Accomplishments/Description

Have you and your friends ever struggled to think of where to eat? We created a restaurant review Android App localized to the Champaign-Urbana area and focusing on the college student demographic. Users will create an account and will be able to view reviews of local restaurants by fellow students, view average ratings, view restaurant details such as price, cuisine, address, etc., and can write their own reviews for restaurants (as well as edit and delete those reviews as well).

Our review system allows users to give a rating out of 5 (by 0.5 increments) as well as write a text review highlighting various aspects of their experience.

Something more unique to our app, is that we created a budgeting feature that allows the user to enter a weekly budget. After visiting a restaurant, they can enter how much they spent there on a given day to record their transactions spent eating at restaurants. We then provide visualizations to track daily spending in a given week to help the user understand their spending habits and what days are higher/lower than expected, and adjust accordingly. We also keep track of key statistics such as how many times they have exceeded their set budget, and what their average weekly spending is for weeks they are using a budget. In addition to viewing their current week's budget and spending data, they can view any past week's data as well, to see their past trends.

We also provide recommendations for our users for food that they might like given their past transactions at given restaurants and the frequency of those transactions. To generate these recommendations, we use a Word2Vec Machine Learning model that recommends restaurants to users based on the cuisine patterns of restaurants they have eaten at and takes into account what restaurants other users frequent that shared some overlapping restaurants with the current user. Thus, the model improves as more data is added by users. These recommendations are generated by a script that we have scheduled to run twice a day (at noon and midnight) on our server.

Usefulness

We created this idea out of a struggle to agree on where to eat while also staying under a given budget. This app allows you to easily search through our dataset of all restaurants in Champaign-Urbana, through a variety of filtering methods including cuisine type, price range, city, etc. and to view details of the restaurant as well as what other users feel and rate the given restaurant. This provides you all the tools and data a user needs to make an informed decision on where they want to eat. However, instead of just allowing you to view the restaurants and decide for yourself, we also created an advanced recommendation system with a Word2Vec Machine Learning model that sends you new recommendations periodically based on the reviews/transactions you have entered into the app to help you more actively find places to eat at.

In addition, our app also helps with budgeting of how much you spend in a given week in the Champaign-Urbana area, since keeping track of spending is essential for college students who have tight budgets, but often find themselves eating out due to a lack of time. Our app's ability to let you add transactions for a given date, restaurant, and price, visualize your daily spending habits, provide statistics about your overall weekly spending habits, and to view all past spending data seamlessly are key tools to help students financially manage the costs of eating out.
Data

We used real restaurant data from pulled from Yelp to populate our database. Specifically, we needed the address, price range, name, phone number, image url, and latitude/longitude for each restaurant. We wrote a script that got this info from the Business Search endpoint of Yelp's API, and that inserted it into our Restaurants table in our database. For each restaurant, we also stored its cuisine types in a different table called "RestaurantCategory" because each Restaurant could have a variable amount of categories, so it made more sense to split it out into it's own relation.

Other than that, we store user data such as the User accounts, the Reviews they have written, the Transactions they have recorded, and the Budgets they have set.

We also store data for the all the Recommendations that our periodic script generates for users. This script is scheduled to run twice a day (once at noon and once at midnight), and to update it's models every week to take into account all the new data that was added.

ER Diagram

Schema

Tables

User(Email, Name, Hashed Password, uID);
Budget(Date, UserEmail, Term, Total);
Transaction(UserEmail, Date, Amount, Restaurant_name, Restaurant_Address);
Recommendation(UserEmail, Restaurant Address, Restaurant Name, Distance);
Review(UserEmail, Restaurant Address, Restaurant Name, Description, Rating, Date);
Restaurant(Address, Name, PriceRange, Phone, Image_URL, Lat, Lon, rID, Overview);
RestaurantCategories(Name, Restaurant Name, Restaurant Address, Category);
RestaurantEmbeddings (rID, c1, c2, c3, c4, c5);
Views (Developer Side Only, Not Part of ER Diagram)

BudgetStatistics(Useremail, Avg_Spend, Total_Spend, Total, Date);

Data Sources

We pulled data from Yelp using the Business Search endpoint of their public API. We created a Python script to make network requests to the API and collect data on address, price range, name, phone number, image url, and latitude/longitude as well as an array of cuisine types for each restaurant in the Champaign-Urbana area in Yelp's data and loaded that each restaurant as a JSON object into a JSON file. After that, we created a separate Python script using the psycopg2 Python library to interface with PostgreSQL to execute the SQL queries to load the data from this JSON file into the our Database tables using INSERT INTO statements for each JSON object.

Application Functionality

- Let users Create an account that allows users to store their email, password, and name. Users can edit their password/name, or delete their account as well.
- Let browse and filter the list of restaurants in the Champaign-Urbana area by name, address, price range, cuisine, and city.
- Let users view specific details of a given restaurant including Address, Position on a Map, Cuisine Types, and Price Range.
- Let users view reviews/ratings written by other users for a given restaurant, as well as the average rating given to that restaurant overall.
- Let users write reviews for local restaurants by assigning a rating (0-5 with 0.5 increments) and by writing a text description. Users can also edit and delete this review.
- Let users keep track of weekly spending habits by adding a budget limit goal for a given week, and the actual total amount spent for that week calculated by transactions that were added.
- Let users add transactions for a given restaurant and the date as well as amount spent for that transaction.
- Let users see a Pie Chart Visualization of daily spending habits for each week that transactions are added, that weights each day according to its share of total spending for that week. This weightage in the Pie Chart clearly allows users to identify what days their spending patterns are too high/too low and adjust accordingly.
- Let users view spending data and visualizations for past weeks as well as the current week seamlessly.
- Let users see overall statistics of number of weeks the budget was satisfied and average spending each week a budget has been set, to further analyze spending patterns.
- Let users receive periodic recommendations based on restaurants that you have added transactions for and the frequency with which you ate at those restaurants.

Basic Function Example

For the browsing and filtering restaurants feature, a user can select any combination of filters by going to the restaurants tab in our app and then hitting the button in the top right to show all available filters. They can filter the total set of restaurants by any combination of Restaurant Name (part or all of the restaurant name), Address, Price Range (equal, greater than or equal, and less than or equal), City, and Category. This feature required a complicated SQL query (shown below) and was one of our key features in allowing the user to more easily explore restaurant options that they desired.

Code Snippet

We used Psycopg2 as the interface to the server. Here's the psycopg2 command that we used to insert filter restaurants.

```python
cur.execute("**\n**\nSELECT "Restaurant".address, "Restaurant".name, "Restaurant".pricerange, "Restaurant".phone, "Restaurant".image_url, "RestaurantCategories".category, "Restaurant".lat, "Restaurant".lon FROM "Restaurant", "RestaurantCategories" WHERE {where_clause} AND "Restaurant".name = "RestaurantCategories".restaurant_name AND "Restaurant".address = "RestaurantCategories".restaurant_address ORDER BY "Restaurant".name ASC, "Restaurant".address ASC"").format(where_clause=where_clause), where_params)
```

Since the user can select any combination of Restaurant Name, Address, Price Range, City, or Category to sort on, part of the query's where clause is dynamic based on the request. Thus, we have a "where_clause" variable (what attributes to filter on) and "where_params" variable which is the actual values for the attributes we need to filter on. Here is an example of 1 such query where we filter on a given price range of 2 dollar signs, and city set to Urbana:
SELECT "Restaurant".address, "Restaurant".name, "Restaurant".pricerange, "Restaurant".phone, "Restaurant".image_url, "RestaurantCategories".category, "Restaurant".lat, "Restaurant".lon
FROM "Restaurant", "RestaurantCategories"
WHERE city = 'Champaign' AND pricerange = 2 AND "Restaurant".name = "RestaurantCategories".restaurant_name
AND "Restaurant".address = "RestaurantCategories".restaurant_address
ORDER BY "Restaurant".name ASC, "Restaurant".address ASC

Dataflow

After the user enters the data on the screen or clicks on a button on the screen, our Android code takes that data and makes an appropriate HTTP request of a given type (GET, PUT, POST, or DELETE) to a given API endpoint/URL routes on our REST API that is running on our server written in Flask, a Python framework. This request made from the Android side includes the proper query parameters or JSON payload to match the formatting the REST API is looking for. To make this request from the Android side, we use a library called Retrofit that maps endpoints/URL routes to methods in a Java interface, and maps HTTP Request/Responses to Java classes with the appropriate variables. After this, Flask sends the request to the proper function in our REST API, which does any preprocessing, and then runs a SQL query on our PostgreSQL database that is running on our server. The REST API function then processes the result of that query into JSON that is then returned as an HTTP response. The Android app will receive this response and serialize it into a Java Object and examine the contents of the response as well as the status code, to determine what to change/display in the app.

An example: The user enters the name of a restaurant and enables the filter for that in the Restaurant Filters panel for the large list of restaurants, and selects the check box for “By Name?”. Then after hitting the check mark at the bottom of the Restaurant Filters panel, the app calls a Retrofit method that creates a GET /restaurants request with the appropriate URL query parameters for the filters to our REST API. The REST API sends the request to the appropriate function, creates an appropriate query and sends it to our database, which executes it. The database executes and returns the data back to the function in the REST API, which serializes the data into JSON and returns an HTTP response with this JSON. The Android app receives this response, and displays the restaurants that were returned in the JSON payload of the HTTP response. The images below show the actions on the app that would enable this example:

Advanced Functions

Budgeting
Functionality

Unique from other restaurant review apps, we want to allow the user to be able to enter a weekly budget goal. Then they can add transactions in a given week for a restaurant they visited, amount spent, and the date of the transaction. Using the data of their transactions in a given week, we create a Pie Chart visualizations for their spending habits day-to-day spending habits within that week. This Pie Chart visualization places larger weight on days that contributed more to the total spent for that week and smaller weight on days that contributes less to the total spent for that week. Thus, it makes it clear to the user what days they are spending more or less, enabling them to adjust their spending habits accordingly.

Furthermore, other than just viewing the visualization and adding transactions to the current week, they can also easily view the visualization and spending data for past weeks by selecting so in the Date Picker in the top right corner of the Budget tab of our app. This allows them to analyze how they have done in past weeks and their past trends to see how they can meet their budget goals better.

Below the Pie Chart visualizations, we also display some wholistic statistics for the user. The first statistic is the ratio of number of weekly budget goals they have met (spend less than or equal to the budget goal) to the number of weekly budget goals they have set overall. This statistic allows them to see how often they are meeting their budget goals, which can be powerful once they have been using the app for several weeks. The other wholistic statistic is the average amount they have spent per week for weeks they have set budget goals. This statistic is also valuable for them to see how much they are spending throughout a given week.

Why It Is Advanced

Supporting all of these functionalities for the Budgeting feature required a great deal of additions to our Database, REST API, and Android app code.

First we had to modify the database to add the “Budget” and “Transaction” relations along with their foreign key constraints to User (for both) and to Restaurant (for Transactions). We also added a View called BudgetStatistics that finds all the transactions within a given Budget's weekly period for each Budget of each user, and computes the total spent during that Budget's weekly period.

Then we had to add new endpoints/URL routes to our REST API to find all transactions for a given user within a given week given a random day in that week. This was a challenging SQL query because we could be given any random day within that week (a Tuesday or Thursday or Saturday) and had to find all the transactions from the Sunday that started the week that given date was in up through the Saturday that ended the week the given date was in for a given user. This required some rare SQL operators such as MOD as well as type casting and Date subtraction. Thus, this query alone took as several hours to get working. Using our View that recorded the “BudgetStatistics” we also had to compute the average spent across all transactions during all budgets the user had entered and also figure out what budgets they went over on and which ones they satisfied. Creating this view and API function also took a couple hours to decide how we wanted to structure the query and to decide that we wanted to create the View. We also had to create some other CRUD endpoints for our new relations “Budget” and “Transaction”.

Finally, the app side required the most complexity and time. We had to create a whole new tab within our app where the user could enter their budget, add transactions, view the Pie Chart visualization for a given week, and see the Key statistics of their success in satisfying budgets and in their average spending per budget week. In this tab, the Adding Transaction screen has a suggested list of restaurants that pops up as you type each letter of the restaurant you are trying to add a transaction for. This feature required a callback function each time a user typed a letter or backspaced, as well as an API call, and required some caching of results to not overload the API. Furthermore, the UI and callbacks for the 0-9 keypad for entering the amount of the transaction was created by us, and not from a library.
The biggest challenge on the app side, was creating the Pie Chart Visualization. The Visualization capability is not native to Android, so we had to use an Open Source Graphing library called MPAndroidChart. This graphing library was challenging to use and we tried out other types of graphs such as line graph and bar graph before deciding that the Pie Chart looked the best and would give users an accurate, elegant, and easily understandable visualization.
Finally, we had to support the user going back to previous weeks and viewing the Visualization/spending statistics of previous weeks seamlessly, which required to add some callbacks to make new API requests and refresh our UI as the user changed the week they were looking at.

Recommendations

We leverage user transaction history to generate a list of recommendations for each user.

Intuitive Description

Recommendations are tailored to recommend restaurants that tend to appear in "clusters."

To explain this consider the following situation:

- User Alice eats at restaurants A, B, and C.
- User Bob eats at restaurants B, C, and D.
- User Carl eats at restaurant B.

Restaurant B occurs alongside restaurants A, C, and D. However, it is not evenly split - it occurs alongside C twice as frequently as A and D.

Our algorithm is sensitive to this pattern; if Carl asked for a single restaurant recommendation, he would receive D because it is much more likely to be eaten at by patrons who visit B.

Machine Learning Algorithm

To find these clusters, we use an algorithm called Word2Vec. It was originally designed to be used to embed words into vector spaces, but it makes use of the same co-occurrence information that is described above.

We adapted Word2Vec into a "Restaurant2Vec" that converts each restaurant into a 5D vector, so that restaurants that appear frequently together are embedded closely in this 5D space.

We build this model using a Python script that generates a TensorFlow/Keras model. The output of this model is the embeddings for each restaurant; these are stored in a table named "RestaurantEmbeddings" that maps each restaurant to its 5D embedding. Because of how much of a load this script imposes on our VM (it is a free-tier GCP VM, and gets 0.5 of a CPU core), this script is scheduled as a cron job to run automatically once a week.
These stored embeddings are used twice a day (again, in another Python script running as a cron job) to generate recommendations for each user. The actual process of generating recommendations is far less heavy, because it is a simple SQL query to generate a user’s restaurant preference vector, and then find the nearest neighbors in the embedding table using Euclidean distance.

**Visualization**

We can visualize what the 5D embeddings for each restaurant look like by mapping them into a 2D space. An example set of embeddings is below:

It’s very obvious that the algorithm identifies clusters in the data.

This visualization was made using a set of fake users, who have three “hidden” categories each. Users only eat at restaurants in their categories, but the model does not know what their categories are. The clusters in the graph correspond roughly to restaurant categories, which gives us confidence that our model is learning useful information.

The choice of a 5D embedding space was made by experimenting with different sizes of embedding spaces and using various metrics to quantify how well they:

- Forced clustering - too many dimensions doesn’t lead to clustering
- Used their “space” - if most of the data can be found on one or two axes, the embedding isn’t very efficient

For the second one, one metric we used was the explained variance ratio of a singular matrix decomposition. This is a measure of each axis on a graph, and how well it “explains” the real position of the data. For example, for the above plot, the explained variance ratios look like this:
We can see that the first component explains a little more than 30% of the true location of each point, the second one explains ~25% of it, while the third and fourth components both explain ~16%. Only the fifth component is "useless," which implies that our 5D representation does a good job of holding useful information.

**Why Advanced?**

This feature is advanced because it:

- Required a great deal of reading
- Needed a lot of optimization
- Adds several columns and tables
- Uses very complex queries
- Is a robust solution to a hard problem
- Uses complicated mathematical concepts applicable to Machine Learning and Statistics.

**Technical Challenge/Advice**

One of the biggest technical challenges we ran into was accurately making network requests in Android App to our REST API Backend to connect the two portions of our project. We were using some older Java Networking classes, and it was very time consuming to properly format each of the requests, with the proper JSON payload, the proper query arguments, the proper Verb type, etc. It was also a challenge to parse responses, since the old Java Networking classes just returned the response as bytes, so we had to manually read the bytes to parse the response. Furthermore, we were trying to manually write multi-threaded code to run these network requests in a background thread to not overload the main UI thread, and were writing manual callbacks when they returned to update the UI, which were both hard to implement manually in Android.

After looking into other easier options, we found the Retrofit2 library for Android. Retrofit2 allowed us to map API endpoints/URL routes to standard Java methods in Java interfaces, and serialized Java objects into JSON for requests, and deserialized JSON into Java objects for responses. This abstracted the networking aspects into more simple and familiar Java fundamentals. Furthermore, specifying the request verb, what parameter should be a URL query arg, what parameter should be the request body, etc. was very simple with the use of Java annotations. Best of all, Retrofit2's Call objects had an enqueue() method with a built-in callback function, which handled the threading and callback method automatically for each network request. Considering we had 30+ API endpoints, and many other groups will as well, Retrofit2's simplification of the networking aspects was huge in helping us build our mobile app fast and connect our app to the backend REST API very easily and efficiently. It is definitely a key to speeding up Android mobile development for the networking aspects of the app.

**Process Compared To Initial Development/Specifications**
Our development went mostly according to the initial development plan since we started out by setting up our PostgreSQL database and schema and by writing the scripts to add the restaurant data from the Yelp API. We then built out our UI and REST API functions to support basic CRUD operations. We finally concluded with working on both of the advanced features. Thus, the timeline events of our Initial Development Plan matched our actual timeline of how we developed the app.

Our specifications also mostly held but we had a few things we decided to change. For one, we were initially planning to do restaurants in the San Francisco Bay Area, but we decided to change to the Champaign-Urbana area because it was more relevant to users, added a local feel to the app improving its relevance, and easier to evaluate and improve our Recommendation feature. We also initially thought we may have a panel for restaurant owners where they could update their restaurant's page, but we realized this was not realistic as it would be hard to verify the accuracy of owners, and that we did not want the data of each restaurant page to be changed from our given accurate data from the Yelp API.

**Division of Labor**

We worked collaboratively on most things and mostly just talked it out through a group chat. The group chat was a very efficient way of us collaborating when not meeting in person, and a quick way for us to identify and solve issues as they arose. We did most of it together over Spring Break, and the advanced features over the past few weeks in smaller subgroups. But painting in broad strokes: Junyoung Kim worked on frontend design on Android and the budgeting visualizations, Raajesh Arunachalam worked on the Android side as well setting up the endpoints for the REST API, Rohit Singh worked on the recommendation feature, Database schema/constraints, and the REST API, Alex Chung worked on getting the real world data from Yelp parsed and into our database, and on the budgeting visualizations.

Overall, we are very pleased as a group with the contributions of each member, and we felt that each member did their assigned part with high quality and communicated well with the group.