SOFE2715 Data Structure and Algorithms

Winter 2017

Group 1: Anna Safonov 100601514

George Zakharov 100588814

Justin Kaipada 100590167

Riley Stephens 100589185

Tutorial 7 Activity: Project Update

For this project, our group has chosen to implement Convex Hull and K-means algorithms. Both algorithms are implemented in Python.

Convex Hull

Where We Are

The main algorithm for convex hull is complete and tested. We are working on improving time complexity for the main method.

Algorithms and Data Structures Used

The following algorithms and data structures are used in our implementation of the Convex Hull:

- Arrays are used to store data and as a stack implementation;
- The stack is used to detect and remove concavities along the boundary of data points in the Graham scan method.
- An insertion sort is used to order data based on slopes data points make with the lowest point in the set;
- Graham scan is used as the main algorithm to find the convex hull of a data set.

The following are Python libraries used in our implementation of Convex Hull:

- csv – CSV file reader used to parse CSV data from the test exercises provided for this project
- math – for operations with square roots in the method that determines Euclidean distance between two given points
- matplotlib.pyplot – to plot the convex hull for visual presentation
- timeit – to calculate running time of the main algorithm for time complexity analysis

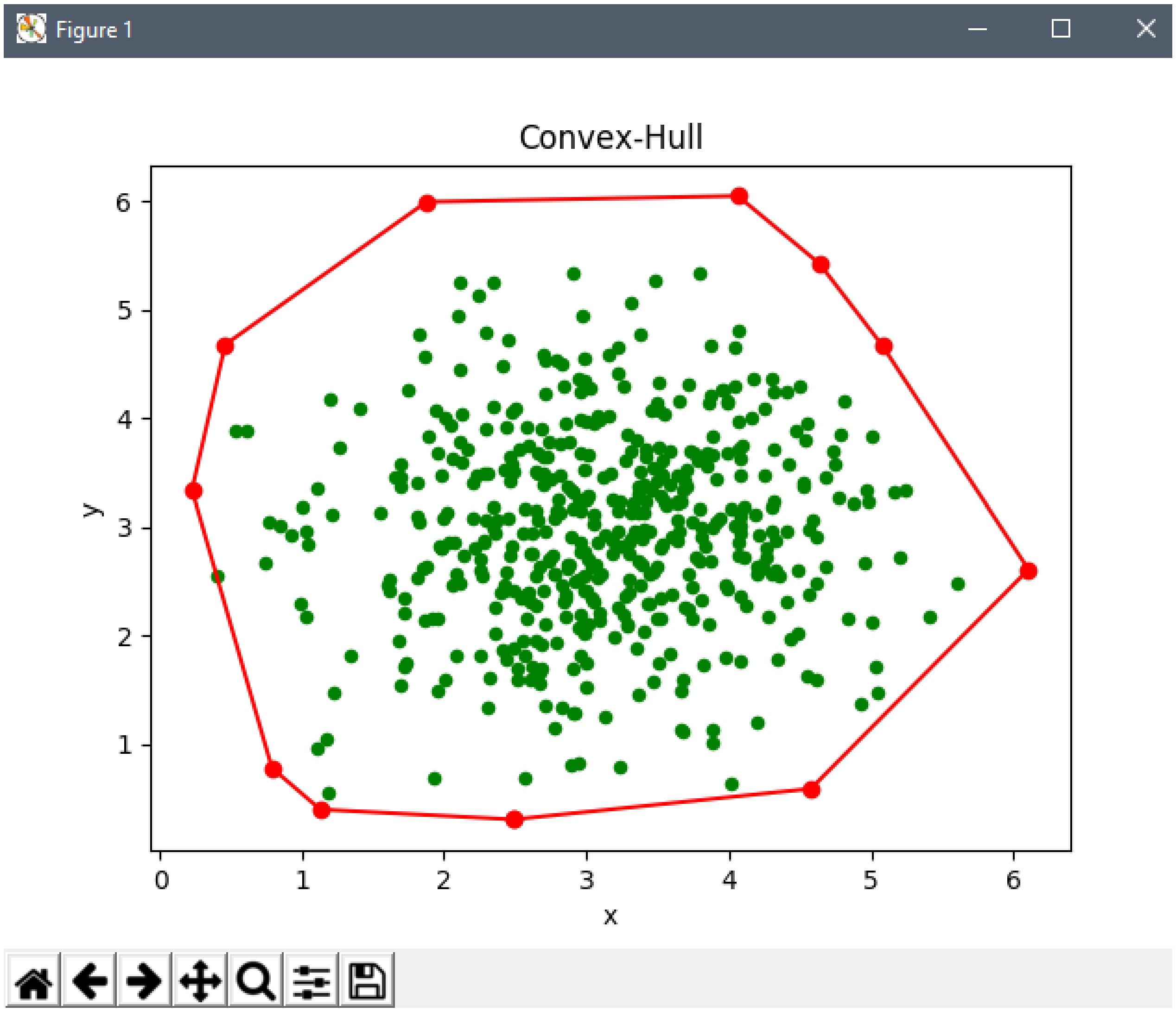
Challenges Faced and Future Directions

Currently, the algorithms takes progressively longer time for larger sets of data, and the bottleneck for time complexity is likely in the insertion sort implemented in the main method. An alternative approach is to consider heap sort, and implementing heap sort instead of insertion sort is our next step for this part of the project.

Subsequently, we will also work on calculating and analyzing the time complexity of the algorithm, developing performance matrix, and completing the final report for presentation and evaluation.

Results:

Running the Algorithm for Exercise 3 gives the following output:



Convex-Hull main method at the time of submission of this assignment:

```python
def scan(data):
    "This method uses grahamScan to create a convex hull from the given points"
    hull = []                # Used like a stack here
    hull.append([])          # For the X-coordinates
    hull.append([])          # For the Y-coordinates
    newPoints = []           # For storing the unused points
    newPoints.append([])     # For the X-coordinates
    newPoints.append([])     # For the Y-coordinates
    backTrackTracker = False
    hull[0].append(data[0].pop(0))
    hull[1].append(data[1].pop(0))
    '''First element of the data is always the starting point of the hull,
       which is the lowest y-coordinate we calculated
    '''
    M = 1  # This Stores the size of the hull stack
    while True:
        if backTrackTracker != True:
            hull[0].append(data[0].pop(0))  # Pop the first most item, add to hull
            hull[1].append(data[1].pop(0))  # Pop the first most item, add to hull
            M += 1
        if len(data[0]) == 0:  # This is true when all points are serviced
            return hull, newPoints
        if leftOrRight(hull[0][M-2], hull[1][M-2],
                       hull[0][M-1], hull[1][M-1],
                       data[0][0], data[1][0]) == 'left':
            backTrackTracker = False

        elif leftOrRight(hull[0][M-2], hull[1][M-2],
                         hull[0][M-1], hull[1][M-1],
                         data[0][0], data[1][0]) == 'right':
            # Backtracking
            newPoints[0].append(hull[0].pop())
            newPoints[1].append(hull[1].pop())
            backTrackTracker = True
            M -= 1
        else:  # The points are collinear
            continue
```

K-means

Where We Are

K-means algorithm is implemented and has been tested on the exercises provided. We are working on implementing better data structures to store data – i.e. trying a linked list instead of an array.

Algorithms and Data Structures Used

The following algorithms and data structures are used in our implementation of the K-means:

- Arrays are used to store data points and clusters;
- A list is used to store centroids;

The following are Python libraries used in our implementation of K-means:

- csv – CSV file reader used to parse CSV data from the test exercises provided for this project
- math – for operations with square roots in the method that determines Euclidean distance between two given points
- matplotlib.pyplot – to plot the K-means clusters for visual presentation
- timeit – to calculate running time of the main algorithm for time complexity analysis
- random – random.choice() function is used to pick random points from the data set to be the centroids at the start of clustering

Challenges Faced and Future Directions

When executing the **K-means.py** more than 10 times consecutively we get a zero division exception. This may be because of the random number generator that we use in the algorithm. We are currently working on this issue.

For the future, we would also like to implement some changes so that the number of clusters and the file to be worked with can be passed to the program from command line.

Subsequently, we will also work on calculating and analyzing the time complexity of the algorithm, developing performance matrix, and completing the final report for presentation and evaluation.

Results

Running the Algorithm for Exercise 3 gives the following output:

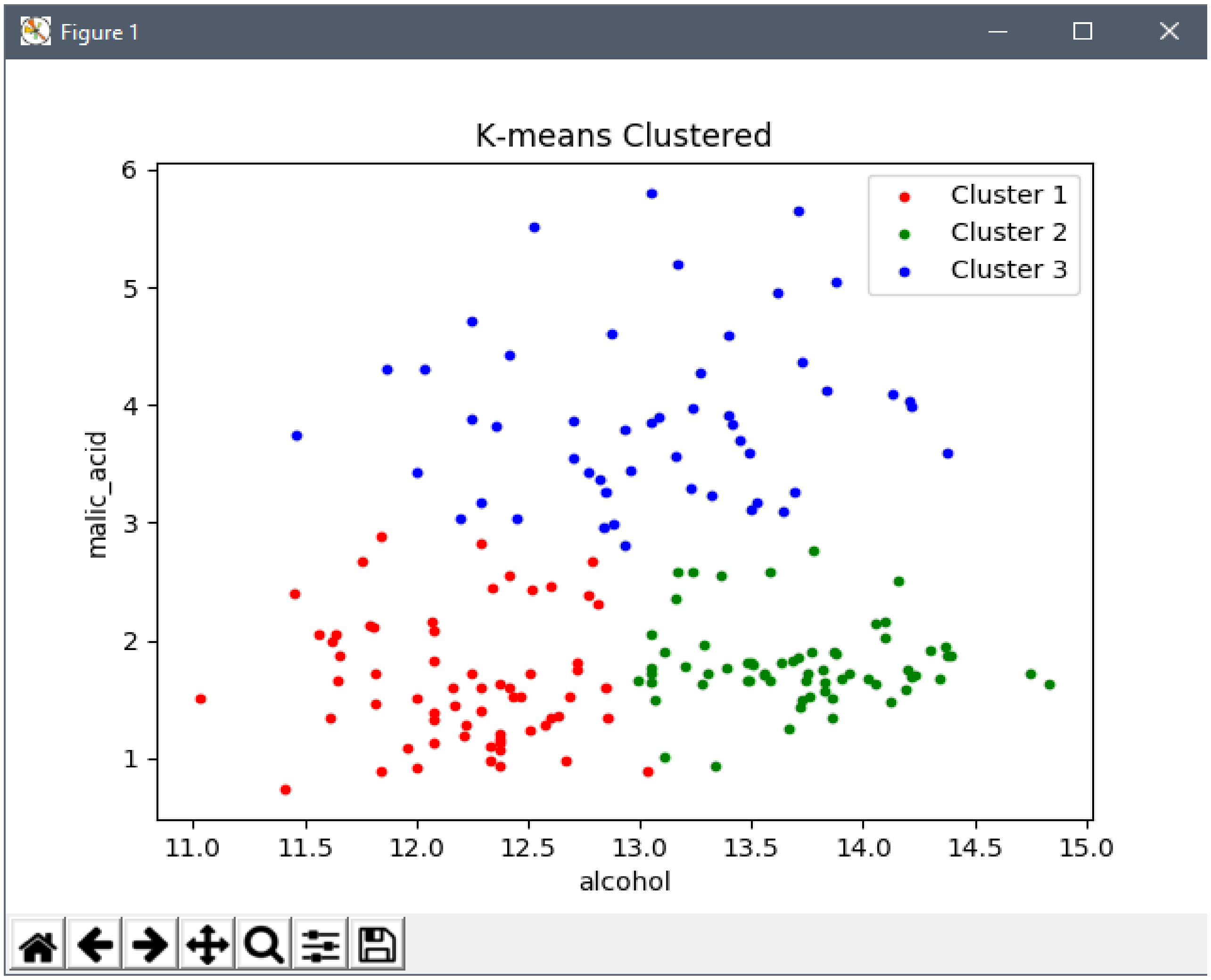


Figure 1
K-means Clustered
Cluster 1
Cluster 2
Cluster 3
malic_acid
alcohol
1
2
3
4
5
6
11.0
11.5
12.0
12.5
13.0
13.5
14.0
14.5
15.0

K-means main method at the time of submission of this assignment:

```
def cluster():
    "This function performs the k-means clustering algorithm"
    global count
    global centroid
    global calculatedCentroid
    global clustered
    global K
    distance = [0]*K
    del clustered[:]  # Empty the old cluster before appending new
    for i in range(K):
        clustered.append([])  # Making a sublist for every cluster

    for val in newData:
        for i in range(0, K):
            distance[i] = euclideanDistance(centroid[i], val)
            # Only this statement should be in the inner loop
            '''
            Here I am calculating the distance between centroid & each
            value to classify them into diffrent clusters
            '''
        minIndex = distance.index(min(distance))  # Both of these should be
        clustered[minIndex].append(val)           # should be in the outer loop

    del calculatedCentroid[:]  # Empty the calculatedCentroid list everytime
    for i in range(0, K):      # Calculate new centriods
        calculatedCentroid.append(meanCords(clustered[i]))

    if cmp(calculatedCentroid, centroid) == 0:  # Comapre old and new centriods
        return count

    centroid = list(calculatedCentroid)  # Copying new centriod to the old var
    count += 1
    cluster()  # Recursively calling cluster() again and again
```