

# Homework 4

## CS540 Fall22

### Assignment Goals

- Process the real-world data
- Implement hierarchical clustering
- 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 stats into a shorter feature vector. For this assignment, you must represent a Pokemon's quality by 6 numbers:

HP, Attack, Defense, Sp. Atk, Sp. Def, Speed

After each Pokemon is represented as a 6-dimensional feature vector  $(x_1, \dots, x_6)$ , you need to cluster the first  $n$  Pokemon with hierarchical agglomerate clustering (HAC). Your function should work similarly to `scipy.cluster.hierarchy.linkage()` with `method='complete'`.

### Program Overview

The data in CSV format can be found in the file `Pokemon.csv`. Note, there is no starter code for this assignment. If you feel stuck, refer to the starter code from past assignments. You will have to write a few python functions for this assignment. 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, and returns the data points as a **list** of **dicts**. [Section 0.1](#)
2. `calc_features(row)` — takes in one row **dict** from the data loaded from the previous function then calculates the corresponding feature vector for that Pokemon as specified above, and returns it as a **numpy array** of shape  $(6,)$ . The dtype of this array should be `int64`. [Section 0.2](#)
3. `hac(features)` — performs complete linkage hierarchical agglomerate clustering on the Pokemon with the  $(x_1, \dots, x_6)$  feature representation, and returns a **numpy array** representing the clustering. [Section 0.3](#)

4. `imshow_hac(Z)` — visualizes the hierarchical agglomerate clustering on the Pokemon's feature representation. [Section 0.4](#)

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__":`. This is discussed more in [Section 0.5](#).

## Program Details

### 0.1 `load_data(filepath)`

Summary. [20pts]

- **Input:** `string`, the path to a file to be read.
- **Output:** `list`; each element of the `list` is a `dict` representing one row of the file read; the key of `dict` is a `string` (such as 'Attack') and the value of the `string` is a `string` (such as '111').

Details.

1. **Read** in the file specified in the argument, `filepath`. Note, the `DictReader` from Python's `csv` module (<https://docs.python.org/3/library/csv.html#csv.DictReader>) is useful but, depending on your python version, this might return `OrderedDicts` 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 keys are `strings`: 'HP', 'Attack', 'Defense', 'Sp. Atk', 'Sp. Def', 'Speed'.

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

### 0.2 `calc_features(row)`

Summary. [20pts]

- **Input:** `dict` representing one Pokemon.
- **Output:** `numpy array` of shape `(6,)` and dtype `int64`. The first element is  $x_1$  and so on with the sixth element being  $x_6$ .

**Details.** This function takes as input the **dict** representing one Pokemon, and computes the feature representation  $(x_1, \dots, x_6)$ . Specifically,

1.  $x_1 = \text{Attack}$
2.  $x_2 = \text{Sp. Attack}$
3.  $x_3 = \text{Speed}$
4.  $x_4 = \text{Defense}$
5.  $x_5 = \text{Sp. Def}$
6.  $x_6 = \text{HP}$

Note, these stats in the **dict** would be **string**. Make sure to convert each relevant stat to int when computing each  $x_i$ . Return a **numpy array** having each  $x_i$  in order:  $x_1, \dots, x_6$ . The shape of this array should be  $(6,)$ . The dtype of this array should be **int64**. Remember, this function works for only one Pokemon at a time, not all of the ones that you loaded in **load\_data** simultaneously. 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 **numpy arrays** of shape  $(6,)$ , where each array is an  $(x_1, \dots, x_6)$  feature representation as computed in [Section 0.2](#). The total number of feature vectors, i.e. the length of the input list, is  $n$ . Note, we test your code on different  $n$ 's as stated in [Section 0.5](#)).
- **Output:** **numpy array**  $Z$  of the 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  $i$ th iteration of the clustering algorithm. Then,  $Z[i, 2] = d(Z[i, 0], Z[i, 1])$  is the complete linkage distance between the two clusters that were merged in the  $i$ th iteration (this will be a real value, not integer like the other quantities). 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  $0, \dots, n-1$ , and the cluster constructed in the  $i$ th iteration ( $i \in \{0, 1, 2, \dots, n-2\}$ ) of the algorithm has cluster index  $(n-1) + (i+1)$ . 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 complete linkage, perform the hierarchical agglomerate clustering algorithm as detailed in lecture. Use the standard Euclidean distance function for calculating the distance between two points. You may implement your own distance function or use `numpy.linalg.norm()`. 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  $n - 1$ . 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  $Z[i, 1]$  and  $Z[i, 0] < Z[i, 1]$ .
  - (b) The complete-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,  $Z[i, 3]$

The index of a cluster with more than one Pokemon is given by:

$$(n - 1) + (i + 1) = n + \text{the row index in which the cluster was created.}$$

3. Before returning the data structure, convert it into a NumPy array if it isn't one already.

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 small  $n$ , such as  $n \leq 20$ .

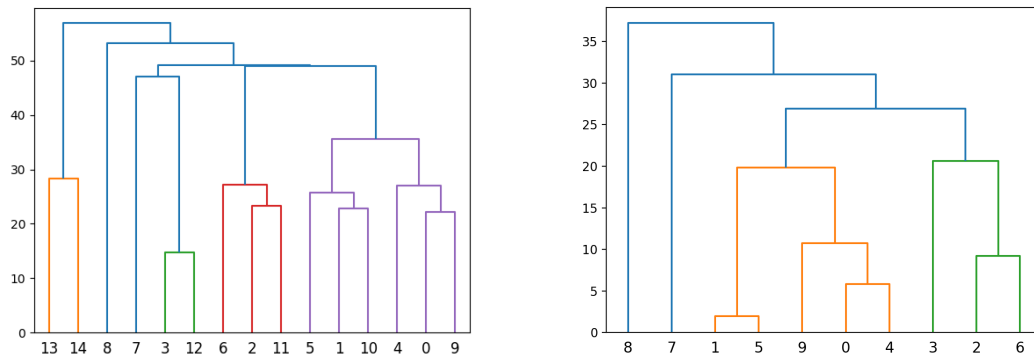
**Tie Breaking.** When choosing the next two clusters to merge, we pick the pair having the smallest complete-linkage distance. In the case that multiple pairs have the same distance, we need additional criteria to pick between them. We do this with a tie-breaking rule on indices as follows: Suppose  $(i_1, j_1), \dots, (i_h, j_h)$  are pairs of cluster indices with equal distance, i.e.,  $d(i_1, j_1) = \dots = d(i_h, j_h)$ , 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  $(i, t_1), (i, t_2), \dots$  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 `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:



## 0.5 Testing

To test your code, try running the following line in a main method or in a jupyter notebook:  
`hac([calc_features(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 (remember, set `method = 'complete'`), and look at the different clustering visualizations.

## 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`
- Be sure to remove all debugging output before submission.