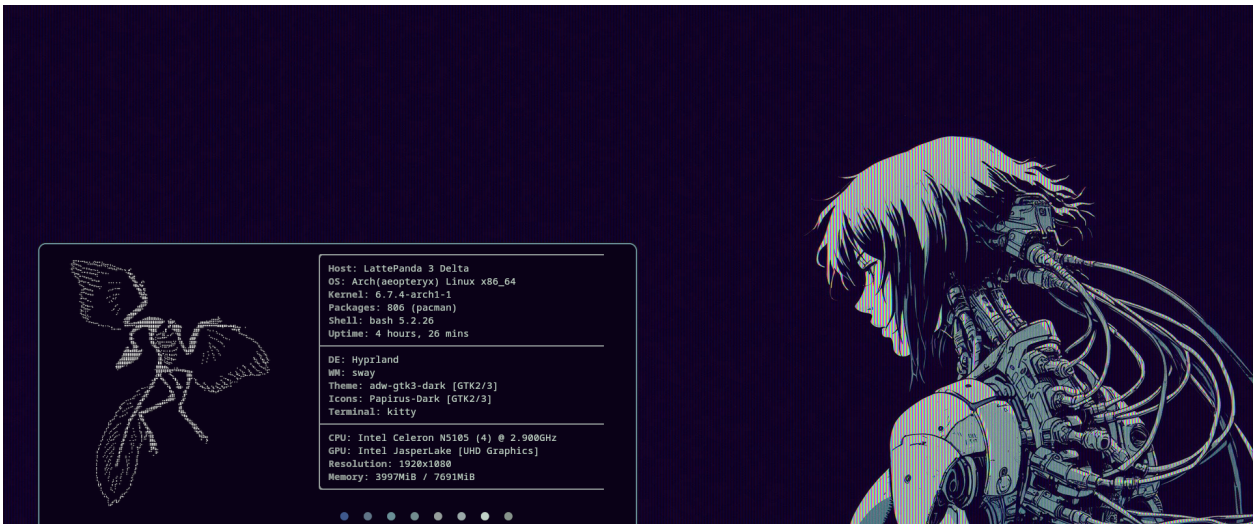


B22EE036_PRML_PA5_REPORT

AUTHOR : K. K. N. SHYAM SATHVIK

ROLL.NO : B22EE036

LINK TO THE COLAB FILE : (The tasks were performed locally and later uploaded to colab, the notebook is also attached with the final submission.)

Google Colaboratory

https://colab.research.google.com/drive/1RKhzZo_I_GjTeDJ-Q5nWxRQTGovB7FkI?usp=sharing

Task 1: Image Compression using K-Means Clustering

K-Means Clustering:

The K-Means algorithm is a unsupervised clustering method used to partition a set of points into K distinct clusters, minimizing the variance within each cluster.

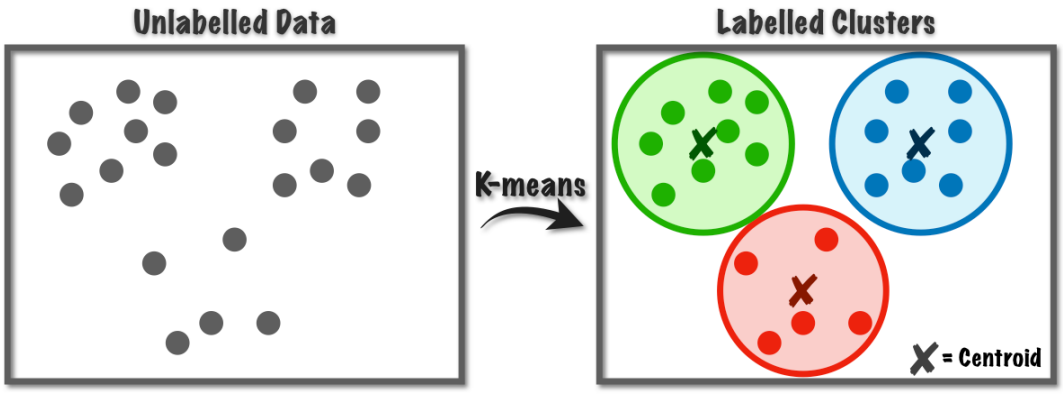
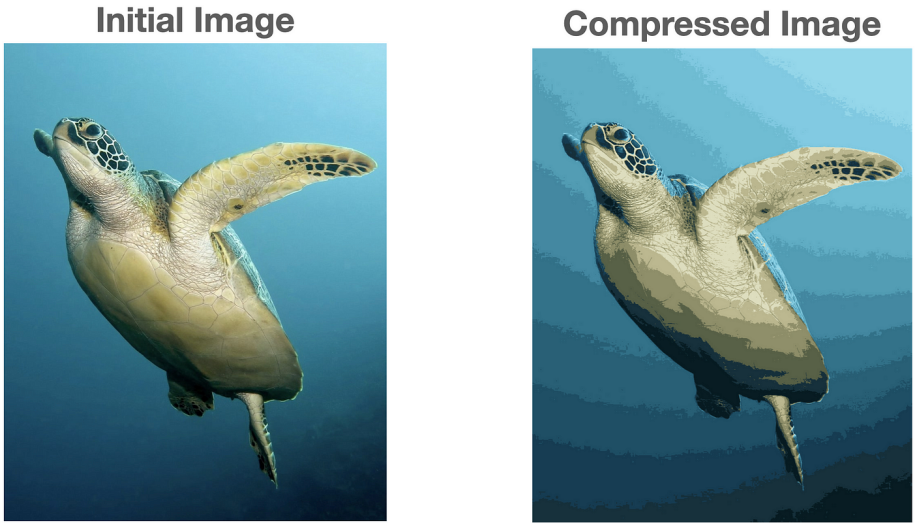


Image compression using K-means clustering is a technique that can be used to reduce the size of an image file while maintaining its visual quality. This technique involves clustering the pixels in an image into a smaller number of groups and then representing each group by its mean color. The resulting image will have fewer colors, which reduces the file size, but the overall appearance of the image is still preserved.



K-Means Clustering Algorithm:

- The core of the K-Means algorithm involves minimizing the within-cluster sum of squares, which is the sum of squared distances between each point and its corresponding centroid. Mathematically, this is expressed as:

$$J = \sum_{k=1}^{K} \sum_{x \in C_k} \|x - \mu_k\|^2$$

- **The objective is to minimize J through iterative refinement of clusters and centroids.**

Algorithm 1 K-means clustering algorithm

Require: K, number of clusters; D, a data set of N points
Ensure: A set of K clusters

1. Initialization.
2. **repeat**
3. **for** each point p in D **do**
4. find the nearest center and assign p to the corresponding cluster.
5. **end for**
6. update clusters by calculating new centers using mean of the members.
7. **until** stop-iteration criteria satisfied
8. **return** clustering result.

Algorithm for K-Means Clustering

Step 1: Initialize Cluster Centers

- Select K points as the initial centroids, usually done randomly from the dataset.

$$\text{Centroids} = \{\mu_1, \mu_2, ..., \mu_K\}$$

where each μ_k represents the center of cluster k.

Step 2: Assign Points to the Nearest Centroid

- Each point in the dataset is assigned to the nearest centroid based on the Euclidean distance. This step forms the clusters.

$$C_k = \{x \mid d(x, \mu_k) \leq d(x, \mu_j) \forall j \neq k\}$$

where $d(x, y)$ denotes the Euclidean distance between points x and y, and C_k is the set of points assigned to cluster k.

Step 3: Update Centroids

- After all points are assigned, update the centroids of each cluster to be the mean of the points in the cluster.

$$\mu_k = \frac{1}{|C_k|} \sum_{x \in C_k} x$$

where $|C_k|$ is the number of points in cluster k.

Step 4: Repeat Until Convergence

- Repeat Steps 2 and 3 until the centroids no longer change significantly, indicating convergence. This is typically determined by a small threshold ϵ for the centroids' movement or a maximum number of iterations.

$$\min_{w,b} \frac{1}{2}\|w\|^2 \quad \text{subject to} \quad y_i(w \cdot x_i + b) \geq 1 \quad \forall i$$

▼ Task-(a):

- Implement a function – computeCentroid, that takes n 3-dimensional features and returns their mean. (2 pts)

```
def compute_centroid(features):
    # generalised for n-dimensional data
    return np.mean(features, axis=0)
```

- Implemented using `np.mean`

▼ Task-(b):

- Implement a function – mykmeans from scratch that takes data matrix X of size m×3 where m is the number of pixels in the image and the number of clusters k. It returns the cluster centers using the k-means algorithm. (3 pts)

- K-Means is implemented from scratch using the above described algorithm. The code shown abstracts away the internal implementation details, see the python notebook (`.ipynb`) for the implementation of the helper funcitons.

```
def mykmeans(X, k, max_iters=100):
    """K-Means Clustering Algorithm"""
    centroids = initialize_centroids(X, k)

    for _ in range(max_iters):
        closest_centroids = assign_clusters(X, centroids)
        new_centroids = update_centroids(X, closest_centroids, k)
        if check_convergence(centroids, new_centroids):
            break
        centroids = new_centroids

    return centroids
```

▼ Task-(c):

- Use the centroids of k-means to represent the pixels of the image.
- Now, show compressed images for different values of k. (1 pts)

```
def compress_image(image_np, centroids, labels):
    # replace each pixel value with its centroid value
    compressed_image = centroids[labels]
    compressed_image = compressed_image.reshape(image_np.shape)
    return compressed_image.astype(np.uint8)
```

- **Compressed Images for different values of K:**

Compressed Image with 2 Colors

Compressed Image with 4 Colors

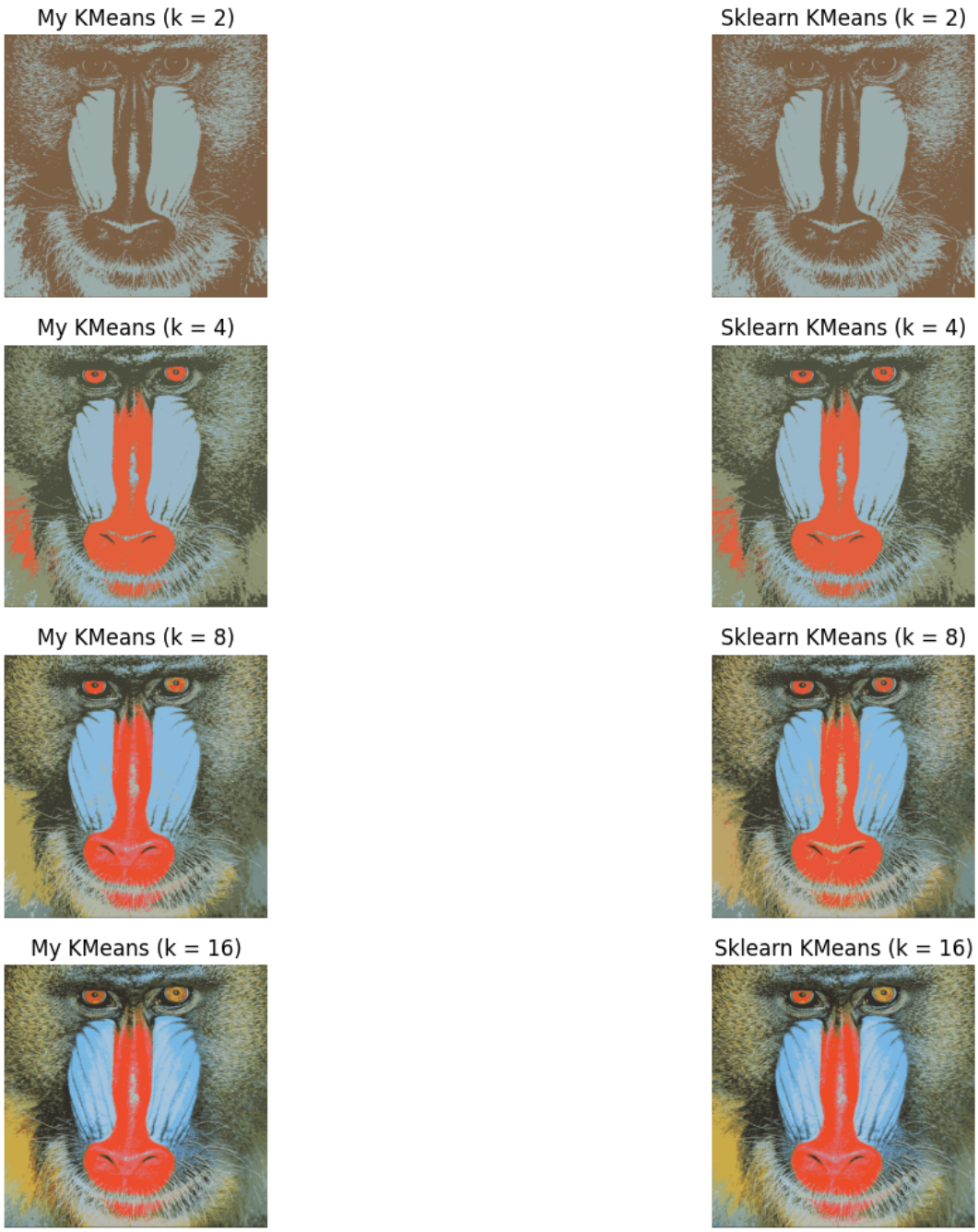
Compressed Image with 8 Colors

Compressed Image with 16 Colors

- We can see the loss of definition in the image as the value of the centroid colors (colors picked to represent a cluster) decreases.

Task-(d):

Show the results of compressed images using the k-means implementation of the sklearn library. What differences do you observe? (2 pts)



Side by Side comparision of my k-means implementation and k-means from sklearn

- While the side by side comparision mostly seems to reveal that both the K-Means offer the same kind of compression. Looking closely, we find that for the K-Means with (k=8) there is a visible difference in the **white patch** on the monkey's nose.
    - The sklearn's K-means compressed image seems to have a higher spread of the white color, whereas in **my kmeans compressed image** the white patch/color appears to have a lower spread or variance, and is limited to a lower area. **This shows the difference in results that arise from the random initialization of the starting centroids.**
    - **It can also be observed that different areas in the images seem to have different levels of details in the compressed images, with none of them being superior to the other. This shows that the differences mostly arise due to the random initialization and given that the model's have close enough initial centroids, the compressed versions will be close enough.**

Task (e) :

- Spatial coherence: Incorporating spatial information helps maintain spatial coherence in the compressed image.
- Pixels that are nearby in the original image are more likely to be assigned to the same cluster, preserving local structures and reducing artifacts like color bleeding or noise.
- How do you implement spatial coherence? Write the idea, implement it, and write down your observation. (2 pts))

My KMeans with Spacial Coherence (k = 2)

My KMeans without Spacial Coherence (k = 2)

My KMeans with Spacial Coherence (k = 4)

My KMeans without Spacial Coherence (k = 4)

My KMeans with Spacial Coherence (k = 8)

My KMeans without Spacial Coherence (k = 8)

My KMeans with Spacial Coherence (k = 16)

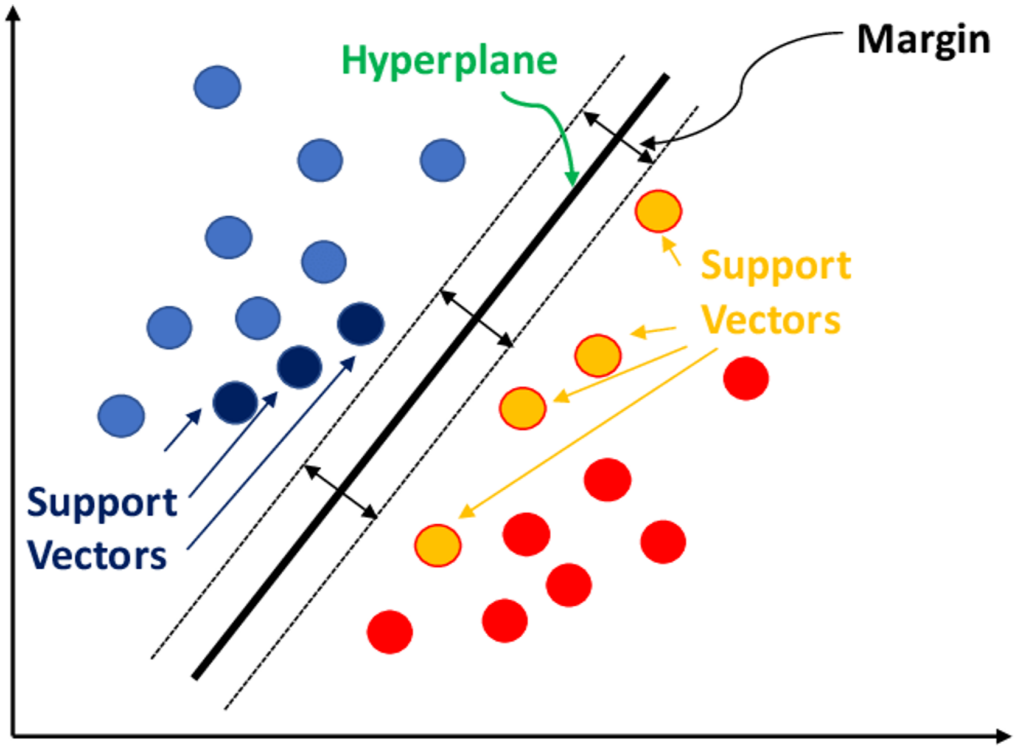
My KMeans without Spacial Coherence (k = 16)

- To incorporate spacial coherence in the image, so as to preserve the local structures and reducing artifacts like color bleeding or noise. The idea I've come up with is to **add two seperate features to the input image given to the K-Means in the form of `X` and `Y` which are the spacial coordinates of the given pixel found from applying a meshgrid on the image.** The input data then has additional information in the form of coordinates which effectively push the k-means into grouping closer pixels (colors or RGB values) together in the same clusters.

- This is the function implemented for transforming the input data into ones with spacial coherence, it takes in the input in the form of the original RGB Image `(, 3) array` and gives the output as a `(, 5) array` with the additional columns being `X and Y`.

```
def add_spatial_features(image_np, scale=0.1):
    m, n, _ = image_np.shape
    X, Y = np.meshgrid(range(n), range(m))
    features = np.c_[image_np.reshape(-1, 3), scale * X.flatten(), scale *
    return features
```

TASK 2: Support Vector Machine



Support Vector Machines (SVM) are a set of supervised learning methods used for classification, regression, and outliers detection. The SVM model represents the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible.

The basic idea of an SVM is to find a hyperplane that best separates the classes in feature space. If the training data is linearly separable, we can select two parallel hyperplanes that separate the two classes of data, so that the distance between them is as large as possible.

The region bounded by these two hyperplanes is called the "margin", and the linear SVM consists of finding the hyperplane that maximizes this margin. The training examples that are closest to the hyperplane, which influence the position and orientation of the hyperplane, are termed support vectors.

The mathematical model for a linear SVM is as follows:

$$f(x) = \text{sgn}(w \cdot x + b)$$

Here:

- `f(x)` represents the output of the SVM, given input vector `x`.
- `w` is the weight vector.
- `b` is the bias term.
- $w \cdot x$ denotes the dot product of `w` and `x`.
- sgn is the sign function, which returns 1 if the argument is positive and -1 otherwise.

Some Important Terms:

Parameter C:

- In the context of SVMs, as we see below, C is a regularization parameter that controls the trade-off between achieving a low error on the training data and minimizing the norm of the weights. It determines the penalty for misclassifying each example in the training data.
- **A small C makes the decision surface smooth, while a large C aims at classifying all training examples correctly by giving the model freedom to select more samples as support vectors.**

Hard Margin SVM

- For the linearly separable case (hard margin), we want to maximize the margin between the two classes, subject to the constraint that all examples are correctly classified. This can be formulated as:

$$\min_{w,b} \frac{1}{2}\|w\|^2 \quad \text{subject to} \quad y_i(w \cdot x_i + b) \geq 1 \quad \forall i$$

Here, y_i are the labels associated with each data point x_i, which are either 1 or -1.

Soft Margin SVM

A soft margin SVM allows some misclassifications to allow for the case when the data is not linearly separable, which applies to most real life data since they have some noise in them. The optimization problem in this case incorporates slack variables
ξ_i to handle data points that are on the wrong side of the margin

$$\min_{w,b,\xi} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\xi_i \quad \text{subject to} \quad y_i(w \cdot x_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0 \quad \forall i$$

Here, C is a penalty parameter that controls the trade-off between increasing the size of the margin and ensuring that the x_i lie on the correct side of the margin.

Dual Formulation

The dual formulation of the SVM optimization problem is derived by applying the method of Lagrange multipliers and involves solving:

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j (x_i \cdot x_j) \quad \text{subject to} \quad \sum_{i=1}^{n} y_i \alpha_i = 0 \quad \text{and} \quad 0 \leq \alpha_i \leq C \quad \forall i$$

In this formulation, α_i are the Lagrange multipliers, and this problem is subject to certain constraints.

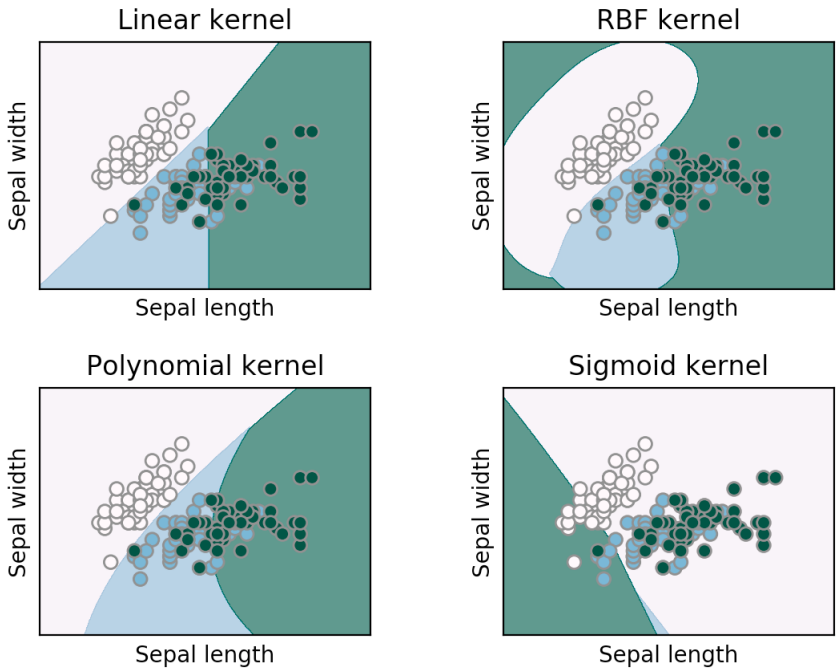
Kernel Trick

- The kernel trick is a technique that allows Support Vector Machines (SVMs) to operate in a high-dimensional, implicit feature space without ever computing the coordinates of the data in that space. Instead, it computes the inner products between the images of all pairs of data in the feature space. This approach lets SVMs efficiently handle cases where the boundary between classes is not linear.

- **Mathematical Representation:**
  - Given a mapping $\phi : X \to F$, which maps input data into a feature space, the kernel function K is defined as:

$$K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$$

- **Types of Kernel Functions:**



- **Linear Kernel**:
  - The simplest kernel, equivalent to no transformation.
  - Defined as: $K(x_i, x_j) = x_i \cdot x_j$

- **Polynomial Kernel**:
  - Creates a non-linear decision boundary in the original space by considering polynomial combinations of the original features.
  - Defined as: $K(x_i, x_j) = (\gamma x_i \cdot x_j + r)^d$
  - Parameters: γ (scale), r (offset), d (degree of the polynomial)

- **Radial Basis Function (RBF) Kernel** (or Gaussian kernel):
  - Maps samples into a higher-dimensional space using a Gaussian distribution centered on the original data points.
  - Defined as: $K(x_i, x_j) = e^{-\gamma \|x_i - x_j\|^2}$
  - Parameter γ controls the width of the Gaussian.

With the above context, The mathematical model for an SVM with a kernel function K is given as follows:

$$f(x) = \text{sgn}\left(\sum_{i=1}^{n} \alpha_i y_i K(x_i, x) + b\right)$$

Here:

- x represents an input feature vector for which you are predicting the class.
- x_i are the support vectors, which are the training examples that lie closest to the decision boundary.
- y_i are the labels of these support vectors.
- α_i are the Lagrange multipliers obtained from solving the SVM's dual problem.
- b is the bias term.
- $K(x_i, x)$ is the kernel function evaluating the inner product between the support vector x_i and the input vector x.

> 💡 The solution for the Dual Problem ensures a unique and optimal solution for the forumulated SVM problem.

Task-1(a):

- Load the Iris dataset using the following code:

```
from sklearn import datasets
```

```
iris = datasets.load_iris(as_frame=True)
```

- For this problem, focus only on the petal length and petal width features. Create a new dataset by selecting only two classes, ‘setosa’ and ‘versicolor’ for binary classification. Normalize the dataset, and split it into train and test by choosing an appropriate split ratio. (2 pts)

Data Normalization and Train-Test Split:

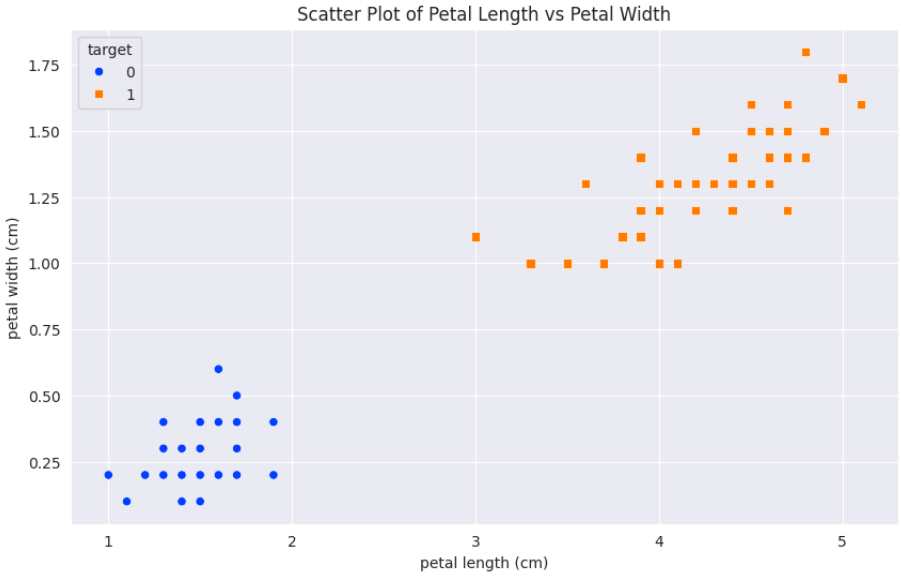
- The class `virginica` was dropped from the dataframe and the data used henceforth in the assignment only contains the classes `setosa` and `versicolor`

- The data was then standardized with the `StandardScaler()` from Sklearn, and was split into
  - `train` : **70%** and `test` : **30%**

  using sklearn’s `train_test_split`

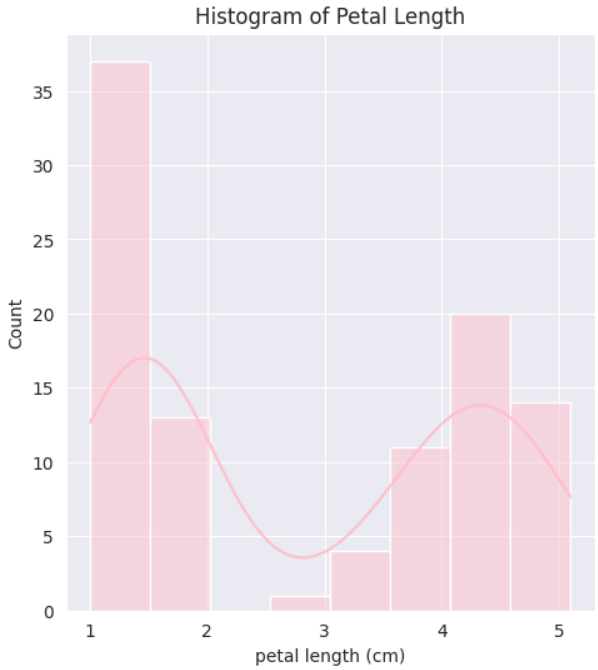
Data Visualization:

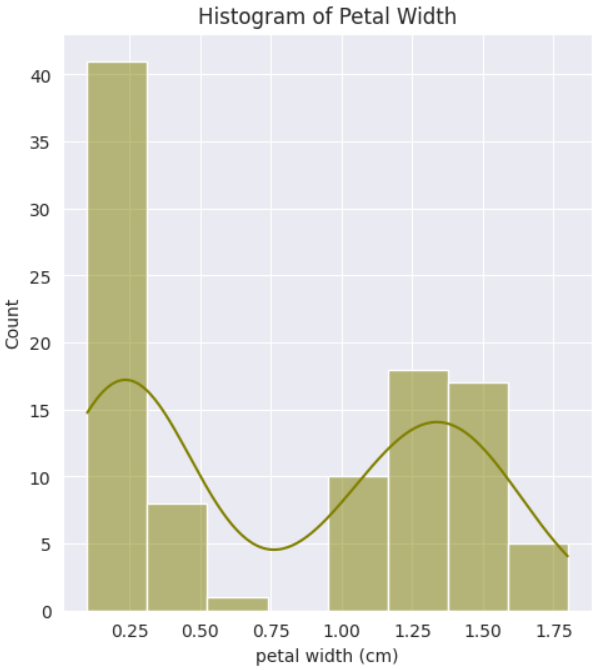
1. Scatter plot of petal length vs petal width



- Petal length and petal width are positively correlated; as one increases, the other also tends to increase.
- Different species cluster in distinct regions, indicating that petal measurements can help differentiate between the species.

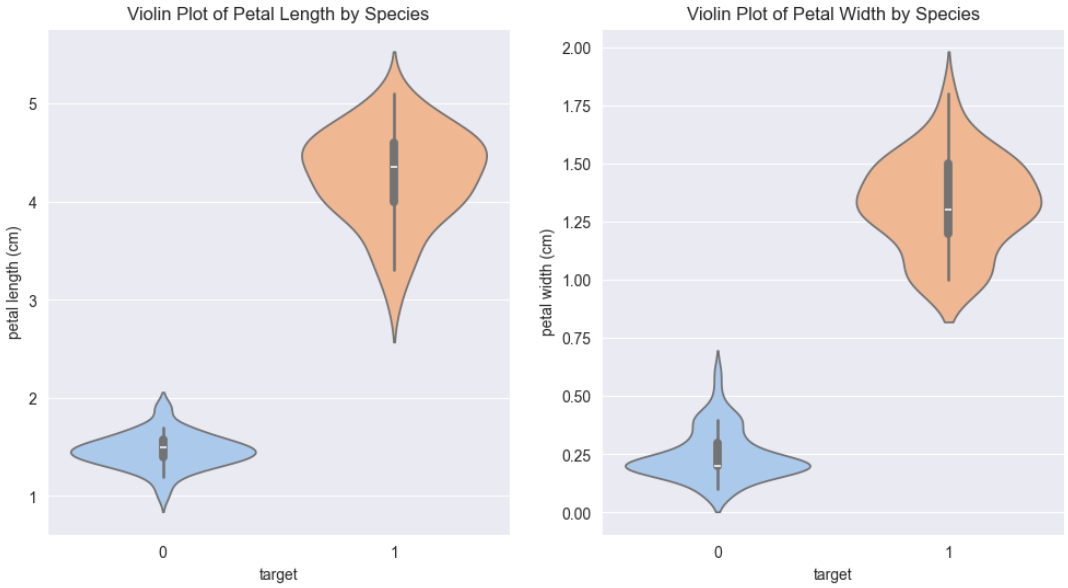
2. Histograms for petal length and petal width





- The distribution of petal length and width appears to be bimodal, reflecting the presence of two species.
- These distributions suggest that petal length varies more distinctly between species than petal width.

3. **Violin Plot Insights**:



- Violin plots provide a better insight into the distribution, showing a clear distinction in feature spread between species.
- The plots suggest 'setosa' has narrower and shorter petals overall.

4. **Pair Plot Insights**:

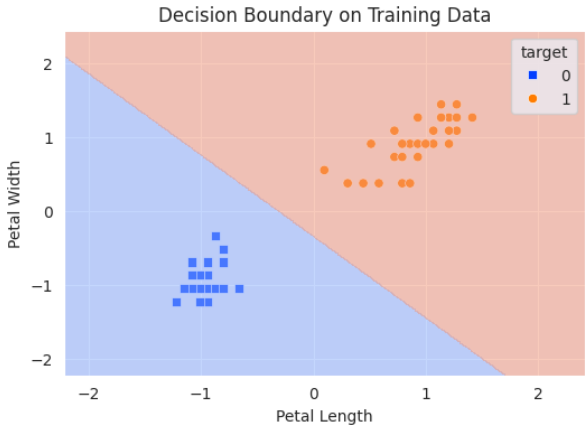
- Pair plots reveal that the separation between species based on petal measurements is quite clear and could be linearly separable to some extent.
- The diagonal plots reaffirm the bimodal nature due to the two species present.

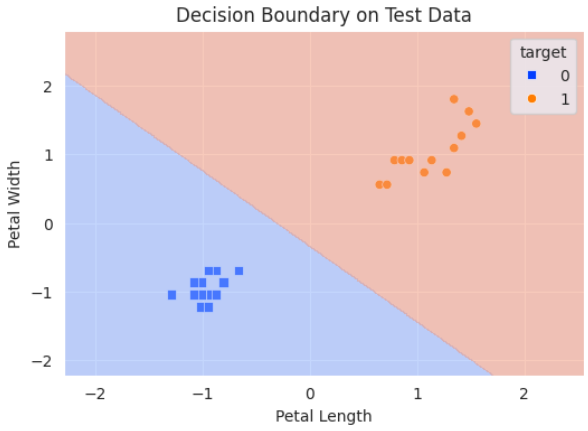
Task-1 (b):

Train a Linear Support Vector Classifier (Linear SVC) on the training data. Plot the decision boundary of the model on the train data, and also generate another plot showing a scatterplot of the test data along with the original decision boundary. (3 pts)

- An **SVC (`kernel = "linear"`) from Sklearn** was trained on the data, and the decision boundaries are as follows:

```
svc = SVC(kernel='linear')
svc.fit(X_train, y_train)
```

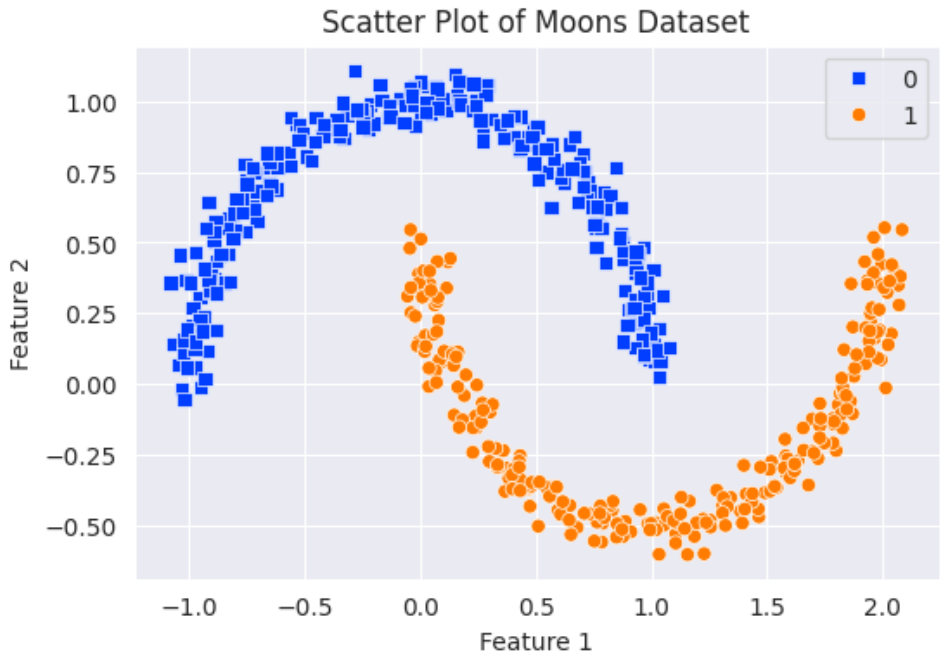




Task-2 (a):

Generate a synthetic dataset using the make moons() function from scikit-learn. Take around 500 data points, and add 5% noise (misclassifications) to the dataset. (1 pts)

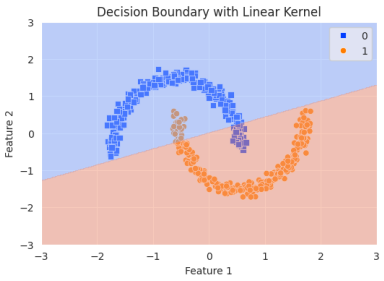
- The synthetic data was generated with the **given parameters** and it looks as follows when plotted:



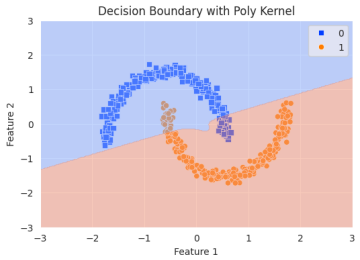
Task-2 (b):

Implement SVM models with three different kernels: Linear, Polynomial, and RBF. Plot the decision boundaries for each kernel on the synthetic dataset. Analyze and comment on the differences in decision boundaries produced by these kernels. (5 pts)

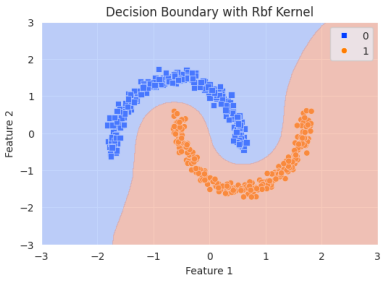
- The models were trained with the given kernels and the decision boundaries are as follows:



Linear Kernel



Polynomial Kernel



Rbf Kernel

- The first kernel used is the `linear` kernel and hence gives a linear boundary. It is obvious from the generated boundary that this kernel should be used for linearly seperable data.

- The second kernel used is the polynomial kernel and it maps the original data to a higher dimensional space while considering the interactions between the data points as new features. From the given decision boundary, while the model doesn't fit the data very well. It appears to take into account the samples it's misclassifying, hence producing a **curvature in the decision boundary.** The model can fit the data better if enough care is taken into tuning the parameters `c` and `d` which correspond to the constant term and the degree in the kernel used respectively.

- The third kernel used is the `rbf` kernel and as explained above it effectively transforms the input space into an infinite-dimensional feature space, enabling the SVM to find a hyperplane that can separate highly complex data patterns. **This kernel considers the square of the Euclidean distance between any two feature vectors, applying a negative exponential function to it. This results in a measure of similarity that decreases with distance**, making it extremely powerful for capturing varied data shapes as seen in the plotted **decision boundary.**

Task-2 (c):

Focus on the RBF kernel SVM model. Perform hyperparameter tuning to find the best values of gamma and C for this model. You can use techniques like grid search or random search. (2 pts)

- After performing hyperparameter tuning using sklearn’s Grid Search with the following parameters:

```
param_grid = {
    'C': [0.1, 1, 10, 100],
    'gamma': ['scale', 'auto', 0.1, 1, 10, 100]
}
```

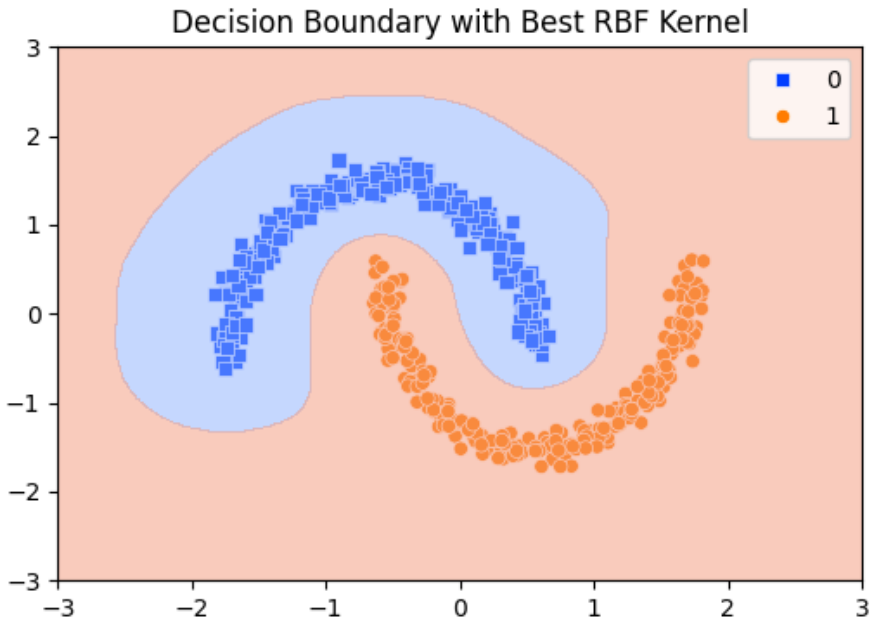
- the best performing model has the following parameters:

```
Best parameters: {'C': 0.1, 'gamma': 10}
```

Task-2 (d):

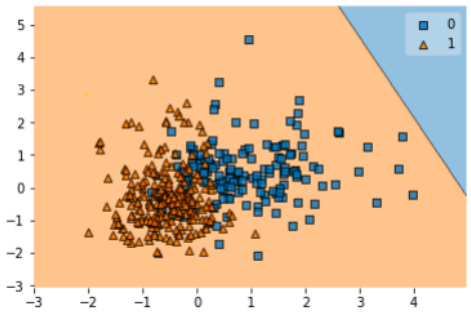
Plot the decision boundary for the RBF kernel SVM with the best Hyperparameters. Explain the impact of the selected gamma and C values on the model’s performance and decision boundary. Note: Ensure to complete each task thoroughly and document your findings in the lab report.

The decision boundary for the model trained with the above tuned parameters looks like follows:

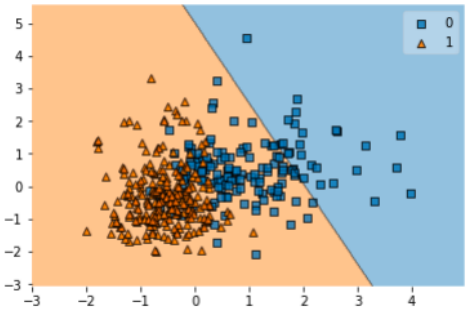


The model’s performance in using the rbf kernel to effectively plot a decision boundary is impacted by the values of `gamma` and `c` in the following ways:

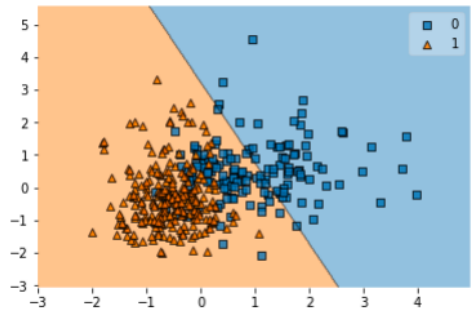
- **C: This parameter controls the effective width of the error margin between the support vectors, it controls the width of the acceptable error thus effectively increasing the sensitivity of the boundary to capture more outliers.**



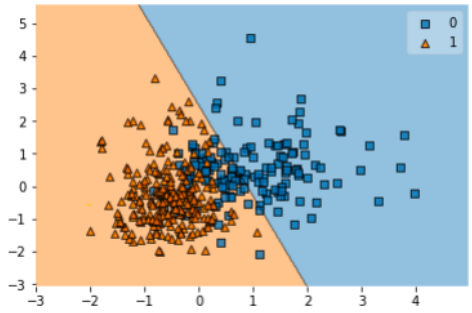
C = 0.001
Accuracy: 62.6%



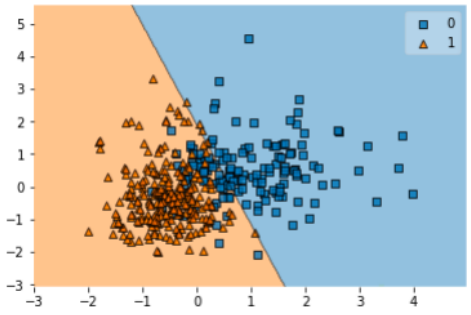
C = 0.002
Accuracy: 71.3%



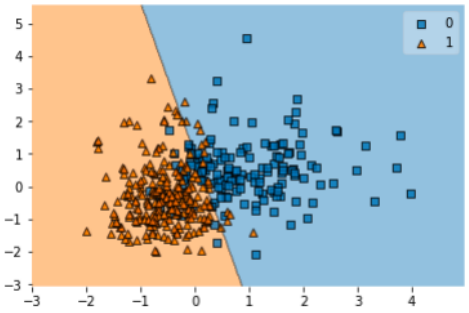
C = 0.003
Accuracy: 81.3%



C = 0.005
Accuracy: 81.7%



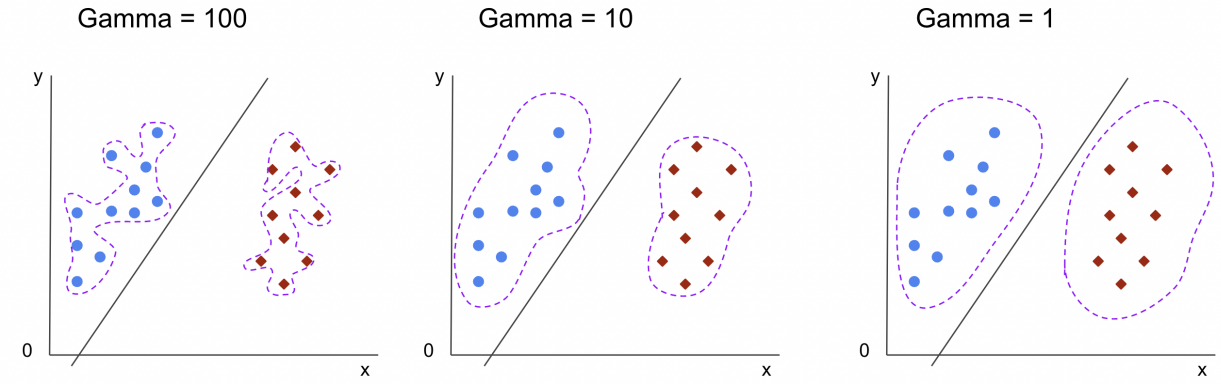
C = 0.01
Accuracy: 89.5%



C = 10.0
Accuracy: 90.1%

- **Gamma: This parameter effectively controls the spread of the data, and acts as a force which controls the pull/push of the decision boundary**

towards the data space. This parameter tunes the variance or the spread of the data thus effectively pulling/pushing the boundary towards the data space while ignoring outliers.



- **OBSERVATION: The model seems to performing better with a lower value of C which means we're keeping the error margin to a higher degree, while the gamma value is on the higher side, which means the learned decision boundary is more compact towards the data.** This shows that, generally a lower value of C and a relatively higher value of gamma is better to learn the decision boundary of a highly defined pattern as seen in our dataset of two moons. On a more general note, proper hyper parameter turning is required to appropriately fit the data depending on the type of underlying patterns.

References:

- SVM:
  - https://shuzhanfan.github.io/2018/05/understanding-mathematics-behind-support-vector-machines/

- https://ankitnitjsr13.medium.com/math-behind-support-vector-machine-svm-5e7376d0ee4d
- https://www.youtube.com/watch?v=efR1C6CvhmE

- K-Means:
  - https://www.youtube.com/watch?v=4b5d3muPQmA
  - https://www.geeksforgeeks.org/image-compression-using-k-means-clustering/

- THE END -