Project #3 Description

For the Project and Application portion of Module 3, you will implement and assess two methods for clustering data. For Project 3, you will also implement two methods for computing closest pairs and two methods for clustering data. In Application 3, you will then compare these two clustering methods in terms of efficiency, automation, and quality. We suggest that you review this handout with the pseudo-code for the methods that you will implement before your proceed.

Provided data

As part of your analysis in Application 3, you will apply your clustering methods to several sets of 2D data that include information about lifetime cancer risk from air toxics. The raw version of this data is available from this website. Each entry in the data set corresponds to a county in the USA (identified by a unique 5 digit string called a FIPS county code) and includes information on the total population of the county and its per-capita lifetime cancer risk due to air toxics.

To aid you in visualizing this data, we have processed this county-level data to include the (x, y) position of each county when overlaid on this map of the USA. You can interactively explore this data set via this CodeSkulptor program. The program draws this map of the USA and overlays a collection of circles whose radius and color represent the total population and lifetime cancer risk of the corresponding county.

The input field on the left side of the CodeSkulptor frame allows you to choose a threshold for the lifetime cancer risk (multiplied by 10^{-5}) and eliminate those counties whose cancer risk is below that threshold. The raw data set includes 3108 counties. Taking the thresholds 3.5, 4.5, and 5.5 yields smaller data sets with 896, 290, and 111 counties, respectively. These four data sets will be our primary test data for your clustering methods and are available for download here in CSV form: (3108 counties, 896 counties, 290 counties, 111 counties).

The `Cluster` class

In the class notes and Homework 3, you considered the problem of clustering sets of points. In this Project, you will cluster high-risk counties into sets as part of your analysis of the cancer data set. To model this process, we have provided a `Cluster` class that allows you to create and merge clusters of counties. The implementation of this class is available here.

The class initializer
`Cluster(FIPS_codes, horiz_center, vert_center, total_population, average_risk)`
takes as input a set of county codes, the horizontal/vertical position of the cluster's center as well as the total population and averaged cancer risks for the cluster. The class definition supports methods for extracting these attributes of the cluster. The `Cluster` class also implements two methods that we will use extensively in implementing the Project.

- `distance(other_cluster)` - Computes the Euclidean distance between the centers of the clusters `self` and `other_cluster`.
- `merge_clusters(other_cluster)` - Mutates the cluster `self` to contain the union of the counties in `self` and `other_cluster`. The method updates the center of the mutated cluster using a weighted average of the centers of the two clusters based on their respective total populations. An updated cancer risk for the merged cluster is computed in a similar manner.

Mathematically, a clustering is a set of sets of points. In Python, modeling a clustering as a set of sets is impossible since the elements of a set must be immutable. This restriction guides us to model a set of clusters as a list of `Cluster` objects. This choice makes implementing an "indexed" set as described in the pseudo-code easy.

Closest pair functions

For this part of the Project, your task is to implement the two algorithms for computing closest pairs discussed in Homework 3. For the fast method, you will also implement its helper function separately to make debugging/testing your code easier. Your implementations should work on lists of `Cluster` objects and compute distances between clusters using the `distance` method. The three required functions are:

- `slow_closest_pair(cluster_list)` - Takes a list of `Cluster` objects and returns a closest pair where the pair is represented by the tuple `(dist, idx1, idx2)` with `idx1` < `idx2` where `dist` is the distance between the closest pair `cluster_list[idx1]` and `cluster_list[idx2]`. This function should implement the brute-force closest pair method described in **SlowClosestPair** from Homework 3.
- `fast_closest_pair(cluster_list)` - Takes a list of `Cluster` objects and returns a closest pair where the pair is represented by the tuple `(dist, idx1, idx2)` with `idx1` < `idx2` where `dist` is the distance between the closest pair `cluster_list[idx1]` and `cluster_list[idx2]`. This function should implement the divide-and-conquer closest pair method described **FastClosestPair** from Homework 3.
- `closest_pair_strip(cluster_list, horiz_center, half_width)` - Takes a list of `Cluster` objects and two floats `horiz_center` and `half_width`. `horiz_center` specifies the horizontal position of the center line for a vertical strip. `half_width` specifies the maximal distance of any point in the strip from the center line. This function should implement the helper function described in **ClosestPairStrip** from Homework 3 and return a tuple corresponding to the closest pair of clusters that lie in the specified strip. (Again the return pair of indices should be in ascending order.)

As one important coding note, you may need to sort a list of clusters by the vertical (or horizontal) positions of the cluster centers. A sort by vertical position can be done in a single line of Python using the `sort` method for lists by providing a `key` argument of the form:

```
cluster_list.sort(key = lambda cluster: cluster.vert_center())
```

Testing the closest pair functions

Before proceeding, we suggest that you thoroughly test your closest pair functions. Although we have provided detailed pseudo-code for everything that you will implement (including a program template), testing and debugging your closest pair functions will be challenging. Simple errors in your implementations of `fast_closest_pair` or `closest_pair_strip` can lead to a situation where your code produces erroneous answers on only very specific configurations of clusters.

OwlTest is designed to test these functions very thoroughly using both small test cases as well as large test cases created from the cancer data set. You are therefore strongly advised to run OwlTest now, before moving on to clustering, and ignore any errors for functions you have not written yet. The small test cases should catch the majority of your errors. For these errors, we suggest walking through your code by hand on the erroneous test case to find the source of the error. If your code happens to fail only a large test case, we suggest that you add calls to `slow_closest_pair` inside `fast_closest_pair` to check the correctness of the answers returned by the various recursive calls inside `fast_closest_pair`. When a large test fails, you will need to be persistent in tracking down its source even though this may be painful. Otherwise, your clustering code will rely on a faulty method for computing closest pairs and produce incorrect results when doing the Application.

Clustering functions

For the second part of the Project, your task is to implement hierarchical clustering and k-means clustering. In particular, you should implement the following two functions:

- `hierarchical_clustering(cluster_list, num_clusters)` - Takes a list of `Cluster` objects and applies hierarchical clustering as described in the pseudo-code **HierarchicalClustering** from Homework 3 to this list of clusters. This clustering process should proceed until `num_clusters` clusters remain. The function then returns this list of clusters.

Note that your implementation of lines 5-6 in the pseudo-code need not match the pseudo-code verbatim. In particular, merging one cluster into the other using `merge_clusters` and then removing the other cluster is fine. Note that, for this function, mutating `cluster_list` is allowed to improve performance.

- `kmeans_clustering(cluster_list, num_clusters, num_iterations)` - Takes a list of `Cluster` objects and applies k-means clustering as described in the pseudo-code **KMeansClustering** from Homework 3 to this list of clusters. This function should compute an initial list of clusters (line 2 in the pseudo-code) with the property that each cluster consists of a single county chosen from the set of the `num_cluster` counties with the largest populations. The function should then compute `num_iterations` of k-means clustering and return this resulting list of clusters.

As you implement **KMeansClustering**, here are a several items to keep in mind. In line 4, you should represent an empty cluster as a `Cluster` object whose set of counties is empty and whose total population is zero. The cluster centers μ_f, computed by lines 2 and 8-9, should stay fixed as lines 5-7 are executed during one iteration of the outer loop. To avoid modifying these values during execution of lines 5-7, you should consider storing these cluster centers in a separate data structure. Line 7 should be implemented using the `merge_clusters` method from the `Cluster` class. `merge_clusters` will automatically update the cluster centers to their correct locations based on the relative populations of the merged clusters.

Testing your clustering functions

To aid you in testing your clustering code, we have provided sample code that creates and visualizes a clustering. We suggest that you modify this code to generate visualizations of the clusterings produced by the two clustering methods. The module alg_project3_viz loads one of the cancer data sets, computes a clustering of a specified size using the original alphabetical ordering of the data, and then visualizes the resulting clustering using either `matplotlib` or `simplegui`. To switch between these visualization modes, set `DESKTOP = True` to use `matplotlib` or `DESKTOP = False` to use `simplegui`. (Note that you will need to download the supporting module alg_clusters_matplotlib into the same directory if you wish to run in desktop mode.)

To use this visualization code effectively, we **strongly suggest** that you develop your solution code for the Project in a separate file and then import this code into a working version of `alg_project3_viz`. You can then modify the body of `run_example` in `alg_project3_viz` to test your implementation of both clustering algorithms. This approach will also allow you to submit your solution code directly to OwlTest and avoid having the various imports in `alg_project3_viz` cause OwlTest to reject your submission. (If you are unfamiliar with how to handle imports in desktop Python and CodeSkulptor, please see this class page from "Principles of Computing".)

As a further aid in testing, we will provide a small test suite, (available in the next class reading) for testing your clustering methods on a 24 county data set. You should test your code with this test suite. You may add tests of your own creation to this suite that can be shared in class. Finally, Question 7 in the Application provides values for the distortion of several clusterings. Be sure and use these values as a final check to verify that your code works correctly.

Grading and coding standards

As you implement the closest pairs functions, test each function thoroughly. For the closest pairs functions, remember that you can use `slow_closest_pair` to test that the distance returned by `fast_closest_pair` is correct. For the two clustering methods, we suggest that you use our supplied test data and visualization code. Once you are confident that your implementation is correct, submit your code to this Owltest page, which will automatically test your project. Note that OwlTest will test your closest pairs functions on both artificially-generated data and actual cancer data. The clustering methods will be tested only on the cancer data. For both clustering methods, OwlTest will accept any ordering of the correct resulting clusters.

OwlTest uses Pylint to check that you have followed the coding style guidelines for this class. Deviations from these style guidelines will result in deductions from your final score. Please read the feedback from Pylint closely. If you have questions, feel free to consult this page and the class forums. When you are ready to submit your code to be graded formally, submit your code to the CourseraTest page for this project that is linked on the main programming assignment page. Remember that submitting to OwlTest does not record a grade for the assignment.

Marquer comme terminé