Final Term Project

Name: Adit Nuwal
UCID: an238
Email: an238@njit.edu
Option Chosen: Option 2 – Unsupervised Data Mining (Clustering)

Table of Contents

Final Term Project Report – Option 2

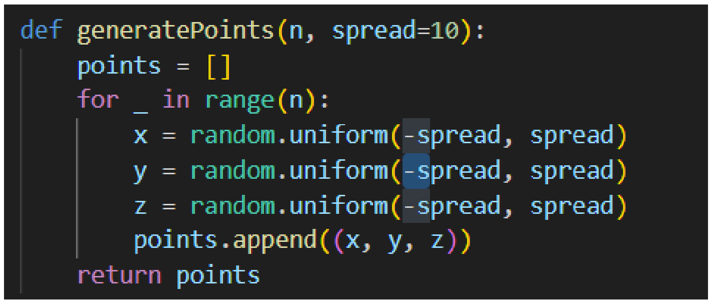
Name: Adit Nuwal
UCID: an238
Email: an238@njit.edu
Option Chosen: Option 2 – Unsupervised Data Mining (Clustering)

1. Introduction

This project implements clustering and outlier detection techniques from scratch using Python. Three datasets of 500 points are randomly generated in 3D Euclidean space, outliers are found out and removed for a cleaner dataset. This new cleaned dataset is then used for clustering and analyzed using K-Means and Hierarchical Agglomerative Clustering (HAC) and Silhouette Coefficient is used to evaluate the performance of the algorithms. Then a comparison is made and the better algorithm for that dataset is printed out.

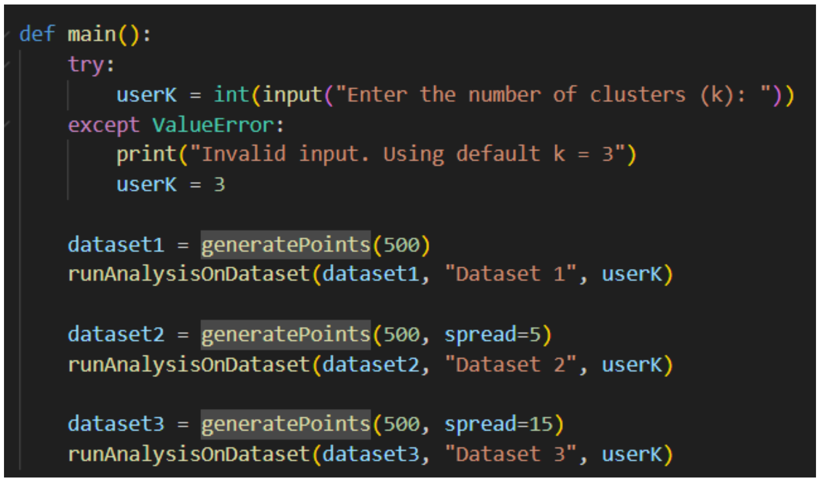
2. Dataset Generation and Outlier Detection

The dataset is created by generating 500 random points in 3D Euclidean space using the function generatePoints(n, spread) from the code. Each point is a coordinate (x, y, z) where the values are generated using random.uniform(-spread, spread) (see line inside generatePoints). This spread parameter controls how compact or dispersed the points are. For example, when spread=5, the points are tightly clustered near the origin, while a spread=15 results in much wider dispersion.



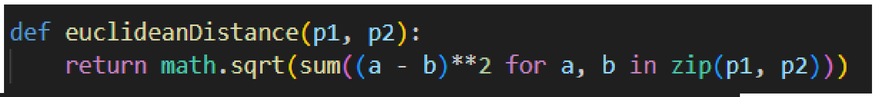
```
def generatePoints(n, spread=10):
    points = []
    for _ in range(n):
        x = random.uniform(-spread, spread)
        y = random.uniform(-spread, spread)
        z = random.uniform(-spread, spread)
        points.append((x, y, z))
    return points
```

By adjusting this parameter, three distinct datasets are created to simulate different clustering scenarios (see the calls to generatePoints() in the main() function).

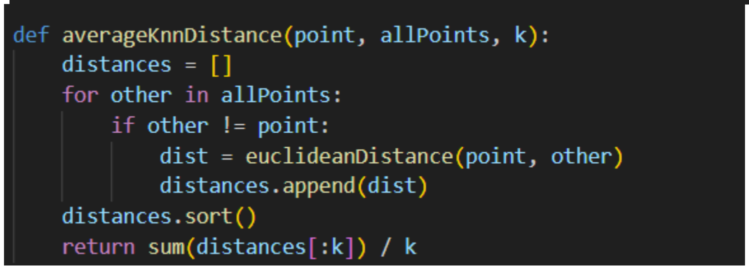


```
def main():
    try:
        userK = int(input("Enter the number of clusters (k): "))
    except ValueError:
        print("Invalid input. Using default k = 3")
        userK = 3

    dataset1 = generatePoints(500)
    runAnalysisOnDataset(dataset1, "Dataset 1", userK)

    dataset2 = generatePoints(500, spread=5)
    runAnalysisOnDataset(dataset2, "Dataset 2", userK)

    dataset3 = generatePoints(500, spread=15)
    runAnalysisOnDataset(dataset3, "Dataset 3", userK)
```

Once the dataset S is generated, the program proceeds to identify and remove outliers that are points that significantly deviate from their local neighborhood. This is done using a distance-based approach in the detectOutliers(points, k=5) function. For each point, the function averageKnnDistance() calculates its average distance to its 5 nearest neighbors using the euclideanDistance(p1, p2) function.

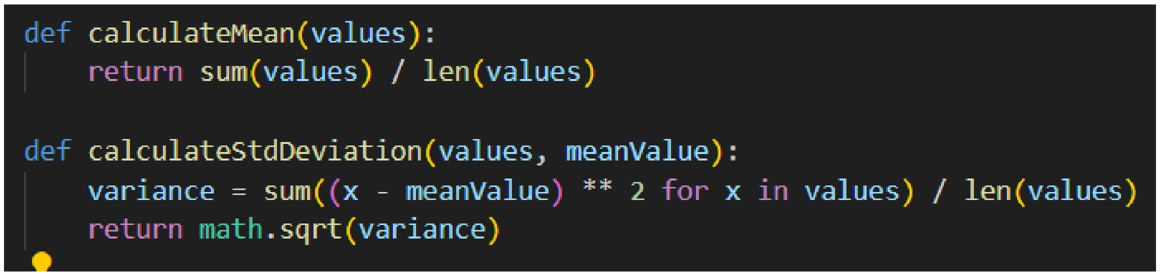


```python
def euclideanDistance(p1, p2):
    return math.sqrt(sum((a - b)**2 for a, b in zip(p1, p2)))
```



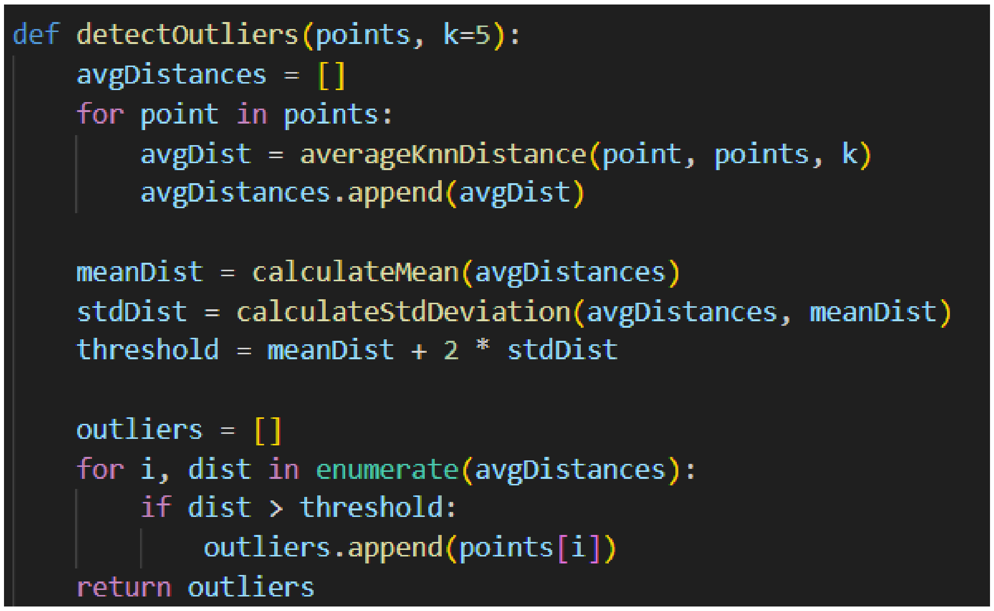
```python
def averageKnnDistance(point, allPoints, k):
    distances = []
    for other in allPoints:
        if other != point:
            dist = euclideanDistance(point, other)
            distances.append(dist)
    distances.sort()
    return sum(distances[:k]) / k
```

These average distances are collected and then used to compute the overall mean and standard deviation via calculateMean() and calculateStdDeviation().



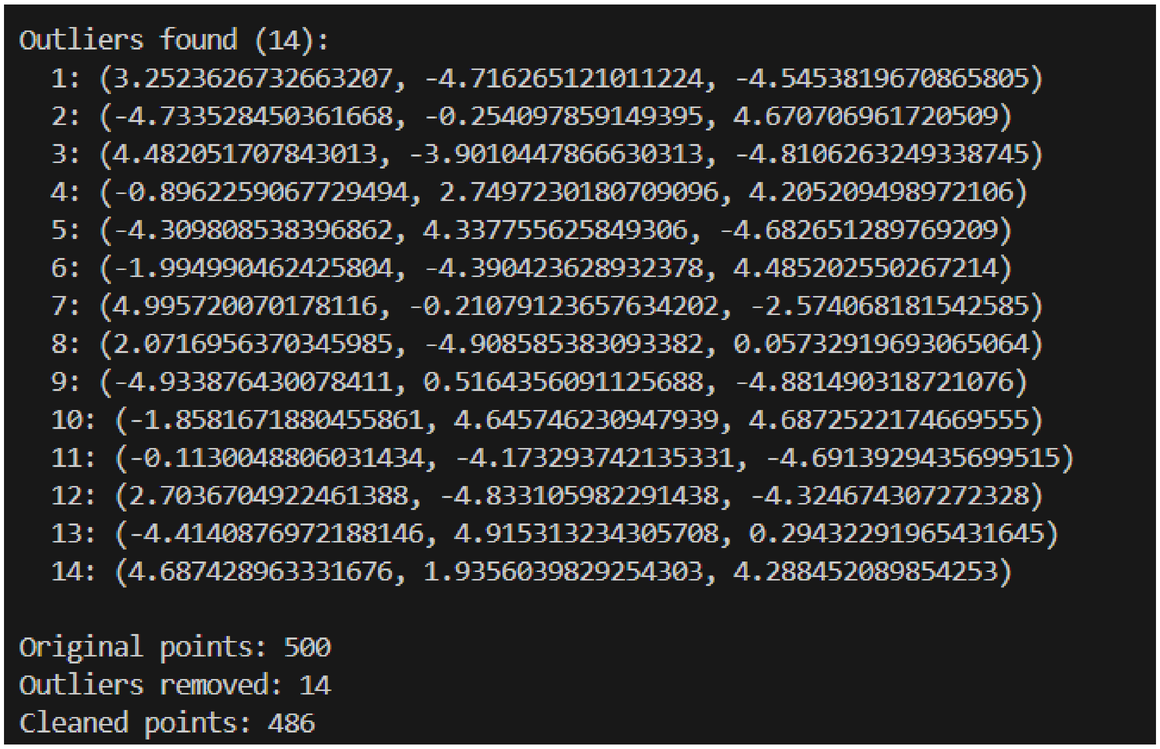
```python
def calculateMean(values):
    return sum(values) / len(values)

def calculateStdDeviation(values, meanValue):
    variance = sum((x - meanValue) ** 2 for x in values) / len(values)
    return math.sqrt(variance)
```

If a point's average distance to its neighbors exceeds the mean by more than 2 standard deviations, it is considered an outlier (see the threshold check inside detectOutliers).



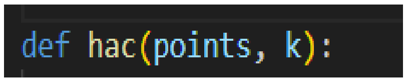
```python
def detectOutliers(points, k=5):
    avgDistances = []
    for point in points:
        avgDist = averageKnnDistance(point, points, k)
        avgDistances.append(avgDist)

    meanDist = calculateMean(avgDistances)
    stdDist = calculateStdDeviation(avgDistances, meanDist)
    threshold = meanDist + 2 * stdDist

    outliers = []
    for i, dist in enumerate(avgDistances):
        if dist > threshold:
            outliers.append(points[i])
    return outliers
```

These outliers are printed to the console and removed from the original dataset.

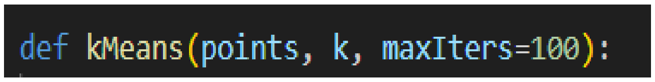


```
Outliers found (14):
  1: (3.2523626732663207, -4.716265121011224, -4.5453819670865805)
  2: (-4.733528450361668, -0.254097859149395, 4.670706961720509)
  3: (4.482051707843013, -3.9010447866630313, -4.8106263249338745)
  4: (-0.8962259067729494, 2.7497230180709096, 4.205209498972106)
  5: (-4.309808538396862, 4.337755625849306, -4.682651289769209)
  6: (-1.994990462425804, -4.390423628932378, 4.485202550267214)
  7: (4.995720070178116, -0.21079123657634202, -2.574068181542585)
  8: (2.0716956370345985, -4.908585383093382, 0.05732919693065064)
  9: (-4.933876430078411, 0.5164356091125688, -4.881490318721076)
  10: (-1.8581671880455861, 4.645746230947939, 4.6872522174669555)
  11: (-0.1130048806031434, -4.173293742135331, -4.6913929435699515)
  12: (2.7036704922461388, -4.833105982291438, -4.324674307272328)
  13: (-4.4140876972188146, 4.915313234305708, 0.29432291965431645)
  14: (4.687428963331676, 1.9356039829254303, 4.288452089854253)

Original points: 500
Outliers removed: 14
Cleaned points: 486
```

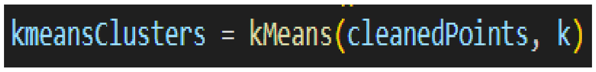
The cleaned dataset is referred to as S’ and is used as input for the clustering algorithms.



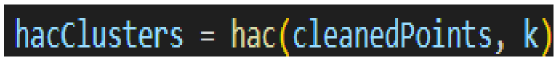
```
def hac(points, k):
```



```
def kMeans(points, k, maxIters=100):
```



```
kmeansClusters = kMeans(cleanedPoints, k)
```



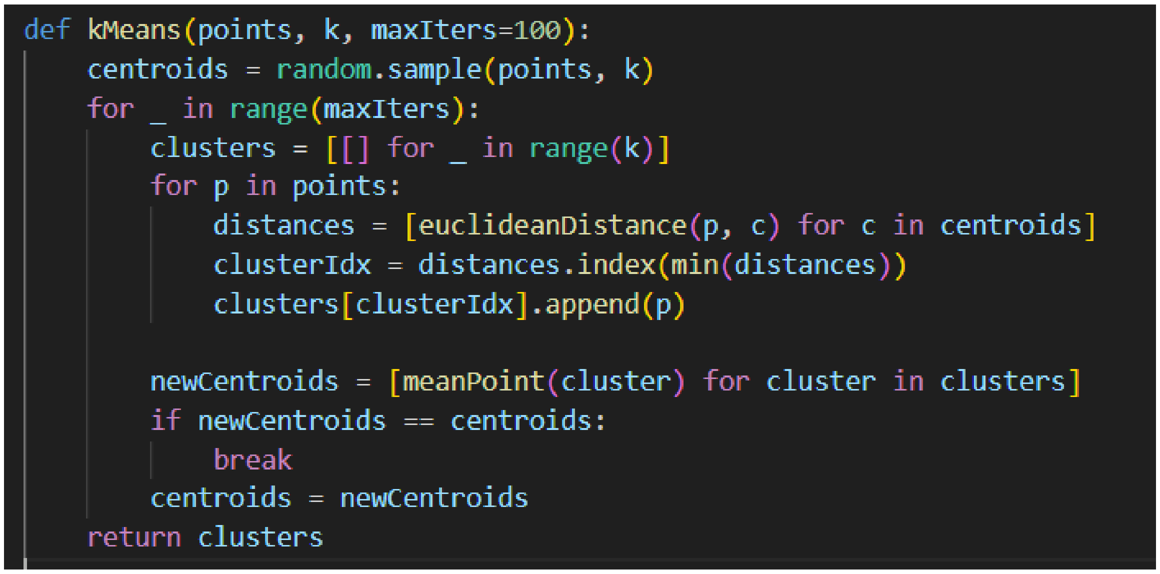
```
hacClusters = hac(cleanedPoints, k)
```

This preprocessing step ensures that noise and extreme points do not skew the clustering results. By focusing only on meaningful, representative data, the clustering algorithms are able to detect more accurate and reliable structures within each dataset.

3. Clustering Algorithms

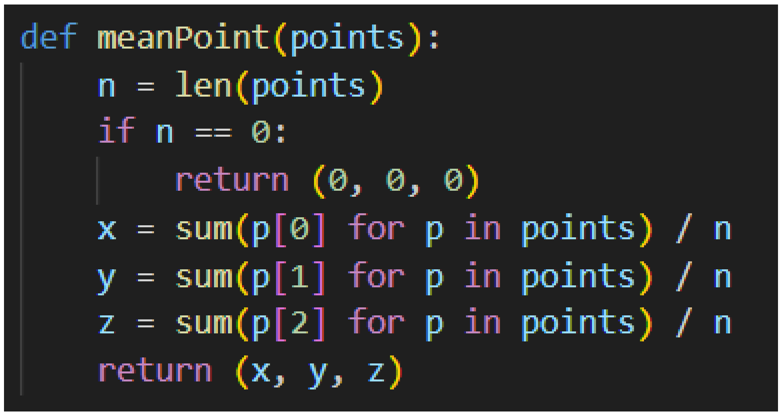
After removing outliers, the cleaned dataset S’ is clustered using two unsupervised learning algorithms: K-Means and Hierarchical Agglomerative Clustering (HAC).

The kMeans(points, k, maxIters=100) function implements the K-Means algorithm, which begins by randomly selecting k initial centroids using the random.sample() method. These centroids serve as initial cluster centers. Each point is then assigned to the cluster corresponding to the nearest centroid, with distance calculated using the custom euclideanDistance() function.



```
def kMeans(points, k, maxIters=100):
    centroids = random.sample(points, k)
    for _ in range(maxIters):
        clusters = [[] for _ in range(k)]
        for p in points:
            distances = [euclideanDistance(p, c) for c in centroids]
            clusterIdx = distances.index(min(distances))
            clusters[clusterIdx].append(p)

        newCentroids = [meanPoint(cluster) for cluster in clusters]
        if newCentroids == centroids:
            break
        centroids = newCentroids
    return clusters
```

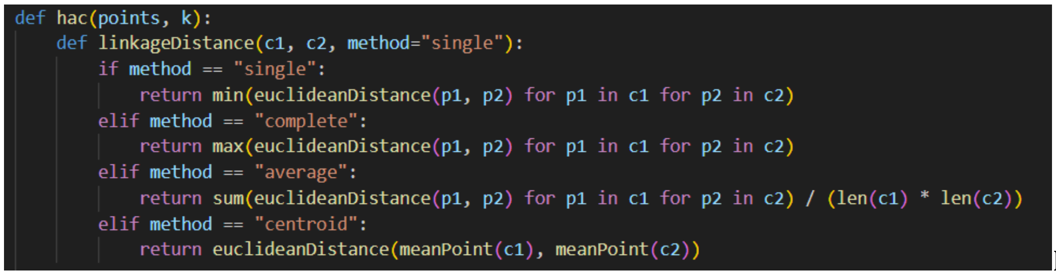
After assigning all points, new centroids are computed for each cluster using meanPoint(cluster), which returns the average of all x, y, and z coordinates within that cluster.



```
def meanPoint(points):
    n = len(points)
    if n == 0:
        return (0, 0, 0)
    x = sum(p[0] for p in points) / n
    y = sum(p[1] for p in points) / n
    z = sum(p[2] for p in points) / n
    return (x, y, z)
```

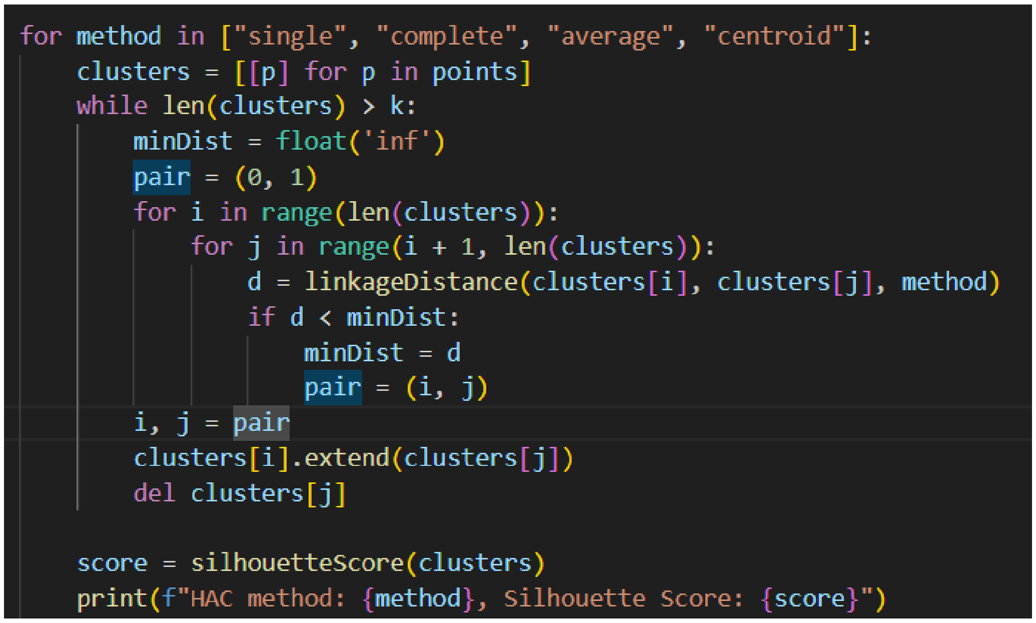
The algorithm iterates over this assignment-and-update process until either the centroids stabilize (i.e., do not change between iterations) or the maximum number of iterations (maxIters) is reached. The final result is a list of k clusters containing the grouped 3D points. The quality of this clustering is then assessed using the silhouetteScore() function, which evaluates intra-cluster compactness and inter-cluster separation. This approach is well-suited to datasets where clusters are relatively spherical and evenly distributed in space.

The hac(points, k) function implements HAC using a bottom-up approach. It starts by placing each point into its own cluster. It then repeatedly merges the two clusters that are closest together until only k clusters remain. The distance between clusters is calculated using the helper function linkageDistance(c1, c2, method), which supports four different strategies: single, complete, average, and centroid linkage.



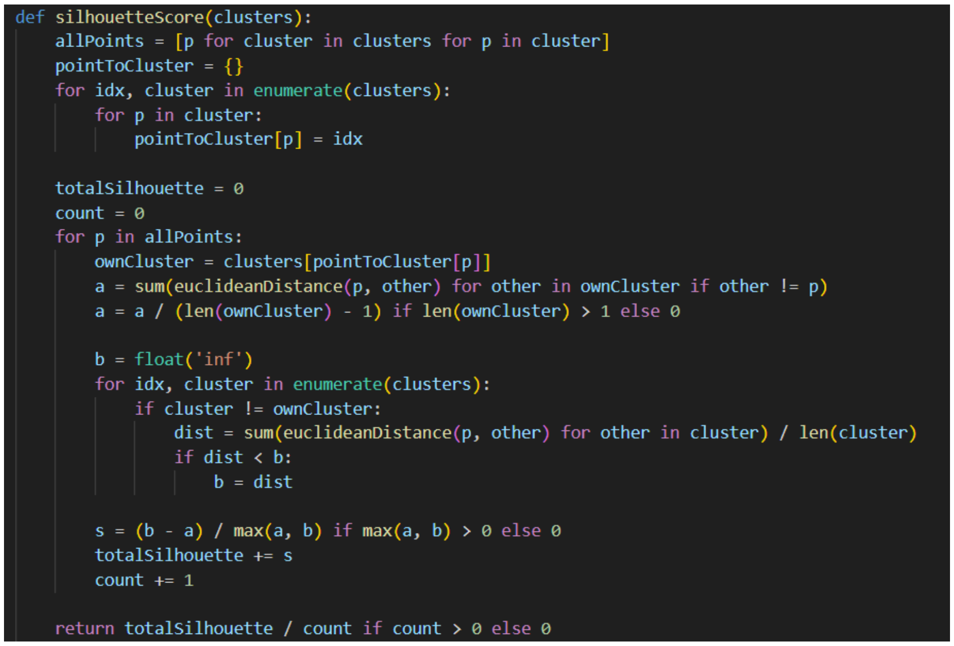
```python
def hac(points, k):
    def linkageDistance(c1, c2, method="single"):
        if method == "single":
            return min(euclideanDistance(p1, p2) for p1 in c1 for p2 in c2)
        elif method == "complete":
            return max(euclideanDistance(p1, p2) for p1 in c1 for p2 in c2)
        elif method == "average":
            return sum(euclideanDistance(p1, p2) for p1 in c1 for p2 in c2) / (len(c1) * len(c2))
        elif method == "centroid":
            return euclideanDistance(meanPoint(c1), meanPoint(c2))
```

Each strategy defines “closeness” differently like single linkage looks at the shortest distance between any two points across two clusters, while centroid computes the Euclidean distance between the cluster means.



```python
for method in ["single", "complete", "average", "centroid"]:
    clusters = [[p] for p in points]
    while len(clusters) > k:
        minDist = float('inf')
        pair = (0, 1)
        for i in range(len(clusters)):
            for j in range(i + 1, len(clusters)):
                d = linkageDistance(clusters[i], clusters[j], method)
                if d < minDist:
                    minDist = d
                    pair = (i, j)
        i, j = pair
        clusters[i].extend(clusters[j])
        del clusters[j]

    score = silhouetteScore(clusters)
    print(f"HAC method: {method}, Silhouette Score: {score}")
```

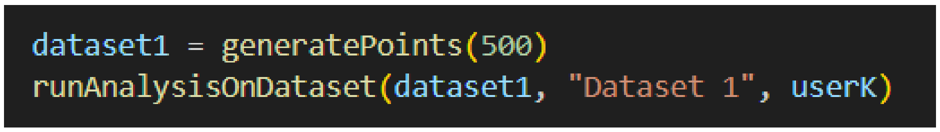
HAC is run separately for each of the four linkage strategies, and the resulting clusters are evaluated using the silhouetteScore() function. The strategy that yields the highest silhouette score is selected as the best method for that dataset. This makes HAC more flexible than K-Means for datasets with irregular cluster shapes, varying densities, or hierarchical structures.



```python
def silhouetteScore(clusters):
    allPoints = [p for cluster in clusters for p in cluster]
    pointToCluster = {}
    for idx, cluster in enumerate(clusters):
        for p in cluster:
            pointToCluster[p] = idx

    totalSilhouette = 0
    count = 0
    for p in allPoints:
        ownCluster = clusters[pointToCluster[p]]
        a = sum(euclideanDistance(p, other) for other in ownCluster if other != p)
        a = a / (len(ownCluster) - 1) if len(ownCluster) > 1 else 0

        b = float('inf')
        for idx, cluster in enumerate(clusters):
            if cluster != ownCluster:
                dist = sum(euclideanDistance(p, other) for other in cluster) / len(cluster)
                if dist < b:
                    b = dist

        s = (b - a) / max(a, b) if max(a, b) > 0 else 0
        totalSilhouette += s
        count += 1

    return totalSilhouette / count if count > 0 else 0
```

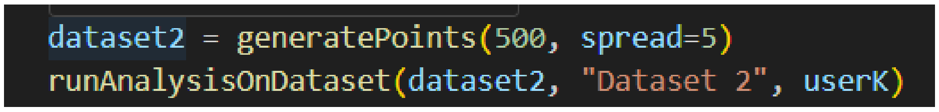
4. Datasets and Parameters

This project evaluates the performance of clustering algorithms across three different datasets, all generated synthetically in a 3-dimensional Euclidean space. Each dataset contains 500 points, and all data is generated using the generatePoints(n, spread) function, where the spread parameter controls how dispersed the data points are in the 3D space. The points are generated with random.uniform(-spread, spread) along each axis (x, y, z), resulting in uniformly distributed values within a cubic region. This allows the generation of datasets with varied clustering difficulty, simulating different real-world data distributions. The use of a fixed random seed (random.seed(42)) ensures reproducibility and the same points are generated across multiple runs for consistent testing and evaluation.
Dataset 1: Default Spread (spread=10) This is the baseline dataset where points are generated in a cube ranging from -10 to 10 on each axis. This spread typically results in moderate separation between natural clusters, allowing both K-Means and HAC to perform fairly well. It serves as a general-purpose dataset to benchmark the effectiveness of both algorithms under standard conditions.



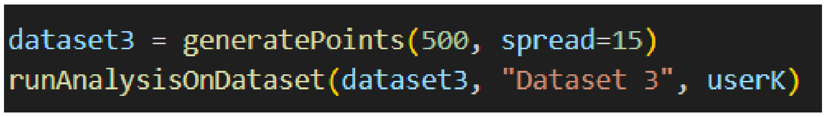
```
dataset1 = generatePoints(500)
runAnalysisOnDataset(dataset1, "Dataset 1", userK)
```

Dataset 2: Tight Spread (spread=5) This dataset has a smaller spread, meaning all points are clustered more closely around the origin (within a -5 to 5 range in each dimension). As a result, the distances between points are shorter and clusters are more compact. This configuration is ideal for K-Means, which favors spherical, equally-sized clusters. However, if the clusters overlap slightly, HAC may still outperform in identifying nuanced boundaries depending on the linkage strategy used.



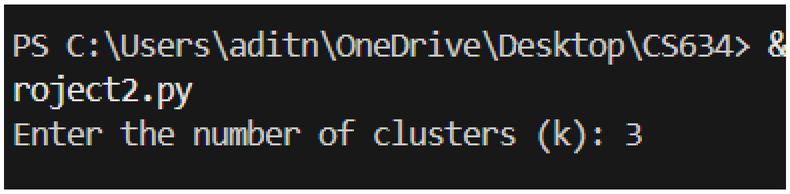
```
dataset2 = generatePoints(500, spread=5)
runAnalysisOnDataset(dataset2, "Dataset 2", userK)
```

Dataset 3: Wide Spread (spread=15) In this dataset, points are generated over a larger area (range: -15 to 15). This produces more sparse, scattered data, which may include loosely formed or unevenly spaced clusters. K-Means often struggles in such scenarios because it assumes cluster compactness and equal density. HAC, especially with average or complete linkage, is typically more adaptable in these conditions and may produce more meaningful groupings.



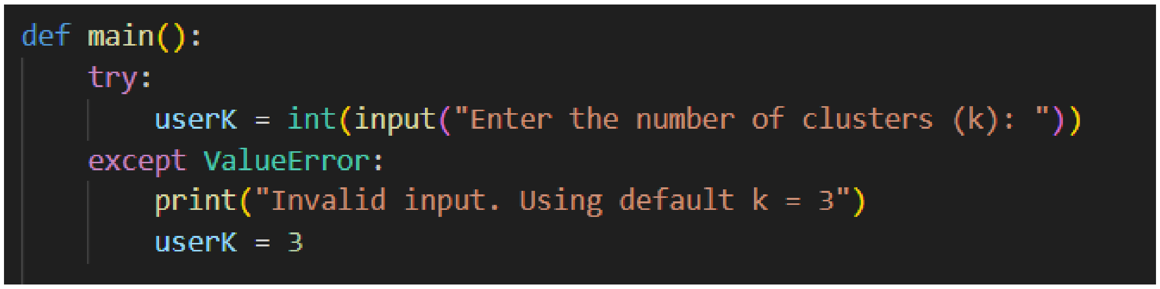
```
dataset3 = generatePoints(500, spread=15)
runAnalysisOnDataset(dataset3, "Dataset 3", userK)
```

User-Specified Parameter – Number of Clusters (k) The number of desired clusters k is specified by the user at runtime via input. This value is passed into both clustering functions: kMeans(points, k) and hac(points, k). Internally, this value controls: The number of centroids in K-Means. The stopping condition in HAC (i.e., stop merging when k clusters remain).



```
PS C:\Users\aditn\OneDrive\Desktop\CS634> &
roject2.py
Enter the number of clusters (k): 3
```

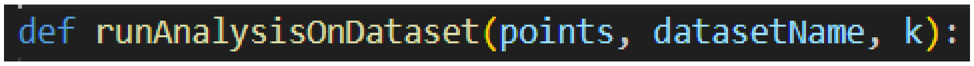
If the user provides invalid input (non-integer or blank), a default value of k = 3 is automatically used. This parameter allows flexibility in testing different cluster configurations and helps simulate varying real-world clustering objectives.



```python
def main():
    try:
        userK = int(input("Enter the number of clusters (k): "))
    except ValueError:
        print("Invalid input. Using default k = 3")
        userK = 3
```

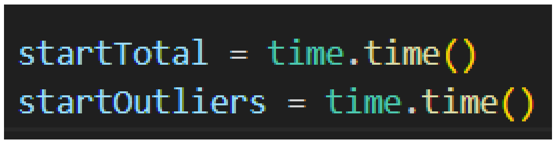
5. Dataset Analysis

The function runAnalysisOnDataset(points, datasetName, k) represents the core analysis pipeline of the project. It takes in a dataset (points), a string label for the dataset (datasetName), and the number of clusters k to apply both clustering algorithms (K-Means and HAC).



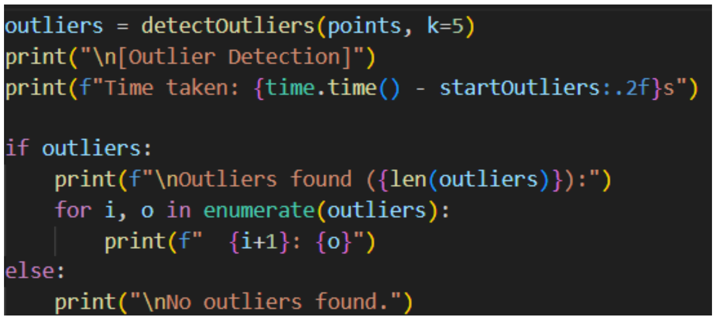
```python
def runAnalysisOnDataset(points, datasetName, k):
```

The function performs a full end-to-end processing and evaluation workflow on a given dataset. First, the function begins by printing a label identifying the dataset and the chosen number of clusters, and it logs the total runtime using startTotal = time.time().



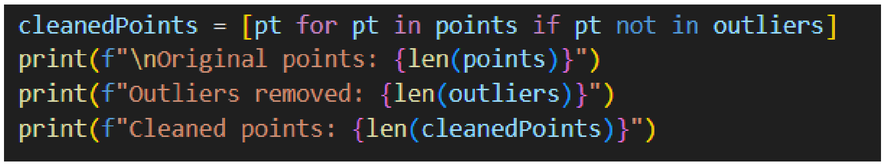
```python
startTotal = time.time()
startOutliers = time.time()
```

The first analytical step is outlier detection, handled by the function detectOutliers(), which finds points whose average distance to their 5 nearest neighbors is statistically anomalous. The time taken to perform this task is measured using startOutliers, and both the number of outliers and the outlier points themselves are printed to the console.



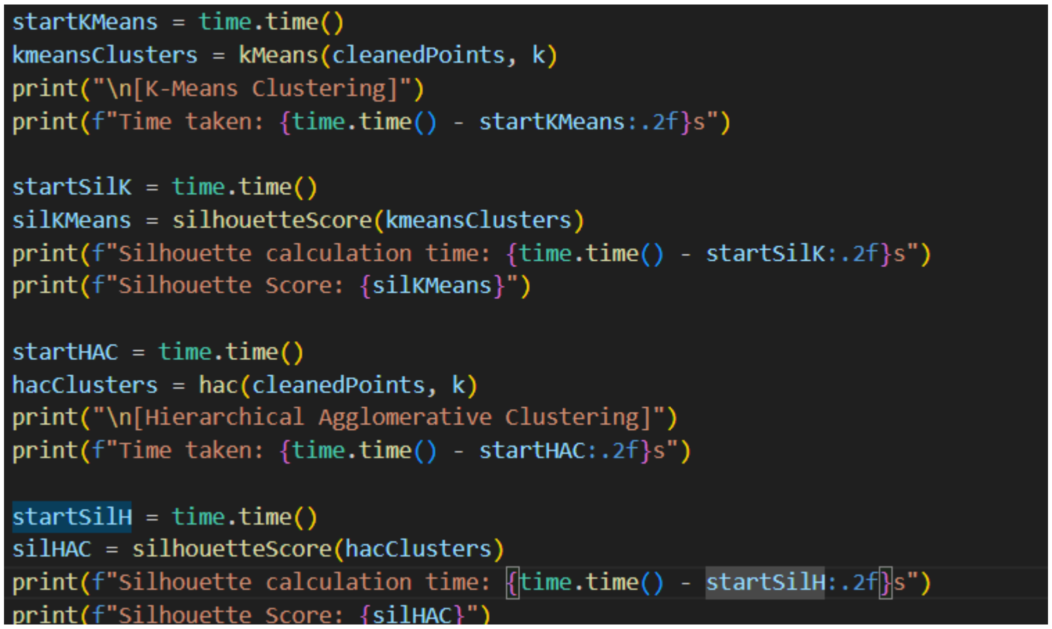
```python
outliers = detectOutliers(points, k=5)
print("\n[Outlier Detection]")
print(f"Time taken: {time.time() - startOutliers:.2f}s")

if outliers:
    print(f"\nOutliers found ({len(outliers)}):")
    for i, o in enumerate(outliers):
        print(f"  {i+1}: {o}")
else:
    print("\nNo outliers found.")
```

Once the outliers are detected, they are removed from the dataset to form a cleaned dataset cleanedPoints, representing *S*'. The script prints how many points were removed and how many remain, ensuring transparency in preprocessing.



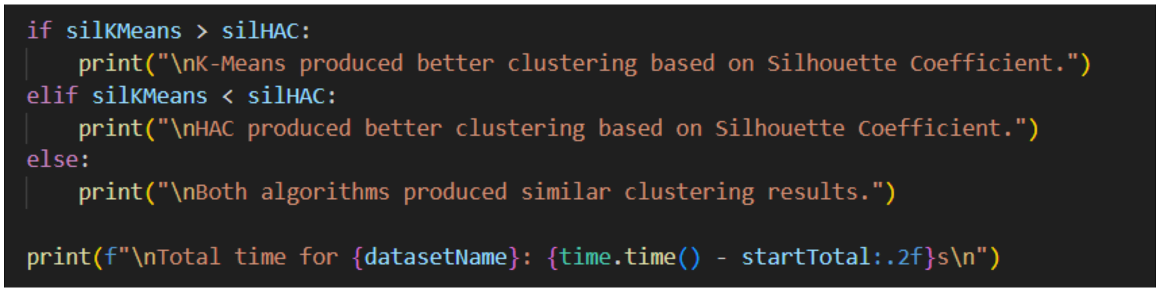
```python
cleanedPoints = [pt for pt in points if pt not in outliers]
print(f"\nOriginal points: {len(points)}")
print(f"Outliers removed: {len(outliers)}")
print(f"Cleaned points: {len(cleanedPoints)}")
```

Next, the function performs K-Means clustering on the cleaned dataset by calling kMeans(cleanedPoints, k). This groups the data into k clusters using iterative centroid optimization. The duration of this step is recorded with startKMeans, and the resulting clusters are passed to the silhouetteScore() function to evaluate the clustering quality based on cohesion and separation. The time to compute the silhouette score is also printed, giving a sense of computational cost. Following this, the same cleaned dataset is clustered using Hierarchical Agglomerative Clustering (HAC) via the hac(cleanedPoints, k) function. HAC is run with all four linkage methods (single, complete, average, and centroid), and the best one is selected based on silhouette score. Like with K-Means, the HAC clustering runtime and silhouette score computation time are both measured and printed.



```python
startKMeans = time.time()
kmeansClusters = kMeans(cleanedPoints, k)
print("\n[K-Means Clustering]")
print(f"Time taken: {time.time() - startKMeans:.2f}s")

startSilK = time.time()
silKMeans = silhouetteScore(kmeansClusters)
print(f"Silhouette calculation time: {time.time() - startSilK:.2f}s")
print(f"Silhouette Score: {silKMeans}")

startHAC = time.time()
hacClusters = hac(cleanedPoints, k)
print("\n[Hierarchical Agglomerative Clustering]")
print(f"Time taken: {time.time() - startHAC:.2f}s")

startSilH = time.time()
silHAC = silhouetteScore(hacClusters)
print(f"Silhouette calculation time: {time.time() - startSilH:.2f}s")
print(f"Silhouette Score: {silHAC}")
```

Finally, the function compares the two silhouette scores from K-Means and from HAC and prints which algorithm produced the better clustering result for that specific dataset. It concludes by displaying the total runtime for all operations, giving a clear performance snapshot.

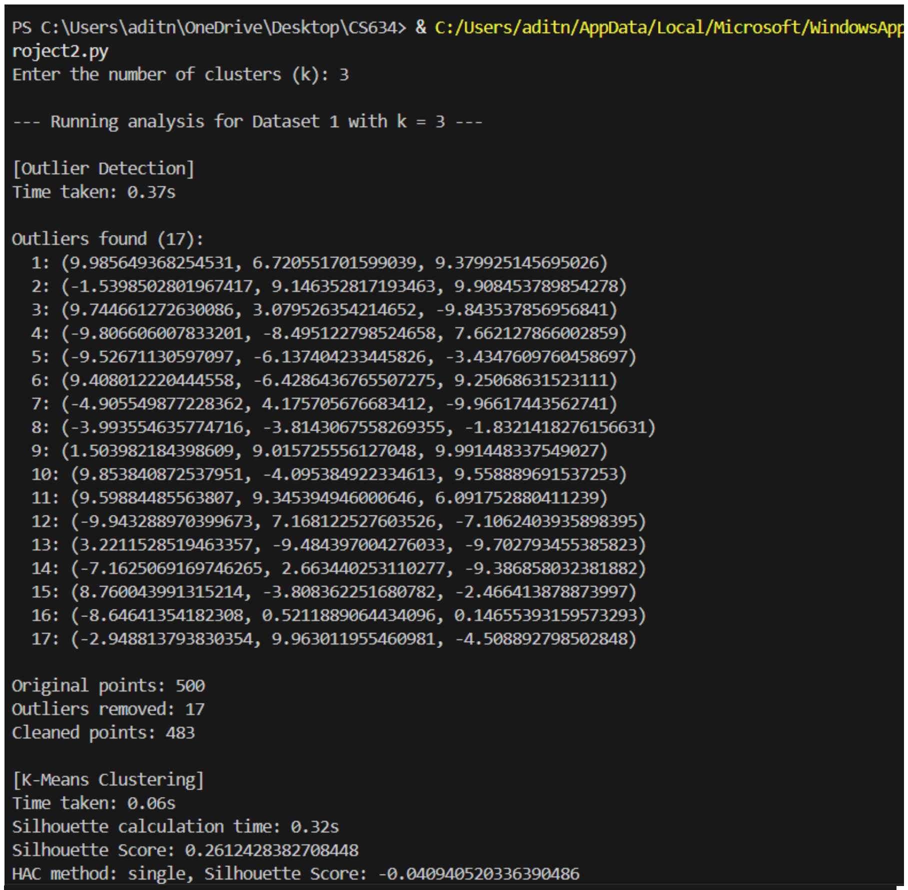


```
if silKMeans > silHAC:
    print("\nK-Means produced better clustering based on Silhouette Coefficient.")
elif silKMeans < silHAC:
    print("\nHAC produced better clustering based on Silhouette Coefficient.")
else:
    print("\nBoth algorithms produced similar clustering results.")

print(f"\nTotal time for {datasetName}: {time.time() - startTotal:.2f}s\n")
```

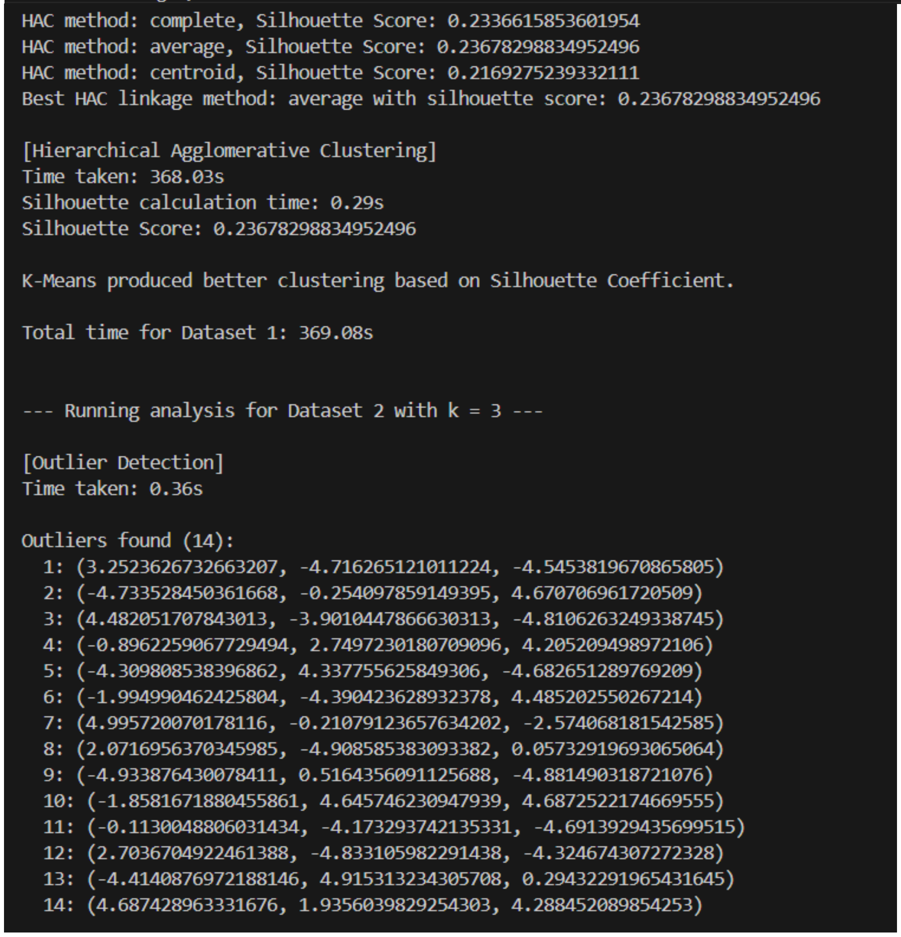
6. Running the Program

To run the clustering program, simply execute the Python script in any Python 3.x environment. When the script starts, it prompts the user to enter the desired number of clusters (k). If the input is invalid or left blank, the program defaults to k = 3. After receiving the input, the script automatically generates three 3D datasets, each containing 500 points with varying spread values (10, 5, and 15) to represent different clustering scenarios. For each dataset, the program performs outlier detection using the k-nearest neighbor distance method, prints all detected outliers, and removes them to form a cleaned dataset. Both K-Means and Hierarchical Agglomerative Clustering (HAC) are then applied to the cleaned data. HAC runs four times per dataset, once for each linkage method (single, complete, average, centroid), and selects the best one based on the Silhouette Coefficient. After clustering, the program prints the silhouette scores for both algorithms, identifies the better one, and logs the execution time for each major step including outlier detection, clustering, and silhouette calculation. This helps the user understand the computational cost and effectiveness of each algorithm across different datasets. All results are printed directly to the console in a well-structured format.

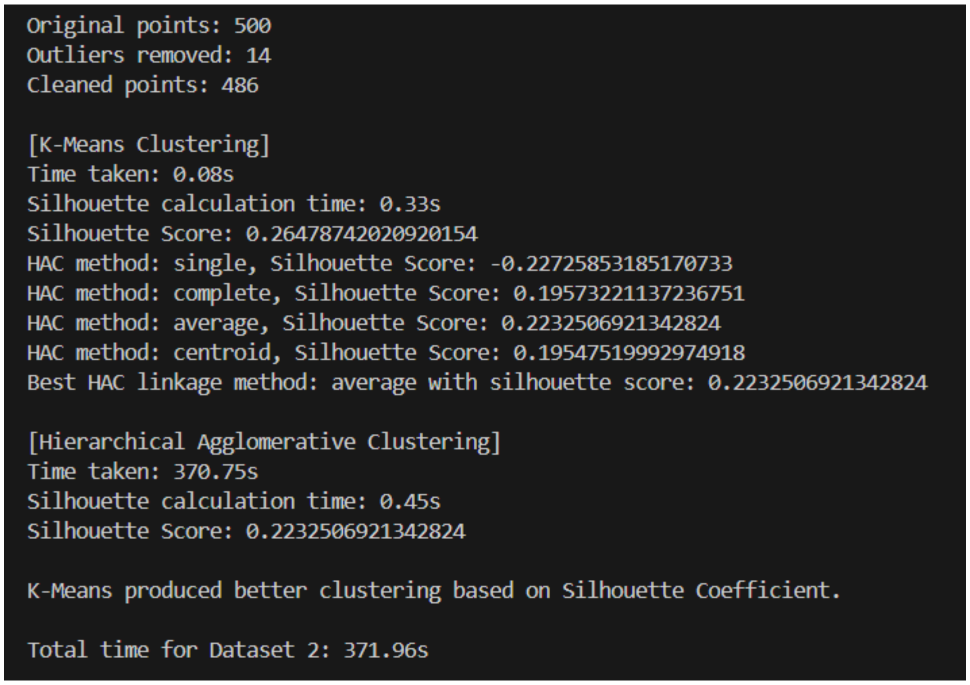
7. Screenshots of Program Execution



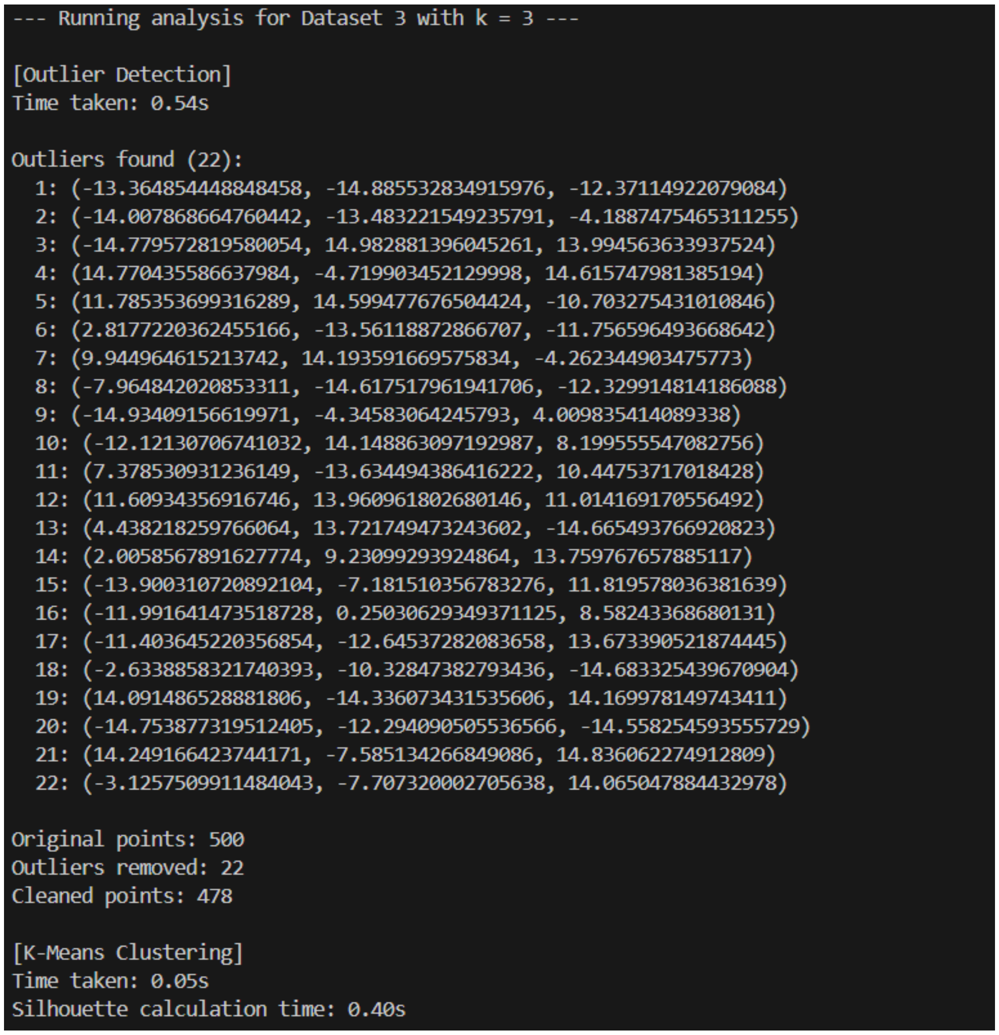
```
PS C:\Users\aditn\OneDrive\Desktop\CS634> & C:/Users/aditn/AppData/Local/Microsoft/WindowsApp
roject2.py
Enter the number of clusters (k): 3

--- Running analysis for Dataset 1 with k = 3 ---

[Outlier Detection]
Time taken: 0.37s

Outliers found (17):
  1: (9.985649368254531, 6.720551701599039, 9.379925145695026)
  2: (-1.5398502801967417, 9.146352817193463, 9.908453789854278)
  3: (9.744661272630086, 3.079526354214652, -9.843537856956841)
  4: (-9.806606007833201, -8.495122798524658, 7.662127866002859)
  5: (-9.52671130597097, -6.137404233445826, -3.4347609760458697)
  6: (9.408012220444558, -6.4286436765507275, 9.25068631523111)
  7: (-4.905549877228362, 4.175705676683412, -9.96617443562741)
  8: (-3.993554635774716, -3.8143067558269355, -1.8321418276156631)
  9: (1.503982184398609, 9.015725556127048, 9.991448337549027)
  10: (9.853840872537951, -4.095384922334613, 9.558889691537253)
  11: (9.59884485563807, 9.345394946000646, 6.091752880411239)
  12: (-9.943288970399673, 7.168122527603526, -7.1062403935898395)
  13: (3.2211528519463357, -9.484397004276033, -9.702793455385823)
  14: (-7.1625069169746265, 2.663440253110277, -9.386858032381882)
  15: (8.760043991315214, -3.808362251680782, -2.466413878873997)
  16: (-8.64641354182308, 0.5211889064434096, 0.14655393159573293)
  17: (-2.948813793830354, 9.963011955460981, -4.508892798502848)

Original points: 500
Outliers removed: 17
Cleaned points: 483

[K-Means Clustering]
Time taken: 0.06s
Silhouette calculation time: 0.32s
Silhouette Score: 0.2612428382708448
HAC method: single, Silhouette Score: -0.040940520336390486
```



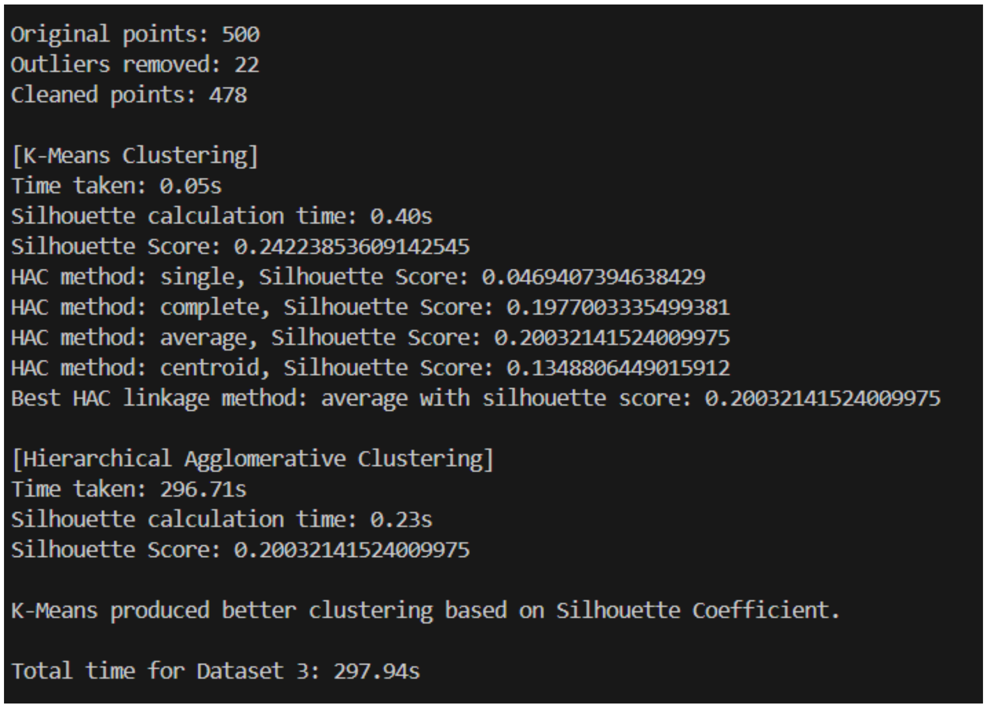
```
HAC method: complete, Silhouette Score: 0.2336615853601954
HAC method: average, Silhouette Score: 0.23678298834952496
HAC method: centroid, Silhouette Score: 0.2169275239332111
Best HAC linkage method: average with silhouette score: 0.23678298834952496

[Hierarchical Agglomerative Clustering]
Time taken: 368.03s
Silhouette calculation time: 0.29s
Silhouette Score: 0.23678298834952496

K-Means produced better clustering based on Silhouette Coefficient.

Total time for Dataset 1: 369.08s


--- Running analysis for Dataset 2 with k = 3 ---

[Outlier Detection]
Time taken: 0.36s

Outliers found (14):
  1: (3.2523626732663207, -4.716265121011224, -4.5453819670865805)
  2: (-4.733528450361668, -0.254097859149395, 4.670706961720509)
  3: (4.482051707843013, -3.9010447866630313, -4.8106263249338745)
  4: (-0.8962259067729494, 2.7497230180709096, 4.205209498972106)
  5: (-4.309808538396862, 4.337755625849306, -4.682651289769209)
  6: (-1.994990462425804, -4.390423628932378, 4.485202550267214)
  7: (4.995720070178116, -0.21079123657634202, -2.574068181542585)
  8: (2.0716956370345985, -4.908585383093382, 0.05732919693065064)
  9: (-4.933876430078411, 0.5164356091125688, -4.881490318721076)
  10: (-1.8581671880455861, 4.645746230947939, 4.6872522174669555)
  11: (-0.1130048806031434, -4.173293742135331, -4.6913929435699515)
  12: (2.7036704922461388, -4.833105982291438, -4.324674307272328)
  13: (-4.4140876972188146, 4.915313234305708, 0.29432291965431645)
  14: (4.687428963331676, 1.9356039829254303, 4.288452089854253)
```



```
Original points: 500
Outliers removed: 14
Cleaned points: 486

[K-Means Clustering]
Time taken: 0.08s
Silhouette calculation time: 0.33s
Silhouette Score: 0.26478742020920154
HAC method: single, Silhouette Score: -0.22725853185170733
HAC method: complete, Silhouette Score: 0.19573221137236751
HAC method: average, Silhouette Score: 0.2232506921342824
HAC method: centroid, Silhouette Score: 0.19547519992974918
Best HAC linkage method: average with silhouette score: 0.2232506921342824

[Hierarchical Agglomerative Clustering]
Time taken: 370.75s
Silhouette calculation time: 0.45s
Silhouette Score: 0.2232506921342824

K-Means produced better clustering based on Silhouette Coefficient.

Total time for Dataset 2: 371.96s
```

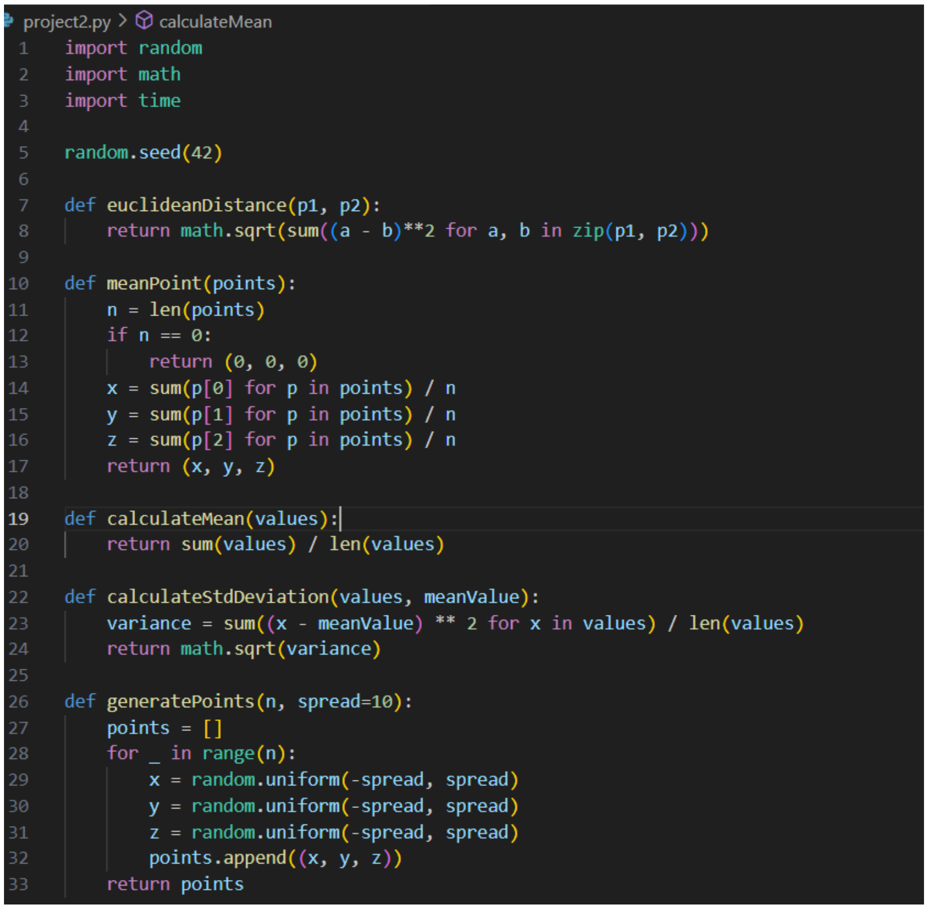


```
--- Running analysis for Dataset 3 with k = 3 ---

[Outlier Detection]
Time taken: 0.54s

Outliers found (22):
  1: (-13.364854448848458, -14.885532834915976, -12.37114922079084)
  2: (-14.007868664760442, -13.483221549235791, -4.1887475465311255)
  3: (-14.779572819580054, 14.982881396045261, 13.994563633937524)
  4: (14.770435586637984, -4.719903452129998, 14.615747981385194)
  5: (11.785353699316289, 14.599477676504424, -10.703275431010846)
  6: (2.8177220362455166, -13.56118872866707, -11.756596493668642)
  7: (9.944964615213742, 14.193591669575834, -4.262344903475773)
  8: (-7.964842020853311, -14.617517961941706, -12.329914814186088)
  9: (-14.93409156619971, -4.34583064245793, 4.009835414089338)
  10: (-12.12130706741032, 14.148863097192987, 8.199555547082756)
  11: (7.378530931236149, -13.634494386416222, 10.44753717018428)
  12: (11.60934356916746, 13.960961802680146, 11.014169170556492)
  13: (4.438218259766064, 13.721749473243602, -14.665493766920823)
  14: (2.0058567891627774, 9.23099293924864, 13.759767657885117)
  15: (-13.900310720892104, -7.181510356783276, 11.819578036381639)
  16: (-11.991641473518728, 0.25030629349371125, 8.58243368680131)
  17: (-11.403645220356854, -12.64537282083658, 13.673390521874445)
  18: (-2.6338858321740393, -10.32847382793436, -14.683325439670904)
  19: (14.091486528881806, -14.336073431535606, 14.169978149743411)
  20: (-14.753877319512405, -12.294090505536566, -14.558254593555729)
  21: (14.249166423744171, -7.585134266849086, 14.836062274912809)
  22: (-3.1257509911484043, -7.707320002705638, 14.065047884432978)

Original points: 500
Outliers removed: 22
Cleaned points: 478

[K-Means Clustering]
Time taken: 0.05s
Silhouette calculation time: 0.40s
```



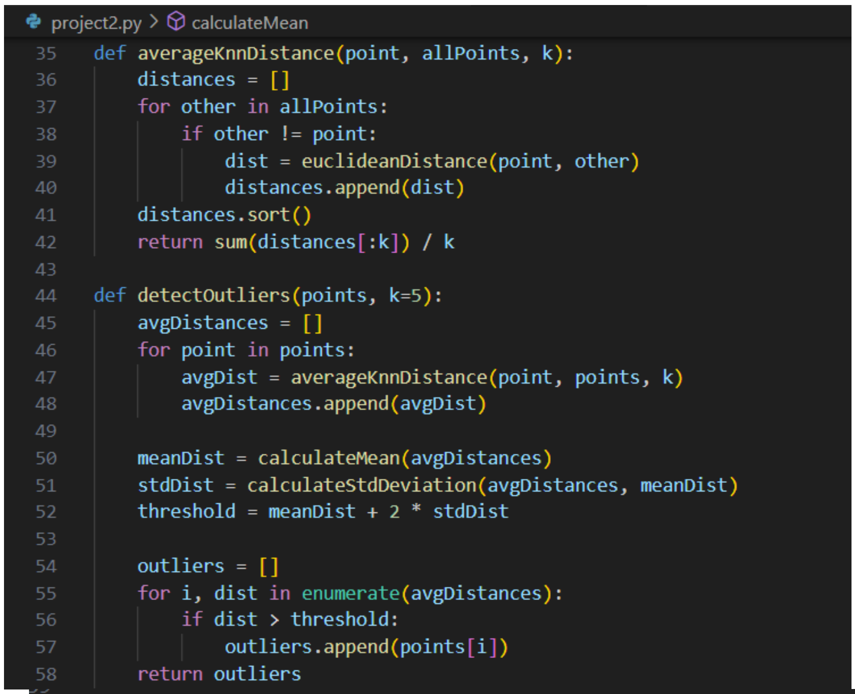
```
Original points: 500
Outliers removed: 22
Cleaned points: 478

[K-Means Clustering]
Time taken: 0.05s
Silhouette calculation time: 0.40s
Silhouette Score: 0.24223853609142545
HAC method: single, Silhouette Score: 0.0469407394638429
HAC method: complete, Silhouette Score: 0.1977003335499381
HAC method: average, Silhouette Score: 0.20032141524009975
HAC method: centroid, Silhouette Score: 0.1348806449015912
Best HAC linkage method: average with silhouette score: 0.20032141524009975

[Hierarchical Agglomerative Clustering]
Time taken: 296.71s
Silhouette calculation time: 0.23s
Silhouette Score: 0.20032141524009975

K-Means produced better clustering based on Silhouette Coefficient.

Total time for Dataset 3: 297.94s
```

8. Source Code

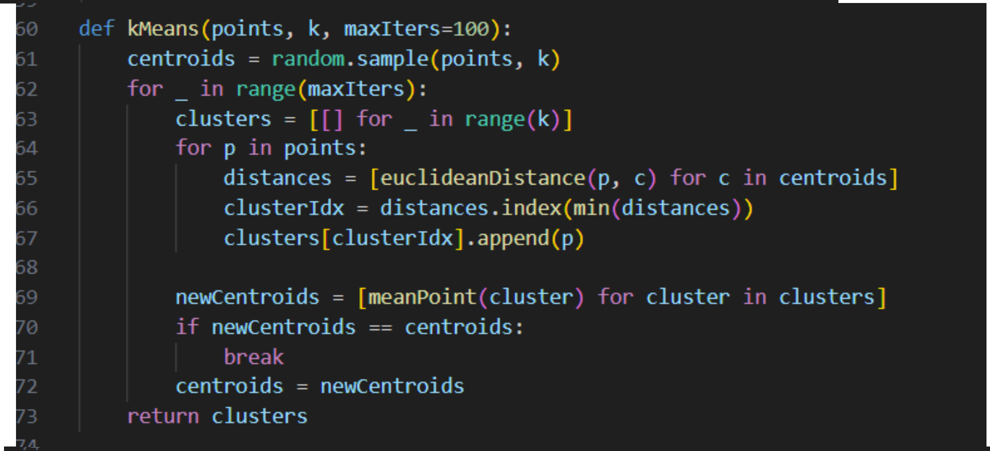


project2.py > calculateMean

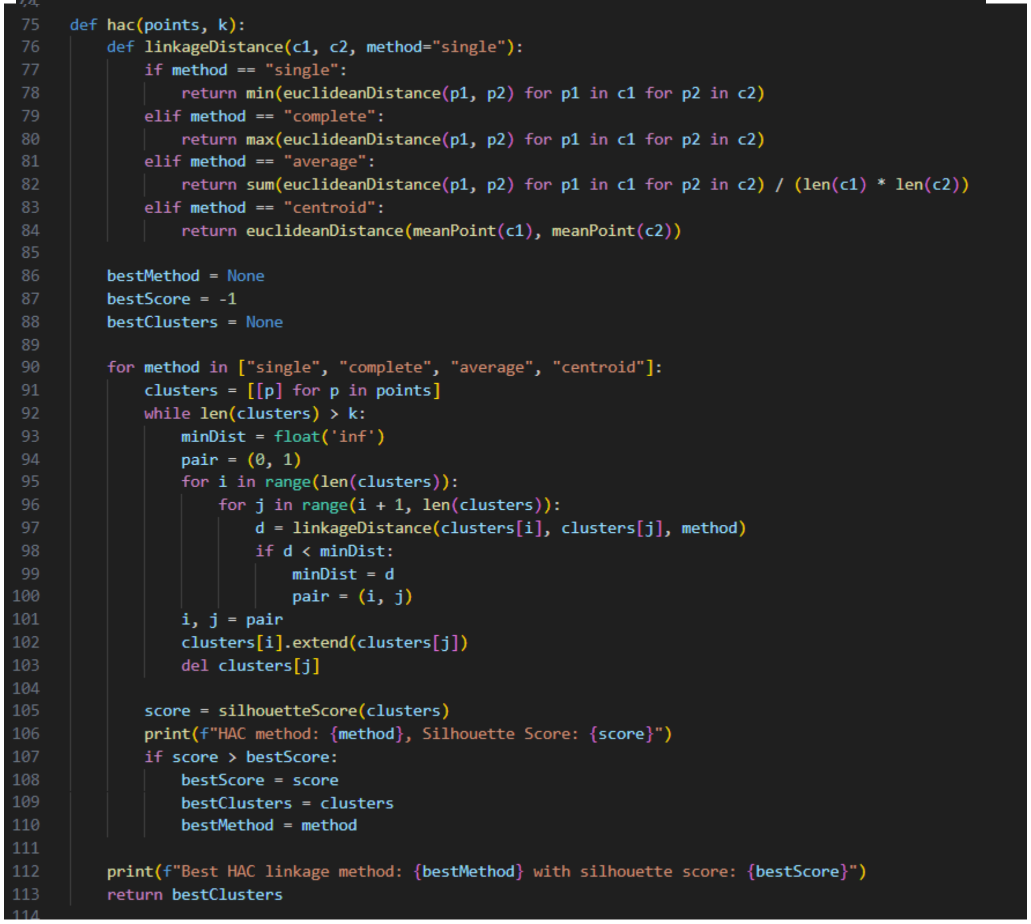
```python
import random
import math
import time

random.seed(42)

def euclideanDistance(p1, p2):
    return math.sqrt(sum((a - b)**2 for a, b in zip(p1, p2)))

def meanPoint(points):
    n = len(points)
    if n == 0:
        return (0, 0, 0)
    x = sum(p[0] for p in points) / n
    y = sum(p[1] for p in points) / n
    z = sum(p[2] for p in points) / n
    return (x, y, z)

def calculateMean(values):
    return sum(values) / len(values)

def calculateStdDeviation(values, meanValue):
    variance = sum((x - meanValue) ** 2 for x in values) / len(values)
    return math.sqrt(variance)

def generatePoints(n, spread=10):
    points = []
    for _ in range(n):
        x = random.uniform(-spread, spread)
        y = random.uniform(-spread, spread)
        z = random.uniform(-spread, spread)
        points.append((x, y, z))
    return points
```



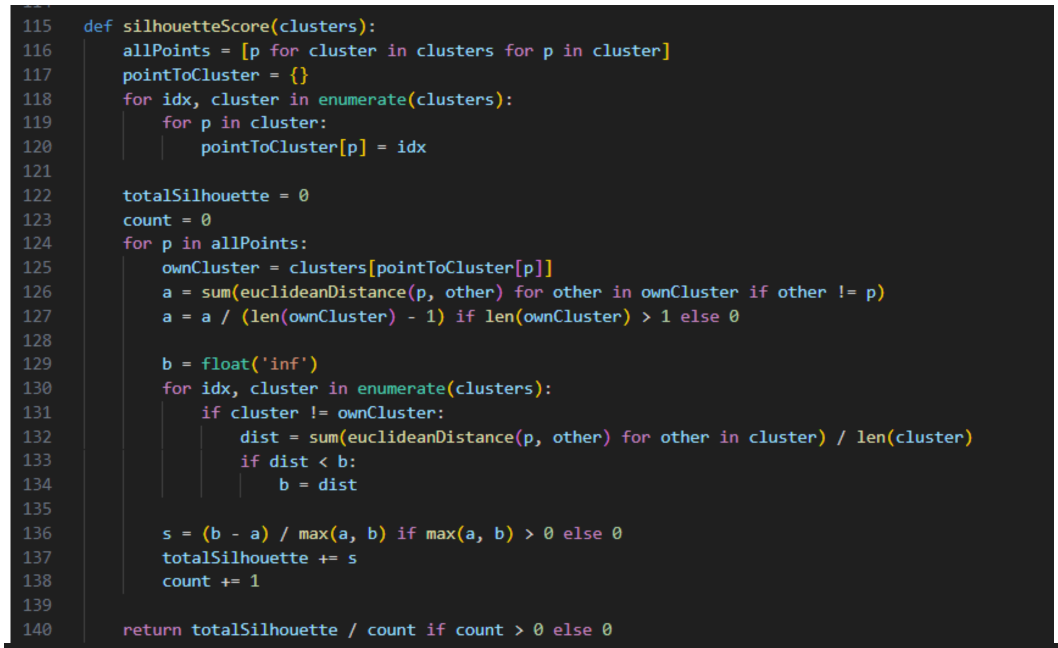
```python
def averageKnnDistance(point, allPoints, k):
    distances = []
    for other in allPoints:
        if other != point:
            dist = euclideanDistance(point, other)
            distances.append(dist)
    distances.sort()
    return sum(distances[:k]) / k

def detectOutliers(points, k=5):
    avgDistances = []
    for point in points:
        avgDist = averageKnnDistance(point, points, k)
        avgDistances.append(avgDist)

    meanDist = calculateMean(avgDistances)
    stdDist = calculateStdDeviation(avgDistances, meanDist)
    threshold = meanDist + 2 * stdDist

    outliers = []
    for i, dist in enumerate(avgDistances):
        if dist > threshold:
            outliers.append(points[i])
    return outliers
```



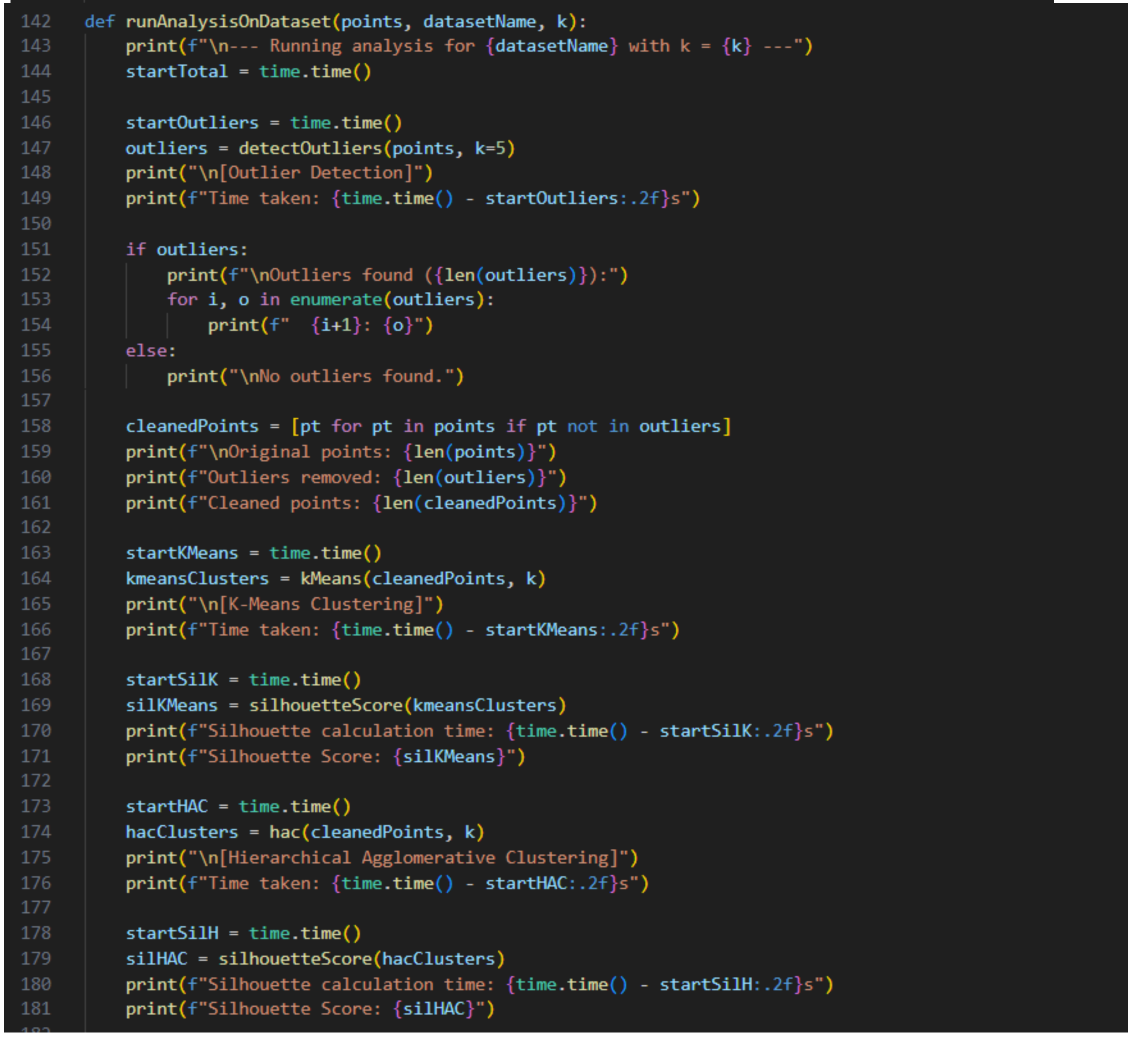
```python
def kMeans(points, k, maxIters=100):
    centroids = random.sample(points, k)
    for _ in range(maxIters):
        clusters = [[] for _ in range(k)]
        for p in points:
            distances = [euclideanDistance(p, c) for c in centroids]
            clusterIdx = distances.index(min(distances))
            clusters[clusterIdx].append(p)

        newCentroids = [meanPoint(cluster) for cluster in clusters]
        if newCentroids == centroids:
            break
        centroids = newCentroids
    return clusters
```



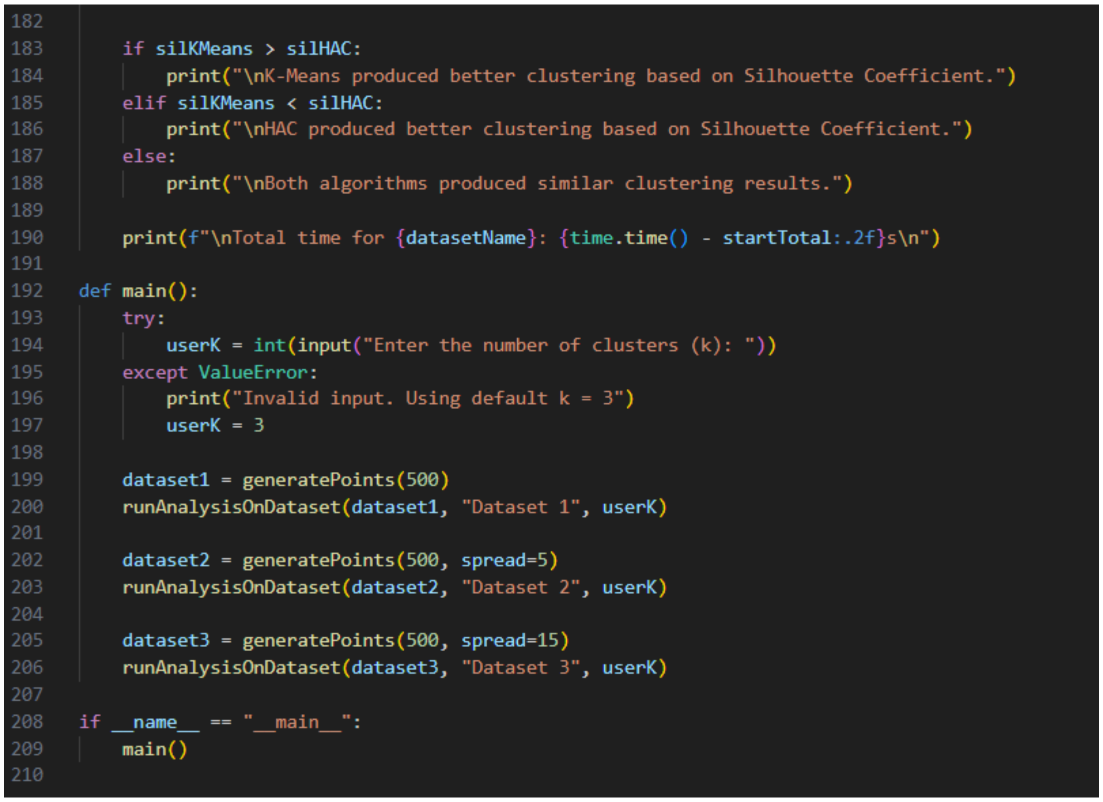
```python
def hac(points, k):
    def linkageDistance(c1, c2, method="single"):
        if method == "single":
            return min(euclideanDistance(p1, p2) for p1 in c1 for p2 in c2)
        elif method == "complete":
            return max(euclideanDistance(p1, p2) for p1 in c1 for p2 in c2)
        elif method == "average":
            return sum(euclideanDistance(p1, p2) for p1 in c1 for p2 in c2) / (len(c1) * len(c2))
        elif method == "centroid":
            return euclideanDistance(meanPoint(c1), meanPoint(c2))

    bestMethod = None
    bestScore = -1
    bestClusters = None

    for method in ["single", "complete", "average", "centroid"]:
        clusters = [[p] for p in points]
        while len(clusters) > k:
            minDist = float('inf')
            pair = (0, 1)
            for i in range(len(clusters)):
                for j in range(i + 1, len(clusters)):
                    d = linkageDistance(clusters[i], clusters[j], method)
                    if d < minDist:
                        minDist = d
                        pair = (i, j)
            i, j = pair
            clusters[i].extend(clusters[j])
            del clusters[j]

        score = silhouetteScore(clusters)
        print(f"HAC method: {method}, Silhouette Score: {score}")
        if score > bestScore:
            bestScore = score
            bestClusters = clusters
            bestMethod = method

    print(f"Best HAC linkage method: {bestMethod} with silhouette score: {bestScore}")
    return bestClusters
```



```python
def silhouetteScore(clusters):
    allPoints = [p for cluster in clusters for p in cluster]
    pointToCluster = {}
    for idx, cluster in enumerate(clusters):
        for p in cluster:
            pointToCluster[p] = idx

    totalSilhouette = 0
    count = 0
    for p in allPoints:
        ownCluster = clusters[pointToCluster[p]]
        a = sum(euclideanDistance(p, other) for other in ownCluster if other != p)
        a = a / (len(ownCluster) - 1) if len(ownCluster) > 1 else 0

        b = float('inf')
        for idx, cluster in enumerate(clusters):
            if cluster != ownCluster:
                dist = sum(euclideanDistance(p, other) for other in cluster) / len(cluster)
                if dist < b:
                    b = dist

        s = (b - a) / max(a, b) if max(a, b) > 0 else 0
        totalSilhouette += s
        count += 1

    return totalSilhouette / count if count > 0 else 0
```



```python
def runAnalysisOnDataset(points, datasetName, k):
    print(f"\n--- Running analysis for {datasetName} with k = {k} ---")
    startTotal = time.time()

    startOutliers = time.time()
    outliers = detectOutliers(points, k=5)
    print("\n[Outlier Detection]")
    print(f"Time taken: {time.time() - startOutliers:.2f}s")

    if outliers:
        print(f"\nOutliers found ({len(outliers)}):")
        for i, o in enumerate(outliers):
            print(f"  {i+1}: {o}")
    else:
        print("\nNo outliers found.")

    cleanedPoints = [pt for pt in points if pt not in outliers]
    print(f"\nOriginal points: {len(points)}")
    print(f"Outliers removed: {len(outliers)}")
    print(f"Cleaned points: {len(cleanedPoints)}")

    startKMeans = time.time()
    kmeansClusters = kMeans(cleanedPoints, k)
    print("\n[K-Means Clustering]")
    print(f"Time taken: {time.time() - startKMeans:.2f}s")

    startSilK = time.time()
    silKMeans = silhouetteScore(kmeansClusters)
    print(f"Silhouette calculation time: {time.time() - startSilK:.2f}s")
    print(f"Silhouette Score: {silKMeans}")

    startHAC = time.time()
    hacClusters = hac(cleanedPoints, k)
    print("\n[Hierarchical Agglomerative Clustering]")
    print(f"Time taken: {time.time() - startHAC:.2f}s")

    startSilH = time.time()
    silHAC = silhouetteScore(hacClusters)
    print(f"Silhouette calculation time: {time.time() - startSilH:.2f}s")
    print(f"Silhouette Score: {silHAC}")
```



```python

    if silKMeans > silHAC:
        print("\nK-Means produced better clustering based on Silhouette Coefficient.")
    elif silKMeans < silHAC:
        print("\nHAC produced better clustering based on Silhouette Coefficient.")
    else:
        print("\nBoth algorithms produced similar clustering results.")

    print(f"\nTotal time for {datasetName}: {time.time() - startTotal:.2f}s\n")

def main():
    try:
        userK = int(input("Enter the number of clusters (k): "))
    except ValueError:
        print("Invalid input. Using default k = 3")
        userK = 3

    dataset1 = generatePoints(500)
    runAnalysisOnDataset(dataset1, "Dataset 1", userK)

    dataset2 = generatePoints(500, spread=5)
    runAnalysisOnDataset(dataset2, "Dataset 2", userK)

    dataset3 = generatePoints(500, spread=15)
    runAnalysisOnDataset(dataset3, "Dataset 3", userK)

if __name__ == "__main__":
    main()

```

9. Conclusion

This project was an excellent opportunity to learn more about the inner workings of clustering. Everything was programmed from scratch, from data generation to outlier detection and grouping, rather than depending on pre-existing tools or libraries. I now have far better knowledge of how K-Means and Hierarchical Clustering work in practice and how their effectiveness varies based on the kind of data they are used on. I was able to determine which algorithm worked best and why by testing with various spreads and using the Silhouette Coefficient to analyze the findings. The results were significantly different after the data was cleaned up by eliminating outliers, highlighting the significance of preprocessing.