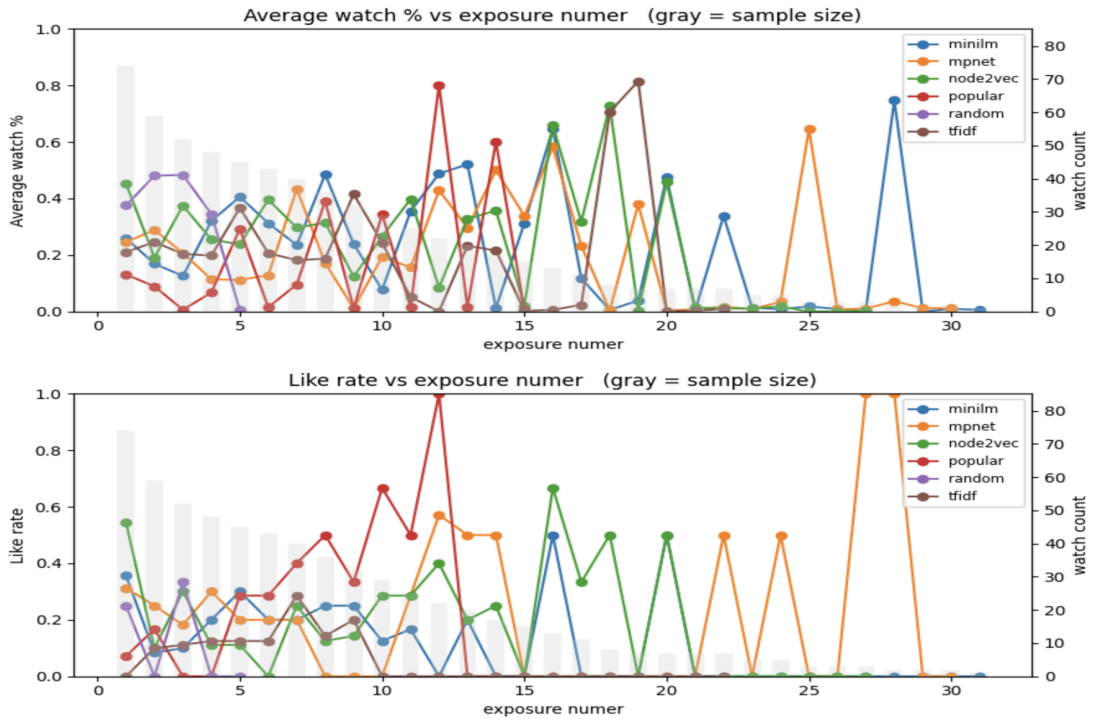
In the progress report, I had mentioned a research question: to what extent are item-only based models feasible **compared to existing** recommendation systems? However, there is no publicly available information on Spotify/YouTube engagement metrics (that is, avg. watch time or avg. like/favourite ratio) to compare our model to. However, we do achieve a respectable 20% like rate for Node2Vec, which is higher than most YouTube Music Videos.

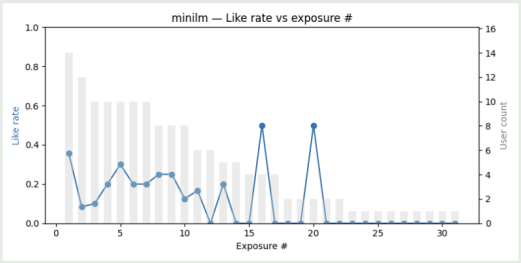
Open-sourced code

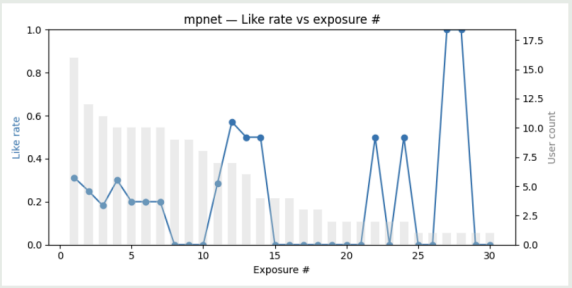
The GitHub repo for the entire project is delivered at:
https://github.com/NavinColumbia/AniMusic_Repo
It contains code for data collection, backend code, frontend Chrome extension code, and scripts for post-deployment analysis. This work is reproducible. ReadMe of the repo contains further details.

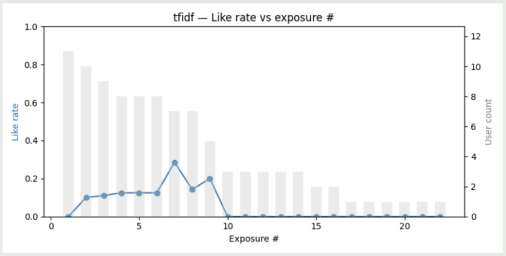
Appendix

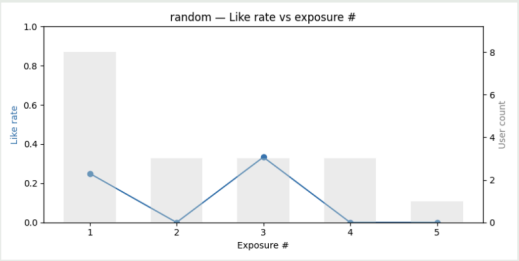
A.1

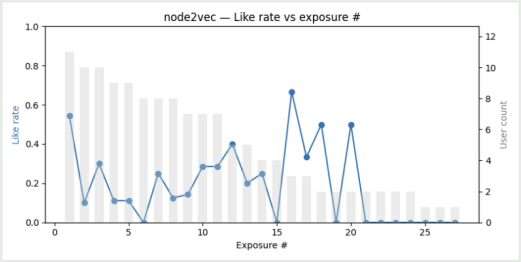


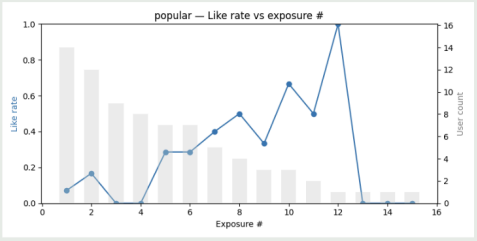

minilm — Like rate vs exposure #
Like rate
User count
Exposure #

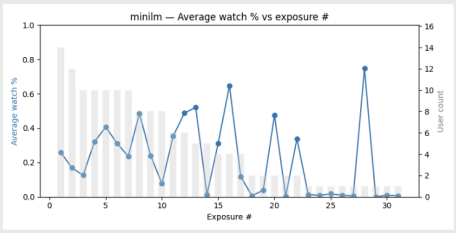

mpnet — Like rate vs exposure #
Like rate
User count
Exposure #

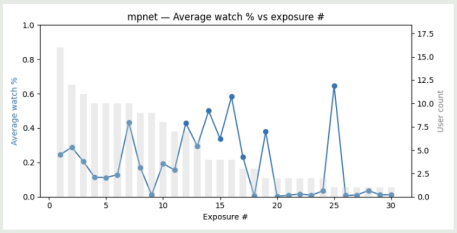

tfidf — Like rate vs exposure #
Like rate
User count
Exposure #

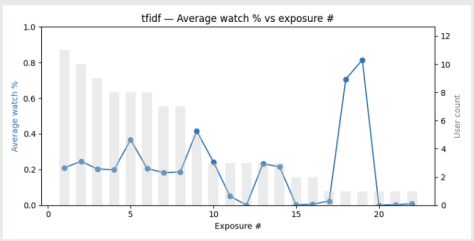

random — Like rate vs exposure #
Like rate
User count
Exposure #

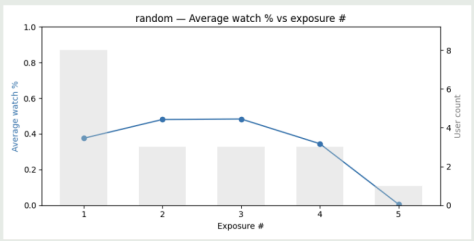

node2vec — Like rate vs exposure #
Like rate
User count
Exposure #

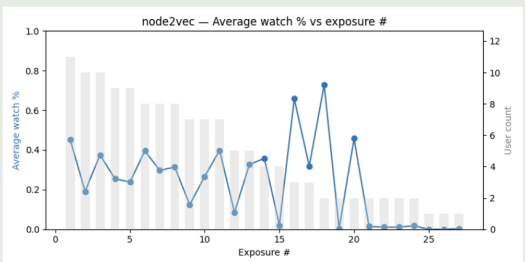

popular — Like rate vs exposure #
Like rate
User count
Exposure #

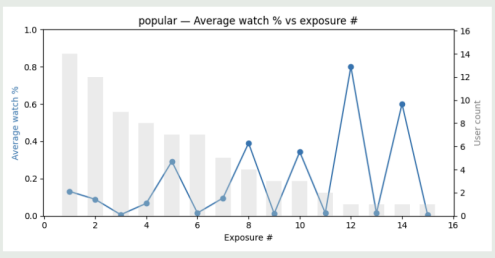

minilm — Average watch % vs exposure #
Average watch %
User count
Exposure #


mpnet — Average watch % vs exposure #
Average watch %
User count
Exposure #


tfidf — Average watch % vs exposure #
Average watch %
User count
Exposure #


random — Average watch % vs exposure #
Average watch %
User count
Exposure #


node2vec — Average watch % vs exposure #
Average watch %
User count
Exposure #


popular — Average watch % vs exposure #
Average watch %
User count
Exposure #