Team 5 M1 Report

Team Members: Benjamin Lo, Fabrizzio Molfino, Gaurav Iyer, Keyu Yao, Yann Bonzom

Learning

Model Overview

Our system is an item-based collaborative filtering movie recommendation engine. It uses movie features such as the genre, overview, production details, and spoken languages to establish similarities between movies. We use cosine similarity to identify how closely movies are related in terms of their contents. Our system suggests movies that are similar in content to those the user has watched, thus personalizing recommendations. For our alternative model, see Appendix: Alternative Model. We qualitatively assessed the two models and found the selected one to provide more meaningful recommendations. The implementation of the model can be found in inferences/main.py.

Recommendation Algorithm

1. From the list of movies, we create a new column called ‘tag’ to combine the overview, genres, production_countries, production_companies, and spoken_languages features. We flatten the dictionary into plain text for each feature, stripping it of any structure.
2. We then apply stemming to the words in the ‘tag’ column using the NLTK library. This helps us standardize the vocabulary to make sure that similar words such as ‘buying’ and ‘buys’ get counted as the same word (in this case, ‘buy’) in future analysis. This facilitates comparing movie tags.
3. The count vectorizer method from scikit-learn to convert the tags column into a numerical matrix that represents the presence of each word in the tag column.
4. For each movie that a given user has watched, we calculate the cosine similarities between that movie and all other movies in our database and append the top 20 most similar ones to an array, avoiding any duplicates.
5. From this large list of movies, we return the top 20 movies sorted by popularity.

Training

Our model is an item-based collaborative filtering model, so it doesn’t have an explicit training phase. It is a lazy learner so it defers the computation until the moment it is run using the data available at the time of prediction.

Evaluation and Limitations

Our model capitalizes on detailed movie features and offers user-independent recommendations, making it suitable for users with limited watch histories. Since the features we consider take into account information about a movie’s content, our recommendations are aimed at being similar in terms of genres/storyline, which we think is a reasonable starting point.

Currently, the model only considers content and does not incorporate movie ratings to make recommendations. Our approach also does not scale well to larger datasets as it is fully in-memory. In addition,our system currently lacks exploration capabilities, which means it primarily recommends movies based strictly on observed content similarities, without introducing potentially new or diverse interests to the user. Overall, it is quite naive and could use some additional complexity. We propose some future steps to address these problems in a later section.

Data and Inference Infrastructure

Overview

For the general setup, we separate it into a data aggregation Docker container and a training+inference container. The data aggregation container aggregates our data into two dictionaries, one for a user database and one for a movie database. These dictionaries are periodically dumped into a pickle file which is stored in a ` volume so that it persists when the container finishes running the data collection process. The training+inference container has the volume with this data pickle file mounted to it so it can load it to a Pandas dataframe on which we compute our model. This container, following this startup model computation period, then functions as the inference API.

Accessing the Apache Kafka Stream

We use the confluent-kafka package, specifically the Consumer class from its Python client, to access messages from the Kafka stream which provides us with user actions. The poll() method allows us to read messages one at a time and perform callbacks to our data aggregator to update our dictionaries. The usage of this API can be seen in /data_parsing/consumer_script.py

Data Aggregation Container

This container connects to the Kafka stream and continuously stores user and movie data and the code is stored in the /data_parsing directory. The Movie and User classes query the course-provided API to collect additional movie and user information. These classes are aggregated into the UserBase() and MovieBase(), which act as wrappers for dictionary objects. The current inference pipeline aggregates data from a fixed number of messages from the Kafka stream (5 million) using our aggregation classes which are then stored in their respective pickled files on disk.

Training+Inference Container

The logic for this is in inferences/main.py. On startup, it loads the dataset from the volume, performs its processing (data preprocessing and calculation of cosine similarities, as described in the Learning section), and then functions as the inference API using Flask. We have also added a /watched/<user_id> endpoint to allow us to verify which movies the user has watched and thus perform a qualitative assessment of the model's recommendations.

Limitations

The most pressing issue is that the data aggregation container crashes after several hours on the VM, likely due to memory constraints. We currently use Python dictionaries that store the whole database in memory and save these to disk using pickle; this is very memory inefficient. This also means that the data we currently have on our users is extremely limited: on a test of querying 68579 users, only 816 had watched a movie given our data, meaning our service only serves approximately 1.19% of users. Additionally, the current model loads the entire dataset to memory to calculate predictions. Due to RAM limitations, this causes the inference container to crash on larger datasets, so this setup cannot scale.

Next Steps

- We are currently using a very small (282MB) and static dataset, and our data aggregation setup does not allow us to scale. Incorporating more data into the model would enable higher-quality recommendations. We're considering moving to a SQL database on Azure so we can store data from the Kafka stream continuously by running the aggregation container constantly.
- We will look into other models that do not require loading the entire dataset into memory for inference. For example, we could train a batched SVD model on a much larger amount of data. We are also considering online training options and considering publicly available pre-trained models for future milestones.
- Plan a life-cycle for our data. Since the Kafka stream is potentially infinite, we would want to investigate a pipeline that might not require storing all our data permanently. For example, a setup could be to store all data gathered from the stream for the day, re-train/fine-tune our model, delete the data, and repeat this every day so that the model is always up-to-date.
- Look into Kafka filters and aggregators so that we process data appropriately before we even save it to the database. We want to become more familiar with and make better use of Kafka's built-in functionality.
- We want to refactor our codebase as soon as possible to accommodate the impactful changes we've planned and remove inefficiencies – for example, at present, both our containers have copies of the code used for data aggregation due to dependencies.

Team Process and Meeting Notes

Reflections and Next Steps

Our team held regular weekly online and in-person meetings every Monday at 2 PM and additional ones as necessary. We mostly followed our initial workflow from A0, but with a few deviations:

1. Instead of using Jira to assign tasks, we relied on our weekly meetings to monitor progress and communicate deadlines. The Jira board felt redundant and wasn't immediately useful, especially since Milestone 1 was challenging to plan. However, our growing understanding will guide better Jira use.
2. Though we intended to rotate between the notetaker and mediator roles, the same individuals often held these positions. This made meetings efficient but limited role diversity. We'll assign fixed roles in the future to simplify things.
3. We used Slack as our primary communication tool. Still, there was a communication gap between the model team and software team. To avoid this lack of communication, we'll focus on checking in more and regularly sharing progress.
4. An unmentioned tool from A0, Gitlabs Wiki, was employed. Members updated it with new code or research, ensuring the team stayed informed. This practice will continue.

Workload Distributions and Individual Contributions

The workload in our team was divided according to the three main sections of our system: 1) the data aggregation system, 2) the recommendation model, and 3) the inference API development and deployment on the VM. Below is the workload distribution and individual contributions for each team member.

Meeting Notes: 18/09/2023 - M1 Planning 25/09/2023 - M1 Next Steps

Benjamin Lo

Ben took charge of the Data Aggregation system, setting up the initial system (commit1) and adding the capability to save the databases with pickle (commit 2). He collaborated closely with Gaurav in both the design and coding phases. Throughout the Docker/script deployment, Ben worked closely with Yann and Gaurav. Additionally, he contributed to the report by helping to write the inference service and team contribution sections. Moving forward, the plan is to develop a server-hosted database, possibly on Azure, to accommodate larger datasets. He was also the mediator during most meetings in practice.

Gaurav Iyer

Gaurav collaborated closely with Ben on the initial data aggregation system. He set up an initial sample dump for the team to work on and successfully researched and executed a basic Confluent Kafka Consumer loop for direct message readings from the stream (see this and this respectively). He focused on the deployment of initial Docker images and shared his insights with the team (see this wiki entry). Gaurav also contributed several ideas on approaching both the model and data infrastructure, in terms of implementation and potential issues. The next step is to enhance the interfacing between the model and infrastructure codebases and explore more task-appropriate recommendation algorithm options.

Yann Bonzom

Yann served as the note-taker throughout all meetings, recording each member's contributions, pinpointing challenges, and setting future tasks. He documented research on Docker, collaborative filtering, and potential model frameworks like SVD in the Wiki. For the inference service deployment, he transformed Keyu's model notebook into a Flask API. This process included cleaning up exploratory code, adding two endpoints via Flask, and managing the Docker build and deployment, all of which is in the inference directory and the main.py file in particular (final commit). In terms of the report, Yann particularly tackled the Data and Inference Infrastructure section while also making contributions to other sections.

Keyu Yao

Keyu developed the main content-based recommender that we currently use. She did so in a notebook, which was transformed into the API by Yann. She collaboratively authored the 'Learning' section of the main model in conjunction with Fabrizzio.

Fabrizzio Molfino

Fabrizzio developed an alternative model (model file), documented below in the appendix. He collaboratively authored the 'Learning' section of the main model in conjunction with Keyu.

Appendix

Alternative Model

Model Overview

Our alternative prediction system uses a collaborative filtering movie recommendation engine based on the Singular Value Decomposition (SVD) algorithm from the Surprise library. Rather than solely focusing on movie content, this system is based on users' movie rating patterns. By understanding how users have rated various movies, the system can predict how a user might rate movies they haven't seen and, subsequently, provide recommendations.

Recommendation Algorithm

1. Retrieve user and movie datasets from saved databases.
2. The user data is expanded to relate individual movie-rating pairs for every user. For missing ratings, the average rating of the movie is used as a substitute.
3. We use this data to fit the SVD based recommender.
4. Then at the time of prediction we first generate the anti-set (list of movies the users haven't rated) using the build_anti_testset method ,then we filter only the missing movies of the current user and subsequently we use the fitted SVD model to generate the missing ratings for all the movies.
5. Lastly, we sort the list according to their rating and take the n highest rated movies.

Training

The library provides a fit method that uses the provided training data to learn how to predict user-item interactions based on a combination of biases and latent factors. It does so by iterating over the data multiple times and adjusting its internal parameters using SGD(Stochastic gradient descent).

Why This Model Wasn’t Selected

Upon comparison, this model predominantly recommends popular or high-rated movies. Given the limited data available for each user, our alternative model delivers more intuitive recommendations. For instance, if a user has viewed animated Disney films, the alternative model suggests similar Disney animations, rather than recommending unrelated movies like Star Wars or Indiana Jones. Another limitation of the current model is its memory inefficiency. We must generate an 'anti-set', which is significantly larger than the original dataset. For example, if four users each watched a unique film, the original dataset comprises 4 rows. However, the anti-set balloons to 12 rows (derived from 16 possible combinations minus the 4 existing combinations) due to all unwatched movie combinations for each user.