

Project: Food Map of Hong Kong

Instructor: Prof. John C.S. Lui

Due: 23:59 on Thu. Nov. 30, 2017

1 Introduction

Hong Kong is a paradise of cuisine. In such a small wonderland, you could find various kinds of foods all over the world. With so many choices, what to eat is a big headache everyday for Alice, who has been living in Hong Kong for only one year. Alice is our friend, and we want to help her to find some really good local restaurants. After a short discussion, we decide to start a project and build a “Food Map” of Hong Kong that knows good restaurants which match our needs. At the same time, our another friend Bob, plans to invest on some restaurants in Hong Kong. In order to earn a free drink in Bob’s restaurants in the future, we’d like to help Bob to carry out a survey about restaurants in Hong Kong just using this “Food Map”.

We notice that some awesome APPs have already provided services to help us select restaurants. Openrice is a good example. It enables us to know the details of each restaurant by collecting many reviews from real customers. However, when Alice does not know exactly what to eat, she finds it time-consuming to do the search on Openrice. Moreover, she does not subscribe any mobile data plan, so she needs the offline services especially when WiFi is not available. Having observed the extraordinary data in Openrice, we decide to utilize Openrice data and build our own Food Map.

“From here to success, I Just Need a Programmer!” Fortunately, by taking CSCI2040, you become the programmer who carries the team to success. In this project, you will write a web crawler to scrape data about restaurants in Hong Kong from the website of Openrice. Then, you will implement a Food Search Engine to help Alice to decide what to eat. Also, you can do data analysis and deliver a report of Hong Kong restaurants to Bob. (Optional: after you finish the project, you may select some restaurants by your scripts to celebrate)



Figure 1: Various Foods in Hong Kong

1.1 General Notes

1. 📄 means that you need to write some Python scripts in a corresponding *.py file;
2. 📝 means that you need to write down your answers to the project report file `report.pdf`.
(Note: There is no strict format requirement for `report.pdf`. You can just write down your answers to a Microsoft Word file `report.docx` and then save it as `report.pdf`)

2 Openrice Crawler

For various needs, we want to grab the data from the Internet. Basically, crawling on the Internet is to automatically record what you see on the browsers. Theoretically, you may crawl anything that is visible to you. As a customer, we are able to know the list of restaurants in certain area, and detailed information about each restaurants. In this part of the project, we will use these publically available data automatically by crawler written in Python.

2.1 Introduction to Scrapy

We use the crawler framework **Scrapy**¹. A typical crawler first gets the text of a web page, and then parses the text to collect information of interests. **Scrapy** is a framework that separates the procedure of getting the web page and the procedure to extract useful information. You are recommended to read some documents and examples of **Scrapy**. This video² may also be helpful.

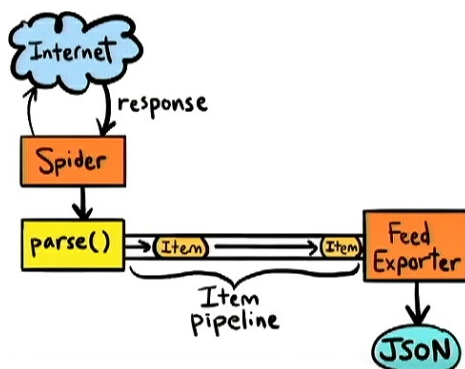


Figure 2: Work flow of Scrapy

2.2 Get restaurants urls (15 pts)

From this section on, we want to crawl from www.openrice.com. Since writing a crawler means writing a script on how to read the web pages, we suggest you to carefully read the Html pages of www.openrice.com first. If you use **Chrome** as your browser, you could press **F12** to see the source Html code of a web page. Our final goal is to get detailed information for a list of restaurants. In particular, every restaurant corresponds to a single web page. Therefore, we should first get a list of urls³ for the restaurants (Section 2.2). Then, for each restaurant in this list, crawl the corresponding web page (Section 2.3).

We have already provided a framework in `openrice_urls_spider.py`. All the provided code is in `project.zip`. You are required to fill in the missing parts to accomplish the following tasks:

1. Get the urls of all restaurants in Openrice for the four places: Shatin, Mong Kok, Tsim Sha Tsui and Causeway Bay. The websites for these places are:

¹<https://scrapy.org/>

²<https://www.youtube.com/watch?v=-JzH8TewqxI>

³<https://en.wikipedia.org/wiki/URL>




- Shatin: `www.openrice.com/en/hongkong/restaurants?where=shatin`.
- Mong Kok: `www.openrice.com/en/hongkong/restaurants/district/mong-kok`.
- Tsim Sha Tsui: `www.openrice.com/en/hongkong/restaurants/district/tsim-sha-tsui`.
- Causeway Bay: `www.openrice.com/en/hongkong/restaurants/district/causeway-bay`.

For each of the above locations, there are 17 pages of restaurants displayed in Openrice, and the number of page is specified by `?page=[some number]` as suffices in the urls. For example, the 13th page of restaurants in Mong Kok maps to the link `https://www.openrice.com/en/hongkong/restaurants/district/mong-kok?page=13`. In summary, you should crawl $4 \times 17 = 68$ webpages and record the urls of each restaurants which appear in these 68 pages.

2. Save the list of urls in the intermediate file named `openrice_urls.txt`. Each url is stored in a single line in this file. For example, this file could be like the following:

```
www.openrice.com/en/hongkong/r-la-terrazza-bar-grill-sha-tin-western-r137877
www.openrice.com/en/hongkong/r-la-kaffa-sha-tin-western-all-day-breakfast-r84196
... # more urls
```

Still don't know what to do? Don't worry. We now give you more detailed guidance on how to fill in the blanks. You need to do the following:

1.  In the function `start_requests(self)` in `openrice_urls_spider.py`, write a loop to generate a list of urls.
2.  In the function `parse(self, response)`, use css selectors⁴⁵ like `xx.css(xx_selector)` to search for the restaurants' urls that are our targets. *Hint:* the function `extract_first()` would be helpful.
3.  Save the list in file `openrice_urls.txt`.
4. After you finish the above three parts, you can run the spider by typing the command line in the folder of your scripts:

```
scrapy runspider openrice_urls_spider.py
```



Now you will get the urls in the file. Note that Openrice may dynamically change the displayed restaurants, so don't worry about the consistency of crawled urls in different runs. Also notice that we add a **header** for each **request** of web pages. To know why we add such header, you could try removing it and see what will happen.

2.3 Get detailed information of restaurants (15 pts)

Now you have some idea on how a crawler works, and you also have the intermediate file that contains the urls of the restaurants. In this part, you will implement a crawler by yourself in `openrice_spider.py`.

⁴https://www.w3schools.com/cssref/css_selectors.asp

⁵<https://stackoverflow.com/questions/16788310/what-is-the-difference-between-css-selector-xpath-which-is-betteraccording-t>

1.  Crawl all the web pages in the list `openrice_urls.txt`. For each web page like `www.openrice.com/en/hongkong/r-chan-kun-kee-sha-tin-guangdong-r7918` (Chan Kun Kee), find the following fields:
 - **cuisine.** (food style, e.g. “Guang Dong/Dai Pai Dong” for Chan Kun Kee)
 - **price-range.** (e.g. \$51-100 for Chan Kun Kee)
 - **address.** (e.g. latitude: 22.3884, longitude: 114.1958)
 - **ratings.** Only the overall rating. (e.g. the rating of Chan Kun Kee is 3.5)
 - **number of reviews.** We should find the number of good, medium and bad reviews. (e.g. the number of good/medium/bad reviews for Chan Kun Kee are 216/95/38 separately. Notice that the sum of good, medium and bad reviews is not necessarily equal to the number of text reviews)
 - **district.** Which of the four districts this restaurant belongs to (e.g., the district of Chan Kun Kee is Shatin)
 - **url.** The url for this restaurant.
2.  Store these detailed information in the data file `openrice_data.json` as a JSON⁶ list where each element in the JSON list contains the detailed information of a restaurant. An example format of this output file is as follows:

```
[
  {
    "name": "Chan Kun Kee",
    "cuisine": ["Guang Dong", "Dai Pai Dong"], # a list of cuisine types
    "price-range": "$51-100",
    "address": [22.3884, 114.1958], # first latitude, second longitude
    "rating": 3.5,
    "reviews": [216, 95, 38], # [good reviews, medium reviews, bad reviews]
    "district": "Shatin",
    "url": "www.openrice.com/en/hongkong/r-chan-kun-kee-sha-tin-guangdong-r7918"
  },
  ... # and other restaurants
]
```

Similar to Section 2.2, we launch our crawler after coding:

```
scrapy runspider openrice_spider.py
```

It may take several minutes to crawl all the data. Some of the url in the urls list may be invalid or direct to other pages. For these errors, just ignore and get information for as many restaurants as possible. Now, you will have all the data in `openrice_data.json` (around 1000 restaurants). Using this data set, you are able to implement some interesting functionalities in the following sections.

⁶<http://www.json.org/>

3 Build the Backend Food Search Engine

With all crawled data at hand, one natural question is how to utilize these data to do something meaningful. One typical use of the data is to build a “food search engine” to do filtering, ranking and recommendations. We can think of the food search engine as the backend behind the website, which takes crawled data as input and return meaningful results for users.

3.1 Overall Structure

The food search engine can be wrapped as a `Food_Search_Engine` class in a python file `food_search_engine.py`. This file is provided and you just need to modify the functions to make the food search engine function work as specified. **Do not change file name, function prototype.** The overall structure are as follows:

```
class Food_Search_Engine:
    # original data from crawled json file
    original_data = []
    # the result after filter/ranking/similarity
    query_result = []
    # more data structures can be added here

    def __init__(self, json_file_name):
        self.load_data(json_file_name)
        self.reset()
    def load_data(self, json_file_name):
        # we provide the code
    def filter(self, filter_cond):
        # your code here
    def rank(self, ranking_weight):
        # your code here
    def find_similar(self, restaurant, similarity_weight, k):
        # your code here
    def print_query_result(self):
        # your code here
    def reset(self):
        # we provide the code
```

The next few parts are instructions about how we develop these functions.

3.2 Loading Data

First of all, we need to load crawled data into our class. The function `load_data(self, json_file_name)` loads data from `openrice_data.json` into `original_data` variable. Once we loaded the data, we can use these data for filtering, ranking and recommending similar restaurants.

3.3 Filtering (13 pts)

From all these restaurants, Alice wants to pick out the restaurants satisfying her requirements. In this section, we will design a function `filter(self, filter_cond)` that takes some required conditions `filter_cond` as input, and store the list of all the restaurants that satisfy the conditions into the variable `query_result`. For example, if our condition requires the restaurants to be in the district in “Shatin”, then any restaurants in “Mong Kok” cannot be in the returned results.

The variable `filter_cond` would be a dictionary where each entry represents a condition. In one entry, the key describes the type of filter and the value is a list of legal values according to this filter. Here, we list the possible conditions and how we represent these conditions.

- **name**: name of a selected restaurant should matches exactly some name in the list. For example, the following condition requires the name of selected restaurants to be either “Chan Kun Kee” or “PizzaExpress”.

```
'name': ['Chan Kun Kee', 'PizzaExpress']
```

- **name_contains**: name of a selected restaurant should contain some string in the list as substring. For example, a restaurant named “Chan Kun Kee” satisfies the following condition while the restaurant “Kun” doesn’t.

```
'name_contains': ['Chan Kun', 'Kun Kee']
```

- **cuisine**: the cuisine (food type) of a restaurant should contain some cuisine in the list. For example, a restaurant providing both Japanese and Guangdong food satisfies the following condition.

```
'cuisine': ['Japanese', 'American']
```

- **district**: the district of a selected restaurant should be one of the districts in the list.

```
'district': ['Shatin', 'Mong Kok']
```

- **price_range**: the price range of a selected restaurant should be a **subset** of the specified price range. For example, a restaurant with price range \$51-100 satisfies the following condition.

```
'price-range': '51-200'
```

Note that we assume the lowest price is \$0 and the largest price is \$1000. This deals with the problem that some crawled data has the price-range ‘Above \$800’ and we can view it as ‘\$800-1000’. Similarly, ‘Below \$50’ can be viewed as ‘\$0-50’.

- **rating**: the rating of a selected restaurant should be **no less than** a certain level. For example, a restaurant with rating 3.5 satisfies the following condition:

```
'rating': 3
```

Altogether, the Guangdong food restaurant “Chan Kee Kee” located in Shatin with rating 3.5 should be selected, and its corresponding dictionary should appear in the `query_result`, when

```
filter_cond = {'name_contains': ['Chan Kun', 'Pai Dong'], 'district': 'Shatin',  
               'rating': 3.0, 'cuisine': ['Guangdong', 'Indian']}
```

3.4 Ranking (13 pts)

After filtering the result, the variable `query_results` is just a list of all restaurants that meets our demand. To display our results in a sequence, we need to decide the order to display the restaurants. In particular, we will assign a score for each restaurant in which a higher score implies the restaurant brings higher utility. After that, we rank the restaurants based on their scores in descending order. The `rank(self, ranking_weight)` function implements the functionality described above. We will show the details of this ranking procedure below.

3.4.1 Representation of a restaurant

In order to assign a score for each restaurant, we first quantitatively represent a restaurant in a vector. We consider four attributes affecting the score of a restaurant, and a restaurant is formally represented as a vector of four elements $\mathbf{v} = (v_1, v_2, v_3, v_4)$, where

- v_1 is the rating of the restaurant, i.e. rating. For example, $v_1 = 4$ if the rating of the restaurant is 4.
- v_2 is the normalized distance between this restaurant and SHB (latitude:22.417875, longitude:114.207263). For two points A, B with latitude and longitude $(\text{lat}_A, \text{lng}_A)$, $(\text{lat}_B, \text{lng}_B)$, the normalized distance between them is defined as $\sqrt{(\text{lat}_A - \text{lat}_B)^2 + (\text{lng}_A - \text{lng}_B)^2}$.
- v_3 is the expected price of the restaurant. Since we only know the price range $[a, b]$ of a restaurant, we use the average $(a + b)/2$ as the expected price. For example, the restaurant with price range \$51-100 has the expected price $(51 + 100)/2 = 75.5$. Also, as previously defined, we assume the lowest price is \$0 and the largest price is \$1000. This deals with the problem that some crawled data has the price-range 'Above \$800' and we can view it as '\$800-1000'. Similarly, 'Below \$50' can be viewed as '\$0-50'.
- v_4 is the ratio of bad reviews. Besides the rating which represents the average satisfaction of customers, we also care about the extreme case that corresponds to the bad reviews. The ratio of bad reviews is defined as $v_4 = \frac{\# \text{ of bad reviews}}{\# \text{ of good reviews} + \# \text{ of medium reviews} + \# \text{ of bad reviews}}$.

3.4.2 Score of a restaurant

With the vector representation of a restaurant, we are able to quantitatively define the score of a restaurant. In general, the score s of a restaurant is a function of the characteristic vector $f(\mathbf{v})$, where f is some mapping function. Here, we adopt the widely used simple mapping function, that is linear and the score s of a restaurant \mathbf{v} is

$$s = f(\mathbf{v}) = \mathbf{w}^T \mathbf{v} = w_1 v_1 + w_2 v_2 + w_3 v_3 + w_4 v_4.$$

In the above formula, \mathbf{w} is the weighting vector and is passed to the function by parameter `ranking_weight` which is a list of four elements. For example, if we have a weight vector `ranking_weight=[1,0,-0.02,-1]`, then the score for a restaurant represented by $\mathbf{v} = (4, 0.1, 75.5, 0.1)$ is $1 \times 4 + 0 \times 0.1 - 0.02 \times 75.5 - 1 \times 0.1 = 2.39$.

We notice that a positive weight on an attribute means we desire a high value of that attribute. For example, if $w_1 > 0$, then a higher rating v_1 will result in the higher score of a restaurants. Moreover, increasing the absolute value of the weight of some attribute will increase the importance of such attribute to the score.

3.4.3 Sort the restaurants according to scores

⇒ We want these restaurants with highest scores to be displayed on the top. Therefore, we need to re-order the `query_result` so that a restaurant with higher score will have smaller index in the list `query_result` after calling the function `rank()`. If two restaurants have the same score, it is ok no matter which one has a smaller index.

3.5 Recommending similar restaurants (optional: 10 pts bonus)

Note: the following method of recommending by similarity is not necessary to be the best one. You can earn 60% of the bonus points by coding exactly the same as we suggested below. But if you have a better solution, we will give you more than 6 points (up to 10).

After implementing `filter` and `rank` function, we can provide Alice with a list of restaurants ranked by scores to meet her need. However, Alice is still unsatisfied because she must specify all the requirements by her own. In fact, she really wants the script to automatically recommend some restaurants based on her flavor. Hence, we plan to write a function `find_similar()` that will recommend some similar restaurants compared to a certain restaurants that Alice likes.

3.5.1 Similarity of two restaurants

Before we proceed, we define the similarity of two restaurants. Indeed, similarity of two restaurants is a metric that judges whether the two restaurants have similar characteristics. For two restaurants represented by \mathbf{u} and \mathbf{v} (recall Section 3.4.1), the similarity between them is defined as:

$$\text{sim}(\mathbf{u}, \mathbf{v}) = w_1|u_1 - v_1| + w_2|u_2 - v_2| + w_3|u_3 - v_3| + w_4|u_4 - v_4|,$$

where the weight $\mathbf{w} = (w_1, w_2, w_3, w_4)$ corresponds to the parameter `similarity_weight` which is a list of four elements. For example, when `similarity_weight=[1,1,0.04,2]`, then the similarity of two restaurants $\mathbf{u} = (3.5, 0.1, 150.5, 0.05)$ and $\mathbf{v} = (4, 0.1, 75.5, 0.1)$ is $\text{sim}(\mathbf{u}, \mathbf{v}) = 1 \times 0.5 + 1 \times 0 + 0.04 \times 75 + 2 \times 0.05 = 0.9$. Note that the smaller the similarity value of two restaurants, the more similar the two restaurants are.

3.5.2 Find the top-k similar restaurants

⇒ Now, we implement the function `find_similar(self, restaurant, similarity_weight, k)` that returns `k` restaurants that has the smallest similarity (i.e. most similar) compared to the target restaurant represented by the parameter `restaurant`. These `k` restaurants does not include the target restaurants. Note that `k` is a positive integer, and `restaurant` is a dictionary in the format of crawled data in Section 2.3. Also, the return value of this function should be a list of restaurants where a restaurants with smaller similarity has smaller index.

If more than two restaurants have the same similarity, return all restaurants whose similarity are less than or equal to the `k` smallest similarity. Notice that the number of top-`k` restaurants could be more than `k`. For example, we have four restaurants with similarity value 1, 2, 2, 3 respectively, compared to the target restaurant. Then the top-2 similar restaurants are the first three restaurants with similarity value 1, 2, 2.

3.6 Test your food search engine (10 pts)

After finishing all the above functions, the next thing we are interested in is to test our food search engine by some requests from users. Suppose Alice has the following requirements:

1. Find all restaurants which has Japanese food in 'Sha Tin' district with price-range \$50-200.
2. Rank the filtered restaurants by their rating(the higher the better) and break ties by their bad reviews ratio of bad reviews(the lower the better).
3. (optional) Find top-10 similar restaurants compared with the top-1 restaurant from the previous ranking result.

✍️ Put down your result after each step in `report.pdf`. Also, try to analyse the result and bring up some comments or suggestions for the ranking method.

4 Data Analysis and Visualization

Using pictures to understand data is much more efficient than using texts. However, so far all the outputs are text-based. To have a better understanding of the data and expose something new about the underlying patterns and relationships, we need to use data visualization. You may use the Food Search Engine you just implemented.

By default our crawler reads the English names of the restaurants. However, some restaurants only have Chinese names so the crawler returns Chinese characters. If you have problems with displaying Chinese characters, please don't worry, just ignore them. Your grades will not be affected.

4.1 Top-10 Most Reviewed Restaurants in Sha Tin (6 pts)

The number of reviews reflects the popularity. So what is the most popular restaurant in Sha Tin? Visualize the names and the number of reviews of the Top-10 most reviewed restaurants in Sha Tin using an horizontal bar chart.

📝 Write scripts in `top10_reviewed.py` to find the Top-10 restaurants in Sha Tin which have the most number of reviews. Then plot the names and the number of reviews in descending order. The x-axis is for the number of reviews and the y-axis is for the names of the restaurants. *Hints:* For this plot, you can use the function `barh()` in `matplotlib`. Figure 3 is an example plot of the Top-10 most reviewed restaurants in Mong Kok.

✍️ Put down the plot in your `report.pdf`.

4.2 The Distribution of Cuisine Types in Mong Kok (6 pts)

A pie chart is a circular statistical graphic which is divided into slices to illustrate numerical proportion. Now we want to display a pie chart to find out the percentage of cuisine types for all the restaurants in Mong Kok.

📝 Write scripts in `type_distribution.py` to plot a pie chart which shows the proportion of all the cuisine types that can be found in Mong Kok. The categories in the pie chart should be the types (e.g. Japanese, Indian). And there should be a label text for each category on the figure. Since there are so many types, we only focus on the Top-10 types with largest proportions and

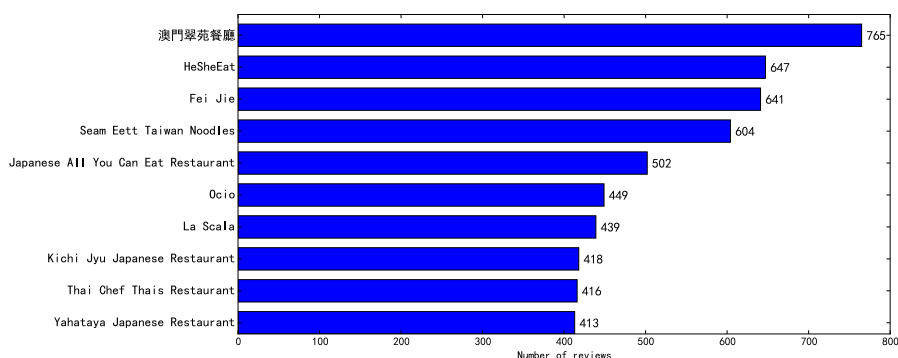


Figure 3: Top-10 most reviewed restaurants in Mong Kok

we categorize the rest types as "others". *Hints:* For this plot, you can use the function `pie()` in `matplotlib`. Figure 4 is an example plot of the proportions of the Top-5 cuisine types in Sha Tin. 📎 Put down the plot in your `report.pdf`, and briefly describe what you observe from the plot.

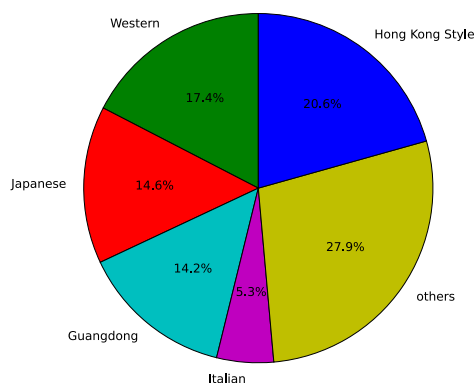


Figure 4: Top-5 cuisine types in Sha Tin

4.3 Relationship Between Price and Popularity in Hong Kong (6 pts)

Alice has found a lot of highly rated restaurants, however, the prices of them are too expensive for her. Is it true that the more expensive the better? The scatter plot is a common way to visualize the relationship between two quantitative variables. Now Alice wants to display a scatter plot to investigate the relationship between price and Popularity.

📁 Write scripts in `price_popularity_relationship.py` to plot a scatter plot which shows the relationship between price and popularity for all the restaurants in Hong Kong. The x-axis is the prices and the y-axis is the number of reviews. Since we only know the range of the price for each restaurant, you are supposed to use the "mean value" of this range as the price. For this plot, you can use the function `scatter()` in `matplotlib`. Figure 5 is an example plot of the relationship

between price and popularity in Mong Kok.

✍️ Put down the plot in your **report.pdf**, and briefly describe what you observe about the relationship between price and popularity from the plot.

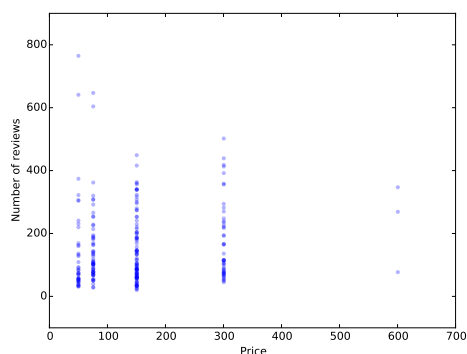


Figure 5: The relationship between price and popularity in Mong Kok

4.4 The Distribution of the number of reviews in Tsim Sha Tsui (6 pts)

A histogram is an accurate graphical representation of the distribution of numerical data. Alice wants to find out the distribution of the number of reviews for all the restaurants in Tsim Sha Tsui.

📝 Write scripts in **reviews_distribution.py** to plot a histogram which shows the distribution of the number of reviews for all the restaurants in Tsim Sha Tsui. The x-axis is the number of reviews (you are free to choose suitable intervals). The y-axis is the number of restaurants. For this plot, you can use the function **hist()** in **matplotlib**. Figure 6 is an example plot of the distribution of the number of reviews in Hong Kong.

✍️ Put down the plot in your **report.pdf**, and briefly describe what you observe from the plot.

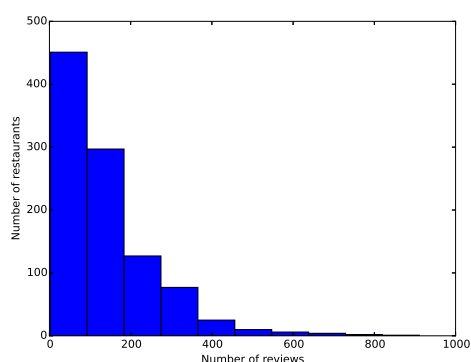


Figure 6: The distribution of the number of reviews in Hong Kong

5 Utilizing open source packages and Google Maps API

Very often, *matplotlib* is not good enough to provide sufficient visualization that can tell a story or solve a problem. However, without sufficient resources, it is sometimes difficult to build a solution from scratch on your own. Thus, in order not to "re-invent the wheel", programmers often utilize software in the open-source projects and make use of publicly available Application programming interface (API). In this final section, you would be able try out the two techniques and tell a good story.

5.1 Gmplot (10 pts)

Gmplot is a open-source library that plots data points onto Google Maps. It generates a html file that renders the data you provide. In order to use this package, you can either make use of pip (python package index) or `setup.py` provided on the github repository to install.

It can be installed with pip (or pip3):

```
pip install gmplot
```

It could also be installed via running the `setup.py` included within the `gmplot` github repository

```
git clone https://github.com/vgm64/gmplot
cd gmplot
python setup.py install
```

When using an open-source package, it is important to try out sample code provided by the author in order to understand how to use the functions. Gmplot provides three main functions `plot`, `scatter`, and `heatmap`. You can run the examples provided on Github (<https://github.com/vgm64/gmplot>) to see what values were taken in and how it can be used.

So while you worked on the previous tasks, Bob decided to open an Japanese restaurant. However, he is not familiar with the market demographic in the four selected areas: Shatin, Mong Kok, Tsim Sha Tsui and Causeway Bay. Luckily, you have the restaurant information for the four areas and it would be a good idea to use a heat-map to demonstrate to give some suggestions.

Write scripts in `gmplot.py` to plot one heat-map of Japanese Cuisine of the four different areas. You should use the Prince Edward Station (22.325222,114.1664163) as the center of the map.

Put down the plot in your `report.pdf` and briefly explain which of these locations should Bob locate his new Japanese restaurant base on the heat-map.

Hints: The default heat-map does not provide a straightforward result, observe `gmplot.py` within the open-source repository to see what additional parameters the package provides for each function.

5.2 Google Maps API (optional: 5 pts bonus)

Sometimes, the tools we want cannot be provided through open-source projects, rather, they are provided as API services by third-party organizations. Google Maps API provides various services such as geo-encoding and place information look-ups. In this final section, we will make use of

Google Maps Distance Matrix API.

In order to use this API service, we need to register for a service Key. This key can be obtained by visiting the Google Maps Direction API web page and select "Get Key". It would prompt you to start a new project as a logged-on user. After confirming the project, a string would be provided as your service key. Each service key allows a user 2500 free usage per day. So we highly recommend you to test it out on a small data set first before applying it to create the final results.

In order to understand the usage of any given API, it is always a good idea to try it out through the examples given on the documentations provided by the service. Please visit this page for documentations: <https://developers.google.com/maps/documentation/distance-matrix/intro?hl=en-us>.

While Bob is setting up his new restaurant, Alice approached you and complained that Openrice only shows the address of the restaurants. She is not able to find a good restaurant in Shatin that can be reached within a hour using the public transport. Alice wants to find food and you recall that Google Maps provides users the function to estimate the amount of time to get to one place. Now you need to help Alice to create a map that shows restaurants in Shatin that she can reach within different time frame. Use blue circle to indicate the places she can arrive within 35 minutes (including 35 minutes). Use red circle to indicate places that requires more than 35 minutes but less than 60. Please note that Alice would depart from Ho Sin-Hang Engineering Building(22.4179252, 114.2027235) at 12:35 pm on a Friday.

➡ Add necessary scripts to `maps_api.py`. For the restaurants in Shatin (based on category from Openrice), plot restaurants that requires 0 ~ 35 minutes to arrive via public transportations in **blue** circles, and plot restaurants that requires 35 ~ 60 minutes to arrive in **red** circles.

Hints: Use online converter to translate human-readable time to Epoch time in seconds in order to define the query. For example October 20, 2017 11:01:08 AM is 1508497268 seconds in Epoch format. You can also use the filter function from Task 3.

📌 Put down the plot in your `report.pdf` and briefly explain what you found. The plot should have a center at Fo Tan Station(22.395607,114.1963153)

Submission rules

1. You are allowed to form a group of two to do this project. Students in the same group would get the same marks. Marks will be deducted if you do not follow the submission rules. Anyone/Anygroup who is caught plagiarizing would get 0 score!
2. The answers in your `report.pdf` should be consistent with the execution results of your scripts. Please check your scripts before the submission.
3. For each group, please pack all your script files as a single archive named as

`<student-id1>_<student-id2>_prj.zip`

For example, `1155012345_1155054321_prj.zip` (replace `<student-id1>` and `<student-id2>` with your own student IDs.) If you are doing the assignment alone, just leave `<student-id2>` empty, e.g, `1155012345_prj.zip`. The structure of the zip file should be the same as `project.zip`:

```
*.zip
├── report.pdf
├── openrice_urls_spider.py
├── openrice_spider.py
├── food_search_engine.py
├── gmplot.py
├── maps_api.py
├── price_popularity_relationship.py
├── reviews_distribution.py
├── top10_reviewed.py
└── type_distribution.py
```

Please do not submit any other files as they would result in a large zip file that could not be sent as a Gmail attachment.

4. Send the zip file to `cuhkcsci2040@gmail.com`,
 - Subject of your Email should be `<student-id1>_<student-id2>_prj` if you are in a two-person group or `<student-id1>_prj` if not.
 - No later than 23:59 on Thu. Nov. 30, 2017
5. **Q&A Board:** If you have any questions, please post on Piazza before directly sending email to TAs. The frequently asked questions are collected via the link <http://bit.ly/csci2040-17>.