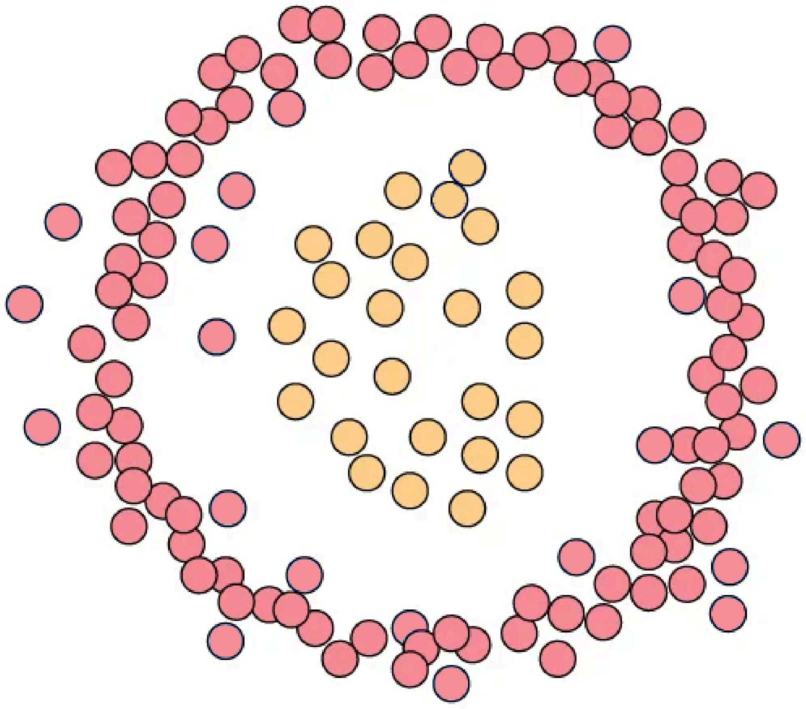
K. N. Toosi
University of Technology



Linear Algebra Applications in Data Clustering: A Python Implementation

Sara Farahani, Tanaz Ghahremani, Mohammad Mirzaee
Directed by Mahdi Lotfi

In this exploration, we exploit fundamental linear algebra principles to implement and validate clustering algorithms, specifically focusing on K-means and Spectral Clustering.

For this project, you are permitted to use Python and the NumPy library. Usage of any machine learning libraries is prohibited, unless we explicitly specify in the task description.

1. Implementation of K-means Algorithm

In this task, your objective is to finalize the `kmeans.py` file, completing the implementation of the K-means clustering algorithm.

```
def k_means_clustering(data, k, max_iterations=100):

    # TODO: Randomly initialize centroids

    # TODO: Iterate until convergence and update centroids

    # TODO: Return labels and centroids
```

Where `data` is a m×n numpy array representing m data points each of dimension n, and `k` denotes the desired number of clusters. There is an optional parameter `max_iterations` that determines the maximum number of iterations. The function is expected to return an m-dimensional vector of labels and a k×n array of centroids representing the cluster centers.

🚨 **Implementation note:** You need to vectorize your code using concepts from the labs. Avoid using loops for vector calculations; only use loops for the outermost iteration of the algorithms, alternating between computing the centroid and assigning points to clusters. Violating this incurs a **50% deduction from the related task score**.

2. Clustering by Spectral Methods

This task involves using spectral clustering to utilize the data's similarity matrix spectrum for dimensionality reduction before clustering by completing the `spectral.py` file.

2.1. Calculate Laplacian matrix

The initial step in spectral clustering involves computing the Laplacian matrix from the affinity (similarity) matrix. We recommend using the symmetric normalized Laplacian, which can be obtained through the following formula:

$$L = I - D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$$

Where:

- I is the identity matrix,
- D is the degree matrix, a diagonal matrix with D_{ii} representing the sum of weights for the i-th row (or column) in the affinity matrix A,
- A is an m×m symmetric affinity matrix representing a pairwise measure of similarity between m data points.

Complete the `laplacian` function to compute the Laplacian of the affinity matrix `A`.

```
def laplacian(A):

    # TODO: Calculate degree matrix

    # TODO: Calculate the inverse square root of the symmetric matrix

    # TODO: Return symmetric normalized Laplacian matrix
```

2.2. Implement Spectral clustering

We will finalize the algorithm implementation by completing the `spectral_clustering` function.

```
def spectral_clustering(affinity, k):

    # TODO: Compute Laplacian matrix

    # TODO: Compute the first k eigenvectors of the Laplacian matrix

    # TODO: Apply K-means clustering on the selected eigenvectors

    # TODO: Return cluster labels
```

Start by computing the graph Laplacian from the affinity matrix. Employ your `laplacian` function for Laplacian calculation.

Form a data representation by concatenating eigenvectors corresponding to the k smallest eigenvalues of graph Laplacian into columns of a matrix. Each row in this matrix serves as a *feature vector* for data points. Then, apply the K-means to the obtained data representation. You may use the `linalg` package from NumPy for eigenvalue decomposition.

It is imperative to utilize your self-implemented `k_means_clustering` algorithm for k-means clustering. If facing challenges in the previous section, consider using the `sklearn.cluster.KMeans`.

K. N. Toosi
University of Technology

3. Evaluate Your Clustering Algorithms!

Now, it's time to compare and test your clustering algorithms. However, before you start, let's implement some prerequisites by writing some stuff in the `validate.py` file.

3.1. Construct the Affinity matrix

Spectral clustering requires an affinity matrix to function. Initially, we will create an affinity matrix from our data by completing the `construct_affinity_matrix` function.
There are several methods available for constructing the affinity matrix, and in this context, you are tasked with implementing the following methods.

- **Gaussian Kernel (RBF Kernel):** Define a similarity measure based on the Gaussian (Radial Basis Function - RBF) kernel:

  $$A_{ij} = exp(-\frac{||x_i - x_j||^2}{2\sigma^2})$$

  where x_i and x_j are data points, $||.||$ denotes the Euclidean distance, and σ is a bandwidth parameter controlling the width of the Gaussian.

- **k-Nearest Neighbors (KNN):** Connect each data point to its k nearest neighbors. The affinity matrix A is binary, with $A_{ij} = 1$ if x_j is among the k nearest neighbors of x_i, and $A_{ij} = 0$ otherwise. Make the matrix symmetric afterward.

```
def construct_affinity_matrix(data, affinity_type, *, k=3, sigma=1.0):

    # TODO: Compute pairwise distances

    if affinity_type == 'knn':

        # TODO: Find k nearest neighbors for each point

        # TODO: Construct symmetric affinity matrix

        # TODO: Return affinity matrix

    elif affinity_type == 'rbf':

        # TODO: Apply RBF kernel
        # TODO: Return affinity matrix
    else:
        raise Exception("invalid affinity matrix type")
```

3.2. Implement evaluation metrics

Congratulations! You are now ready to execute your clustering algorithms. Imagine running these algorithms on the following incredibly straightforward dataset:

Feature vector	**Ground truth label**
p_1	0
p_2	0
p_3	1

What outcomes can be expected from clustering algorithms? Possible results include:

Feature vector	**Predicted label 1**	**Predicted label 2**
p_1	0	1
p_2	0	1
p_3	1	0

Observe that both predicted labels exhibit accurate clustering with 100% precision! Your objective is to finalize the function `clustering_score` in the `metrics.py` file to quantify the accuracy of algorithms, reflecting the ratio of correctly clustered points, while addressing the described issue. You could employ simple techniques like brute-force and greedy algorithms, or implement well-known metrics such as *Normalized Mutual Information (NMI)* or *Adjusted Rand Index (ARI)* to receive **extra credit.**

```
def clustering_score(true_labels, predicted_labels):
    # TODO: Calculate and return clustering score
```

K. N. Toosi
University of Technology

3.3. Compare & Visualize algorithms

Now, let's bring everything together and run the `validate.py` file.

Find three diverse toy datasets in the `datasets` directory. Your objective is to complete the code, creating a 3×4 grid with 12 scatter plots. Rows depict different datasets, and columns showcase the results of each algorithm alongside the ground truth labels. Keep in mind to use different colors for separate clusters in each scatter plot.

```
if __name__ == "__main__":

    datasets = ['blobs', 'circles', 'moons']

    # TODO: Create and configure plot

    for ds_name in datasets:

        dataset = np.load("datasets/%s.npz" % ds_name)

        X = dataset['data']      # feature points

        y = dataset['target']    # ground truth labels

        n = len(np.unique(y))    # number of clusters

        k = 3

        sigma = 1.0

        y_km, _ = k_means_clustering(X, n)

        Arbf = construct_affinity_matrix(X, 'rbf', sigma=sigma)

        y_rbf = spectral_clustering(Arbf, n)

        Aknn = construct_affinity_matrix(X, 'knn', k=k)

        y_knn = spectral_clustering(Aknn, n)

        print("K-means on %s:" % ds_name, clustering_score(y, y_km))

        print("RBF affinity on %s:" % ds_name, clustering_score(y, y_rbf))

        print("KNN affinity on %s:" % ds_name, clustering_score(y, y_knn))

        # TODO: Create subplots

    # TODO: Show subplots
```

Besides visualization, you need to compare the accuracy of the algorithms.

Finally, you should end up with the following accuracy report:

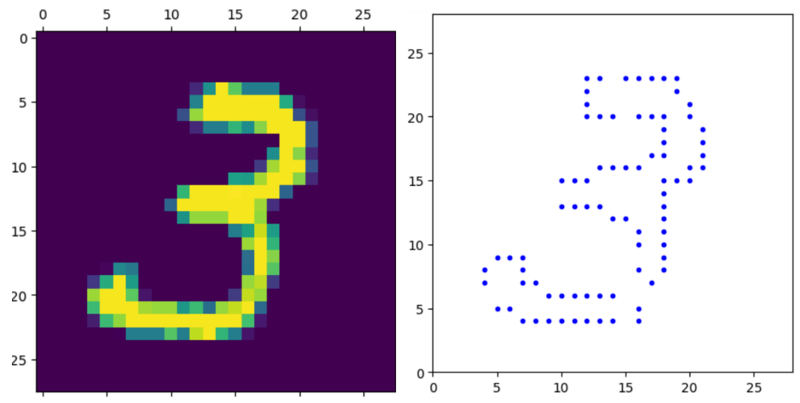
	K-means	RBF + Spectral	KNN + Spectral
Blobs			
Circles			
Moons			

💡 **Note:** Explore different `sigma` and `k` values to find the suitable accuracy. Observe how adjusting these values influences clustering results and why.

4. Cluster Non-Euclidean Data

A point cloud is a set of spatially defined points in space, often used to represent the shape or features of physical objects.

The data set we will work on in this task is derived from the MNIST dataset. Each digit image in the MNIST dataset is transformed into a point cloud by sampling 80 points from its edge pixels.



We intend to employ spectral clustering to classify these digits by doing some challenges in the `mnist.py` file.

Hold on! How could we measure the similarity between two point clouds?

K. N. Toosi
University of Technology

4.1. Implement Chamfer distance

We'll leverage one of the most widely-used methods for calculating dissimilarity between two point clouds: the Chamfer distance. This symmetric metric computes the average distance from each point in one point cloud to the nearest point in the other point cloud:

$$D(A, B) = \frac{1}{|A|} \sum_{a \in A} min_{b \in B} ||a - b||^2 + \frac{1}{|B|} \sum_{b \in B} min_{a \in A} ||a - b||^2$$

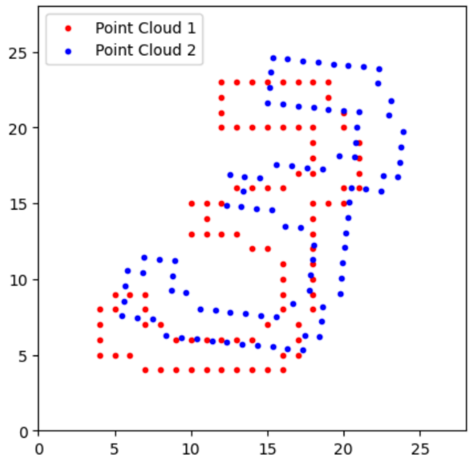
Where:

- A and B are point clouds, m×n matrices representing coordinates of m data points in the n-dimensional space.
- $D(A, B)$ represents the Chamfer distance between point sets A and B.
- a's and b's denote points in sets A and B, respectively.
- The symbol $||a - b||$ represents the Euclidean distance between the pair of points a and b.

```
def chamfer_distance(point_cloud1, point_cloud2):

    # TODO: Calculate distances from each point in point_cloud1 to the
nearest point in point_cloud2

    # TODO: Calculate distances from each point in point_cloud2 to the
nearest point in point_cloud1

    # TODO: Return the Chamfer distance, sum of the average distances in
both directions
```

4.2. Register Point clouds

Consider the following pair of point clouds. Are they similar or not? Is their Chamfer distance large or small? What happens to the distance if we transform and rotate the first or the second point cloud?



The process of finding a spatial transformation to align two point clouds and optimize their distance is called *Registration*.

One of the most popular methods for point set registration is the Iterative Closest Point (ICP) algorithm.

The ICP algorithm iteratively aligns two sets of points in space by establishing point correspondences, calculating a rigid transformation, and applying it to minimize the distance between corresponding points. The process repeats until convergence.

Let's start by finding the rigid transformation between two sets of points.

```
def rigid_transform(A, B):

    # TODO: Subtract centroids to center the point clouds A and B

    # TODO: Construct Cross-Covariance matrix

    # TODO: Apply SVD to the Cross-Covariance matrix

    # TODO: Calculate the rotation matrix

    # TODO: Calculate the translation vector

    # TODO: Return rotation and translation matrices
```

The Cross-Covariance matrix ($H = A^T B$) captures the relationships between the corresponding points in the two sets. You may use the `linalg` package from NumPy for SVD decomposition ($H = U\Sigma V^T$). The rotation matrix could be obtained from the SVD decomposition ($R = VU^T$) and the translation vector is the difference between centroid of point set B and the centroid of the point set A after rotation ($t = \mu_B - R\mu_A$).

Now we can use the rigid transformation function to iteratively adjust the source point cloud with the nearest neighbors in the target point cloud.

```
def icp(source, target, max_iterations=100, tolerance=1e-5):

    # TODO: Iterate until convergence

    # TODO: Find the nearest neighbors of target in the source

    # TODO: Calculate rigid transformation

    # TODO: Apply transformation to source points
    # TODO: Calculate Chamfer distance

    # TODO: Check for convergence

    # TODO: Return the transformed source
```

The algorithm iteratively identifies the nearest neighbors of the target points within the source point set, computes a rigid transformation between the source and the closest points, and then applies this transformation to the source. The iteration continues until the change in error falls below a specified threshold or the maximum number of iterations is reached.

4.3. Construct the Affinity matrix

Now you're ready to construct your affinity matrix using the Chamfer distance. Finish the implementation of the `construct_affinity_matrix` function to generate a symmetric affinity matrix from a set of point clouds.

Keep in mind that the Chamfer distance measures dissimilarity between two point clouds while the concept of affinity matrix is based on pairwise similarity between two data points.

Additionally, it's crucial to register the point clouds before distance measurement to ensure optimal similarity.

```
def construct_affinity_matrix(point_clouds):

    # TODO: Iterate over point clouds to fill affinity matrix

    # TODO: For each pair of point clouds, register them with each other

    # TODO: Calculate symmetric Chamfer distance between registered clouds
```

🚨 **Implementation note:** Your algorithms **must** have computational stability. Correctly handle small distances and consider floating point errors.

4.4. Evaluate & Visualize results

In the last step, we will assemble our functions to cluster hand-written digits.

The provided snippet loads the dataset from the `datasets` directory, constructs an affinity matrix, and then calls the spectral clustering algorithm. The dataset includes only the digits 2, 5 and 9 from the MNIST dataset, and your goal is to cluster these digits.

Conclude your work by reporting the clustering results. Measure the clustering score, and visualize the affinity matrix in 3D using the first 3 eigenvectors. Represent each cluster with a distinct color for clear visualization.

```
if __name__ == "__main__":

    dataset = "mnist"

    dataset = np.load("datasets/%s.npz" % dataset)

    X = dataset['data']      # feature points

    y = dataset['target']    # ground truth labels

    n = len(np.unique(y))    # number of clusters

    Ach = construct_affinity_matrix(X)

    y_pred = spectral_clustering(Ach, n)

    print("Chamfer affinity on %s:" % dataset, clustering_score(y, y_pred))

    # TODO: Plot Ach using its first 3 eigenvectors
```

5. Optional: Speed Up Your Code!

How much time did it take to cluster point clouds in your runtime environment?

Acceleration is a widely recognized strategy to enhance program speed. Numba, an open-source JIT compiler, translates a subset of Python and NumPy code into efficient machine code, contributing to improved performance.

In this task, your goal is to enhance the efficiency of your algorithms in the `speedup.py` file and reevaluate the outcomes of Task 4 using the improved version.

K. N. Toosi
University of Technology

5.1. Rewrite the Algorithms

In this task you should use Numba to improve performance of your algorithms. However it's important to note that Numba might not work well for all source codes. While we recommend using vectorized operations in NumPy, Numba likes loops! Your objective is to optimize the code for both performance and speed.

Rewrite your algorithms and functions to operate with Numba. Replace vectorized code with loops. Avoid using object mode. Try to take advantage of parallelization features of Numba such as fastmath, automatic parallelization and `prange`. Try to run your code on GPU and utilizing CUDA features for further acceleration.

Scoring will be based on the efficiency of your code; higher scores will be awarded for faster execution and the effective utilization of Numba's features.

```
# TODO: Rewrite the k_means_clustering function

# TODO: Rewrite the laplacian function

# TODO: Rewrite the spectral_clustering function

# TODO: Rewrite the clustering_score function

# TODO: Rewrite the chamfer_distance function

# TODO: Rewrite the rigid_transform function

# TODO: Rewrite the icp function

# TODO: Rewrite the construct_affinity_matrix function
```

5.2. Compare the results

Now, test your functions and reevaluate the clustering of the point cloud dataset using the optimized versions with Numba. Finally, provide an evaluation of the effectiveness of Numba and compare the running times between your original algorithms and the newly enhanced versions written with Numba.

```
if __name__ == "__main__":

    dataset = "mnist"

    dataset = np.load("datasets/%s.npz" % dataset)
    X = dataset['data']      # feature points

    y = dataset['target']    # ground truth labels

    n = len(np.unique(y))    # number of clusters

    # TODO: Run both the old and speed up version of your algorithms and
capture running time

    print("Old Chamfer affinity on %s:" % dataset, clustering_score(y,
y_pred_old))

    print("Chamfer affinity on %s:" % dataset, clustering_score(y, y_pred))

    # TODO: Compare the running time using timeit module
```

References & Resources

- K-means clustering
- Spectral clustering
- Point cloud
- Iterative closest point
- Linear Algebra and Learning from Data, by Gilbert Strang, 2019 IV.7.
- Lecture 35 of MIT 18.065, 2018
- Numba