Spectral Clustering: Theory and Application to Relevant Synthetic Examples

Roxana Leal
Department of Economics
Johns Hopkins University
Baltimore, MD 21218
`rlealto1@jh.edu`

Nader Najjar
Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218
`nnajjar2@jh.edu`

Kyle Schneider
Department of Physics and Astronomy
Johns Hopkins University
Baltimore, MD 21218
`kschne23@jh.edu`

Abstract

Spectral clustering is one of the most commonly used statistical learning clustering techniques in modern research. Though spectral clustering can refer to many similar yet separate statistical techniques, this report aims to describe the theoretical statistical foundation, as formulated by the literature, of the most commonly utilized form of the algorithm: unnormalized spectral clustering. This algorithm takes as an input finitely many unlabeled data points and outputs clusters or groups of these points that have similar parameters. From the unlabeled data, we compute the similarity matrix and the Laplacian of this system, and we determine how the eigendecomposition of the Laplacian computes clusters of data for the system. Further, this report uses this theoretical foundation to analyze the efficacy and optimality of a Python implementation of the algorithm, including its space/time complexity. Then, this report applies this algorithm to various interesting and informative synthetic examples, measuring their respective performance with and without presence of noise. In particular, we find the most disparity between the spectral clustering and k-means clustering algorithms when it comes to data shaped as concentric circles and semi-circles.

THE ALGORITHM FOR THIS REPORT CAN BE FOUND IN THE GITHUB REPOSITORY FOUND HERE AND THE GOOGLE COLAB FOUND HERE.

1 Theoretical Foundations and Literature Review

1.1 Introduction

The recent growth of analytical research in the scientific fields has been accompanied by a corresponding rise in the necessity for increasingly advanced statistical methods to aid in the analysis of ever expanding sources of data. To better determine trends in data, numerous statistical techniques have been developed that have been ubiquitously adopted by the scientific community, most of which have strong statistical and probabilistic foundations. The vast amount of these statistical methods and techniques have been largely characterized into four types of statistical learning, where each method is either supervised or unsupervised and either discrete or continuous. As many sources of data in the various fields of scientific research contain an extensive amount of unlabeled data, much focus has been used in determining the most advanced statistical techniques for the statistical methods that fall under unsupervised discrete statistical learning.

Unsupervised statistical learning involves analysis of data sets that are composed of a finite number of unlabeled or raw data points. Naturally, much scientific value can be obtained by determining trends in this unlabeled data, and further value can be gained by determining "groups" in the data, where each member of each group has considerable and noticeable similarities with the other members of their group. Due to this usefulness, statisticians of the past few decades have sought to determine a statistical process that could automatically and simply reduce unlabeled data points to clusters of similar points. As will be discussed in depth further, the use of the eigenvalues, or the spectrum, of matrices akin to the similarity matrix of the data has proved fruitful in this endeavor, which has led to the statistical learning technique referred to as spectral clustering, currently one of the most popular and useful clustering algorithms.

Though modern research relies on the following graph theory and statistics in many applications, it was only a few decades ago when much of the framework for spectral clustering was determined. Centuries after the conception of graph theory as a field of study, [1] expanded upon the idea of a Laplacian for a graph, while [2] furthered this idea to elaborate on spectral graph partitioning, the core concept of spectral clustering. Then, [3] studied how to consider the size of partitions to be nonneglible, which led to the current theory of spectral clustering. Many others expanded on these works to determine separate ways to cluster, how to determine the algorithm in different mathematical interpretations, and how to determine uniqueness of the algorithm (see [6], [9], [7], and [8]). In the next section, we will follow a similar route that these statisticians took in determining an effective algorithm for grouping clusters of large, unlabeled, discrete data sets.

1.2 Statistical and Graph Theoretical Foundation of Spectral Clustering

As unsupervised learning deals with unlabeled data points, it is natural to view the collection of these points as a graph $G(V, E)$ composed of the data points, or vertices, V and edges E that act as the link, or relationship, between any two vertices. For sources of large unlabeled data, one can connect "similar objects" in the data set with different "weights," where each weight determines how "similar" each data point is to another. Though "similarity" is a vague term, it is often up to the user what they may define similarity to be and to determine an appropriate scale for similarity. For example, for data sets consisting of the location of objects, one can define similarity to be the inverse of the Euclidean distance between objects, so that objects that are close are considered more similar than those far away. Many spectral clustering algorithms determine that two data points are "similar" if they have a nonzero similarity, though other algorithms use a threshold $\epsilon > 0$ to determine if two data points are similar. Though typically more difficult, some clustering algorithms also consider the k closest neighbors of a data point to determine if data points are similar.

To best construct groups or clusters of data most similar, the user of spectral clustering must define similarity between each of the n finite data points in the data set. As defining a scale of similarity is often difficult, many uses of spectral clustering reduce similarity to a binary scale, where two points are similar, or adjacent, if they pass whichever similarity test is used, and they are unsimilar, or nonadjacent, if otherwise.

Once similarity between data is determined, we want to determine which of the data is "most similar," and group the data points according to these similarities. When alternatively considering these data points as vertices of a graph with edges (with respective nonnegative weights) connecting them, then we would like to find clusters or groups where the highest weighted edges between some vertices are together. In other words, we would like to "cut" the graph at edges with the least amount of weight. Because we are separating groups in the graph, we can consider spectral clustering to be a partition of the graph; more specifically, it is a partition of the graph where the "boundaries" between groups of the partition have the least amount of weight as possible, and where the groups have high edge weights within them.

In order to solve the problem of cutting the graph with minimum weight being cut, it is useful to represent the graph's edges and vertices in a matrix. For a data set with n data points, we define its similarity matrix A to be the $n \times n$ symmetric matrix containing the similarities, where

$$A_{ij} = A_{ji} = \text{defined similarity between point } i,j \tag{1}$$

If we consider when similarity is binary, and we solely consider adjacency, we define the adjacency matrix A' to be a special case of the similarity matrix, where

$$A'_{ij} = A'_{ji} = \begin{cases} 1 & \text{if } i,j \text{ adjacent, or connected} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

The similarity matrix for any data set is clearly symmetric, as the similarity between i and j is necessarily the same as the similarity between j and i. Also, the similarity matrix must necessarily be nonnegative, as dissimilar objects have similarity 0, and similar objects have similarity ≥ 0. By convention and for mathematical simplicity, we define $A_{ii} = 0$, as a vertex's similarity with itself is undefined.

Once the similarity matrix is created for a data set, we have obtained a good representation for the connectedness of the graph and of our data set. From this similarity matrix, we can define another useful representation of the similarity of the vertices: the $n \times n$ diagonal degree matrix D, which represents the total summed similarity for the ith vertex, defined by

$$D_{ii} = \sum_j A_{ij}, \tag{3}$$

where $D_{ij} = 0$ if $i \neq j$, implying that D too is nonnegative and symmetric.

Then, we can "normalize" the similarity matrix by considering the Laplacian L of the system of data points. Interestingly, there are numerous different Laplacian matrices we can use for spectral clustering, all of which are variously useful for defining varied clusters within the data. In this report, we will focus on the most general Laplacian

$$L = D - A \tag{4}$$

As aforementioned, A is completely symmetric and nonnegative with 0 on the diagonal, and D is a completely diagonal matrix, so one can envision L as a $n \times n$ symmetric matrix with nonnegative diagonal and nonpositive nondiagonal entries. However, in different mathematical interpretations of spectral clustering, we can define similar Laplacians

$$L = D - A \tag{General Laplacian}$$
$$L_\rho = L - \rho D \tag{Relaxed Laplacian}$$
$$L_{\text{norm}} = D^{-\frac{1}{2}} L D^{-\frac{1}{2}} = I - D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \tag{Normalized Laplacian}$$
$$L_{\text{random walk}} = D^{-1} L = I - D^{-1} A \tag{Random Walk Laplacian}$$

Unless specified, all further mentions of a Laplacian are referring to the general Laplacian. For a more detailed review of the properties of the different types of Laplacians, please see [10]. For a further detailed review of just the general Laplacian, please see [4] and [5].

Although these Laplacians are a way to consider the "normalized" similarities between data points, their usefulness isn't immediately obvious. However, for any $x \in \mathbb{R}^n$,

$$\begin{aligned} x^T L x &= x^T (D - A) x \\ &= \sum_i^n x_i D_{ii} x_i - \sum_{i,j}^n x_i A_{ij} x_j \\ &= \frac{1}{2} \left[\sum_i^n x_i D_{ii} x_i + \sum_j^n x_j D_{jj} x_j - 2 \sum_{i,j}^n x_i A_{ij} x_j \right] \\ &= \frac{1}{2} \left[\sum_i^n x_i^2 D_{ii} + \sum_j^n x_j^2 D_{jj} - 2 \sum_{i,j}^n A_{ij} x_i x_j \right] \\ &= \frac{1}{2} \left[\sum_{i,j}^n x_i^2 A_{ij} + \sum_{i,j}^n x_j^2 A_{ij} - 2 \sum_{i,j}^n A_{ij} x_i x_j \right] \\ &= \frac{1}{2} \sum_{i,j}^n A_{ij} (x_i - x_j)^2 \end{aligned} \tag{5}$$

and a similar result follows for any Laplacian, implying that L is a symmetric positive semi-definite matrix, as for all $x \in \mathbb{R}^n$, $x^T L x \geq 0$. This is of extroadinary relevance, as it implies that all eigenvalues of L are nonnegative.

Further, for the general Laplacian L, because the ith diagonal entry of L (the ith diagonal entry of D) is the negative sum of the rest of the entries in its row or column, we can quickly conclude that $L * 1_n = L * [1, 1, ..., 1]^T = 0 = 0 * [1, 1, ..., 1]^T$, implying that the smallest eigenvalue of L is $\lambda_1 = 0$, with corresponding eigenvector $v_1 = 1_n = [1, 1, ..., 1]^T$. Thus, for all n eigenvalues of L, we have at least one zero eigenvalue and n nonnegative eigenvalues.

Clearly, the Laplacian of our data set has a lot of interesting properties. However, it is still not immediately obvious how the Laplacian assists with spectral clustering. Consider the eigenvalue of our Laplacian $\lambda_1 = 0$, and consider an eigenvector v_1 associated with λ_1. Then, from above,

$$\begin{aligned} 0 = Lv_1 &= v_1 L v_1 \\ &= \frac{1}{2} \sum_{i,j}^{n} A_{ij}((v_1)_i - (v_1)_j)^2 \end{aligned} \tag{6}$$

We can immediately notice that this summation is of nonnegative terms. This implies that for all vertices i, j, we necessarily must have $A_{ij}((v_1)_i - (v_1)_j)^2 = 0$. However, consider when the similarity between vertices i and j is nonzero (or nonnegligible): this implies that $A_{ij} \neq 0$, so that for connected vertices, we must have $(v_1)_i = (v_1)_j$, so that all connected components of the graph must have the same corresponding value in our eigenvector v_1. If we envision our data points to be in an order that groups the $k > 0$ connected components of the graph, then we can consider A to be of the form

$$A = \begin{bmatrix} A_1 & 0 & ... & 0 & 0 \\ 0 & A_2 & ... & 0 & 0 \\ \vdots & & & & \vdots \\ 0 & 0 & ... & 0 & A_k \end{bmatrix},$$

where each A_i is a block matrix of A with zero diagonal and is nonnegative and symmetric. Thus, L is of the form

$$L = \begin{bmatrix} L_1 & & & \\ & L_2 & & \\ & & \ddots & \\ & & & L_k \end{bmatrix},$$

where each L_i is a Laplacian that obeys all of the previous rules as before, and is the Laplacian for the ith connected component of the graph. As discussed above, for connected components, $(v_1)_i = (v_1)_j$, so that each L_i has eigenvalue $(\lambda_i)_1 = 0$ with corresponding eigenvector $1_{|L_i|}$. As L is a block matrix composed of L_i, the spectrum of $L = \cup_i$ (spectrum of L_i). Thus, for each L_i, there exists an eigenvector $1_{|L_i|}$, so the corresponding eigenvector for L is the vector consisting of 1's in the indices of L_i, and 0's in all other indices.

The previous discussion has an extremely important implication: the multiplicity of the $\lambda_1 = 0$ eigenvalue of L is equal to the number of connected components of the data set! Consequently, the eigenspace corresponding to $\lambda_1 = 0$ is the span of $1_{|A_i|}$ of each of the connected components!

1.3 Determination of Algorithm for Clustering

Once the previous linear algebra prerequisites are considered, we can consider the actual technique of spectral clustering. Using the notation as above, if we would like to group our data into $k > 0$ connected components, then we would like to minimize the total edge weight between separate components of the graph. More formally, we would like to find

$$min \left[\frac{1}{2} \sum_{i}^{k} \left(\sum_{m \in A_i, n \in \bar{A}_i} A_{m,n} \right) \right]$$

To clarify, we determine the total edge weight between the elements of a component and all vertices outside of the component, and we would like to minimize the sum of this totaled weight over all k

components. However, this minimization would have a very trivial solution: just make every vertex its own component, and this sum equates to 0, which would give us no information at all! Thus, we would like $|A_i|$, the size of the clusters, to be taken into account. More specifically, we would like $|A_i|$ to be large enough for us to gather enough information from our clustering. Thus, a better minimization problem would be to find a more balanced minimum; hence, we would like to find

$$min\left[\frac{1}{2}\sum_{i}^{k}\left(\frac{\sum_{m\in A_i,n\in\bar{A}_i}A_{m,n}}{|A_i|}\right)\right]$$

The exact analytic solution to this minimization problem is quite complex and is often unnecessary for many research applications. Thus, a relaxed approach to solving this problem is much more appropriate, which is exactly what spectral clustering seeks. To conduct a cut that minimizes the ratio between the total different component edge weight sum and the size of the components, we will follow a derivation that resembles the one in [10], yet is found ubiquitously in the literature. For a data set with n data points v_i that are to be clustered into k components, we can define a $H \in \mathbb{R}^{n\times k}$ where

$$H_{i,j} = \begin{cases} \frac{1}{\sqrt{|A_j|}} & \text{if } v_i \in A_j \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

for $i \in [1,n]\cap\mathbb{Z}, j \in [1,k]\cap\mathbb{Z}$. If we assume that each data point v_i is clustered into one and only one group, then each column will have Euclidean length 1 and each column will be completely orthogonal to all others, so that H is composed of orthonormal columns and is semi-orthogonal, so that $H^TH = HH^T = I$. Then, with the notation that H_i is the ith column of H, we can use Eq. 5, Eq. 6, and the argument that followed to express

$$\begin{aligned} H_i^TLH_i &= \frac{1}{2}\sum_{i,j}^{n}A_{ij}(H_i - H_j)^2 \\ &= \frac{1}{2}\sum_{i\in A_i,j\in\bar{A}_i}A_{ij}\left(\sqrt{\frac{1}{|A_i|}}+\sqrt{\frac{1}{|\bar{A}_i|}}\right)^2 + \frac{1}{2}\sum_{i\in\bar{A}_i,j\in A_i}A_{ij}\left(-\sqrt{\frac{1}{|A_i|}}-\sqrt{\frac{1}{|\bar{A}_i|}}\right)^2 \\ &= \frac{1}{2}\sum_{i}^{k}\left(\frac{\sum_{m\in A_i,n\in\bar{A}_i}A_{m,n}}{|A_i|}\right), \end{aligned} \tag{8}$$

which is exactly what we are trying to minimize. Due to the box matrix nature of L, it is easily shown that $H_i^TLH_i = (H^TLH)_{ii}$. Finally, combining these facts, **it can be concluded that spectral clustering aims to minimize**

$$\begin{aligned} \frac{1}{2}\sum_{i}^{k}\left(\frac{\sum_{m\in A_i,n\in\bar{A}_i}A_{m,n}}{|A_i|}\right) &= \sum_{i=1}^{k}H_i^TLH_i \\ &= \sum_{i=1}^{k}(H^TLH)_{ii} \\ &= \boxed{\text{Tr}(H^TLH)} \end{aligned} \tag{9}$$

As aforementioned, an exact solution to this is quite difficult to find, but is far easier if we apply a slight relaxation; instead of requiring H to be defined by Eq. 7, we can simply require that H satisfy $H^TH = I$ so that it is still composed of orthonormal columns, but consisting of arbitrary values. This is now just a general minimization of a trace problem, so according to the Rayleigh-Ritz theorem, we can find a relaxed solution to the minimization of the trace by choosing H to be composed of the first k eigenvectors of L.

However, to find a discrete partition between the n data points, we need to discretize the solution H, which consists of real valued values. This can be done by considering the k-means algorithm on the rows of H, which then uniquely determines a discrete partition of the n data points.

To summarize, spectral clustering minimizes the total edge weight between separate clusters by minimizing the ratio of the separate cluster edge weights and the size of the clusters, which was shown to be equivalent of the trace of $H^T LH$, where $L \in \mathbb{R}^{n\times n}$ is the Laplacian of the data set, $H \in \mathbb{R}^{n\times k}$ and $H^T H = I$. This is minimized when H is composed of the first k eigenvectors of L. H is then discretized by considering the k-means algorithm on its rows.

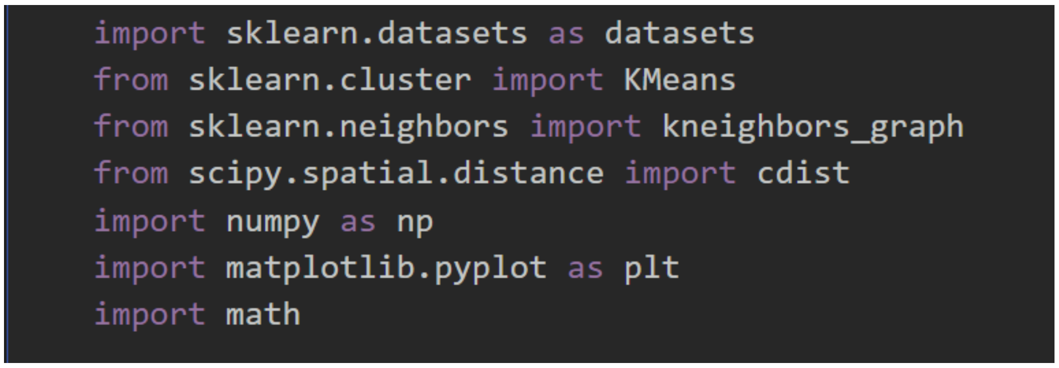
2 Algorithm

In this section, we will summarize the algorithm, describe the python implementation of the algorithm in-depth, show how the algorithm works on various 2D and 3D data sets, and explore some of the characteristics of the algorithm, including space+time complexity.

As described in the theory section, spectral clustering is an unsupervised learning technique (or, if tweaked slightly, a dimensionality reduction technique). The input to any unsupervised clustering algorithm is a set of data points, and the output is a clustering of the input data (i.e. a set of labels, one for each input data point, where points with the same label are in the same cluster). If used for dimensionality reduction, the output will be a new set of data points, with each data point having the same or less features than the original data points.

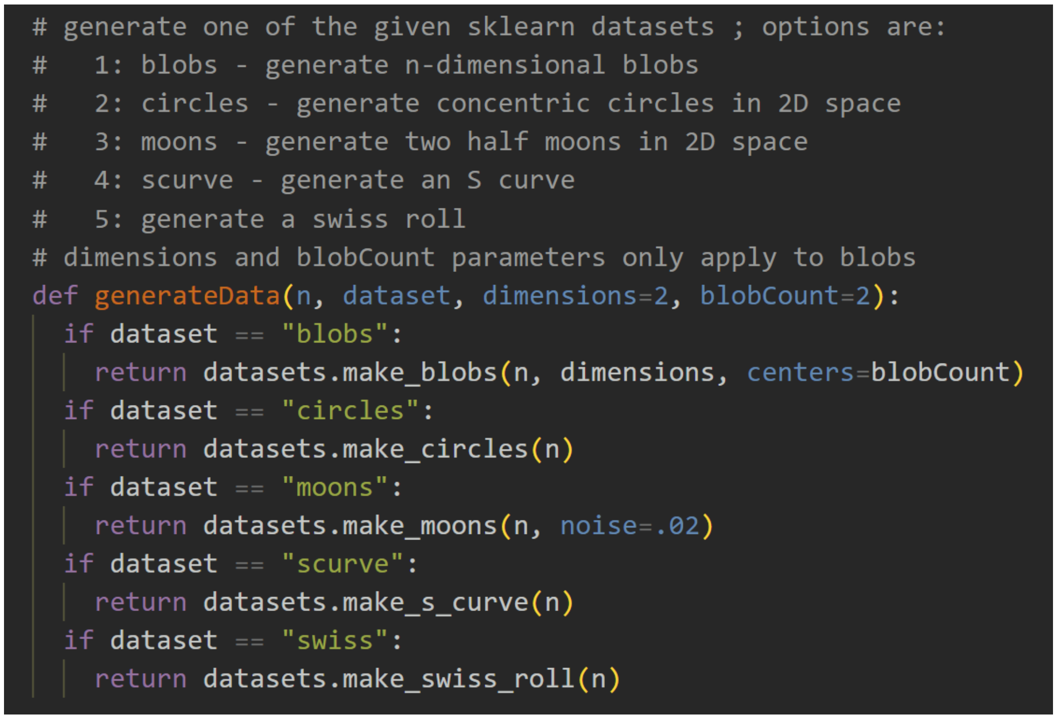
We will now look at a python implementation of spectral clustering. The theory has been explained in section 1, so we will focus on what the code is actually doing here. Each step is broken up into an individual function, so we can discuss what the code is doing and analyze its runtime piece-by-piece. A full copy of the code is available both in the github repository (linked here) and in the Google Colab (linked here).

First, here are the relevant import statements. In particular, note that `sklearn.datasets` is imported as `datasets`. This module provides many sample datasets that we make use of below.



```python
import sklearn.datasets as datasets
from sklearn.cluster import KMeans
from sklearn.neighbors import kneighbors_graph
from scipy.spatial.distance import cdist
import numpy as np
import matplotlib.pyplot as plt
import math
```

Next, we have a `generateData` function. This is a wrapper for several of the functions in `sklearn.datasets`, and we use it extensively below to test the spectral clustering algorithm.

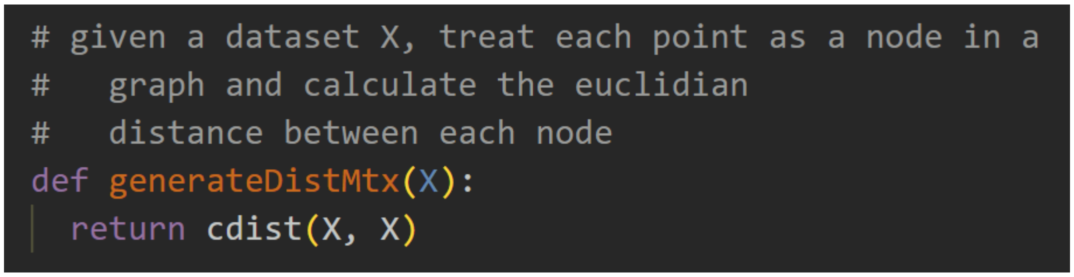


```python
# generate one of the given sklearn datasets ; options are:
#   1: blobs - generate n-dimensional blobs
#   2: circles - generate concentric circles in 2D space
#   3: moons - generate two half moons in 2D space
#   4: scurve - generate an S curve
#   5: generate a swiss roll
# dimensions and blobCount parameters only apply to blobs
def generateData(n, dataset, dimensions=2, blobCount=2):
  if dataset == "blobs":
    return datasets.make_blobs(n, dimensions, centers=blobCount)
  if dataset == "circles":
    return datasets.make_circles(n)
  if dataset == "moons":
    return datasets.make_moons(n, noise=.02)
  if dataset == "scurve":
    return datasets.make_s_curve(n)
  if dataset == "swiss":
    return datasets.make_swiss_roll(n)
```

This function is relatively self-explanatory - `n` is the number of data points to generate, and `dataset` is a string that determines which sklearn function to call. The `dimensions` and `blobCount` parameters only apply to the blob dataset, and they determine the number of features per data point and number of cluster centers, respectively. All of these datasets are visualized below while testing the algorithm, so please see the later sections for a better understanding of what these datasets look like.

The runtime and space complexity of this function is not part of the spectral clustering algorithm, but we can still analyze it: since each sklearn function generates `n` data points, with each data point generated in the same way, the runtime of this function is $O(n)$. Since this function returns an nxd matrix, where d is the number of dimensions/features, the space complexity of this function is $O(nd)$.

Next, we have the `generateDistMtx` function. This is the first piece of the spectral clustering algorithm.

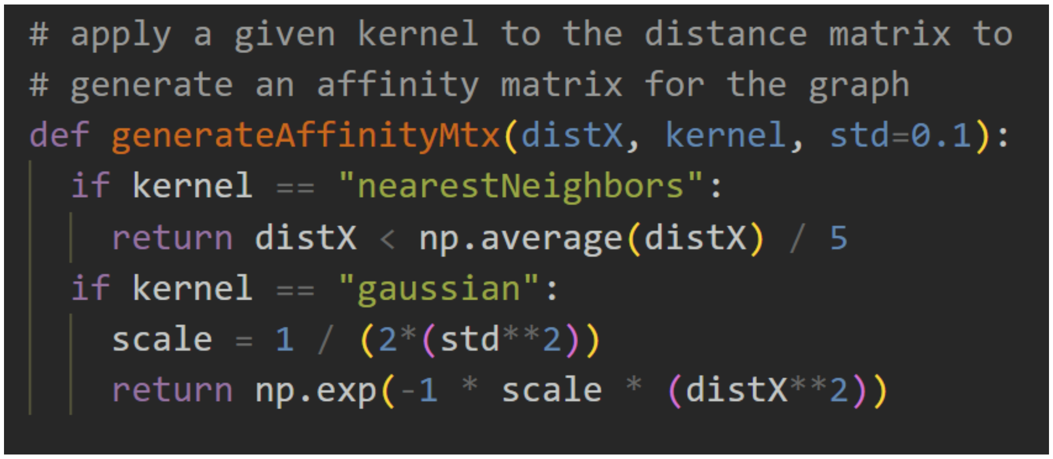


```python
# given a dataset X, treat each point as a node in a
#   graph and calculate the euclidian
#   distance between each node
def generateDistMtx(X):
  return cdist(X, X)
```

The input parameter `X` is a dataset. Suppose this dataset has `n` points, each with `k` features. This function treats each point in the dataset as a node in a graph (as discussed in the theory section), and calculates an `nxn` matrix where the `ijth` entry is the euclidian distance between the `ith` and `jth` node (data point). Note that the scipy function `cdist` is used to generate this matrix. The `cdist(X, Y)` function generates a matrix containing the distance between every point in the set X to every point in the set Y, so `cdist(X, X)` gives us the desired matrix.

Because the cdist function calclulates the distance from every point in the first dataset to every point in the second dataset, the runtime of this function is $O(n^2)$, where n is the number of data points in X. Since the function allocates and returns an `nxn` matrix, as discussed above, the space complexity is $O(n^2)$ as well.

Next, we have the `generateAffinityMtx` function. As discussed in the theory section, this function forms the affinity matrix of the graph using the distance matrix generated in the previous function.

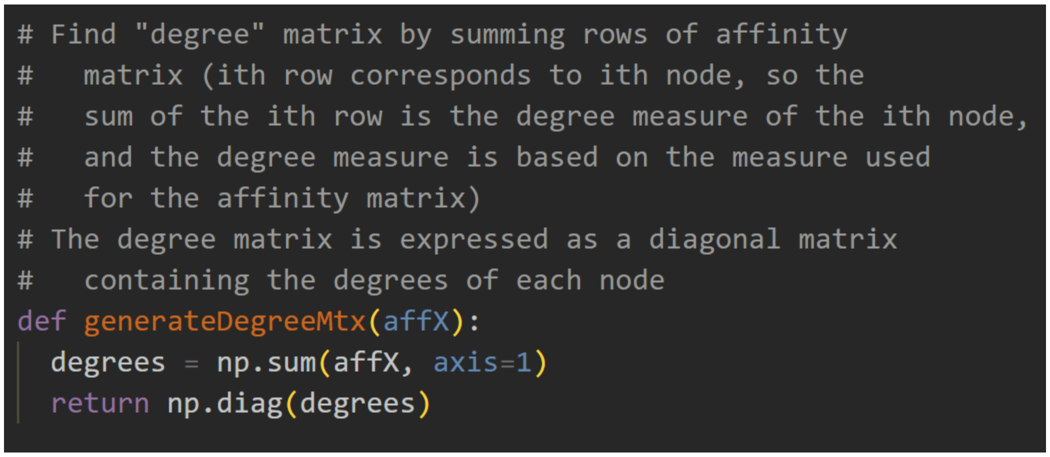


```
# apply a given kernel to the distance matrix to
# generate an affinity matrix for the graph
def generateAffinityMtx(distX, kernel, std=0.1):
  if kernel == "nearestNeighbors":
    return distX < np.average(distX) / 5
  if kernel == "gaussian":
    scale = 1 / (2*(std**2))
    return np.exp(-1 * scale * (distX**2))
```

The input parameter `distX` is the previously generated distance matrix. This function gives the option to apply one of two kernels to the distance matrix to get the affinity matrix. The first kernel is the nearest neighbors kernel. It transforms the distance matrix such that the only nonzero entries in the affinity matrix are those that correspond to nodes that are "sufficiently close". The second kernel is the Gaussian kernel, which generates the affinity matrix entries by squaring, scaling, then exponentiating the distance matrix entries. The `std` parameter corresponds to the standard deviation used in the scale factor. As we will see during testing, the Gaussian kernel is much more versatile than the nearest neighbors kernel.

The runtime of this function is $O(n^2)$ because it performs a constant time operation on every element of the distance matrix. It does not utilize any additional non-constant space, so the space complexity is $O(1)$

Next, we have the `generateDegreeMtx` function. This function generates a diagonal matrix with the degree of the ith node in the ith diagonal component.

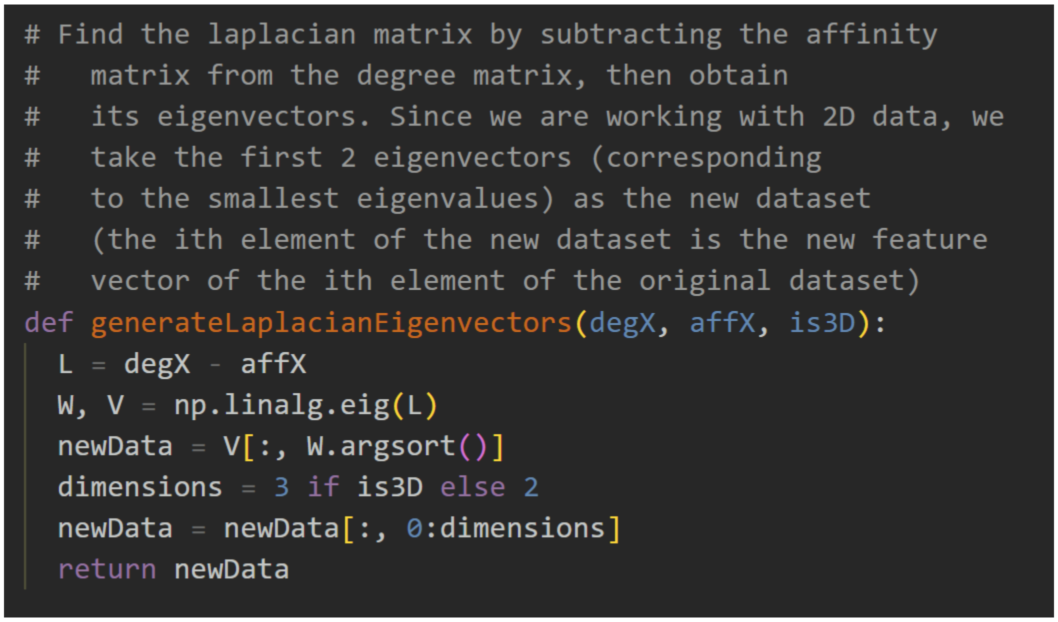


```python
# Find "degree" matrix by summing rows of affinity
#   matrix (ith row corresponds to ith node, so the
#   sum of the ith row is the degree measure of the ith node,
#   and the degree measure is based on the measure used
#   for the affinity matrix)
# The degree matrix is expressed as a diagonal matrix
#   containing the degrees of each node
def generateDegreeMtx(affX):
    degrees = np.sum(affX, axis=1)
    return np.diag(degrees)
```

Since the ith row of the affinity matrix contains all the connection information for node i, the sum of row i gives a degree measure of the ith node. Note that this degree measure is dependent on the measure used in the affinity matrix; if the gaussian kernel is used, the degree will be the sum of all the gaussian-scaled distances to other nodes. If the nearest neighbors kernel is used, the degree will be the total number of nodes that the ith node is "close" to. The `np.diag` function just puts these degree values into a diagonal `nxn` matrix.

The runtime of this function is $O(n^2)$ because the sum must iterate over every element in the `nxn` matrix. The space complexity is $O(n^2)$ since another `nxn` matrix is allocated and returned.

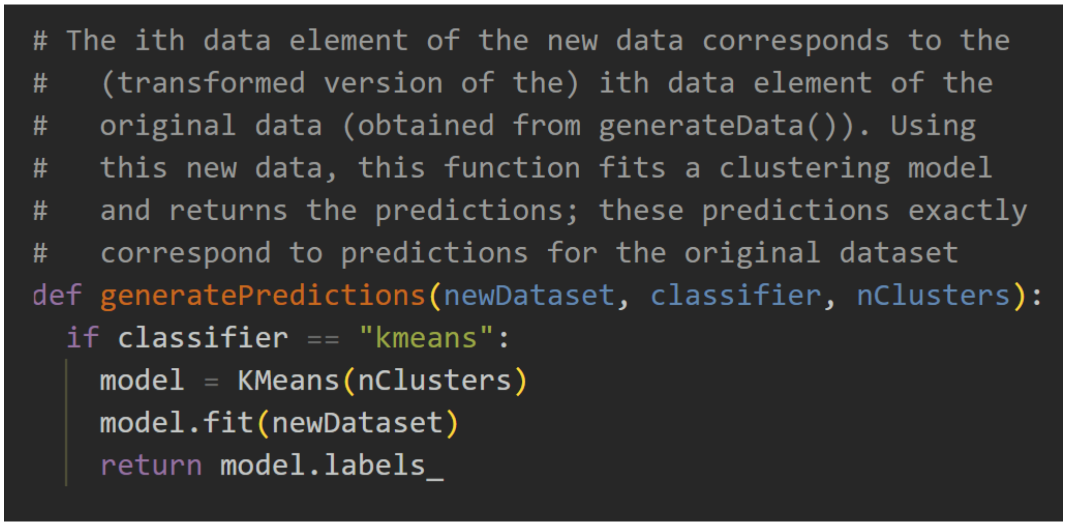
Next, we have the `generateLaplacianEigenvectors` function. This finds the laplacian of the graph, as discussed in the theory section, by subtracting the affinity matrix from the degree matrix. The new dimensionality-reduced dataset is then formed by concatenating the smallest $j(\leq k)$ eigenvectors of the laplacian, where k is the number of features of the original dataset. In this implementation, we use $j = k$.



```python
# Find the laplacian matrix by subtracting the affinity
#   matrix from the degree matrix, then obtain
#   its eigenvectors. Since we are working with 2D data, we
#   take the first 2 eigenvectors (corresponding
#   to the smallest eigenvalues) as the new dataset
#   (the ith element of the new dataset is the new feature
#   vector of the ith element of the original dataset)
def generateLaplacianEigenvectors(degX, affX, is3D):
    L = degX - affX
    W, V = np.linalg.eig(L)
    newData = V[:, W.argsort()]
    dimensions = 3 if is3D else 2
    newData = newData[:, 0:dimensions]
    return newData
```

The most time-intensive component of this function is the numpy eigenvalue decomposition, which is $O(n^3)$, so the runtime of this function is $O(n^3)$. The largest matrix that gets allocated is `nxn`, so the space complexity of this function is $O(n^2)$.

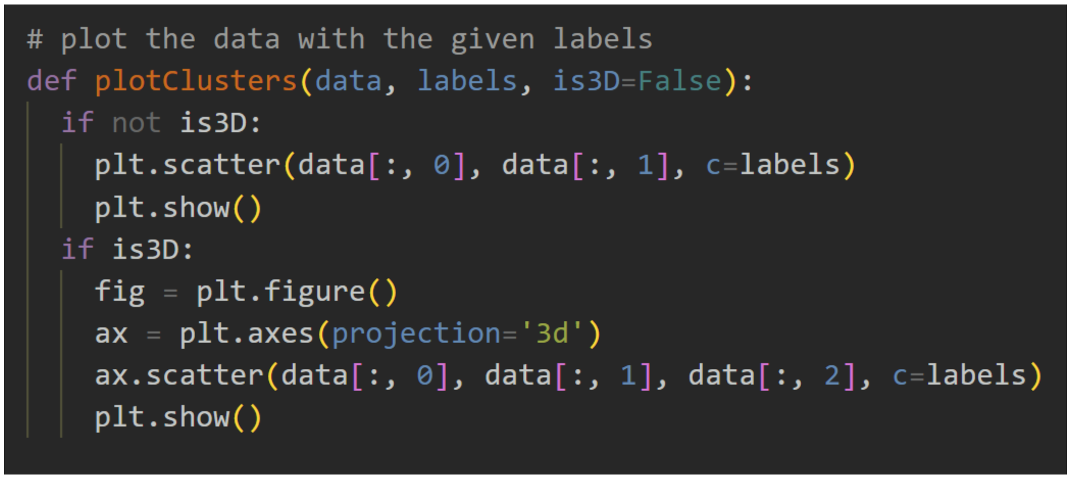
Next, we have the `generatePredictions` function. This function simply takes in a dataset (which, in our case, will be the transformed dataset obtained from the `generateLaplacianEigenvectors` function) and fits an sklearn kmeans model to the data, returning the computed labels. This is how the final clustering (which is the final step described in the theory section) is done.



```python
# The ith data element of the new data corresponds to the
#   (transformed version of the) ith data element of the
#   original data (obtained from generateData()). Using
#   this new data, this function fits a clustering model
#   and returns the predictions; these predictions exactly
#   correspond to predictions for the original dataset
def generatePredictions(newDataset, classifier, nClusters):
  if classifier == "kmeans":
    model = KMeans(nClusters)
    model.fit(newDataset)
    return model.labels_
```

The runtime of this function is determined by the `kmeans.fit` function, as it is the most expensive. Since kmeans is an iterative expectation-maximization algorithm that updates the cluster assignments then centers repeatedly (where each iteration is $O(n)$), the runtime is determined by the total number of iterations. According to the sklearn documentation, the default is 300. Regardless, since the number of iterations is small relative to n, the runtime of this function is $O(n)$. Kmeans internally uses an `nxc` matrix to keep track of the cluster assignments, where c is the number of predetermined clusters (i.e. a constant relative to n). So the space complexity of this function is also $O(n)$

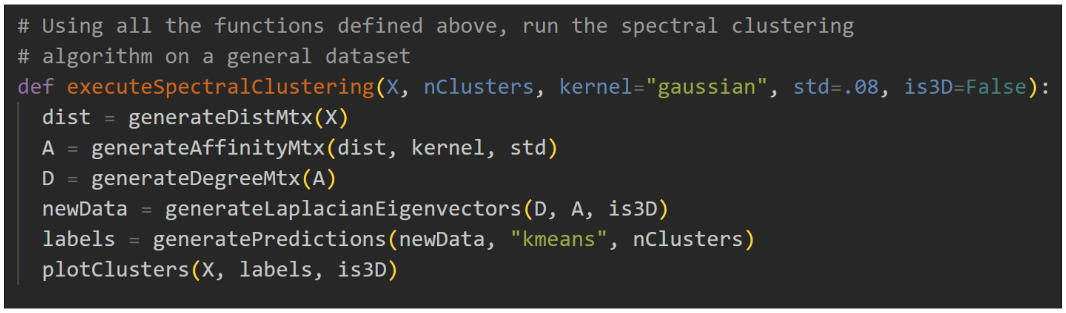
Next, we have the `plotClusters` function, which just plots a given set of data and corresponding labels using matplotlib. In our case, we use this function to plot the original data with the labels obtained from the spectral clustering algorithm. The one interesting thing to note here is that we can plot both 2D and 3D data, which leads to the interesting visualizations shown below.



```
# plot the data with the given labels
def plotClusters(data, labels, is3D=False):
  if not is3D:
    plt.scatter(data[:, 0], data[:, 1], c=labels)
    plt.show()
  if is3D:
    fig = plt.figure()
    ax = plt.axes(projection='3d')
    ax.scatter(data[:, 0], data[:, 1], data[:, 2], c=labels)
    plt.show()
```

This function is not part of the "raw" spectral clustering algorithm, so we will not include its time/space complexity.

Finally, we have the `executeSpectralClustering` function. This function combines all the sub-functions described above to fully execute (and display the results of) spectral clustering on a given dataset.



```
# Using all the functions defined above, run the spectral clustering
# algorithm on a general dataset
def executeSpectralClustering(X, nClusters, kernel="gaussian", std=.08, is3D=False):
  dist = generateDistMtx(X)
  A = generateAffinityMtx(dist, kernel, std)
  D = generateDegreeMtx(A)
  newData = generateLaplacianEigenvectors(D, A, is3D)
  labels = generatePredictions(newData, "kmeans", nClusters)
  plotClusters(X, labels, is3D)
```

The runtime of this function is the maximum of the time complexities of all the sub-functions, which is $O(n^3)$ (due to the eigendecomposition). Similarly, the space complexity of this function is $O(n^2)$.

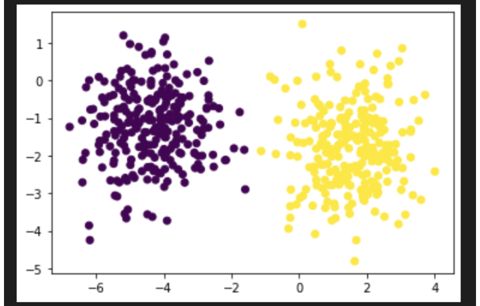
So, we see that the runtime of this spectral clustering algorithm is $O(n^3)$, and the space complexity is $O(n^2)$.

It is important to note that even though this is the theoretical bound, it is assuming that there is only one processor in play. In reality, numpy, scipy, and other python libraries are heavily optimized and use parallelism under the hood (especially for simple operations such as those involving matrices). This parallelism allows multiple cores on a computer to work on a given program at the same time, speeding up the execution greatly. This is why if the runtime is measured, it will likely be faster than the theoretical bound obtained here (on most computers). The runtime can be further optimized by precompiling some of the python code (so that it is compiled rather than interpreted)

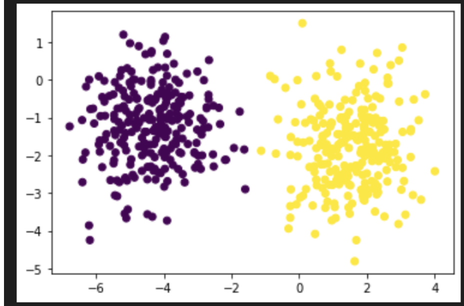
We will now show how the algorithm works on various datasets, and discuss any interesting results.

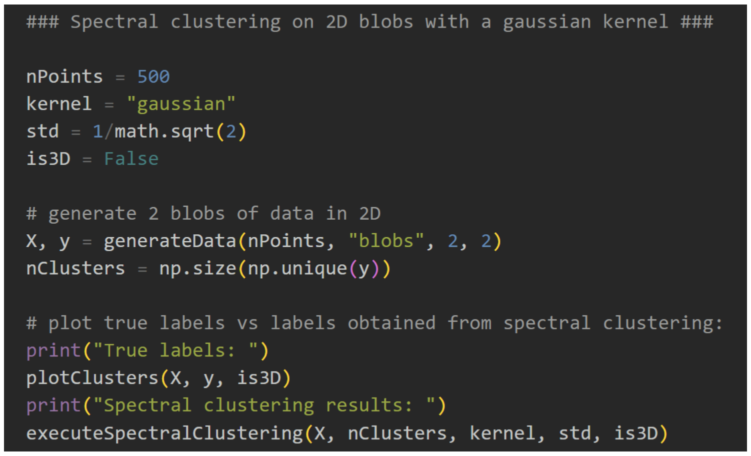
The algorithm works as expected on 2-dimensional blobs of points:

```
True labels:
```



```
Spectral clustering results:
```

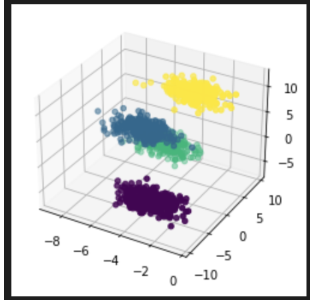




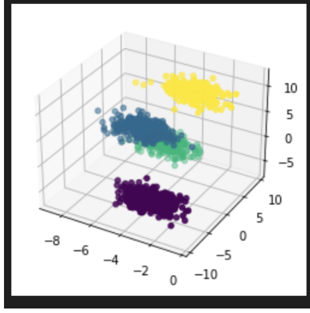
```
### Spectral clustering on 2D blobs with a gaussian kernel ###

nPoints = 500
kernel = "gaussian"
std = 1/math.sqrt(2)
is3D = False

# generate 2 blobs of data in 2D
X, y = generateData(nPoints, "blobs", 2, 2)
nClusters = np.size(np.unique(y))

# plot true labels vs labels obtained from spectral clustering:
print("True labels: ")
plotClusters(X, y, is3D)
print("Spectral clustering results: ")
executeSpectralClustering(X, nClusters, kernel, std, is3D)
```

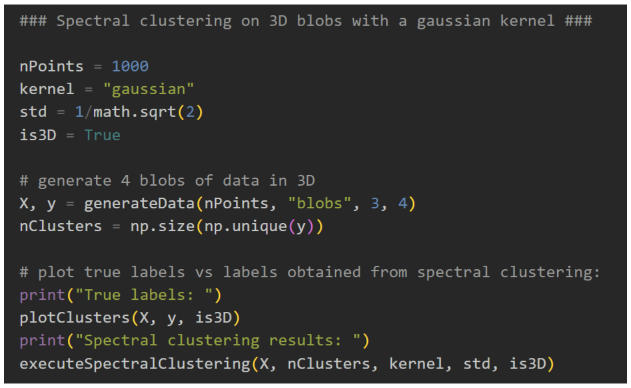
And similarly on 3-dimensional blobs of data:

```
True labels:
```



```
Spectral clustering results:
```

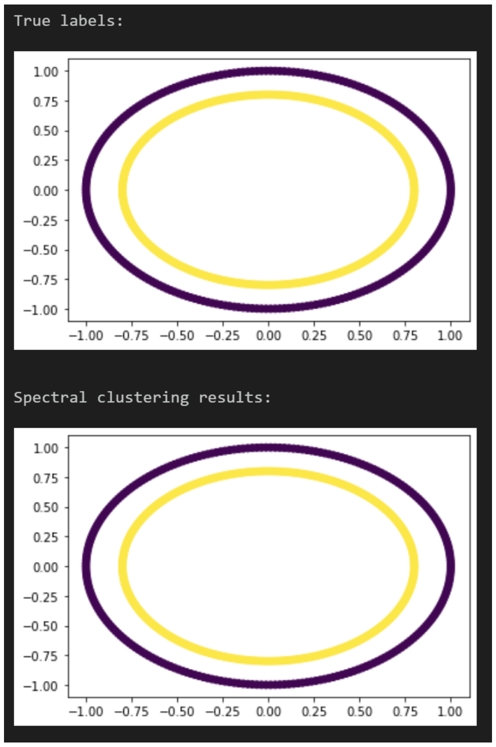




```
### Spectral clustering on 3D blobs with a gaussian kernel ###

nPoints = 1000
kernel = "gaussian"
std = 1/math.sqrt(2)
is3D = True

# generate 4 blobs of data in 3D
X, y = generateData(nPoints, "blobs", 3, 4)
nClusters = np.size(np.unique(y))

# plot true labels vs labels obtained from spectral clustering:
print("True labels: ")
plotClusters(X, y, is3D)
print("Spectral clustering results: ")
executeSpectralClustering(X, nClusters, kernel, std, is3D)
```

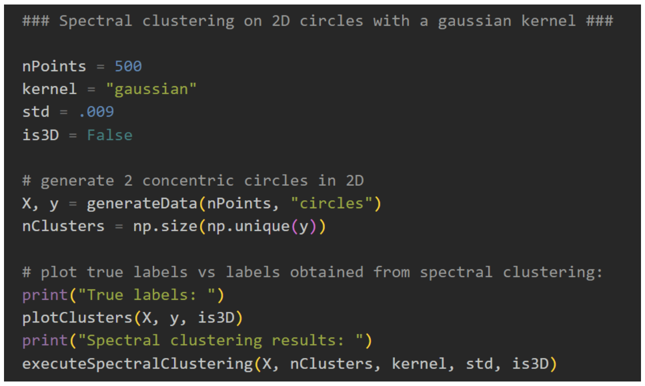
These results are not as interesting - kmeans would give the same results since it is a radial clustering algorithm. The true advantage of spectral clustering is shown on the next few datasets.

Here are the results of spectral clustering on concentric circles:



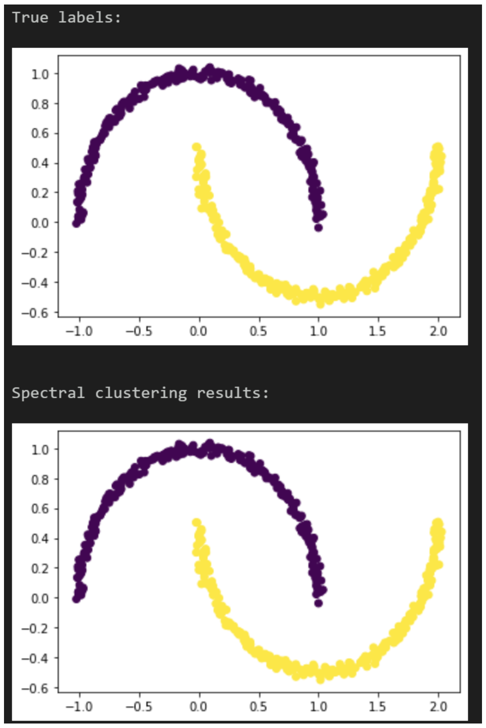
```
True labels:
```

```
Spectral clustering results:
```



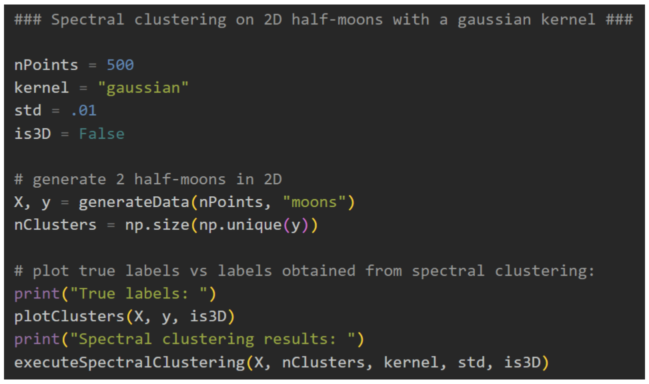
```
### Spectral clustering on 2D circles with a gaussian kernel ###

nPoints = 500
kernel = "gaussian"
std = .009
is3D = False

# generate 2 concentric circles in 2D
X, y = generateData(nPoints, "circles")
nClusters = np.size(np.unique(y))

# plot true labels vs labels obtained from spectral clustering:
print("True labels: ")
plotClusters(X, y, is3D)
print("Spectral clustering results: ")
executeSpectralClustering(X, nClusters, kernel, std, is3D)
```

And on two half-moons:



```
True labels:
```

```
Spectral clustering results:
```

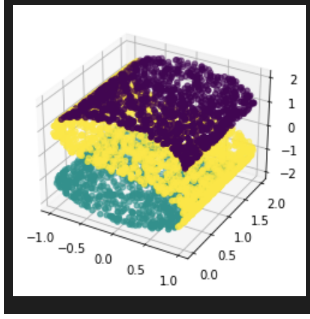


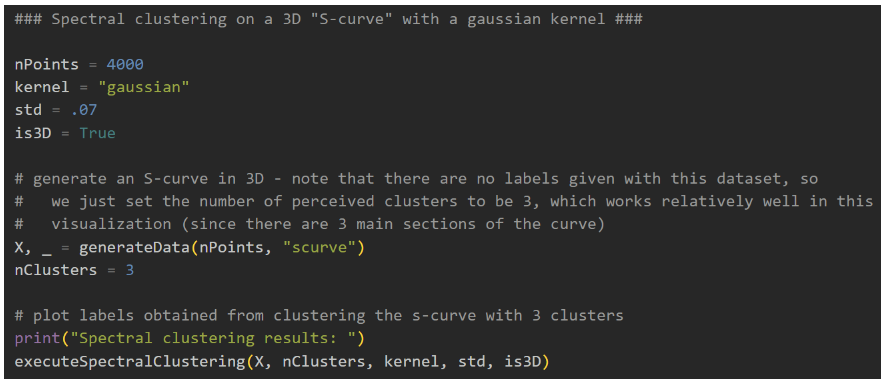
```
### Spectral clustering on 2D half-moons with a gaussian kernel ###

nPoints = 500
kernel = "gaussian"
std = .01
is3D = False

# generate 2 half-moons in 2D
X, y = generateData(nPoints, "moons")
nClusters = np.size(np.unique(y))

# plot true labels vs labels obtained from spectral clustering:
print("True labels: ")
plotClusters(X, y, is3D)
print("Spectral clustering results: ")
executeSpectralClustering(X, nClusters, kernel, std, is3D)
```

Clearly, spectral clustering works very well on these datasets, whereas kmeans would have given results that are completely incorrect due to its radial nature (see section 3 below for a visualization of how kmeans would act on this data).

Finally, we can see how spectral clustering acts on some 3-dimensional data:

```
Spectral clustering results:
```



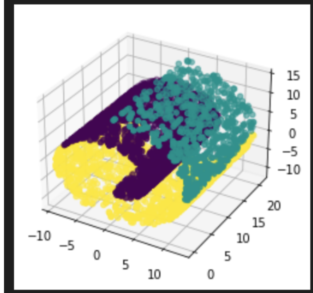


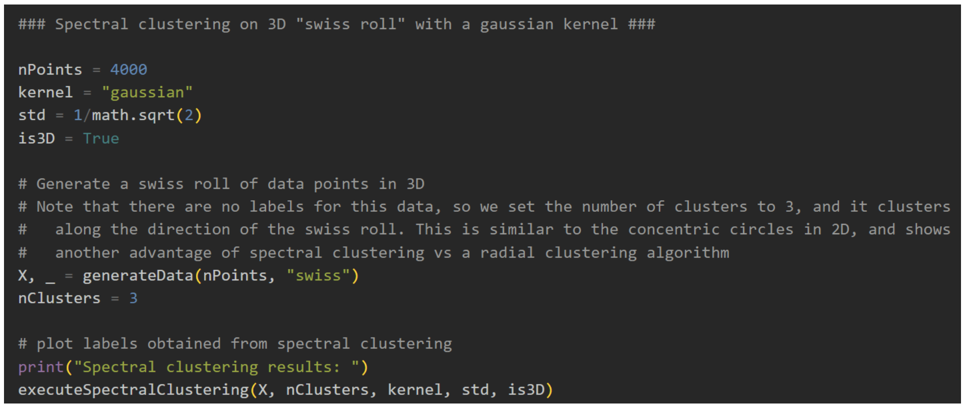
```
### Spectral clustering on a 3D "S-curve" with a gaussian kernel ###

nPoints = 4000
kernel = "gaussian"
std = .07
is3D = True

# generate an S-curve in 3D - note that there are no labels given with this dataset, so
#   we just set the number of perceived clusters to be 3, which works relatively well in this
#   visualization (since there are 3 main sections of the curve)
X, _ = generateData(nPoints, "scurve")
nClusters = 3

# plot labels obtained from clustering the s-curve with 3 clusters
print("Spectral clustering results: ")
executeSpectralClustering(X, nClusters, kernel, std, is3D)
```

We can see that the spectral clustering neatly splits the S-curve into three even clusters according to layer - something that kmeans would again not be able to do.

Here is a 3-dimensional analog of the 2D concentric circles from above:

```
Spectral clustering results:
```





```
### Spectral clustering on 3D "swiss roll" with a gaussian kernel ###

nPoints = 4000
kernel = "gaussian"
std = 1/math.sqrt(2)
is3D = True

# Generate a swiss roll of data points in 3D
# Note that there are no labels for this data, so we set the number of clusters to 3, and it clusters
#   along the direction of the swiss roll. This is similar to the concentric circles in 2D, and shows
#   another advantage of spectral clustering vs a radial clustering algorithm
X, _ = generateData(nPoints, "swiss")
nClusters = 3

# plot labels obtained from spectral clustering
print("Spectral clustering results: ")
executeSpectralClustering(X, nClusters, kernel, std, is3D)
```

Again, spectral clustering neatly splits the spiral in a way that kmeans would not be able to do, as points are grouped by their proximity to the center (or, equivalently, their parameterized position on the spiral).

3 Data Exploration

The Adjusted Rand Index (ARI) is a measure used to evaluate the quality of a clustering algorithm by comparing the predicted cluster assignments to the true cluster assignments (ground truth). It is based on counting the pairs of elements in the same or different clusters in both the predicted clustering and the true clustering.

Let C be the set of true clusters and K be the set of predicted clusters. Define the following variables:

- n_{ij}: The number of elements that are common to both cluster C_i and cluster K_j.
- a_i: The sum of all n_{ij} for a given true cluster C_i.
- b_j: The sum of all n_{ij} for a given predicted cluster K_j.
- n: The total number of elements in the dataset.

The Rand Index (RI) is calculated as the sum of true positive (TP) and true negative (TN) pairs divided by the total number of pairs:

$$RI = \frac{TP+TN}{TP+TN+FP+FN}$$

where TP represents pairs of elements that are in the same cluster in both the predicted clustering and the true clustering, TN represents pairs of elements that are in different clusters in both the predicted clustering and the true clustering, FP represents pairs of elements that are in the same cluster in the predicted clustering but not in the true clustering, and FN represents pairs of elements that are in the same cluster in the true clustering but not in the predicted clustering. The Rand Index can be reformulated using the variables defined earlier:

$$RI = \frac{\sum_{i=1}^{|C|}\sum_{j=1}^{|K|}\binom{n_{ij}}{2}+\binom{n}{2}-\sum_{i=1}^{|C|}\binom{a_i}{2}-\sum_{j=1}^{|K|}\binom{b_j}{2}}{\binom{n}{2}}$$

where $\binom{x}{2} = \frac{x(x-1)}{2}$.

However, the Rand Index does not account for the fact that random cluster assignments can also result in a non-zero RI value. To adjust for this, we compute the expected Rand Index (Expected RI) under the assumption of randomness and then normalize the RI by subtracting the Expected RI and dividing by the maximum possible RI minus the Expected RI:

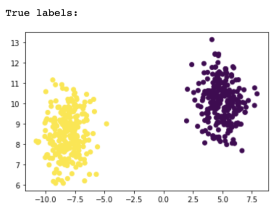
$$ARI = \frac{\sum_{ij}\binom{n_{ij}}{2}-\left[\sum_i\binom{a_i}{2}\sum_j\binom{b_j}{2}\right]/\binom{N}{2}}{\frac{1}{2}\left[\sum_i\binom{a_i}{2}+\sum_j\binom{b_j}{2}\right]-\left[\sum_i\binom{a_i}{2}\sum_j\binom{b_j}{2}\right]/\binom{N}{2}}$$

The resulting ARI score ranges from -1 to 1, with 1 indicating perfect clustering, 0 indicating random clustering, and -1 indicating completely opposite clustering compared to the ground truth.

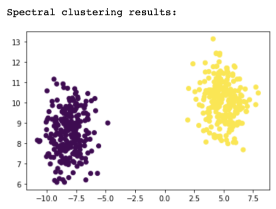
The following figures encapsulate our results using various synthetic data sets generated with SKlearn. The odd-numbered figures show three graphs each, with graphs (a) representing the actual synthetic data, and graphs (b) and (c) showing the results of our spectral clustering algorithm and the SKlearn implementation of the K-means clustering algorithm, respectively.

The even-numbered figures plot levels of noise vs the corresponding adjusted random index for both algorithms. In the case of the figure 2 and 4, the noise tests for the "blob" clusters, the noise corresponds to an increase in the default standard deviation of each blob. The X-axis represents the level of noise increasing from 0 to 1, but for the sake of visual clarity, the information is presented as quartile averages such that Q1 corresponds to adding 0.25 noise to the data, Q2 corresponds to 0.5, etc. In the case of the figure 2 and 4, the noise tests for the "blob" clusters, the noise corresponds to an increase in the default standard deviation of each blob. For the rest of the images, the noise corresponds to the standard deviation of the gaussian noise.

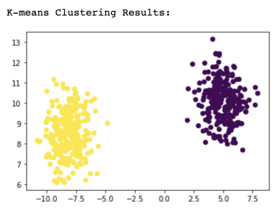
Additionally, there is one table for each type of data set. These tables show the sample mean of the adjusted random index corresponding to a sample of 100 indices for each algorithm. The tables also contain confidence intervals for these means at the 0.05 significance level. In the case of the tables corresponding to the concentric circles and semi-circles data sets, a noise level of 0.02 was introduced in order to produce meaningful data since our spectral algorithm achieves an ARI of 1.0 nearly every time with these two data sets. For the 2D and 3D blob data sets no additional noise was introduced, so each cluster had the default standard deviation of 1. The last table corresponds to a five-dimensional space with three clusters and noise level 0.1. During testing it became apparent that both algorithms perform better with higher dimensional data, at least when looking at the blob distribution in Sklearn. When exceeding 7 dimensions, both algorithms achieved an ARI of 1.0 nearly every time, so we are only showcasing this 5-dimensional case with some noise included.



(a) True Cluster Labels



(b) Spectral Clustering



(c) K-means Clustering

Figure 1: 2D Blobs

Table 1: 2D Blobs ARI distribution

Algorithm	ARI Sample Mean	Confidence Interval
Spectral Clustering	0.9550	(0.9285, 0.9816)
K-means Clustering	0.9510	(0.9236, 0.9784)

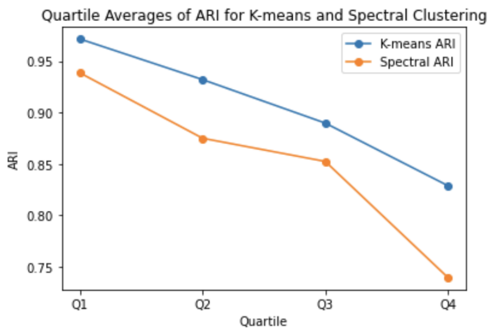
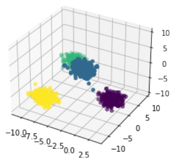
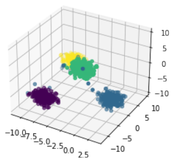


Figure 2: 2D Blobs Noise Sensitivity


(a) True Cluster Labels


(b) Spectral Clustering

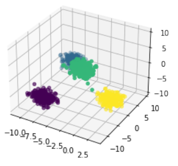

(c) K-means Clustering

Figure 3: 3D Blobs

Table 2: 3D Blobs ARI distribution

Algorithm	ARI Sample Mean	Confidence Interval
Spectral Clustering	0.9835	(0.9747, 0.9923)
K-means Clustering	0.9635	(0.9472, 0.9798)

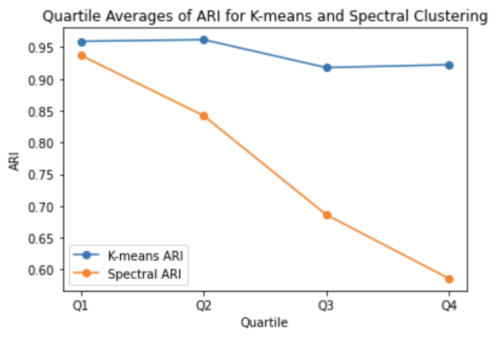
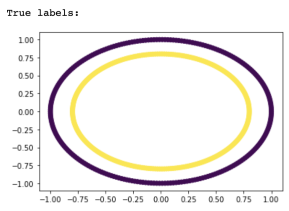
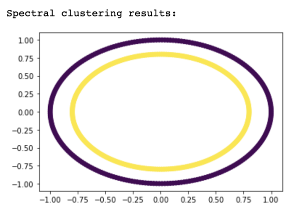


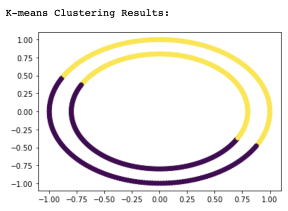
Figure 4: 3D Blobs Noise Sensitivity



(a) True Cluster Labels



(b) Spectral Clustering



(c) K-means Clustering

Figure 5: Concentric Circles

Table 3: Concentric Circles ARI distribution

Algorithm	ARI Sample Mean	Confidence Interval
Spectral Clustering	-0.0020	(-0.0020, -0.0020)
K-means Clustering	0.1976	(0.1394, 0.2557)

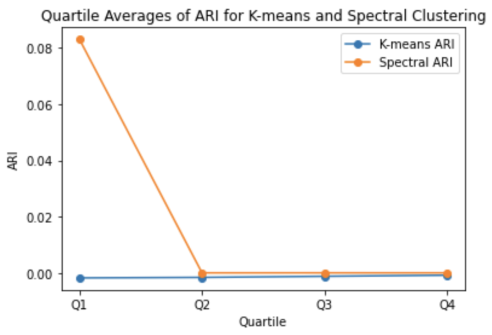
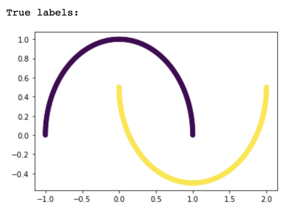
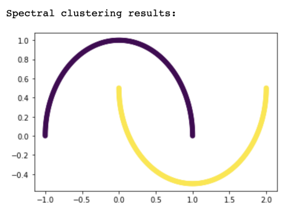


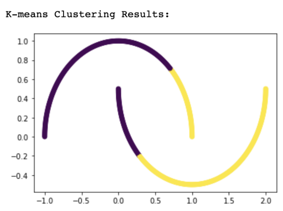
Figure 6: Concentric Circles Noise Sensitivity



(a) True Cluster Labels



(b) Spectral Clustering



(c) K-means Clustering

Figure 7: Semi-Circles

Table 4: Semi-Circles ARI distribution

Algorithm	ARI Sample Mean	Confidence Interval
Spectral Clustering	0.2505	(0.2492, 0.2518)
K-means Clustering	0.9910	(0.9756, 1.00)

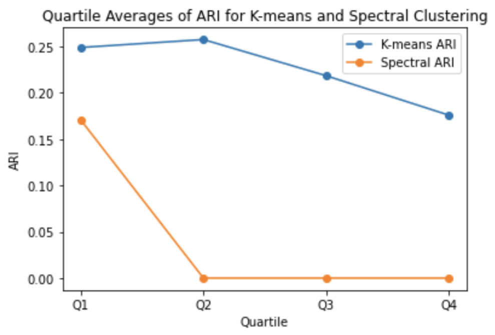


Figure 8: Semi-Circles Noise Sensitivity

Table 5: 5D Blob ARI distribution

Algorithm	ARI Sample Mean	Confidence Interval
Spectral Clustering	0.9984	(0.9966, 1.0)
K-means Clustering	0.9776	(0.9598, 0.9954)

Perhaps the most noticeable result is how much better the spectral clustering algorithm performs relative to K-means clustering when studying either concentric circles or semi-circles. The predictions made by the K-means clustering algorithm are almost as good as random, as shown in their respective tables. Nevertheless, our implementation of the spectral clustering algorithm is more sensitive to noise than we had expected. Indeed, despite the k-means ARI for the semi-cirle data set being quite close to 0 (quite random identification), our spectral clustering algorithm still fails to beat it when some noise is introduced, as shown in figure 8. Only for the concentric circles data set, where the k-means identification is truly as good as random, does our algorithm beat k-means (figure 6). Nevertheless, our algorithm achieved high adjusted random indices for all data sets when little to no noise was introduced, as shown by the tables. This implies our implementation introduced sensitivity to noise to the spectral clustering algorithm.

References

[1] M. Fiedler, “Algebraic connectivity of graphs”, eng, Czechoslovak Mathematical Journal **23**, 298–305 (1973).

[2] A. Pothen, H. D. Simon, and K.-P. Liou, “Partitioning sparse matrices with eigenvectors of graphs”, SIAM Journal on Matrix Analysis and Applications **11**, 430–452 (1990).

[3] L. Hagen and A. Kahng, “New spectral methods for ratio cut partitioning and clustering”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **11**, 1074–1085 (1992).

[4] B. Mohar, “Laplace eigenvalues of graphs—a survey”, Discrete Mathematics **109**, 171–183 (1992).

[5] B. Mohar, “Some applications of laplace eigenvalues of graphs”, edited by G. Hahn and G. Sabidussi, 225–275 (1997).

[6] J. Shi and J. Malik, “Normalized cuts and image segmentation”, IEEE Transactions on Pattern Analysis and Machine Intelligence **22**, 888–905 (2000).

[7] M. Belkin and P. Niyogi, “Laplacian eigenmaps and spectral techniques for embedding and clustering”, **14**, edited by T. Dietterich, S. Becker, and Z. Ghahramani (2001).

[8] C. Ding, X. He, H. Zha, M. Gu, and H. Simon, “A min-max cut algorithm for graph partitioning and data clustering”, 107–114 (2001).

[9] M. Meilă and J. Shi, “A random walks view of spectral segmentation”, Proceedings of Machine Learning Research **R3**, edited by T. S. Richardson and T. S. Jaakkola, Reissued by PMLR on 31 March 2021., 203–208 (2001).

[10] U. von Luxburg, “A tutorial on spectral clustering”, Stat Comput **17**, 395–416 (2007).