Updated prior contributions in M3-report.md
Rishabh Thaney authored just now

⊘   18698339

M↓   **M3-report.md** 51.46 KiB

Milestone 3

Containerization

We have four running containers for our inference service at all times: two containers for the stable deployment, one for the canary deployment and one for the load balancer. We additionally have containers running Prometheus and cadvisor as part of our monitoring service.

Container set up

Flask application

The containers for the stable and canary deployment of our application are built using the same Dockerfile with configurable options. They use a lightweight Python alpine image as the base image to minimize the size of the resulting image. Extra system dependencies needed to run the `surprise` library [1] for our recommendation service are installed onto the base image. Our container set up packs both the model and the inference service within the same container. Gunicorn is the web server used to deploy the Flask application. Gunicorn is a Python WSGI HTTP server ideal for deploying Python web applications in production [2].

Load balancer

We use NGINX as our load balancer [3] to distribute the load between the stable and canary deployments. We again use an alpine version of the NGINX image to reduce the container size. Our custom NGINX configuration is mounted as a volume onto the container.

Orchestration and automatic container creation in CI/CD pipeline

We use docker compose to orchestrate the deployment of our containers. For the initial launch, all our containers can be created simply by running a single command in the `app` directory: `docker compose up -d`. During subsequent launches, a bash script in our CI/CD pipeline handles the creation and replacement of containers. Docker compose is used to take down or relaunch the required services. Containers are automatically created as part of our canary release pipeline. This process is explained in detail in the Releases section.

Automated model updates

We implemented the automatic training and deployment of the models by developing a python script (`auto_deployment/auto_deploy.py`) which broadly performs the following tasks:

1. Data collection and pre-processing from the Kafka stream:
The raw data is collected from the Kafka stream for 15 minutes and pre-processed. After this it is appended to the records previously collected in `data/clean_rating.csv`.

2. Training the models with the new data:
The newly collected data is now used to train the model \

3. Checking if RMSE score is < 1:
We do our offline evaluation i.e. checking if RMSE score is less than 1 after the model has successfully been trained as described previously. If this condition is not satisfied, the new model doesn't get deployed and the user gets an email mentioning that the new model wasn't deployed and also reports the RMSE score.

4. Versions the new model using DVC:
If the offline evaluation metric is satisfied, the new model gets versioned by DVC.

5. Pushes the newly collected data to GitLab:
Finally, if everything works fine till this point, the newly collected data as well as the version is pushed to GitLab. This triggers the automated testing pipeline and also deploys the new model.

6. Users get notified via an email:
After the successful deployment of the model the users get an automated email confirming that the deployment was successful.

Additionally, we developed a Python scheduler script (`auto_deployment/scheduler.py`) which runs in the background on our team-4 server and triggers the auto deployment pipeline every 2 days.

Links to the scripts involved:

- Auto deployment script: https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/blob/main/app/auto_deployment/auto_deploy.py
- Scheduler: https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/blob/main/app/auto_deployment/scheduler.py

Releases

Triggering releases

As described in the previous section, when the new data is periodically pushed to our GitLab repository, the pipeline is triggered which passes through data quality checks (test-data) and tests for the model (test-model) and inference service (test-app). If the tests run successfully, the final job of the pipeline `deploy` establishes an SSH connection to our server and pulls the latest version of our repository on the server. It then launches a background script to perform the canary release.

Monitoring the release

The release script performs the following actions: (1) It kills the current canary container, builds a new image containing the new model and recreates the canary container. (2) It waits for 12 hours to allow the new deployment to stabilize and receive a fair amount of requests (3) After 12 hours, it sends a curl request to the Prometheus HTTP API to fetch the average response time of the successful requests over the past 12 hours. (4) Our threshold response time is 500ms. If the average response time is below the threshold, it builds a new image with the `stable` tag and recreates the two stable containers. At this point our canary release is successfully completed. If the average response time is greater than the threshold, the canary release is aborted by removing the canary container and sending an email notification to our team informing of the failed release.

After the canary release is complete, we retain the canary container along with the newly deployed stable containers as this will increase availability of our service. This means that all three containers serve the stable deployment at the time. Though this may not be an ideal practice in production, we chose to follow this approach as it allows us to serve a greater number of successful requests.

Load balancing

Our NGINX load balancer is configured to execute a 80-20 split of the incoming traffic between the two containers of the stable deployment and the canary container respectively.

```
upstream backend {
        server inference_stable:5000 weight=4;
        server inference_canary:5001 weight=1;
    }
```

NGINX automatically distributes the load to available services in case any service is unavailable. For instance, when the canary container is recreated to deploy a new model, all traffic will be routed to the stable service when the container is being created. Similarly, all traffic will be momentarily directed to the canary container when the stable containers are being recreated to finish the new release. In any case, the downtime while recreating the containers is negligible, however, our load balancer ensures that we have zero downtime.
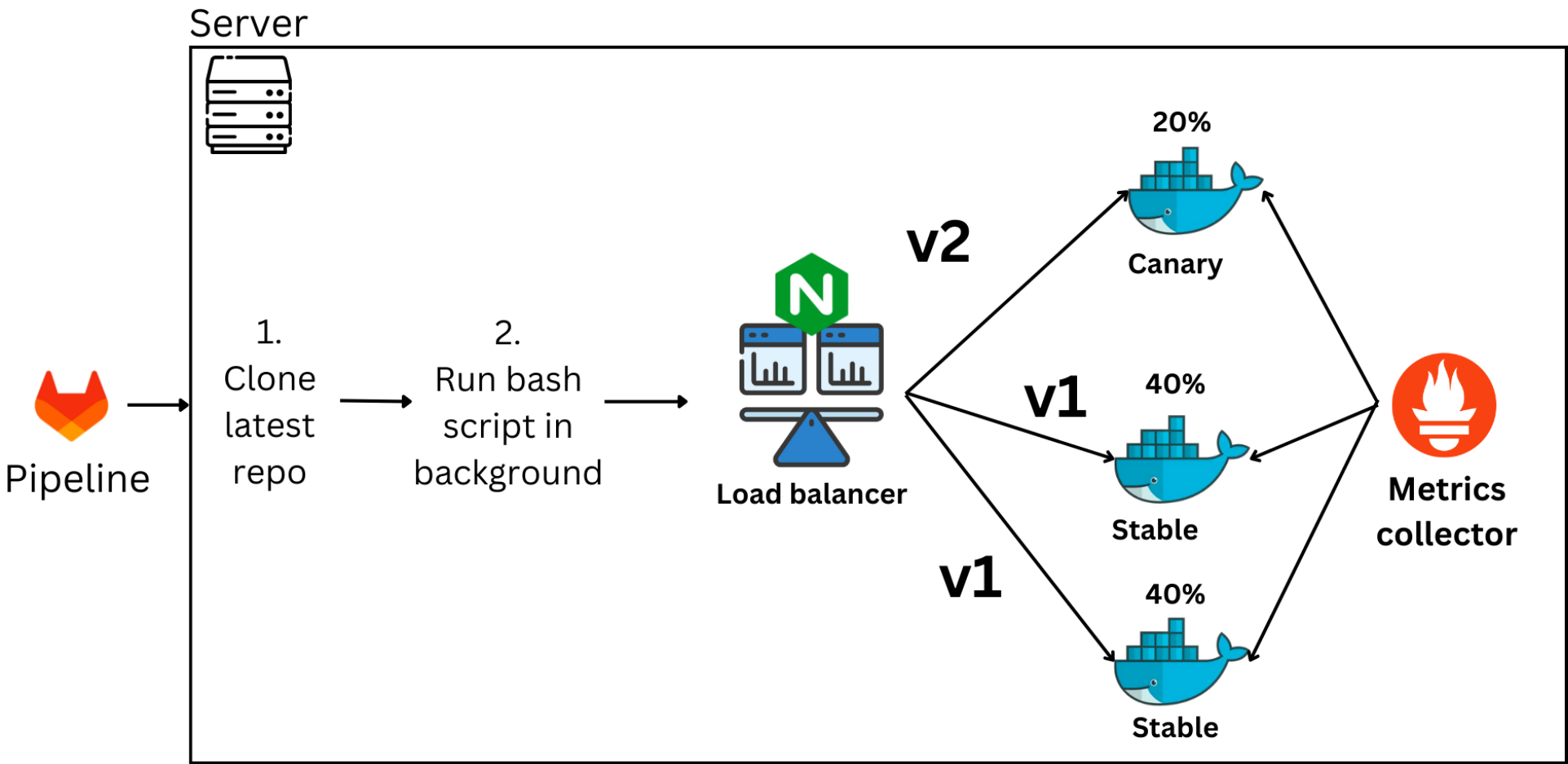

Figure 1: Workflow of canary release

Metrics

Prometheus was set up to track metrics for our Flask application using `prometheus-flask-exporter` [4]. This is a Python package that needs to be installed in our Flask application (check app.py). It exports some default metrics like the response time for each request and the number of successful and failed requests.

```
from prometheus_flask_exporter import PrometheusMetrics

app = Flask(__name__)
metrics = PrometheusMetrics(app)
```

To determine whether to perform the canary release, we perform the following PromQL query which computes the average of all recommendation requests. `flask_http_request_duration_seconds_sum` is a metric that gives the time taken for each request in seconds and `lask_http_request_duration_seconds_count` records the number of requests received.

```
sum(flask_http_request_duration_seconds_sum{path=~"/recommend/.*"}) /
sum(flask_http_request_duration_seconds_count{path=~"/recommend/.*"})
```

Provenance

Our idea behind having provenance in the system was to bind the data and models to our gitlab commits. We wanted to include precise commit messages to help us track the evolution of data/ model with each iteration. As the pipeline code is already tracked based on the commits, we figured this would be a good idea. It is to be noted that uploading models to gitlab is impractical due to size issues. Also, our deployment works through the code changes on our repo and it is heavily making use of gitlab CI/CD so we wanted the benefits of versioning using gitlab for models as well.

We do the following: On the host machine (our team-4 server), we have initialized dvc in the deployed repo. We use it to create .dvc files for models. These files are metadata for the models which can be uploaded to gitlab. Multiple files can be created corresponding to different versions of the model. These files primarily consist of md5 hash and as such are small in size. But they (.dvc files) provide a way for us to link models to commits without actually committing model files. We decided it would be practical to add the model.pkl.dvc file for tracking in the same pipeline as auto-deployment. So whenever, our automated data collection script appends processed data to existing data on the host machine (outside of any container), we also initiate training of the model and include the resulting .dvc file of the trained model to the github commit. (**link to code demontrating that:** dvc tracking within automated data creation pipeline) It can be argued that the host machine does not possess resources to store different versions of model. This is because each model is roughly 2GB in size. We can offload those models to remote storage using dvc with the .dvc files acting as symbolic links. Since we did not have access to a free remote storage to host multiple models, as a proof of concept: we kept different versions of models but in the form of .dvc files along with the data files. We committed these to the gitlab repo too. This is to denote the remote offloading in real scenario. (**link to code demonstrating that**: remote offloading and versioning) This allows us to have a great versioning track record of the models and data as depicted in the highlights towards the end of this report.

Further enhancement: With the above approach, we were able to track the evolution of data and models along with the pipeline code but linking each individual recommendation with the pipeline code, model and data used to make that recommendation was difficult. We worked out the following approach:

- We created a new mysql container which would host a database on our team's server host machine. To put in perspective, this will be a separate container apart from the release containers and the host machine itself where the auto-data collection scripts were scheduled with cron job. Also, the mysql container has a persistently mounted storage. It is connected to the same docker network as the release containers and is part of docker compose.yml for releases. This was necessary to ensure database connection from within release containers to the mysql docker container. (**link to code implementation**: docker compose file)
- We created two tables in the database namely:
  - reccom (version_number INT, user_id INT);
  - tracking (version_number INT, data_creation TIMESTAMP, trained_on TIMESTAMP, model_rmse FLOAT);
- Since the mysql container had its port exposed, the host machine could insert data in the tracking table while the release containers were bound to the mysql database when the app was launched. The connection was opened and closed using app_context and tear_down methods of flask. The release containers would insert records in reccom table linking version_number and user_ids of the recommendations being served.
- Lastly, with each iteration of data training and model updation: we increment a unique identifier in our auto_deployment module known as version_number which essentially tracks and relates the changes in data to models. It is to be noted that the version.txt file having the number is part of our commit whenever auto-deployment updation happens. Now, each container would have its own copy of the version number (code link) depending on which model it is serving which in turn is dependent on the data being used to train it.
- As a result, we store the version_number, and the userID served by the container running that version of model+data along with the data creation timestamp, model creation timestamp and model_rmse score for that version_number in our database. Using a join query helps deliver the output of granular provenance per userID request. (**link to concrete example**: *we recorded a small video depicting it in action and explaining more*) **example:** Following screenshot depicts the last two rows of the result set sorted in descending order for user_ids. We could not display all the rows due to space limitations in the screenshot.

```
mysql> select A.version_number, A.user_id,
    -> B.data_creation, B.trained_on, B.model_rmse
    -> from reccom A, tracking B
    -> where A.version_number = B.version_number
    -> ORDER BY A.version_number DESC LIMIT 2 ;
+----------------+---------+---------------------+---------------------+------------+
| version_number | user_id | data_creation       | trained_on          | model_rmse |
+----------------+---------+---------------------+---------------------+------------+
|             37 |  545850 | 2023-11-30 01:27:53 | 2023-11-30 01:28:59 |   0.721723 |
|             37 |  637838 | 2023-11-30 01:27:53 | 2023-11-30 01:28:59 |   0.721723 |
+----------------+---------+---------------------+---------------------+------------+
```

- ink to mysql insertion query updates from container image: link
  - helper functions: link
- link to host insertion queries: link
  - helper functions: link

Fairness

Definition

In order to analyze the fairness of our model, it is important to define what we mean by fairness. In the scientific literature, there are numerous definitions of fairness, and a taxonomy of these definitions can be found in [this paper](). The definition of fairness that we will use in our project is based on three principles:

- Our definition is **outcome-oriented**, meaning that it considers the fairness of the outputs of our model rather than the process.
- We evaluate fairness in relation to **groups of individuals** (for example, based on their gender, age, etc.) rather than the individuals themselves.
- Our definition is based on the concept of **Consistent Fairness**, meaning that two similar groups of individuals should be treated similarly. In the context of our project, this implies that two groups of people should have a similar quality of recommended outcomes, regardless of the social group to which they belong.

Potential issue 1

A first issue that may arise is that our model does not have the same quality of recommendation depending on the social group to which an individual belongs. The quality of recommendations can be evaluated based on how close the model's predictions are to reality, so we can use the Root Mean Square Error (RMSE). To detect this problem, we can separate our test dataset into different social groups and compare the RMSE for each social group. For example, in the context of our project, we can assess differences in RMSE for each gender, age, and occupation.

Potential issue 2

A second issue that can arise in the context of recommendation systems is that two groups of individuals may not have the same variety of recommended items. To address this, we can compare the variety of movies that a social group watches and compare it with the variety of recommendations provided. Two groups of individuals with the same variety of movie genres watched should have the same variety of genres in their recommendations. To detect this problem, we can evaluate the variance of movie genres proposed in the recommendations for each social group and compare it to the variance of the movies watched by that group.

Reduce potential issues

Issues of fairness often have numerous sources. First, we will discuss one source that can lead to fairness problems: the dataset. Our recommendation system relies on the proximity between users, whether through our demographic filter grouping individuals by social groups or our collaborative filter, which is also influenced by the demographics of individuals because our tastes often align with those of people in the same social groups. Consequently, if we have too little data on a particular social group, the quality and variety of recommendations for that group may be poorer. It is crucial to have a diverse and representative dataset. Note that the dataset should not only contain sufficient data for all social groups but also for all intersections of social groups. One way to address this issue would be to collect more data on social groups with less data.

The lack of variety in our system's recommendations can also be attributed to the implementation of our model. Our model includes a demographic filter that acts solely based on a person's gender. This can lead to gender stereotyping of recommendations, especially if combined with an imbalanced dataset between the two genders, as the variety of films appreciated by one gender might be less well captured. One way to address this problem would be to consider additional demographic factors in our demographic filter.

Feedback loop

Potential issue 1 - Echo chamber

Description

The first feedback loop is called the "echo chamber". This feedback loop occurs when an item is highly recommended, leading to more users watching it. In the context of our model, this can happen because we consider the overall popularity of a movie to estimate its recommendation, and thus, movies that are generally well-liked are highly recommended.

The echo chamber can have both positive and negative consequences. The positive impact of this feedback loop is that most of the time, when a movie is liked by many users, there's a high probability that it will appeal to the majority of users. Consequently, it will appear in the recommendations of more and more users as it gains popularity.

But, if a movie becomes highly rated because it is extensively watched by a bubble of people who like that type of movie, it will then be propelled into everyone's recommendations, including those not particularly interested in that type of movie. With this movie repeatedly appearing in recommendations, even users who are not interested may eventually watch it. If it turns out not to be to their liking, they might give it a poor rating. The impact of this can vary depending on the recommendation model used: if the model primarily considers the number of people who have watched the movie, it might further boost its recommendations. However, if the model is based on ratings (like ours) or viewing time, the movie might cease to be recommended altogether, even to those who initially enjoyed that type of movie.

On the contrary, this can also lead to some movies never recommended. For instance, if a movie receives an initial poor rating, it may not appear in the recommendations for anyone and, consequently, never get watched, so never get rated again, etc.

Detection

This feedback loop can be detected by observing how the ratings of certain movies evolve based on how much they have been recommended. It's also crucial to examine which films are never recommended. Monitoring these patterns can provide insights into the presence and impact of the echo chamber feedback loop.

Mitigation

A solution to mitigate this is to place less emphasis on the popularity of a movie if the positive ratings come from users who have little resemblance (this can be assessed using the similarity matrix generated by collaborative filtering, for example), and to give it more weight if it comes from a bubble of people with similar tastes.

Another solution, especially to prevent movies from never being recommended, is to introduce a small element of randomness. This allows certain movies to have a second chance and be recommended to users, even if they haven't received high popularity or ratings initially.

Potential issue 2 - Filter bubble

Description

The second feedback loop is the "Filter Bubble." This feedback loop occurs when a user's recommendations become increasingly personalized, showing the user only a very limited and homogeneous subset of all the films on the platform.

The primary positive consequence of this feedback loop is that users will receive recommendations with a very high probability of enjoyment. However, the downside is that it confines users to a bubble where the content they consume is limited, giving them the impression that their preferences are generalizable to everyone. Moreover, these bubbles can be observed demographically and may contribute to reinforcing social biases. For example, a movie about a princess might be more recommended to little girls, thus reinforcing the social bias as it is more watched by them, even if little boys might like it too.

Detection

Analyzing the variety of films recommended to users and how it varies across user demographics can provide insights into the existence and impact of this type of feedback loop.

Mitigation

To mitigate this feedback loop, we can introduce a small element of randomization into everyone's recommendations. This helps recommend slightly more diverse types of films, breaking the homogeneity of personalized recommendations. Even though adding this random element may lead to a decrease in accuracy, it can also enable users to discover new interests. It strikes a balance between personalized recommendations and the exploration of diverse content.

Analysis of Problems in log Data

Fairness

Dataset analysis

From our data analysis, it seems that social groups are distributed similarly between the ratings dataset and the users dataset.

However, the distribution within the datasets is highly uneven between social groups themselves. As seen in the diagrams below, the dataset contains significantly more men than women, with the majority of individuals being between 24 and 35 years old, and students being much more represented than other occupations.

We have also examined intersections of social groups. For example, we observed that, on average, women are older than men, and 11% of women are academics/educators compared to 2% of men. Regarding differences in occupations between women and men, we would like to emphasize that achieving balance in this intersection of social groups can be more complex than simply having a similar representation across occupations between women and men because a person's occupation is not an independent variable from their gender, unlike age and gender, which can be considered independent.
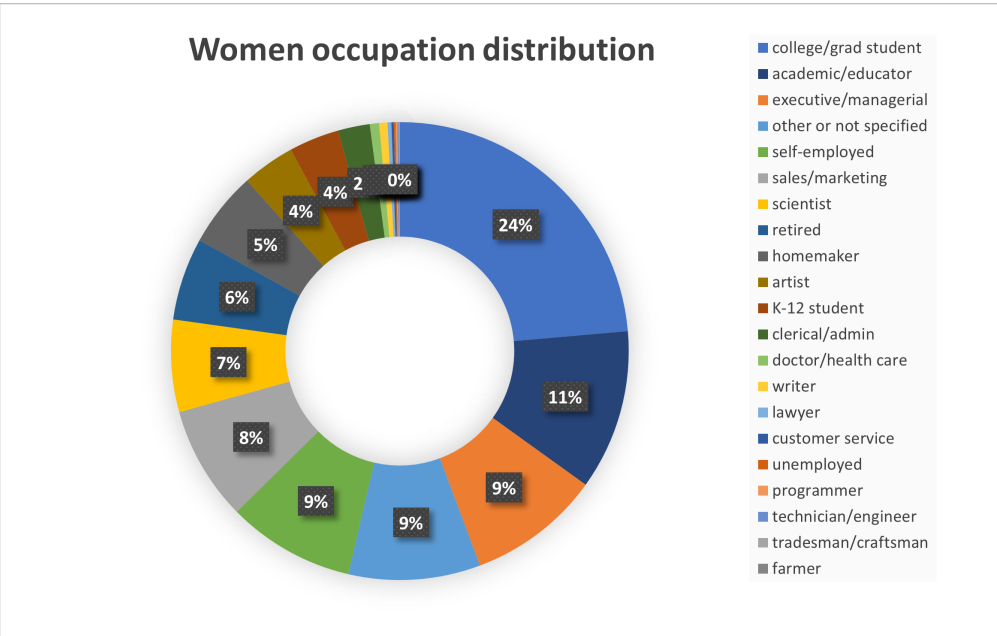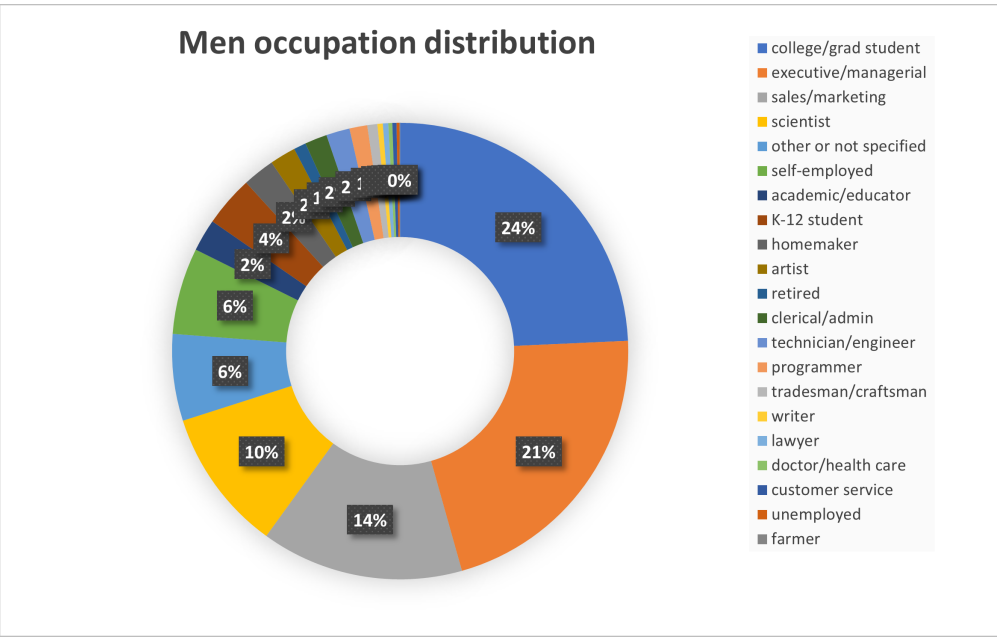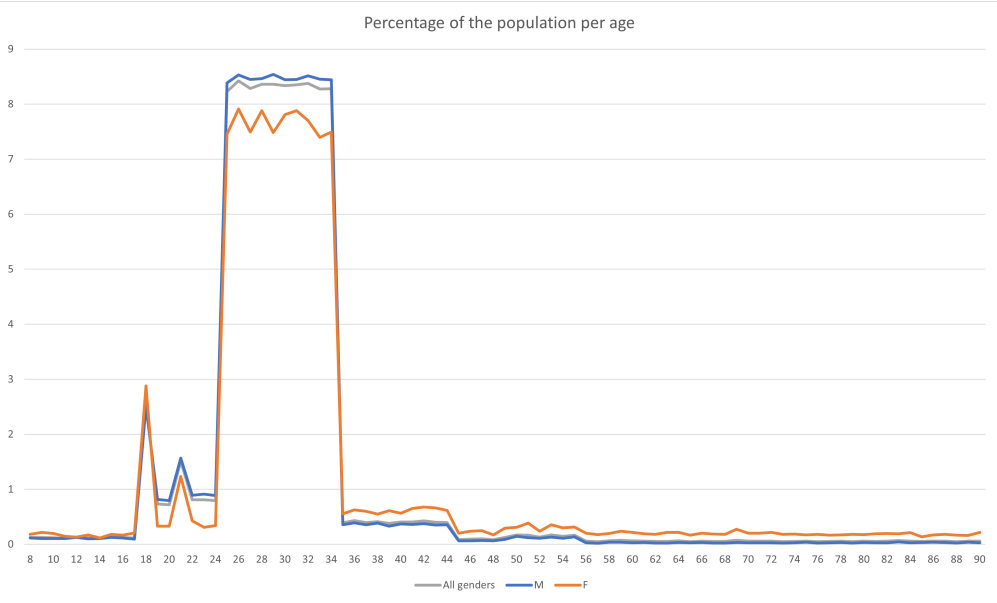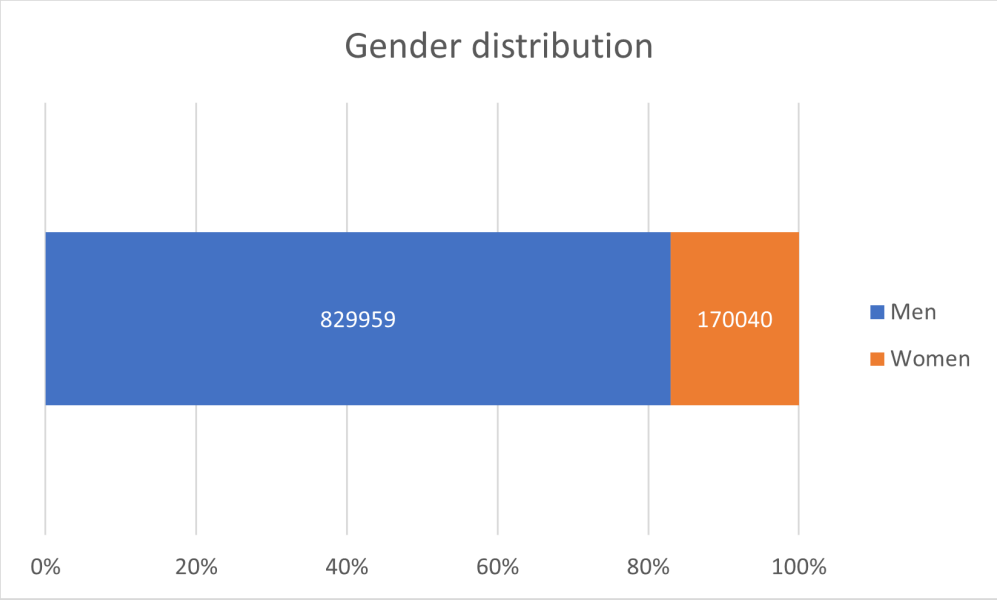
You can find all the statistics and results of our analysis here (in particular, the results of all the groups intersections), and the code to generate these results here.

Quality of recommendations

We then analyzed the differences in the quality of recommendations based on gender, age, and occupation. After training our model on the train dataset, we divided the test dataset according to the gender, age, and occupation of the users, calculating the RMSE for these sub-datasets of the test dataset.

We then displayed our results in the graphs below. We can observe that while there is only a small difference in RMSE between women and men, the disparity in RMSE is much greater when calculated based on age or occupation. The explanation for this could be that our demographic filter only takes into account the gender, and not other demographic factors such as age or occupation. Expanding the demographic filter to include

these additional factors may help address the observed disparities in RMSE across different age groups and occupations.

We notice also that for age, the less data we have for an age group, the more heterogeneous the RMSE is.

RMSE per gender

RMSE per age

■ RMSE

Chart Title
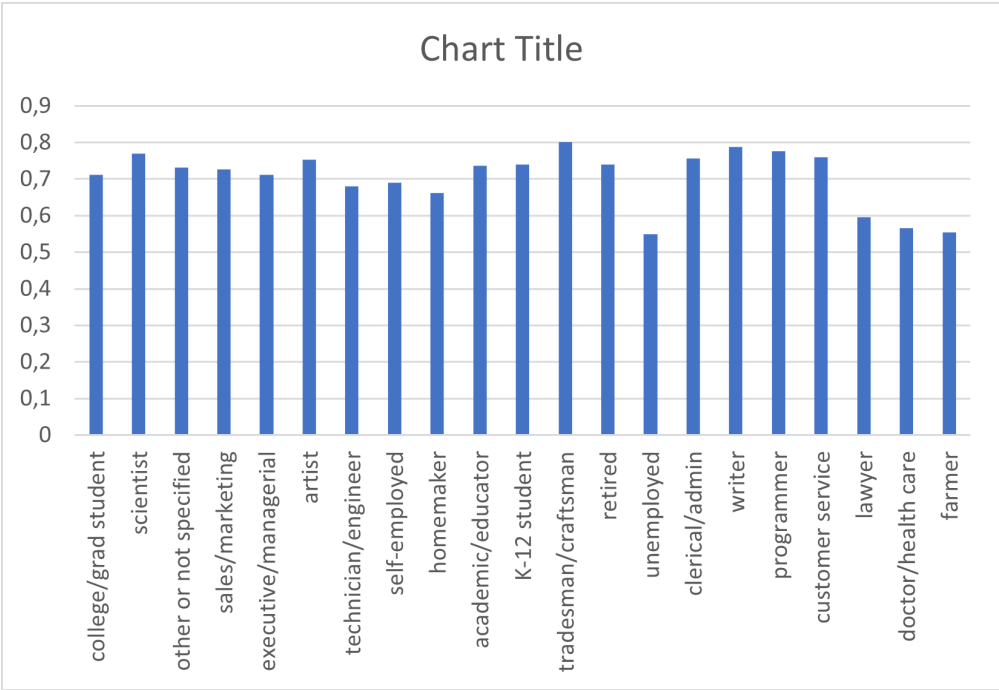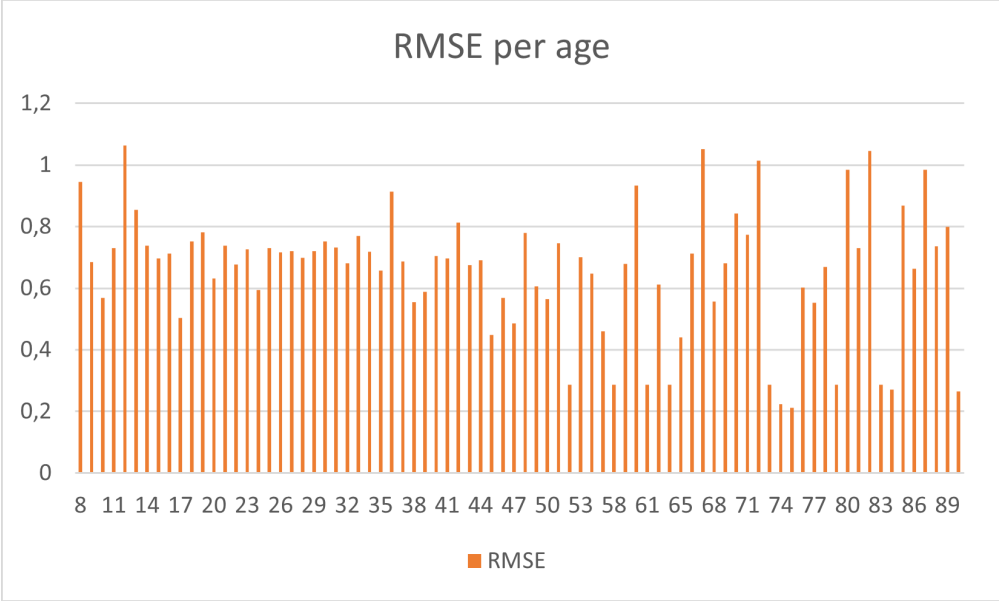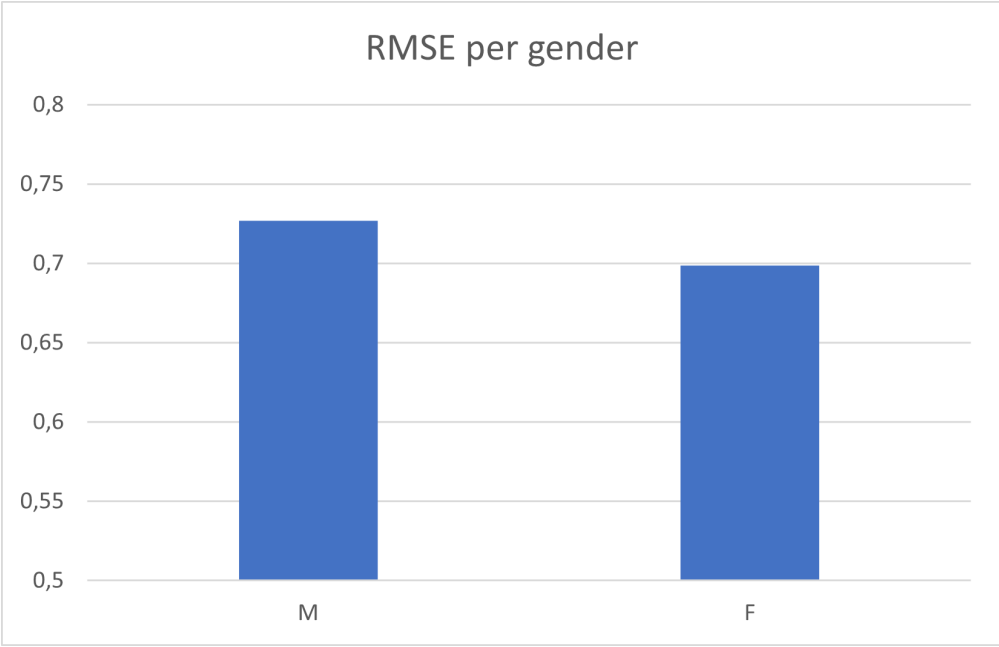
You can find all the statistics and results of our analysis here, and the code to generate these results here.

Feedback loop

We decided to analyze the Echo Chamber feedback loop. To gather a new dataset of ratings and examine how ratings have evolved since the implementation of our model, we calculated the average ratings for each movie in our initial training set and in the new dataset we have just collected. We then compare the evolution of the ratings:

In an Excel spreadsheet, we compared the correlation between the ratings given to these movies in our training dataset and the difference (both absolute and non-absolute) in the average ratings these movies received between the beginning of the semester and now. We have observes correlations, but these can be explained by an other factor than the Echo Chamber feedback loop.

For example, we observe a very significant negative correlation when comparing with the non-absolute difference with the previous mean of ratings (r = -0.61221059, p = 3.0293E-250). This means that the most liked films are the ones that have lost the most in popularity, and the least liked films are the ones that have gained the most in popularity. However, this can be explained by the fact that movies that are nearly rated 5/5 have little room for improvement, and movies that are nearly rated 1/5 have little room for deterioration.

You can find the implementation of this part here.

Reflections on Recommendation Service

We had no shortage of problems, both technical and logistic, in developing our recommendation system. But in the end, we are happy to have encountered these problems as you learn more from these challenges than you would if everything worked perfectly fine. Here are 3 main challenges we dealt with:

Cold start problem

Challenge: One of our biggest challenges in developing a decent recommendation system was the lack of sufficient data, which limited the model's effectiveness and accuracy. Since we had a million users, we would need a substantial amount of data related to their watch history and ratings to give meaningful predictions. A major limitation with the collected data was duplication, since there was a log entry for each minute of the movie watched by a user. When we filtered these logs to extract unique entries, there was often a 70-80% reduction. For example, 5 million records would reduce to just 50,000. However, the cold start problem is common in recommendation systems and better data collection pipelines would have helped us improve our models.

Future Improvement: Moving forward, the focus will be on enhancing data collection and curation processes. We would need to set up a more robust system to collect, parse and store our logs. Acquiring more comprehensive and diverse datasets is a primary goal to improve the model's performance and reliability.

Telemetry Collection System

Challenge: The initial challenge was managing a vast volume of logs. Loki was chosen for its promise of efficient storage, but it presented limitations in log retrieval as we could only retrieve 5,000 logs in one API request and it also averaged high CPU usage.

Current Solution: Our team transitioned to a file-based system, opting to store logs in CSV files. This approach, while simpler, demands more storage and requires regular log cleanups. However, given the timeframe of the project, we decided this was a reasonable approach.

Future Direction: With additional resources, the ideal solution would be to implement a database specifically for telemetry post-processing, aiming to enhance storage efficiency and retrieval capabilities.

Load Balancing and Kubernetes

Challenge: Implementing Kubernetes for load balancing and canary deployment was challenging, particularly due to the lack of root access and difficulties in configuring load balancers on port 8082. We installed Docker using Kubernetes in Docker (KIND) which is alright for test environments. While we were able to set up all pods, deployments and services, and our service worked on port 80 of the team URL, we were not able to get it to work on port 8082, the port to which the simulator sends requests. We believe this was an ingress configuration issue, but could not figure out a solution. After putting in around three to four days of relentless effort into fixing this without success, we decided to move on.

Current Solution: Our team reverted to a simpler solution using an NGINX load balancer and bash scripts for canary deployment. All the containers are orchestrated using docker compose and Docker networks.

Reflection: This part of the project highlighted the importance of balancing ambitious technological implementations with practical project management considerations. Kubernetes may have been a bit overkill for the scope of our project, but we wanted to kill two birds with one stone (load balancing and canary releases), which did not turn out very well. It is important to consider the scale of the project and available resources and limitations before venturing into a big technology.

Reflections on Teamwork

What went well

Our experience in this team has been quite a rollercoaster ride, but in the end, we believe that things turned out fairly well. We did not have the most ideal team kickoff, but we managed to settle disputes and patch up stronger after that. Over time, we understood each other better and our workability and compatibility increased. We also some exchanged some skills with each other and got to learn new things and workstyles from each other. Especially during M3, everyone was active and pitched in their ideas and collaborated well. The whole project helped us develop not only technical skills, but also some valuable interpersonal skills that we present below.

Challenges

- **Ensuring that all team members feel included**: We started off on a not-so-good note as there were some initial conflicts with regard to the involvement of all team members. Some team members knew each other outside of the course and this led to misunderstandings with regard to involving other team members, though it was never the intention. However, these were resolved by talking things out and setting expectations right. After the small tiff, our team cohesion was quite good and we had no differences with each other for the most part.

- **Juggling commitments and varying availabilities**: Each team member had other commitments in terms of coursework from other classes or research projects. This made it hard to find a common time to meet and to have a continual progress on the milestones. We were quite diligent with our meetings at the start, but as we got burdened with increasing workloads converging at the same time, the regularity of our meetings started to wane. We realize that this is typical and expected of a university student, so it is unfair to expect team members to devote a hundred percent of their time toward the project. But it is important to set up an appropriate workflow to work asynchronously and sync up regularly to ensure the project does not stall.

- **Responsiveness**: As the team got busier, we experienced communication gaps that slowed down our progress as a whole. Sometimes it used to be tough to get a response or acknowledgement from all team members when issues were being discussed or work was being allocated via WhatsApp. We realized that messaging, in general, tends to be a less engaging form of communication and actual conversations are important

from time to time.

- **Balancing expertise**: Though our team had varying domains of expertise that helped cover different aspects of the project, this also posed a challenge at times, especially in M1. As only one member in our team had solid hands-on machine learning experience, there was a high dependance on them to get things moving for M1. However, this was taken care of in subsequent milestones as the project's scope widened out.

Takeaways

Based on the challenges we discussed, here are the lessons we learnt to make future collaborations more productive and successful:

- **Being accountable and taking initiative**: The essence of teamwork is the team working together and playing an equally active role towards the advancement of the project, at least in projects in such a context. Despite one's packed schedule, if each team member takes personal responsibility to allot some time for the project at hand rather than leaving the heavylifting to just a few members, it can improve the overall quality of the project and can prevent cramming work close to the deadline.

- **Accept constructive criticism**: We did not have a major problem in this regard, but something that we observed in general is that it is important for one to be open minded and accept suggestions and different perspectives rather than be fixated on one's own idea. You should also be willing to accept that you don't know things and should be willing to learn from your teammates.

- **Pair programming is a good productivity booster**: We did not have extensive pair programming for the first two milestones, however, we did this fairly regularly for Milestone 3 and found it to be quite effective to bounce off ideas and get work done by sharing each other's expertise. We were able to complete our work faster and accomplish more this way. This is something we would definitely want to do more of in future projects.

Some points worth highlighting about our implementation:

- Our commit messages are linked to the version of the data and model being committed. It gives a clear overview when observed within the repo as follows: (more description on why this is so - added in provenance)

cleaned_rating_35.csv	Data and Model version update: [35]	10 hours ago
cleaned_rating_36.csv	Data and Model version update: [36]	9 hours ago
cleaned_rating_37.csv	Data and Model version update: [37]	4 hours ago
cleaned_rating_38.csv	Data and Model version update: [38]	6 hours ago
cleaned_rating_39.csv	Data and Model version update: [39]	6 hours ago
cleaned_rating_4.csv	Data-integration -> development	11 hours ago
cleaned_rating_40.csv	Data and Model version update: [40]	6 hours ago
cleaned_rating_41.csv	Data and Model version update: [41]	6 hours ago
cleaned_rating_42.csv	Data and Model version update: [42]	5 hours ago

| main ⌄ | team-4 / app / model / versioning / | + ⌄ |

Name	Last commit
..	
📄 model_30.pkl.dvc	Data-integration -> development
📄 model_31.pkl.dvc	Data-integration -> development
📄 model_32.pkl.dvc	Data-integration -> development
📄 model_33.pkl.dvc	Data-integration -> development
📄 model_34.pkl.dvc	Data-integration -> development
📄 model_35.pkl.dvc	Data and Model version update: [35]
📄 model_36.pkl.dvc	Data and Model version update: [36]
📄 model_37.pkl.dvc	Data and Model version update: [37]
📄 model_38.pkl.dvc	Data and Model version update: [38]
📄 model_39.pkl.dvc	Data and Model version update: [39]
📄 model_40.pkl.dvc	Data and Model version update: [40]

| 📄 version.txt | Data and Model version update: [44] | 1 hour ago |

(commit includes the version number which we save in the version txt)

- If the average response time of the deployed canary release is greater than 500ms then the canary release is abandoned. We get slack and email alerts for the same.

> **incoming-webhook** `APP` 2:57 PM
> The Canary release could not be deployed. Switching from canary to stable deployment. The average response time was more than 500ms: 700ms.
>
> ✅ 👀 🙌 😅 💬 ↪ 🔖 ⋮
>
> **incoming-webhook** `APP` 11:25 PM
> The Canary release could not be deployed. Switching from canary to stable deployment. The average response time was more than 500ms: 650ms.

ALERT: Canary release was aborted.

C ci.cd.team4@gmail.com   ☺ ↩ Reply ↩ Reply All → Forward 🗂 ⋯
To Aayush Kapur   Wed 11/29/2023 2:57 PM

[You don't often get email from ci.cd.team4@gmail.com. Learn why this is important at https://aka.ms/LearnAboutSenderIdentification]

The Canary release could not be deployed. Switching from canary to stable deployment. The average response time was more than 500ms: 700ms.

- If the rmse_score is acceptable after auto-updation of data and models then we proceed with deployment of the pipeline. Upon successful auto-updation, we get the following email. This is separate from the alerts we get after aborting a release which was deployed as canary but could not be switched to the main container due to average response time issue mentioned above.

Successfully deployed new model

C   ci.cd.team4@gmail.com
        To   Aayush Kapur

☺   ↩ Reply   ↩ Reply All   → Forward   🔑   ...

Wed 11/29/2023 5:07 PM

[You don't often get email from ci.cd.team4@gmail.com. Learn why this is important at https://aka.ms/LearnAboutSenderIdentification]

Your deployment of the new model was successful !

- The screenshot of how our mysql database looks for per request tracking, has been attached under provenance.

- We have a nifty logging module. Following is the screenshot of how we create the logs. We tried to create ours in a similar fashion as a production app.

```
team-4 > app > auto_deployment > ≡ logging.txt
  1   2023-11-29-23-22-06 auto_deploy INFO     Update the repo in Server.
  2   2023-11-29-23-22-07 auto_deploy INFO     Data collection & pre-processing
  3   2023-11-29-23-22-52 auto_deploy INFO     Update the repo in Server.
  4   2023-11-29-23-22-52 auto_deploy INFO     Data collection & pre-processing
  5   2023-11-29-23-23-54 auto_deploy DEBUG    Run data processing.
  6   2023-11-29-23-23-54 auto_deploy DEBUG    Append the cleaned data.
  7   2023-11-29-23-23-57 auto_deploy DEBUG    Containers for Kafka images deleted.
  8   2023-11-29-23-23-59 auto_deploy INFO     Train the model with new data.
  9   2023-11-29-23-23-59 auto_deploy INFO     RMSE score for the trained model: 0.722037367975316.
 10   2023-11-29-23-24-01 auto_deploy DEBUG    Model and data versioning using version [44].
 11   2023-11-29-23-24-01 auto_deploy DEBUG    Stage dvc model file and data versioning files.
 12   2023-11-29-23-24-01 auto_deploy DEBUG    Stage new data and new_version: [44].
 13   2023-11-29-23-24-01 auto_deploy INFO     Insert the tracking information in the db.
 14   2023-11-29-23-24-08 auto_deploy INFO     Successful auto updation of the model.
 15   2023-11-29-23-25-46 canary_fail ERROR    The Canary release could not be deployed. Switching from canary to stable deployment.
 16   2023-11-29-23-25-46 canary_fail ERROR    The average response time was more than 500ms: 650
 17   2023-11-29-23-25-50 canary_fail DEBUG    Successfully sent email alert for Canary container abort.
 18   2023-11-29-23-25-50 canary_fail DEBUG    Successfully sent slack notification for Canary container abort.
 19
```

Contributions

Each team member was alloted one major task and helped other team members in their tasks. The distribution was as follows:

- Aayush: Provenance
- Rishabh: Automatic model updates and deployment
- Varun: Canary releases
- Tamara: Feedback loops and fairness
- Yaoqiang: Reflection on recommendation service and automatic deployment

Each team member worked on a separate branch for their respective features. In general, we tried to follow the following naming convention for all branches: `name-feature` .

A summary of our meeting notes can be found here

Contributions by Tamara

Conceptual Analysis of Potential Problems: Tamara (commit 1, commit 2, commit 3, ... (all the commits on the report)) Analysis of Problems in log Data: Tamara (commit 1, commit 2, commit 3, commit 4)

Contributions by Yaoqiang

Data processing scripts for automated model updates:

I worked on the Auto Development for M3. One of the significant contributions is which I implemented a script, kafka_consumer_data_appender.py, that streamlined our data handling. This script automates the collection and processing of real-time movie ratings from a Kafka server, elegantly solving the challenge of efficiently managing and integrating large data streams.

Relevant commits:

!54 (480f1c1d) !54 (66c0123a) !55 (cd2667c6) !55 (ecbc4098) !55 (9672fb95)

Meeting management: led meetings on fairness analysis approaches and documented these sessions, providing clear and concise meeting notes for team reference.

Relevant commits:
Meeting notes created: !54 (1c4f4051)

Pull requests reviewed by Yaoqiang:

!52 !51

Report writing:

6e72b5ae

- Also brainstormed on the Auto Development with Rishabh.

Contributions by Rishabh:

- **Developed the automated model updates script**
  (Corresponding issue: https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/blob/main/app/auto_deployment/auto_deploy.py

  - Developed the complete auto-deployment script, linked with data integration from Kafka stream, integrated the emailer, scheduler, git commit logic, etc.

- **Developed the Python scheduler:** https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/blob/main/app/auto_deployment/scheduler.py

  - Created a Python scheduler to trigger the auto deployment script. This was done because the cron job that I earlier created had some issues with the reletavive path of the installation of Python on the server.

- **Developed the email notification service:** https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/blob/main/app/auto_deployment/emailer.py

- **Pipeline/Offline eval metric** (Corresponding issue: #52)
  Code file: https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/blob/main/app/tests/model/test_model.py#L149
  Added the offline evaluation metric that I developed, i.e. chacking if the RMSE score is < 1 to the pipeline.

  - **Commits:**
    1. da16a681
    2. ddf07bcb

- **Debugging and making sure that the pipeline is working fine**: #52

  - Checked if the pipeline is working fine and covers all the componenets required. Made some changes to fix the pipeline.

- **Brainstormed with Varun on Canary releases, helped with the correct paths/routes and failing test cases.**

  - commit: fc883bdf
  - Helped resolve the issues with Canary deployment integration and corrected the routes in the code to fix this issue.

- **Assisted Yaoqiang with Kafka data processing:**

  - Helped Yaoqiang create the data processing scripts required for the auto_deploy script and fixed the issues with his code.
  - Did an in-depth review of the code as well, making sure that everything was well integrated.

- **Helped Aayush with data integration in the `auto_deploy.py` script**

- **Repo quality assurance:**

  1. Added rule to have at least 1 approval for MR
  2. Added rule to merge only when the pipeline succeeds

Merge requests reviewed by Rishabh: Reviewed majority of the merge requests, linking some for your reference:

- Fixed the hanging bug: !55
- Canary release set up and scripts: !52
- Add the db provenance changes from data-integration to development: !63
- Tamara fairness feedback loop: !67

Meeting notes created: https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/wikis/M3:-Issue-discussion,-debugging-and-development-sync-up

Report writing:

- Added the automatic model updats part to the report

Contributions by Varun

- **Containerization of the application**: Had developed the containers as part of M1 and M2. Updated the container and docker compose configuration to match M3 requirements (64ddda2d, df6d166e)

  Files: https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/blob/64ddda2d946a8bfe6ca8d13efbb8d2a3e28d9b06/app/docker-compose.yml, https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/blob/main/app/nginx/nginx.conf?ref_type=heads

- **Reconfiguration of monitoring service to track canary and stable deployments** (5a15738a,)

Files: https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/blob/main/app/nginx/nginx.conf?ref_type=heads, https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/blob/development/monitoring/prometheus/prometheus.yml#L24

- **Set up of the canary release pipeline**: Spent a considerable amount of time experimenting with Kubernetes before switching to the current version of the release workflow. (b85b9e54, 64ddda2d)

  Files: https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/tree/varun_k8s/kubernetes, https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/blob/b85b9e545f1aa2e99387641365b22434d15d25c9/scripts/deploy.sh, https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/blob/development/.gitlab-ci.yml

- **Participated in regular dev sync ups with other team members to bounce off ideas and help in development.** Pointers from meeting notes:

  - https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/wikis/M3-Meeting-Notes#progress-sync-up
  - https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/wikis/M3-Meeting-Notes#meeting-debugging-01
  - https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/wikis/M3-Meeting-Notes#meeting-debugging-01

- **Other debugging and fixing activities**:

  - Setting a fixed current working directory for `movie_rec.py` to resolve varying path issues: https://gitlab.cs.mcgill.ca/comp585_2023f/team-4/-/blob/726a1df1af5df3b3d019997b03369782aea57477/app/model/movie_rec.py#L10

- **Report writing**:

  - Containerization
  - Releases
  - Reflections on Teamwork
  - Refined Reflections on Recommendation Service

Merge requests reviewed:

- dev --> main: !59
- Offline evaluation metric: !51

Contributions by Aayush:

I have provided explanation on what I did and linked the commit for the same. The issue number is added in the commit message. Helps in linking issues to commit on gitlab.

- **Developed the dvc infrastructure locally on the host- connected it to auto data and model updates,Set up the versioning for data files and models**: commit

- **Developed per request tracking solution** - Created the mysql container deployment, connected it with the auto updates, linked release containers to it so that insertion queries go through: commit, commit

- **Fixed stale docker containers issue during data collection**: commit

- **Created slack and email alerts for failed canary containers deployment**: This is different from the pipeline deployment failure mentioned above. This alert comes from the deploy script based on the average response time of the canary container: commit

- **Added the infrastructure for logging** - for auto_deploy, canary container fail: commit

- **Helped Rishabh with auto-deployment part**- Data integration for automated model updates: commit

- I took initiative in the earlier part of the project to get things rolling because I had back to back exams and other deliverables near to M3 deadline. I raised this point with prof. too. As a result, spent quite a bit of time to get deliverables in perspective. I discussed extensively with Varun. We combined all the expectation, project requirements, report requirements and points distribution in a single page to have a clear idea. link

- **Kubernetes**: I spent a lot of time trying to get kubernetes to run on our team's server. I worked closely with Varun and we spent around 4 days cumulatively. I learnt so much during the process but it led to a lot lost time which could have been used elsewhere. We were able to get it working only on port 80 of the exposed api which sadly was not practical for this project. The intention behind using kubernetes was the production grade capabilities for load balancing, canary releases and replicas to ensure availability. I even reached out to Deeksha for the same. I have extensively documented the issues observed and the command line remnants from one of the troubleshooting sessions. [In the end, we ditched Kubernetes and went with a simpler implementation for load balancing, canary releases and replicas.]

  - link to issues in detail: here, here
  - link to troubleshooting: link
  - We tried:
    - Reverse proxy through nginx container outside the cluster was tried.
    - ip tables was tried.
    - creating pods with nginx + ubuntu was tried inside the cluster.
    - creating docker containers to ditch cluster was tried then using redirection requests. get was tried as well.

- Helping out the team with the general stuff: report writing, presentation, coordination and team management.

Merge requests reviewed: Reviews given: [request 57](), [request 62]()

MR approval based workflow was followed within team: [example]()

Report writing: `Add_provenance`, `Report_writing_commit`

Meeting notes created: We followed an evolving doc approach for making meeting notes. In the single wiki page, team members kept on adding more info. for M3. My contribution can be tracked from history tab of that page. Link to evolving wiki page: [here]()

Other notes: M3 stuck on port issue for kubernetes cluster: [here]()

Project session Nov 7 notes: [here]()

Note: We felt that everyone worked great for M3. Each member really got involved in the project and pushed as best as they could.

1. https://surpriselib.com/ ↩

2. https://gunicorn.org/ ↩

3. https://www.nginx.com/ ↩

4. https://pypi.org/project/prometheus-flask-exporter/ ↩