



University of St.Gallen

Skills: Programming Introduction Level

Yelp Maps

Group Members:

Goncalo Marques	18-625-210
Chia Yu Yang	18-626-085
Xiaoqian Zhu	18-624-056

Contents

1	Background	1
2	Introudction.....	1
2.1	Explain the Voronoi Diagram:	1
3	K-Means Clustering Alogorithm.....	3
4	Prediction - Simple Least Squares Regression	4
5	Execution Code for Visulization – Explaining Each Arguments	7
6	Examples.....	8

1 Background

Our report is based on an online project called “Yelp maps” available in Berkley online platform, therefore we used data sets from their website that can be accessed to via [CS61a](#).

We download the dataset [maps.zip](#) from this website, including all the starter codes we need and we coded on three files: [utils.py](#) ; [abstractions.py](#) ; [recommend.py](#). The first two files are more related to the helper functions and preparation works, while the last one is the main file for execution and visualization.

In this project, we create a visualization of restaurant clustering and rates prediction round the city of Berkley, depending on certain criteria such as the given user and the type of restaurant we are searching. We choose this project because we found it’s very interesting and easily applicable to our daily lives, if we have the data of the restaurants in St.Gallen, we can do the same thing. Besides, not just in what respects to restaurants but also to facilities from other fields such as hospitals or libraries for example.

2 Introudction

2.1 Explain the Voronoi Diagram:

Our visualization we will be based on a [Voronoi diagram](#) which represents the division of a plane (in this case a map) into regions. And this partition will be based on the distance of points in a specific subset of the plane. These set of points will be specified beforehand and will be responsible for generating our parted map given that for each point (or seed) there will be a corresponding region containing all the points that are closer to that seed than to any other.

In our representation the seeds will be dots representing the location of the restaurants in the map and we use a Voronoi diagram to divide the map because we often find useful to find nearest restaurant within a given area at which we (as a user) might be at.

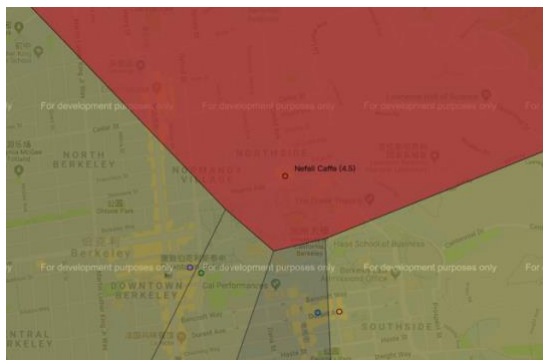


Figure 1 Illustration of Voronoi Diagram

As an illustrative example we can see that in this map the red region will stand for the set of points that will be closer to the restaurant “*Nefell Caffee*” (located at the red dot) than to any other restaurant present in the map.

Trough out the implementation of our code we will be given a finite set of points (restaurant locations) in the map and, for each point, the corresponding Voronoi cell (region/area) consists of all the locations closer to it than to any of the other points (restaurants). Berkeley will therefore be segmented into regions, where each region is shaded by the predicted rating of the closest restaurant, Yellow regions stand for a 5 stars restaurants, which means the highest rating, and blue regions represents 1 star restaurants.

To implement the colour band for the region of each restaurant depending on their rating the part of the code used was the following:

```
var clusterColor = d3.scale.category10();  
var fillColor = d3.scale.linear().domain([1, 5]).range(["blue", "yellow"]);  
var fillOpacity = 0.3;  
var hoverOpacity = 0.7;
```

This code is written in JavaScript and was already given on the file [visualize/voronoi.js](#)

To visualize this difference in ratings graphically we can consider the following map, were due to the user’s preferences the northside restaurants present low ratings (and consequently have their regions shaded by darker colours – approaching blue) whereas the southside restaurants present higher ratings (having their regions shaded by brighter colours – approaching yellow).

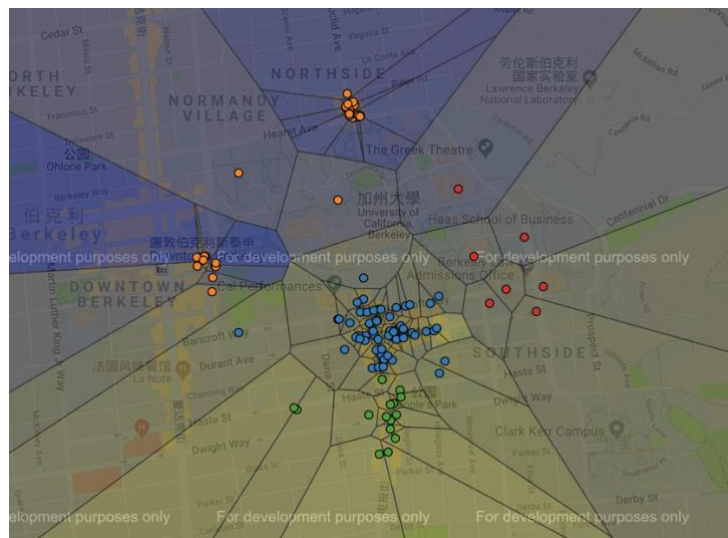


Figure 2 Illustration of color band

3 K-Means Clustering Alogorithm

Restaurants are tend to appear in clusters, i.e. in the southside of the city, in the downtown area, etc. We implemented **k-means algorithm** to group the data points(restaurants) by determining their center positions(which are called centroids).

Firstly, the k-means algorithm begins by choosing k centroids at random.

```
# Select initial centroids randomly by choosing k different restaurants
centroids = [restaurant_location(r) for r in sample(restaurants, k)]
```

Based on the randomly chose k centroids, group the restaurants into clusters based on their location. In each cluster contains all restaurants that are closest to the same centroid. We used the Euclidean distance. The following code is a part of the process of clustering.

```
def find_closest(location, centroids):
    """ Return the centroid in centroids that is closest to location.
    If multiple centroids are equally close, return the first one."""
    ### first find the minimum value
    min_value = min([distance(c, location) for c in centroids])
    for c in centroids:
        if distance(c, location) == min_value:
            return c
```

After the first time of clustering, we're going to compute a new centroid (average position) for each new cluster. During this process, the restaurants in each clusters will remain unchanged, but the centroid may different from the previous one.

```
def find_centroid(cluster):
    """Return the centroid of the locations of the restaurants in cluster."""
    loc = [restaurant_location(restaurant) for restaurant in cluster]
    lat, lot = [x for x, y in loc], [y for x, y in loc]
    return [mean(lat), mean(lot)]
```

Then we will repeat the above steps to update the centroids until the optimal list of centroid is found or the centroid locations no longer change significantly in each time (based on the maximum updates threshold). In this case, we set the maximum update times to 100 as default.

```
def k_means(restaurants, k, max_updates=100):
    """Use k-means to group restaurants by location into k clusters."""
    assert len(restaurants) >= k, 'Not enough restaurants to cluster'
    old_centroids, n = [], 0
    # Select initial centroids randomly by choosing k different restaurants
    centroids = [restaurant_location(r) for r in sample(restaurants, k)]
```

```

while old_centroids != centroids and n < max_updates: # iteration
    old_centroids = centroids
    clusters = group_by_centroid(restaurants, old_centroids)
    centroids = [find_centroid(lists) for lists in clusters]
    n += 1
return centroids

```

With code `python3(or `py -3` in windows) recommend.py -k 4 -p` in *Terminal* for MacOS of *cmd* for Windows, we could get the following figure:

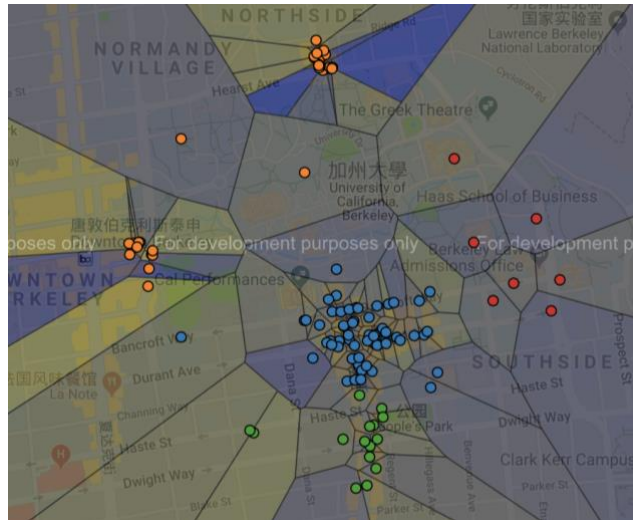


Figure 3 K-means clustering

Where we assigned k equals to 4, we can see that there are four different colors of dots in the figure, which means four different clusters. Based on the location, it seems that the **oranges** dots represent the restaurants in northwest area of the figure. The **green** ones means the southside of the city, and the **red** dots are located in the east. Those restaurants in **blue** are more likely to be centered around downtown area.

4 Prediction - Simple Least Squares Regression

Not only we did the clustering things, but also predicted an user's rating that he would give for a restaurant. Based on the user's past ratings, we try to predict the rating that he might give to a new restaurant by **simple least-squares linear regression**, which is aimed to minimized the value:

$$\sum_{r=1}^n (y_r - \hat{y}_r)^2 = \sum_{r=1}^n (y_r - (a + b * feature_fn(r)))^2$$

Then take the first order partial derivatives on a and b respectively, then equal to 0, we can get the value of a and b as follows:

$$S_{xx} = \sum_{r=1}^n (x_r - \bar{x})^2, S_{yy} = \sum_{r=1}^n (y_r - \bar{y})^2, S_{xy} = \sum_{r=1}^n (x_r - \bar{x})(y_r - \bar{y})$$

$$a = \bar{y} - b * \bar{x}, \quad b = \frac{S_{xy}}{S_{xx}} = \frac{\sum_{r=1}^n (x_r - \bar{x})(y_r - \bar{y})}{\sum_{r=1}^n (x_r - \bar{x})^2}$$

Where y_r is the rating of each restaurants, \hat{y}_r is the predicted rating based on the regression formula and the feature, \bar{y} is the mean of the all ratings, n is the number of observations, i.e. the number of restaurants.

The `predictor` function is in the formula like : $y = a + b * \text{feature_fn}(r)$, where y is the predicted rating of the typical restaurant r , `feature_fn` is the feature of restaurant of r , such as its mean rating or price. And the performance evaluation is based on R-squared value.

$$R - \text{squared} = \frac{\text{Explained sum of squares (ESS)}}{\text{Total sum of squares (TSS)}} = \frac{\sum_{r=1}^n (y_r - \hat{y}_r)^2}{\sum_{r=1}^n (y_r - \bar{y})^2}$$

After some simplification, we can get:

$$R - \text{squared} = \frac{S_{xy}^2}{S_{xx}S_{yy}} = \frac{\sum_{r=1}^n (x_r - \bar{x})(y_r - \bar{y})}{\sum_{r=1}^n (y_r - \bar{y})^2 * \sum_{r=1}^n (x_r - \bar{x})^2}$$

```
def find_predictor(user, restaurants, feature_fn):
    """Return a rating predictor (a function from restaurants to ratings), for a
    user by performing least-squares linear regression using feature_fn on the items in
    restaurants. Also, return the R^2 value of this model.

    Arguments:
    user -- A user
    restaurants -- A sequence of restaurants
    feature_fn -- A function that takes a restaurant and returns a number
    """

    reviews_by_user = {review_restaurant_name(review): review_rating(review)
                        for review in user_reviews(user).values()}

    xs = [feature_fn(r) for r in restaurants]
    ys = [reviews_by_user[restaurant_name(r)] for r in restaurants]

    S_xx, S_yy = sum([(i - mean(xs))**2 for i in xs]), sum([(j - mean(ys))**2 for
    j in ys])
    S_xy = sum([(i - mean(xs)) * (j - mean(ys)) for i, j in zip(xs, ys)])
    b = S_xy / S_xx
    a = mean(ys) - b * mean(xs)
    r_squared = S_xy ** 2 / (S_xx * S_yy)
    def predictor(restaurant):
```

```
    return a + b * feature_fn(restaurant)
```

```
    return predictor, r_squared
```

Since the restaurants have more than one features, we can find the optimal predictor feature(function) for a given user that has the highest R-squared value.

```
def best_predictor(user, restaurants, feature_fns):
```

```
    """Find the feature within feature_fns that gives the highest R^2 value for
    predicting ratings by the user; return a predictor using that feature.
```

```
    Arguments:
```

```
    user -- A user
```

```
    restaurants -- A list of restaurants
```

```
    feature_fns -- A sequence of functions that each takes a restaurant
```

```
    """
```

```
    reviewed = user_reviewed_restaurants(user, restaurants)
```

```
    func = max(feature_fns, key=lambda x: find_predictor(user, reviewed, x)[1])
```

```
    return find_predictor(user, reviewed, func)[0]
```

After choosing the best predictor, we will use this optimal predictor function to get the predicted rates for all restaurants.

```
def rate_all(user, restaurants, feature_fns):
```

```
    """Return the predicted ratings of restaurants by user using the best
    predictor based on a function from feature_fns.
```

```
    Arguments:
```

```
    user -- A user
```

```
    restaurants -- A list of restaurants
```

```
    feature_fns -- A sequence of feature functions
```

```
    """
```

```
    predictor = best_predictor(user, ALL_RESTAURANTS, feature_fns)
```

```
    reviewed = user_reviewed_restaurants(user, restaurants)
```

```
    all_restaurant = {}
```

```
    for restaurant in restaurants:
```

```
        name = restaurant_name(restaurant)
```

```
        if restaurant in reviewed:
```

```
            all_restaurant[name] = user_rating(user, name)
```

```
            # assign the value to the keys of dictionary
```

```
        else:
```

```
            all_restaurant[name] = predictor(restaurant)
```

```
return all_restaurant
```

Besides, each restaurants can belong to several categories like ‘Sandwiches’, ‘Chinese’, ‘Fondue’ do the prediction of a typical users of a typical category. The following is the code:

```
def search(query, restaurants):  
    """Return each restaurant in restaurants that has query as a category.  
  
    Arguments:  
    query -- A string  
    restaurants -- A sequence of restaurants  
    """  
    return [r for r in restaurants if query in restaurant_categories(r)]
```

5 Execution Code for Visualization – Explaining Each Arguments

-u is used to select a type user from the user’s directory. We can choose among these documents: likes_everything.dat, likes_expensive.dat, likes_southside.dat, one_cluster.dat, etc. Omitting the **-u** argument will default to `test_user.dat`. By putting this data in, we will have the result of previous users’ restaurant rating. We can even create our own rating by copying one of the `.dat` files. The following is the code that coded as the main subject in the `recommend.py` file.

```
parser.add_argument('-u', '--user', type=str, choices=USER_FILES,  
                    default='test_user',  
                    metavar='USER',  
                    help='user file, e.g.\n' +  
                    '{{{}}}'.format(', '.join(sample(USER_FILES, 3))))
```

-k stands for the number of clusters. (**-k 2**) means the restaurants will be divided in to 2 clusters base on the restaurants’ locations. It is possible to get more fine-grained groupings by increasing the number of clusters. Dots (restaurants) that have the same color in the map belong to the same cluster of restaurants.

```
parser.add_argument('-k', '--k', type=int, help='for k-means')
```

We use the **K-Means Algorithm** method to discover the centroids of clusters to group together restaurants that are close to each other.

-p option can be used if we wish to predict what rating a user would give a restaurant, even if they haven't rated the restaurant before. The document “`rate_all`” will assign the restaurant the rating computed by the best predictor for the user. In other words, by analyzing a user's past ratings, we can then try to predict what rating the user might give to a new restaurant that hasn’t been rated by this

user before. If a restaurant was already rated by the user, “rate_all” will assign the restaurant the user's rating.

```
parser.add_argument('-p', '--predict', action='store_true',  
                    help='predict ratings for all restaurants')
```

-q stands for different types of restaurants. Using this parameter, we can enter one category that we particularly want to search for. In other words, -q option allows you to filter restaurants based on a different categories such as: Chinese ; Mexican ; Vegetarian etc.

```
parser.add_argument('-q', '--query', choices=CATEGORIES,  
                    metavar='QUERY',  
                    help='search for restaurants by category e.g.\n' +  
                    '{{{}}}'.format(','.join(sample(CATEGORIES, 3))))
```

6 Examples

With code:

```
python3(or `py -3` in windows) recommend.py -u likes_expensive -k 2  
-p -q Sandwiches
```

We use the past rates of one user called likes_expensive (the argument after -u) to predict (with the character -p) the rating of the restaurants belong to the ‘Sandwiches’ category (the argument after -q) that the user haven’t rated before, otherwise it will be the previous rating that the user gave .

And we also assign the value for k-means to 2 (the argument after -k), the ‘Sandwiches’ restaurants are divided into clusters based on their location, one is in blue dots and the other is in orange dots.

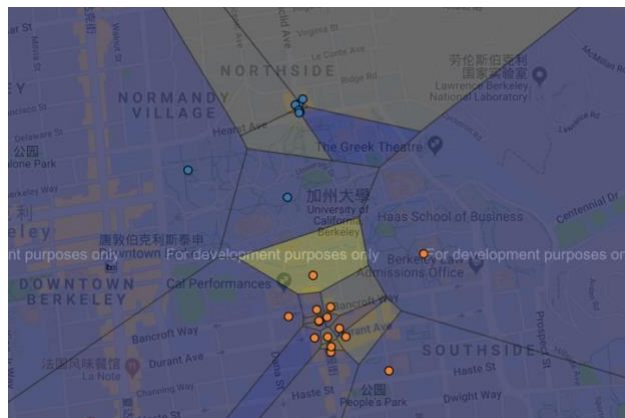


Figure 4 Outcome of example with prediction

With code:

```
python3(or `py -3` in windows) recommend.py -u likes_expensive -k 2  
-q Sandwiches
```

We got the following figures, the only difference is we delete the characters ‘-p’, then we will get the all the past rates of ‘Sandwiches’ restaurants of this specific users.



Figure 5 Example without predictions

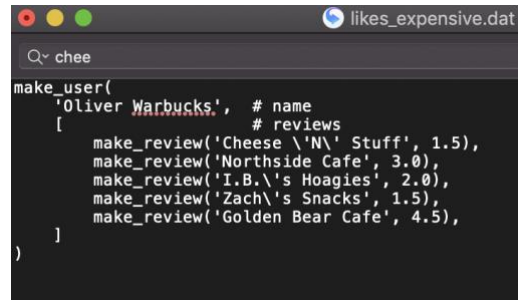


Figure 6 Past rates of ‘likes_expensive’ user

With code:

```
python3(or `py -3` in windows) recommend.py -u likes_everything -k  
3 -p
```

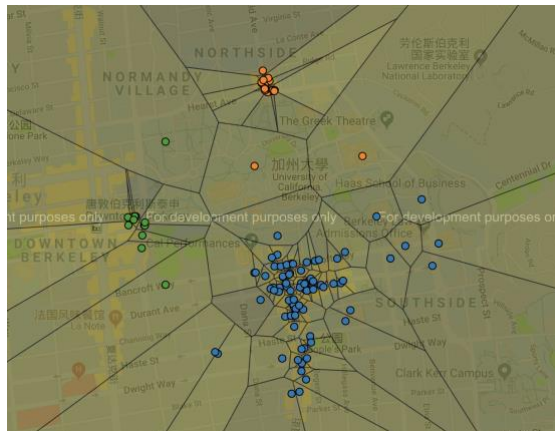


Figure 7 Outcome of K-means example

We change the users to ‘likes_everything’, and assign the k in k-means algorithm equals to 3, then we get three clusters in **orange**, **green** and **blue** dots respectively. Since we didn’t input the query argument, i.e. the specific restaurants categories, it just get the prediction of all restaurants.

with code:

```
python3(or `py -3` in windows) recommend.py -u likes_everything -k  
5 -p
```

In this case, we just change the value of k from 3 to 5, then we get five clusters in **red**, **orange**, **blue**, **green**, **purple** dots respectively.

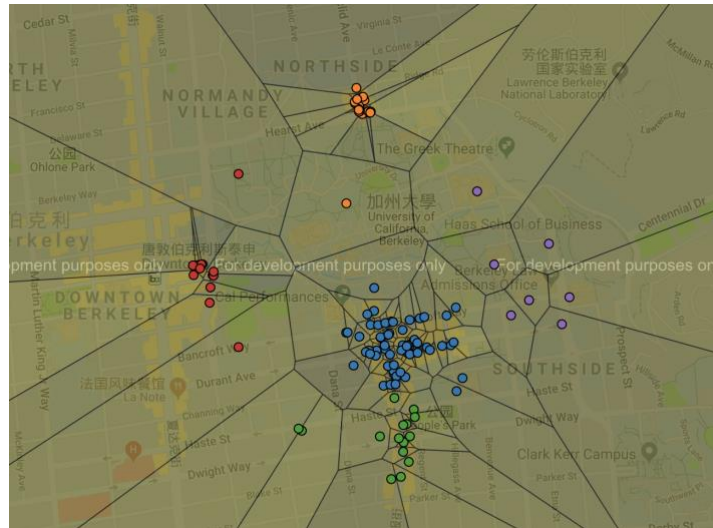


Figure 8 Outcome of K-Means example

All the code can be found at [Github](#).