CS5691
Pattern Recognition and Machine Learning
Assignment 1

PCA and *K*-means Clustering

Instructor: Prof. Arun Rajkumar
Student: Janmenjaya Panda

Release Date: 07/02/2024
Submission Deadline: 10/03/2024

Department of Computer Science & Engineering
Indian Institute of Technology Madras

Contents

1 Principal Component Analysis

Download the MNIST dataset from https://huggingface.co/datasets/mnist. Use a random set of 100 images chosen from each class (0 to 9) as your dataset.

1. Write a piece of code to run the PCA algorithm on this dataset. Show the images of the principal components that you obtain. How much of the variance in the dataset is explained by each of the principal components?

2. Reconstruct the dataset using different dimensional representations. How do these look like? If you had to pick a dimension d that can be used for a downstream task where you need to classify the digits correctly, what would you pick and why?

3. Write a piece of code to implement the Kernel PCA algorithm on this dataset. Use the following kernels :

   (a) $\kappa(x, y) = (1 + x^T y)^d$ for $d \in \{1, 2, 3\}$

   (b) $\kappa(x, y) = \exp\left\{\frac{-(x-y)^T(x-y)}{2\sigma^2}\right\}$ for various choices of σ that you think are reasonable.

   Plot the projection of each point in the dataset onto the top 2 components for each kernel. Use one plot for each kernel and in the case of (b), use a different plot for each value of σ.

4. Which Kernel do you think is best suited for this dataset and why?

We downloaded the MNIST dataset and a sample image is shown below:

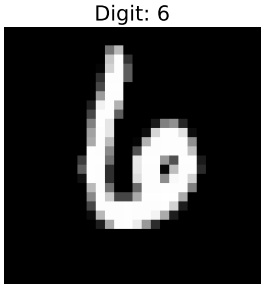


Figure 1: A random MNIST image

1. The following code explains the PCA algorithm implemented.

```
'''
Principal Component Analysis (PCA) Implementation
'''

import numpy as np

class PCA:
```

```
def __init__(self, n_components):
    # Attributes to store computed results
    self.X_fit = None
    self.n_components = n_components
    self.total_components = None
    self.components = None
    self.eigenvectors = None
    self.eigenvalues = None
    self.covariance_matrix = None
    self.mean = None
    self.std = None
    self.original_shape = None
    self.explained_variance_ratio = None
    self.cumulative_variance_ratio = None

def _centralize_data_(self, X):
    # Center and normalize input data
    mean = np.mean(X, axis=0)
    std = np.std(X, axis=0)
    centered_X = (X - mean)
    normalized_X = (X - mean) / (std + 1e-10)  # Add a small value to avoid division
    ↪ by zero
    return centered_X, mean, std

def _compute_covariance_matrix_(self, normalized_X):
    # Compute the covariance matrix
    return np.cov(normalized_X, rowvar=False)

def _compute_eigenvalues_and_vectors_(self, covariance_matrix):
    # Compute eigenvalues and eigenvectors
    eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)
    sorted_indices = np.argsort(eigenvalues)[::-1]
    return eigenvalues[sorted_indices], eigenvectors[:, sorted_indices]

def _select_top_eigenvectors_(self, eigenvectors):
    # Select the top n_components eigenvectors
    return eigenvectors[:, :self.n_components]

def fit_transform_(self, X):
    # Capture X_fit
    self.X_fit = X

    # Capture the total components
    self.total_components = self.X_fit[0].size

    # Normalize the data
    centered_X, self.mean, self.std = self._centralize_data_(self.X_fit)

    # Compute the covariance matrix
    self.covariance_matrix = self._compute_covariance_matrix_(centered_X)

    # Compute eigenvalues and eigenvectors
    self.eigenvalues, self.eigenvectors =
    ↪ self._compute_eigenvalues_and_vectors_(self.covariance_matrix)

    # Select the top n_components eigenvectors
```

```
        self.components = self._select_top_eigenvectors_(self.eigenvectors)

        # Project the data onto the new feature space
        transformed_X = centered_X.dot(self.components)

        return transformed_X

    def inverse_transform_(self, X_transformed):
        if self.X_fit is None:
            raise ValueError("Fit the model before attempting to inverse transform.")

        # Project the X back to the original space using components up to i
        reconstructed_X = X_transformed @ self.components.T

        # Denormalize the reconstructed data
        reconstructed_X = reconstructed_X + self.mean

        return reconstructed_X

    def explained_variance_ratio_(self):
        # Compute and return the explained variance ratio
        total_variance = np.sum(self.eigenvalues)
        self.explained_variance_ratio = self.eigenvalues / total_variance
        return self.explained_variance_ratio[:self.n_components]

    def cumulative_variance_ratio_(self):
        # Compute and return the cumulative variance ratio
        self.explained_variance_ratio_()
        self.cumulative_variance_ratio = np.cumsum(self.explained_variance_ratio)
        return self.cumulative_variance_ratio[:self.n_components]
```

The images of the principal components are shown below in a cumulative manner:

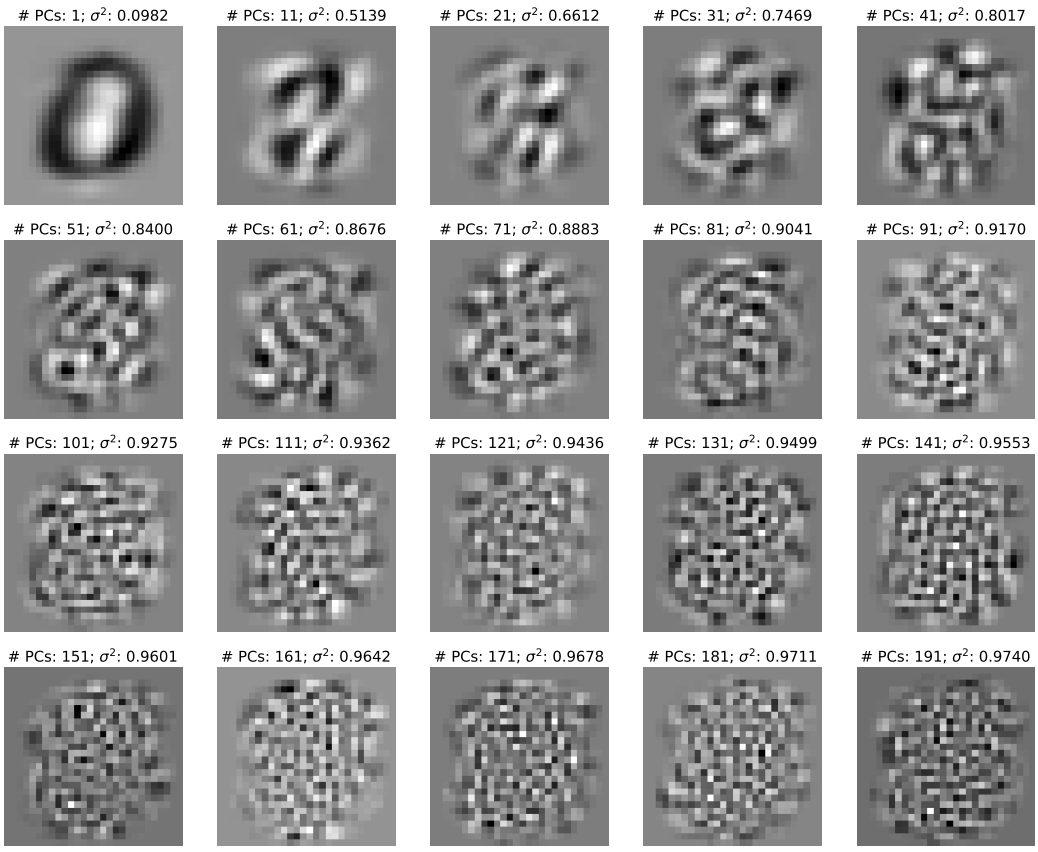


Figure 2: Images of the first 20 principal components

The cumulative variance ratio explained by each of the components is captured in the following plot. As it can be observed, the initial few principal components capture the majority of the details and as we keep on adding more and more components it follows a diminishing return.

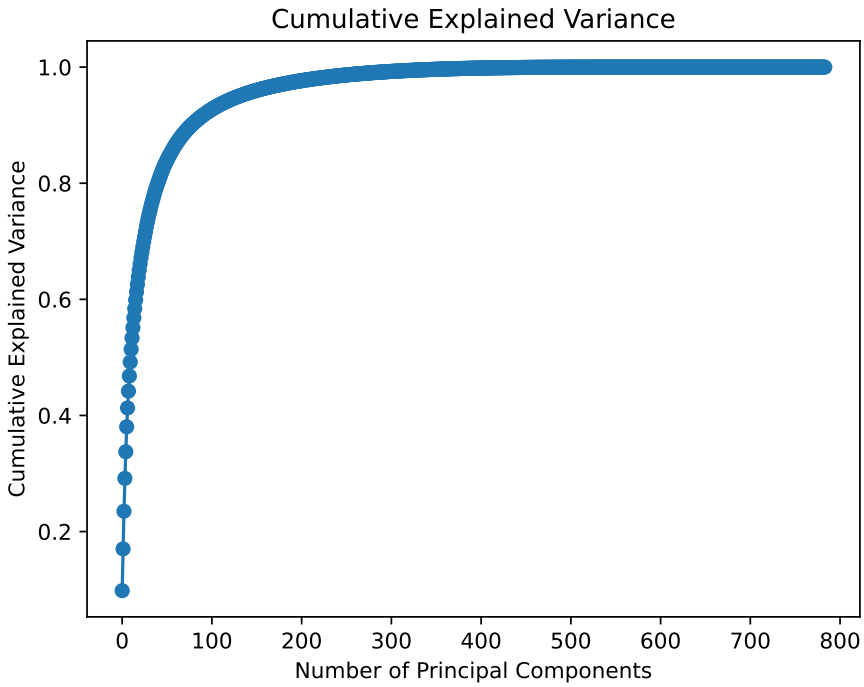


Figure 3: Cumulative Explained Variance

The following one shows the exact variance explained by each of the principal components.

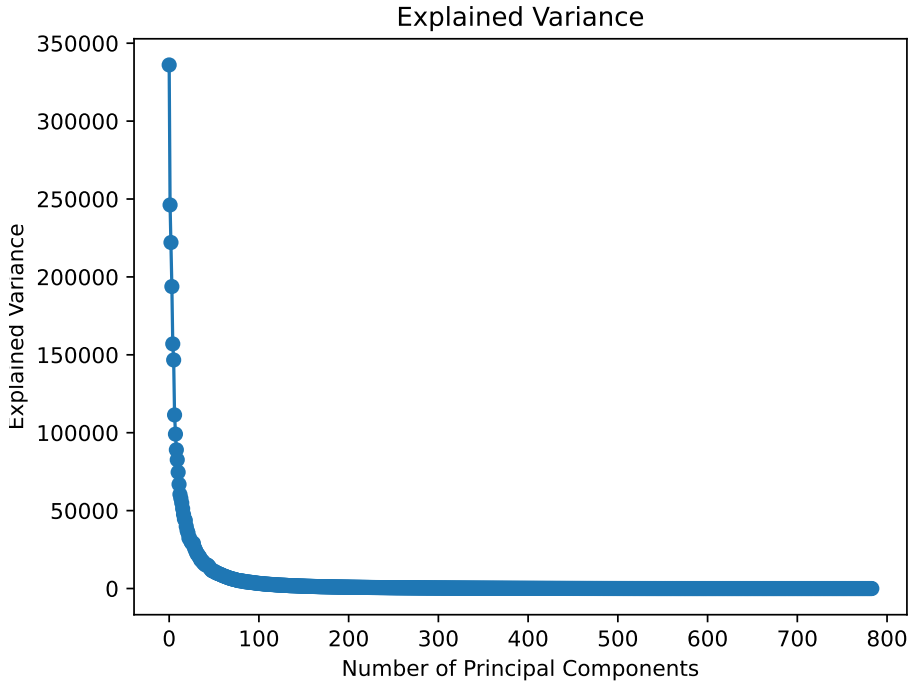


Figure 4: Cumulative Explained Variance

Principal Component	Individual Variance	Cumulative Variance Ratio
1	336037.4646	0.0982
2	246187.9788	0.1701
3	222096.0762	0.2350
4	193820.8358	0.2916
5	157019.4640	0.3375
10	82636.3438	0.4921
20	39909.0109	0.6504
50	11121.3021	0.8368
100	3341.8100	0.9266
250	532.4986	0.9864
500	17.0491	0.9999
750	0.0000	1.0000
784	0.0000	1.0000

Table 1: PCA and variance

The tabular representation:

2. We have reconstructed the datapoint shown in Figure 1 using various sets of principal components.

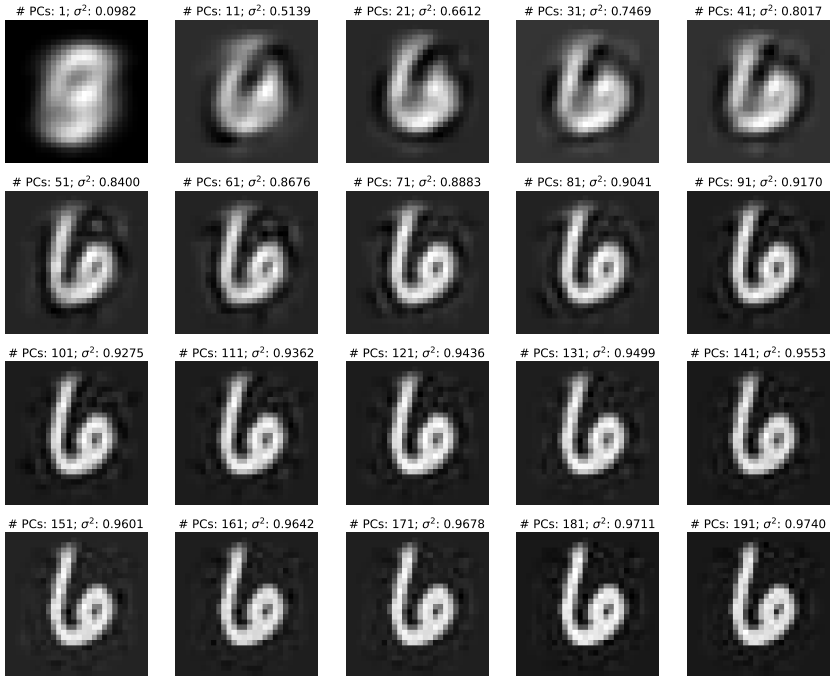
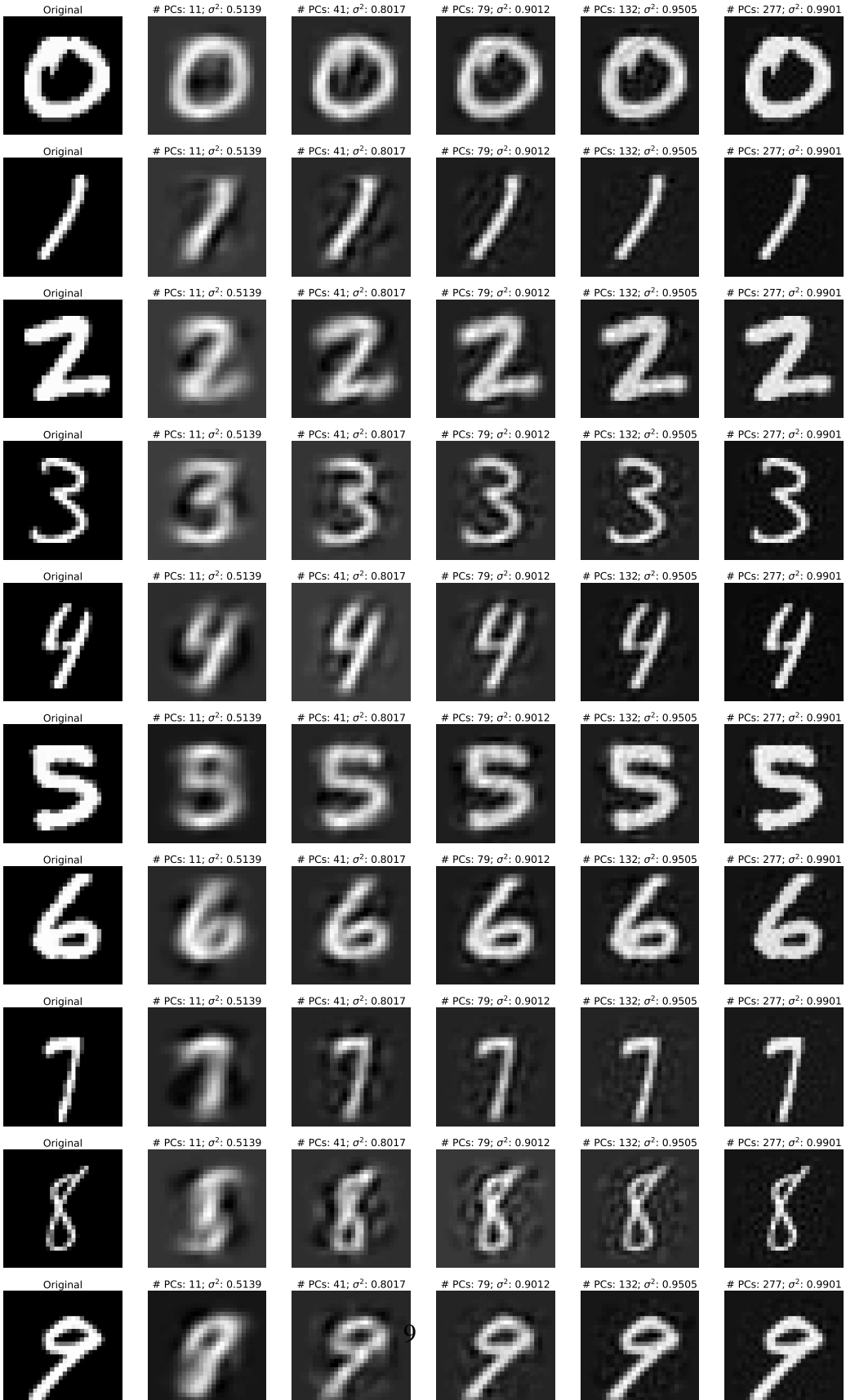


Figure 5: PCA performed on the datapoint shown in Figure 1

Clearly, as we can observe, the PCA with 81 components achieves a reasonably good result capturing $\geqslant 0.9$ of total variance. But, we shall also check with different digits to make sure we choose the number of components that is unbiased to the specific data. Here we have plotted several datapoints, one from each label with multiple candidate # of Components.


Original
PCs: 11; σ^2: 0.5139
PCs: 41; σ^2: 0.8017
PCs: 79; σ^2: 0.9012
PCs: 132; σ^2: 0.9505
PCs: 277; σ^2: 0.9901

Using binary search, we observed that 0.95% of the variance is captured by 132 # of PCs. This is an industry-standard and the reader may observe that PCA with 132 components generates a very analogous image to the original one.

3. The following code implements the Kernel PCA for several kernels. This class comes as an extension to the earlier PCA Class.

```
class KernelPCA(PCA):
    def __init__(self, n_components, kernel=None, gamma=None, degree=1, const_coeff=0):
        super().__init__(n_components)
        self.kernel = kernel
        self.gamma = gamma
        self.degree = degree
        self.const_coeff = const_coeff
        self.K = None

    def _compute_kernel_matrix_(self, X, Y):
        if self.kernel == 'linear':
            return np.dot(X, Y.T)
        elif self.kernel == 'rbf':
            X_norm = np.sum(X**2, axis=1, keepdims=True)
            Y_norm = np.sum(Y**2, axis=1, keepdims=True)
            return np.exp(-self.gamma * (X_norm - 2 * np.dot(X, Y.T) + Y_norm.T))
        elif self.kernel == 'poly':
            return (self.gamma * np.dot(X, Y.T) + self.const_coeff) ** self.degree
        else:
            raise ValueError('Unsupported kernel type.')

    def _center_kernel_matrix_(self, K):
        # Compute row and column means
        row_means = np.mean(K, axis=1)
        col_means = np.mean(K, axis=0)

        # Compute the overall mean
        kernel_mean = np.mean(K)

        # Center the kernel matrix
        centered_K = K - row_means[:, np.newaxis] - col_means + kernel_mean

        return centered_K

    def fit_transform_(self, X):
        # Capture X_fit
        self.X_fit = X

        # Capture the total components
        self.total_components = self.X_fit[0].size

        # Normalize the data
        centered_X, self.mean, self.std = self._centralize_data_(self.X_fit)

        # Compute the kernel matrix
        self.kernel_matrix = self._compute_kernel_matrix_(centered_X, centered_X)

        # Center the kernel matrix
        centered_kernel_matrix = self._center_kernel_matrix_(self.kernel_matrix)
```

```
        # Compute eigenvalues and eigenvectors
        self.eigenvalues, self.eigenvectors =
         ↪  self._compute_eigenvalues_and_vectors_(centered_kernel_matrix)

        # Select the top n_components eigenvectors
        self.components = self._select_top_eigenvectors_(self.eigenvectors)

        # Project the data onto the new feature space
        transformed_X = centered_kernel_matrix @
         ↪  self.components/np.sqrt(self.eigenvalues[:self.n_components])

        return transformed_X

    def inverse_transform_(self, X_transformed):
        raise ValueError('This is kernelPCA.')
```

Here we have plotted the projection of each point in the dataset onto the top two components for each kernel.

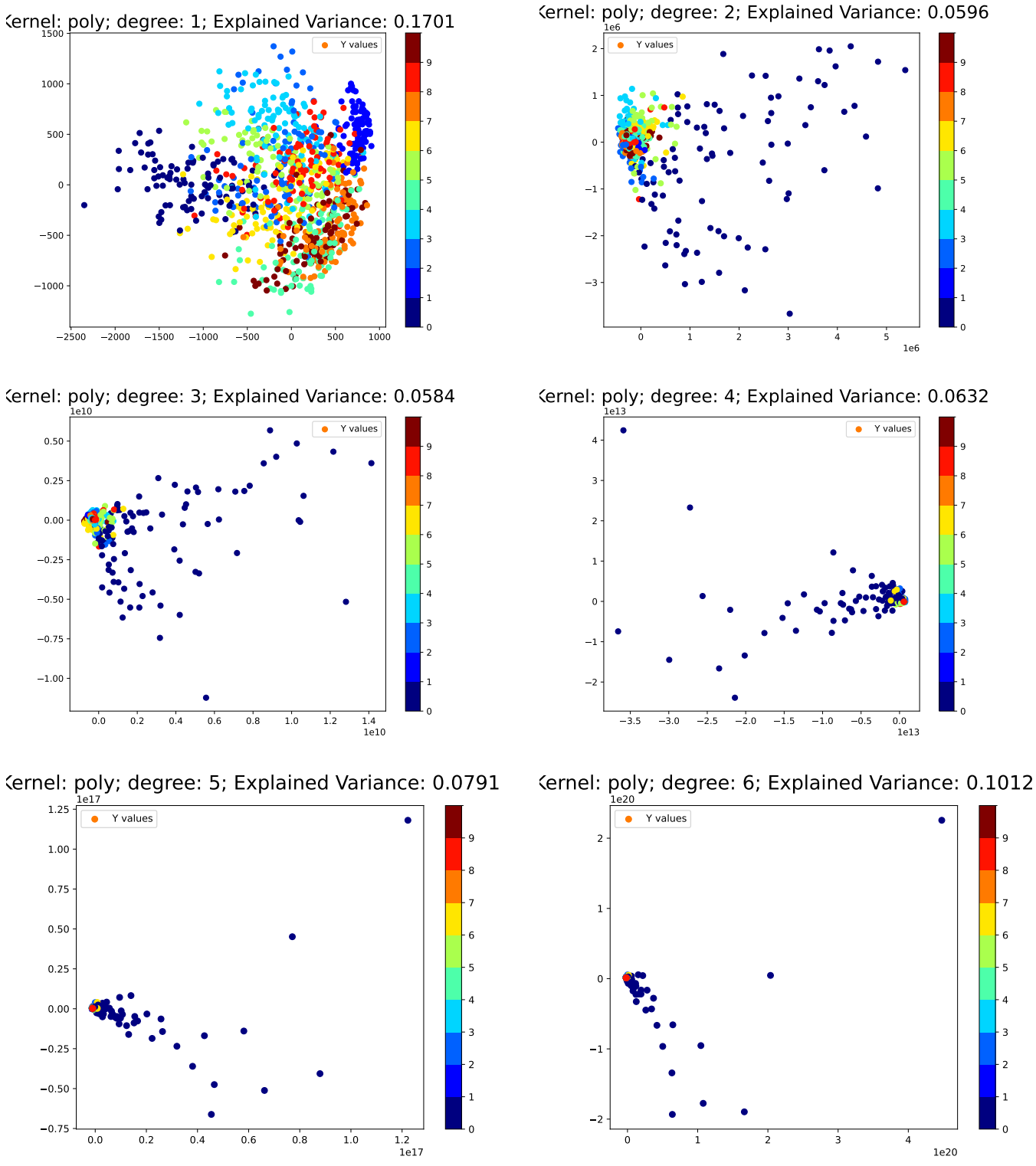


Table 2: Polynomial Kernel

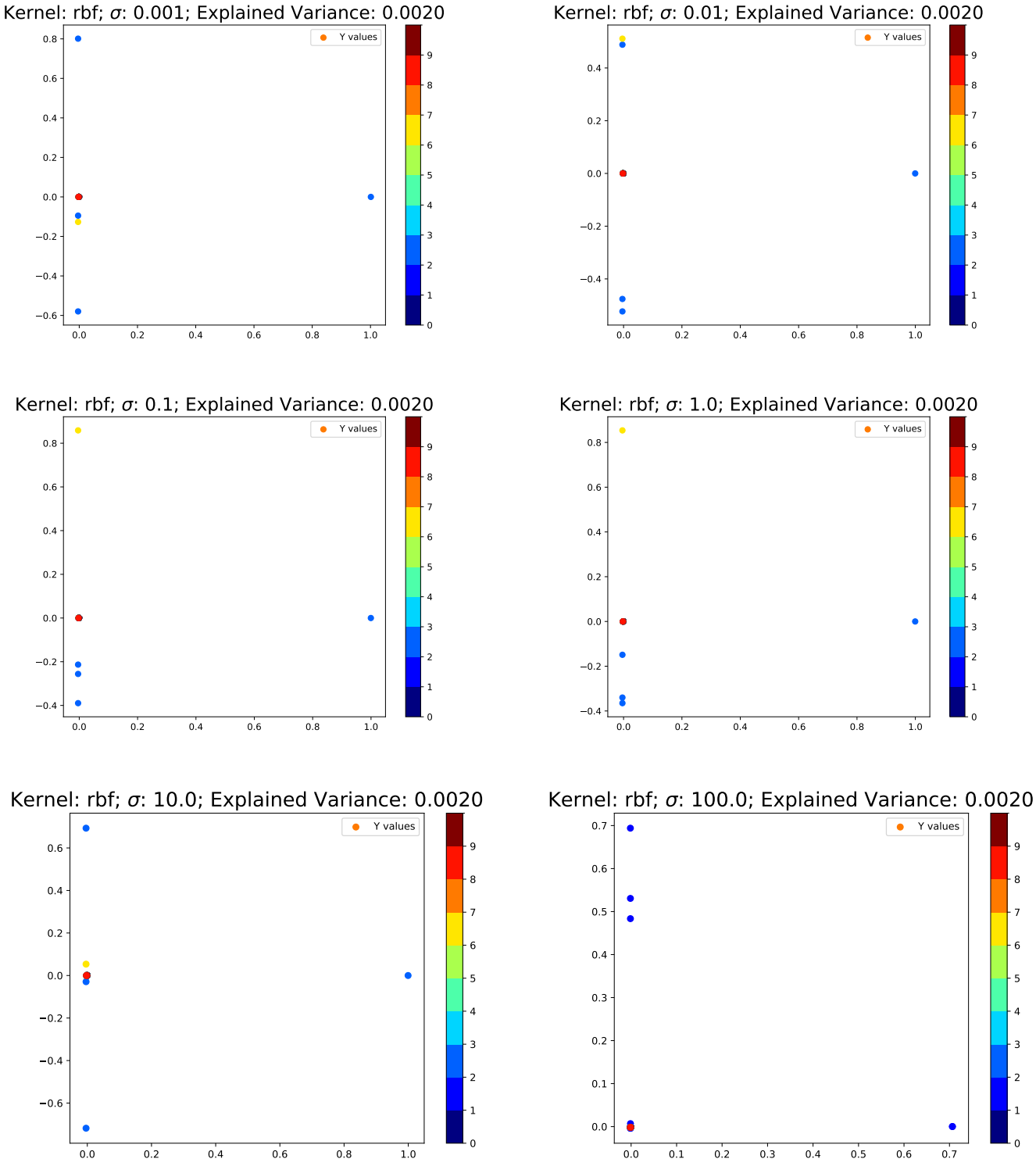


Table 3: Radial Basis Kernel

4. For the following two discussions, our consideration will be to compare the results of the previous question only using $K = 2$, that is – two principal components. For the MNIST dataset, which consists of handwritten digits, the choice of kernel in Kernel PCA can impact the results. However,

the reasons for the polynomial kernel with 2 principal components performing better than the RBF kernel may be related to the nature of the data and the characteristics of the kernels.
Here are some possible explanations specific to handwritten digits:

(a) Local Patterns in Digits: Handwritten digits often exhibit local patterns that may be better captured by a polynomial kernel with a lower degree. The polynomial kernel might be effective at capturing the shapes and contours of the digits in a more localized manner.

(b) Simplicity of Patterns: The simplicity of the digit patterns in MNIST may favor a simpler model like the polynomial kernel with a lower degree. RBF kernels are more flexible and may capture more complex relationships, but if the digit patterns are relatively simple, a simpler model may suffice.

(c) Effect of Dimensionality Reduction: Kernel PCA is a dimensionality reduction technique. In some cases, reducing the dimensionality too much (using only 2 principal components) with a complex kernel like RBF might result in loss of information, whereas a simpler polynomial kernel may be more effective for such a low-dimensional representation.

(d) Hyperparameter Settings: The specific hyperparameters chosen for the polynomial and RBF kernels can significantly influence their performance. It's possible that the hyperparameters for the polynomial kernel, such as the degree, were well-tuned for the MNIST dataset, leading to better results.

Also, we may observe that as we increase the degree of the polynomial the cumulative variance ratio also increases. It can be explained as follows:

(a) Capturing Higher-Order Relationships: A higher-degree polynomial kernel introduces higher-order terms in the feature space, allowing the algorithm to capture more complex relationships in the data. In the case of Kernel PCA, this means that the algorithm can represent more intricate patterns and non-linear structures present in the data.

(b) Flexibility of the Model: Higher-degree polynomials are more flexible and can fit more complex functions. This increased flexibility allows the model to adapt to intricate variations in the data, potentially resulting in a better fit and higher explained variance.

(c) Overfitting Considerations: While higher-degree polynomials provide greater flexibility, there is a risk of overfitting, especially when the degree is excessively large relative to the complexity of the underlying patterns in the data. Careful tuning of hyperparameters, including the degree of the polynomial kernel, is necessary to find the right balance between capturing complex relationships and avoiding overfitting.

(d) Data Complexity: The effectiveness of a higher-degree polynomial kernel also depends on the complexity of the underlying data. If the data contains intricate, non-linear relationships that can be better captured by higher-degree polynomials, then increasing the degree may lead to a better representation and higher explained variance.

Thus, we a for $K = 2$, a polynomial kernel with larger degree is a better choice.

2 $K-$means Clustering

You are given a data set with 1000 data points each in $\mathbb{R}^2$ (cm dataset 2.csv).

1. Write a piece of code to run the algorithm studied in class for the K-means problem with $K = 2$. Try 5 different random initialization and plot the error function w.r.t iterations in each case. In each case, plot the clusters obtained in different colours.

2. Fix a random initialization. For $K \in \{2, 3, 4, 5\}$, obtain cluster centers according to K-means algorithm using the fixed initialization. For each value of K, plot the Voronoi regions associated with each cluster center. (You can assume the minimum and maximum value in the dataset to be the range for each component of $\mathbb{R}^2$).

3. Run the spectral clustering algorithm (spectral relaxation of K-means using Kernel-PCA) $K = 2$. Choose an appropriate kernel for this data set and plot the clusters obtained in different colours. Explain your choice of kernel based on the output you obtain.

4. Instead of using the method suggested by spectral clustering to map eigenvectors to cluster assignments, use the following method: Assign data point i to cluster ℓ whenever

$$\ell = \arg\max_{j \in [K]} v_i^j$$

where $v^j \in \mathbb{R}^n$ is the eigenvector of the Kernel matrix associated with the j-th largest eigenvalue. How does this mapping perform for this dataset? Explain your insights.

---

1. The dataset is as follows:

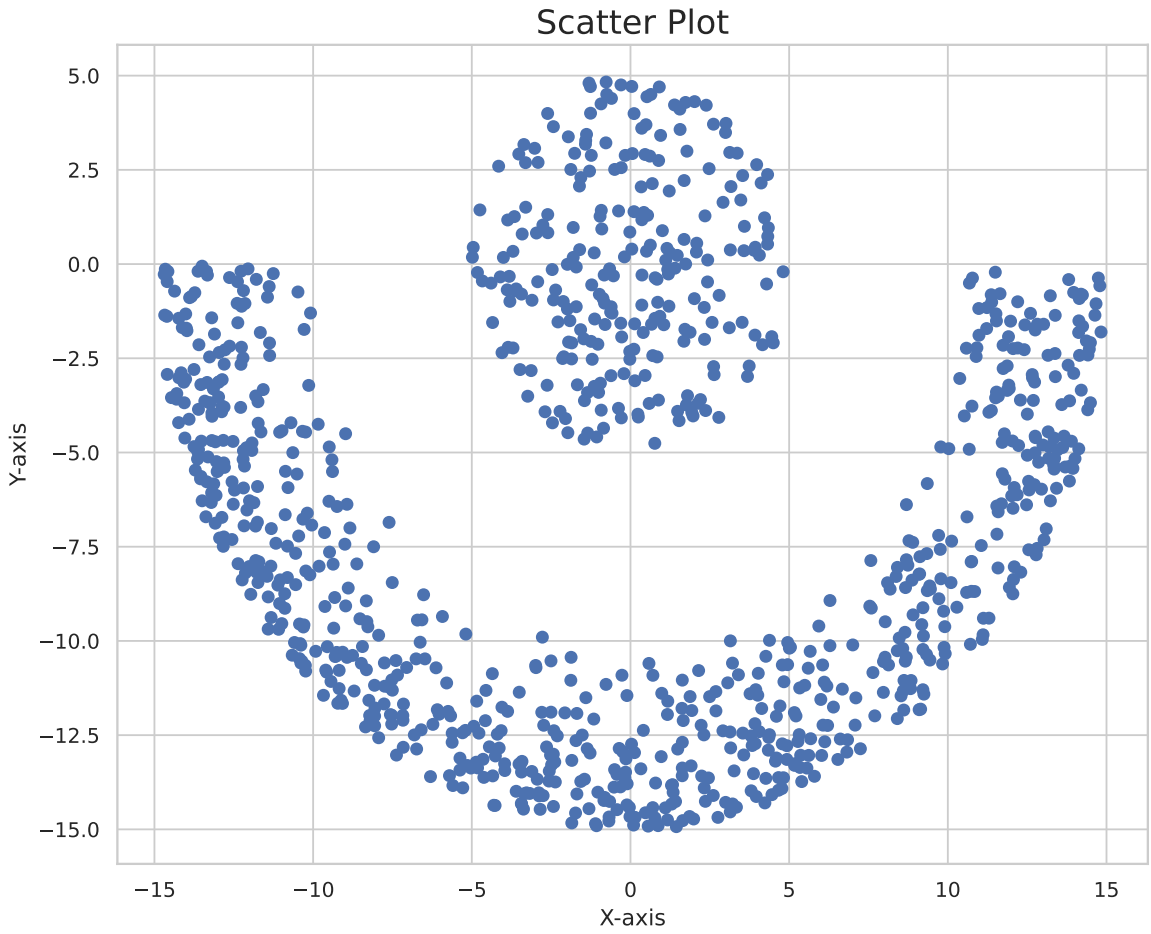


Figure 7: Dataset

The code can be found below:

```
class KMeans:
    def __init__(self, k=3, max_iters=500, tol=1e-5, k_means_plusplus=False):
        """
        Initialize KMeans instance.

        Parameters:
        - k: Number of clusters.
        - max_iters: Maximum number of iterations for k-means.
        - tol: Tolerance for convergence.
        - k_means_plusplus: Whether to use k-means++ initialization for centroids.
        """
        self.k = k
        self.max_iters = max_iters
        self.tol = tol
        self.k_means_plusplus = k_means_plusplus
        self.centroids = None
        self.labels = None
        self.n_samples = None
```

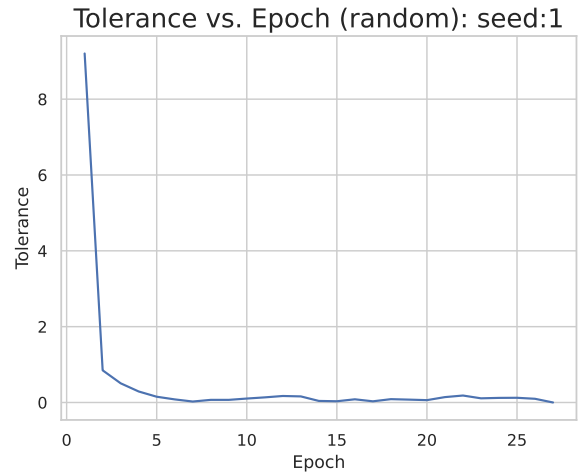
```
        self.n_features = None
        self.objective_value = None

    def fit(self, X):
        """
        Fit KMeans to the data.

        Parameters:
        - X: Input data.

        Returns:
        - labels: Cluster labels for each data point.
        - centroids: Final centroids.
        """
        self.n_samples, self.n_features = X.shape

        # Initialization of centroids
        if self.k_means_plusplus:
            self.centroids = self.k_means_plusplus_init(X)
        else:
            self.centroids = X[np.random.choice(self.n_samples, self.k, replace=False)]

        for _ in range(self.max_iters):
            # Assign each data point to the closest centroid
            distances = np.linalg.norm(X[:, np.newaxis] - self.centroids, axis=2)
            self.labels = np.argmin(distances, axis=1)

            # Update centroids based on the mean of points in each cluster
            new_centroids = np.array([X[self.labels == j].mean(axis=0) for j in
             ↪  range(self.k)])

            # Check for convergence
            if np.linalg.norm(new_centroids - self.centroids) < self.tol:
                break

            self.centroids = new_centroids

        self.labels = np.argmin(distances, axis=1)  # Assign cluster labels
        self.objective_value = self.calculate_objective_value(X)  # Calculate the
         ↪  objective value
        return self.labels, self.centroids  # Return the cluster labels for each data
         ↪  point

    def k_means_plusplus_init(self, X):
        """
        Initialize centroids using k-means++ algorithm.

        Parameters:
        - X: Input data.

        Returns:
        - centroids: Initialized centroids.
        """
        # Initialize the centroids
        self.centroids = np.zeros((self.k, self.n_features), dtype=X.dtype)
```

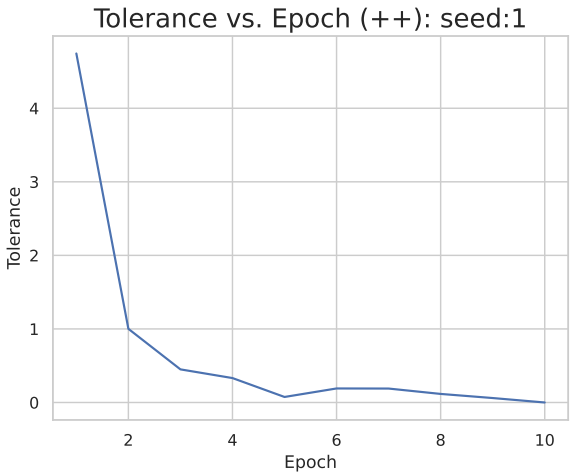
```
        # Randomly choose the first centroid
        self.centroids[0] = X[np.random.choice(self.n_samples)]

        # Choose the remaining centroids using the k-means++ initialization
        for i in range(1, self.k):
            min_distances = np.full((self.n_samples,), np.inf)

            for j in range(i):
                distances = np.linalg.norm(X - self.centroids[j], axis=1)
                min_distances = np.minimum(min_distances, distances)

            # Handle zero min_distances
            if np.all(min_distances == 0):
                # If all distances are zero, select a random point as the next centroid
                self.centroids[i] = X[np.random.choice(self.n_samples)]
            else:
                # Handle NaN values in min_distances
                min_distances[np.isnan(min_distances)] = 0

                # Check for NaN in probabilities
                if np.any(np.isnan(min_distances)) or np.any(min_distances == 0):
                    # If there are NaN values or all distances are zero, select a random
                    ↪ point
                    self.centroids[i] = X[np.random.choice(self.n_samples)]
                else:
                    # Calculate probabilities and choose the next centroid
                    probabilities = min_distances**2 / np.sum(min_distances**2)
                    probabilities[np.isnan(probabilities)] = 0  # Handle NaN in
                    ↪ probabilities
                    self.centroids[i] = X[np.random.choice(self.n_samples,
                    ↪ p=probabilities)]

        return self.centroids

    def plot_clusters(self, X):
        """
        Plot the clusters along with centroids.

        Parameters:
        - X: Input data.
        """
        plt.scatter(X[:, 0], X[:, 1], c=self.labels, cmap='viridis', alpha=0.5)
        plt.scatter(self.centroids[:, 0], self.centroids[:, 1], marker='X', s=200,
        ↪ c='red', label='Centroids')
        h = 0.02
        x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
        y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
        xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
        Z = self.predict(np.c_[xx.ravel(), yy.ravel()])  # Use predict method to get
        ↪ cluster labels for mesh grid
        Z = Z.reshape(xx.shape)
        plt.contour(xx, yy, Z, colors='k', linewidths=2, alpha=1)
        if self.k_means_plusplus == False:
            plt.title(f'K-means Clustering (K = {self.k})', fontsize=18)
        else:
            plt.title(f'K-means Clustering++ (K = {self.k})', fontsize=18)
```

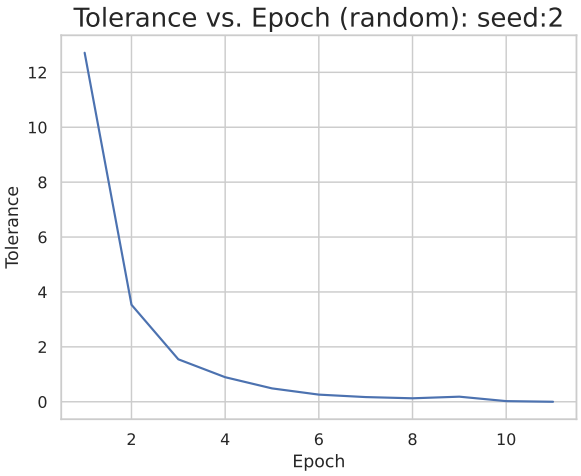
```
        plt.xlabel('Feature 1')
        plt.ylabel('Feature 2')
        plt.legend()
        if self.k_means_plusplus == False:
            plt.title(f'K-means_Clustering_({self.k})', fontsize=18)
        else:
            plt.title(f'K-means_Clustering_plusplus_({self.k}).pdf', fontsize=18)
        plt.show()

    def predict(self, X):
        """
        Predict cluster labels for new data points.

        Parameters:
        - X: New data points.

        Returns:
        - labels: Cluster labels for the new data points.
        """
        distances = np.linalg.norm(X - self.centroids[:, np.newaxis], axis=2)
        return np.argmin(distances, axis=0)

    def calculate_objective_value(self, X):
        """
        Calculate the objective value (sum of squared distances).

        Parameters:
        - X: Input data.

        Returns:
        - objective_value: Objective value.
        """
        distances = np.linalg.norm(X - self.centroids[self.labels], axis=1)
        return np.sum(distances**2)

    def calculate_aic(self, X):
        """
        Calculate the Akaike Information Criterion (AIC).

        Parameters:
        - X: Input data.

        Returns:
        - aic: AIC value.
        """
        k = self.k
        n = X.shape[0]
        obj_value = self.objective_value
        aic = 2 * k + n * np.log(obj_value / n)
        return aic

    def calculate_bic(self, X):
        """
        Calculate the Bayesian Information Criterion (BIC).

        Parameters:
```

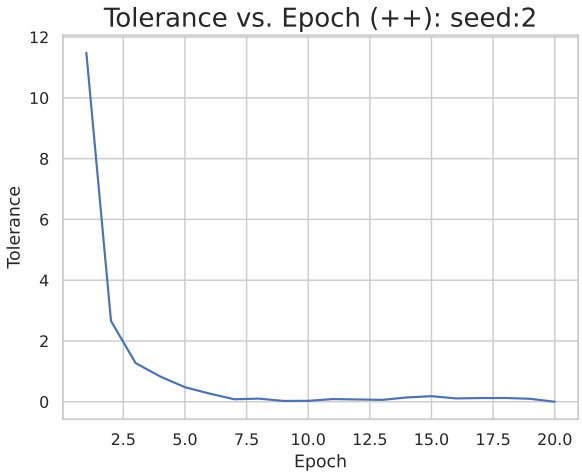
```
        - X: Input data.

        Returns:
        - bic: BIC value.
        """
        k = self.k
        n = X.shape[0]
        obj_value = self.objective_value
        bic = k * np.log(n) + n * np.log(obj_value / n)
        return bic
    def plot_tolerance_vs_epoch(self, X):
        """
        Plot tolerance vs. epoch for a single K-means initialization.

        Parameters:
        - X: Input data.
        """
        self.centroids = None
        self.labels = None
        self.objective_value = None

        # Initialization of centroids
        if self.k_means_plusplus:
            self.centroids = self.k_means_plusplus_init(X)
        else:
            self.centroids = X[np.random.choice(self.n_samples, self.k, replace=False)]

        tolerance_values = []

        for epoch in range(self.max_iters):
            # Assign each data point to the closest centroid
            distances = np.linalg.norm(X[:, np.newaxis] - self.centroids, axis=2)
            self.labels = np.argmin(distances, axis=1)

            # Update centroids based on the mean of points in each cluster
            new_centroids = np.array([X[self.labels == j].mean(axis=0) for j in
             ↪  range(self.k)])

            # Check for convergence
            tolerance = np.linalg.norm(new_centroids - self.centroids)
            tolerance_values.append(tolerance)
            if tolerance < self.tol:
                break

            self.centroids = new_centroids

        # Plotting tolerance vs. epoch
        plt.plot(range(1, len(tolerance_values) + 1), tolerance_values)
        plt.xlabel('Epoch')
        plt.ylabel('Tolerance')
        plt.title('Tolerance vs. Epoch for K-means Clustering')
        plt.show()
```

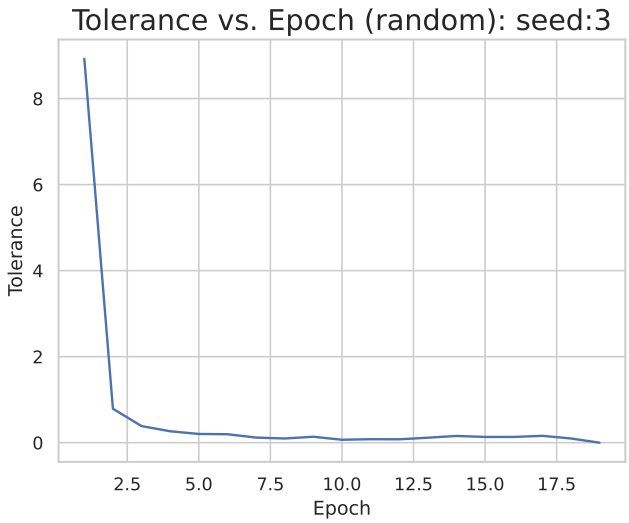
Tolerance vs epoch plot:











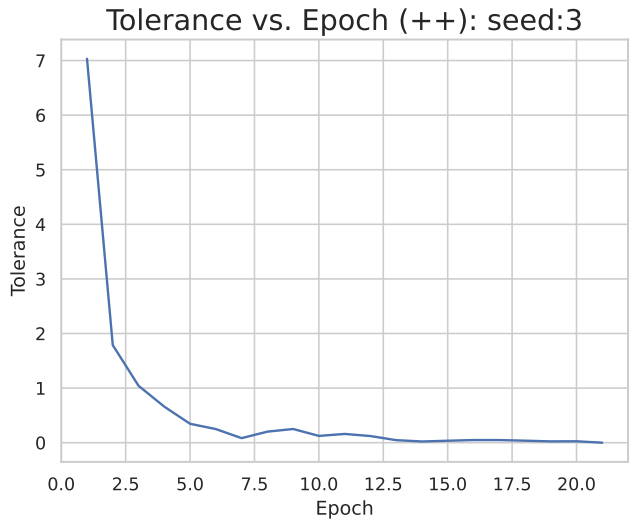


Table 4: Tolerance vs epoch Part I

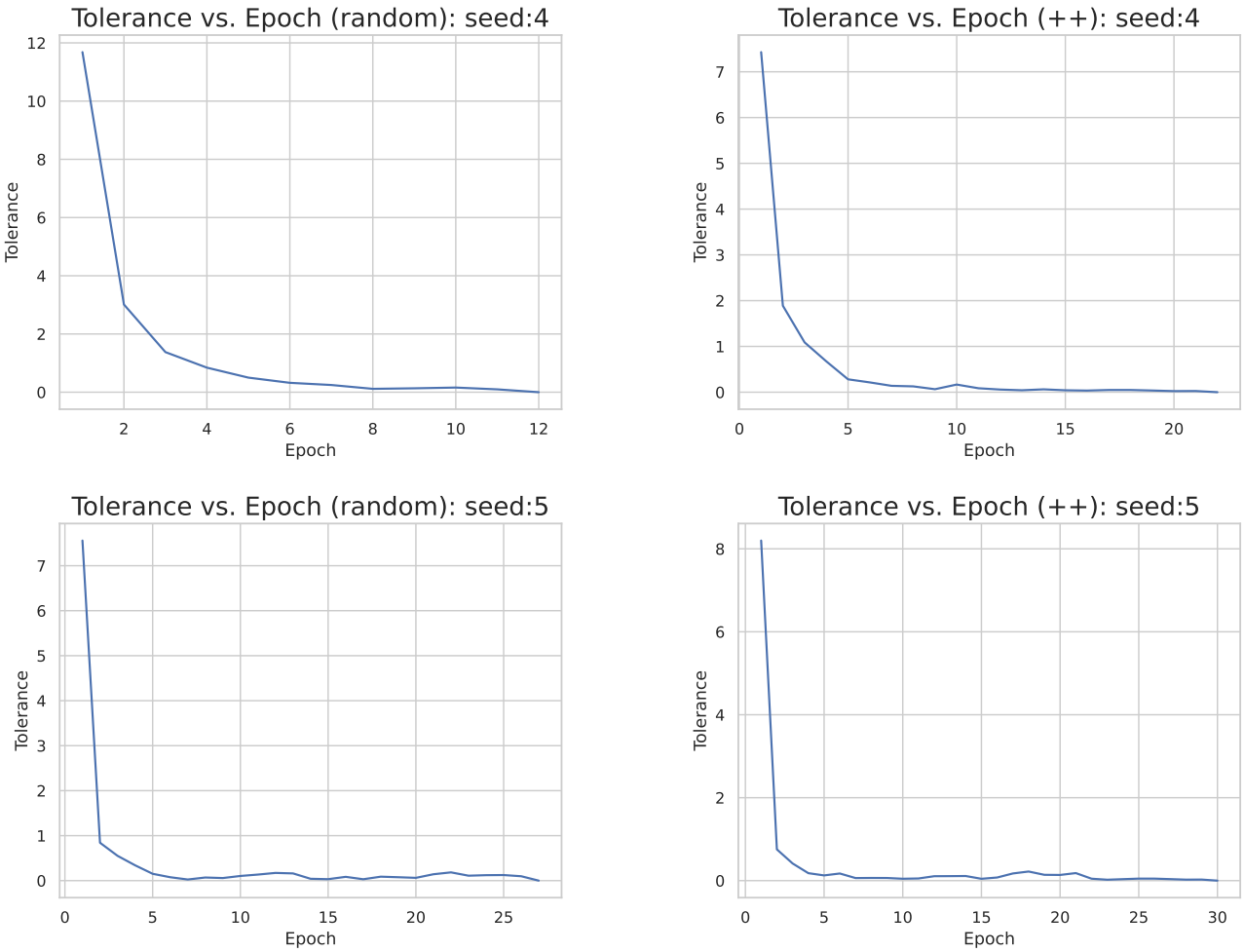


Table 5: Tolerance vs epoch: Part II

2. The plot goes as follows:

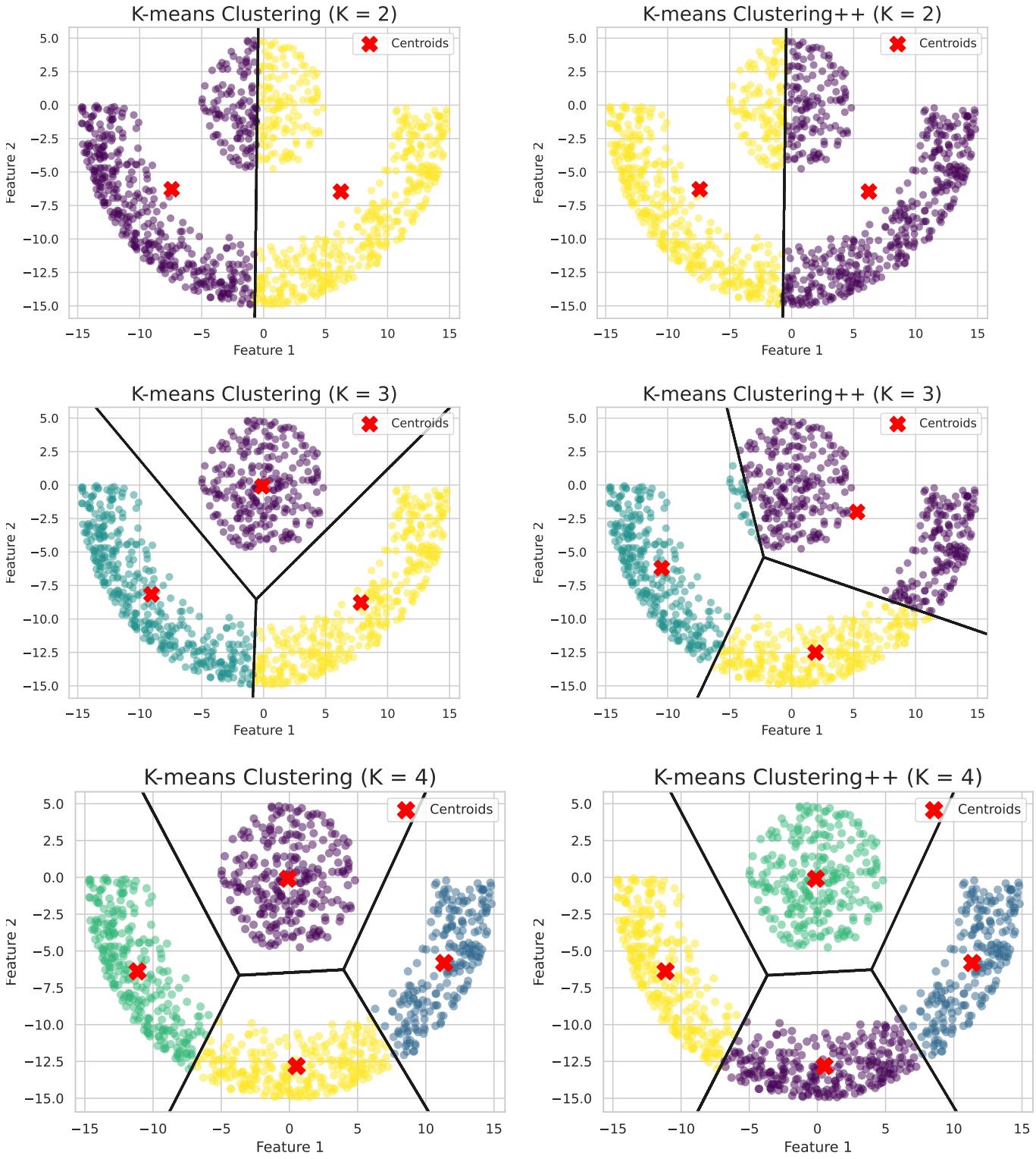


Table 6: K means and K means++ Clustering for Various values of K: Part I

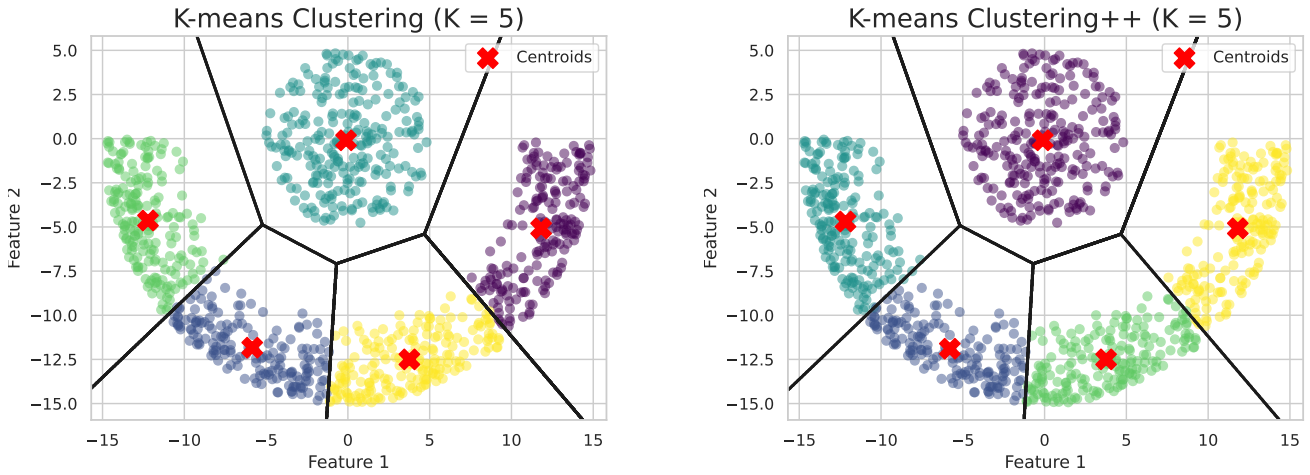
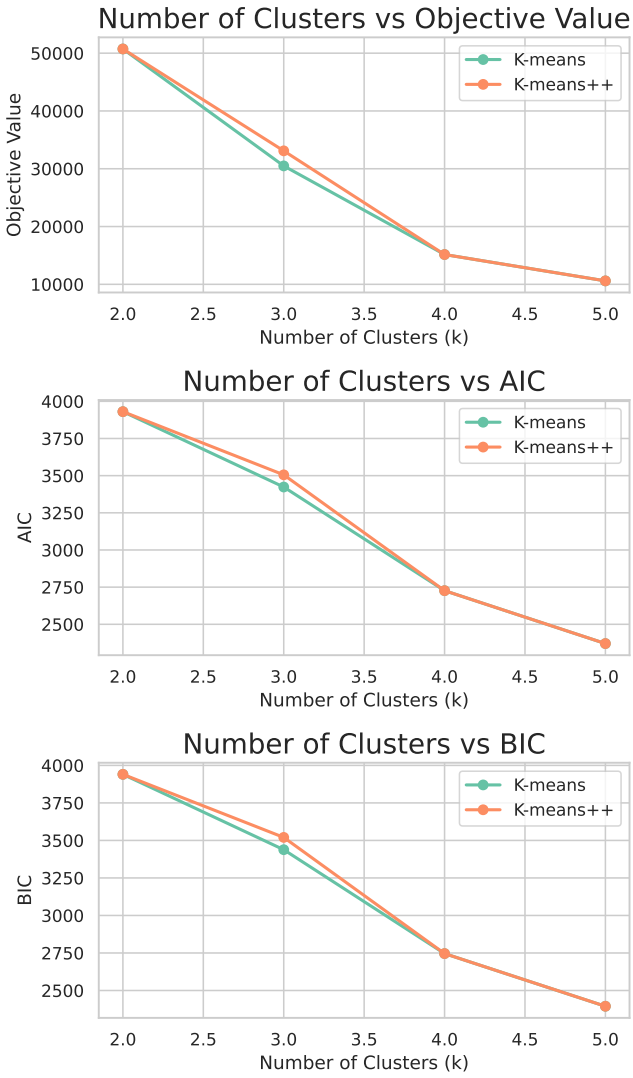


Table 7: K means and K means++ Clustering for Various values of K: Part II

These are the plots for objective value, AIC and BIC.



3. The Spectral Clustering code goes as follows:

```
import numpy as np
import matplotlib.pyplot as plt

class SpectralClustering(KMeans):
    def __init__(self, k=3, kernel='rbf', degree=None, gamma=None, const_coeff=1,
    ↪ k_means_plusplus=False):
        """
        Initialize SpectralClustering instance.

        Parameters:
        - k: Number of clusters.
        - kernel: Type of kernel ('linear', 'rbf', 'poly').
        - degree: Degree for polynomial kernel (if kernel='poly').
        - gamma: Gamma parameter for RBF kernel (if kernel='rbf').
        - const_coeff: Constant coefficient for polynomial kernel (if kernel='poly').
        - k_means_plusplus: Whether to use k-means++ initialization for centroids.
        """
        super().__init__(k=k, k_means_plusplus=k_means_plusplus)
        self.degree = degree
        self.gamma = gamma
        self.const_coeff = const_coeff
        self.kernel = kernel
        self.eigenvalues = None
        self.eigenvectors = None
        self.K = None

    def _compute_kernel_matrix_(self, X, Y):
        """
        Compute the kernel matrix based on the chosen kernel.

        Parameters:
        - X, Y: Input data.

        Returns:
        - kernel_matrix: Computed kernel matrix.
        """
        if self.kernel == 'linear':
            return np.dot(X, Y.T)
        elif self.kernel == 'rbf':
            X_norm = np.sum(X**2, axis=1, keepdims=True)
            Y_norm = np.sum(Y**2, axis=1, keepdims=True)
            return np.exp((-1) * self.gamma * (X_norm - 2 * np.dot(X, Y.T) + Y_norm.T))
        elif self.kernel == 'poly':
            return (np.dot(X, Y.T) + self.const_coeff) ** self.degree
        else:
            raise ValueError('Unsupported kernel type.')

    def _compute_eigenvalues_and_vectors_(self, covariance_matrix):
        """
        Compute eigenvalues and eigenvectors of a given covariance matrix.

        Parameters:
        - covariance_matrix: Covariance matrix.

        Returns:
```

```
        - eigenvalues: Computed eigenvalues.
        - eigenvectors: Computed eigenvectors.
        """
        eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)
        sorted_indices = np.argsort(eigenvalues)[::-1]
        return eigenvalues[sorted_indices], eigenvectors[:, sorted_indices]

    def _select_top_eigenvectors_(self, eigenvectors):
        """
        Select the top 'k' eigenvectors.

        Parameters:
        - eigenvectors: All eigenvectors.

        Returns:
        - selected_eigenvectors: Top 'k' eigenvectors.
        """
        return eigenvectors[:, :self.k]

    def fit(self, X):
        """
        Fit SpectralClustering to the data.

        Parameters:
        - X: Input data.

        Returns:
        - labels: Cluster labels for each data point.
        - centroids: Final centroids.
        """
        # Compute the kernel matrix
        self.kernel_matrix = self._compute_kernel_matrix_(X, X)

        # Compute eigenvalues and eigenvectors
        self.eigenvalues, self.eigenvectors =
        ↪  self._compute_eigenvalues_and_vectors_(self.kernel_matrix)

        # Form the matrix H
        H = self._select_top_eigenvectors_(self.eigenvectors)

        # Normalize each row of H to have unit length
        normalized_H = H / np.linalg.norm(H, axis=1, keepdims=True)

        # Cluster the rows of H using K-means
        self.labels, self.centroids = super().fit(normalized_H)

    def plot_clusters(self, X):
        """
        Plot the clusters along with additional information.

        Parameters:
        - X: Input data.
        """
        plt.scatter(X[:, 0], X[:, 1], c=self.labels, cmap='viridis', alpha=0.5)
        plt.legend()
        if self.gamma is None:
```

```
            if not self.k_means_plusplus:
                plt.title(f'Spectral Clustering (random): Kernel: {self.kernel}; degree:
                 ↪ {self.degree}', fontsize=18)
                plt.savefig(f'images/problem-2/Spectral Clustering (random): Kernel:
                 ↪ {self.kernel}; degree: {self.degree}.pdf')
            else:
                plt.title(f'Spectral Clustering (++): Kernel: {self.kernel}; degree:
                 ↪ {self.degree}', fontsize=18)
                plt.savefig(f'images/problem-2/Spectral Clustering (++): Kernel:
                 ↪ {self.kernel}; degree: {self.degree}.pdf')
        else:
            if not self.k_means_plusplus:
                plt.title(f'Spectral Clustering (random): Kernel: {self.kernel}; sigma:
                 ↪ {np.sqrt(1/(2*self.gamma))}', fontsize=18)
                plt.savefig(f'images/problem-2/Spectral Clustering (random): Kernel:
                 ↪ {self.kernel}; sigma: {np.sqrt(1/(2*self.gamma))}.pdf')
            else:
                plt.title(f'Spectral Clustering (++): Kernel: {self.kernel}; sigma:
                 ↪ {np.sqrt(1/(2*self.gamma))}', fontsize=18)
                plt.savefig(f'images/problem-2/Spectral Clustering (random): Kernel:
                 ↪ {self.kernel}; sigma: {np.sqrt(1/(2*self.gamma))}.pdf')
        plt.show()
```

The Spectral Clustering plots:

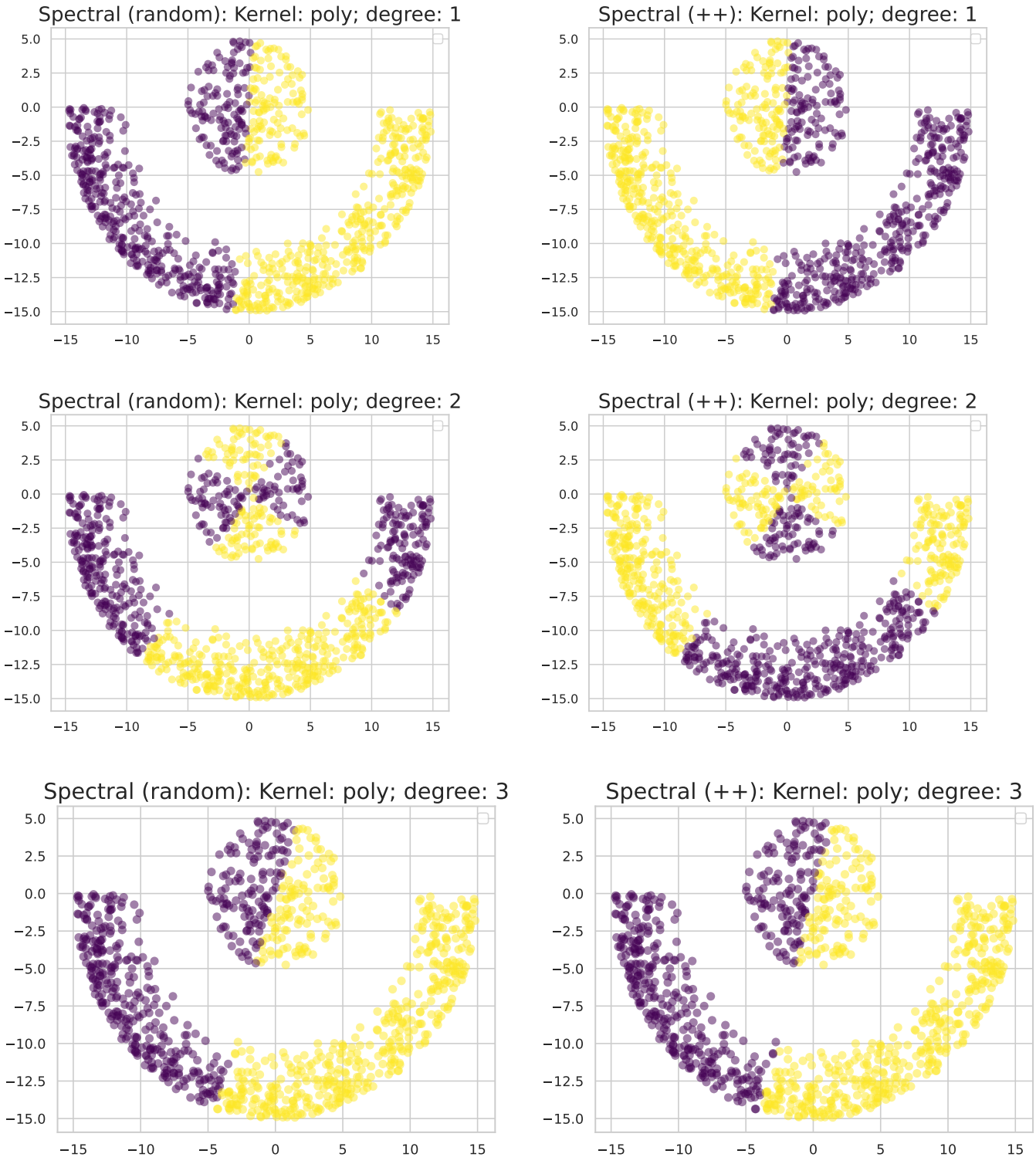


Table 8: K means and K means++ Clustering Using Spectral Clustering for Various values of K using polynomial kernel: Part I

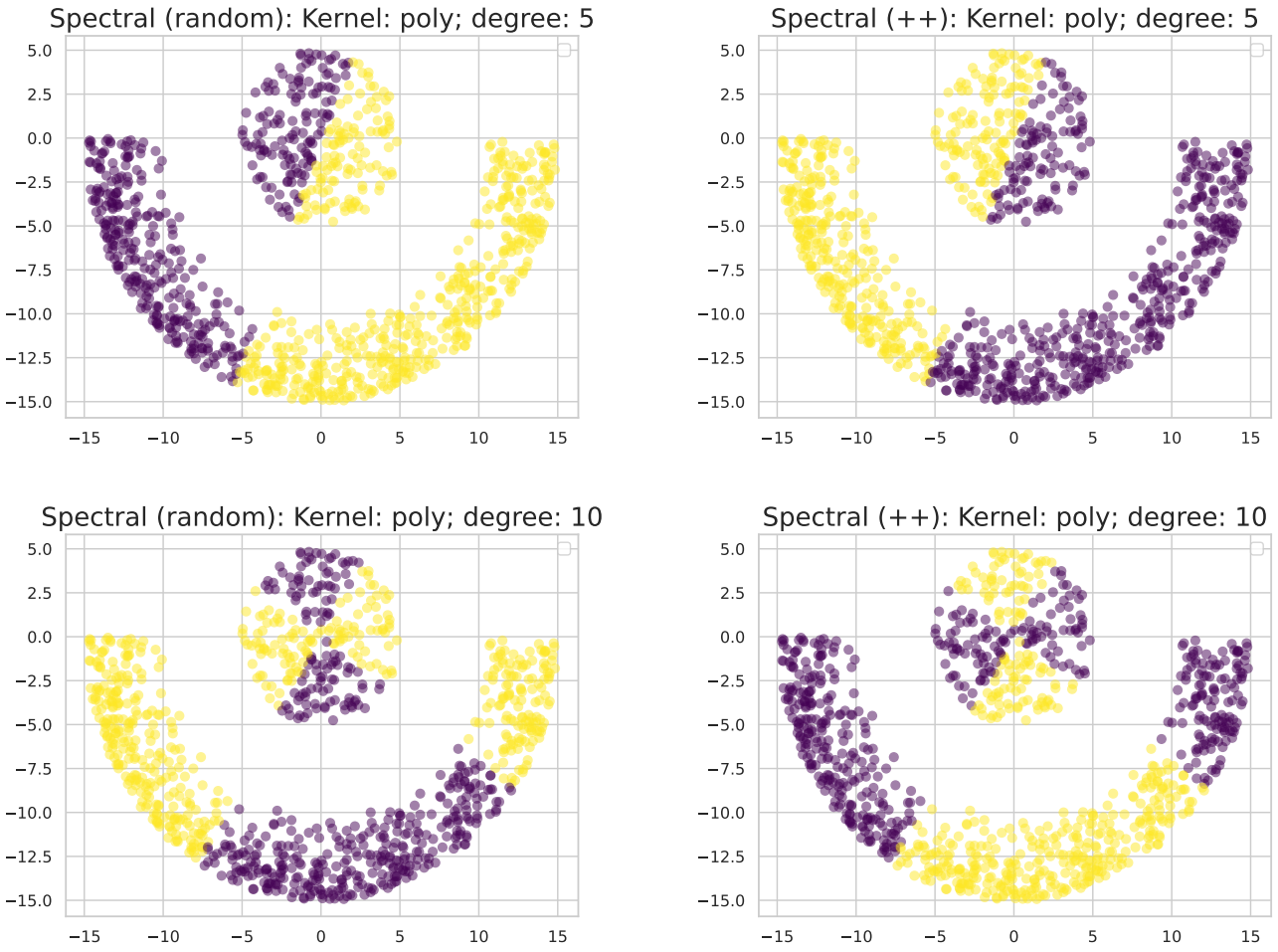


Table 9: K means and K means++ Clustering Using Spectral Clustering for Various values of K using polynomial Kernel: Part II

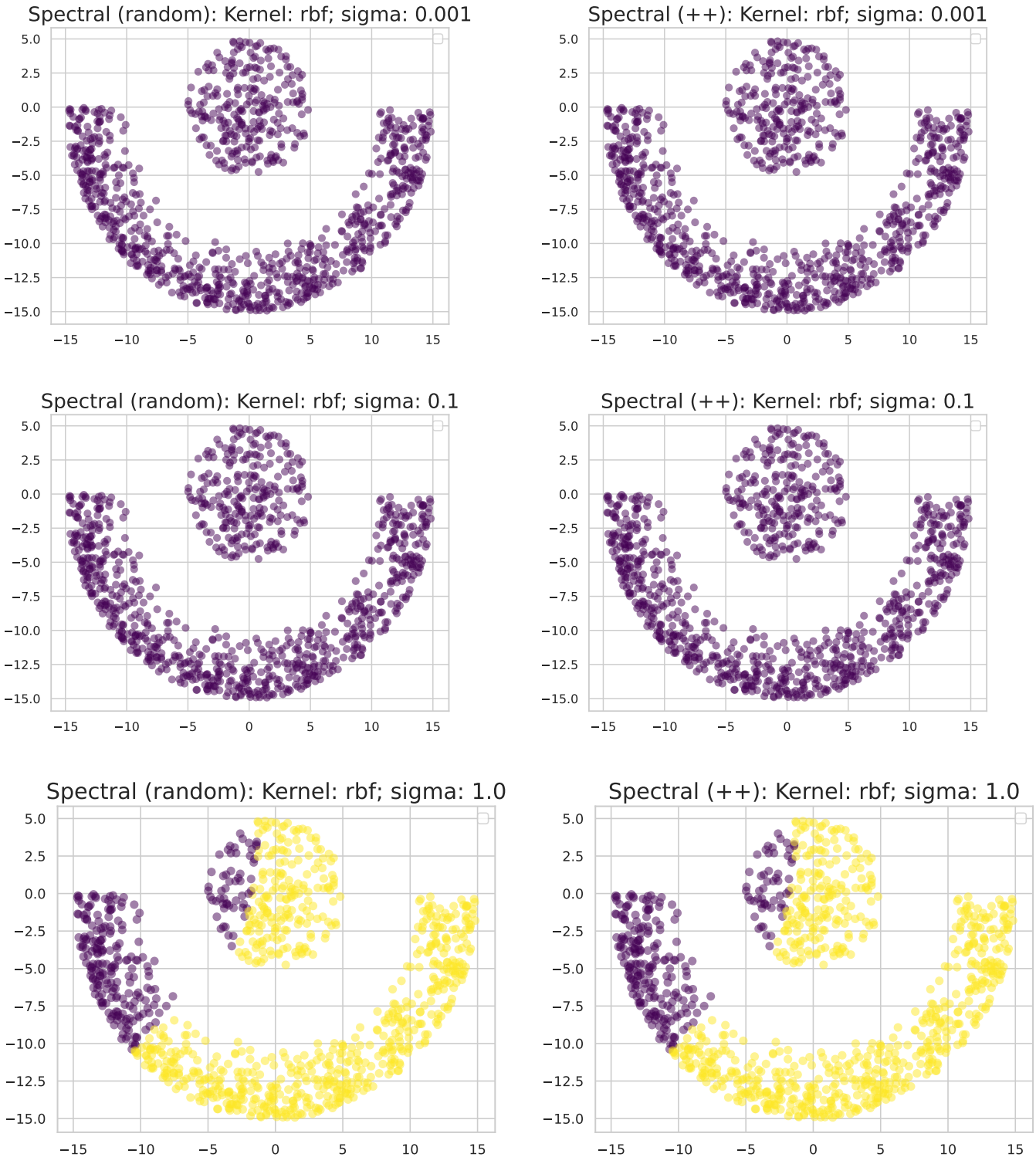


Table 10: K means and K means++ Clustering Using Spectral Clustering for Various values of K using RBF kernel: Part I

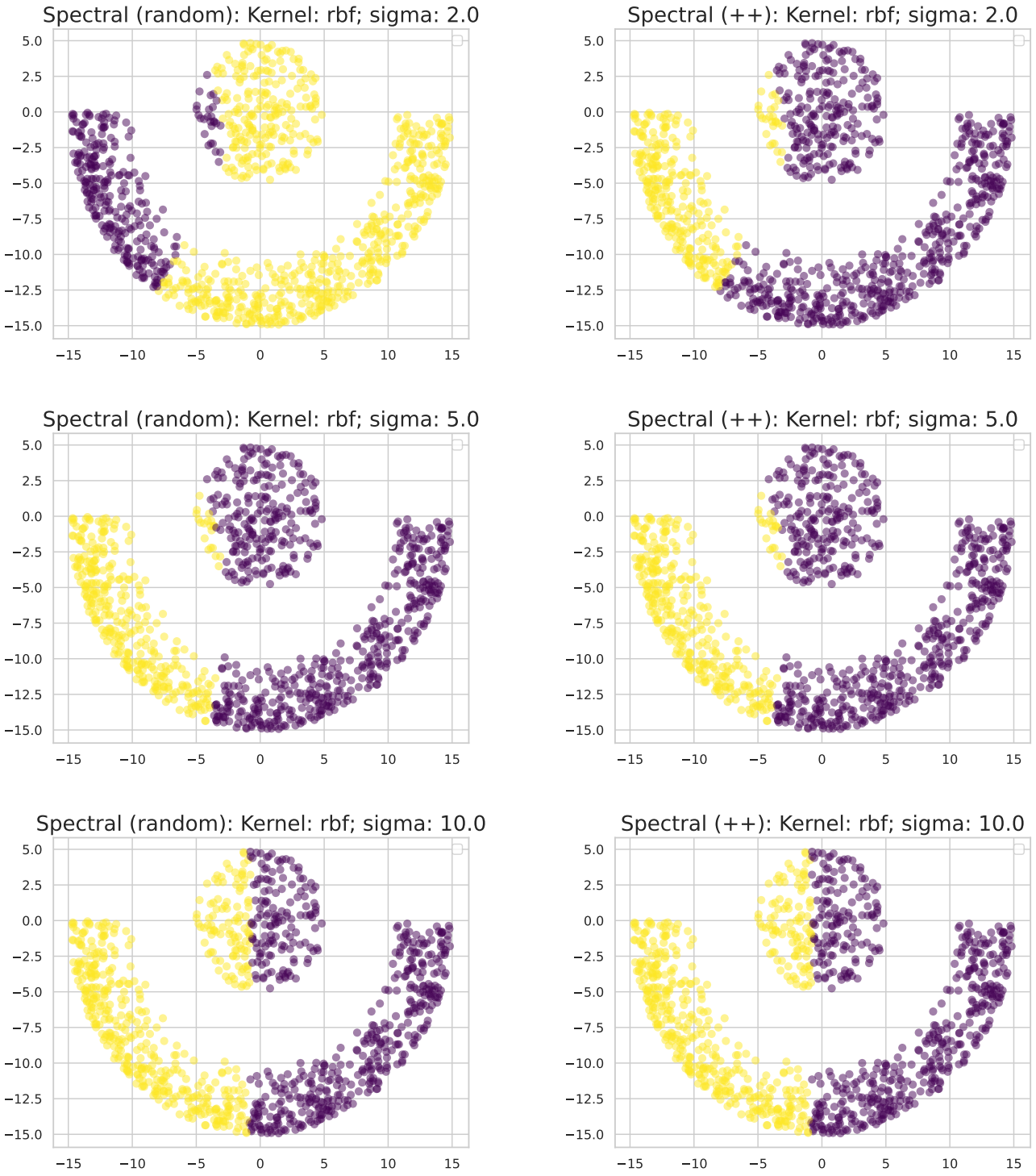


Table 11: K means and K means++ Clustering Using Spectral Clustering for Various values of K using RBF Kernel: Part II