Group Members:
Xavier Beynon (xbb55), SQL/JS
Chukwudi Iwueze (cci233), SQL
William Nation (whn94), API
Anthony Garza (aag2423), Phase 3 Leader/UI/Search
Po-Chen Yang (pc22369), Phase 2 Leader/Django/Database/API Support
Edward Lee (el8366), Phase 1 Leader/UI

Operation Repo IDB3 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.

Potential use cases for our website include looking at aggregate information from the Yelp Data Set in the form of easy to read charts and graphs. Another potential use case for our website is to serve as an example of some of the possibilities of what can be done using the django web framework in conjunction with custom css and javascript. Because our website runs on the yelp dataset, theoretically it would be possible to plug in a smaller or larger database using the data from the Yelp Data Set to test new features and determine their efficiency on different size data sets. Potentially our website should be dynamic enough to handle any other database that that deals with business, users customers and reviews assuming that the database is stored in a similar format. In that sense it should be possible for other teams to simply plug-in a different heroku database and have our web app up and running with little to no revisions.

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.
Yelp URL	The Yelp url of the business
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.
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

In addition to the seven requests listed above, we also added a GET request to return the coordinates for every business. The results of this call are then used to produce the heat map displayed on our home page.

C. RESTful API for Models

We created our RESTful API to reflect what we had for the Apiary API so that all of the GET, POST, PUT and DELETE API calls would follow the format displayed on Apiary. This allowed others the ability to pull out information from our database based on the specific relationship specified in the url. However, actually creating the API posed another set of problems as not one of us has had the experience of creating an API before. After switching back and forth between tastypie and Django's REST framework, we decided to implement our RESTful API using the latter as it was simpler to use with our models.

To make a distinction between POST and PUT, we designed it to where POST requests would create a new resource whereas PUT would update a specified object. Given the complexity of handling multi-value attributes, we decided that all the related fields must be included in order to make valid POST and PUT requests. This removed a lot of tedious checking for equality and it provided a simple, uniform way of extracting multi-value attributes and formatting them to the desired design.

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.

There have been several updates to our overall look and feel on the websiteWe have since removed feature that would include an accordion like section used to display the raw JSON data as that can be handled by our api. Next, have created index pages for each of the three categories to allow the user to see all of the objects for the businesses, users and reviews. By default the lists of business and user objects are displayed in alphabetical order, and the reviews are displayed according to the date the review was posted. Each of these objects is then a link to the corresponding html page that displays the data for that object. Specifically for our business pages,we added code that dynamically handles displaying the single value attributes as well as the multi-valued attributes from our database for our business pages. Also the google maps widget on our business pages now generates the map view dynamically rather than relying on was hard-coded data. For the attributes of the business pages we incorporated a responsive design by making the attribute lists float. When the viewing resolution size changes, the sections under the main “Attributes” shift to properly fit in the given space. Lastly, we have added a scrollable table on our business pages that displays information about the reviews associated with the current business and links to the those review pages.

In addition, to the updates that we have made in the previous iteration, we have also added some new features to our website. One of the immediately noticeable interface changes that we have added to the index page is the carousel that displays two graphics relating to two queries that we ran on our database in addition to a “heat map” based on the location of the businesses in our database. Next by refactoring our code we have been able to move the navbar into it’s own separate html file (navbar.html), which allows for changes to navbar to only occur once versus having to change each of the static templates individually. On the navbar itself we have added drop down options that allow the user to pick either businesses, users or reviews based on specific common attributes or to display a list containing all of the businesses, users or reviews. For example, clicking on the Business tab on the navbar allows the user to have our website display all of the businesses in our data set or groups of businesses from our data set that fall under the listed categories, such as “Pizza”, “Restaurants”, “Nightlife”, etc. Another feature that we have added is the ability to change the background of the web pages to three different background themes consisting of a lighter tone, a medium tone and a darker tone for aesthetics as well as different contrast backgrounds to view our website based on the time of day or viewing angles.

E. Database Design:

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.

Later on, we were told that we were required to create a rigid relation model for our database. Thus, we created a models in which object properties had a separate columns and in which all of the tables were linked through normalized relations. One of the major challenges that this created is that we had not realized originally that the business and users were not linked with one another. Rather the only relationship the two shared was the reviews contained both a user and review id. This means that if we wanted to see which businesses a user had reviewed, we would need to first find the reviews that the user made and then get the business ids from there. Same thing for businesses. Luckily, we were able to ensure that our database would not result in any broken relations by selecting only the users and reviews that were related to 10 businesses that we had picked form the Yelp Academic DataSet. Thus we decided to remove the friends table in order to guarantee that there would not be any broken links to the different users.

III. Implementation

A. 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”.

Our Heroku database have a limitation. We can only have 10,000 row in the database. One of the attribute for the User Model was a friend list, which contain a list of friends the user have. Each of our user has hundreds of friends, we end up with 11,842 rows after inserting everything into the database. What we end up having to do was remove the FriendList Attribute from User in order to keep the database under the limit. One other thing we did was removing the neighborhoods attribute for Business Model, since none of the Business have neighborhoods values, although it was provided in the original JSON.

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. (See below for the simplified view.)

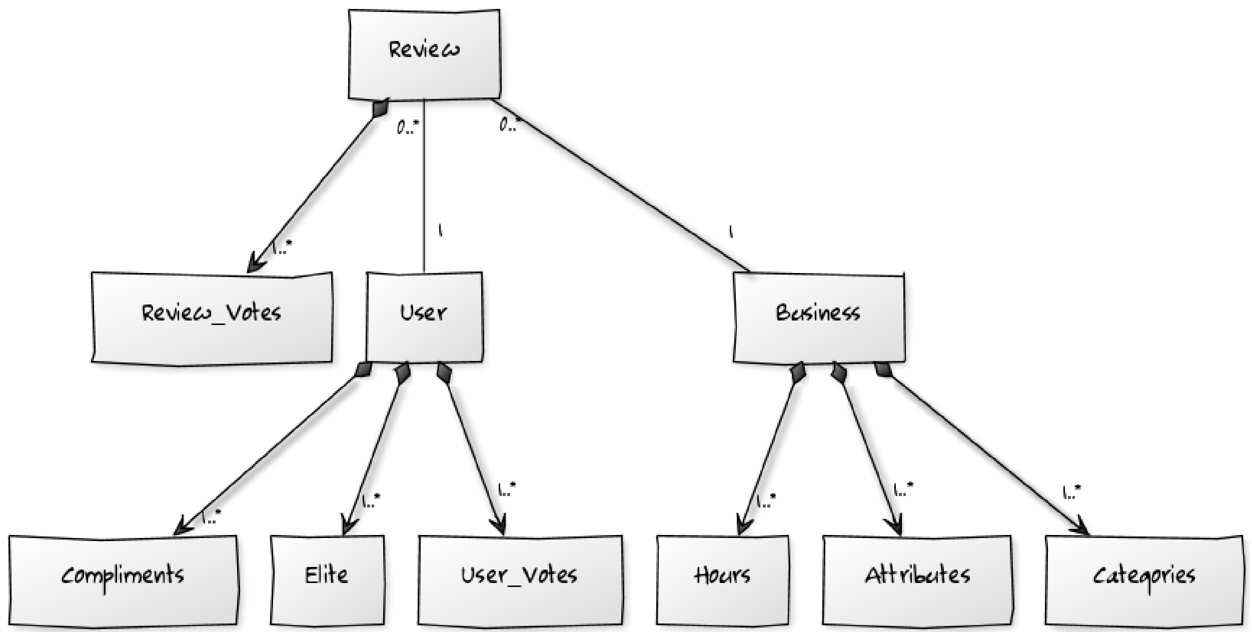


Figure 1. Django models UML

i. Django Views & Templates

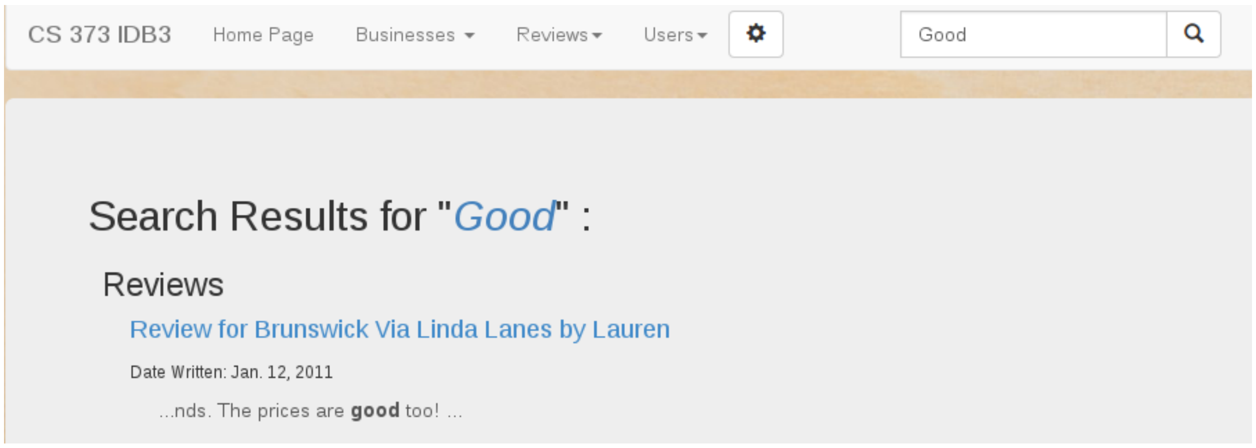
We used Django's templating language to create the pages of the website dynamically. We wrote a different html template for each of the four base pages we had; splash, business, review, and user. Django would then populate the pages based on what inputs were passed to it through the caller's url. When 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.

One of the challenges that we had when designing the pages was determining how to represent our multi-value attributes, attributes whose values were more attributes. (We will refer to these attributes of other attributes as inner attributes.) In the end we decided on creating two dictionaries, one for the single-value attributes and one that contained the multi-value attributes where the name of the attribute was the key and the value was a dictionary containing the inner attributes for that specific key.

However, one of the nice things about this method of using the the views to serve our web pages is that we can manipulate the data in the back-end without worrying about what data the user can or cannot see. This would work really well in situations where one is working with sensitive information such as credit card information or account information.

B. Search:

We chose to implement our own search engine by using a Django form to submit a search query.   The form submits two pieces of information, the search query and a hidden input, format, which determines how the output will look. We implemented two formats. The ‘pretty’ format serves up the entire html document, rendering the search results in a human readable way. Setting the format to ‘json’ will produce just the json associated with the search query. We used this for our unit tests. We compared the search json results with json objects we had pre computed. Django filters the Review, Business, and User objects based on the name property for the Businessses and Users, and the text property for the Reviews. If a review’s ‘text’ involves any of the words in the search query it will be displayed in the search results. If a Business or User name is in the search query it is included in the search results. These queries are performed in a Django view made specifically for the search capability. We used Javascript to highlight the search terms used for the query. The Javascript looks for the key words and if it finds one will take a substring of the surrounding text. It replaces the word with a bolded version of it.



Search Results for "*Good*" :

Reviews

Review for Brunswick Via Linda Lanes by Lauren

Date Written: Jan. 12, 2011

...nds. The prices are **good** too! ...

Picture 1. Example of Search

C. SQL:

The five SQL queries are included in two text documents. Static/txt/InterestingQueries.txt lists the queries and static/txt/InterestingQueriesOutput.txt lists the queries and their output. The latter is linked on the home page. Several of the queries were drawn as graphs that cycle through the carousel.

The queries were drafted in pgAdmin and run through pSQL. They each begin with a question and produce an answer:

1. Does the West side of Phoneix have a higher average rating than the East side?
    a. Yes, by a few hundredths of a star.
2. Do people with more compliments give higher ratings?
    a. People with many and few compliments give higher ratings - people with a

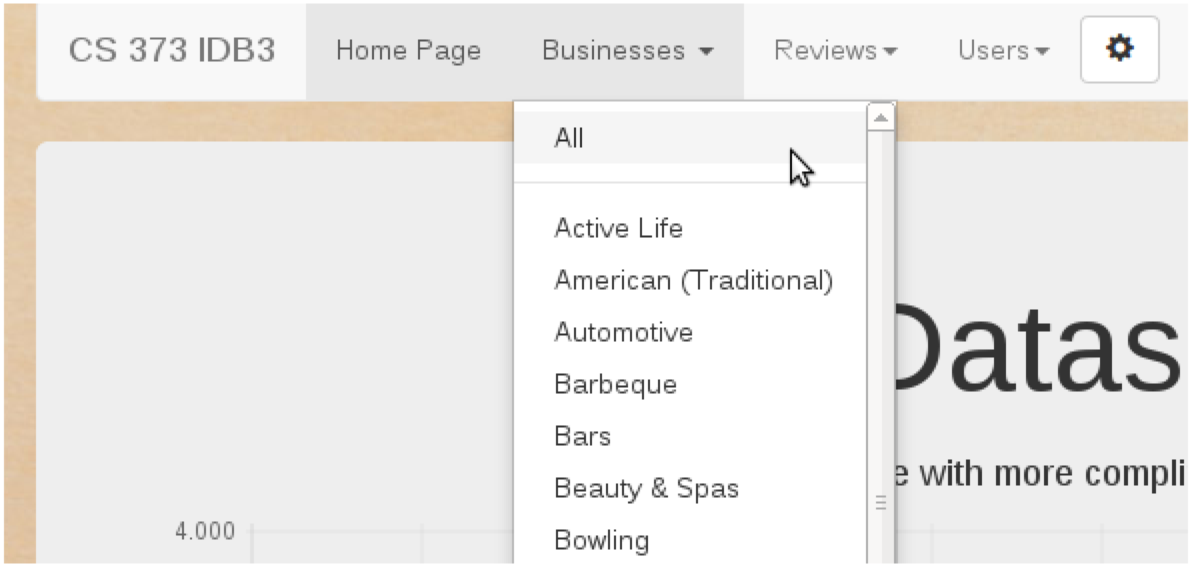
few compliments give the lowest rating.

3. Are businesses with longer hours more popular?
    a. No - businesses with longer hours have consistently lower ratings.
4. Do people with more consonants in their name have more fans?
    a. No - there is no correlation between these two properties.
5. What are the most common words used in reviews?
    a. Simple words like the, and, to. Interestingly the most popular color is blue.

D. Site Navigation:

In phase 2 our site was still organized in a very rudimentary way. The data was divided into 3 sections (Businesses, Reviews, Users) that would list every record in the database. Clicking on a record would lead you to the record's data page. While complete and simple, this approach was overwhelming for first time visitors.

In phase 3 we implemented a more intuitive top level navigation to guide new users through the site. The top 3 links have been changed to drop downs that categorize its dataset. The businesses tab lists the available categories, the reviews lists the 5 star categories, and the users are organized by their first letter (A-Z). These links lead the user to a reduced set of related records. The original "all list" functionality has been preserved with an "All" link at the top of each drop down list.



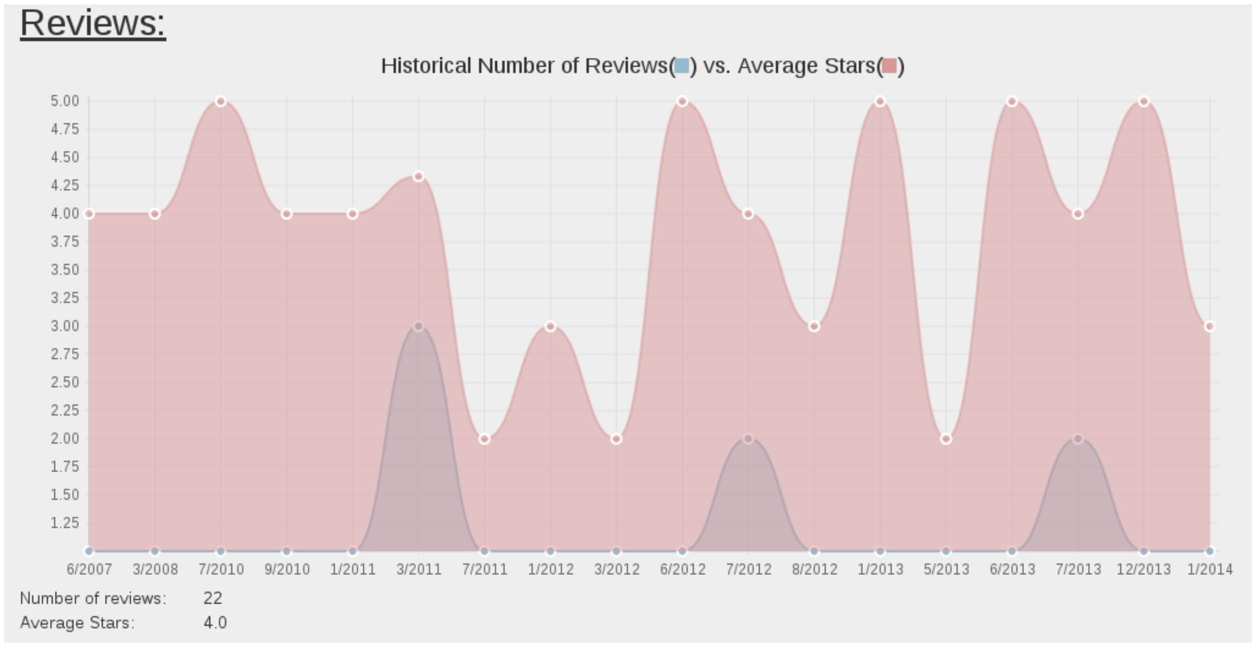
Picture 2. Example of Business Categories

E. Widgets:

Because our site contains a large amount of numerical, historical, and categorical data, we chose to embed charts throughout the pages to present and summarize information. We relied on the Charts.JS framework to render html5 charts using canvas elements. Examples include:

- The home page lists several charts based off of our 5 interesting queries data. These cycle via a bootstrap carousel control.
- Business pages have a line chart contrasting the volume of reviews vs. ratings over time. (See **Chart 1**)
- Review pages have a pie chart that break down the proportion of compliments a review received. If there are no compliments, a chart is not displayed.
- User pages have 2 small pie charts breaking down user compliments and votes. Again if there is no data available, no charts are displayed.

Additionally, the business page features a google map centered at the business's location. It uses a heat map overlay to visualize review volume and ratings. This feature is provided through the Google Maps API.



Number of reviews:	22
Average Stars:	4.0

Chart 1. Volume of reviews vs. ratings over time chart on the Business page

III. Tests

A. Unit Tests:

For our API tests, we created a test for the seven different combinations of HTTP methods and API endpoints for each of the three resources. In addition, we created a test to get the GPS coordinates for each business, as well as some border cases, which gave us a total of 28 unit tests for the API. 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 created dummy values to use for our tests by modifying objects that already exist in our database. This entailed changing the id for the particular resource as well as some of its other fields. After running into problems with the order in which the tests were being executed, we discovered that the tests are ran in alphabetical order. To fix this, we changed the test method names to where the delete request was last in order to delete the dummy values at the end.

For the search unit tests, we used the json response format for the search queries. We compared what was returned from a get request of the search query to some precomputed values. Due to our implementation of Search we weren't able to easily test them like the api calls. We had to set up a 'format' in the form to represent how the data would be returned to the user. When the format is 'json' the view will only return a json representation of the search result. When the format is anything else the search results will render normally.