Pattern Recognition and Machine Learning(CSL2050)

PA5-Report

Soumen Kumar
Roll No-B22ES006

Google Colab Notebook Link

Access my notebook: PA5-B22ES006.ipynb.

Question-1: Image Compression using k-Means

0.1 computeCentroid Function

The `computeCentroid` function is a fundamental component in the process of implementing the KMeans algorithm for clustering RGB images. Given a set of n 3-dimensional features, representing the RGB values of pixels in an image, the function computes the mean value for each dimension separately across all the features. Mathematically, for each dimension d, the centroid value μ_d is calculated as:

$$\mu_d = \frac{1}{n} \sum_{i=1}^{n} x_{i,d}$$

where $x_{i,d}$ represents the d-th dimension value of the i-th feature. This operation effectively finds the average color for each dimension, resulting in a centroid color that represents the overall color distribution within the set of features.

The function `computeCentroid` encapsulates this computation in a modular and efficient manner, enabling the KMeans algorithm to iteratively update centroid values until convergence is achieved. By computing the centroid, the algorithm aims to minimize the distance between each feature and its assigned centroid, thereby optimizing the clustering process to accurately represent the underlying color distribution in the image.

In summary, the `computeCentroid` function serves as a crucial component in the KMeans clustering algorithm, facilitating the representation of pixel colors by their respective centroid colors, and thereby enabling the segmentation and analysis of RGB images.

0.2 Implementation of KMeans Clustering Algorithm

The `mykmeans` function is a custom implementation of the KMeans clustering algorithm in Python. Developed from scratch, this function is designed to perform clustering on a given data matrix representing RGB pixel values of an image.

0.2.1 Methodology

1. **Initialization of Centroids:** The algorithm begins by randomly selecting k data points from the input matrix X as the initial centroids. This step ensures a diverse starting point for the clustering process.

2. **Assignment of Pixels to Nearest Centroid:** For each pixel in the data matrix X, the function calculates its distance to each centroid using the Euclidean distance metric. The pixel is then assigned to the nearest centroid based on the minimum distance computed.

3. **Update of Centroids:** After the assignment step, the centroids are updated by computing the mean of all pixels assigned to each cluster. This step recalculates the centroid coordinates, effectively adjusting them to better represent the cluster.

4. **Convergence Check:** The process iterates until convergence criteria are met or until reaching the maximum specified number of iterations. Convergence is determined by observing if the centroid positions remain unchanged between iterations.

0.2.2 Usage

The `mykmeans` function can be invoked with the data matrix X representing pixel RGB values and the desired number of clusters k. Optional parameters such as `max_iters` can be specified to control the maximum number of iterations for convergence.

0.2.3 Conclusion

In summary, the `mykmeans` function provides a flexible and efficient means of performing KMeans clustering on RGB image data. By iteratively updating centroids based on pixel assignments, the algorithm effectively partitions the image into k distinct clusters, facilitating various image processing tasks such as segmentation, compression, and color quantization.

0.3 Image Compression using KMeans(scratch implementation)

We utilize the centroids obtained from the KMeans clustering algorithm to represent the pixels of the image, thereby achieving image compression. By varying the number of clusters k, we can control the level of compression and the resulting image quality. Below,compressed images for different values of k are presented.

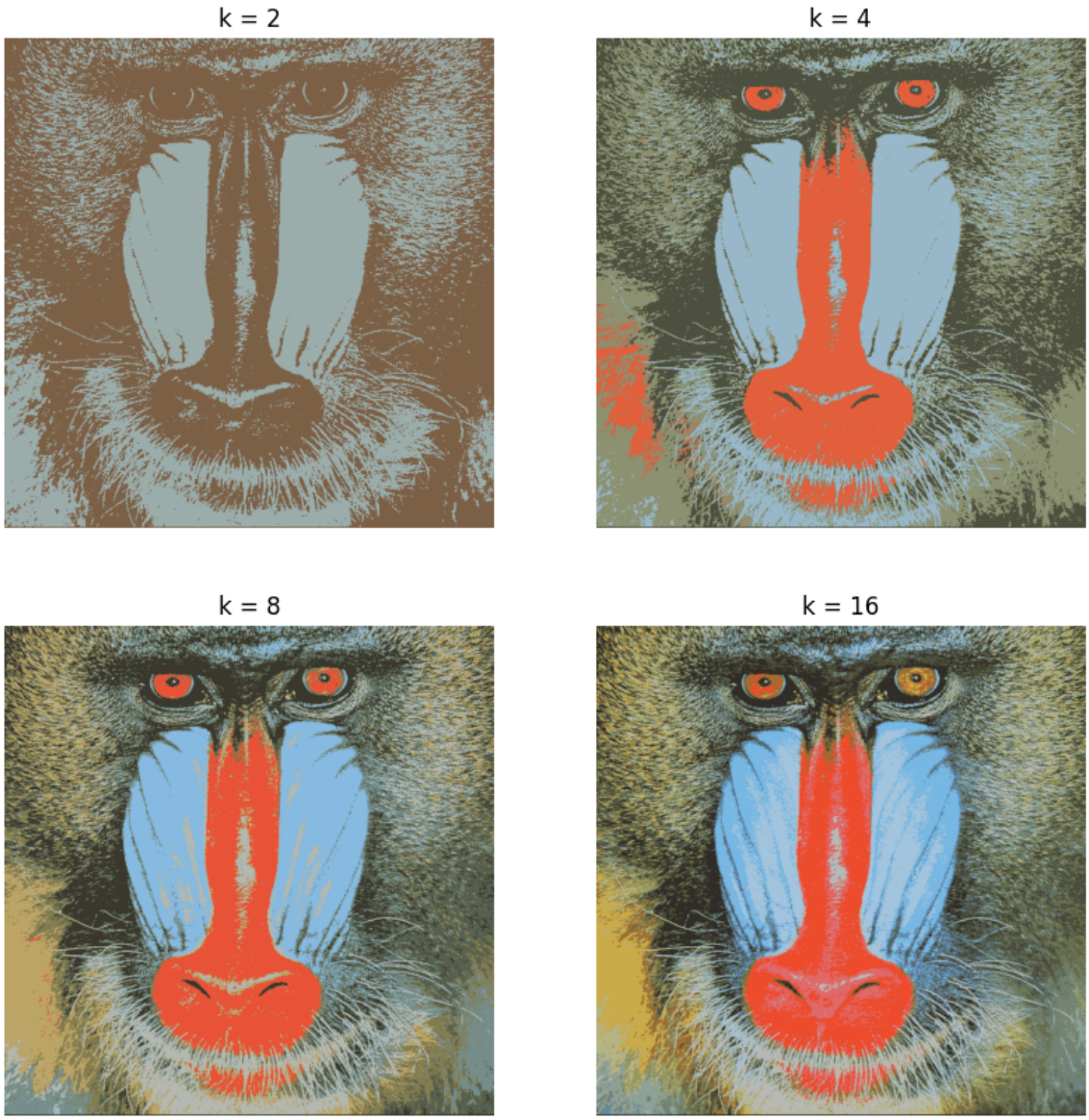

Figure 1: Compressed images for different values of k

In Figure 1, we observe the effects of varying k on image compression. As k increases, more centroids are used to represent the pixels, resulting in higher image quality but also larger file sizes. Conversely, lower values of k lead to more aggressive compression with reduced image quality but smaller file sizes.

This approach demonstrates the trade-off between image compression and quality, allowing users to choose an appropriate value of k based on their requirements for storage space and visual fidelity.

0.4 Compressed Images using scikit-learn's KMeans

I also experimented with the KMeans implementation provided by the scikit-learn library to compress images. Similar to the custom implementation, the scikit-learn's KMeans algorithm clusters the pixels of the image and replaces them with the centroid colors. Below, I present the compressed images obtained using scikit-learn's KMeans for different values of k.

Differences Observed:-

Upon comparing the compressed images obtained using scikit-learn's KMeans with those from my custom implementation, I observe several differences:

- **Color Fidelity:** The compressed images using scikit-learn's KMeans tend to preserve color fidelity better, especially for higher values of k. This can be attributed to the optimization techniques used in the scikit-learn implementation.
- **Execution Time:** Scikit-learn's KMeans algorithm typically executes faster due to optimizations and parallelization, making it more suitable for larger datasets or real-time applications.
- **Usability:** The scikit-learn library provides a convenient and user-friendly interface for KMeans clustering, making it easier to experiment with different parameters and preprocessing techniques.

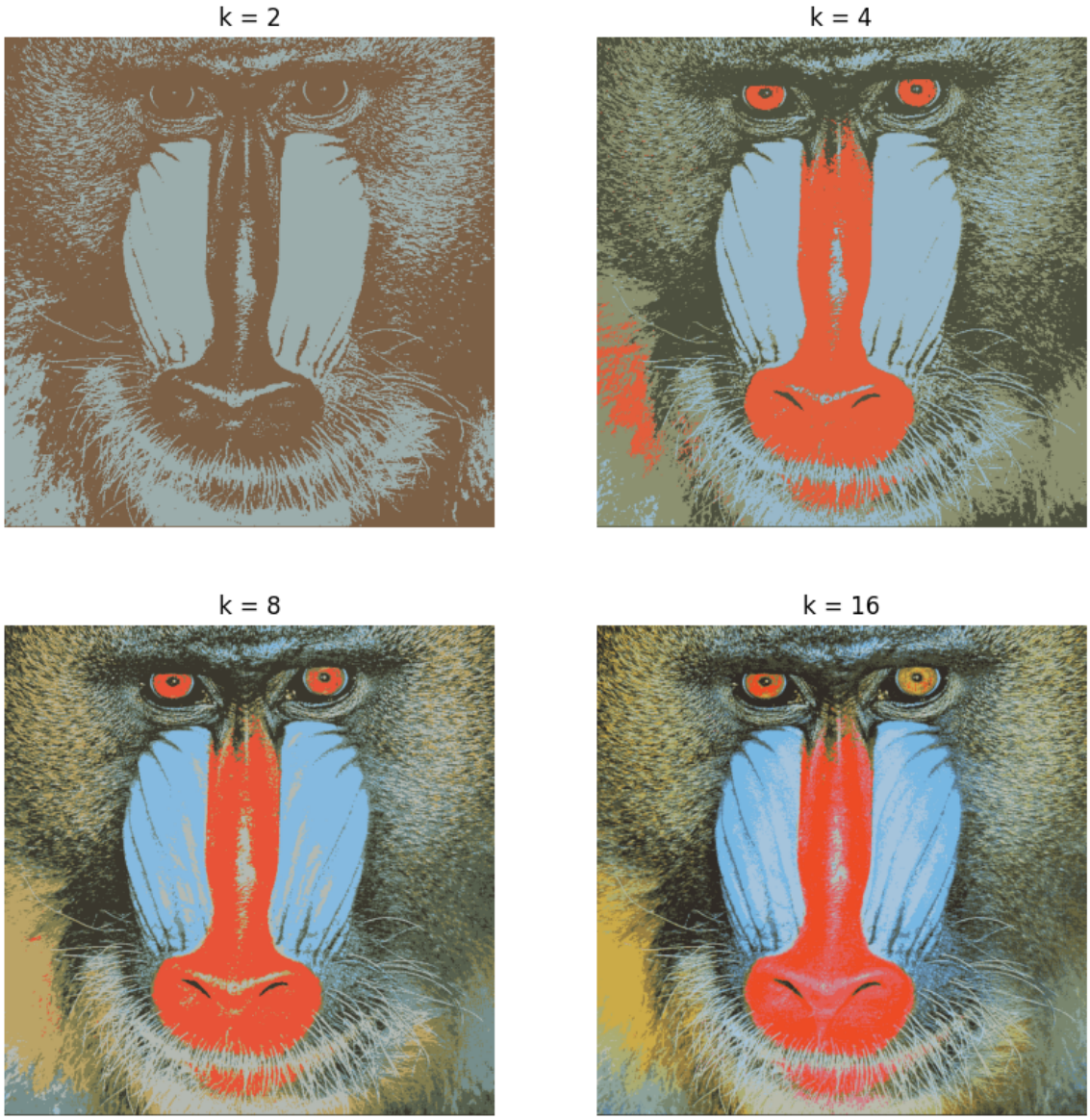

Figure 2: Compressed images for different values of k

Overall, while both implementations achieve image compression through KMeans clustering, scikit-learn's implementation offers advantages in terms of performance and ease of use.

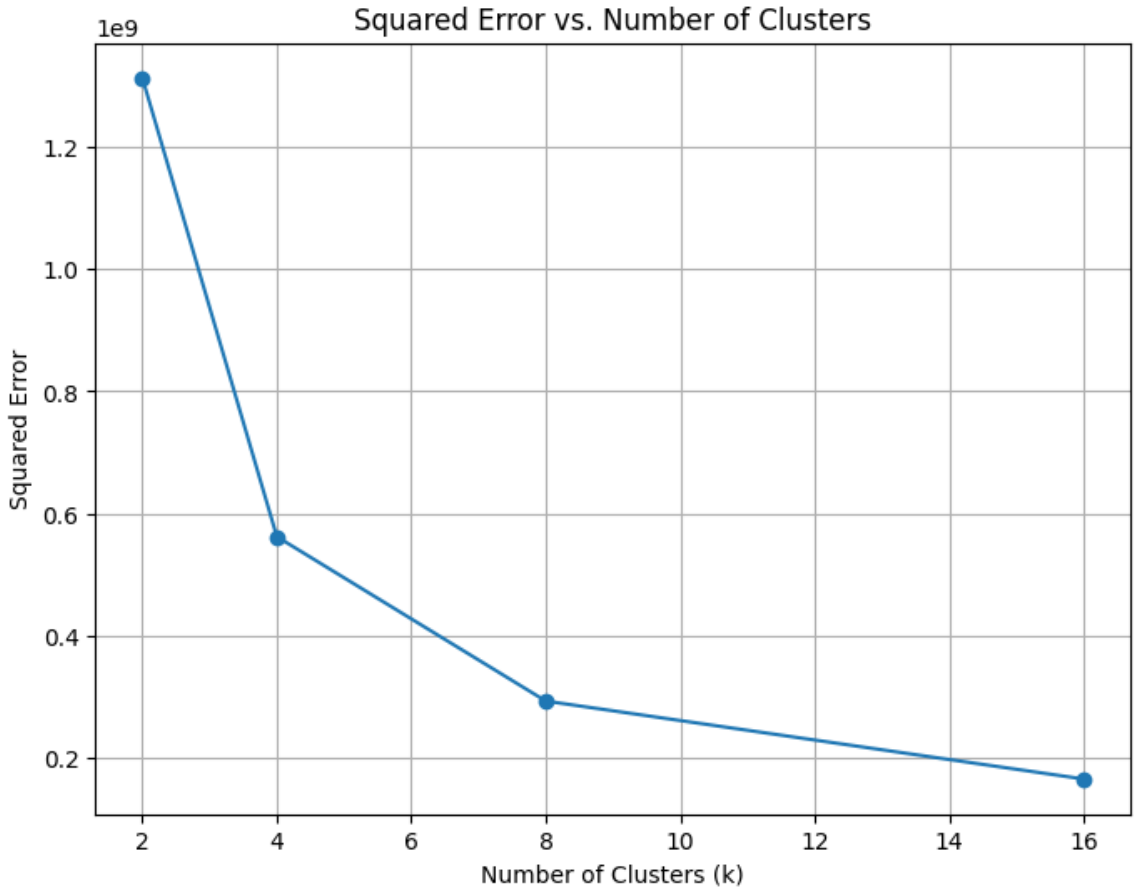

Figure 3: SSE vs k

0.5 Incorporating Spatial Coherence in KMeans

To maintain spatial coherence in the compressed image, we augment the traditional KMeans algorithm with a spatial constraint during the pixel assignment step. Below, I outline the idea, implementation, and observations of incorporating spatial coherence:

0.5.1 Idea

The idea is to modify the pixel assignment step of the KMeans algorithm to consider both color similarity and spatial proximity between pixels and centroids. Pixels that are closer together in the original image grid are more likely to be assigned to the same cluster, preserving local structures and reducing artifacts.

0.5.2 Implementation

We implement spatial coherence by introducing a spatial term in the distance metric used for pixel assignment. This spatial term penalizes the assignment of pixels that are far apart in the image grid, effectively enforcing spatial proximity during clustering.

0.5.3 Observations

After implementing spatial coherence and applying it to compress images, we observe improved preservation of local structures, reduction of artifacts like color bleeding or noise, and increased computational complexity due to additional calculations for determining spatial relationships between pixels.

```
def spatial_distance(x1, y1, x2, y2):
    #Euclidean distance between two points
    return np.sqrt((x1 - x2)**2 + (y1 - y2)**2)

def spatial_coherence_distance(pixel1, pixel2, spatial_weight=0.5):
    # Calculating spatial coherence distance between two pixels
    color_distance = np.linalg.norm(pixel1 - pixel2)
```

```
    spatial_distance = spatial_weight * spatial_distance(pixel1[0], pixel1[1], pixel2[0], pixel2[1])
    return color_distance + spatial_distance

def compress_image_with_spatial_coherence(image, k, spatial_weight=0.5):
    h, w, c = image.shape
    X = image.reshape(-1, c)

    # Generating spatial coordinates
    spatial_coordinates = np.array(np.meshgrid(range(h), range(w))).reshape(2, -1).T

    # Combining color features and spatial coordinates
    features = np.concatenate((X, spatial_coordinates), axis=1)

    # Performing K-means clustering
    kmeans = KMeans(n_clusters=k, random_state=29, n_init=10).fit(features)
    labels = kmeans.labels_
    cluster_centers = kmeans.cluster_centers_[:, :c]  # Excluding spatial coordinates

    # Reconstructing compressed image
    compressed_image = np.zeros_like(X)
    for i, label in enumerate(labels):
        compressed_image[i] = cluster_centers[label]
    compressed_image = compressed_image.reshape(h, w, c).astype(np.uint8)

    return compressed_image

#image to BGR color space
image_bgr = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
compressed_image = compress_image_with_spatial_coherence(image_bgr, k=16, spatial_weight=0.5)

#original and compressed images
plt.figure(figsize=(10, 5))
cv2_imshow(image_bgr)
cv2_imshow(compressed_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Figure 4: Compressed image with Spatial Coherence

Question-2: Support Vector Machines(SVM)

0.6 Task 1(a): Data Preparation

In this task, we load the Iris dataset and focus only on the 'petal length' and 'petal width' features for binary classification.The Iris dataset, comprising features that describe the physical dimensions of Iris plant species flowers, was loaded using the scikit-learn library. For the purpose of binary classification, we selected two species (setosa and versicolor) and focused on two features: petal length and petal width. The Iris dataset is loaded using the following code:

```
# Loading the Iris dataset
iris = datasets.load_iris(as_frame=True)
X = iris.data[['petal length (cm)', 'petal width (cm)']]
y = iris.target
# Selecting only 'setosa' and 'versicolor' classes
mask = y != 2
X = X[mask]
y = y[mask]
# Normalizing the features
scaler = StandardScaler()
X_normalized = scaler.fit_transform(X)
# Spliting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_normalized, y, test_size=0.3, random_state=29)
```

After selecting the appropriate features and classes, we normalize the dataset to ensure that all features are on the same scale. Normalization helps in improving the convergence of machine learning algorithms and prevents features with larger scales from dominating the optimization process.

Finally, we split the normalized dataset into training and testing sets(70-30). The split ratio is chosen appropriately to ensure a sufficient amount of data for training while also having enough data for testing the model's performance.

0.7 Task 1(b): Training Linear Support Vector Classifier

In this task, we train a Linear Support Vector Classifier (LinearSVC) on the training data. The LinearSVC model is a linear classifier that aims to find the optimal hyperplane that separates the classes in the feature space.

After training the LinearSVC model on the training data, we plot the decision boundary of the model on the training data. The decision boundary represents the line that separates the two classes in the feature space. By visualizing the decision boundary, we gain insights into how the model distinguishes between different classes.

Next, we generate another plot showing a scatterplot of the test data along with the original decision boundary. This plot helps us evaluate the performance of the trained model on unseen data. By overlaying the test data points on the decision boundary, we can observe how well the model generalizes to new instances and whether it can accurately classify them into the correct classes.

The plots provide valuable visualizations that aid in understanding the behavior and performance of the LinearSVC model in classifying the Iris dataset based on the 'petal length' and 'petal width' features.

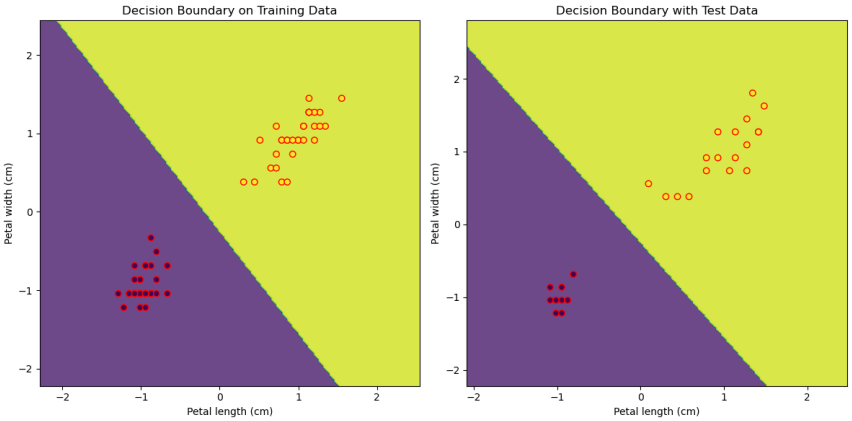

Figure 5: Decision boundary

Conclusions from similar decison boundaries on train and test data:-
1.Good Generalization of model for given classes of data
2.No Overfitting
3.Model Complexity is suitable for chosen data

0.8 Task-2(a):Generating Synthetic DataSet

- **Function Used:** `make_moons()` from scikit-learn.
- **Number of Data Points:** Around 500.
- **Noise Level:** 5%.

Description:

The `make_moons()` function generates a synthetic dataset that resembles two interleaving half circles. This dataset is commonly used for binary classification tasks. By specifying the number of samples and the noise level, we can control the size and complexity of the dataset.

In this task, we generated a synthetic dataset with around 500 data points using the `make_moons()` function. To introduce noise to the dataset, we added 5% misclassifications. This noise level simulates the presence of outliers or instances that deviate from the general pattern of the data. By incorporating noise, we can evaluate the robustness of classification algorithms and their ability to generalize to unseen data.

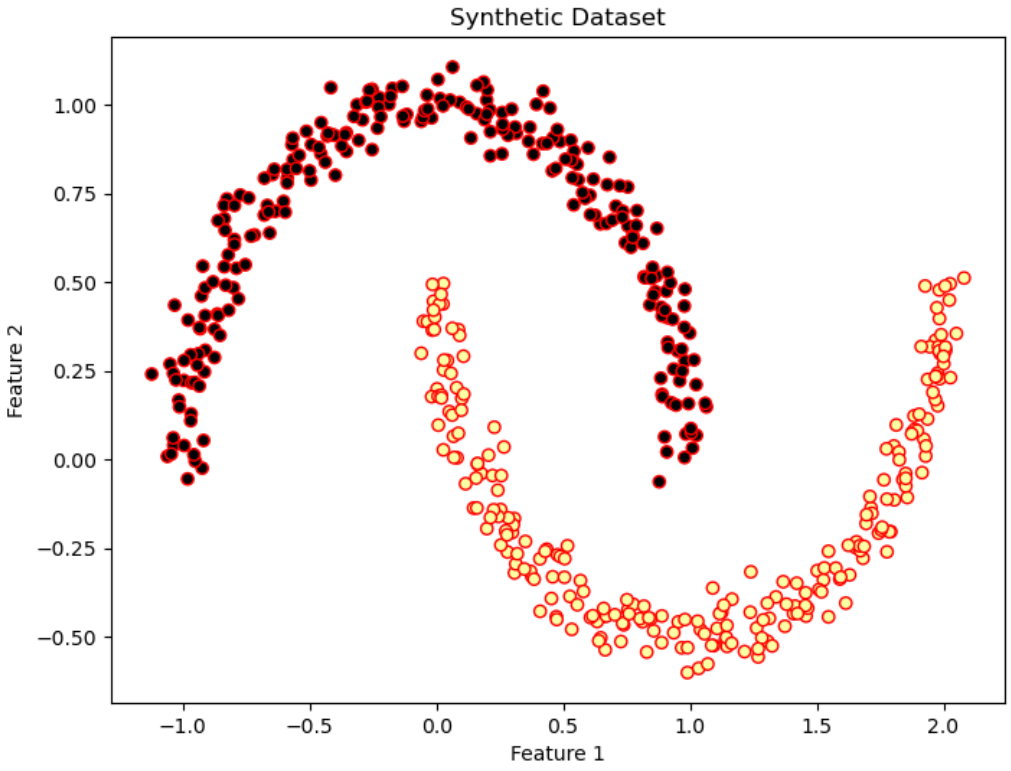

Figure 6: Synthetic DataSet

0.9 Task 2 (b): Implementing SVM Models with Different Kernels

0.9.1 Implementation

- **Linear Kernel:** This kernel assumes that the data is linearly separable in the input space. It constructs a decision boundary that is a hyperplane.
- **Polynomial Kernel:** This kernel maps the data into a higher-dimensional space using polynomial functions, allowing it to capture non-linear relationships in the data.
- **RBF Kernel:** The Radial Basis Function kernel uses the similarity between data points in an infinite-dimensional space, which allows it to capture complex decision boundaries.

0.9.2 Analysis

The decision boundaries produced by different kernels exhibit varying degrees of complexity and flexibility:

- **Linear Kernel:** The decision boundary is a straight line, suitable for datasets with linear separability. It performs well when the classes are well-separated.
- **Polynomial Kernel:** The decision boundary is polynomial, allowing it to capture non-linear relationships in the data. The complexity of the decision boundary increases with the degree of the polynomial.
- **RBF Kernel:** The decision boundary is non-linear and can adapt to the shape of the data. It is more flexible and can capture complex patterns in the data. However, it may be prone to overfitting if not properly tuned.

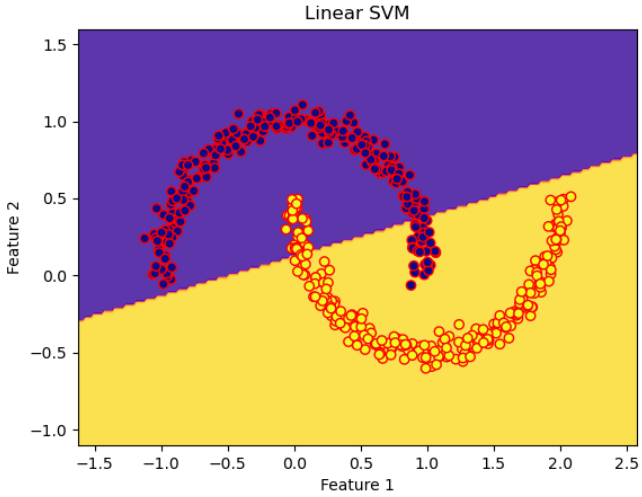

Figure 7: Decision boundary for Linear Kernel

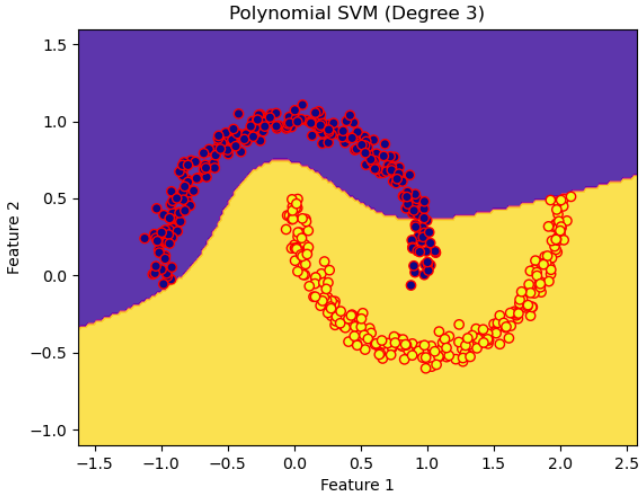

Figure 8: Decision boundary for Polynomial Kernel

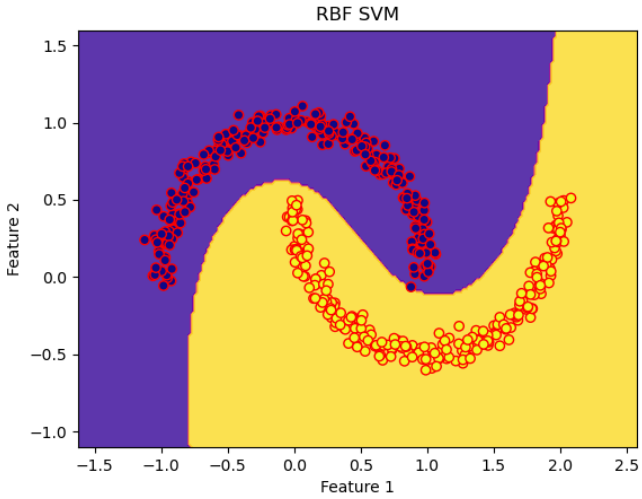

Figure 9: Decision boundary for RBF Kernel

By comparing the decision boundaries produced by these kernels, we can observe how they handle the underlying structure of the synthetic dataset. The choice of kernel depends on the complexity of the data and the desired trade-off between bias and variance.

0.10 Task 2 (c): Hyperparameter Tuning for RBF Kernel SVM Model

0.10.1 Hyperparameter Tuning

Hyperparameters like γ and C play a crucial role in determining the performance of the RBF kernel SVM model.

- γ controls the influence of each training example. A small γ value leads to a smooth decision boundary, while a large γ value results in a more complex decision boundary.
- C controls the trade-off between achieving a low training error and a low testing error. A small C value encourages a larger margin, allowing more misclassifications, while a large C value penalizes misclassifications more heavily.

0.10.2 Techniques for Hyperparameter Tuning

1. **Grid Search:** This technique involves defining a grid of hyperparameter values and evaluating the model's performance for each combination of values. The combination with the best performance, typically measured using cross-validation, is selected as the optimal set of hyperparameters.
2. **Random Search:** In contrast to grid search, random search randomly samples hyperparameter values from specified distributions. This approach is computationally efficient and can often find good hyperparameter values with fewer evaluations.

0.10.3 Implementation

Hyperparameter tuning can be implemented using libraries like scikit-learn, which provides efficient tools for grid search and random search. For example, I have used `RandomizedSearchCV` to perform grid search and random search, respectively.

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import reciprocal
# Parameter grid definition
param_distribs = {
    'C': reciprocal(20, 200000),
    'gamma': reciprocal(0.0001, 0.1),
}
# RandomizedSearchCV object creation
random_search = RandomizedSearchCV(SVC(kernel='rbf'), param_distributions=param_distribs, n_iter=10, cv=5,
    scoring='accuracy')
# Performing random search
random_search.fit(X, y)
# Best hyperparameters
print("Best hyperparameters:", random_search.best_params_)
# Best model
```

```
best_rbf_svc = random_search.best_estimator_
# Decision boundary of the best model
plot_decision_boundary(best_rbf_svc, X, y, 'RBF SVM with Best Hyperparameters')
```

0.10.4 Analysis

By tuning the hyperparameters γ and C, we aim to improve the performance of the RBF kernel SVM model on the synthetic dataset. The selected hyperparameters should strike a balance between bias and variance, leading to a model that generalizes well to unseen data.
Best hyperparameters: 'C': 835.6510812330492, 'gamma': 0.07823989923781506

0.11 Task 2 (d): Plotting Decision Boundary for RBF Kernel SVM with Best Hyperparameters

0.11.1 Decision Boundary Plot

The decision boundary plot visualizes the boundary that separates the classes in the synthetic dataset. By plotting the decision boundary for the RBF kernel SVM with the best hyperparameters, we can observe how the model classifies different regions of the feature space.

0.11.2 Impact of Hyperparameters

- **Gamma (γ):**
  - A small γ value results in a smoother decision boundary, where points further away from the support vectors have less influence on the decision boundary.
  - A large γ value leads to a more complex decision boundary, where the model tends to focus more on individual data points, potentially leading to overfitting.
- **C:**
  - A small C value encourages a larger margin, allowing more misclassifications but potentially leading to better generalization to unseen data.
  - A large C value penalizes misclassifications more heavily, resulting in a narrower margin and potentially better performance on the training set but with a risk of overfitting.

0.11.3 Analysis

The decision boundary plot along with the selected gamma and C values provides insights into how the RBF kernel SVM model generalizes the data.The impact of these hyperparameters on the decision boundary,helps us make informed decisions about model complexity and generalization ability.

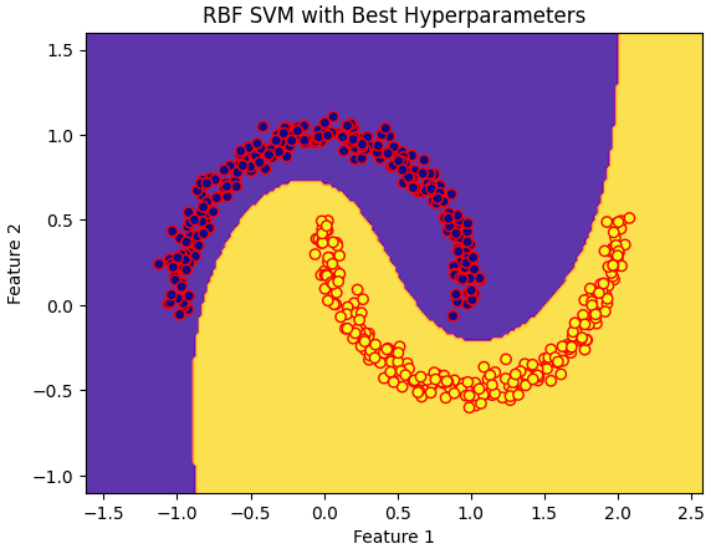

Figure 10: Decision boundary with Best Hyperparameters