Homework 4
CS540 Sp22

Assignment Goals

- Process fun real-world data
- Implement hierarchical clustering
- Randomly generate your own data
- Visualize the clustering process

You are to perform hierarchical clustering on publicly available Pokemon stats. Each Pokemon is defined by a row in the data set. Because there are various ways to characterize how strong a Pokemon is, we summarize the stat sheet into a shorter feature vector. For this assignment, you must represent a Pokemon's quality by two numbers: x and y.

$$x = \text{Attack} + \text{Sp. Atk} + \text{Speed}$$

represents the Pokemon's total offensive strength. Similarly,

$$y = \text{Defense} + \text{Sp. Def} + \text{HP}$$

represents the Pokemon's total defensive strength. After each Pokemon is represented as a two-dimensional feature vector (x, y), you need to cluster the first n Pokemon with hierarchical agglomerate clustering (HAC). Your function should work similarly to `scipy.cluster.hierarchy.linkage()`

Program Overview

Download the data in CSV format: Pokemon.csv. Note, there is no starter code for this assignment. If you feel stuck, refer to the starter code from past assignments. You have to write a few python functions. Here is a high level description of each for reference:

1. `load_data(filepath)` — takes in a string with a path to a CSV file formatted as in the link above, and returns the data points in a single structure. Section 0.1

2. `calculate_x_y(row)` — takes in one row from the data loaded from the previous function, calculates the corresponding x, y values for that Pokemon as specified above, and returns them in a single structure. Section 0.2

3. `hac(features)` — performs single linkage hierarchical agglomerate clustering on the Pokemon with the (x, y) feature representation, and returns a data structure representing the clustering. Section 0.3

4. `random_x_y(n)` — takes in the number of samples we want to randomly generate, and returns these samples in a form that can be input into `hac()`. Section 0.4

5. `imshow_hac(Z)` — visualizes the hierarchical agglomerate clustering on the Pokemon with the (x, y) feature representation. Section 0.5

You may implement other helper functions as necessary, but these are the functions we are testing. In particular, your final python file is just a suite of functions, you should not have code that runs outside of the functions. To test your code, you may want a "main" method to put it all together. Make sure, you either delete any testing code running outside functions or wrap it in a if `__name__=="__main__":` as noted at the bottom of the page.

Program Details

0.1 `load_data(filepath)`

Summary. [20pts]

- **Input:** `string`, the path to a file to be read.
- **Output:** `list`, where each element is a `dict` representing one row of the file read. Certain elements in the dict must be cast as `int`.

Details.

1. **Read** in the file specified in the argument, `filepath`. Note, the `DictReader` from Python's csv module is useful but, depending on your python version, this might return `OrderedDict`s instead of normal `dicts`. Make sure you convert to `dict` as appropriate if you choose to use this function.

2. **Return** a `list` of `dictionaries`, where each row in the dataset is a dictionary with the column headers as keys and the row elements as values. The primary numerical attributes of Pokemon (6 columns: *attack, sp. attack, speed, defense, sp. defense, HP*) should be converted to integers in this function.

You may assume the file exists and is a properly formatted CSV.

0.2 `calculate_x_y(row)`

Summary. [10pts]

- **Input:** `dict` representing one Pokemon.
- **Output:** `list` with 2 integer elements. The first element being the x computed for the given Pokemon and second element being the y computed.

Details. This function takes as input the `dictionary` representing a single Pokemon, and computes the feature representation (x, y). Recall, the formulas for x and y:

$$x = \text{Attack} + \text{Sp. Atk} + \text{Speed}$$

$$y = \text{Defense} + \text{Sp. Def} + \text{HP}$$

You must return a list having first element being x and second element being y. Note, this function works for only a single Pokemon at a time, not all of them simultaneously that you loaded in `load_data`. Make sure you are outputting the exact data structures with appropriate types as specified or you risk a major reduction in points.

0.3 `hac(features)`

Summary. [50pts]

- **Input:** `list` of 2-element `lists`, where each 2-element list is an (x, y) feature representation as computed in Section 0.2. The total number of feature vectors, i.e. the length of the input list, is n.
- **Output:** `numpy array` of shape $(n-1) \times 4$. For any i, $Z[i, 0]$ and $Z[i, 1]$ represent the indices of the two clusters that were merged in the ith iteration of the clustering algorithm. Then, $Z[i, 2] = d(Z[i, 0], Z[i, 1])$ is the distance between the two clusters that were merged in the ith iteration. Lastly, $Z[i, 3]$ is the size of the new cluster formed by the merge, i.e. the total number of Pokemon in this cluster. Note, the original Pokemon are considered clusters indexed by $0, \ldots, n-1$, and the cluster constructed in the ith iteration ($i \geq 1$) of the algorithm has cluster index $(n-1)+i$. Also, there is a tie-breaking rule specified below that must be followed.

Details. For this function, we would like you to mimic the behavior of SciPy's HAC function, `linkage()`. You may not use this function in your implementation, but we strongly recommend using it to verify your results! This is how you can test your code.

Distance. Using single linkage, perform the hierarchical agglomerate clustering algorithm as detailed in lecture. Use a standard Euclidean distance function for calculating the distance between two points. Note, we recommend you implement your own distance function, this can be done with one line of code if done perfectly. Other distance functions might not work as expected so check it works on the CSL machines first! You are liable for any reductions in points you might get for using a package distance function.

Outline. Here is one possible path you could follow to implement `hac()`

1. Number each of your starting data points from 0 to m-1. These are their original cluster numbers.

2. Create an $(n-1) \times 4$ array or list. Iterate through this array/list row by row. For each row,

   (a) Determine which two clusters you should merge and put their numbers into the first and second elements of the row, $Z[i,0]$ and $Z[i,1]$. The first element listed, $Z[i,0]$ should be the smaller of the two cluster indexes.

   (b) The single-linkage distance between the two clusters goes into the third element of the row, $Z[i,2]$

   (c) The total number of Pokemon in the cluster goes into the fourth element, $Z[i,3]$

   If you merge a cluster containing more than one Pokemon, its index (for the first or second element of the row) is given by n + the row index in which the cluster was created.

3. Before returning the data structure, convert it into a NumPy array.

Note, it may be convenient to maintain a distance matrix throughout the process to avoid having to recalculate the distances between points or clusters. This is especially important if your code is slow for even small n, such as $n \leq 20$.

Tie Breaking. When choosing the next two clusters to merge, we pick the pair having the smallest euclidean distance. In the case that multiple pairs have the same distance, we need additional criteria to pick between them. We do this with a tie-breaking rule on indices as follows: Suppose $(i_1, j_1), \ldots, (i_h, j_h)$ are pairs of cluster indices with equal distance, i.e., $d(i_1, j_1) = \cdots = d(i_h, j_h)$, and assume that $i_t < j_t$ for all t (so each pair is sorted). We tie-break by picking the pair with the smallest first index, i. If there are multiple pairs having first index i, we need to further distinguish between them. Say these pairs are $(i, t_1), (i, t_2), \ldots$ and so on. To tie-break between these pairs, we pick the pair with the smallest second index, i.e., the smallest t value in these pairs. Be aware that this tie-breaking strategy may not produce identical results to `linkage()`.

0.4 `random_x_y(n)`

Summary. [10pts]

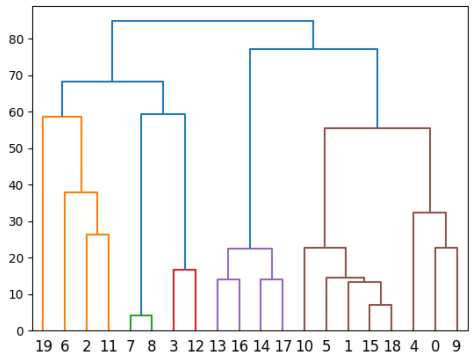
- **Input:** `integer` n.
- **Output:** `list` of length n where each element is a 2 element integer list. The first and second element of each 2 element list should be an integer drawn uniformly at random between 0 and 360. This output should be in the same format as the input to `hac()`.

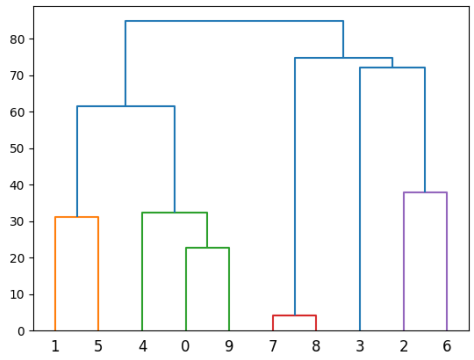
0.5 `imshow_hac(Z)`

Summary. [10pts]

- **Input:** `numpy array` Z output from `hac`.
- **Output:** None, simply plt.show() a graph that visualizes the hierarchical clustering. You should use `dendrogram` in the `scipy` module.

Here are some examples of successful visualizations for different sized lists of Pokemon:





Testing

To test your code, try running the following line in a main method or in a jupyter notebook:
`hac([calculate_x_y(row) for row in load_data('Pokemon.csv')][:n])`
for various choices of n. We do not test on very large values of n so looking at $n \leq 30$ should be sufficient. You can then compare your clustering to what `linkage()` would give you, and look at the different clustering visualizations. You can also try inputs generated by `random_x_y()`.

Submission Details

- Please submit your files in a zip file named **hw4_<netid>.zip**
- Inside your zip file, there should be only one file named: **hw4.py**
- All code should be contained in functions or under a `if __name__=="__main__":`
- Be sure to remove all debugging output before submission.