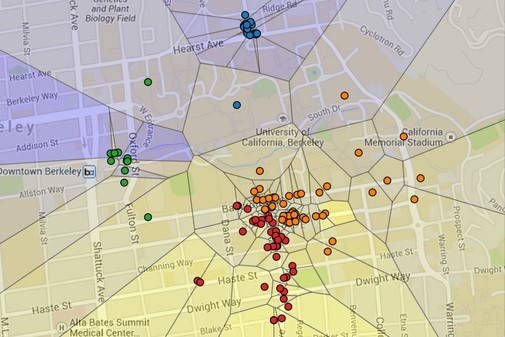
Project 1: Maps



*Let's go out to eat!*

[maps.zi](https://c88c.org/fa22/proj/maps/maps.zip)

[p](https://c88c.org/fa22/proj/maps/maps.zip)

[(maps.zip](https://c88c.org/fa22/proj/maps/maps.zip)

[)](https://c88c.org/fa22/proj/maps/maps.zip)

*Show me places I would like By learning my tastes.*

# Introduction

[In this project, you will create a visualization of restaurant scores using machine learning and the Yelp academic dataset (https://www.yelp.com/dataset). In this visualization, Berkeley is segmented into](https://www.yelp.com/dataset) regions, where each region is shaded by the predicted score of the closest restaurant (yellow is 5 stars, [blue is 1 star). Specifically, the visualization you will be constructing is a Voronoi diagram (https://en.wikipedia.org/wiki/Voronoi\_diagram).](https://en.wikipedia.org/wiki/Voronoi_diagram)

In the map above, each dot represents a restaurant. The color of the dot is determined by the restaurant's location. For example, downtown restaurants are colored green. The user that generated this map has a strong preference for Southside restaurants, and so the southern regions are colored yellow.

[This project uses concepts from Sections 2.1 (http://composingprograms.com/pages/21](http://composingprograms.com/pages/21-introduction.html)[introduction.html), 2.2 (http://composingprograms.com/pages/22-data-abstraction.html), 2.3](http://composingprograms.com/pages/23-sequences.html)

[(http://composingprograms.com/pages/23-sequences.html), and 2.4.3](http://composingprograms.com/pages/24-mutable-data.html#dictionaries)

[(http://composingprograms.com/pages/24-mutable-data.html#dictionaries) of Composing Programs (http://composingprograms.com/). It also introduces techniques and concepts from *machine learning*](http://composingprograms.com/), a growing field at the intersection of computer science and statistics that analyzes data to find patterns and make predictions.

The [maps.zip (maps.zip)](https://c88c.org/fa22/proj/maps/maps.zip) archive contains all the starter code and data sets. The project uses several files, but all of your changes will be made to utils.py , abstractions.py , and recommend.py .

abstractions.py : Data abstractions used in the project recommend.py : Machine learning algorithms and data processing

utils.py : Utility functions for data processing ucb.py : Utility functions for miscellaneous and debugging data : A directory of Yelp users, restaurants, and reviews ok : The autograder

proj1.ok : The ok configuration file tests : A directory of tests used by ok users : A directory of user files

visualize : A directory of tools for drawing the final visualization

You can obtain all the files needed for this project by downloading this [zip archive (maps.zip).](https://c88c.org/fa22/proj/maps/maps.zip)

# Logistics

This is a 2 week project. You may work with one other partner. You should not share your code with students who are not your partner or copy from anyone else's solutions.

In the end, you will submit one project for both partners. The project is worth 40 points.

You will turn in the following files:

utils.py abstractions.py recommend.py

You do not need to modify or turn in any other files to complete the project. To submit the project, run the following command:

python3 ok --submit

You will be able to view your submissions on the [Ok dashboard (https://okpy.org/)](https://okpy.org/).

For the functions that we ask you to complete, there may be some initial code that we provide. If you would rather not use that code, feel free to delete it and start from scratch. You may also add new function definitions as you see fit.

However, please do **not** modify any other functions. Doing so may result in your code failing our autograder tests. Also, please do not change any function signatures (names, argument order, or number of arguments).

In the end, you will submit one project for both partners. The project is worth 40 points. 36 points are assigned based on correctness for your final submission, and 4 points are for the mid-project checkpoint.

Make sure you have added your partner on OK before submitting the project.

**The assignment is due on Wednesday, 10/12, at 11:59pm. We will have a mid-project checkpoint due Monday, 10/3 at 11:59pm. To learn about extra credit, keep reading!**

Part 1: Phases 0 and 1 are due on Monday, 10/3, at 11:59pm. To submit, run python3 ok -submit . **You cannot use slip days on the mid project checkpoint.**

The entire project (including Part 1) is due on Wednesday, 10/12, at 11:59pm. To submit, run python3 ok --submit . We will only grade your last submission before the deadline. If you submit

the entire project 2 days early (by Monday 10/10, at 11:59 pm), you will receive **2 points of EC**.

# Testing

Throughout this project, you should be testing the correctness of your code. It is good practice to test often, so that it is easy to isolate any problems.

We have provided an **autograder** called ok to help you with testing your code and tracking your progress.

The primary purpose of ok is to test your implementations, but there is a catch. At first, the test cases are *locked*. To unlock tests, run the following command from your terminal.

python3 ok -u

Run the specific version of this command as detailed in each problem. This command will start an interactive prompt that looks like:

=====================================================================

Assignment: Maps

OK, version ...

=====================================================================

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Unlocking tests

At each "? ", type what you would expect the output to be.

Type exit() to quit

---------------------------------------------------------------------

Question 0 > Suite 1 > Case 1

(

cases remaining:

1)

>>>

Code here

?

At the ? , you can type what you expect the output to be. If you are correct, then this test case will be available the next time you run the autograder.

The idea is to understand *conceptually* what your program should do first, before you start writing any code.

Once you have unlocked some tests and written some code, you can check the correctness of your program using the tests that you have unlocked:

python3 ok

Most of the time, you will want to focus on a particular question. Use the -q option as directed in the problems below.

The tests folder is used to store autograder tests, so make sure **not to modify it**. You may lose all your unlocking progress if you do. If you need to get a fresh copy, you can download the [zip archive (maps.zip)](https://c88c.org/fa22/proj/maps/maps.zip) and copy it over, but you will need to start unlocking from scratch.

If you are ever stuck on a problem, you should add print statements to your code to help debug. Use print("DEBUG:", variable) to see the value of variable and still have your tests pass.

# Phase 0: Utilities

All changes in this phase will be made to

utils.py

.

## Problem 0 (4 pt)

Before starting the core project, familiarize yourself with some Python features by completing utils.py . Each function described below can be implemented in one line. The functions you implement here can optionally be used in future phases of the project but are not strictly needed. As you work through this phase, you can unlock the test cases for these exercises and check your solutions by running ok :

python3 ok -q 00 -u

python3 ok -q 00

### Note on Lambdas

Lambdas are anonymous function definitions. They allow us to define a function and use it without explicitly giving it a name. They are useful because they allow us to define functions for one-time use without clogging up our namespace. *(Coming up with new names is hard, you know?)* A lambda takes in a number of arguments. It evaluates a single expression using those arguments, and returns the value that the expression evaluates to. Lambdas have the following syntax:

lambda

<

**arguments**

>

:

<

**expression**

using

arguments

>

Below are a couple of examples of lambda expressions:

>>>

square =

**lambda**

x: x \* x

>>>

square(

4

)

16

>>>

plus =

**lambda**

x,y: x + y

>>>

plus(

2

,

3

)

5

>>>

(

**lambda**

x: x[

0

])([

1

,

2

,

3

])

1

*Note that as with other functions, a lambda function's lexical parent is the frame in which it was defined.* Note on Dictionaries

You've actually seen several abstract data types! List, tuples, ranges, and even strings are examples of abstract data types. Dictionary is another example of abstract data types.

Dictionaries are unordered sets of key-value pairs. To create a dictionary, use the following syntax:

>>>

singers = {

'Iggy Azalea'

:

'Fancy'

,

'Beyonce'

:

'Flawless'

,

'Adam Levine'

:

'Maps'

}

The curly braces denote the key-value pairs in your dictionary. Each key-value pair is separated by a comma. For each pair, the key appears to the left of the colon and the value appears to the right of the colon. (This is a dictionary's constructor!) You can retrieve values from your dictionary by "indexing" using the key:

>>>

singers[

'Beyonce'

]

'Flawless'

>>>

singers[

'Iggy Azalea'

]

'Fancy'

You can update an entry for an existing key in the dictionary using the following syntax. What this means is that each key is unique. Be careful, adding a new key follows identical syntax!

>>>

singers[

'Beyonce'

]

=

'Survivor'

>>>

singers[

'Beyonce'

]

'Survivor'

>>>

singers[

'Nicki Minaj'

]

=

'Anaconda'

# new entry!

>>>

singers[

'Nicki Minaj'

]

'Anaconda'

You can also check for membership of keys!

>>>

'Adam Levine'

**in**

singers

**True**

Recall how we can iterate through a list using for-loops. For example, you can do something like this:

>>>

a = [

1

,

2

,

3

]

>>>

**for**

each

**in**

a:

...

print(each)

1

2

3

What happens if you iterate through a dictionary? Can you even iterate through a dictionary?? Notice what happens:

>>>

shopping\_cart = {

"apple"

:

3

,

"bananas"

:

4

,

"orange"

:

6

}

>>>

**for**

each

**in**

shopping\_cart:

...

print(each)

apple

bananas

orange

Notice that when you iterate through a dictionary, the set of keys is what you iterate through. How would you print out values instead? You can simply do:

>>>

shopping\_cart = {

"apple"

:

3

,

"bananas"

:

4

,

"orange"

:

6

}

>>>

**for**

each

**in**

shopping\_cart:

...

print(shopping\_cart[each])

3

4

5

### Problem 0.1: Using list comprehensions

Recall that a list comprehension constructs a new list from an existing sequence by first filtering the given sequence, and then computing an element of the result for each remaining element that is not filtered out. A list comprehension has the following syntax:

[

<

**map**

expression

>

for

<

**name**

>

in

<

**sequence**

expression

>

if

<

**filter**

expression

>

]

For example, if we wanted to square every even integer in range(10) , we could write:

>>>

[

x \* x

**for**

x

**in**

range(

10

)

**if**

x %

2

==

0

]

[

0

,

4

,

16

,

36

,

64

]

In utils.py , implement map\_and\_filter . This function takes in a sequence s , a one-argument function mapper , and a one-argument function filterer . It returns a new list containing the result of calling mapper on each element of s for which filterer returns a true value. *Make sure your solution is only one line and uses a list comprehension.*

### Problem 0.2: Using min

The built-in min function takes a sequence (such as a list or a dictionary) and returns the sequence's smallest element. The min function can also take a keyword argument called key , which must be a oneargument function. The key function is called with each element of the list, and the return values are used for comparison. For example:

>>>

min([-

1

,

0

,

1

])

# no key argument; smallest input

-

1

>>>

min([-

1

,

0

,

1

]

, key

=

**lambda**

x: x\*x)

# input with the smallest square

0

In utils.py , implement key\_of\_min\_value , which takes in a dictionary d and returns the key that corresponds to the minimum value in d . This behavior differs from just calling min on a dictionary, which would return the smallest key. *Make sure your solution is only one line and uses the min function.*

### Problem 0.3: Using zip

The zip function defined in utils.py takes multiple sequences as arguments and returns a list of lists, where the i-th list contains the i-th element of each original list. For example:

>>>

zip([

1

,

2

,

3

]

,

[

4

,

5

,

6

])

[[

1

,

4

]

,

[

2

,

5

]

,

[

3

,

6

]]

>>>

**for**

triple

**in**

zip([

'a'

,

'b'

,

'c'

]

,

[

1

,

2

,

3

]

,

[

'do'

,

're'

,

'mi'

]):

...

print(triple)

[

'a'

,

1

,

'do'

]

[

'b'

,

2

,

're'

]

[

'c'

,

3

,

'mi'

]

In utils.py , use the zip function to implement enumerate , which takes a sequence s and a starting index start . It returns a list of pairs, in which the i-th element is i+start paired with the i-th element of

s . *Make sure your solution is only one line and uses the zip function and a range* .

*Note*: zip and enumerate are also built-in Python functions, but their behavior is slightly different than the versions provided in utils.py .

# Phase 1: Data Abstraction

All changes in this phase will be made to

abstraction.py

.

Complete the data abstractions in abstractions.py . Two of the data abstractions have already been completed for you: the *review* data abstraction and the *user* data abstraction. Make sure that you understand how they work.

## Data Abstraction Descriptions

### Review Data Abstraction

|  |  |  |
| --- | --- | --- |
| **variable** | **type** | **description** |
| restaurant\_name | string | restaurant name of the review |
| review\_score | float between 1 and 5 | number of stars given by the review |

### User Data Abstraction

|  |  |  |
| --- | --- | --- |
| **variable** | **type** | **description** |
| name | string | name of user |
| reviews | dictionary from restaurant names to review data abstractions | reviews that user has written |

### Restaurant Data Abstraction

|  |  |  |
| --- | --- | --- |
| **variable** | **type** | **description** |
| name | string | name of restaurant |
| location | list containing two floats: latitude and longitude | location of restaurant |
| categories | list of strings | categories that restaurant belongs to |
| price | integer | price of restaurant |
| scores | list of scores (floats between 1 to 5) | list of scores based on restaurant reviews |

## Problem 1 (2 pt)

Complete the implementations of the constructor and selectors for the *restaurant* data abstraction. You can use any implementation you choose, but the constructor and selectors must be defined together to satisfy the following description. A starter implementation using a dictionary is provided.

make\_restaurant : return a restaurant constructed from five arguments:

name (a string) location (a list containing latitude and longitude) categories (a list of strings)

price (a number)

reviews (a list of review data abstractions created by make\_review )

restaurant\_name : return the name of a restaurant restaurant\_location : return the location of a restaurant restaurant\_categories : return the categories of a restaurant restaurant\_price : return the price of a restaurant restaurant\_scores : return a list of scores (numbers) Use OK to unlock and test your code:

python3 ok -q 01 -u

python3 ok -q 01

## Problem 2 (2 pt)

Implement the restaurant\_num\_scores and restaurant\_mean\_score functions, *without assuming any particular implementation of a restaurant*.

Be sure not to violate abstraction barriers! Note that if these two functions were part of the restaurant ADT, there would be no abstraction barrier to break. For this problem, assume the two functions are not part of the ADT and they are just practice. Test your implementation before moving on:

python3 ok -q 02 -u

python3 ok -q 02

When you finish, you should be able to generate a visualization of all restaurants rated by a user. Use -u to select a user from the users directory. You can even create your own.

python3 recommend.py

python3 recommend.py -u one\_cluster

*Note*: You may have to refresh your browser to update the visualization. Submitting for the Mid-Project Checkpoint

When you have finished with Phase 0 and Phase 1, you can submit to the midpoint checkpoint:

python3 ok --submit

Because you have not completed the entire project, some tests are still locked. If you have fully completed Phase 0 and Phase 1, you will receive the message "7 test cases passed! No cases failed" but if you have completed anything beyond Phase 1, the number of passed cases may be higher.

# Phase 2: Unsupervised Learning

All changes in this phase will be made to

recommend.py

.

[Restaurants tend to appear in clusters (e.g. Southside restaurants, Gourmet Ghetto](http://www.gourmetghetto.org/)

[(http://www.gourmetghetto.org/)). In this phase, we will devise a way to group toget](http://www.gourmetghetto.org/)her restaurants that are close to each other.

The **k-means algorithm** is a method for discovering the centers of clusters. It is called an *unsupervised* learning method because the algorithm is not told what the correct clusters are; it must infer the clusters from the data alone.

The k-means algorithm finds k *centroids* within a dataset that each correspond to a cluster of inputs. To do so, k-means begins by choosing k centroids at random, then alternates between the following two steps:

1. Group the restaurants into clusters, where each cluster contains all restaurants that are closest to the same centroid.
2. Compute a new centroid (average position) for each new cluster.

This [visualization (http://tech.nitoyon.com/en/blog/2013/11/07/k-means/)](http://tech.nitoyon.com/en/blog/2013/11/07/k-means/) is a good way to understand [how the algorithm works, and this explanation (https://medium.com/dataseries/k-means-clusteringexplained-visually-in-5-minutes-b900cc69d175) might also help in understanding k-means.](https://medium.com/dataseries/k-means-clustering-explained-visually-in-5-minutes-b900cc69d175)

## Glossary

As you complete the remaining questions, you will encounter the following terminology. Be sure to refer back here if you're ever confused about what a question is asking.

**location**: A pair containing latitude and longitude **centroid**: A location (see above) that represents the center of a cluster **restaurant**: A restaurant data abstraction, as defined in abstractions.py **cluster**: A list of restaurants **user**: A user data abstraction, as defined in abstractions.py **review**: A review data abstraction, as defined in abstractions.py

**feature function**: A single-argument function that takes a restaurant and returns a number, such as its mean score or price

## Problem 3 (2 pt)

Implement find\_closest , which takes a location and a sequence of centroids (locations). It returns the element of centroids closest to location .

You should use the distance function from utils.py to measure distance between locations. The distance function calculates the [Euclidean distance (http://mathworld.wolfram.com/Distance.html)](http://mathworld.wolfram.com/Distance.html)

between two locations.

If two centroids are equally close, return the one that occurs first in the sequence of centroids .

*Hint*

:

Use the

min

function.

Use OK to unlock and test your code:

python3 ok -q 03 -u

python3 ok -q 03

## Problem 4 (4 pt)

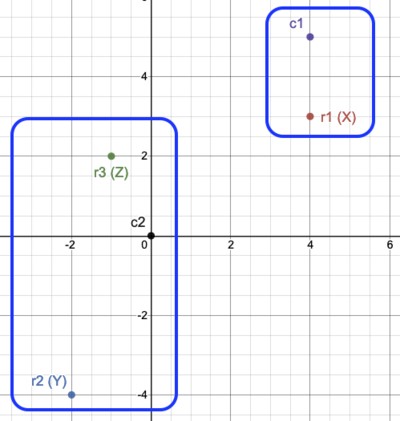
Implement group\_by\_centroid , which takes a sequence of restaurants and a sequence of

centroids (locations) and returns a list of clusters. Each cluster of the result is a list of restaurants that

are closer to a specific centroid in centroids than any other centroid. The order of the list of clusters returned does not matter.

If a restaurant is equally close to two centroids, it is associated with the centroid that appears first in the sequence of centroids .

The example below is a visualization of the doctest of group\_by\_centroid .Restaurant r1 is in a group by itself because it is closest to centroid c1 , while restaurants r2 and r3 are grouped together because they are closer to centroid c2 .



*Hint*

:

Use the provided

group\_by\_first

function to group together all values for the same key in a

list of

[

key, value

]

pairs. You can look at the doctests to see how to use it.

Be sure not violate abstraction barriers! Test your implementation before moving on:

python3 ok -q 04 -u

python3 ok -q 04

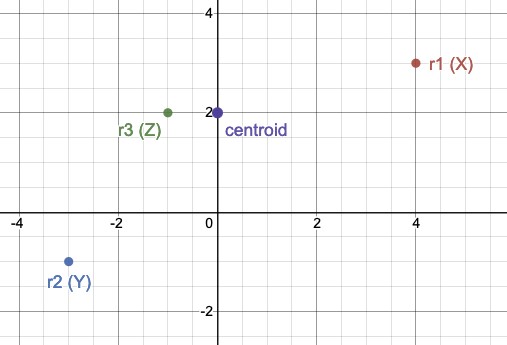
Common Mistakes

[Key Error (https://stackoverflow.com/questions/10116518/im-getting-key-error-in-python)](https://stackoverflow.com/questions/10116518/im-getting-key-error-in-python) - the key does not exist in the dictionary

## Problem 5 (4 pt)

Implement find\_centroid , which finds the centroid of a cluster (a list of restaurants) based on the locations of the restaurants. The centroid latitude is computed by averaging the latitudes of the restaurant locations. The centroid longitude is computed by averaging the longitudes.

The example below is a visualization of the doctest of find\_centroid . The centroid has been computed based on the locations of restaurants r1 , r2 , and r3 .



*Hint*

:

Use the

mean

function from

utils.py

to compute the average value of a sequence of

numbers.

Be sure not violate abstraction barriers! Test your implementation before moving on:

python3 ok -q 05 -u

python3 ok -q 05

## Problem 6 (2 pt)

Complete the implementation of k\_means . In each iteration of the while statement,

1. Group restaurants into clusters, where each cluster contains all restaurants closest to the same centroid. (*Hint*: Use group\_by\_centroid )
2. Bind centroids to a new list of the centroids of all the clusters. (*Hint*: Use find\_centroid ) Use OK to unlock and test your code:

python3 ok -q 06 -u

python3 ok -q 06

Check out this short youtube [video (https://www.youtube.com/watch?v=yR7k19YBqiw)](https://www.youtube.com/watch?v=yR7k19YBqiw) if you'd like a visualization of the algorithm in action.

Your visualization can indicate which restaurants are close to each other (e.g. Southside restaurants, Northside restaurants). Dots that have the same color on your map belong to the same cluster of restaurants. You can get more fine-grained groupings by increasing the number of clusters with the -k option.

python3 recommend.py -k 2

python3 recommend.py -u likes\_everything -k 3

Congratulations! You've now implemented an unsupervised learning algorithm.

# Phase 3: Supervised Learning

All changes in this phase will be made to

recommend.py

.

In this phase, you will predict what score a user would give for a restaurant. You will implement a *supervised* learning algorithm that attempts to generalize from examples for which the correct score is known, which are all of the restaurants that the user has already scored. By analyzing a user's past scores, we can then try to predict what score the user might give to a new restaurant. When you complete this phase, your visualization will include all restaurants, not just the restaurants that were scored by a user.

To predict scores, you will implement **simple least-squares linear regression**, a widely used statistical method that approximates a relationship between some input feature (such as price) and an output value (the score) with a line. The algorithm takes a sequence of input-output pairs and computes the slope and intercept of the line that minimizes the mean of the squared difference between the line and the outputs.

## Problem 7 (6 pt)

Implement the find\_predictor function, which takes in a user , a sequence of restaurants , and a feature function called feature\_fn . find\_predictor returns two values: a predictor function and an r\_squared value.

Use least-squares linear regression to compute the predictor and r\_squared . This method, described below, computes the coefficients a and b for the predictor line y = a + bx . The r\_squared value measures how accurately this line describes the original data.

One method of computing these values is by calculating the sums of squares, S\_xx , S\_yy , and S\_xy :

*Sxx* = Σ*i* (*xi* - mean(*x*))2

*Syy* = Σ*i* (*yi* - mean(*y*))2

*Sxy* = Σ*i* (*xi* - mean(*x*)) (*yi* - mean(*y*))

After calculating the sums of squares, the regression coefficients ( a and b ) and r\_squared are defined as follows:

*b*

=

*S*

/

*S*

*a*

= mean(

*y*

)

-

*b*

\* mean(

*x*

)

*R*

=

*S*

/ (

*S*

*S*

)

*Hint*

:

The

mean

function can be helpful here.

Use OK to unlock and test your code:

*xy*

*xx*

2

*xy*

2

*xx*

*yy*

python3 ok -q 07 -u

python3 ok -q 07

## Problem 8 (4 pt)

Implement best\_predictor , which takes a user , a list of restaurants , and a sequence of feature\_fns . It uses each feature function to compute a predictor function, then returns the predictor

that has the highest r\_squared value. All predictors are learned from the subset of restaurants reviewed by the user (called reviewed in the starter implementation).

*Hint*

:

The

max

function can also take a

key

argument, just like

min

.

Use OK to unlock and test your code:

python3 ok -q 08 -u

python3 ok -q 08

## Problem 9 (4 pt)

Implement rate\_all , which takes a user and list of restaurants . It returns a dictionary where the keys are the names of each restaurant in restaurants . Its values are scores (numbers).

If a restaurant was already scored by the user, rate\_all will assign the restaurant the user's score.

Otherwise, rate\_all will assign the restaurant the score computed by the best predictor for the user. The best predictor is chosen using a sequence of feature\_fns .

*Hint*

:

You may find the

user\_score

function in

abstractions.py

useful.

Be sure not violate abstraction barriers! Test your implementation before moving on:

python3 ok -q 09 -u

python3 ok -q 09

Common mistakes

[TypeError: unhashable type: 'dict' (https://stackoverflow.com/questions/13264511/typeerrorunhashable-type-dict) - Check the type of your dictionary key.](https://stackoverflow.com/questions/13264511/typeerror-unhashable-type-dict)

## Problem 10 (2 pt)

To focus the visualization on a particular restaurant category, implement search . The search function takes a category query and a sequence of restaurants. It returns all restaurants that have query as a category.

*Hint*

:

you might find a list comprehension useful here.

Be sure not violate abstraction barriers! Test your implementation:

python3 ok -q 10 -u

python3 ok -q 10

Congratulations, you've completed the project! The -q option allows you to filter based on a category. For example, the following command visualizes all sandwich restaurants and their predicted scores for the user who likes\_expensive restaurants:

python3 recommend.py -u likes\_expensive -k 2 -p -q Sandwiches

# Submit

Make sure to submit this assignment by running:

python3 ok --submit

# Predicting your own scores

Once you're done, you can use your project to predict your own scores too! Here's how:

1. In the users directory, you'll see a couple of .dat files. Copy one of them and rename the new file to yourname.dat (for example, michael.dat ).
2. In the new file (e.g. michael.dat ), you'll see something like the following:

make\_user(

'Michael Box'

,

# name

[

# reviews

make\_review(

'Jasmine Thai'

,

4.0

)

,

...

]

Replace the second line with your name (as a string).

1. Replace the existing reviews with reviews of your own! You can get a list of Berkeley restaurants with the following command:

python3 recommend.py -r

Rate a couple of your favorite (or least favorite) restaurants.

1. Use recommend.py to predict scores for you:

python3 recommend.py -u michael -k 2 -p -q Sandwiches

(Replace michael with your name.) Play around with the number of clusters (the -k option) and try different queries (with the -q option)!

How accurate is your predictor?