Team 3 - Milestone 2

Discussion and overview

To build a better foundation for our recommender system in this milestone, we started by addressing the key limitations of our existing model. After reviewing the results from Milestone 1, we determined that our most promising opportunities for improvement would come from migrating away from PySpark to a more flexible ML library like Tensorflow. So, we decided to switch our model to TFRS (Tensorflow Recommender System), an open-source library built on Keras.

This switch allowed us to address some limitations discussed in Milestone 1. Namely, we could integrate more features into the model, including information about users (age, occupation, and gender) and movies (duration, genres, original language, and year). It directly helps the system with the cold start problem by allowing it to infer recommendations based on specific traits for any user. The new model uses the idea of two separate embedding towers (users and movies) connected via a neural network to predict the probability of positive interaction. Its performance also eliminated the need to rely solely on a pre-processed cached table for speedily responding to API requests.

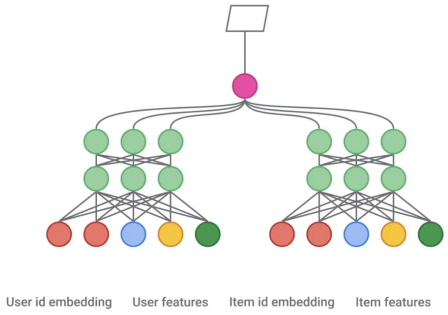

Fig: Model structure

The original idea was to construct a cascading model by retrieving 200 movies using the retrieval model and then using the ranking model to cherry-pick the top 20 movies. However, we later altered this idea due to hardware limitations which caused the inference times to far exceed the standard API response time. So, the system only uses the retrieval inference model to generate the top 20 movies (i.e., without cherry-picking and ranking from 200 movies).

Additionally, since our model does not evolve continuously in production, we only compute the recommendations through our model once and then store the recommendations for each user (in Redis, for fast I/O) to later serve the same recommendations without having to re-compute at every request. This strategy helped us greatly reduce the CPU load on our system, ensuring more stable and consistent performance over time.

Offline evaluation

The system uses deep learning and starts with two datasets, the first one being the "watched" dataset, where we utilize the Kafka stream to grab several watched movies' data by having the threshold as 10 minutes. To be included in the data, each movie had to have been watched for at least 10 minutes by the user and received a rating of at least 3. To reduce bias/skew in our data, we collected the training and validation datasets from Kafka over several days, ensuring that we have sufficient diversity and representation in these datasets. We deliberately truncated much of the watched data to guarantee the implicit dataset is not overflooded by negative interactions. We then merged it with the user and movie data, such as the age and gender of the user, the genre and length of the movie, etc., to form our implicit dataset. We dropped duplicate rows in our datasets to reduce the chances of data leakage, ensuring all rows in training and testing sets are unique. The data collected was then randomly split into 80% for training and 20% for validation to reduce the chances of overfitting while also having sufficient training data. We used a combination of implicit (watched movies) and explicit datasets (rating movies) to train the retrieval model. We only use the explicit dataset to train the ranking model; we use all rating data, merging it with user and movie data. We also kept the number of epochs we trained our model on at a level that reduced any chances of overfitting to the training data. Cross-validating with test data ensured the model performed well for unseen data. The retrieval model is responsible for retrieving 200 movies for that specific user, and the ranking model ranks the 200 films and finds the best 20.

The evaluation strategy we used for the retrieval model is the “Top K” strategy which refers to checking whether the movie the user watches is among the first K movies recommended. The K we used to evaluate our retrieval model was 5, 20, and 500. We kept the RMSE evaluation strategy for the ranking model because we are predicting a specific rating for this movie and user pair. We trained the retrieval model with 50 epochs and the ranking with 10. The testing results are as follows: For the retrieval model, the Top 5 accuracy is around 0.124, the Top 20 accuracy is 0.144, and the Top 500 accuracy is 0.328. For the ranking model, the RMSE was 0.717. In comparison, our model from Milestone 1 had an RMSE score of approximately 1.70. Using different evaluation strategies for each of the models also helps us to avoid the problem of misleading aggregated metrics, as one model is always independent of another when we carry out the basic statistics. Since we are evaluating the retrieval model and the ranking model individually, this avoids the common pitfall of using misleading aggregated metrics to guarantee that the model is improving individually.

See our model's (auto-generated) evaluation report here:
github.com/COMP585Fall2022/Team-3/pull/35#issuecomment-1295946893

The code implementations can be found at
1. Data processing: github.com/COMP585Fall2022/Team-3/Datasets/
2. Model training/testing:
github.com/COMP585Fall2022/Team-3/Model%20Code%20and%20Tuning/modularize_tensorflow/

Online evaluation

For online evaluation, the key metric we use to measure the quality of our recommendations is whether users select movies from the movie recommendations. After several iterations of different ways of interpreting Kafka data, we designed our online evaluation KPI to use Mean Reciprocal Rank (MRR) calculated from *watched movie* logs in Kafka.

Firstly, we record recommendation logs from the Kafka stream for every individual user whom our model makes a recommendation for. In parallel, we monitor if the user watches any movies from the list of recommendations we sent. If a movie from the recommendations is watched after the recommendation is made, we calculate a *rank* based on the movie's position in the list of recommendations. For example, if a user watched the third movie from the recommendation set, the rank for the recommendation is set to ⅓. The MRR is then calculated by averaging all rank values over a specific time period, which in our case is every 10-minute time-buckets in Grafana. Since the order of the recommendations is expected to be from highly to least likely to watch, this metric helps us evaluate how well our model predicted the order of the recommendations. The code for this monitoring and metric calculation can be found here: github.com/COMP585Fall2022/Team-3/kafka-consumer/monitor.py

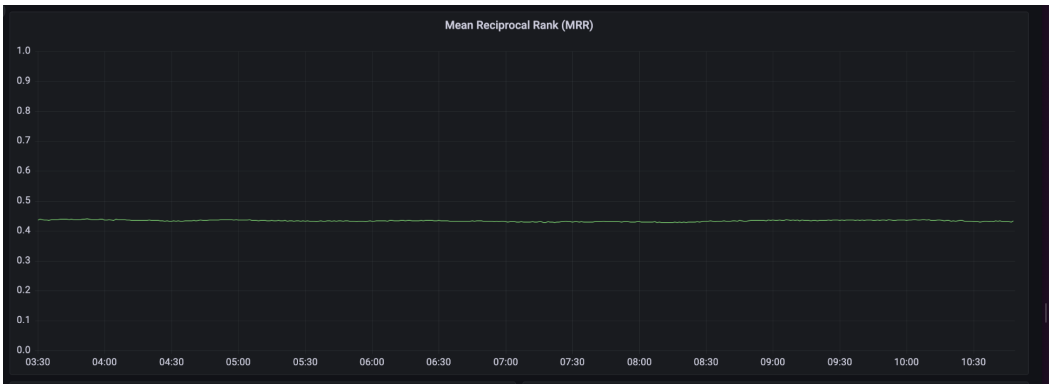

Fig: MRR of our model visualized in Grafana over eight hours; stable at approx. 0.44

In addition to the MRR, we also monitor and record the total number of (recommended) movies users watch over time. When this data is plotted over time, this visualization helps us determine whether our model performs better/worse over time; a straight line with a positive gradient would mean that the quality of the predictions doesn't change, whereas a curve with an increasing gradient would imply that the model predictions get better (more and more of our recommended movies are being watched), and vice versa.

Fig: Visualization of the total number of recommended movies watched over eight hours; a straight line.

We are also tracking how long it takes between a recommendation being made and watching a movie from these recommendations. Visualized over time, this indicates how users are responding to our recommendations.

Fig: Average time until a recommended movie is watched; stable at approx. 1hr 10mins

Additionally, to track our system health (uptime, CPU consumption, memory, total no. of requests served, etc.), we set up monitoring/alerting infrastructure and dashboards around our API using Prometheus, Alertmanager (with Slack integration), and Grafana. These services can be accessed at

1. Grafana: http://fall2022-comp585-3.cs.mcgill.ca:3000/ [username: **admin**, password: **pass@123**]
2. Prometheus: http://fall2022-comp585-3.cs.mcgill.ca:9090/
3. Alertmanager: http://fall2022-comp585-3.cs.mcgill.ca:8082/

The code configurations for these services can be found at:
github.com/COMP585Fall2022/Team-3/Monitoring/

Data Quality

As described briefly in *Offline Evaluation*, we created several filters and post-processed the data we collected through Kafka, giving us a reasonably good set of datasets to start with. In addition, we further processed the data to ensure that every empty value in the data was filled with a default value. We also filtered out the movies that did not have good ratings and those that were not popular enough (based on the frequency of appearance) since we did not want the system to recommend poorly-rates or least-viewed movies. The movies with low ratings in the dataset are also not likely to be the first twenty movies we can recommend as they already have a lower score than those with good ratings. This helped decrease the training time significantly while not having any evident adverse effects on the evaluation metrics for the final model's recommendations.

To check for the quality of our datasets, we created automated unit tests which test for the criteria we have determined for 'good' data. This includes checks for duplicates, null values, out-of-range values for columns, invalid/unexpected data types in rows, and data distribution for training/testing, to name a few. [See implementation code here]

We also filter and post-process the Kafka logs we're actively reading for Online Evaluation to ensure the system parses out invalid logs and handles any exceptions when parsing fails. This guarantees that the metrics we see for online evaluation accurately represent the data. [See implementation code here]

Pipeline implementation and testing

To help us test, validate, and review contributions to the repository, we've established several contribution guidelines. Notably, not pushing directly to the main branch and instead creating feature branches paired with Pull Requests. These are then aided by automated testing using GitHub Actions to quality test the code and stop the integration process if there are errors. Peers also review contributions to help catch any issues that automated tests cannot detect. Together, these tests and reviews help ensure that the code is high quality and meets standard software engineering principles. [See implementation code here]

A part of our automated test strategy is based on unit testing, but there are some cases where black-box testing is also applied. Black-box testing is used for testing ML models (Model, discussed in the next section). In contrast, white-box testing checks individual software components (Kafka consumer, Database, Data quality, Flask API). Contributors write unit tests with a good understanding of the code's internal structure. These unit tests are automatically run when the code is pushed to GitHub, and a code coverage report is generated (e.g., see a coverage report). We've also established a minimum acceptable code coverage threshold of 70%, below which the pipeline throws an error and fails.

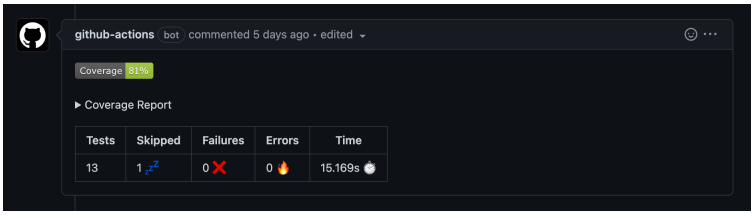

Fig: Auto-generated code coverage report on Pull Requests.

Each contribution to the repository is tested for code formatting and styling (lint testing). This helps us proactively detect and fix styling issues such as unused variables, unused imports, accessing undeclared variables, etc. These tests are done using the python packages: black, flake8, pylint, pycodestyle, and isort, helping us ensure that the code is readable and consistent across all members.

In addition to unit and lint testing, our contributions are also tested for build errors. This process automatically compiles our entire project code in a simulated environment on Docker containers (similar to how we expect to build and deploy in production) and checks for any build errors.

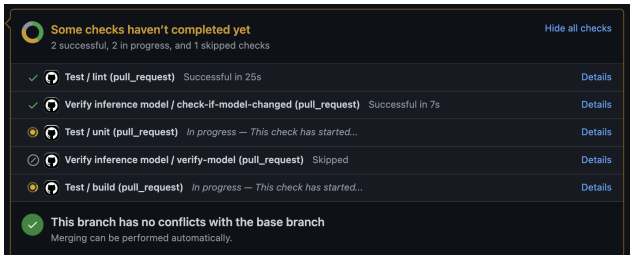

Fig: Automated testing pipeline checking for errors on Pull Requests.

Continuous integration

In addition to individually testing the different components of our project as described above, we have also set up automated CI workflows which trigger when changes are made in our model code which runs model training/testing and displays the evaluation metrics and baseline comparisons for review in a fully automated, hands-off method. This is made possible by utilizing Pull Requests and GitHub Actions paired with Weights&Biases.

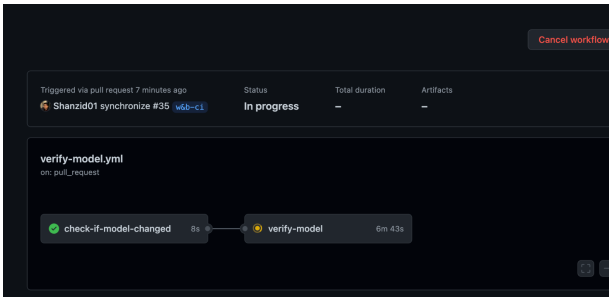

Fig: GitHub Actions script checking for changes in model code and triggering model verification

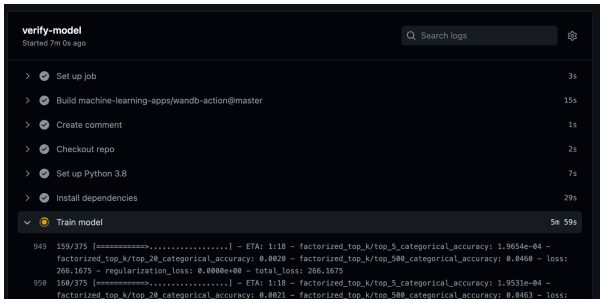

Fig: Model training running automatically in GitHub actions and sending the data to W&B.

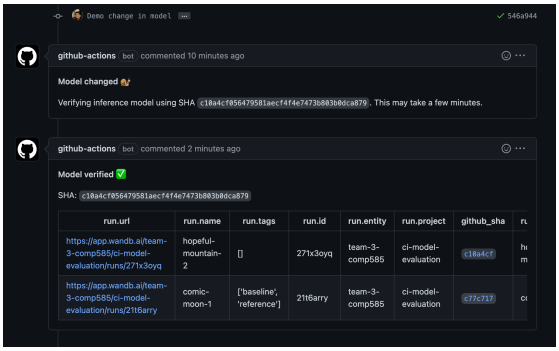

Fig: GitHub Actions automatically displaying the model evaluation results in the PR.

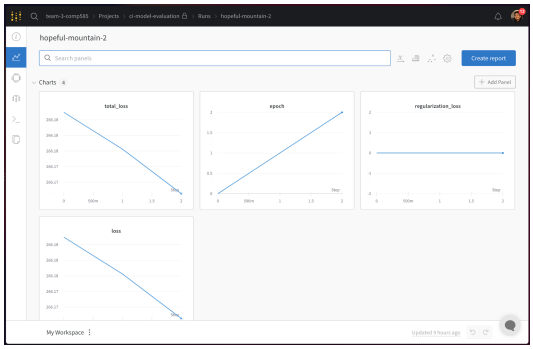

Fig: Detailed evaluation metrics viewable by team members from Weights&Biases.

As an example of this workflow, see https://github.com/COMP585Fall2022/Team-3/pull/35. For access to Weights&Biases, please contact Shanzid Shaiham. See the code for Actions here.

NOTE: Due to hardware limitations, the CI examined here are toy examples we use to check whether certain parameters work by using smaller datasets and smaller epochs. After examining it, we can choose to train the model with that set of parameters using much more epochs and bigger datasets. This allows us to examine the quality of some parameters while not taking a big chunk of time to train the model with all the data and a large number of epochs in such limited hardware.

Individual Contributions

Shanzid Shaiham (SWE)	Setting up the infrastructure for - testing code quality (unit tests, formatting, code coverage, etc.) - system monitoring (Grafana+Prometheus+Alertmanager+Slack) - Docker configurations to run and orchestrate project - CI/CD with GitHub Actions Set up Redis for data storage in online monitoring & evaluation, as well as set up scripts used to calculate and visualize the metrics in Grafana. Also contributed to writing the report and proofreading. See contributions authored with Pull Requests: https://github.com/COMP585Fall2022/Team-3/pulls?q=is:pr+is:closed+author:Shanzid01
Barry Li (ML)	- Study and add more embedding for the models. - Preprocessed the data for learning. - Modularized the machine learning code - Implement the Continuous Integration with Weights & Bias - Unit test the model See contributions authored with Pull Requests: https://github.com/COMP585Fall2022/Team-3/issues?q=is%3Aclosed+author%3AWeienLi The second version model and data preprocessing can be found here: (Did not use Pull requests since it was shared using GDrive): https://github.com/COMP585Fall2022/Team-3/blob/main/Model%20Code%20and%20Tuning/Second_Version_Tensorflow_RS_v4.ipynb
Kua Chen (SWE)	Online evaluation, data quality tests, Kafka consumer tests, data collection See contributions authored with Pull Requests: https://github.com/COMP585Fall2022/Team-3/pulls?q=is%3Apr+author%3AChenKua+is%3Aclosed
Marcos Souto Jr. (ML)	Study and implementation of new models using the TFRS library (Tensorflow Recommender Systems) and the basic code for integration with the Weights & Bias tool (tracking the training step of the models). See contributions authored with Pull Requests: https://github.com/COMP585Fall2022/Team-3/pulls?q=is%3Apr+is%3Aclosed+author%3Amsoutojr
Zeyu Li (ML)	Evaluate the model, Take the meeting notes, Write report

Meeting Notes

Oct. 17

1. Discuss the new model(neural network)(not trained yet) which solves the previous problem(cold start and problems for the ALS). Generates users with movies. The drawback is users might not have seen enough movies, less than 20. Accuracy is measured with “top k”.
2. Data: maintain 20/80(the gold standard). Build train, validation, and test for both data and model. This can deal with missing and duplicated data. Decide a minimum threshold.
3. Online evaluation: feedback loop, running time, on the server
4. Offline evaluation: valid dataset

Oct. 19

1. Mechanism to collect implicit data.
2. Interpret how we consider if the user watched the movie: set up a threshold for user review time
3. Set up unit tests, and make sure the predict function works
4. Online evaluation: able to detect how the model behaves, and whether it is performing as expected, the matrix for online evaluation is not the same as offline.
5. Fix the API problem from the previous milestone
6. Collect data1: user id, movie id, whether the user watched the movie(boolean, using threshold, only the ones that pass the threshold), time stamp(when the user watches the movie, if possible)(5-6 million)
7. Collect data2: user id, movie id, ratings(1 million)
8. Make sure the data quality is clean and correct

Oct. 22

1. Set up the automatic request for data required and an alert manager, alerts are sent to slack
2. The model has been done with features(age, gender etc.) added. After the optimization, the RMSE is 0.7, which is 58.8% lower than previously detected
3. The model has a little problem with the feedback loop, and it will be fixed soon
4. The model is still trying to learn(has not converged yet), which needs more time to train and will focus more on the feedback loop
5. Ranking: RMSE, Offline evaluation: top k
6. Offline evaluation tests are required(unit test for API)
7. Deal with boundary cases(users that don’t exist)

Oct. 24

1. Errors often occur when memory goes out, but through improvement, there are much fewer errors reported
2. Not all movies are selected as candidates, but all users are contained.

3. Consider the weights and biases in TensorFlow. An evaluation is generated using weights and biases, which shows the report of the model, and how it behaves, and this is considered to be offline training.
4. The training time is 5 minutes and 30 minutes, which is not a problem anymore.
5. Online evaluation: MRR: checks the recommended movie ranking in the first 20 movies we expected. Even if a movie we didn't consider is chosen, zero is reported. Make a recommendation first, and then check the Kafka stream if the user watched the movie.
6. Data quality: index the gender and occupation. We have data for all users but not all movies.
7. Pipeline implementation: recreate a pull request(already done)
8. (SWE + ML): Setting up a CI pipeline that trains our model automatically and reports the evaluation metrics through Weights & Biases [2]. This would fall under the Continuous Integration topic of the milestone.
9. (SWE): Setting up a system that lets us know if users have watched the movies we're recommending. We should store the recommendations we're sending for each user and then constantly monitor Kafka logs to see if a user watches a movie. We should then log these results and preferably display them on Grafana.

Oct. 26

1. With the installation problem discovered, we changed the strategy to run the model in the conda environment inside the docker, which required more time. The expected time is 120ms per request.
2. CI: test whether the code trains the model, retraining. Retrains when there is new data. Weights & biases can display.
3. Start with a pull request, trigger GitHub actions, change in the model directory, package the files to the team server, run training, and send to W&A to display

Pull requests

To avoid code conflicts and collaborate more easily, we work on our tasks within a feature branch. We then create pull requests from this branch where we iteratively push changes and build out the feature we're working on. As mentioned previously, the pull requests also help us track the quality of our work (e.g., pass all unit tests, following coding formatting style). After we've coded the feature, we assign other teammates to conduct a code review. Once teammates review the changes, we merge our changes to the main branch, where it gets automatically deployed to production.

Examples	
Shanzid Shaiham	PR 35: Adding continuous integration for model testing
	PR 34: Adding capabilities for online evaluation

	PR 30: Adding strict unit test checks and code coverage thresholds
Kua Chen	PR 32: Adding tests for data quality, kafka consumer
	PR 9: Improving data collection process
	PR 28: Updating utility scripts, developing online evaluation
Barry Li	PR 19: Modularize the New Model Code
	PR 22: Model Unit Testing
	PR 33: Implementation of Continuous Integration
Marcos Souto Jr.	PR 18: 2 new models: retrieval and ranking (trained saved models)
	PR 23: Final notebook used as base to create the python modules
	PR 20: New inference code using the tensorflow models
Zeyu Li	

Reference:

1. “Mean Reciprocal Rank (MRR),” *Mean Reciprocal Rank (MRR) - Machine Learning Glossary*, 24-Dec-2017. [Online]. Available: https://machinelearning.wtf/terms/mean-reciprocal-rank-mrr/. [Accessed: 28-Oct-2022].
2. “Weights & Biases – developer tools for ML,” Weights & Biases – Developer tools for ML. [Online]. Available: https://wandb.ai/site. [Accessed: 28-Oct-2022].