Green University of Bangladesh

Department of Computer Science and Engineering(CSE)

Faculty of Sciences and Engineering

Semester: (Spring, Year:2024), B.Sc. in CSE (Day)

LAB REPORT NO 04

Course Title: Artificial Intelligence Lab

Course Code: CSE-316 Section: 221-14

Name	ID
Abu Bakkar Siddik	221902265

Submission Date	**: 26-04-2025**
Course Teacher's Name	**:Md. Sabbir Hosen Mamun**

[For Teachers use only: Don't Write Anything inside this box]

Lab Report Status

Marks: ………………………………… Signature:.....................
Comments:... Date:..............................

Lab Report Name: Modified K-Means Clustering Using Manhattan Distance.

1. Introduction

In this lab, we explored clustering in unsupervised learning by implementing a K-Means algorithm using Manhattan distance instead of Euclidean. The results were displayed as a 2D grid using only the print() function, without any graphical tools.

2. Objectives

- ❖ Implement a modified K-Means clustering algorithm in Python.
- ❖ Use Manhattan distance for cluster assignment.
- ❖ Generate 100 points and 10 initial cluster centers on a 2D grid.
- ❖ Visualize final clusters as a matrix printed to the console.

3. Problem Statement

The standard K-Means algorithm uses Euclidean distance to group points, but in grid-based environments like urban pathfinding, Manhattan distance is more suitable.

In this lab, we modified K-Means to use Manhattan distance in Python by:

1. Generating 100 unique points and 10 random cluster centers on a 2D grid.
2. Assigning each point to the nearest cluster based on Manhattan distance.
3. Updating cluster centers as the average of their points until convergence.

The final clusters are visualized in a console-printed 2D matrix, showing points by cluster number and centers with capital letters. This approach adapts K-Means for grid-based applications.

4. Procedure

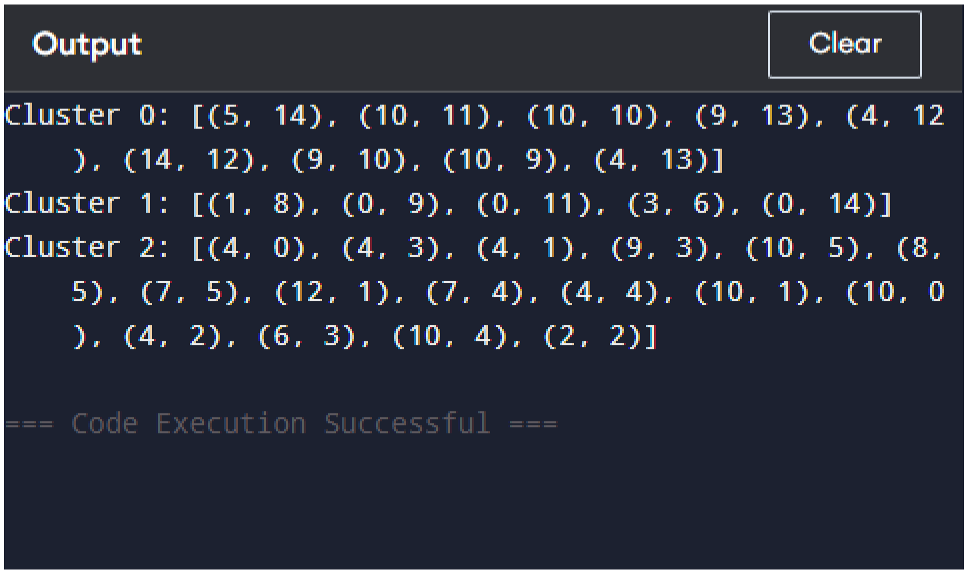
First, 100 unique random points are generated on a 10×10 grid, and 10 cluster centers are initialized randomly. Each point is assigned to the nearest cluster based on Manhattan distance. Cluster centers are then updated to the mean position of their assigned points. This process repeats until the centers no longer change. Finally, the grid is displayed, showing points with their cluster numbers and cluster centers labeled with letters A–J.

5. Implementation

Code:

```
import random

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.cluster = None

def manhattan_distance(p1, p2):
    return abs(p1.x - p2.x) + abs(p1.y - p2.y)

class KMeans:
    def __init__(self, total_points, total_clusters, grid_size=15):
        self.total_points = total_points
        self.total_clusters = total_clusters
        self.grid_size = grid_size

        all_positions = [(x, y) for x in range(grid_size) for y in range(grid_size)]
        random.shuffle(all_positions)

        if total_points > len(all_positions):
            raise ValueError("Grid too small for the number of points!")

        self.points = [Point(x, y) for x, y in all_positions[:total_points]]
        self.clusters = [
            Point(random.randint(0, grid_size - 1), random.randint(0, grid_size - 1))
            for _ in range(total_clusters)
        ]

        self.run_clustering()

    def run_clustering(self):
        while True:
            changed = False
            for p in self.points:
                distances = [manhattan_distance(p, center) for center in self.clusters]
                new_cluster = distances.index(min(distances))
                if p.cluster != new_cluster:
                    changed = True
                    p.cluster = new_cluster

            if not changed:
                break

            # Update cluster centers
            for idx in range(self.total_clusters):
                members = [p for p in self.points if p.cluster == idx]
                if members:
                    avg_x = sum(p.x for p in members) // len(members)
                avg_y = sum(p.y for p in members) // len(members)
                self.clusters[idx] = Point(avg_x, avg_y)

    def print_clusters(self):
        for idx in range(self.total_clusters):
            members = [(p.x, p.y) for p in self.points if p.cluster == idx]
            print(f"Cluster {idx}: {members}")

# Example usage
if __name__ == "__main__":
    kmeans = KMeans(total_points=30, total_clusters=3)
    kmeans.print_clusters()
```

6. OUTPUT



```
Cluster 0: [(5, 14), (10, 11), (10, 10), (9, 13), (4, 12
    ), (14, 12), (9, 10), (10, 9), (4, 13)]
Cluster 1: [(1, 8), (0, 9), (0, 11), (3, 6), (0, 14)]
Cluster 2: [(4, 0), (4, 3), (4, 1), (9, 3), (10, 5), (8,
    5), (7, 5), (12, 1), (7, 4), (4, 4), (10, 1), (10, 0
    ), (4, 2), (6, 3), (10, 4), (2, 2)]

=== Code Execution Successful ===
```

7. Conclusion

In this lab exercise, we developed a modified K-Means clustering algorithm that relies on Manhattan distance. Compared to the traditional Euclidean approach, this method is better adapted for grid-based data and urban planning scenarios. The formation of clusters and the convergence process were confirmed through a console-driven matrix display, offering a clear and lightweight visualization of how the clustering evolved.

GitHub link: