Group Members:
Xavier Beynon (xbb55),
Chukwudi Iwueze (cci233),
William Nation (whn94)
Po-Chen Yang (pc22369),
Anthony Garza (aag2423),
Edward Lee (el8366)

Operation Repo IDB1 Technical Document

Table of Contents:

I. Introduction:

For our project, we used data provided by Yelp on businesses, reviews and reviewers. This data was released to enable academics to compete in the “Yelp Dataset Challenge”. The challenge is to use the given data come up with a machine learning project that predicts different attributes of a business, reviewer or review. For example, one could come up with an algorithm that predicts the rating that a certain reviewer would give to a certain business or which business a reviewer is likely to review next.

The data we have collected was not used to participate in the competition but to create a web app that presents the data in an intuitive manner. To represent the data, we created models using Django. Our app is hosted on Heroku with Twitter Bootstrap used for the interface. Each of our data objects has their own attributes. The attributes for each data object are as follows:

Business

Attribute	**Description**
Business ID	The encrypted business ID.
Name	The name of the business.
Neighborhoods	The name of the Neighborhood that the business is situated in.
Full Address	The address of the business.
City	The city that the business is located in.
State	The state that the business is located in.
Latitude	The latitudinal position of the business.
Longitude	The longitudinal position of the business.
Stars	The rating of the business in stars.
Review Count	The number of reviews given to the business.

Categories	The categories that the business is evaluated on.
Open	Tells whether the business is open or closed at a given time.
Hours	This includes the days of the week that the business is open as well as operating hours for each day.

Table 1

Review

Attribute	**Description**
Business ID	The encrypted ID for the business being reviewed.
User ID	The encrypted ID for the reviewer giving the review.
Stars	The rating given to the business in the review.
Text	The review write-up.
Date	The date that the review was given.
Votes	The type of vote given to the review (useful, funny, cool) and the number of votes given.

Table 2

User

Attribute	**Description**
User ID	The encrypted ID of the reviewer.
Name	The first name of the reviewer.
Review Count	The number of reviews that the reviewer has given.
Average Stars	The average rating (in stars) that the reviewer gives to businesses in their reviews.
Votes	The type of votes as well as number of votes given by the reviewer.
Friends	The User IDs of friends of the reviewer.
Elite	The number of years that the reviewer has been an elite member of Yelp.
Yelping Since	The month that the reviewer joined Yelp.
Compliments	Types of compliments received by the reviewer.

Fans	Number of fans that the reviewer has.

Table 3

Throughout the course of this report, the process of creating our web app (including github workflow, creating an API and creating models using Django) will be explained in detail.

II. Design

A. GitHub WorkFlow:

For our github workflow, we chose to employ the Feature Branch Workflow. The idea behind this model is that each developer works on a specific feature of the project. While working on the individual features, each developer creates a branch pertinent to the feature that he or she is working on and pushes commits to that branch. This enables each developer to work on a specific feature without affecting the work of the other developers. The master branch is unaffected, thus preventing broken code from being added to the main codebase.

In order to add features to the main codebase, we used pull requests. Pull requests gave each of us an opportunity to review the code for each feature branch before it was added to the master branch. This enabled collaboration as well as the preservation of order within the project. The Feature Branch Workflow provides a form of encapsulation that lets each developer work on his or her part of the project without interference. When a developer is stuck in the middle of writing a feature, pull requests serve as a means to initiate discussion around that feature and also for the developer to seek advice on how to proceed from colleagues.

This model facilitated the delegation of responsibility among each of us for each feature of the project and it provided an intuitive method for us to collaborate and review code before it was pushed to the master branch. It also increased productivity through the division of labor.

https://www.atlassian.com/git/workflows#!workflow-feature-branch

B. RESTful API on Apiary

Our RESTful API was designed using Apiary's legacy syntax. We decided to go this route as the legacy syntax allows for a much cleaner interface as it enabled us to group all the HTTP methods together for each individual resource. This was advantageous over the new syntax as it separated the methods if the requests had differing URLs. As a result, this would have cluttered the documentation as each resource has a single GET method to list the two other related resources.

Originally the API used the plural form of the resources for the URLs while the JSON referred to the resources by their plural form. We made the decision to switch to the singular form of the resources in the API in order to maintain consistency across the board. In order to separate our HTML requests from our JSON requests, we decided to prepend '/api' before each resource. Just for example, the endpoint of the API would be '/api/business' opposed to just '/business'.

The API includes seven different requests for each resource. These are listed in Table 4 below:

Method	URL	Description
GET	/api/resource_name	Lists every resource in the database
POST	/api/resource_name	Create a new resource entry
GET	/api/resource_name/{id}	Retrieves the information for a particular resource. id is used as primary key in the database
PUT	/api/resource_name/{id}	Updates an existing resource
DELETE	/api/resource_name/{id}	Deletes an existing resource
GET	/api/resource_name/{id}/other_resource_1	Lists all related (other_resource_1)
GET	/api/resource_name/{id}/other_resource_2	Lists all related (other_resource_2)

Table 4

C. Django Implementation

Our Django models design is based on the JSON Schema from the Yelp Dataset Challenge page. In order to convert the dataset to the Django models, we had to make some modification to the original format.

In the actual Yelp Dataset, each business has multiple reviews, which is a one-to-many relationship. Each reviews is written by exactly one users, which is an one-to-one relationship. In our case, however, we chose one business and one of it’s review from a user. Our Django models still keep that one-to-many and the one-to-one structure, just the data we will use doesn’t actually takes the advantage of one-to-many relationship.

The advantage of JSON is the ability to create multiple layers of dictionaries and lists, however our database is a little bit different, specifically the business model. In the Business model, we have a category called attribute that holds a list of categories.

Therefore, we had to create another table to hold those categories and that corresponds to the Business model’s primary key.

Another issue we ran into was in the attribute name. One of the attribute names from the dataset is “open”, which is a keyword in Python. We had to use “is_open” instead of “open”. In addition, there was a repeated multivalued attribute name “vote” in both the review model and the user model. Since it is multivalued, we needed a table for both of these attributes. However, a duplicated table name is not allowed. To resolve this we renamed the attributes to “Review_vote” and “User_vote”.

For the purpose of our project, we need to be able to have a primary key in review Our Django models design is based on the JSON Schema from the Yelp Dataset Challenge page. In order to convert this to Django models, we had to made some modifications to the original format.

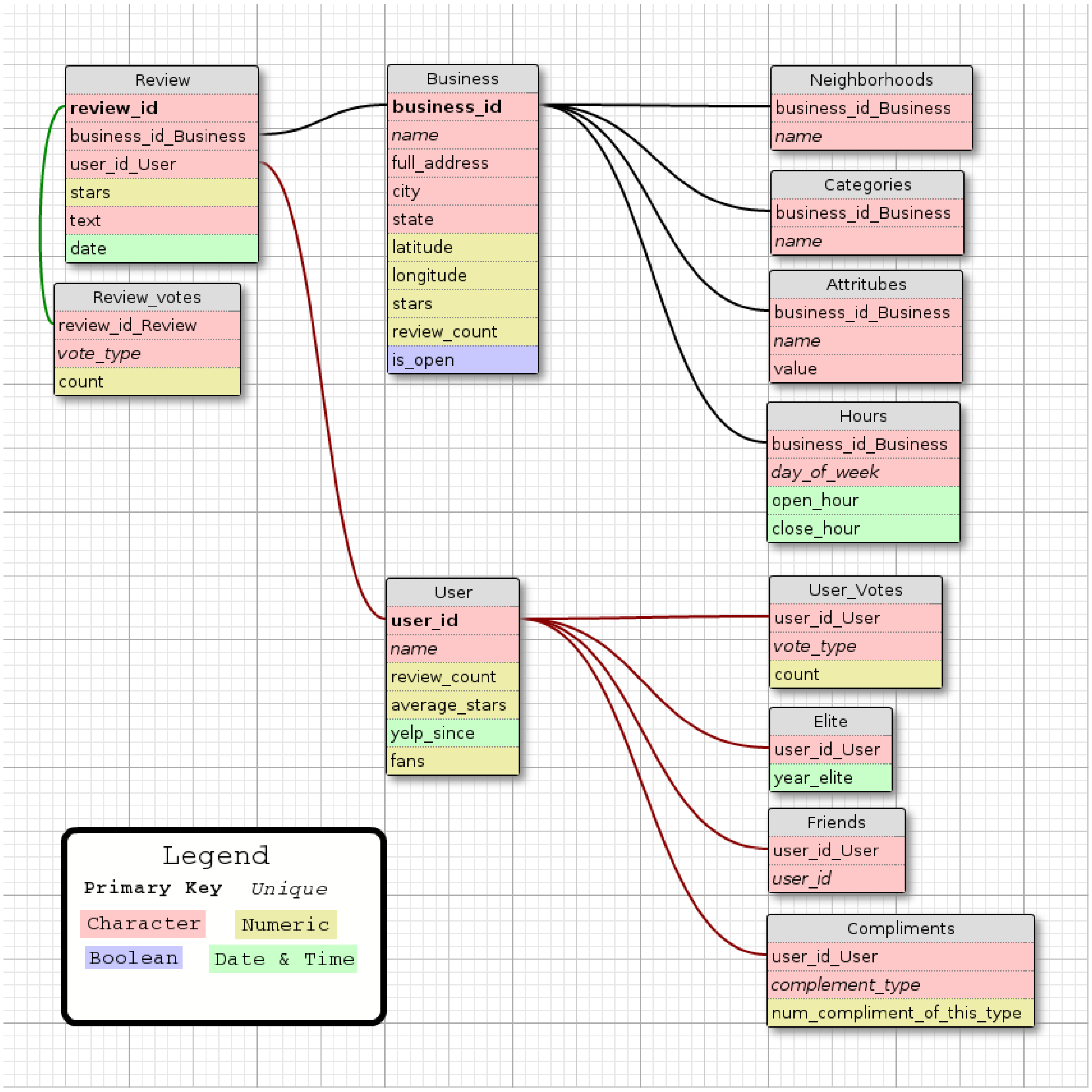

Figure 1. Django models design

i. Django Views & Templates

We used Django's templating language to create the pages of the website dynamically. We wrote a different html template for each of the four base pages we had; splash, business, review, and user. Django would then populate the pages based on what inputs were passed to it through the caller’s url. When a user requests a page from the website, Django’s parses the url, looking for any input values after the page name using a regular expression. It then calls the view and sends it the page number the user wishes to see as input. One view was created for each of the pages. The called view will check the input for which context dictionary should be applied to the given template.

D. Responsive Design of the Website

Next we set out to create web pages which would respond to the user’s device screen size. For this, we used Twitter Bootstrap’s built in div classes. An example of this can be seen in the navigation bar at the top of each page. It adjusts its size based on how big the screen is and even disappears entirely when viewed on a smartphone, leaving only a drop down menu icon in the top right of the page. In addition, we were able to make the current web page/ category (business, review or user) highlighted by using the active class for our navbar. In addition, we the place majority of our content in a div that uses Bootstrap’s jumbotron class, allowing for a design that will scale with each user’ s device screen size.

Another feature that we included was an accordion like section that can be used to display the raw JSON data if the user wishes.

E. Database Design:

Though not used for phase 1, we implemented a PostgreSQL database to store the data. The motivation for this was to give team members a way to write complex queries without downloading the data or using grep commands. We figured it would also be a worthwhile investment for future phases.

To develop the database structure, we evaluated the effort involved in building a rigid relational model; one that would involve alotting a column for each object property and linking them through normalized relations. Realizing that this would involve an enormous amount of unnecessary manipulation of the JSON data, we searched for another more informal approach. PostgreSQL 9.3 added a native JSON type - in other words, the properties of JSON objects could be accessed with familiar operators like -> directly within queries. By simply creating tables with a single JSON column, we could directly import the yelp data and still be able to query it with the usual SQL join, filter, group, and ordering syntax.

Each of the 5 top level objects specified in the yelp data set was given a separate table with two columns: a primary key used for joins, and a single JSON column named data. Views to access common data shapes were written and will be accessed as django models in phase 2.

III. Tests

A. Unit Tests:

For phase 1 we created a test for the seven different combinations of HTTP methods and API endpoints for each of the three resources which gave us a total of 21 unit tests. Our tests checked for two things in particular, namely the status code and the content body of the response if the method is supposed to return anything. We were not able to test a few of the status codes listed in the API documentation as we do not have any restrictions on requests.

We created our API using the data from the first entry of each resource's respective database which enabled us to test the content directly opposed to using dummy values. The only exception to this was the GET method that is supposed to list all the entries for a given resource. Given that each entry contains a long dictionary of values, it was decided to use a small subset of the entry’s attributes as the content.