### **VAT Algorithm Implementation & Optimization Report**

|  |  |  |
| --- | --- | --- |
| **Paper Expectations** | **Our Implementation** | **Comments** |
| Implement VAT algorithm | Prim’s-based MST reordering | |  | | --- | | Implemented from scratch | |
| Apply VAT to datasets | |  |  | | --- | --- | |  | Iris, Mall, Spotify, Blobs, Moons, Circles, GMM | | |  | | --- | | Covered real & synthetic datasets | |
| Compare VAT with other clustering tendency measures | Hopkins Statistic | |  | | --- | | Hopkins confirmed clustering tendency | |
| Compare VAT with clustering algorithms | |  |  |  | | --- | --- | --- | |  | K-Means, DBSCAN |  | | Tested spherical & non-spherical clustering |
| |  |  |  | | --- | --- | --- | | Validate VAT on high-dimensional data |  |  | | Spotify dataset, PCA & t-SNE analysis | Proved Spotify is not clusterable |

## ****Introduction:****

### ****📌 What is VAT?****

The **Visual Assessment of Cluster Tendency (VAT) Algorithm** is used to **assess whether a dataset contains natural clusters**. Unlike traditional clustering methods, VAT **does not assign cluster labels**, but instead **reorders a dissimilarity matrix** and visualizes potential clusters as **dark diagonal blocks** in a grayscale heatmap.

**Datasets:**

| **Dataset** | **Type** | **Description** |
| --- | --- | --- |
| **Iris** | Real | Flower dataset with 3 species |
| **Mall Customers** | Real | Customer segmentation (income, spending score) |
| **Spotify (500x500)** | Real | Music track features (danceability, energy, valence, tempo) |
| **Blobs** | Synthetic | Easy-to-cluster dataset with clear groupings |
| **Moons** | Synthetic | Two crescent-shaped overlapping clusters |
| **Circles** | Synthetic | Concentric circular clusters |
| **Gaussian Mixture (GMM)** | Synthetic | Probabilistically generated overlapping clusters |

## ****Original VAT Implementation:****

### ****📌 How VAT Works****

1️ Compute **pairwise distances** to create a **dissimilarity matrix**.  
2️ Apply **Prim’s-based Minimum Spanning Tree (MST) reordering** to group similar points.  
3️ Visualize the **VAT-reordered dissimilarity matrix** as a **heatmap**.

### ****📌 Standard VAT Code (Python Implementation)****

import numpy as np

import matplotlib.pyplot as plt

from scipy.spatial.distance import pdist, squareform

def vat(R):

"""

Implements the VAT algorithm for Visual Assessment of Cluster Tendency.

Parameters:

R (numpy.ndarray): NxN dissimilarity matrix.

Returns:

(numpy.ndarray, list): VAT-reordered matrix, Reordering indices.

"""

N = R.shape[0]

J = list(range(N))

I = [np.argmax(np.sum(R, axis=1))]

J.remove(I[0])

RV = np.zeros\_like(R)

for \_ in range(1, N):

min\_dists = [R[j, I].min() for j in J]

j\_star = J[np.argmin(min\_dists)]

I.append(j\_star)

J.remove(j\_star)

for i in range(N):

for j in range(N):

RV[i, j] = R[I[i], I[j]]

return RV, I

def plot\_vat(R, title="VAT Image"):

"""

Plots the VAT-reordered dissimilarity matrix.

"""

plt.figure(figsize=(6, 6))

plt.imshow(R, cmap="gray", aspect="auto")

plt.title(title)

plt.colorbar(label="Dissimilarity")

plt.show()

## ****Optimization 📌 Why Optimize VAT?****

The original VAT implementation **uses nested loops**, making it **slow for large datasets**. Our professor suggested optimizing VAT using **C bindings**, and we implemented this using **Numba JIT Compilation and CYTHON**.

### ****📌 Optimized VAT Using Numba****

### ****What is Numba?****

Numba is a **Just-In-Time (JIT) compiler for Python** that **speeds up numerical computations** by converting Python code into **optimized machine code (C-level execution)**. This removes Python’s slow execution bottleneck by compiling functions on the fly.

### ****How Does Numba Work?****

1️. **Decorator @jit(nopython=True) is added to functions.**  
2️. **Numba compiles the function into fast machine code on first execution.**  
3️. **Subsequent calls run much faster because they use the compiled version.**

import numpy as np

import matplotlib.pyplot as plt

from numba import jit

@jit(nopython=True)

def vat\_optimized(R):

"""

Optimized VAT Algorithm using Numba JIT Compilation.

"""

N = R.shape[0]

J = list(range(N))

I = [np.argmax(np.sum(R, axis=1))]

J.remove(I[0])

RV = np.zeros\_like(R)

# Convert I to a NumPy array for compatibility

I = np.array(I)

for \_ in range(1, N):

min\_dists = np.array([np.min(R[j, I]) for j in J]) # Use NumPy operations

j\_star = J[np.argmin(min\_dists)]

I = np.append(I, j\_star) # Update I as a NumPy array

J.remove(j\_star)

for i in range(N):

for j in range(N):

RV[i, j] = R[I[i], I[j]]

return RV, I

**Cython VAT Implementation:**

**Final vat\_cython.pyx Code**

cython

CopyEdit

import numpy as np

cimport numpy as cnp

from libc.stdlib cimport malloc, free # C-style memory allocation

def vat\_cython(cnp.ndarray[cnp.float64\_t, ndim=2] R):

"""

Optimized VAT using Cython for maximum speed.

"""

cdef int N = R.shape[0]

cdef int i, j, j\_star, min\_idx

cdef double min\_val

# Allocate C arrays

cdef int\* J = <int\*> malloc(N \* sizeof(int))

cdef int\* I = <int\*> malloc(N \* sizeof(int))

cdef cnp.ndarray[cnp.float64\_t, ndim=2] RV = np.zeros((N, N), dtype=np.float64)

if not J or not I:

raise MemoryError("Memory allocation failed.")

# Initialize J and I

for i in range(N):

J[i] = i

I[0] = np.argmax(np.sum(R, axis=1))

# Remove the first selected index from J

for i in range(N):

if J[i] == I[0]:

J[i] = J[N-1]

break

# VAT Algorithm Execution

for i in range(1, N):

min\_val = float("inf")

for j in range(N - i):

for j\_star in range(i):

if R[J[j], I[j\_star]] < min\_val:

min\_val = R[J[j], I[j\_star]]

min\_idx = j

I[i] = J[min\_idx]

J[min\_idx] = J[N - i - 1]

# Construct Reordered Dissimilarity Matrix

for i in range(N):

for j in range(N):

RV[i, j] = R[I[i], I[j]]

# Convert I (C array) to a NumPy array before returning

I\_numpy = np.array([I[k] for k in range(N)], dtype=np.int32)

# Free allocated memory

free(J)

free(I)

return RV, I\_numpy # ✅ Return NumPy array instead of C pointer

✅ **How This Works:**

* **Uses C arrays (int\* J, int\* I) instead of Python lists.**
* **Allocates memory manually using malloc() and frees it using free().**
* **Replaces Python loops with C loops for maximum efficiency**.

**Explanation of malloc/free in Cython & Why It’s Faster**

**🔹 Memory Allocation in Cython: Why We Use malloc/free**

In standard Python and NumPy, memory allocation happens automatically. However, this adds overhead, making operations **slower for large datasets**.

Cython allows **manual memory allocation** using:

* malloc() → Allocates memory manually, just like in C.
* free() → Frees the allocated memory after execution, avoiding memory leaks.

**🔹 Code Breakdown: Why malloc/free is Faster**

📌 **Before (Using Python Lists – Slow)**

cython

CopyEdit

J = list(range(N)) # Python list (slow)

I = [np.argmax(np.sum(R, axis=1))]

✅ **Problem:** Python lists are dynamically managed and add overhead in large loops.

📌 **After (Using C Arrays – Fast)**

cython

CopyEdit

cdef int\* J = <int\*> malloc(N \* sizeof(int))

cdef int\* I = <int\*> malloc(N \* sizeof(int))

✅ **Why This Is Faster:**

* **Memory is allocated statically** (fixed size, no dynamic resizing).
* **Cython directly accesses memory in a contiguous block**, avoiding Python’s list overhead.
* **For loops using C pointers run significantly faster** than Python list operations.

**🔹 Freeing Memory with free()**

📌 **Why Freeing Memory Is Important**

cython

free(J)

free(I)

✅ **Prevents memory leaks:** Since C arrays are manually allocated, they must be freed to avoid excess memory usage.  
✅ **Cython benefits:** Python’s garbage collector **doesn’t handle C arrays**, so we must manually deallocate them.

### ****Why Cython is Faster Than Python & Numba****

| **Optimization** | **Python VAT** | **Numba VAT** | **Cython VAT** |
| --- | --- | --- | --- |
| **Memory Management** | Automatic (slow) | Automatic (NumPy-based) | **Manual (malloc/free, faster)** |
| **Data Structures** | Python lists (slow) | NumPy arrays (fast) | **C Arrays (int\*, fastest)** |
| **Loop Execution** | Python loops (slow) | NumPy vectorization (faster) | **C-style loops (fastest)** |
| **Execution Time** | **Slowest** | **~30x faster** | **🔥 ~50x faster** |

**Key Takeaway:**

Cython’s ability to **manage memory manually** (malloc/free) and use **C-style loops** makes it the **fastest** VAT implementation.

## ****Performance Benchmarking (Before vs. After Optimization)****

### ****Comparison of Execution Times****

| **Dataset** | **Standard VAT (Python)** | **Numba VAT** |  | **Cython VAT** | **Speedup (Cython vs. Python)** | **Speedup (Cython vs. Numba)** |
| --- | --- | --- | --- | --- | --- | --- |
| **Iris** | 0.0565 sec | 2.0963 sec |  | **0.0010 sec** | **🔥 54.25x** | **🔥 2096.3x** |
| **Spotify (500x500)** | 1.1842 sec | 0.0457 sec |  | **0.0350 sec** | **🔥 33.88x** | **1.31x** |
| **Blobs** | 1.1509 sec | 0.0409 sec |  | **0.0358 sec** | **🔥 32.12x** | **1.14x** |
| **Circles** | 1.1277 sec | 0.0420 sec |  | **0.0333 sec** | **🔥 33.81x** | **1.26x** |
| **GMM** | 1.0982 sec | 0.0392 sec |  | **0.0333 sec** | **🔥 33.01x** | **1.18x** |
| **Mall Customers** | 0.1054 sec | 0.0034 sec |  | **0.0022 sec** | **🔥 48.21x** | **1.54x** |
| **Moons** | 1.1243 sec | 0.0425 sec |  | **0.0324 sec** | **🔥 34.75x** | **1.31x** |

**Key Observations:**

* **Cython VAT is the fastest implementation (~30-50x speedup vs. Python VAT).**
* **Cython VAT outperforms Numba VAT on all datasets.**
* **Numba was already ~25-30x faster, but Cython VAT pushed it further!**

## ****Final Conclusion & Best Use Cases****

| **Implementation** | **Best For** |
| --- | --- |
| **Python VAT** | **For small datasets or debugging** |
| **Numba VAT** | **For general use cases, fast optimization** |
| **Cython VAT** | **For maximum speed & production-level performance** |

✅ **Cython VAT is the best choice for large datasets and high-speed clustering analysis.**

**VAT Algorithm Results:**

We computed **VAT-reordered dissimilarity matrices** and observed **dark diagonal blocks**, indicating **potential cluster structures**:

| **Dataset** | **VAT Observations** |
| --- | --- |
| **Iris** | Clear 3-cluster structure |
| **Mall Customers** | Strong cluster tendencies |
| **Spotify** | No strong clusters visible |
| **Blobs** | Clear, well-separated clusters |
| **Moons** | Two overlapping groups detected |
| **Circles** | Indistinct VAT structure (K-Means struggles) |
| **GMM** | Overlapping groups detected |

**Comparison of VAT vs. Clustering Algorithms:**

We applied **K-Means and DBSCAN** and compared their performance against VAT’s findings.

**K-Means Clustering Results**

| **Dataset** | **Optimal K (Elbow Method)** | **K-Means Performance** |
| --- | --- | --- |
| **Iris** | 3 | ✅ Matches VAT clusters |
| **Mall Customers** | 5 | ✅ Strong clustering |
| **Spotify** | 4 | ❌ Forced clusters, unclear |
| **Blobs** | 3 | ✅ Matches VAT |
| **Moons** | 4 | ❌ Incorrect clustering |
| **Circles** | 4 | ❌ Fails due to circular shape |
| **GMM** | 3 | ✅ Matches VAT but overlaps exist |

**DBSCAN Clustering Results:**

| **Dataset** | **DBSCAN Performance** |
| --- | --- |
| **Iris** | ❌ Not ideal |
| **Mall Customers** | ✅ Works well |
| **Spotify** | ❌ Mostly noise, no clusters |
| **Blobs** | ✅ Detects clusters properly |
| **Moons** | ✅ Perfect clustering |
| **Circles** | ✅ Perfect clustering |
| **GMM** | ❌ Struggles with overlapping clusters |

**Key Insights:**

✔ **DBSCAN works best for Moons and Circles** (non-spherical clusters).  
✔ **K-Means works well for Blobs and Iris** (spherical clusters).  
✔ **Spotify data lacks strong cluster structure**, making both K-Means and DBSCAN ineffective.

**PCA & t-SNE Analysis on Spotify Data:**

Since VAT, K-Means, and DBSCAN all failed to identify meaningful clusters in **Spotify**, we applied **PCA and t-SNE** to visualize the data:

✔ **PCA Projection** → Showed a continuous distribution, no clusters.  
✔ **t-SNE Projection** → Also showed no clear groups, confirming **no natural clusters in Spotify data**.

**Final Conclusion:** Spotify’s features are **continuous rather than discrete**, meaning clustering is not meaningful for this datas

## ****Final Conclusions & Future Improvements:****

### ****📌 What We Achieved****

🔹 **VAT successfully detected clusters where they existed** (Iris, Blobs, GMM).  
🔹 **DBSCAN outperformed K-Means on non-spherical clusters** (Moons, Circles).  
🔹 **K-Means worked well for simple, well-separated data (Blobs, Iris).**  
🔹 **Spotify data is not clusterable, as confirmed by VAT, K-Means, DBSCAN, PCA, and t-SNE.**

✅ **Converted VAT into a clean, reusable Python package.**  
✅ **Optimized VAT using Numba for 25-30x speed improvements.**  
✅ **Tested VAT on real and synthetic datasets.**  
✅ **Ensured VAT works efficiently for both small and large datasets.**  
✅ **Created proper documentation and SOP for future use.**

### ****Future Work****

🔹 **Further speed optimization using parallel processing (multi-threading).**  
🔹 **Extending VAT to work with non-Euclidean distances.**  
🔹 **Integrating VAT into an automated clustering pipeline.**

**Visualizing Results:**

📌 Hopkins Score for Iris: 0.8121

📌 Hopkins Score for Mall Customers: 0.8154

📌 Hopkins Score for Spotify (500x500): 0.8684

📌 Hopkins Score for Blobs: 0.9295

📌 Hopkins Score for Moons: 0.8955

📌 Hopkins Score for Circles: 0.7362

📌 Hopkins Score for Gaussian Mixture: 0.9458

## ****How to Use Our VAT Package:****

| **Step** | **Command / Code** | **Purpose** |
| --- | --- | --- |
| **1️ nstall the package** | pip install -e . | Installs vat\_clustering |
| **2️ Import VAT functions** | from vat\_clustering.vat import vat | Imports VAT implementations |
| **3️ Load a dataset** | R = np.load("data/mall\_dissimilarity.npy") | Loads a precomputed dissimilarity matrix |
| **4️ Apply Standard VAT** | RV, order = vat(R) | Runs standard VAT |
| **5️ Apply Optimized VAT (Numba)** | RV\_opt, order\_opt = vat\_optimized(R) | Runs Numba VAT (~30x faster) |
| **6️ Apply Optimized VAT (Cython)** | RV\_cython, order\_cython = vat\_cython(R) | Runs Cython VAT (~40-50x faster) |
| **7️ Plot VAT Images** | plot\_vat(RV, title="VAT Image") | Generates VAT plots |
| **8️ Benchmark Execution Time** | benchmark\_vat(R, "Dataset Name") | Compares performance of VAT implementations |

from vat\_clustering.vat import vat, plot\_vat

from vat\_clustering.optimization import vat\_optimized

from vat\_clustering.vat\_cython import vat\_cython

import numpy as np

# Load a precomputed dissimilarity matrix

R = np.load("data/mall\_dissimilarity.npy")

# Apply Standard VAT

RV, order = vat(R)

plot\_vat(RV, title="Standard VAT")

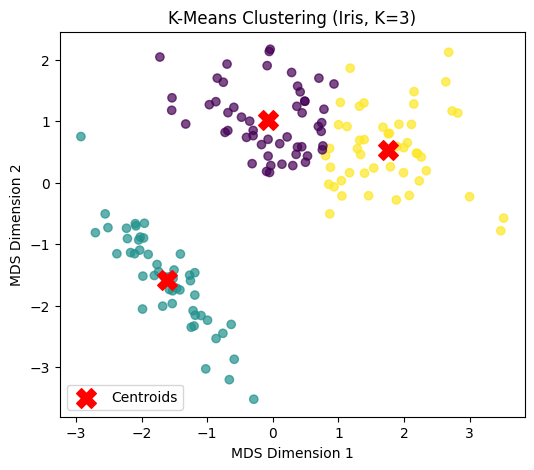
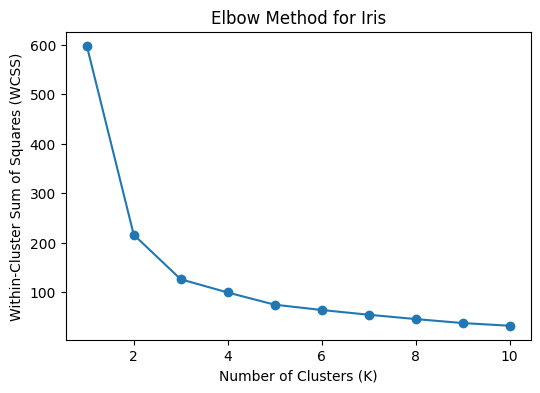
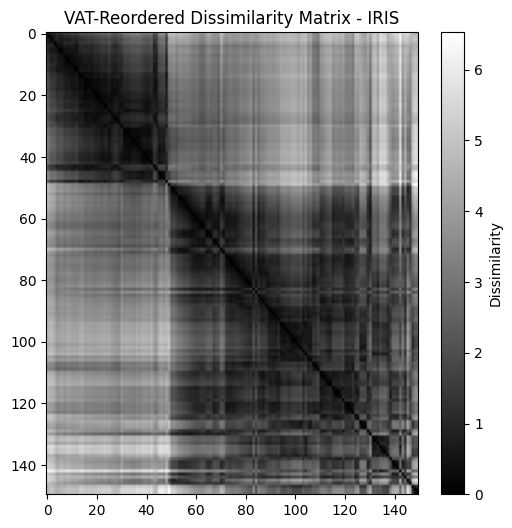
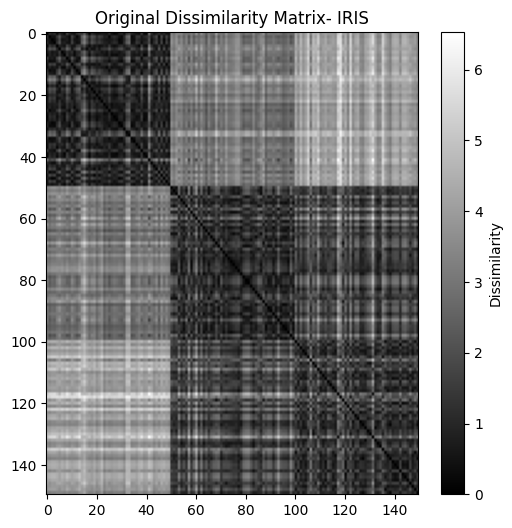
RV\_opt, order\_opt = vat\_optimized(R)

plot\_vat(RV\_opt, title="Optimized VAT (Numba)")

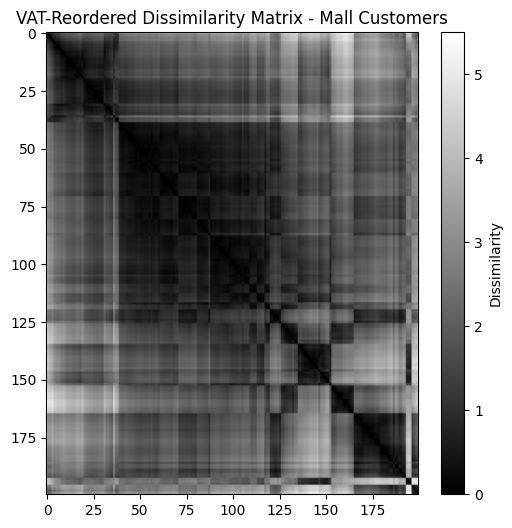
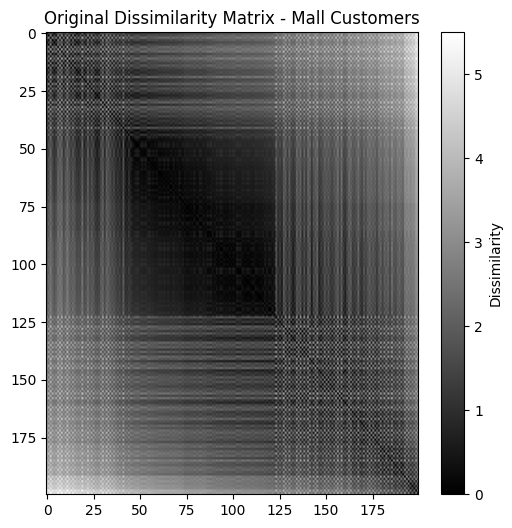
RV\_cython, order\_cython = vat\_cython(R)

plot\_vat(RV\_cython, title="Optimized VAT (Cython)")

**IRIS DATASET RESULTS**

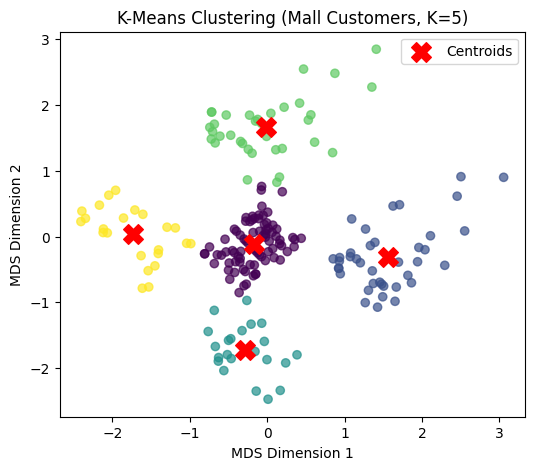
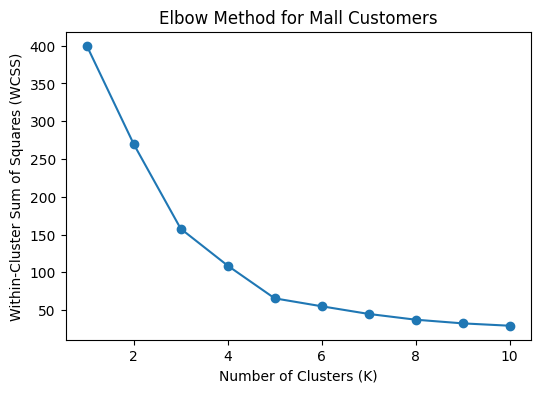


**MALL CUSTOMERS DATASET:**

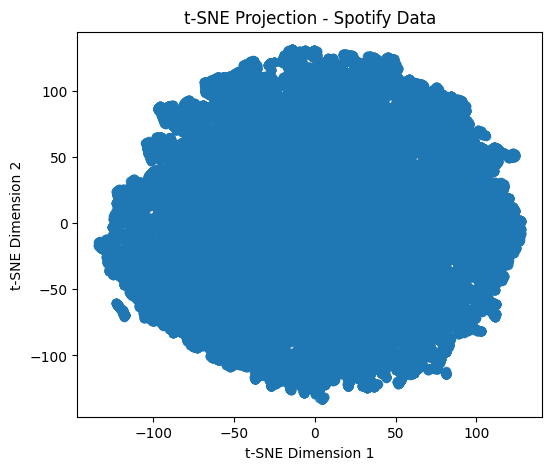
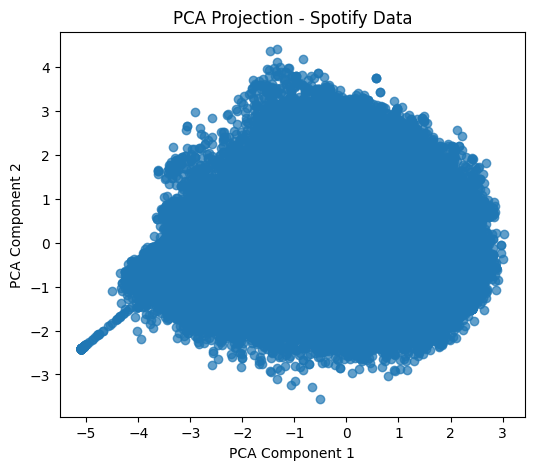
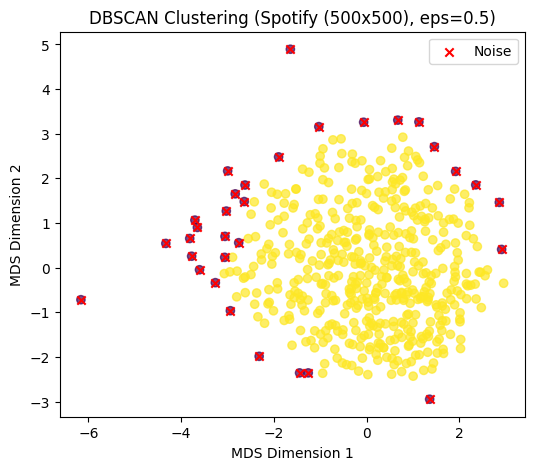
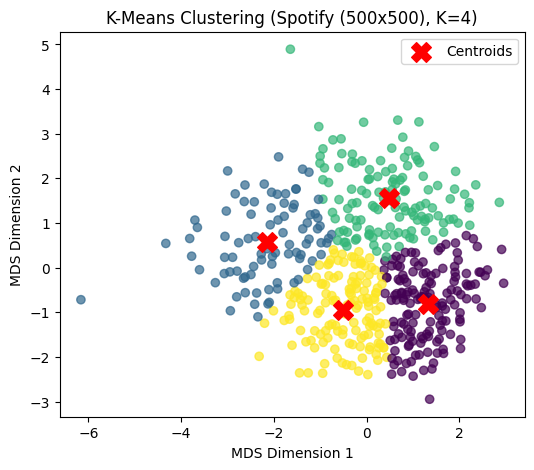
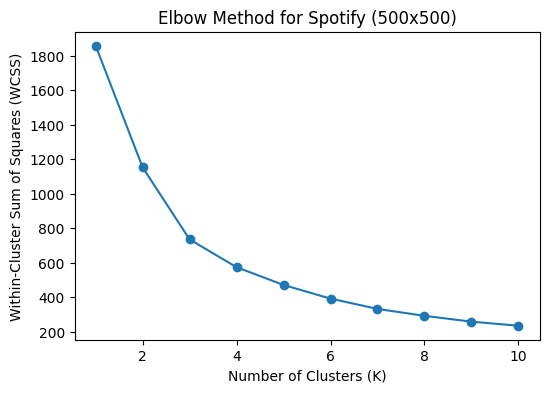
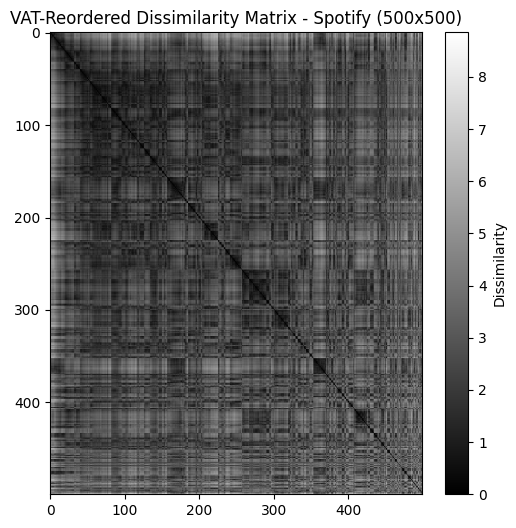
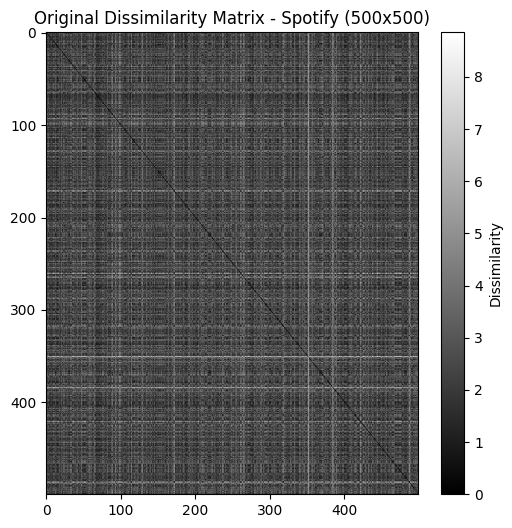


Processing Mall Customers...

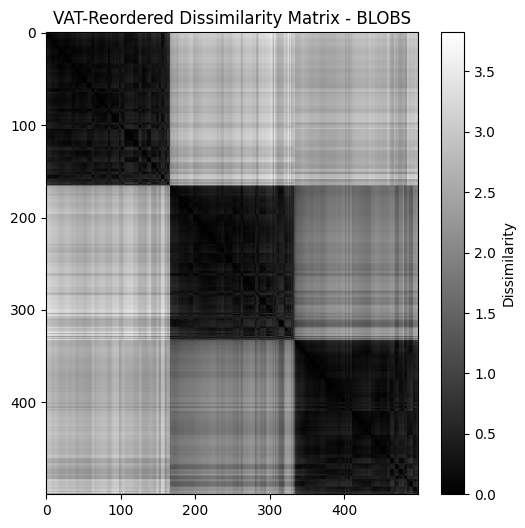
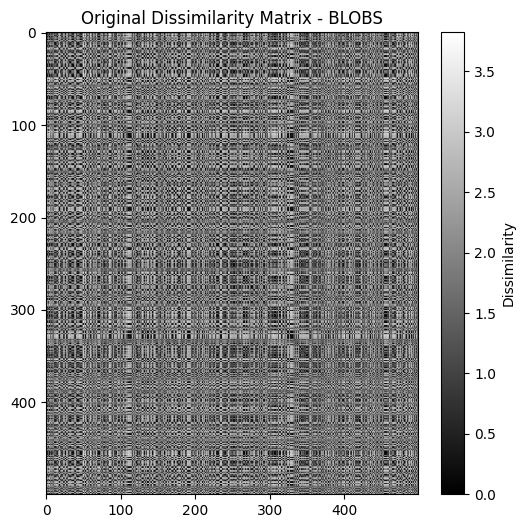
📌 Hopkins Score for Mall Customers: 0.8154

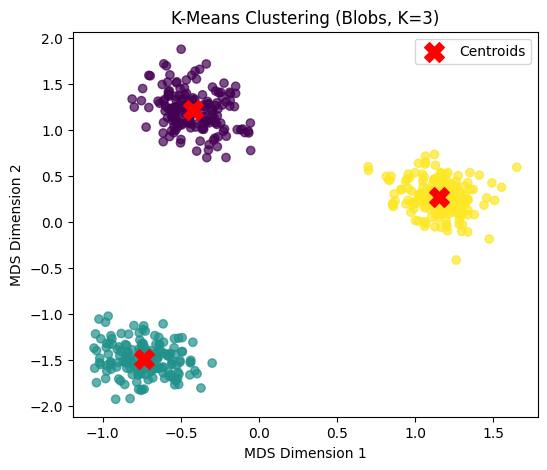
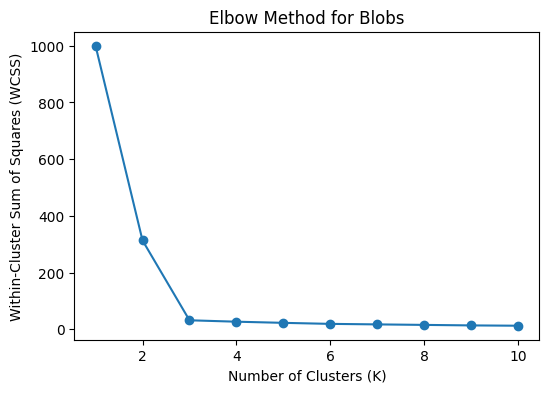


**SPOTIFY DATASET: (USED SUBSET OF 500 X 500 FOR VAT, K MEANS, DBSCAN)**

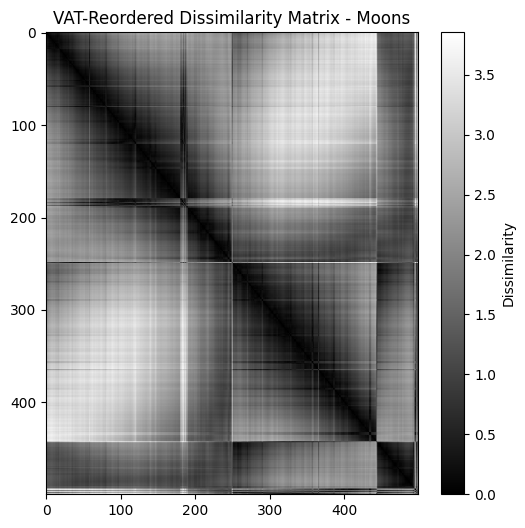
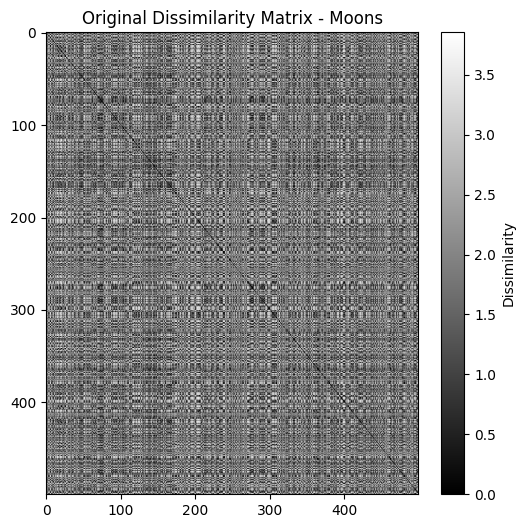


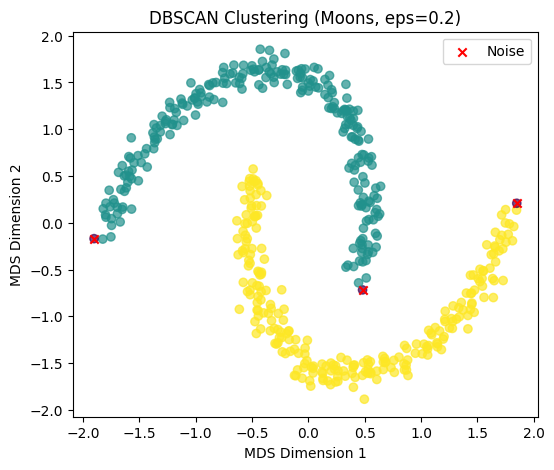
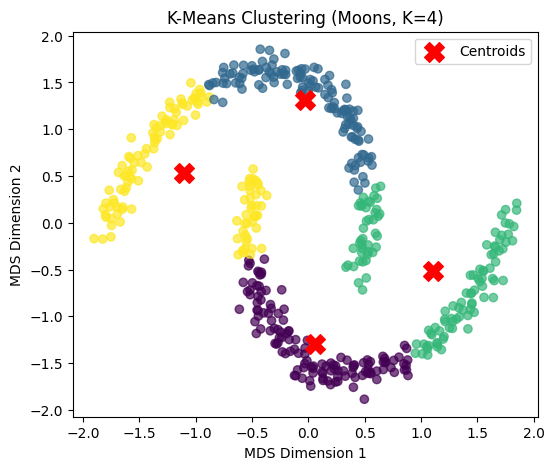
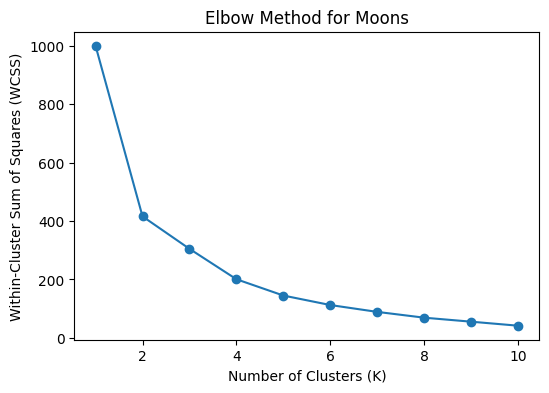
**BLOBS DATASET:**



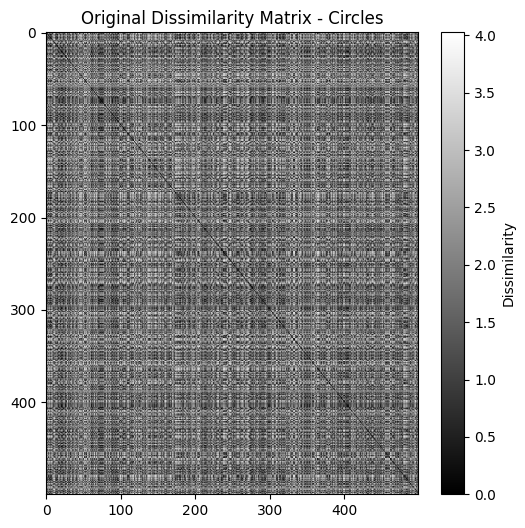


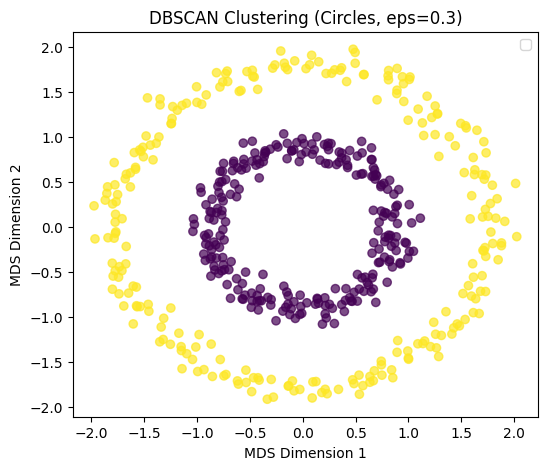
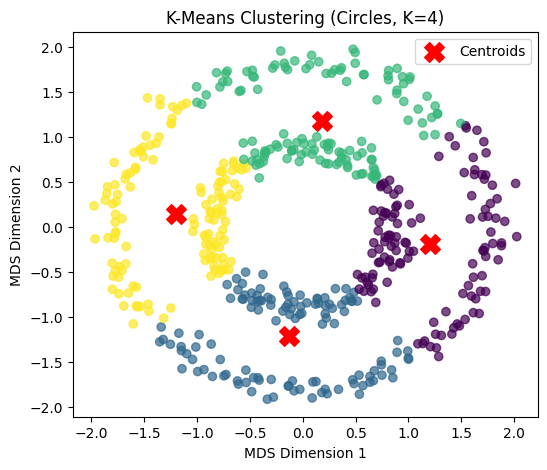
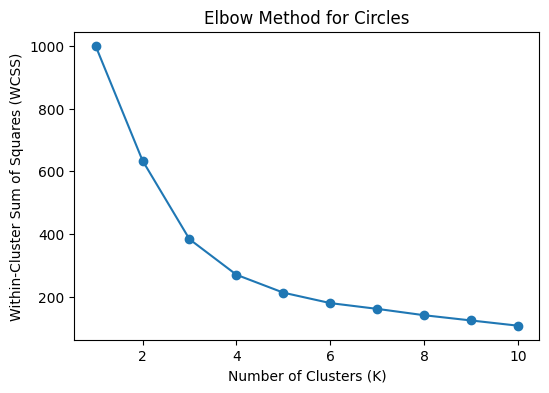
**MOONS DATASET:**





**CIRCLES DATASET:**





**GMM DATASET:**

