Yelp Recommender System

Benjamin Wang bwang4@scu.edu
Frederick Tan ftan2@scu.edu
Jianqiao Ge jge2@scu.edu
Yunyan Shi yshi3@scu.edu

June 8, 2022

Abstract

Recommendation systems have become an integral part of the online experience. This is especially true today as many online platforms such as YouTube and Amazon have applied recommendation systems to their products and services in order to enhance user experience and generate more income. Recommendation systems provide users with a list of objects based on what they have clicked on or selected in the past. Given the prevalence of these systems on the internet, for this project, we decided to build our own recommendation system based on a Yelp dataset, which is publicly available. The aim of our project is to better suit customer needs of Yelp users and therefore enhance their user experience while using cloud resources to deploy our project. The outcome of this project is a web application that randomly selects a user from the dataset and returns a list of recommended businesses based on previous ratings that user had given on Yelp.

1 Introduction

Society is at a place now where the amount of information available to the population far exceeds the actual population. The advent of the Internet has given way to this phenomenon. A side effect of this is the fact that too much information can become overwhelming to the human brain. If one wanted to buy a pair of shoes, they can go to either platform x, y or z, where each platform may have hundreds or even thousands of choices, each with its own set of reasons to persuade the individual to purchase said shoes. Or one could be looking for a new show to watch, but again thanks to the internet, more shows have become more accessible leading to the tough decision of what show to watch given the limited amount of time one has in a day. Thus, the problem is how do we filter all this information to arrive at the information we want?

Enter recommendation systems, information filtering systems that sift through the information overload and come up with a small fraction of that amount according to user preferences, interest or observed behaviour. Recommendation systems are beneficial to both the provider and the user by enhancing revenue for the provider, and speeding up and improving the decision-making process for the user. Nowadays, major online platforms employ some kind of recommendation system, whether it is through collaborative filtering, content-based filtering or a hybrid of techniques. In this project, we construct our own recommendation system based on a dataset supplied by Yelp, using prediction algorithms from the Surprise library and hosting our application on Amazon Web Services (AWS). More details on how each piece is setup and works is given in the later sections. Like with many platforms on the internet, the purpose of our project is to enhance the user experience on Yelp by supplying them with a list of recommendations based on that user's rating history. On a personal level, this project also gives us the opportunity to learn the ins and outs of the recommendation system and how cloud computing concepts can assist in creating an efficient and scalable application that brings many societal benefits in the digital information age.

2 Background and Related work

As recommender systems have taken more and more place in our lives and are becoming unavoidable in our online experience, the techniques used in developing recommender systems have gained much

popularity. There are two major paradigms of recommender systems: collaborative filtering (CF) and content-based filtering. A lot of research has been done on both paradigms but we would like to expand further on the first one due to the fact that collaborative filtering builds a model based on users' past behavior from which we can more easily collect data. There are a variety of prediction methods that falls into the category of collaborative filtering recommender systems, like Singular Value Decomposition (SVD), K-Nearest Neighbors (KNN), Probabilistic Matrix Factorization (PMF), Non-Negative Matrix Factorization (NMF), etc. Since each algorithm has its own practical tradeoffs, a library that provides various ready-to-use CF algorithms as well as the tools to evaluate, analyze and compare the algorithms' performance would meet our requirements of implementing a recommender system. Surprise is one of such kind of libraries and we decided to integrate it in our project.

In addition to the prediction algorithms, we also need data of users' past behavior, for example, previously rated movies, restaurants, purchased items, etc. And this is where Yelp dataset comes into play. Yelp dataset is a sub collection of Yelp's data and can only be used for personal, educational and academic purposes. The dataset contains information about users, businesses and reviews, meaning with the dataset we would be able to view users' past ratings on business and present them with the recommendations of further business to engage with.

Last but not least, we need to choose a Cloud platform to host our application. There are many hosting providers and each of them may offer multiple types of hosting. We decided to focus on some of the best infrastructure-as-a-service (IaaS) platforms, which essentially provide on-demand virtual services in more flexible cost. Finally, we determined that Amazon Web Services (AWS) would be with us for our application development as it has been named the No.1 leader of Iaas providers for years. And more importantly, the wide variety of services AWS offers can cover our needs in relational databases, virtual computers and enabling front ends.

In summary of the background information, this project is intended to design and implement a recommender system given that recommendation systems are almost everywhere in our everyday lives. We would like to learn about the recommendation models along with the dataset we will experiment with. We also plan to make our application accessible using cloud resources as many businesses are migrating to cloud than ever before. We hope this project can offer us a great learning opportunity about the basics of recommender systems and cloud computing, and inspire us to deploy a scalable recommender system over a real-time big data architecture in the future.

Some related works that are similar to our project but not referenced by us during the developmental phase of this project include Sawant's team's similar work on a "Yelp food recommendation system" [SP13], with the difference in that Sawant's project focused on a detailed examination and evaluation of different prediction algorithms while ours focused on utilizing cloud technology to deploy a basic functioning recommendation system. There are also more detailed recommender systems based on the Yelp Open Dataset that utilizes cloud technology even more than our project did, such as the Capstone_Yelp project by Shi and his team that is far more technical in nature but also uses Amazon RDS and Flask framework [SOK+17], and demonstration project by Tindel on making a CF recommendation system using Yelp dataset on Amazon Neptune [Tin00]. One common factor among these two more technical recommendation system projects based on the cloud is that they were built for research and do not provide a graphic interface on the frontend for visitors to interact with. This is an improvement our project has made over these three related works since they all use command-line outputs and we provide a nice web app frontend GUI.

3 Approach

3.1 System Architecture

We approached building our project by dividing it up into three components: Frontend, Backend, and Database. As seen in the high-level architectural diagram below in figure 1, each of these component uses a different piece of cloud technology to host it.

Users interacts graphically with the web application Frontend via a web browser at `https://main.d1jqb41q7u1nka.amplifyapp.com/`. This frontend is hosted on AWS Amplify and its code is contained within its own git repository. Frontend then connects to the Flask app Backend hosted on an Amazon EC2, which queries the MySQL database hosted on Amazon RDS.

In terms of inter-component communication, the frontend communicates with the backend using

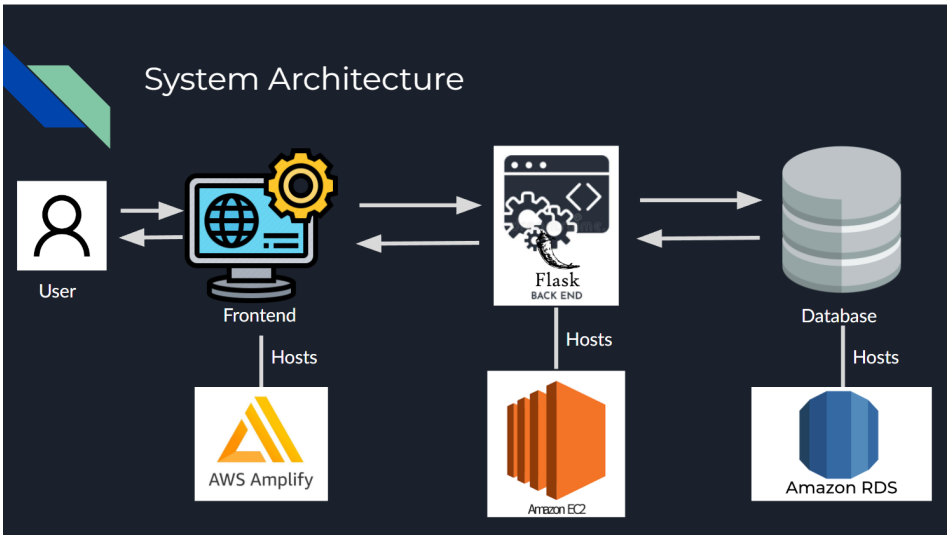

Figure 1: High-Level System Architecture Diagram

HTTP Requests via RESTful API, with the API defined and handled in the backend, and the backend communicates with the database using SQL queries through the python driver for MySQL Connector. When the frontend has a request for the database, such as fetching the photo-id of a business by its business-id, the backend acts as the intermediary between them: The frontend request is received by the backend which then reformats the request as a SQL query before sending the query to the database. The backend then receives query results and reformat that result as JSON object before sending it back to the frontend, completing the process.

For the cloud resources utilized, we are using the free versions of AWS Amplify and Amazon RDS for MySQL. One limitation that arose from using the free version of Amazon RDS for MySQL is that our database is a truncated version of the original Yelp Open Dataset due to storage constraints [Yel04]. Similarly, due to the initial target of live update to the database and live training of the recommender during the demo, we cannot use the free version of EC2 with its insufficient memory for training the recommendation algorithm. Instead, we are using a paid t2.small instance of Amazon EC2 with 2 CPU cores, the 2GB of memory required for training our recommendation algorithm and 30GiB of storage for storing the pictures of businesses. This EC2 instance is "Anaconda with Python 3 (x86_64)", the "Official Anaconda Distribution AMI (x86_64), published by Anaconda, Inc" as it describes itself on AWS Marketplace, which conveniently provides a relatively up-to-date Anaconda environment for Python development without needing to install most of the dependencies for the libraries used.

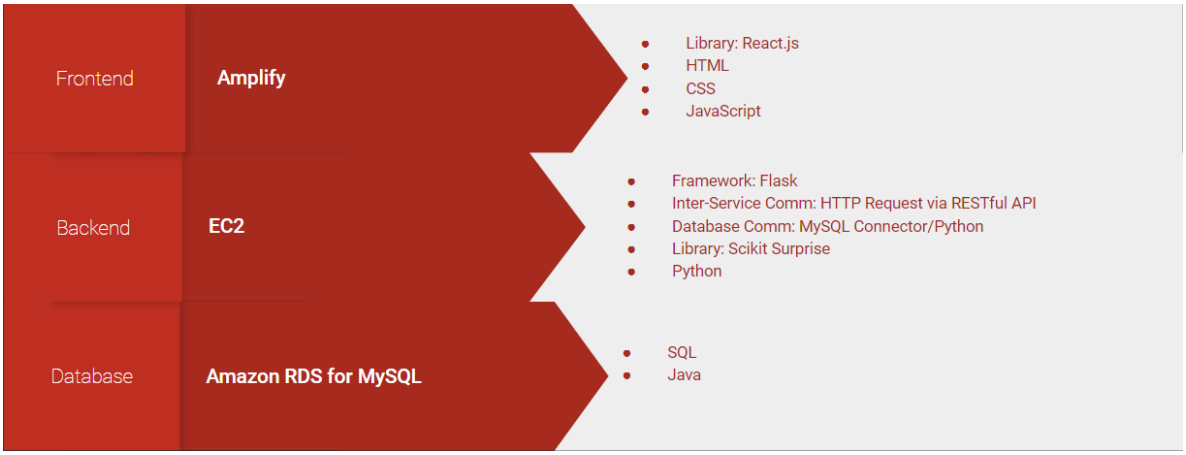

Figure 2: Technology Stack for the Frontend, Backend, and Database

More specific details about the different programming languages, frameworks, and libraries used in our project can be found in the tech stack in figure 2. As seen in the figure, the web application frontend hosted on AWS Amplify is written in JavaScript using the React.js library with HTML and CSS, the Flask app backend hosted on Amazon EC2 is written in Python using the Flask framework and Surprise recommender system library, and the MySQL database is hosted on Amazon RDS for MySQL where the original dataset was loaded into the database using Java.

3.2 Software Architecture

Expanding on the high-level architectural overview in the previous section, the detailed overall workflow of the entire project can be seen in figure 3

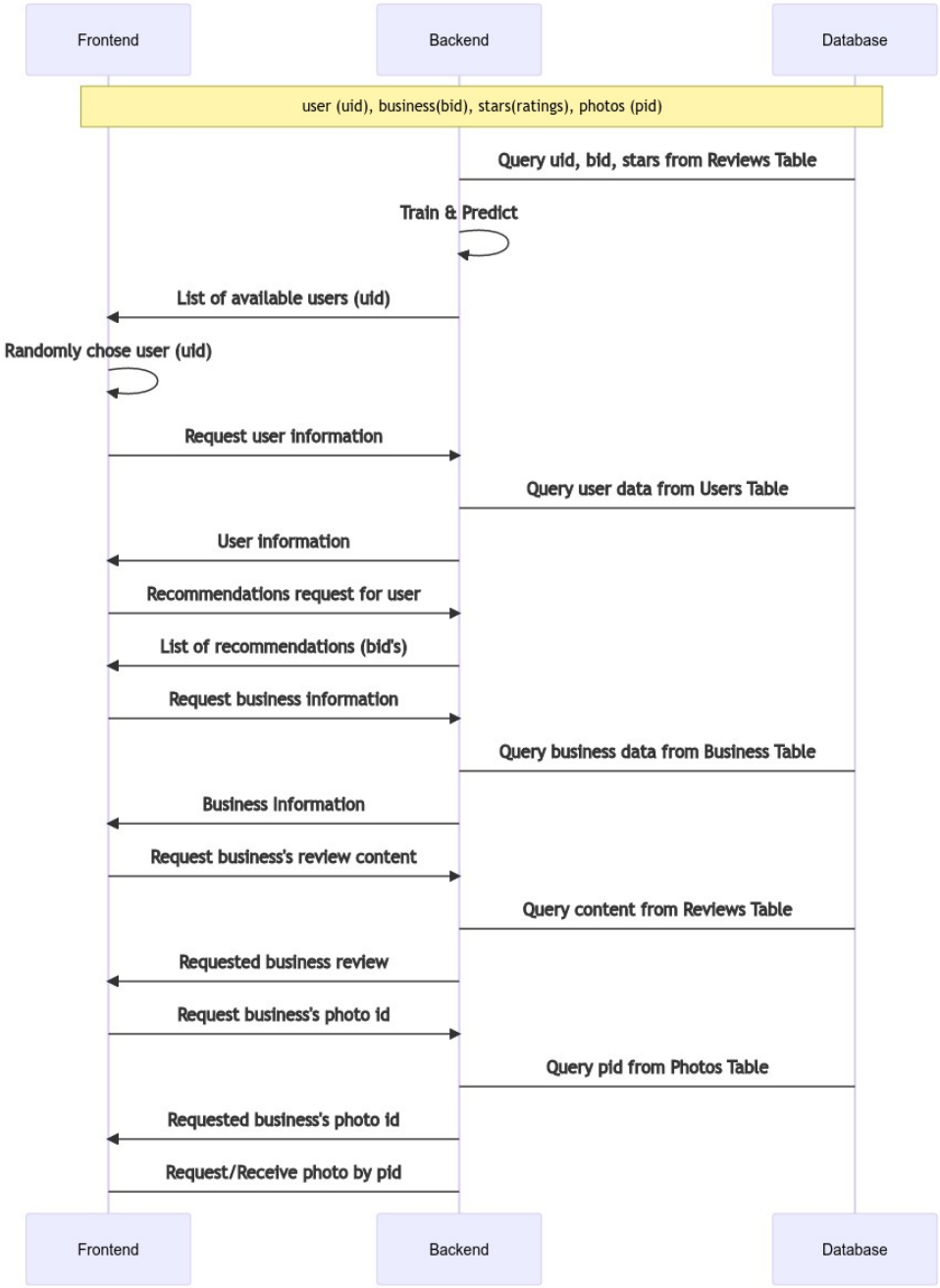

Figure 3: Workflow of the project

The frontend keeps a list of test users that the Flask App backend has prepared predictions for. When a user visits the web app, the frontend randomly chooses a test user from the list in the sense that we pretend the visiting user is the chosen test user and present the recommendations accordingly. The frontend makes a total of four requests to the databases that uses the backend as the intermediary, and two requests to the backend itself. The four requests to the database is to request user information, business information, review content, and photo-id of a business, while the two requests to the backend itself are to request for the list of recommendations for a specified user and retrieve the photo by photo-id as the photos are stored on the EC2 instances instead of the database due to reaching the storage limitation of the free version of Amazon RDS for MySQL.

Before the visiting user interacts with the frontend, however, the backend first makes a query to the Reviews table of the MySQL database for all available user-id, business-id, and stars (ratings), the three fields required by the Reader object of the Surprise library that parses data. We then split the dataset into training-set and testing-set, then use the fit() function of the Surprise library to train the algorithm on the training-set and test() function to make predictions on the testing-set. The result of training and prediction are saved locally on the EC2 so the Flask App can be configured to read from previously trained model or prediction. After the predictions has been made, we then generate the list of top-10 recommendations with the highest predicted ratings for each test-user, which is then again saved locally. Thus when the frontend makes a request for the list of recommendations for a particular user, we check if training, prediction, and top-10 recommendations generation has already been preformed before, and read from the saved results instead of re-doing the process again. Besides acting as the intermediary between the frontend and the backend as mentioned before, the backend also stores the photos for the businesses with the naming convention of "photo-id.png" and returns the requested photos by photo-id to the frontend.

Prior to the backend querying the reviews dataset from the database, however, the database was first defined in the "createdb.sql" SQL file in our main GitHub repository's root directory, illustrated below in figure 4. The original Yelp Open Dataset [Yel04] is then loaded into the MySQL database hosted on Amazon RDS using a Java program (/DatabaseManager/src/Main.java in main GitHub repository) that connects to the Amazon RDS instance and parse the original dataset stored as lists of JSON objects, and populate the corresponding tables.

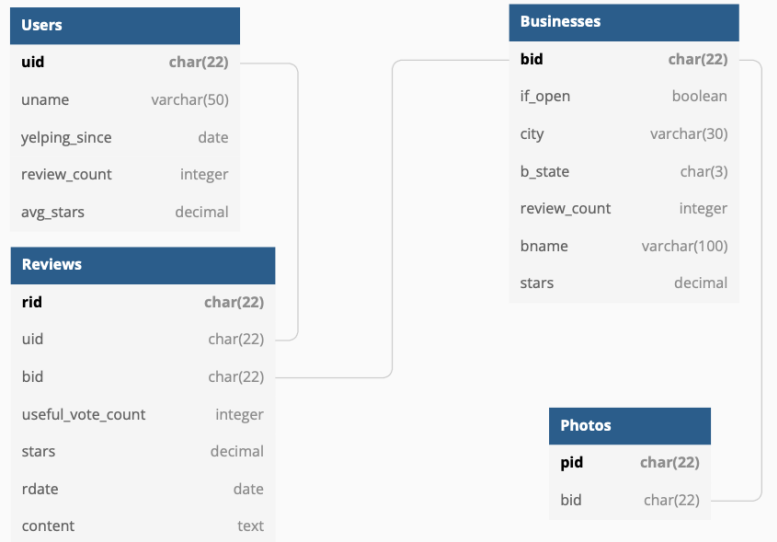

Figure 4: Database Schema

As can be seen in figure 4, we are only using a subset of the original Yelp Open Dataset, which contains more non-relevant information [Yel04]. Also shown in the figure with lines are the three foreign keys uid, bid, and pid that are foreign keys that are set to have entries in the child tables deleted automatically if deleted in the parent table.

3.3 Algorithm Development

Surprise is a Python scikit library that describes itself as the library for "building and analyzing recommender systems that deal with explicit rating data" [Hug20]. We chose it because it provides many ready-to-use prediction algorithms and standard evaluation methods as well as handlers for datasets such as the previously mentioned Reader object that parses data. There are also the Dataset object that loads in the dataset and use the Reader object specified by the user, and train-test-split function that split the dataset. We use these three functions together to import and prepare the dataset for training and prediction. During our development phase before we had the python driver for MySQL Connector set up we used a local copy of the dataset to test different prediction algorithms, then later modified the driver code of the recommender system to be configurable as to whether to use the dataset stored locally or retrieved through SQL query to the MySQL database. Luckily, Surprise provides ready-made methods in its Dataset object for both cases.

During initial development of the recommender system we used a Python notebook (/Recommender/surprise-recommender.ipynb of our main GitHub repository) for easier experimentation with different functions provided by the Surprise library. As can be seen in the Python notebook, we tested most prediction functions provided by the library, including basic algorithms like NormalPredictor, BaselineOnly, neighborhood algorithms like KNNBasic, KNNWithMeans, KNNWithZScore, and KNNBaseline, matrix factorization-based algorithms like SVD, SVD++, and NMF, as well as other collaborative filtering algorithms like SlopeOne and CoClustering. Unfortunately due to the unoptimized nature of the Surprise library for lower level functions like training and prediction, specifically its lack of support for multithreading, caching, algorithm optimization, and sharding, we were unable to test the SlopeOne and any KNN neighborhood-based prediction algorithms due to SlopeOne requiring 167 GiB of memory for training and KNN neighborhood algorithms requiring a staggering 12.5 TiB of memory that is quite unreasonable considering our original dataset only consist of a couple GB of data. Results of testing the prediction algorithms we were able to run on our PC are presented below in figure 5.

	Algorithms	test_rmse	fit_time	test_time
0	SVD	1.3025332424607239	141.97499787807465	22.80950176715851
1	SVDpp	1.3101656108009496	993.4315007925034	77.74049925804138
2	NMF	1.4911652176123789	218.62451767921448	20.68850016593933
3	NormalPredictor	1.9015793378418226	2.453497886657715	19.583502888679504
4	BaselineOnly	1.3044595390621168	3.8379987478256226	18.20549964904785
5	CoClustering	1.437949887236462	113.82850813865662	17.987001180648804

Figure 5: Testing results for predictions algorithms we were able to run

To elaborate more on figure 5, the test-rmse is the root-mean-square-error (RMSE) score of the predictions made on the testing-set, fit-time is how long it took in seconds for the prediction algorithms to be trained (fit()) and test-time is how long it take for the algorithms to make predictions on the testing-set (test()). These results are the averages of 5 iterations of re-splitting the dataset, training the algorithms, and making predictions. As we can see, the RMSE scores for SVD, SVDpp, and BaselineOnly outperforms all other prediction algorithms tested. Of these three prediction algorithms, BaselineOnly is the only one with a training time short enough to be considered using dynamically on request with reasonable time for making predictions. Considering that our initial target for the demo is to have a live demo of updating the database and re-training the algorithm and re-doing the predictions consequentially, we chose to use the BaselineOnly algorithm as it comes second in all three categories of RMSE, training-time, and prediction-time without any major drawbacks in any of the categories tested. We then optimized the hyperparameter of the BaselineOnly prediction algorithm by calling on the GridSearchCV function provided by the Surprise library that takes in a range of available hyperparameter options and test all combinations to find the best hyperparameter.

Implementation-wise, the BaselineOnly prediction algorithm in the Surprise library is based off of section 2.1 Baseline Estimates of the Koren research paper [Kor10] where Koren defines the algorithm as figures 6, 7, 8, and 9.

To summarize, the Baseline Estimates algorithm computes the average rating of all items, or businesses in our case, and estimate the observed deviation, or how one tends to rate or be rated on

$$b_{ui} = \mu + b_u + b_i$$

Figure 6: Definition of BaselineOnly algorithm

$$b_u = \frac{\sum_{i:(u,i)\in\mathcal{K}}(r_{ui} - \mu - b_i)}{\lambda_3 + |\{i|(u,i) \in \mathcal{K}\}|}$$

Figure 7: Definition of the observed deviation of user

$$b_i = \frac{\sum_{u:(u,i)\in\mathcal{K}}(r_{ui} - \mu)}{\lambda_2 + |\{u|(u,i) \in \mathcal{K}\}|}$$

Figure 8: Observed deviation of item

average, for each user and business. Finally, it puts the average rating together with the deviations for the user and business to predict how a user would rate a business.

$$\min_{b_*} \sum_{(u,i)\in\mathcal{K}} (r_{ui} - \mu - b_u - b_i)^2 + \lambda_1 \left(\sum_u b_u^2 + \sum_i b_i^2 \right)$$

Figure 9: Coupled form of the deviations estimation formulas in figures 8 and 7

As implemented in the Surprise library [Hug20], the BaselineOnly algorithm uses a coupled form of the deviation formulas in figure 9 where it tries to minimize the regularized square error to solve for both deviations while using regularization to "avoid overfitting by penalizing the magnitude of the parameters" [Kor10]. Surprise provides two ways to estimate the Baselines through Stochastic Gradient Descent (SGD) and Alternating Least Squares (ALS), and through our hyperparameter optimization with the GridSearchCV mentioned earlier, we know that we are using SGD in the final configuration. SGD has several hyperparameter options, defined in figure 10.

- `'reg'`: The regularization parameter of the cost function that is optimized, corresponding to λ_1 in [Koren:2010]. Default is `0.02`.
- `'learning_rate'`: The learning rate of SGD, corresponding to γ in [Koren:2010]. Default is `0.005`.
- `'n_epochs'`: The number of iteration of the SGD procedure. Default is 20.

Figure 10: Hyperparameter options for SGD estimation of BaselineOnly prediction algorithms

After going through the lengthy hyperparameter optimization process using GridSearchCV, we landed on the following hyperparameters in figure 11 that provides the best RMSE score for all tested configurations

3.4 Codes referenced and our contributions

Our frontend is written entirely by Frederick Tan and does not use any code written by someone else other than using the React.js library. We were unfortunately not able to include the frontend

```
{'bsl_options': {'method': 'sgd', 'reg': 0.02, 'learning_rate': 0.01, 'n_epochs': 20}}
```

Figure 11: Optimal hyperparameters found through exhaustive grid search of configuration combinations

repository in our main backend and database GitHub repository due to issues with git and being unable to properly import the frontend library.

For our backend, the development process of choosing the prediction algorithm and optimizing its hyperparameter was done solely by Benjamin Wang in the Python notebook using usage examples provided in the documentation of the Surprise library. The code snippet that loops through all available prediction algorithms and run cross-validation on each of them, saving the results locally in case of crashes, and reading back previous results from local storage after crashes does not reference any code and is written entirely by Benjamin Wang.

For the final backend Flask App, the main() function, also known as the /recommend API function, preparation and load-settings functions are written by Benjamin Wang, the get-top-n function is code snippet from usage example provided in the Surprise library, the read-sql-into-dataframe function as well as the code snippet in other functions for connecting to the database is written by Yunyan Shi, and all other functions and API as well as modifying the original python script into a Flask App using the Flask framework are written and done by Jianqiao Ge. It should be noted that while these functions on the backend are each primarily written by a single person, we did debug the backend collaboratively as a team during the integration phase of our project so some credits should be shared universally.

The database, including both the SQL definition of the MySQL database's tables, and the Java code for populating the MySQL database with the original Yelp dataset, is written entirely by Yunyan Shi without using any external codes.

Repository links to our implementation:

- Frontend: https://github.com/ftan95/yelp-recommend-frontend
- Backend & Database: https://github.com/yunyanshi/YelpPersonalizedRecommendationEngine

4 Outcome

The final outcome of our project is that we have a web app accessible by visitors with a GUI interface that allow the visitor to click on the "Generate Recommendations" button and be presented with the user information of the test user we randomly chose from the testing-set to represent the visitor, list of recommended business, their business information and reviews, and a photo for each business. The frontend is hosted on AWS Amplify, and the application can be accessed by the public.

We run the Flask app backend on the EC2 as a detached process in the background using the screen utility tool so that we can close ssh connections to the EC2 instance without stopping the Flask app backend. When the /recommend API, or main() function, is called, it first reads in the configurations for the recommender saved in /Recommender/recommender-settings.json to know whether or not to do prediction, training, top-10-recommendations generation, load from database or local copy of dataset, and where the internal data like previously done prediction and training are saved. These configurations are updated manually with a check that save a default version if none exists in the load-settings function. Base on this configuration, we typically only set the backend to do training, prediction, and top-10-recommendation generation once since we decided to not do live modification to the database and thus live training and prediction in the final version of the demo. Once these has been done and saved to local storage, for the rest of the time the backend is run we simply read back the saved data for responding to front-end requests for list of recommendations. The inbound security rule of the EC2 instance was modified to allow for traffic on ports 8000 and 8080 to allow for frontend requests to reach the backend.

We have the MySQL database running constantly on the Amazon RDS for MySQL and manually configure the inbound security rule to allow for the IP address of the Flask app backend to make queries to it using the MySQL Connector/Python.

In terms of end-to-end performance of our system, after running the application a few times and taking the average, it takes about 2.9 seconds from the time the frontend starts communicating with the backend until it receives all requested data and files. Obviously, if more recommendations are generated, the application would take longer to retrieve them. The size of the pictures that the application has to retrieve also accounts for runtime performance. In testing performance, the minimum runtime was 2.25 seconds, while the maximum runtime was 4.32 seconds. Performance would be significantly longer if we had committed to doing live demo with live training and predictions, as the average execution time of the BaselineOnly algorithm for training and prediction making are 3.8 seconds and 18.2 seconds, respectively, as shown in 5. It should be noted that connecting to the MySQL database from the Flask app backend using MySQL Connector/Python can take 1-2 seconds on fresh connections.

In terms of resource utilization, the backend uses about 21GB of storage, which includes the Anaconda environment, library like Surprise and their dependencies, local copy of the reviews part of the dataset used during development, saved internal data checkpoints, and the storage of the photos of businesses to be requested by photo-id. During runtime, at idle it uses about 96MB to 114MB of memory (RAM) shown in figure 12, up to 200 MB observed during nominal operations, and about 1.2 GB observed during training and prediction. As far as CPU utilization, on idle and nominal operations it is fairly low, hovering around 1% utilized, but during training and prediction the CPU usage has been observed empirically up to sustained full 100% utilization.

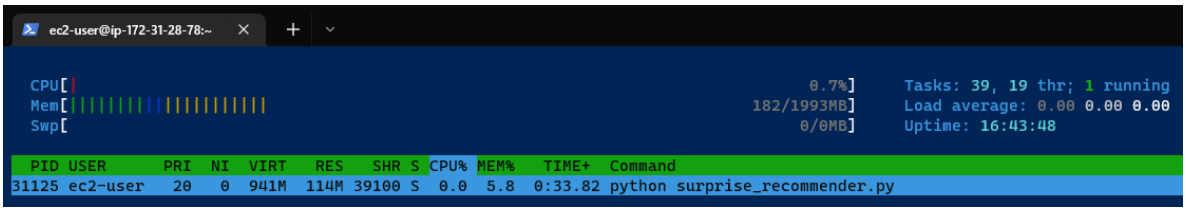

Figure 12: Resource Utilization of the Flask app backend on idle monitored using htop utility tool

5 Analysis and Future Work

A couple of concerns are still considered for this project: one of the major shortcomings that our recommender system still has is that our current RDS base does not have enough quota to contain the entire Yelp Open Dataset, thus we decide to proceed using a subset of the original database. Although the subset contains more relevant information as we mentioned, certain glitch would still appear when the front end fetches a random key that does not belong to the subset, and could cause confusion; we also have other considerations over improving our current front end graphical interface: as we explained, the current front end is only producing results recommended generated by a random key user id, and does not have such functionality over locating a specific user for special interests and are unattached with Yelp's logging feature. Last but not least, the current model only contains manually imported database from Yelp's open data set and does not have a real-time live update as Yelp continues to grow and increase its data set; meanwhile potential scalability issues regarding increased database size could also be encountered as more user and business data flushes in.

Some future works are also to be considered regarding our project: to lower the run time and also increase the efficiency of the recommender system, we are looking into using an optimized recommender library with potential multi-threading and sharding support that can greatly benefit the performance of the system, and also increase I/O throughput, storage capacity and scalability of the project; in the meantime, a potential switch to machine learning models is also considered in this case to substitute traditional algorithm for recommendation system; a lot more parameters should be considered: for example, an increase of consideration over MAE, R^2 and RMSE could help us better understand the model and its real-life performance; more novelties should be considered for user with less or no data points, and could also aid to illustrate or portrays existing users with more freshly interesting ideas outside of their current custom habits; For the front end interface, we are also considering adding abilities to create new users and implement login features, or even connect with Yelp's own APIs for searching user past reviews or modify current user information. In the meantime, similar

editing features regarding business such as making new reviews and ratings for a business is to be implemented as well; regarding the fact that the current model is only built for current database, a potential event-driven hook could be added to re-train the model and update predictions when the database is sufficiently modified. To test different backend functionalities and/or improve the current scalability of our flask backend, we are also considering deploy the Flask App on a (Docker) container and use K8s for orchestration, so the application is scalable or deploy using a Serverless service like AWS Lambda to make predictions only upon frontend user request for recommendations.

6 Conclusion

In this project, many current algorithms, and state-of-the-art paradigms of recommending system are studied to shape a cost-efficient collaborative filtering model that best reflects the user interests regarding their Yelp profiles. Multiple varieties of algorithms such as SVD, NMF and NormalPredictors are compared for their trade-offs and performances under our dataset and BaselineOnly is selected based on its outstanding RMSE scores; meanwhile, BaselineOnly also outperforms other models with its fast-training time that enables us to perform dynamic predictions upon request.

We also explored the frontend GUI and related Flask with Restful APIs to guarantee the successful full stack built for the project; in the demo some real-time estimates were made based on user request and the corresponding recommended business were shown to the customers with graphical content and in-detail reviews as well as ratings.

This project has helped us better understand the functionalities of IaaS cloud platforms, particularly with Amazon Web Services as we are explicitly using their RDS, Amplify and EC2 instances to construct the full prospect of our system from database to frontend.

The primary limitation of our work would be the lack of scalability of our database considering the cost of using AWS services; since we are self-sponsoring this project we could only afford to use a limited quota for RDS and EC2 which resulted to the fact that we were using a subset instead of the entire Yelp Open Dataset; unfortunately, the App is also running temporarily and needs to be shut down when not tested or used just to save budget due to the same reason.

References

[Hug20] Nicolas Hug. Surprise: A python library for recommender systems. *Journal of Open Source Software*, 5(52):2174, 2020.

[Kor10] Yehuda Koren. Factor in the neighbors: Scalable and accurate collaborative filtering. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 4(1):1–24, 2010.

[SOK+17] Shi, O'Mullane, Kickham, Rad, and Rubino. Yelp food recommendation system, 2017.

[SP13] Sumedh Sawant and Gina Pai. Yelp food recommendation system, 2013.

[Tin00] Chad Tindel. Using collaborative filtering on yelp data to build a recommendation system in amazon neptune, 300300.

[Yel04] Yelp. Yelp open dataset, 2004.