Project Mile Stone 2

Tianye Song(tianyes), Udaikaran Singh(udaikars), Luís Gomes (lfgomes),
Liangwei Zhang (liangwez), Tushar Vatsa(tvatsa)

March 2022

1 Offline Evaluation

1.1 Offline Evaluation Pipeline

For offline evaluation, we constructed a pipeline similar to the previous milestone. We take a snapshot of the dataset and treat this as a static dataset. At the time of final evaluation, we had 828,355 ratings within our dataset. We believe that this is an appropriate amount of data to make an appropriate estimate of the quality of our model. Although we didn't go into the nitty-gritty of the dataset, we believe that there is enough ratings to build an effective estimate of the performance of our models.

We considered reducing the size of the training data by throwing out some of the earlier data. The motivation behind this choice is avoid the case of data drift. For example, if there is a statistical difference behind the distribution of rating data, then the earlier data actually becomes ineffective in creating an effective model. However, we felt that evaluating data drift was a step that is beyond the scope of this assignment, and it's inappropriate to throw out data when it is unnecessary. The link to the implementation of our data parsing code is: https://github.com/cmu-seai/group-project-s22-dsu/blob/master/ml/preprocess.py.

1.2 Training/Testing Subsets

For our offline training, we created train, validation, and test datasets from our (user, movie) matrix. We also performed data cleaning, where we filtered out users with few rating reocrds and movies with few number of ratings. We made the split along users. This means that we assign users to the groups of train, validation, and test. The mechanism for this that we made train-test splits on the whole dataset, then allowed for the creation of a validation set using cross validation. In our case, we assigned 20% of the data as the test set, which we keep seperate during training. For the validation, we use k-fold cross validation. This separates the training data into k random partitions, using k-1 parts for training and evaluating on the left out partition. In our code, we use k = 3 as the number of folds in our k-fold cross validation.

The goal of k-fold cross validation (and generally for the training and validation dataset) is for the purpose of hyper-parameter tuning. The hyperparameters we used for K-NN was the k parameter, the number of epochs, and distance metric. For Matrix factorization, the huperparameters we tuned are the size of the latent space (k), the regularization parameter, and the number of epochs. The goal of this hyperparameter tuning is to find the best model on values that cannot be learned directly from a computationally tractable optimization algorithm. The validation dataset is used as a proxy to unseen data.

The link to the implementation of our training/testing on SVD an be found at: https://github.com/cmu-seai/group-project-s22-dsu/blob/master/ml/KNN.pyL85-L89.

1.3 Metrics

The metrics we used for our evaluation are: MSE, MAE, and RMSE. The metrics are defined below as:

$$MSE(r, r') = \frac{1}{|R|} \sum_{r'_{ui} \in R} (r'_{ui} - r_{ui})^2$$

$$MAE(r, r') = \frac{1}{|R|} \sum_{r'_{ui} \in R} |r'_{ui} - r_{ui}|$$

$$RMSE(r, r') = \sqrt{\frac{1}{|R|} \sum_{r'_{ui} \in R} (r'_{ui} - r_{ui})^2}$$

Because we fundamentally doing a regression problem of predicting the rating a user would give to an unseen movie, these metrics can be used to measure the difference between the predicted rating and ground-truth rating. All of these are norms, but you can see that the absolute value (MAE) formula does not punish large errors as much as MSE and RMSE. The MSE and RMSE are highly correlated, however RMSE is very useless due to its statistical interpretation.

The link to the implementation of calculating the offline metrics by SVD and KNN respectively are: https://github.com/cmu-seai/group-project-s22-dsu/blob/master/ml/cf$_svd.pyL87-L90$ and https://github.com/cmu-seai/group-project-s22-dsu/blob/master/ml/KNN.pyL34-L38.

Metric	K-NN	SVD
MAE	0.587847	0.521996
MSE	0.589984	0.467402
RMSE	0.76810	0.683667

2 Online Evaluation

We measured three qualities for this recommendation system. In this section we present those qualities, the respective metrics and the way we operationalized them. The telemetry data collected is real time based. In this milestone, we collect the timestamp of dates in the latest day, the latest week, and the latest month. The link to the implementation of telemetry data collection is the following: https://github.com/cmu-seai/group-project-s22-dsu/blob/master/ml/online$_e$*valuation.py*$L129 - L141$.

The operationalization of the metric includes evaluating any single date, and the overall average evaluation results. This implementation can monitor both the average metrics over a time interval as well as the metric in one day so that abnormal patterns can be detected. The link to the implementation of operationalization is the following: https://github.com/cmu-seai/group-project-s22-dsu/blob/master/ml/online$_e$*valuation.py*$L46-L127$.

2.1 Prediction Accuracy

For accuracy, we define that the recommendation is correct if the user has watched the movie and rated it. We calculate two aspects of accuracy. The first one denotes the record accuracy, which is obtained by dividing the number of recommended movies users have watched by the total number of movies users have watched. The link to the implementation of this metric is the following: https://github.com/cmu-seai/group-project-s22-dsu/blob/master/ml/online$_e$*valuation.py*$L36 - L37$.

$$\text{Record Accuracy} = \frac{\text{No. recommended movies watched}}{\text{No. movies users have watched}}$$

The second one is the recommendation quality, which denotes the ratio of users watching the recommended movies on the total number of users requesting. recommendations. The link to the implementation of this metric is the following: https://github.com/cmu-seai/group-project-s22-dsu/blob/master/ml/online$_e$*valuation.py*$L38-L39$.

$$\text{Recommendation Accuracy} = \frac{\text{No. users watching the recommended movies}}{\text{Total Number of users requesting. recommendations}}$$

2.2 Average Rating Score

We measure the quality of the recommendation by calculating the rating scores of those movies. We evaluate the average rating of those recommended movies which have been watched and rated by users.

$$\text{Average Rating Score} = \frac{\text{Total rating scores for recommended movies watched}}{\text{No. recommended movies users have watched}}$$

The link to the implementation of this metric is the following: https://github.com/cmu-seai/group-project-s22-dsu/blob/master/ml/online$_e$*valuation.py*$L40 - L41$.

2.3 Average Top Movie Rating Score

We also focus on the evaluation of our top recommendation, which is vital for users' impression on our recommendation system. Therefore, we also calculate the average rating for the top recommendations.

$$\text{Average Top Movie Rating Score} = \frac{\text{Total rating scores for the top recommended movies watched}}{\text{No. top recommended movies users have watched}}$$

The link to the implementation of this metric is the following: https://github.com/cmu-seai/group-project-s22-dsu/blob/master/ml/online$_e$*valuation.py*$L42 - L43$.

2.4 Result

We collect the latest week from March 24 to March 30. We collect the history records (users and their recommendations during this period) as telemetry for online evaluation. We present the result in the following tables, where the "No.Users" denotes the number of users who request recommendation on that day, "Total Records" for the number of movies those users have rated and watched, "Total Recommendation" for the number of recommendation movies for those users, and "Correct Recommendation" denotes the number of recommended movies watched by that user.

Date	Correct Recommendations	Total Records	Users watched Recommendation	Total users
2022-03-30	2243	10585	1832	8472
2022-03-29	5867	28562	4777	22554
2022-03-28	7168	33852	5810	25227
2022-03-27	4563	21549	3708	16222
2022-03-26	3068	14410	2492	10731
2022-03-25	2990	14326	2423	10628
2022-03-24	3186	14971	2544	10861
Summary	29085	138255	23586	104695

Date	Record Accuracy	Recommendation Accuracy	Average Rating	Average Top Movie Rating
2022-03-30	21.19%	21.62%	3.8529	3.9130
2022-03-29	20.54%	21.18%	3.8822	3.9577
2022-03-28	21.17%	23.03%	3.8626	3.8482
2022-03-27	21.17%	22.86%	3.8847	3.9468
2022-03-26	21.29%	23.22%	3.8791	3.8627
2022-03-25	20.87%	22.80%	3.8458	3.8275
2022-03-24	21.28%	23.42%	3.855	3.8351
Summary	21.04%	22.53%	3.8685	3.8877

3 Data Quality

For data quality, there are two parts of data our group pay attention to, the first part is cleaning the model input data which include the saved data in the database and the incoming future data from Kafka and Public API, and the other part is validating the service request to the movie recommendation API.

3.1 Model input data

Issues detected and handled at the data cleaning part include:

- Fixed data schema issue. If there are required feature columns missing, the cleaning part will add these columns and fill them with null value. https://github.com/cmu-seai/group-project-s22-dsu/blob/master/data$_quality/dL40$.
- Dealt with duplicate data issue. At this stage, we update the original data when there comes a duplicate record and only save the latest version. https://github.com/cmu-seai/group-project-s22-dsu/blob/master/mongodb/db.pyL8-L45.
- Fixed missing data. At this stage, we fill these data with null value to prevent errors in the model training part. https://github.com/cmu-seai/group-project-s22-dsu/blob/master/data$_quality/db_quality_check.pyL59-L73$.
- Deleted redundant white space for string data at the beginning and at the ending of the string. https://github.com/cmu-seai/group-project-s22-dsu/blob/master/data$_quality/db_quality_check.pyL42-L56$.
- Checked integer feature data range and data type. For example, the age feature should rage from 0 to 100, the rating feature should range from 1 to 5, the user id should be a series of digit. Besides, the data type of these features could be string, float or null value in the incoming data and the prepossessing code will transform these wrong type to integer or null value. https://github.com/cmu-seai/group-project-s22-dsu/blob/master/data$_quality/db_quality_check.pyL102 - L178$.
- Transferred string type date data to date-time type. For this part, there might be the case that the month is bigger than 12 and day is smaller than 12, in this case, month and day will swap. https://github.com/cmu-seai/group-project-s22-dsu/blob/master/data$_quality/db_quality_check.pyL75-L100$.

3.2 Service request

The API service provide recommendation based on the user id provided. The user id should be a series of digits for the system to work; therefore, before passing the user id to the service, quality check is needed for the user id.

When the user id contains characters other than digits, for example letters or punctuation, or the user id is null, a reminder requesting correct format user id will be sent back.https://github.com/cmu-seai/group-project-s22-dsu/blob/master/data$_quality/api_request_check.pyL3 - L19$.

4 Pipeline implementation and testing

4.1 Pipeline implementation

The infrastructure of the whole recommendation system is briefly described in Figure 1.

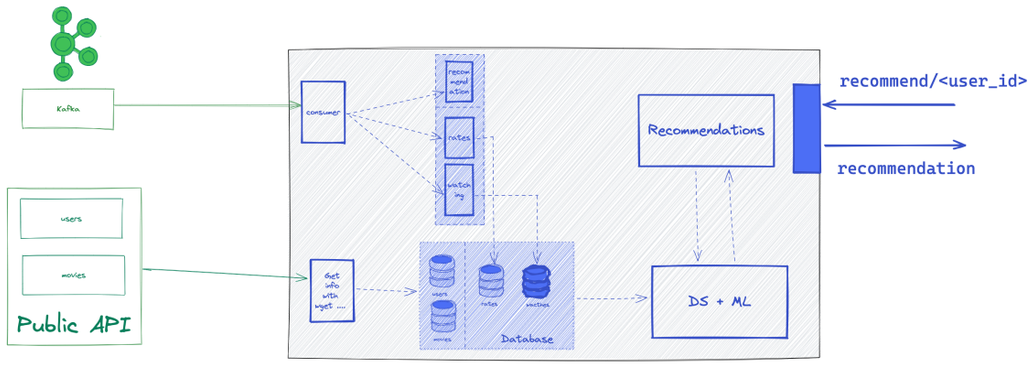


Figure 1: System Infrastructure

In the prediction phase, for each user, we predict the ratings for all movies and then rank them by the predicted ratings. Then we choose the top 20 movies by its predicted score. If the user has never rated any movie before, which is a cold-start problem, we randomly return 20 of the top-50 popular movies. Popular movies are chosen by the vote average ratings among those most voted movies, which takes both most viewed and high quality into consideration.

Regarding the prediction service implementation, we pre-compute the recommended result for all users stored, and store the result in the database. The recommendations manager is responsible for loading the model on startup and use the predictions every time a new request is made. Request are handled by the Flask API on port 8082.

To save metrics and continuously acquire data to improve the models, our system saves all past recommendation history and new users and movies (acquired from the Kafka broker).

The Machine Learning component presented in Figure 1 is composed from 3 major steps: Prepossessing data, where we use a *pandas* DataFrame with clean data from the database (see Data Quality section); Train the model with cleaned data (See Milestone1); Offline evaluation, online evaluation and calculate the metrics(see Online evaluation and Offline evaluation section).

The links to the implementation of each part of the system involved on the recommendation service are the following:

- *Flask API*: https://github.com/cmu-seai/group-project-s22-dsu/blob/master/flask/api.py;
- *Recommendations Manager*: https://github.com/cmu-seai/group-project-s22-dsu/blob/knnmodel/recom_manager.py.
- *DB Manager*: https://github.com/cmu-seai/group-project-s22-dsu/blob/knnmodel/db_manager.py
- *Model*: https://github.com/cmu-seai/group-project-s22-dsu/tree/master/ml

4.2 Tests

The testing phase was divided into two main parts : Infrastructure testing and Model testing. All the tests can be found in the tests suite on our github repository https://github.com/cmu-seai/group-project-s22-dsu/tree/master/tests.

4.2.1 Infrastructure Testing

- Kafka Test : We want to test our kafka consumer to check if it is able to consume all the messages produced by the producer. We create a mock producer which inserts a number value into the topic which is later consumed by the consumer. A report is generated which keeps count of the number of records to insert, inserted and consumed. During the reading process, we increment our variable to know how many messages have been read. This test makes sure that the connection between kafka broker and consumer is established and all the messages are being consumed in an order. https://github.com/cmu-seai/group-project-s22-dsu/blob/master/tests/kafka$_test.py$.
- Flask Test : We want to make sure that our API is able to receive the messages sent by the producer. We recommend the movies based on the user id provided. Here, user id is supposed to be an integer. So, we conduct tests to make sure that our recommend function is working properly. We take similar values like 0, 00, 000, 0000 and test to see if the response being produced by the recommendation system is the same. We also test with non-integer values and expect the recommendation system to return a None value. This test works as a unit test as well as an Integration test because it tests the connectivity of our API and also the flow of output from our recommendation system to the public API. It also makes sure that our recommendation system is working properly https://github.com/cmu-seai/group-project-s22-dsu/blob/master/tests/flask$_test.py$.

- Database Test : We want to make sure the values being stored in our database can be retrieved properly. Here, we are using two different databases to store information about the user(i.e. users collection) and movies(i.e. movies collection). We tried with mongo mock to see if the data related to the movie and user can be retrieved. We also tried putting a few random data points in the database and retrieving those values. This is not the best way to do it but based on the amount of data that we have, we think it will not harm the performance of our recommendation model. This test also makes sure that the connection to mongodb server is established properly https://github.com/cmu-seai/group-project-s22-dsu/blob/master/tests/mongo$_test.py$.
- Query Test :This test is to make sure that our parser works fine and our API is able to return recommendations based on the input user id. We test two functions here : getmovie and getuser. https://github.com/cmu-seai/group-project-s22-dsu/blob/master/tests/query$_test.py$.

4.2.2 Model testing

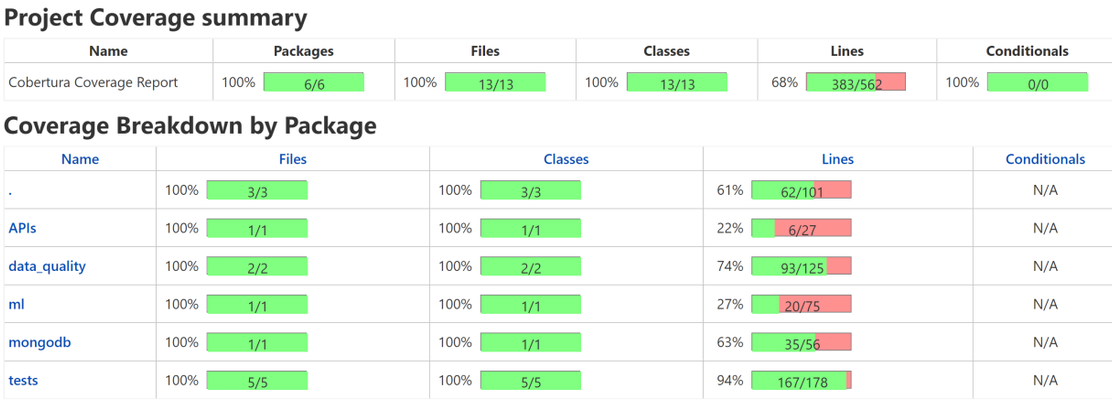
- Model Test : The test ensures that our model is able to produce recommendations for different users. Here we use two different training sets, one with 6 times more training data than the other. The test is to make sure that the root mean square error, mean square error, mean absolute error are less for the model which uses more training data. This ensures that our training part is being correctly done and also the metrics used are reasonable. We also tested with the training time and it should be more for the model with a bigger training data set. https://github.com/cmu-seai/group-project-s22-dsu/blob/master/tests/model$_test.py$.

Ideally we would like to have 100% coverage, but any increase is a good one. We do realize, though, that getting 100% coverage is not always possible. There could be platform-specific code that simply will not execute for us, errors in the output, etc. We can use our judgement as to what should and should not be covered, but being conservative and assuming something should be covered is generally a good rule to follow. Here, we tried to test our individual components which make sure that all these units are fully functional in a black box. We also used integration testing to make sure that the different units are able to work together and are able to produce meaningful results altogether. Network testing was also a part of our tests suite to make sure that the Kafka server and MongoDB server are connected.
The coverage of the unit tests, integration tests and network test is good and therefore we conclude that the testing was sufficient.

5 Continuous integration

Regarding continuous integration we use Jenkins service for automating the steps of the learning pipeline and testing the our infrastructure. Every time that there are code or data changes on Github, a new build is triggered by a Webhook. Only changes from the main branch (master) trigger a new build. In every build, the service reruns the learning pipeline, training and evaluating a new model.

The unit test suite is also executed for every build. In this case, a coverage report is created on Jenkins by the *cobertura* plugin, that uses the *coverage.xml* file generated from *pytest*. An example of the test coverage is shown in Figure 2.



Project Coverage summary

Name	Packages	Files	Classes	Lines	Conditionals
Cobertura Coverage Report	100% 6/6	100% 13/13	100% 13/13	68% 383/562	100% 0/0

Coverage Breakdown by Package

Name	Files	Classes	Lines	Conditionals
.	100% 3/3	100% 3/3	61% 62/101	N/A
APIs	100% 1/1	100% 1/1	22% 6/27	N/A
data_quality	100% 2/2	100% 2/2	74% 93/125	N/A
ml	100% 1/1	100% 1/1	27% 20/75	N/A
mongodb	100% 1/1	100% 1/1	63% 35/56	N/A
tests	100% 5/5	100% 5/5	94% 167/178	N/A

Figure 2: Code coverage report example.

Our continuous integration pipeline has 3 stages:

- **Install dependencies:** Install Python libraries that are needed.
- **Run Learning Pipeline:** Execute the learning pipeline, including data preprocessing, model training and offline evaluation.
- **Test Infrastructure Code:** Run the tests for the components of our architecture and create the coverage report.

The CI pipeline is defined through the Jenkins file at: https://github.com/cmu-seai/group-project-s22-dsu/blob/master/jenkins/Jenkinsfile.

To access the Jenkins service at the machine, please use the following URL using a tunnel from the machine: http://localhost:8080/job/production_build/

The credential to access Jenkins are the following:
user: lfgomes
password: vWi4L2Hi253N7o