Project 3 - Closest pairs and clustering algorithms

Help

Overview

For the Project and Application portion of Module 3, we will implement and assess two methods for clustering data. For Project 3, you will 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 pairs

For this part of the Project, your task is to implement two algorithms for computing closest pairs as discussed in Homework 3. Your implementations should work on lists of `Cluster` objects and compute distances between clusters using the `distance` method. The two required functions are:

- `slow_closest_pairs(cluster_list)` - Takes a list of clusters and returns the **set** of all closest pairs where each pair is represented by the tuple `(dist, idx1, idx2)` with `idx1 < idx2`. This function should implement the brute-force closest pair method described in **BFClosestPair** from Homework 3 with the two differences: a set of all closest pairs is returned consisting of all pairs that share the same minimal distance and the returned indices are ordered.
- `fast_closest_pair(cluster_list)` - Returns a closest pair where the pair is represented by the tuple `(dist, idx1, idx2)` with `idx1 < idx2`. This function should implement the fast divide-and-conquer closest pair method described in Homework 3 with the exception that the returned indices are ordered. Note this method should compute horizontal and vertical orderings of the clusters and call a recursive helper function that will do the majority of the work.

To help you in this task, we have created a program template that provides you with a substantial amount of code. Most importantly, the template includes a partial implementation of `fast_closest_pair`. If you decide to use the template, your task is to implement the helper function `fast_helper(cluster_list, horiz_order, vert_order)` using the pseudo-code in **FastDCClosestPair**. Note that an efficient implementation of `fast_closest_pair` should not call `sort()` inside `fast_helper`.

To achieve maximal performance in `fast_closest_pair`, you should focus on ensuring that your implementations of lines 9-10, 14 in the pseudo-code are $O(n)$ where n is the number of clusters currently being processed. As a hint, you should remember that the lists H_l and H_r described in the pseudo-code can be temporarily converted to sets to improve the performance of the membership tests in your implementation of lines $9-10$ of the pseudo-code.

Clustering methods

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 clusters `cluster_list` 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_cluster` and then removing the other cluster is fine. Note that mutating `cluster_list` is allowed to improve performance.

- `kmeans_clustering(cluster_list, num_clusters, num_iterations)` - Takes a list of clusters `cluster_list` 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_cluster` method from the `Cluster` class. `merge_cluster` will automatically update the cluster centers to their correct locations based on the relative populations of the merged clusters.

Visualizing clusterings

To aid you in debugging your clustering code, we have created a module alg_project3_viz that loads one of the cancer data sets, computes a clustering of a specified size using the sequential 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".)

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_pairs` to test `fast_closest_pair`. 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.

IMPORTANT: Testing your code

Although we have provided detailed pseudo-code for everything that you will implement, testing and debugging the functions that you build in this Module, especially `fast_closest_pair`, will be challenging. Simple errors in your implementation of `fast_closest_pair` can lead to a situation where your code produces erroneous answers for only very specific configurations of clusters. Your approach should be to test everything that you build as thoroughly as possible, leveraging the fact that `slow_closest_pairs` and `fast_closest_pair` should return similar answers. When a test fails, you

need to be persistent in tracking down its source.

OwlTest is also designed to thoroughly test `fast_closest_pair`. However, scoring 100% on OwlTest does not guarantee that your version of `fast_closest_pair` is 100% correct. To help you further, we will provide a small test suite, that will be available in the forums, 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.

Created Fri 15 Aug 2014 2:03 AM CST
Last Modified Wed 24 Sep 2014 8:48 AM CST