UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

Computer Science 334
Project 2 Report

Group 14:
S. Ajimudin - 21689326
A. Karbanee - 21865728
P. Marais - 22078754
S. Meinie - 19827784
K. Norton - 19142838
A. Nutt - 19933150

24 April 2020

Contents

1 Introduction

1.1 Problem Statement

Use the Yelp dataset to develop a web application that will make restaurant recommendations. The recommendations will be based off of criteria specified by the user. This criteria is: Cuisine served at the restaurant, the city that the restaurant is located in, the day and time that the user would like to visit the restaurant.

1.2 Requirements

The project must satisfy the following requirements:

- A restaurant recommendation engine must be developed
  - The Yelp dataset must be used
  - The top restaurant, for a specific city and cuisine, must be returned. It is based on the highest value for "stars", and if there is a tie, the one with the highest "review count"
  - The recommendation for the top restaurant must take into account trading hours
  - For the top restaurant, return the review with the most votes for "useful"
  - From the most useful review, return 50 (if possible) friends and friends of friends of the user/reviewer
  - Return 5 more restaurants (if possible), offering the same cuisine and from the same city as the top restaurant. These restaurants must have been reviewed by a user/reviewer present in the list of friends and friends of friends.
- Neo4j must be used
- A web application must be created
  - the web-app will be developed using Jinja and Flask
  - the web-app shall receive user input
    - Desired cuisine
    - Desired city
    - Desired time of day to visit restaurant

- the web-app shall return information relevant to the input
  - Display the top restaurant's name, full address, stars and review count
  - From the top restaurant, display photos with the label "food" and their caption
  - From the top restaurant, display the text of the most "useful" review in the last two years. Also display the "name" of the user/reviewer and "stars" of the specific review
  - Display 5 more restaurants (if possible), offering the same cuisine and from the same city as the top restaurant
- the web-app must be visually pleasing and user friendly
- the web-app must be responsive

2 Solution

2.1 Domain Model

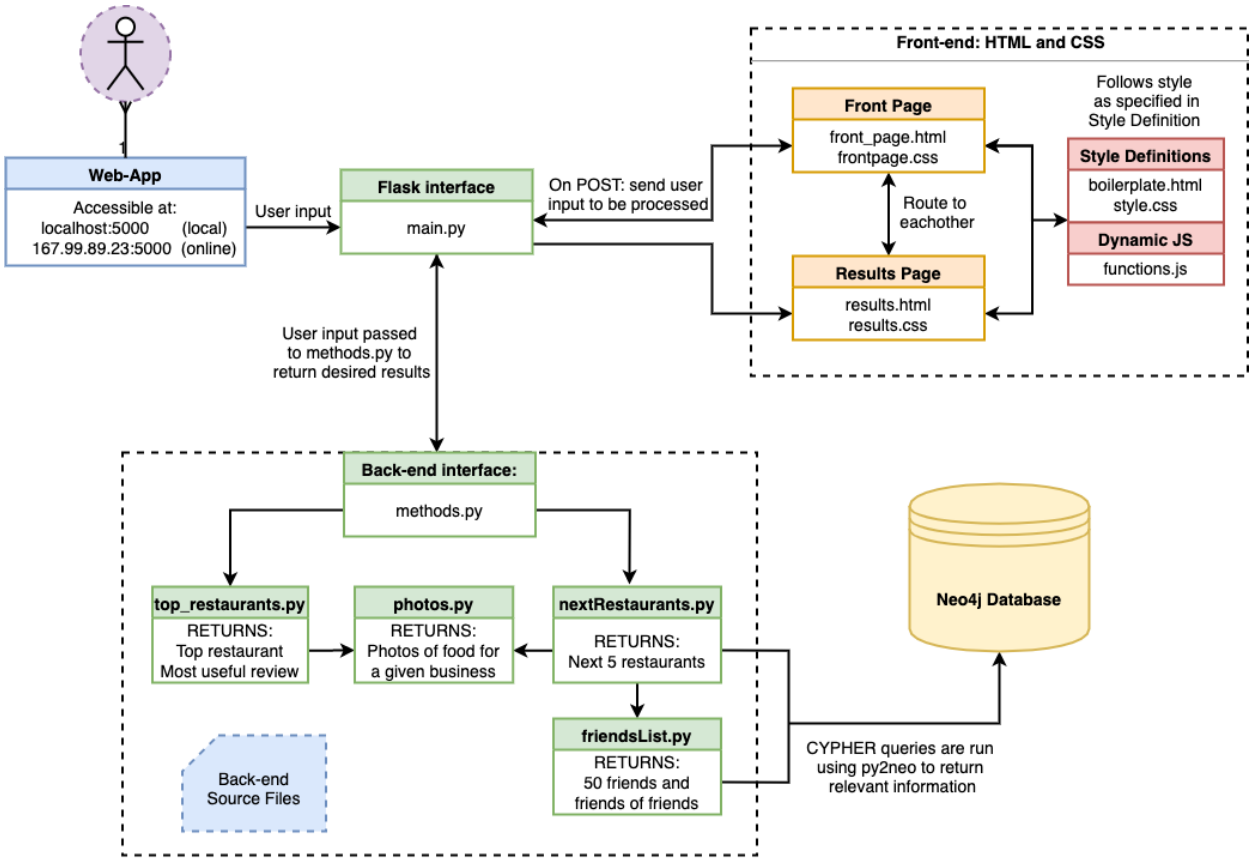

Figure 1: The domain model illustrating the interaction between different components of the system

2.2 Dataset used

The full Yelp challenge dataset is large and contains many attributes we do not require. The dataset is processed to reduce it to a manageable size by process of normalization and feature selection. The only files we consider are `business.json`, `review.json` and `user.json`. The selected features are shown in Table 1

Table 1: Selected features

Business	User	Review
business_id	user_id	review_id
name	name	user_id
address	review_count	business_id
city	friends	stars
state		date
postal_code		text
stars		useful
review_count		
is_open		
categories		

2.2.1 Normalized Files

The normalization of data is done in 4 steps

1. From `business.json`, we extract only the businesses that contain the category 'Restaurants', that are currently open for business, and is referenced at least once in `photo.json`. The result is saved into a new file `business_norm.json`. The business hours are saved into a separate file called `business_hrs.json`, which only contains the business ID and business hours.

2. `review.json` is investigated to cross-reference the businesses included and the users that have made reviews of them. A list of user IDs is compiled.

3. The list of users from the previous step, is used to normalize `user.json`. Only users that have made reviews of the businesses in `business_norm.json` are extracted. Hereafter users that have no friends are removed, and friends that are not in the list of users that have made relevant reviews, are removed. The result is saved into a new file `user_norm.json`.

4. From `review.json`, only the reviews that are related to businesses left in `business_norm.json` and the users left in `user_norm.json`, are extracted. The result is saved into 2 new files `review_norm.json` and `review_text.json`, the latter only containing the review ID and review text.

The size of the files after normalization is as follows:

- `business_norm.json` contains 21 569 businesses from 709 different cities, size 6.5 MB
- `user_norm.json` contains 462 747 users, size 258.5 MB
- `review_norm.json` contains 1 640 698 reviews, size 290.4 MB
- `review_norm_withtext.json` contains 1963714 reviews, size 1.3 GB

2.2.2 Subset Files

Creation of the subset follows the same steps as the normalization, but with more specific criteria. Multiple different subsets are created for testing purposes.

The different subsets are created as follows:

1. The city with the most restaurants (Toronto) is found, and only restaurants in that city (and thus also only relevant users and reviews) are included.
    - `business_sub.json` contains 2705 businesses, size 799 kB
    - `user_sub.json` contains , size 9.5 MB
    - `review_sub.json` contains 123 325 reviews, size 21.8 MB
2. Only 10% of data from the original .json files is included.
    - `business_sub.json` contains 2092 businesses from 209 different cities, size 631 kB
    - `user_sub.json` contains 63 153 users, size 38 MB
    - `review_sub.json` contains 141 681 reviews, size 25.1 MB

For the end goal of the project, it was decided that we will use the 10% dataset, as it is small enough in size to run fast queries on, but still complex enough to return correct and interesting results. The diversity in results makes the project relatable to a larger audience.

2.3 The Neo4j Database

The **10% Data Subset** is loaded into the database by running `CYPHER` queries with use of `py2neo` and the `APOC` plugin for Neo4j.

The following diagram illustrates the different types of nodes and edges that can be found in the database.

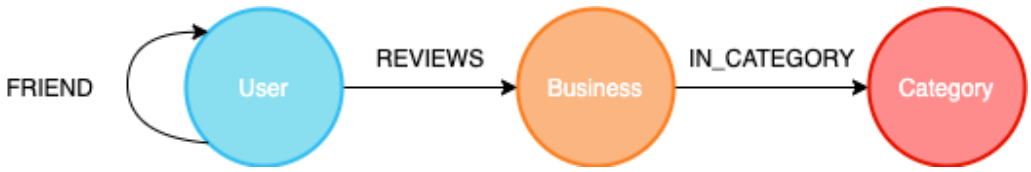

Figure 2: The interaction between nodes and edges

2.3.1 Entity Properties

The entity properties are described in the table below. The edges `IN_CATEGORY` and `FRIENDS` were omitted, as they contain no attributes.

Node: `User`	Edge: `REVIEWS`	Node: `Business`	Node: `Category`
id	review_id	id	id
name	date	name	
review_count	stars	address	
	useful	city	
		state	
		postal_code	
		stars	
		is_open	

2.3.2 Database Size

- Total amount of nodes: 104 673
  - `User`: 102 317
  - `Business`: 2092
  - `Category`: 264
- Total amount of edges: 941 522
  - `REVIEWS`: 134 830
  - `FRIEND`: 796 626
  - `IN_CATEGORY`: 10 066

2.4 How missing values and ties are treated

2.4.1 If there haven't been any reviews in the last two years

As any reviews older than 2 years are most likely no longer relevant, if a restaurant doesn't have any reviews from the last 2 years no reviews will be displayed. Instead, a message saying "There haven't been any reviews in the last two years" will be displayed.

```
#if there haven't been any reviews in the last 2 years
most_useful_review = {
           "Most Useful Review Author": None,
           "Stars of most useful review": None,
           "Most Useful Review": "There haven't been any reviews in
               the last two years"
       }
```

We do however need to display similar restaurants according to the "top reviewer's" friends and friends of friends. In this case, we find the all-time most useful review (most votes for 'useful' and disregarding 'date') and use the user ID of the user who made this review.

2.4.2 Ties with stars when finding the top restaurant

When there is a tie in stars when sorting the list of filtered restaurant, the review count is used (according to the guidlines) to break the tie as in the code below.

```
    sort = sorted(filtered, key = lambda kv:(kv['stars'],
        kv['review_count']), reverse=True)
    if filtered == None or len(filtered) == 0:
        return None
    else:
        return sort[0]
```

The process of gathering the next best similar restaurants, is also based on this principle.

2.4.3 Missing values for opening hours

After filtering through all of the restaurants in the given city, that provide the specified cuisine we then filter those restaurants for the given day and hour. If a restaurant doesn't have any opening hours available for the specified day or for any days, we assume that the restaurant is in fact open at the specified time. As you can see in the code below (code snippet from `top_resturant.py`), if there are no hours (hours == None) or the day is not in the hours list (`day not in hours`), the restaurant still gets appeneded to the filtered list (in the else statement).

```
        for resturant in resturants_hours:
        hours = resturant['hours']
        if hours != None and day in hours:
            if hours[day] != None and hours[day] != 'None':
                hours1 = hours[day].split('-')
                min_ = (hours1[0]).split(':')
                max_ = (hours1[1]).split(':')

                min_time = datetime.time(int(min_[0]), int(min_[1]),
                    0)
                max_time = datetime.time(int(max_[0]), int(max_[1]),
                    0)

                if time_in_range(min_time, max_time, time):
                    #find the resturant with the corrosponding
                        business id
                    for business in businesses:
                        if business['business_id'] ==
                            resturant['business_id']:
                            filtered.append(business)
        else:
            for business in businesses:
                        if business['business_id'] ==
                            resturant['business_id']:
                            filtered.append(business)
```

On the front-end side, if a user does not enter the day or time, we assume the day and time to be the current time of the search request (the code snippet below is from main.py).

```
city = request.form['city_input']
day = request.form['day_input']
category = request.form['category_input']
time = request.form['time_input']
if not day :
    days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
        'Friday', 'Saturday', 'Sunday']
    d = datetime.today().weekday()
    day = days[d]
if not time :
    t = datetime.now()
    time =
```

3 Visual appeal and user experience design

The interface for the website consists of two pages. The first is the front page of the website and the second is the results page. With the design of the website, we aimed to create the user interface in such a way that the content was well spread out, to avoid the clutter and confusion that comes with too much information being displayed to the user at once. Thus, the pages consist of multiple panels and buttons. The pane;s are used to group together like-content, and the buttons used to navigate. Bearing this in mind, we ensured that the visual experience was not to complex, as it is possible for users to get lost in a website, if there are too many pages, panels, dashboards, etc.

3.1 Boilerplate

Both pages extend a Jinja template, called the boilerplate, which displays the title and footer elements on both pages. This ensures visual consistency on both pages, with the same basic content displayed on the pages. The title element simply contains the title text, as well as a short description of the website (or a catchphrase). The footer element displays the names of our group members, ordered by surname in alphabetic order. This footer element is kept at the bottom of the page in full screen mode, regardless of where one is on the page, using absolute positioning in CSS.

3.2 Front Page

The front page is the first page that loads when accessing the website. It consists of a small greeting message followed by basic search panel, to be used to search for your restaurant. The search is done through 4 separate input boxes, to ensure that communication between the user and the server regarding the search parameters is unambiguous and clear; from both the user's perspective and from the perspective of the program, which can subsequently parse the search parameters with ease.

Figure 3: Screenshot of the front page.

3.3 Results Page

The results page is divided into 3 sections: The top restaurant result, the top five other recommended restaurants and the search bar The Top Restaurant results is the default display of the page, as it is the main focus and purpose of the search. The search results is displayed in three panels for ease of view. These panels display the restaurant details, the most useful review as well as the pictures from the restaurant. Of importance is the display of the pictures. In order to allow for all pictures found to be displayed to the user, the pictures are separated into groups of 4. Only 4 of the images from the set of restaurant's images are displayed at once, and using the arrow button on the panel, one can view the other pictures. This was accomplished through the use of Jinja to compile python code within the HTML form to allow for a more dynamic and streamline display of the images, rather than displaying them all on one panel, or not displaying some. Furthermore, if there is no search result, or no reviews for the restaurant, a subsequent message is displayed to the user. The second section is the Other Restaurants result. This displays the basic details of recommended restaurants, along with a single image from the restaurant. Up to 5 restaurants can be displayed, with fewer than 5 being displayed if five are not available. If there are no recommended restaurants, a subsequent message is displayed. The third section is the search bar. This can be used to allow the user to search the database for another restaurant. It uses the same format as the search bar in the front page, and serves the same functionality.

Figure 4: Screenshot of the results page, displaying the top restaurant and its most useful review.

Figure 5: Screenshot of the results page, displaying the other top restaurants based off of friends of friends.

3.4 Mobile Mode

The full website is supported on mobile. Through the use of responsive CSS, all elements scale down to be visually appealing from a mobile perspective

4 How to run test cases

4.1 Using Our Code From GitLab

Once you have downloaded the git repo at
`https://git.cs.sun.ac.za/Computer-Science/rw334/2020/group-projects/group14-rw334`
and are in the directory, follow the instructions below from your terminal.

Instructions:
0. Open the folder src/neo4j and follow the README. This will set up the database and populate it with the data from src/dataset

1. Setup a virtual environment, using the `requirements.txt` provided:

   1.1 If you have not already install virtualenv: `sudo pip3 install virtualenv`
   1.2 Now create a virtual environment: In the main directory type: `virtualenv venv`
   1.3 Lastly activate your virtual environment: `source venv/bin/activate`

2. Connect to the neo4j database with the following details:
   username: neo4j
   password: YourPassword

3. run `python3 main.py`

4. Follow the link to the website produced

4.2 Using Our Server

Enter the following into your browser to be taken to our website
`http://167.99.89.23:5000/`

5 Distribution of work and Contribution of each member

Backend

P. Marais (22078754)

Consider users/reviewers from the list of friends, and friends of friends (as in `user.json`) of the original selected reviewer/user. Based on highest review count for users/reviewers, select (if possible) 50 users/reviewers.

S. Meinie (19827784)

I created files `top_resturant.py` and `photos.py`. Within these files I find the name, the address, amount of stars and review count of the top restaurant (based off of the stars, and if there is a tie, the review count) that satisfies the given criteria and then return it to the client-side.
Return the review, the amount of stars of the review and the author of the review with the highest value for 'useful' in the last two years.
Select randomly a number of photos of the proposed restaurant with the label "food" (as in `photo.json`), and display these photos with captions (as in `photo.json`).

A. Nutt (19933150)

Database administrator, normalization of data and extracting a data subset to be loaded into the Neo4j database. Extract relevant business categories and cities from data subset to return to frontend, to populate drop-down lists. Writing `nextRestaurants.py` to display the names of 5 more restaurants (if possible) offering cuisine in the same category (and also selected city) as specified by the user of the web application, based on highest "stars" value (of reviews made by any of the 50 users/reviewers). Compiled `methods.py` for easy communication between front-end and back-end. Wrote code to return the 'all-time best review' user ID, if there wasn't a recent review within 2 years.

Frontend

S. Ajimudin (21689326)

Connecting the user interface with the backend i.e.the server, database and application of the project. Running tests and understanding the functionality of `top_restaurant.py` as well as `nextRestuarants.py`. Editing code of the user interface and backend to allow for information flow.
There were several problems that arose when handling backend and connecting it to the user interface, which have been fixed. The layout of the

`top_restaurant.py` `nextRestuarants.py`, the subset of data used by `nextRestuarants.py` was too large to be handled by `top_restaurant.py`. The subset of data used by the backend did not include all categories and cities.

A. Karbanee (21865728)
Creation of the interface to be used in by the program, for which the code is stored in the HTML files within `/templates`. Creation of the Jinja template design (in `/templates/boilerplate.html`) within the `/templates` folder to be extended to all other HTML files, to allow for a more streamline coding process using template inheritence. Creation of multiple CSS files to create more appealing visual content. Creation of JavaScript functions in `/static/functions.js` to create more responsive, dynamic pages. Providing means by which users can input their search parameters, as well as means by which user's can view the content of the search result.

K. Norton (19142838)
Setup of a remote server to allow the web-application to be viewed online. Uploaded the necessary files to the web server and maintained the server so it remains live until marking is complete. Brought together all the resources necessary from back-end and front-end to compile the video presentation. Testing and then suggesting changes/bug fixes to polish up the website. Added final, small details to the High-Level Domain Layout of our project