Introduction

In this project, you will create a visualization of restaurant ratings using machine learning and the Yelp academic dataset. In this visualization, Berkeley is segmented into regions, where each region is shaded by the predicted rating of the closest restaurant (yellow is 5 stars, blue is 1 star). Specifically, the visualization you will be constructing is a 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, 2.2, 2.3, and 2.4.3 of Composing Programs. It also introduces techniques and concepts from *machine learning*, a growing field at the intersection of computer science and statistics that analyzes data to find patterns and make predictions.

Download starter files

The 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 CS 61A
- `data`: A directory of Yelp users, restaurants, and reviews
- `ok`: The autograder
- `proj2.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

Logistics

This is a 10-day project, due on Thursday, 02/22/18. This is a solo project, so you will complete this project without a partner. You should not share your code with any other students, or copy from anyone else's solutions.

Remember that you can earn an additional bonus point by submitting the project at least 24 hours before the deadline.

The project is worth 22 points. 20 points are assigned for correctness, and 2 points for the overall composition of your program.

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.

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).

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. However, you should not be testing *too* often, to allow yourself time to think through problems.

We have provided an **autograder** called `ok` to help you with testing your code and tracking your progress. The first time you run the autograder, you will be asked to **log in with your Ok account using your web browser**. Please do so. Each time you run `ok`, it will back up your work and progress on our servers.

The primary purpose of `ok` is to test your implementations, but there are two things you should be aware of.

First, some of the test cases are *locked*. To unlock tests, run the following command from your terminal:

```
python3 ok -u
```

This command will start an interactive prompt that looks like:

```
=====================================================================
==================
Assignment: Yelp 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.

We recommend that you submit **after you finish each problem**. Only your last submission will be graded. It is also useful for us to have more backups of your code in case you run into a submission issue.

The `tests` folder is used to store autograder tests, so **do not 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 and copy it over, but you will need to start unlocking from scratch.

If you do not want us to record a backup of your work or information about your progress, use the `--local`option when invoking `ok`. With this option, no information will be sent to our course servers.

Phase 0: Utilities

All changes in this phase will be made to `utils.py`.

Problem 0 (2 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.

Problem 0.1: Using list comprehensions

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 `map_fn`, and a one-argument function `filter_fn`. It returns a new list containing the result of calling `map_fn` on each element of `s` for which `filter_fn` 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 one-argument 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`. The behavior of the built-in variants will be described later in the course.

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
```

Problem 1 (1 pt)

Implement the `mean` function which takes in a sequence of numbers, `s`, and returns the arithmetic mean, or average, of that sequence. The sequence cannot be empty: add an `assert` statement to ensure that empty sequences are not allowed.

Use Ok to unlock and test your code:

```
python3 ok -q 01 -u
python3 ok -q 01
```

Phase 1: Data Abstraction

All changes in this phase will be made to `abstractions.py`.

Problem 2 (2 pt)

Complete the implementations of the constructor and selectors for the *restaurant* data abstraction 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.

You can use any implementation you choose, but the constructor and selectors must be defined together such that the restaurant selectors return the correct field from the constructed restaurant.

- `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_ratings: return a list of ratings (numbers)

Use Ok to unlock and test your code:

```
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 by copying one of the .dat files!

```
python3 recommend.py
python3 recommend.py -u one_cluster
```

Omitting the -u argument will default to user test_user.

You may have to refresh your browser to update the visualization.

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). In this phase, we will devise a way to group together 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 is a good way to understand how the algorithm works.

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. Note that this is not a data abstraction, so we can assume its implementation is a two-element sequence.
- **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 grouped around a centroid
- **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 rating or price

Problem 3 (1 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 between two locations. It has been imported for you.

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 (2 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`.

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
```

Problem 5 (2 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.

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)

Finally, implement the full `k_means` algorithm. We've already filled out the first step of the algorithm. To complete the implementation, do the following in each iteration of the `while` statement:

1. Group `restaurants` into clusters, where each cluster contains all restaurants closest to the same centroid.
2. Bind `centroids` to a new list of the centroids of all the clusters.

Hint: Use the `group_by_centroid` and `find_centroid` helper functions.

The algorithm converges when an update doesn't change the list of centroids *or* after `max_updates` iterations.

Use Ok to unlock and test your code:

```
python3 ok -q 06 -u
python3 ok -q 06
```

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 rating a user would give for a restaurant. You will implement a *supervised* learning algorithm that attempts to generalize from examples for which the correct rating is known, which are all of the restaurants that the user has already rated. By analyzing a user's past ratings, we can then try to predict what rating 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 rated by a user.

To predict ratings, 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 rating) 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 (3 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`:

- $S_{xx} = \Sigma_i (x_i - \text{mean}(x))^2$
- $S_{yy} = \Sigma_i (y_i - \text{mean}(y))^2$
- $S_{xy} = \Sigma_i (x_i - \text{mean}(x)) (y_i - \text{mean}(y))$

After calculating the sums of squares, the regression coefficients (`a` and `b`) and `r_squared` are defined as follows:

- $b = S_{xy} / S_{xx}$
- $a = \text{mean}(y) - b * \text{mean}(x)$
- $R^2 = S_{xy}^2 / (S_{xx} S_{yy})$

Hint: The `mean` and `zip` functions can be helpful here.
Use Ok to unlock and test your code:

```
python3 ok -q 07 -u
python3 ok -q 07
```

Problem 8 (2 pt)

Implement `best_predictor`, which takes a `user`, a list of `restaurants`, and a sequence of `feature_fns`. It computes a predictor function for each feature function and 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 (2 pt)

Implement `rate_all`, which takes a `user`, a list of `restaurants`, and a sequence of `feature_fns`. It returns a dictionary where the keys are the names of each restaurant in `restaurants`. Its values are ratings (numbers).

If a restaurant was already rated by the user, `rate_all` will assign the restaurant the user's rating. Otherwise, `rate_all` will assign the restaurant the rating 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_rating` 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
```

In your visualization, you can now predict what rating a user would give a restaurant, even if they haven't rated the restaurant before. To do this, add the `-p` option:

Note (2/20/18): The `-p` flag does not currently work with the visualizer. Using this option, either by itself or in conjuction with other options, will produce a blank map; this *does not* mean your code is incorrect.

```
python3 recommend.py -u likes_southside -k 5 -p
```

If you hover over each dot (a restaurant) in the visualization, you'll see a rating in parentheses next to the restaurant name.

Problem 10 (1 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 ratings for the user who`likes_expensive` restaurants:

```
python3 recommend.py -u likes_expensive -k 2 -p -q
Sandwiches
```

Predicting your own ratings

Once you're done, you can use your project to predict your own ratings 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, `john.dat`).
2. In the new file (e.g. `john.dat`), you'll see something like the following:

```
   make_user(
3.     'John DoeNero',      # name
4.     [                    # reviews
5.         make_review('Jasmine Thai', 4.0),
6.         ...
7.     ]
```

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

8. 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.

9. Use `recommend.py` to predict ratings for you:

```
python3 recommend.py -u john -k 2 -p -q
Sandwiches
```

(Replace `john` 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?