Hierarchical Clustering

September 19, 2021

[1]:
```
import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt
```

[2]:
```
class BinaryTree:
    '''

    '''
    def __init__(self, root_data=None, left_child=None, right_child=None):
        '''

        '''
        self.root_data = root_data
        self.left_child = left_child
        self.right_child = right_child

    def get_right_child(self):
        '''

        '''
        return self.right_child

    def get_left_child(self):
        '''

        '''
        return self.left_child

    def set_root_data(self, data):
        '''

        '''
        self.root_data = data

    def get_root_data(self):
        '''
```

```
        '''
        return self.root_data

    def is_empty(self):
        '''

        '''
        return self.root_data == None

    def insert_left(self, new_data):
        '''

        '''
        if self.left_child == None:
            self.left_child = BinaryTree(new_data)
        else:
            t = BinaryTree(new_data)
            t.left_child = self.left_child
            self.left_child = t

    def insert_right(self, new_data):
        '''

        '''
        if self.right_child == None:
            self.right_child = BinaryTree(new_data)
        else:
            t = BinaryTree(new_data)
            t.right_child = self.right_child
            self.right_child = t

    def pre_order_traversal(self):
        '''

        '''
        if not self.is_empty():
            self.pre_order_helper(self)
            print()
        else:
            print("Empty tree")

    def pre_order_helper(self, tree):
        '''

        '''
        if tree is not None:
            print(tree.root_data, end=" ")
```

```
            self.pre_order_helper(tree.left_child)
            self.pre_order_helper(tree.right_child)

    def post_order_traversal(self):
        '''

        '''
        if not self.is_empty():
            self.post_order_helper(self)
            print()
        else:
            print("Empty tree")

    def post_order_helper(self, tree):
        '''

        '''
        if tree is not None:
            self.post_order_helper(tree.left_child)
            self.post_order_helper(tree.right_child)
            print(tree.root_data, end=" ")

    def level_order_traversal(self):
        '''

        '''
        if not self.is_empty():
            queue = [self.root_data]
            self.level_order_helper(self, queue)
            for data in queue:
                print(data, end=" ")
            print()
        else:
            print("Empty tree")

    def level_order_helper(self, tree, queue):
        '''

        '''
        if tree is not None:
            if tree.left_child is not None:
                queue.append(tree.left_child.root_data)
            if tree.right_child is not None:
                queue.append(tree.right_child.root_data)
            self.level_order_helper(tree.left_child, queue)
            self.level_order_helper(tree.right_child, queue)
```

```
    def in_order_traversal(self):
        '''

        '''
        if not self.is_empty():
            self.in_order_helper(self)
            print()
        else:
            print("Empty tree")

    def in_order_helper(self, tree):
        '''

        '''
        if tree is not None:
            self.in_order_helper(tree.left_child)
            print(tree.root_data, end=" ")
            self.in_order_helper(tree.right_child)
```

[3]:
```
class HierachicalCluster(BinaryTree):
    def __init__(self, name, root=None, left_child=None, right_child=None,␣
 ↪distance=0):
        self.name = name
        self.root = root
        self.left_child = left_child
        self.right_child = right_child
        self.distance = distance

    def __getitem__(self, index):
        return self.root[index]

    def __len__(self):
        return len(self.root)

    def compute_centroid(self):
        centroid = []
        if self.left_child is None and self.right_child is None:
            return self.root
        else:
            rc = self.right_child.get_leaf_count()
            lc = self.left_child.get_leaf_count()

            for i in range(len(self.right_child)):
                val = ((self.right_child[i]*rc + self.left_child[i]*lc) /␣
 ↪(rc+lc))
                centroid.append(val)
        self.root = centroid
```

```
        return self.root

    def get_leaf_count(self):
        if self.left_child == None:
            return 1
        else:
            return self.left_child.get_leaf_count() + self.right_child.
↪get_leaf_count()


    def set_right_child(self, r_child):
        self.right_child = r_child

    def set_left_child(self, l_child):
        self.left_child = l_child

    def is_leaf(self):
        return self.left_child == None and self.right_child == None

    def get_root(self):
        return self.root

    def get_distance(self):
        return self.distance

    def get_display_name(self):
        if self.is_leaf():
            return self.name
        else:
            return str(round(self.distance, 2))

    def print(self):
        thislevel = [self]
        while thislevel:
            nextlevel = []
            for n in thislevel:
                print(n.get_display_name()+' ', end='')
                if n.left_child:
                    nextlevel.append(n.left_child)
                if n.right_child:
                    nextlevel.append(n.right_child)
            print('')
            thislevel = nextlevel

    def get_all_leaves(self):
        leaves = []
        self.get_all_leaves_rc(self, leaves)
```

```python
        return leaves

    def get_all_leaves_rc(self, hc, leaves):
        if hc is None:
            return
        hc.get_all_leaves_rc(hc.left_child, leaves)
        hc.get_all_leaves_rc(hc.right_child, leaves)
        if hc.left_child is None and hc.right_child is None:
            leaves.append(hc)
```

[4]:
```python
class Pair:
    def __init__(self, HC_member1, HC_member2):
        self.HCmember1 = HC_member1
        self.HCmember2 = HC_member2
        self.distance = self.compute_distance()

    def __lt__(self, p2):
        return self.distance < p2.get_distance()

    def compute_distance(self):
        x = 0
        for i in range(len(self.HCmember1)):
            x += (self.HCmember1[i] - self.HCmember2[i])**2
        return math.sqrt(x)

    def get_distance(self):
        return self.distance

    def get_hc_member1(self):
        return self.HCmember1

    def get_hc_member2(self):
        return self.HCmember2
```

[5]:
```python
class BinaryHeap:

    def __init__(self):
        '''
        heap_list[0] = 0 is a dummy value (not used)
        '''
        self.heap_list = [0]
        self.size = 0

    def __str__(self):
        return str(self.heap_list)

    def __len__(self):
```

```
        return self.size

    def __contains__(self, item):
        return item in self.heap_list

    def is_empty(self):
        return self.size == 0

    def find_min(self):
        '''
        the smallest item is at the root node (index 1)
        '''
        if self.size > 0:
            min_val = self.heap_list[1]
            return min_val
        return None

    def insert(self, item):
        '''
        append the item to the end of the list (maintains complete tree property)
        violates the heap order property
        call percolate up to move the new item up to restore the heap order property
        '''
        self.heap_list.append(item)
        self.size += 1
        self.percolate_up(self.size)

    def del_min(self):
        '''
        min item in the tree is at the root
        replace the root with the last item in the list (maintains complete tree property)
        violates the heap order property
        call percolate down to move the new root down to restore the heap property
        '''
        min_val = self.heap_list[1]
        self.heap_list[1] = self.heap_list[self.size]
        self.size = self.size - 1
        self.heap_list.pop()
        self.percolate_down(1)
        return min_val

    def min_child(self, index):
        '''
```

```
        return the index of the smallest child
        if there is no right child, return the left child
        if there are two children, return the smallest of the two
        '''
        if index * 2 + 1 > self.size:
            return index * 2
        else:
            if self.heap_list[index * 2] < self.heap_list[index * 2 + 1]:
                return index * 2
            else:
                return index * 2 + 1

    def build_heap(self, alist):
        '''
        build a heap from a list of keys to establish complete tree property
        starting with the first non leaf node
        percolate each node down to establish heap order property
        '''
        index = len(alist) // 2 # any nodes past the half way point are leaves
        self.size = len(alist)
        self.heap_list = [0] + alist[:]
        while (index > 0):
            self.percolate_down(index)
            index -= 1

    def percolate_up(self, index):
        '''
        compare the item at index with its parent
        if the item is less than its parent, swap!
        continue comparing until we hit the top of tree
        (can stop once an item is swapped into a position where it is greater␣
↪than its parent)
        '''
        while index // 2 > 0:
            if self.heap_list[index] < self.heap_list[index // 2]:
                temp = self.heap_list[index // 2]
                self.heap_list[index // 2] = self.heap_list[index]
                self.heap_list[index] = temp
            index //= 2

    def percolate_down(self, index):
        '''
        compare the item at index with its smallest child
        if the item is greater than its smallest child, swap!
        continue continue while there are children to compare with
        (can stop once an item is swapped into a position where it is less than␣
↪both children)
```

```
        '''
        while (index * 2) <= self.size:
            mc = self.min_child(index)
            if self.heap_list[index] > self.heap_list[mc]:
                temp = self.heap_list[index]
                self.heap_list[index] = self.heap_list[mc]
                self.heap_list[mc] = temp
            index = mc
```

[6]:
```
def main():
    df = pd.read_csv(r'cancer.csv', header=None)
    df_size = len(df.values)
    df_list = []
    for i in range(df_size):
        df_sample = toList(df, i)
        df_list.append(df_sample)

    hc_list = []
    for samp in df_list:
        hc = HierachicalCluster(samp[0], samp[1:])
        hc_list.append(hc)

    while len(hc_list) > 1:
        bmh = BinaryHeap()
        pairs_list = all_pairs(hc_list)
        for pair in pairs_list:
            bmh.insert(pair)

        min = bmh.del_min()
        new_cluster = HierachicalCluster('branch', None, distance=min.
 ↪compute_distance())
        new_cluster.set_left_child(min.get_hc_member1())
        new_cluster.set_right_child(min.get_hc_member2())
        new_cluster.compute_centroid()
        hc_list.append(new_cluster)
        hc_list.remove(new_cluster.get_left_child())
        hc_list.remove(new_cluster.get_right_child())

def toList(df, ri):
    a = df.loc[ri]
    b = a.to_numpy()
    c = b.tolist()
    return c

def all_pairs(hc_list):
    pairs = []
    for i in range(len(hc_list)-1):
```

```
        for j in range(i+1,len(hc_list)):
            pair = Pair(hc_list[i],hc_list[j])
            pairs.append(pair)
    return pairs

main()
```

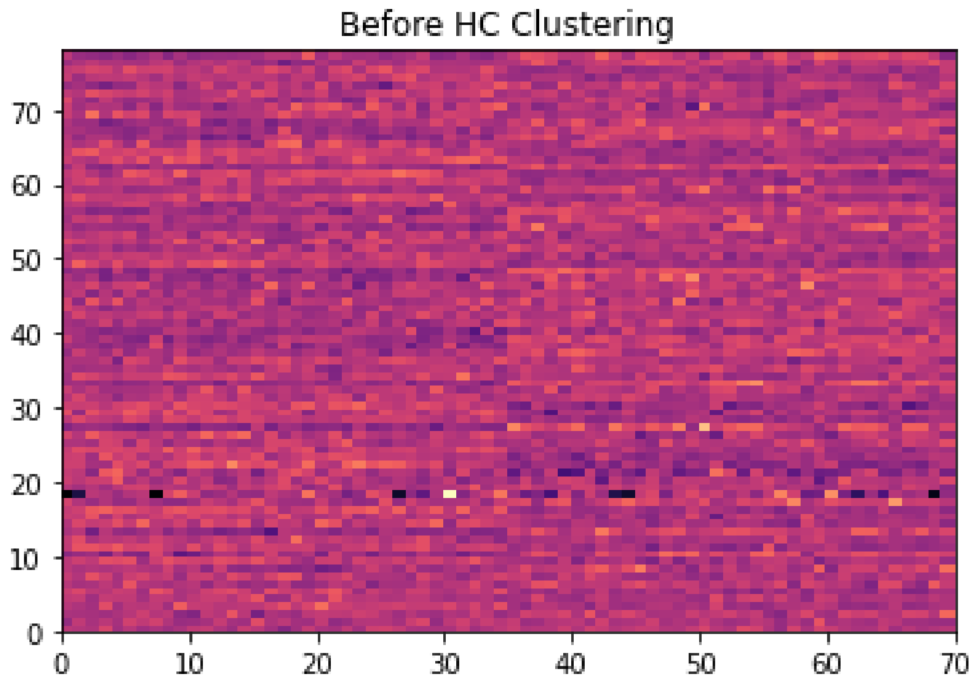
[7]:
```
df = pd.read_csv(r'cancer.csv', header=None)
df_size = len(df.values)
df_list = []
for i in range(df_size):
    df_sample = toList(df, i)
    df_list.append(df_sample)

hc_list = []
for samp in df_list:
    hc = HierachicalCluster(samp[0], samp[1:])
    hc_list.append(hc)

hc1 = []
for i in range(len(hc_list)):
    hc1.append(hc_list[i].get_root())
print('clustering in process...')
while len(hc_list) > 1:
    bmh = BinaryHeap()
    pairs_list = all_pairs(hc_list)
    for pair in pairs_list:
        bmh.insert(pair)

    min = bmh.del_min()
    new_cluster = HierachicalCluster('branch', None, distance=min.
 ↪compute_distance())
    new_cluster.set_left_child(min.get_hc_member1())
    new_cluster.set_right_child(min.get_hc_member2())
    new_cluster.compute_centroid()
    hc_list.append(new_cluster)
    hc_list.remove(new_cluster.get_left_child())
    hc_list.remove(new_cluster.get_right_child())

leaves = hc_list[0].get_all_leaves()
roots = []
for i in range(len(leaves)):
    roots.append(leaves[i].get_root())
df = pd.DataFrame(data=roots)
```

```
clustering in process…
```

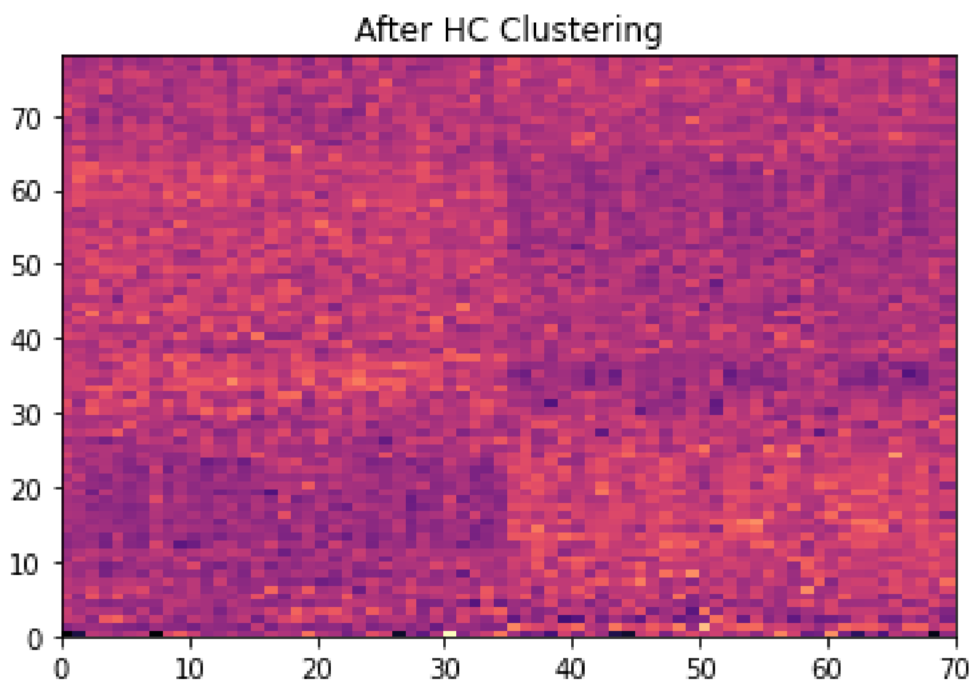
[8]:
```
df
```

[8]:
```
           0         1         2         3         4         5         6   \
0  -6.743823 -5.055445  0.131029 -0.179264 -0.905948 -0.099369  0.171932
1  -1.569939 -0.976442 -0.776502 -0.903095 -1.530858 -1.691495 -2.058896
2   0.504952  0.193136 -1.430769  1.992230  2.415003  0.555135 -0.886940
3   0.688423 -0.664907 -0.032537 -0.265434 -0.178374 -1.381730  0.331276
4  -0.362366 -0.532900 -1.130017 -1.638415 -1.325531 -0.975372 -1.210770
..       …         …         …         …         …         …         …
73  0.128002 -0.052396 -0.243591  1.119036 -0.638129  0.035529 -0.686474
74 -0.619225  0.549554  0.220727  0.883504 -0.611348 -0.092708  0.521462
75  0.001240 -0.572502 -0.021985 -0.713520  0.129617  0.808276  0.937815
76 -0.482456 -0.878758 -0.354395  0.785844 -1.035394  0.293666 -0.665914
77  0.051278 -0.884038 -0.539067  0.027545 -1.164839 -0.525712 -1.102827

           7         8         9   …        60        61        62        63  \
0  -6.667616  1.012630  2.135013  …  3.629780 -2.003763 -4.489750 -0.832883
1  -0.854075 -1.392909 -1.389785  …  0.239395  2.538446  2.015366  2.125202
2   0.542794 -0.639944 -1.572939  … -2.271288 -1.692297  0.177905  1.025060
3  -1.491776  0.223004 -0.723611  …  1.042043  0.238790  0.466701  0.493270
4  -1.218476  0.169423  0.228067  …  1.048464 -0.731945 -0.728190 -0.706582
..       …         …         …  …       …         …         …         …
73 -0.317596 -0.775309 -0.382444  …  0.554033 -0.041529 -0.406905 -0.297769
74  0.175020 -1.170122 -0.475816  …  0.740247  0.072675 -0.172259 -0.533751
75 -0.155640  0.169423 -0.168765  … -0.807258  0.705989  0.322303  0.423472
76 -0.469430 -0.823250 -0.059232  …  0.438452  0.742327  0.195955  0.616246
77 -0.631386  0.251205 -0.511728  …  0.085286  0.789047 -0.168649 -0.134908

           64        65        66        67        68        69
0  -3.425178  0.029338 -0.789247 -0.383363 -6.173325 -1.197897
1   1.456331 -0.688572  2.237339  2.230814 -0.081404  1.270326
2  -1.832721 -2.023436 -0.376697 -1.279755  0.227700 -2.085043
3   0.085169  0.040555  1.123818  0.838015 -1.037694 -1.725831
4  -0.635125  0.982812 -1.409898 -0.822631  1.312783  0.130099
..       …         …         …         …         …         …
73  0.497385 -0.677355  0.050457 -0.447646 -0.155460  0.059345
74  0.380229 -0.868050 -0.500827 -0.286939  1.422257 -0.229113
75  0.085169 -0.666138  0.868256  0.430889  0.002312 -1.401995
76  0.740376 -0.419356  0.295067  0.166614  0.710675 -0.065835
77  0.414942 -0.660529 -0.077324 -0.086947  1.522072 -0.468588

[78 rows x 70 columns]
```

[9]:
```
plt.pcolor(hc1,cmap='magma')
plt.title('Before HC Clustering')
plt.show()
```



[10]:
```
plt.pcolor(df,cmap='magma')
plt.title('After HC Clustering')
plt.show()
```



```
[ ]:
```