Learning

For our movie recommender model, we used collaborative filtering and other filtering techniques to produce the recommendation.
Following is a brief description of the learning methods involved:

Collaborative filtering

Collaborative filtering is a widely used technique in recommender systems based on the idea that the users who previously liked the same movies in the past are more likely to like the same movies in the future. It estimates the similarities between the users and uses it to produce the prediction. To compute the similarities, we use the k-nearest neighbors algorithm with cosine similarity function. We chose this technique because it is very popular for recommendation systems and allows us to have a good starting point that we can improve on later. To estimate how much a user liked a movie, we use the ratings given by that user.

Cold start

To improve our algorithm, we decided to reduce the cold start problem. The cold start problem is one of the biggest problems of collaborative filtering: when a user has not yet given enough ratings, the collaborative filter does not have enough data to find similar users. In case there is not enough data on a user, we decided to use their demographic data to find similar users and also the IMDB rating of the movies to recommend the movies liked by everyone based on matching priority. For now, the only demographic info taken in account by our model is gender.

Adult movie filter

We also put an adult movie filter that avoids recommending adult movies to kids.

Implementation

For our implementation, we utilized the Python library Pandas to manage the databases, and we employed Surprise for the machine learning component. Our model's primary function is to estimate movie ratings for those films a user has not yet rated. We then select the top 20 highest-rated movies, sorted in descending order.

In the case of adult-oriented movies and users under the age of 18, the predicted rating for such movies is set to 0. If we possess sufficient data to employ collaborative filtering, we rely on its predictions. In situations where collaborative filtering data is insufficient, our predictions are derived from a combination of the IMDb rating and the average ratings given by users of the same gender for the respective movies.

You can access our model through the following link:
https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/tree/development/app/model

Limitations

Due to the substantial user base, collaborative filtering necessitates an extensive volume of ratings to yield meaningful results. The process of gathering data from the stream is time-consuming, and as a result, we faced limitations in accumulating a sufficient amount of data to make our model truly effective. This situation has led to the "cold start problem" affecting the majority of users, as we currently lack the requisite data for them.

Our solution moving forward entails planning to retrain our model once we have amassed a more substantial dataset. This proactive approach will help mitigate the cold start problem and enhance the overall effectiveness of our model.

Inference service

Implementation of the recommendation service

The training weights were exported to the model file which can be used for making predictions in response to incoming requests from the simulator. We placed the model, app and its dependencies in a docker container which interfaces to port 8082 of the VM in McGill infrastructure.

Design decisions

Since we did not want to retrain the model for each request, we are calling the training function when the docker container is started and it is mounted to persist across rebuilds. With the model loaded, we are calling the prediction on it after loading supporting data from the csv which is added in our container so that we do not have to query the simulator for user or movie attributes to provide as input to our model. To maintain the principle of separation of concerns, we kept the model and the prediction service in two distinct modules.

We wrapped the call to model in the prediction service and exposed it as the API endpoint for the get requests from the simulator. We would return the list of movies to the simulator. The list of movies is sorted by the rating score when provided to the prediction service as the output from the model.

Architecture

The simulator hits the following endpoint 'recommend/' running on our team's machine (fall2023-comp585-4.cs.mcgill.ca) with 'GET' requests which then routes it to prediction service for handling. This was implemented in flask. To further reduce the load in terms of data loading on the container, we made a user_details fetcher in the request handler which would send back a query to the simulator on the user_api to fetch attributes (age, occupation and gender). As we had already scrapped the whole user data set (1 million users) by the time we submitted M1, we did not end up using back_queries for now. The predictions are returned to the handler service from the model (model's working described in learning.md). We return the predicted list (which is already sorted based on the rating's ranking) as a string to the simulator for which we see the success response in the kafka logs. We found that a list object (or) dictionary object is not interpreted as a successful response by the simulator.

Load and infrastructure considerations

We deliberated on the following system design related issues which may arise due to scaling the requests. For instance, a load balancer could be exposed to the simulator API consisting of a buffer which will redirect the requests to multiple containers. Each container will consist of the model prediction service. Further, a performance argument could be made against the redundancy of maintaining multiple copies of the same model across different containers to ensure load balancing. This could be problematic while incorporating model updates and testing different model versions. Lastly, if we have huge datasets to be used for training, deploying the same datasets on multiple containers to train multiple models for load sharing can be further optimized. A fine tuned approach could be to split the dataset across containers so that training is distributed and we can combine the weights in a single container to form the model which can live on a separate container. We can use caching to serve the model weights or apply concepts similar to Content Delivery Networks to ensure fast servicing of prediction requests from multiple containers to which our load balancer is redirecting the requests. It is to be noted that we did not implement their resolutions considering the deadline for M1.

Links to commits for the same:

Integrate the inference service (link)

Add inference support (link)

Refactor flask service (link)

Dockerization of our inference service

We have placed the model file and the app driver code in the same container. Port (8082) inside the container is mapped to the port (8082) of the host machine (fall2023-comp585-4.cs.mcgill.ca) such that the requests received at the VM are forwarded to container's port when the container is deployed. We tried using the alpine version of the Python image to keep our container lightweight, but we found that packages end up with dependency errors. So we decided to use the Python base image (though larger) for the moment due to time constraints. We will work on optimizing the Docker container in the future. We set the working directory as app (base directory) in the container where we place the model and data directories. When we launch the container, we run the app.py to start our inference service. When started for the first time, we launch the training on the container for model creation to avoid packaging the model (2GB in size) within the containers as it would slow down the deployment.

CI/CD pipeline

Though not part of milestone 1, we decided to leverage GitLab's CI/CD capabilities to set up a pipeline in the early stages of development. This is an important step in automating the development lifecycle. A pipeline was set up to automatically deploy the inference service onto a Docker container running on our remote server whenever code is merged into the main branch. A GitLab runner was set up on one of our teammate's local machine in order to run the CI/CD jobs. The CI/CD job itself used an Ubuntu Docker container to SSH into our remote server using a private key generated on the server. The private key was converted to base64 format and added as a masked GitLab CI/CD variable, which means that the variable is hidden in job logs. This is vital from a security standpoint. The job pulls the latest code from the GitLab repository, builds the Docker image and deploys the container. The CI/CD pipeline simplifies the process of deployment, precluding the need for manual intervention during the deployment stage.

Links to commits for the same:

Docker setup (link)

CI/CD setup (link)

Testing CI/CD (link)

Team Process and Meeting Notes

Team Organization

Our team was organized with an intention to contribute activly and equally. We set up the roles in our workflow (A0) and assigned the team members based on individual strengths and experience, ranging from data engineering, backend development to machine learning. This effective categorization in roles helped us to better collaborate and ensuring project completion.

Communication Channels

Our team mostly used Slack and WhatsApp for day-to-day communication and updates. two weekly meetings were scheduled after classes, with both in-person and virtual Zoom sessions. To maintain documentation, GitLab was our platform of choice.

Work Division & Responsibilities

As from our meeting, the work was divided as:

Yaoqiang (Luke):
Data Engineer responsible for data creation and preprocessing.

Tamara: ML Engineers in charge of implementing the machine learning model.

Rishabh: ML Engineer in charge of implementing the machine learning model and collecting & pre-processing the datasets

Aayush: Backend Engineer tasked with developing the inference service.

Varun: MLOps Engineer, overlooking deployment and CI/CD

Our meetings served as a platform to discuss progress, brainstorm ideas, set up ToDos, and divide work for the coming week. We made discussions on model type, data creation, and task distribution for a each weeks delivery and assiged the tasks to the corresponding team member. In terms of individual contributions, Luke took charge of dataset creation, while Tamara and Rishabh began implementing the ML models. Aayush dived deep into backend development, and Varun focused on deployment essentials.

Links to our meeting notes

https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/tree/development/meeting-notes

Links to our team member commits

Data collection and preprocessing(Yaoqiang):
https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/commit/bf82820d1affab62a0677fb71e49774f738ef5f1

Varun: Dockerfile setup:
https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/commit/fdf0481ed0bbdb3ed729a817ced950fc713c5047 CI/CD setup:
https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/commit/8fbe0ff3d371123cc565c335b7cb49c1c6261d50 Documentation:
https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/commit/79f5136fab86fcd14e0b49b9884c904052103170

Inference service(Aayush):
Create inference service, help with deployment: here (link) here (link)
Ratings and Users - fetch and dataset creation: here (link)
Documentation and restructuring of code: here (link) here (link)

Model implementation(Tamara & Rishabh):

Rishabh:
Collaborative filter model:
https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/commit/be9a020284ce5d361e44da83cbe26beeeab1bb62
Data fetching and pre-processing scripts:
https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/commit/a4d64024d42ddc13dc4405cefc2d8e7458d40f30
Compiled datasets:
https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/commit/b07ff1b98e6c50dfdd10cf72aa6850db6bdff545

Tamara:
Collaborative filter:
https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/commit/d1f1596cb2e8ad67773bbbd1cc62f12afeadc956
Other filters:
https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/commit/2da2427553fface20fc5ab54fd375ddecf4336fd
Documentation, cleaning and re-structuration of the code:

https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/commit/f1c5309b01b8b125c9b4592423c64c742233b87e

Comparison to our original workflow

Compared to the original workflow, every team member performed the allotted tasks on time and in a collaborative manner. Regular meetings were held amongst group members and the work was done smoothly.

Frequency of meetings

- Initially we planned on meeting once a week but as the development progressed we had to meet at least twice in a week as the deadline for M1 approached.
- Each team member was present in the meetings either virtually or in person

What are the responsibilities of each team member during the meetings?

- We synced up on development, discussed any issues being faced by teammates and resolved them in each meeting.
- Notes and meeting minutes were created by almost each team member in the meetings
- Overall, each team member delivered on their responsibilities

How will internal deadlines be set, managed, and followed?

- We used GitLab issues to assign tasks to each team member and used the `Due date` parameter to manage deadlines. Additionally we created labels for each functionality.
- For some tasks like implementation of the model and dataset collection, it was taking some time in researching and coming up with the most optimal recommender approach to be used. During this time, one of our team members fell sick so we had to adjust the deadlines. In the end we were able to achieve the allotted tasks despite the unfortunate delay

How will you coordinate your work?

- As explained above, we used GitLab for syncing the code, allotting tasks and managing deadlines.
- We also used our team server to share large files like the data files which were used in training the model

What will be done in case disagreement arises?

Fortunately, we didn’t have any major disagreements since we communicated & collaborated effectively.

Modes of communication?

We used the following modes of communication as was discussed in our original workflow:

- In person meetings
- GitLab issues for dividing work
- Slack and WhatsApp groups for easy communication
- Zoom meetings and voice calls

Conclusion

While our orignial workflow was mostly outlined in our initial workflow, a few adaptions were made in response to some challenges we faced. For instance, we initially underestimated the complexity of model deployment, leading us to revisit and refine our approach. Additionally, We had believed that the complexity of our data wouldn't necessarily to be a large dataset. However, as we delved deeper into the implementation phase, it became an evidence that It is important to get a larger dataset. So we spent a bit more time on collecting and generating the dataset.