Final report
Team 3

Conceptual Analysis of Potential Problems

Feedback Loops
The two most significant feedback loops to consider:

1. User Explicit/Implicit Feedback Loop: In this feedback loop, the explicit data (user ratings) and implicit data (watched movies) collected by the API are used to train the recommender system model, which in turn provides recommendations to the user. This loop can cause the model to become overfitted and provide inaccurate recommendations, leading to a poorer user experience. This can be detected by monitoring the accuracy of the model over time and by comparing the model's predictions to actual user ratings.

2. Monitoring Feedback Loop: The monitoring system provides API feedback in this feedback loop. For example, if the API is not responding quickly enough, or if there are spikes of errors occasionally, the monitoring system can alert the developers, who can further investigate the root cause of the issue. This can result in, for instance, finding bottlenecks in the system, such as hardware limitations, bugs that cause excessive use of resources, etc. The monitoring system can be part of this experimentation and debugging pipeline. Changes can be rolled out to see how the system responds and then iteratively improve the metrics observed. The monitoring feedback loop also extends to online evaluation of the model, where new model deployments can be observed in real-time to understand how they perform against older versions.

Fairness
There are two potential fairness issues to consider:

1. Movie-based Fairness: In this case, the API may provide different recommendations based on the type of movie, such as genre. This can be detected by monitoring the API's recommendations for different types of movies and comparing them for any disparities. This issue can be reduced by carefully selecting the data used to train the model, ensuring it is representative of all movie types, and using fairness-aware machine learning algorithms.

2. User-based Fairness: In this fairness issue, the API may provide different recommendations to users from different demographic groups (age, occupation, etc.) since our training data does not guarantee fair representation from each demographic. This may lead to some users being provided with higher-quality recommendations than others, which can lead to a poorer experience for some users. This can be detected by monitoring the accuracy of the recommendations in different demographic groups and comparing them for any disparities. It can be reduced by ensuring that the Recommender System model considers demographic data when providing recommendations. This can be achieved by carefully selecting the data used to train the model, ensuring it is representative of the entire user base, and using fairness-aware machine learning algorithms.

In both cases, adding the fairness metrics would introduce complexity in the monitoring pipelines as well as complexity in the training data processing/validation pipelines, both of which would require significant development time and resources.

Attacks

There are four potential attacks to consider:

1. Attacks through Kafka: In this exploit, an attacker can attempt to spoof messages in the Kafka stream that our system observes. Since the entire foundation of our recommender system is built on the trust that the Kafka stream is the one source of truth, any success in being able to tamper with the Kafka stream will severely harm our model. Using this vector, an attacker can effectively hide errors from parts of our monitoring stacks and even tamper with the training data that we collect from the Kafka stream for training future versions of the model. Although detecting attacks like this is difficult, having multiple ways to monitor the health of the API and having concrete metrics to determine the validity of data used in model training can temporarily reduce short-term harmful effects. It can be mitigated by using encrypted protocols to access the Kafka stream, having multiple fallback Kafka streams, protecting the Kafka streaming server, and protecting credentials to access the server, just to name a few. If detected, immediate steps should be taken to ensure access to the Kafka stream is secured, and messages are not tampered with.

2. Data Leakage Attack: In this attack, the attacker attempts to gain access to the user data stored by the API. Our system is especially vulnerable to this type of exploit since any computer within the VPN can access our recommender and make requests for any user any number of times. This can effectively allow an attacker to reverse engineer our recommender system with enough time, even uncovering user data used to train the model leading to severe privacy breaches. This can be detected by monitoring the API's access logs and by monitoring the API's network traffic (especially the source and nature of the incoming traffic). It can be mitigated by using encryption and access control lists to restrict access to user data. If detected, the attacker's IP address can be blocked.

3. Denial of Service (DoS) Attack: In this attack, the attacker attempts to overwhelm the API with requests, leading to a decrease in the performance and availability of the API. This is valid since access to our recommender API is currently open within the VPN and not restricted (e.g., authenticated users only, rate limits, etc.). Attacks like these can be detected by monitoring the number of requests made to the API over time and by monitoring the response time of the API. It can be mitigated by using rate-limiting and setting up a fallback service. If detected, the attacker's IP address can be blocked.

4. Man-in-the-Middle (MITM) Attack: In this attack, an attacker attempts to intercept traffic between the API and its users. Since our API responses are only served through HTTP protocol, attempting this attack on our system is possible. This can be detected by monitoring the API's logs and any unusual pattern of requests from specific sources. This attack can be

mitigated by using HTTPS, ensuring that any data sent to and from the API is encrypted. Another way to detect a MITM attack is to use authentication and authorization mechanisms. By requiring users to provide proof of identity before accessing sensitive information or resources, it becomes more difficult for an attacker to impersonate a legitimate user and gain access to the system. If an attacker attempts to access the system without providing the appropriate authentication credentials, their attempt will be detected, and the system can take appropriate action. Once detected, the attacker's IP address can be blocked.

In addition to the four main types of attacks mentioned above, we can still imagine that in a highly popular commercial recommendation system involving a very financially powerful industry such as the movie industry, it is important to be prepared for attacks related to money and human aspects with the objective, for example, to obtain direct or indirect access to the system and inject biased data or even modify model parameters that may influence users to consume certain movies based on recommendations.

Analysis of Problems in Log Data

A consistent user feedback loop plays an important role in the continuous evolution of our recommendation system. This section will discuss how we analyzed the data distribution issue in training datasets for user feedback loops and how we avoid the model from overfitting in terms of telemetry data.

User-based Fairness

Initially, our model was trained with a large dataset. Then in the subsequent feedback loops, the model was re-trained with smaller datasets which consist of newly-collected data from the latest Kafka logs. Therefore, it is important to consistently guarantee that the model is always trained and re-trained (during feedback loops). To be more specific, we want the data distribution in training datasets to be consistent with each other. Moreover, we do not want the model to overfit on a specific dataset either. In this project, we considered gender as the focal point of the demographic we specifically analyzed. This is because the gender attribute is highly complete for users (i.e., all users have gender attributes) and is also the easiest one to analyze. Moreover, it has a considerably direct impact on our society.

With that goal in mind, we first applied an iterative approach to grab one million user data through the user API and store it in a CSV file. Then we did a quick data analysis with respect to gender. To guarantee fairness in each subsequent feedback loop, within our data feedback pipeline script, we added a function that analyzes the data with respect to gender and stores it as metadata when we store the preprocessed data as artifacts on weights and biases. To prevent the model from overfitting, each time the model feedback pipeline is triggered, it will graph the training losses and the training accuracy for that feedback action and put it on the pull request on GitHub. To resolve such issues, each time a feedback action is completed, it requires one of the team members to manually review the metadata mentioned previously and the training accuracy and lost graph, then decide whether there are any fairness concerns or overfitting issues for the model produced in that feedback loop.

From the report (see the Pointers to the Artifacts subsection) generated below for the two Feedback automatic updates and the report generated for the one million user data, we could clearly see that the metadata distribution aligns well with the true distribution of the user data with respect to gender. As for the true distribution, there are around 83% males and 17% females. During the feedback loops, the dataset comprises 82% males and 18% females for the first model updates after the data preprocessing. At the same time, the dataset consists of 83% males and 17 % females for the second model updates. The difference between the true distribution of data and each feedback loop's distribution of data is sufficiently neglectable. This is a big factor of why both models were merged into the main branch and later deployed after manual reviews since it does not contain any user-based fairness issue with respect to gender, as mentioned above.

Movie-based Fairness

Besides the user-based data distribution mentioned above, we also considered movie-based data distribution due to some special aspects of our model implementation. We designed the model only to recommend movies from the candidate list. Initially, many movies were collected, which we named raw movies. We processed this data and acquired movies with ratings above the threshold, which we named candidate movies.

We selected the movie category as the analysis topic and considered raw movies as the true distribution of movie data. The categorical distribution of raw movies is shown in Figure 1. As mentioned above, we processed raw movies into candidate movies. During this process, we want to maintain a similar categorical distribution. If the categorical distribution changes dramatically, it could indicate man-made unfairness. Fortunately, as shown in Figure 2, the candidate movies demonstrate similar categorical distribution.

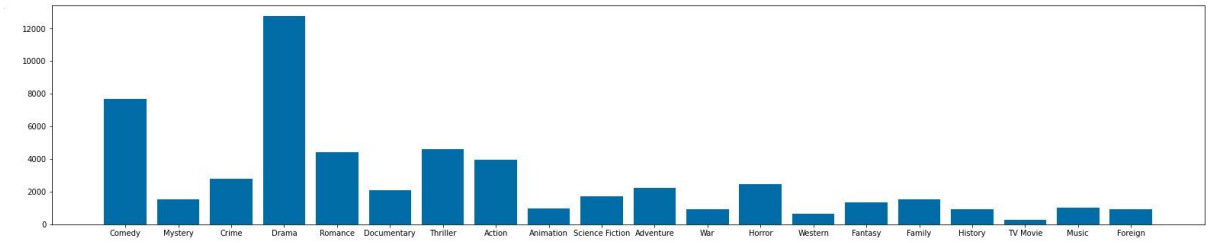

Figure 1: Categorical Distribution for Raw Movies

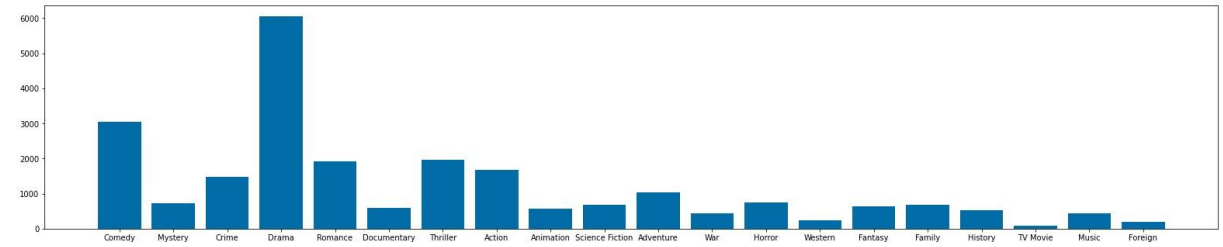

Figure 2: Categorical Distribution for Candidate Movies

Pipeline Overfitting

There is also a discrepancy in the size of each training dataset. The dataset for the initial training is considerably large, but the datasets created for feedback loops are relatively small in comparison. Therefore, it is also a concern whether the model would be overfitting to the new and smaller dataset during each feedback loop. We took this concern into consideration during the development phase and only used 5 epochs to retrain the model with the new dataset for each feedback loop. Meanwhile, many metrics (e.g., loss, top-5 categorical accuracy) were used and displayed in the Weight and Bias report to help developers determine if the model was overfitting to the dataset. In the worst case that the model overfitted to a certain dataset, we could roll back to the previous version with provenance.

Pointers to the Artifacts:

The whole one million user data can be found within our repo from this link: https://github.com/COMP585Fall2022/Team-3/blob/main/Datasets/data_user.csv and the data analysis report can be found here: https://wandb.ai/team-3-comp585/milestone3/reports/User-Data-Analysis--VmlldzozMTA0ODkz

All user Data Analysis Report: https://wandb.ai/team-3-comp585/milestone3/reports/User-Data-Analysis--VmlldzozMTA0ODkz

Github Code Link for all user data analysis: https://github.com/COMP585Fall2022/Team-3/blob/main/Model%20Code%20and%20Tuning/Data_Analysis.ipynb

First Feedback loop automatic Update Report: https://wandb.ai/team-3-comp585/milestone3/reports/First-Automatic-Model-Update-Report--VmlldzozMTA0NzU0

Second Feedback loop automatic Update Report: https://wandb.ai/team-3-comp585/milestone3/reports/Second-Automatic-Model-Update-Report--VmlldzozMTA0ODQ3

Reflection on Recommendation Service

The system, in its current form, has proved to perform reasonably well under the current load based on the metrics we have been observing, which include the Kafka logs for errors, CPU/RAM usage, average response time, etc.

One of the most challenging aspects of this project was designing and implementing the recommendation service with the TensorFlow recommender system model. It required a deep understanding of machine learning algorithms and methods to properly design, test, and deploy the model, especially given the hardware and software constraints we were under. Moreover, our choice of using TensorFlow for the recommender introduced compatibility issues when we deployed the model on the team server due to how old the CPU on the team server was. The latest version of TensorFlow that the team tested and built everything on locally did not work on the team server's Xeon-based CPU. We had to resort to using virtualized Conda environments inside Docker to run the model, which ultimately affected the performance and is often still unstable under peak loads.

Another challenging aspect of this project was deploying the recommendation service on the server with Docker-compose with all the additional services. It was challenging to ensure that all of the services were configured correctly and integrated with one another, particularly the canary releases and Traefik load balancing. Since our CI testing pipeline cannot easily catch integration issues spanning across several services simultaneously (e.g., simulating container health issues to test how the load balancer reacts), we had to test changes manually every time.

At scale in production, the recommendation service would be unstable and require additional investment to ensure its stability. This could include investing in more powerful servers with GPUs, increasing the number of servers, and using a more robust monitoring system to detect and address any issues. Additionally, more testing would need to be done to ensure the accuracy of the model's recommendations and any metrics that the additional services may be collecting.

With better hardware, we can invest time in re-introducing the ranking system we had to remove from our current model for better response times [*Context: To make the responses fast enough in our current model given the limited hardware, we took out a "ranking" component from generating our recommendations and are only relying on a "retrieval" component to generate recommendations (see milestone 2 report, "Overview" section for details)*]. Re-introducing the ranking model is expected to improve the order of the recommended movies significantly.

Additionally, with more time and resources, we would invest in more powerful, distributed servers to ensure that the recommendation service could handle an increased load. Additionally, we would invest in additional monitoring and testing tools to ensure the accuracy of the model's recommendations and the accuracy of the metrics collected by the additional services. Finally, we would invest in more robust load balancing, such as using Kubernetes, to ensure that the service could handle an increased load.

Reflection on Teamwork

Overall, our team worked well throughout the project. We delegated tasks among ourselves in an organized and efficient manner, and each team member was able to complete their tasks promptly. Everyone had their own unique skill sets, which we were able to utilize to our advantage. Given the complexity of the technologies we were dealing with, several of which we had no prior experience in, it is refreshing to see how the team did not shy away from taking the opportunity to learn these technologies and apply them efficiently in the project.

The team was divided into “the ML team” and “the software engineering team.” While it worked well overall, sometimes it resulted in an unbalanced workload as the milestones progressed because more activities were software engineering related. Additionally, it could have been helpful to create a schedule for each milestone that considers the expected duration for each task and identifies critical paths. At times, we had some resources waiting for the completion of another task, and if it was noticed during the planning stage, an action plan could be put into practice to improve the flow of activities.

One of the main challenges we faced was communication and collaboration. We had a few instances where tasks weren't communicated properly, and it caused delays. In the future, we should ensure everyone is on the same page and clearly understands their tasks.

Another challenge we faced was technical difficulties. We had a few instances where our code was not working correctly, and it took us a while to figure out the issues. For example, when we ran out of credits on GitHub Actions to run our testing pipelines, several code changes that broke our unit tests remained undetected and unfixed for days. This introduced severe, long-lasting instability issues when the code was released in production. In the future, we should thoroughly test our code before pushing it to production.

Overall, I think our team worked well together and learned much from each other. With better communication and thorough testing, our future collaborations will be even better.

Individual Contributions

Each team member had a specific responsibility based on skill sets and was given a corresponding set of tasks to complete. We kept track of our progress using Github. We held regular team meetings (both in-person and online) to discuss the progress of the project and our meeting notes documented the details of these discussions.

Shanzid Shaiham	Primarily in charge of building out the infrastructure of the whole system. This included setting up containers/services in Docker, setting up monitoring/alerting services, and ensuring system reliability. Contributions also spanned beyond just technical, into providing inputs for system architecture across several sub-components (e.g., Kafka monitor, CI/CD pipelines, etc.), understanding system requirements, discovering tools and processes that can be used to meet the requirements, delegating tasks to team members, and creating plans to ensure the successful completion of the milestones. Technical contributions are tracked using pull requests on GitHub: 1. https://github.com/COMP585Fall2022/Team-3/pulls?q=author:Shanzid01+ 2. https://github.com/Shanzid01/Team-3/pulls?q=author:Shanzid01+
Marcos Souto Jr.	Recommendation model reformulation using a more modern technology that would bring more flexibility and performance and allow the use of more features related to users and movies to achieve better accuracy. Also involved in the interface with the platform responsible for tracking the models and datasets, in addition to the inference and training pipeline. Technical contributions are tracked using pull requests on GitHub: 1. https://github.com/COMP585Fall2022/Team-3/pulls?q=is%3Apr+author%3Amsoutojr+ 2. https://github.com/Shanzid01/Team-3/pulls?q=author%3Amsoutojr
Kua Chen	Participated in software infrastructure construction. Focused on handling Kafka logs, including data preprocessing, data collection, and building the pipeline for the database. In the

	later development phase, put more effort into testing. Developed and implemented unit tests to ensure the correctness of functions. Technical contributions are tracked using pull requests on GitHub: 1. https://github.com/COMP585Fall2022/Team-3/pulls?q=author%3AChenKua+ 2. https://github.com/Shanzid01/Team-3/pulls?q=author%3AChenKua+
Barry Li	Took parts in both software infrastructure and the model development. Some key contributions are: 1. Built the very first Pyspark ALS model 2. helped develop by trying to add more features to the TensorFlow model 3. data preprocessing, data analysis and its corresponding feedback pipeline script, 4. modularization of the TensorFlow code for CI pipeline 5. feedback workflow 6. Model unit testing Technical contributions are tracked using pull requests on GitHub: 1. https://github.com/COMP585Fall2022/Team-3/pulls?q=author%3AWeienLi+ 2. https://github.com/Shanzid01/Team-3/pulls?q=author%3AWeienLi+ And two commit of model notebook without pull request: https://github.com/COMP585Fall2022/Team-3/blob/main/Model%20Code%20and%20Tuning/Second_Version_Tensorflow_RS_v4.ipynb https://github.com/COMP585Fall2022/Team-3/blob/main/Model%20Code%20and%20Tuning/Pyspark_ALS.ipynb

Meeting Notes

Milestone 1:
https://github.com/COMP585Fall2022/Team-3/blob/main/Reports/Milestone%201/Milestone%201.pdf
Milestone 2:
https://github.com/COMP585Fall2022/Team-3/blob/main/Reports/Milestone%202/Milestone%202.pdf
Milestone 3:
https://github.com/COMP585Fall2022/Team-3/blob/main/Reports/Milestone%203/Milestone%203.pdf