Matúš Rusňák : 116286

K-means, Divisive and Agglomerative clustering algorithms

In this project I have implemented K-means, Divisive and Agglomerative algorithms that server to cluster data into K clusters. Data are represented by matrix of randomly generated points in a 2D coordinates. The program starts by generating those points and at first it generates 20 unique points and later chooses one of the already generated points and creates another close to it by offset radius 100. All of these algorithms output generated graphical plot, time it took to create clusters, what center calculation was used and the success rate of generated clusters.

1. Measurements

As I mentioned above for certain algorithms we can use 2 different cluster center calculations.

```python
def centroid_calculation(cluster: List[List[int]]) -> tuple[int, int]:
    """
    function to calculate centroid location in cluster
    :param cluster: cluster you want to calculate for
    :return: calculated centroid
    """
    x = 0
    y = 0
    for point in cluster:
        x += point[0]
        y += point[1]
    return int(x / len(cluster)), int(y / len(cluster))
```

This function takes as an argument a cluster and then it returns its centroid location. Centroid is calculating by taking the average location of the whole cluster.

```python
def medoid_calculation(cluster: List[List[int]]) -> tuple[int, int]:
    """
    helper function that calculates medoid coordinates from given cluster and returns its location
    :param cluster: we want to calculate medoid for
    :return: calculated medoid position
    """
    centroid = np.mean(cluster)
    medoid = cluster[np.argmin([sum((x - centroid) ** 2) for x in cluster])]
    return medoid[0], medoid[1]
```

This function takes as an argument a cluster and returns its medoid location. Calculation is made faster by using the NumPy library that can calculate centroid and medoid faster than in raw python. Function returns closest points in cluster to cluster center.

2. K-means clustering

This algorithm is used when we have a large number of data we want to cluster. Since its really fast. However on the other hand since it works on random samples the algorithm might sometimes be really bad compare to how we can cluster the data by just using our eyes. Therefore I repeat the K-means algorithm N times (in my case 10) and select the best cluster out of all generated clusters. The selection is done based on the overall success rate of generated clusters. Whole algorithm except center calculation is located in an object called KMeans.

Main logic of K-means algorithm:

```
for i in range(K_MEANS_ITERATIONS):
    init_clusters = self._choose_init_clusters()
    assigned_points = self._assign_points_to_init_clusters(init_clusters)
    while True:
        calculated_center_points = self._calculate_center_points(assigned_points)
        assigned_points = self._assign_points_to_recalculated_centers(calculated_center_points)
        if self.already_assigned_center_points.get(tuple(calculated_center_points)):
            break
        self.already_assigned_center_points[tuple(calculated_center_points)] = True
```

We repeat this whole process N times – Firstly we randomly choose k clusters from out generated clusters and then we calculate distances from all points to all the centers. We assign each point to the closest center point. Once that is done we repeat the following process until a condition: Calculate new center points again assign our generated points to clusters. If center points have been already created, we stop the while loop. Then do this again until you have done this N times.

At the end we choose the best clusters that have been generated from N iterations. We calculate for each output its success rate which is calculated by taking the average distance from the center, if that distance is greater than 500, we mark the cluster as unsuccessful. After this is done, we create a plot using matplotlib library to show how the algorithm clustered generated points

For this specific algorithm you can choose which center calculation should happen. Either centroid or medoid calculation. The results should differ a lot but we will look at that in the testing phase.

3. Divisive clustering

Divisive clustering follows a top-down approach. Meaning we start with only cluster that contains all the points and its later split into more based on our splitting logic. The way I implemented this algorithm is that I use reverse k-means algorithm to split clusters. Meaning we split the First initial cluster into 2 and then those new 2 to 4 and so on. The process runs until we have generated the wanted number of k clusters.

Main divisive logic:

```
def _top_down_k_means(self, cluster: List[List[int]]) -> List[List[int]] or None:
    """
    method that implements top down k-means meaning it selects 2 random points from given cluster and continues
    k- means algorithm like when we want to have 2 final clusters
    :param cluster: cluster we want to split
    :return: list of 2 new clusters
    """
    created_center_points = {}
    if self.current_num_of_clusters == self.k:
        return
    chosen_points = self._choose_2_points_as_clusters(cluster)
    assigned_points = self._assign_points_to_clusters(chosen_points, cluster)
    while True:
        recalculated_center_points = self._calculate_new_center_points(assigned_points)
        assigned_points = self._reassign_points_to_center(cluster, recalculated_center_points)
        if created_center_points.get(tuple(recalculated_center_points)):
            break
        created_center_points[tuple(recalculated_center_points)] = True
    self.current_num_of_clusters += 1
    return assigned_points.values()
```

This function chooses 2 random points from given sample. Then it assigns points again to closest centers like in K-means algorithm. Later it recalculates new center points based on center calculation which in this case will always be centroid and lastly it reassigned points to newly calculated centers. Process of center recalculation continues until we try calculate new center points that have been already created. Then it stops and returns 2 clusters.

We repeat the whole process N times (in my case 5) and then we choose the best generated clusters iteration. The selection is done based on the best variance of created clusters. Meaning the more the clusters of one iteration are evenly split the better. We choose one with the best variance using this function:

```
def _select_best_variance(self) -> List[List[List[int]]]:
    """
    method that selects best variance out of generated clusters, we repeat the whole k-means process n times (in my
    case 5 times) and then based on variance we select the best one. Best variance means that generated k clusters
    have the most evenly split number of points in all clusters
    :return: k clusters with the best variance
    """
    variances = []
    total_length = len(self.clusters[0])
    for final_cluster in self.final_clusters:
        variance = 1
        for values in final_cluster:
            variance = variance * (len(values) / total_length)
        variances.append(variance)
    max_variance_index = variances.index(max(variances))
    return self.final_clusters[max_variance_index]
```

4. Agglomerative clustering

Agglomerative clustering works by a down-top approach. Meaning we calculate all distances between clusters and merge two closest cluster into one. To not always calculate all distances between points we use a 2D matrix which will store all distances from points. When a cluster is merged we delete its distances from the matrix and recalculate new distances to add to the matrix. This way we can make it faster by a lot.

Main logic is pretty simple:

```
self._create_distance_heap()
while len(self.clusters) != self.k:
    index = self._find_closest_clusters()
    self._merge_closest_clusters(index)
```

At the start of the program we create the distance heap (matrix), after that is done we find the lowest distance in the matrix and return its index. By indexes we merge selected clusters into one

Merging is done by this method:

```
def _merge_closest_clusters(self, index: tuple[int, int]) -> None:
    """
    method that merges clusters by index and handles heap logic
    :param index: 2 indexes of closest clusters
    """
    cluster1 = self.clusters[index[0]]
    cluster2 = self.clusters[index[1]]
    new_cluster = []
    for cluster in cluster1:
        new_cluster.append(cluster)
    for cluster in cluster2:
        new_cluster.append(cluster)
    center_point = self.center_calculator(new_cluster)
    self._remove_points_from_heap_and_dict(index[0], index[1])
    self._recalculate_distances_in_heap(center_point)
    self.cluster_centers_by_index[len(self.clusters)] = center_point
    self.clusters.append(new_cluster)
```

Method merges points into one cluster, calculates new center points, calls a function to remove old cluster distances from distance heap and a dictionary, calls another function to recalculate distances in heap, and lastly adds merged cluster into an array of current clusters.

This whole process is repeated until we have our wanted k number of clusters. At the end we also calculate the generated clusters success rate.

5. Testing

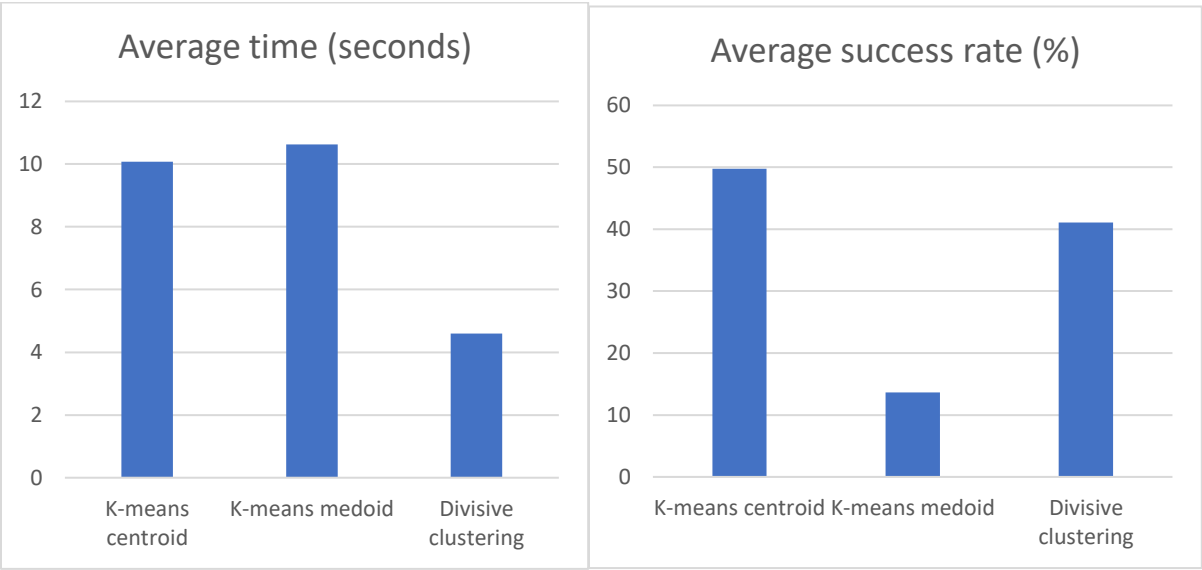
It is important to note that since all of these algorithms use some kind of random I implemented a seed to each random, that way results generated on your computer are reproducible. For this test we generated 40020 random unique points to be clustered. Here are my test results:

5.1 Main results

	Num of clusters	Time (seconds)	Success rate (%)
K-means centroid	3	2.6108842	0
	5	4.7726376	40
	7	5.5533682	42.85714286
	11	13.4888448	72.72727273
	15	23.9541765	93.33333333
K-means medoid	3	6.4086241	0
	5	7.2208319	0
	7	8.326435	14.28571429
	11	12.7186304	27.27272727
	15	18.4816199	26.66666667
Divisive clustering centroid	3	4.5484395	0
	5	2.7644438	20
	7	5.3330913	28.57142857
	11	4.7632638	63.63636364
	15	5.5618971	93.33333333

These results were run on my machine so you might get different values. All of these results are also stored in ./test_results.txt

5.2 Comparison

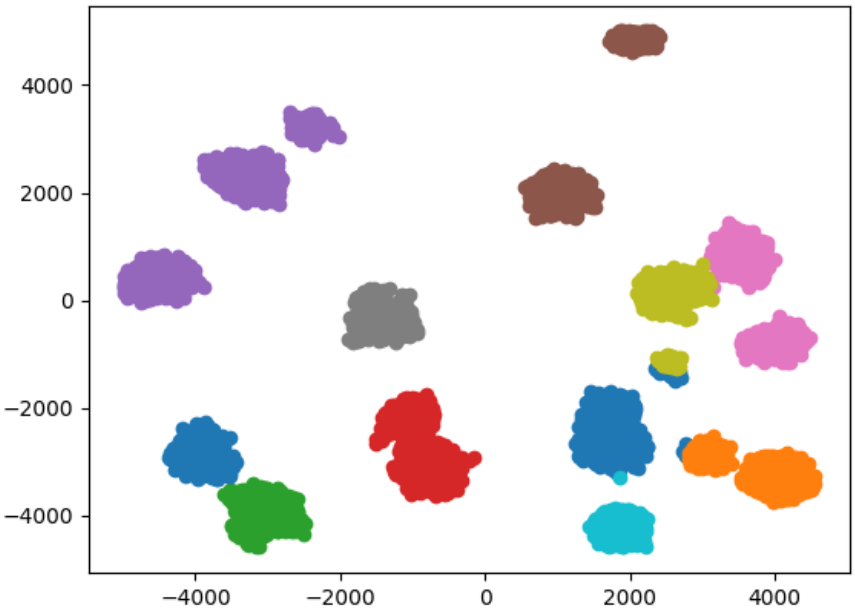


From comparison of time we can see that divisive clustering is the fastest. Divisive clustering was only tested on centroid calculation which is faster than medoid but I am not exactly sure why its that much faster compare to k-means. The only explanation that comes to my mind is that when we have only 2 center points which is always in divisive less iterations are needed to stop k-means algorithm.

From comparison of success rate we can see that K-means centroid is the clear winner. The default k-means center calculation is centroid that is why I think this shows the best results. But then again when we have centroid calculation in my opinion it handles outliers better than medoid. Which is why these test results look like this

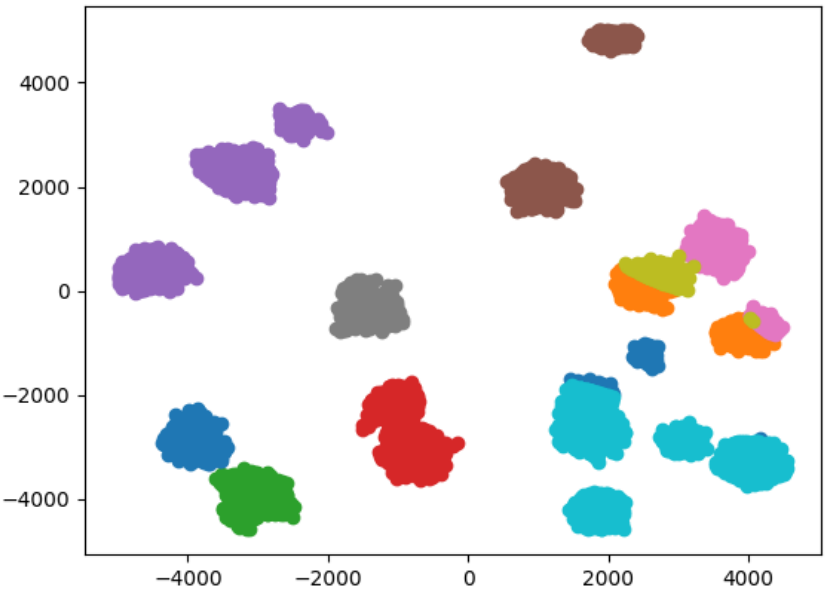
5.3 Visualization examples

K- means, centroid, 11 clusters

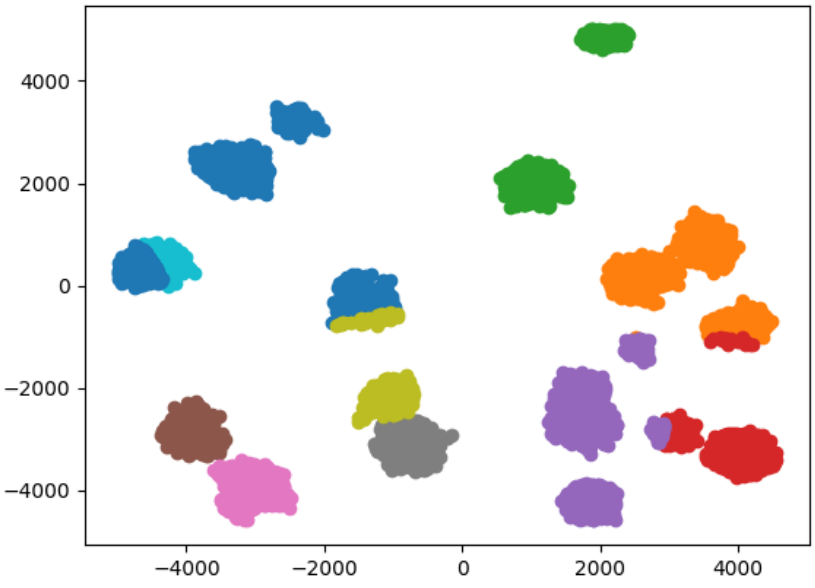


Matúš Rusňák : 116286

K -means, medoid, 11 clusters



Divisive clustering, centroid, 11 clusters



If you want to find more generated plots you can find them in ./Pictures. All plots were generated during this testing. Pictures are named after what they represent (example 3c – 3 = num clusters, c = centroid k-means)

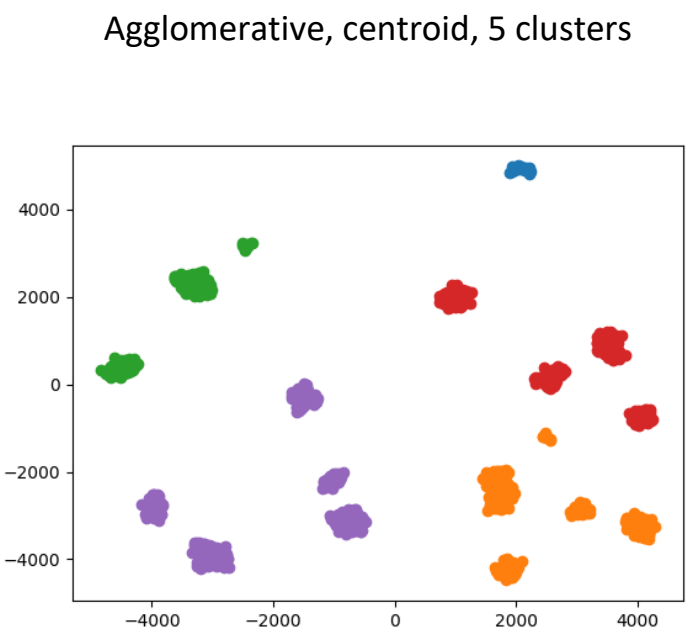
5.4 Agglomerative testing

Since agglomerative clustering is really slow testing 40020 randomly generated points would take a long time. That is why this has its own category. I did not optimize this algorithm to the fullest so its runtime is still N^3. Here are the test results:

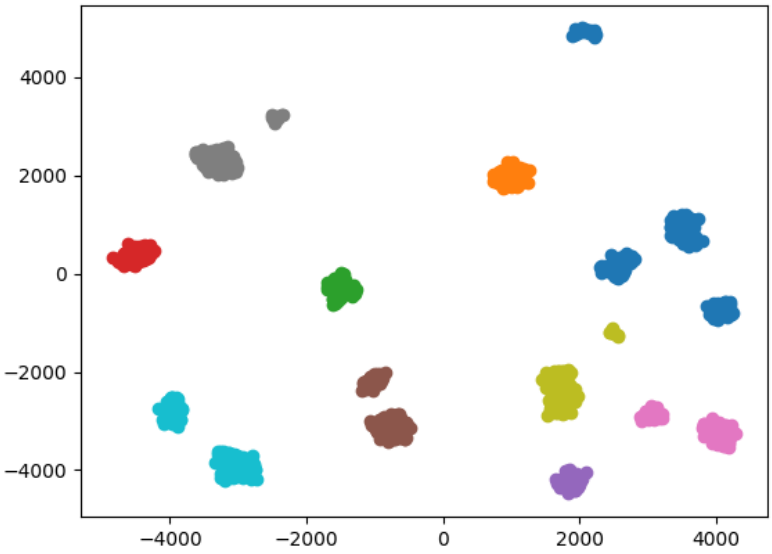
	Num of clusters	Time (seconds)	Success rate (%)
Agglomerative 1020	3	7.0471439	33.33333333
	5	6.1872581	20
	7	7.1394223	42.85714286
	9	6.4911562	66.66666667
	11	6.5996434	81.81818182
	15	6.3240708	93.33333333
Agglomerative 2020	3	64.4662673	33.33333333
	5	64.5844098	20
	7	66.9601268	42.85714286
	9	63.7039295	66.66666667
	11	63.5052718	81.81818182
	15	63.3929951	93.33333333

These tests were done on Agglomerative clustering with 1020 and 2020 random points. We can already see that a few more and the time it takes to run this algorithm skyrockets.

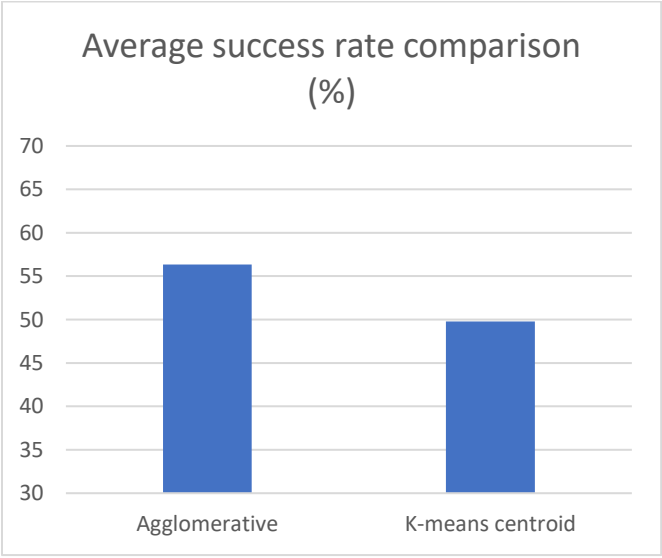
5.5 Agglomerative visualization examples



Agglomerative, centroid, 11 clusters



5.6 Test conclusion



If we look at this graph, we can see that agglomerative performs better than k-means. It is also not correct to compare these 2 averages because the number of generated points is vastly different. But what I wanted to say is that Agglomerative outperforms K -means by 5 %. With considering the amount of time Agglomerative costs to run is in my opinion not worth it. If we have a large dataset we should use K-means clustering and if we have relatively small, then we could try to run agglomerative since its more precision.

6. Usage

If you want to run this program change directory to project default and run the following command:

python main.py -k x -a y

Where *x* is the number of clusters you want the algorithms to create. X must be greater than 0 and lower then the actual number of points generated.

Where *y* is the clustering algorithm you want to use. Options: a = agglomerative, c = k-means with centroid calculation, m = k-means with medoid calculation, d = divisive clustering with centroid calculation

If you want to change the random seed, algorithm iterations, number of points generated look into the ./helpers/consts.py file. This file stores all constants used in this project. If you want to make changes, change a variable, save the file and run the project command again.

At the end programs outputs algorithm statistics and with the help of matplotlib it also creates a graphical plot.

7. Dependencies

This project was developed in ***Python 3.11*** so I recommend you use this version of python to avoid troubles. It also greatly improves runtime compared to the older version of python.

All library dependencies are stored in ***requirements.txt:***

NumPy==1.23.5

matplotlib==3.6.2