PA5: K-Means and SVM

Colab Link: PA5_B22CS090.ipynb

Question 1:

Note: I have implemented the K-means algorithm by referring to the scikit-learn documentation for K-means. Hence, the flow of the algorithm might look similar to that of the scikit-learn K-means algorithm.

- To implement K-means from scratch, a class **Kmeans_scratch** has been implemented, which contains all the necessary functions required for image segmentation.
- An image is read and converted to a numpy array. In this case, the image is of dimensions **512 x 512 x 3**.

Test image

- The height and width of the image is flattened to transform the array into an array of dimensions **262,144 x 3**. The pixel values are also scaled by dividing by 255.
- Initially, cluster centres are assigned randomly.
- **calc_distance()** function is used to calculate distance between every pixel and every cluster centre.
- For every pixel, cluster centre which is at minimal distance is obtained.
- **update_cluster_centers()** updates the present cluster centres. For example, for a cluster centre A, the new cluster centre in place of A is the mean of the data points which are closer to A than any other cluster centre.
- After cluster centres are updated, the Frobenius norm of the difference of the new and old cluster centres is calculated. If the value is less than a certain tolerance / threshold, the algorithm is said to have **converged** and training ends.
- If convergence doesn't occur, training runs for **max_iter** iterations.

- This overall procedure happens for **n_init** times, where n_init determines the number of times cluster centres will be initialised randomly from the start, and will be ran for max_iter times at max.
- **Inertia term** helps to capture the best cluster centres along with best assigned labels. Lower the inertia term, better is the result.
  Inertia = scaled summation of minimum distances between every datapoint and a cluster centre.
- Cluster centres and labels corresponding to the lowest inertia value is returned by **mykmeans** function.
- The following pictures show images obtained from scratch implementation and from scikit-learn's implementation.

Number of clusters used (values of **k**, in my implementation) = **[2, 4, 6, 8, 10]**
One cluster captures one single colour.

Kmeans from scratch, k = 2

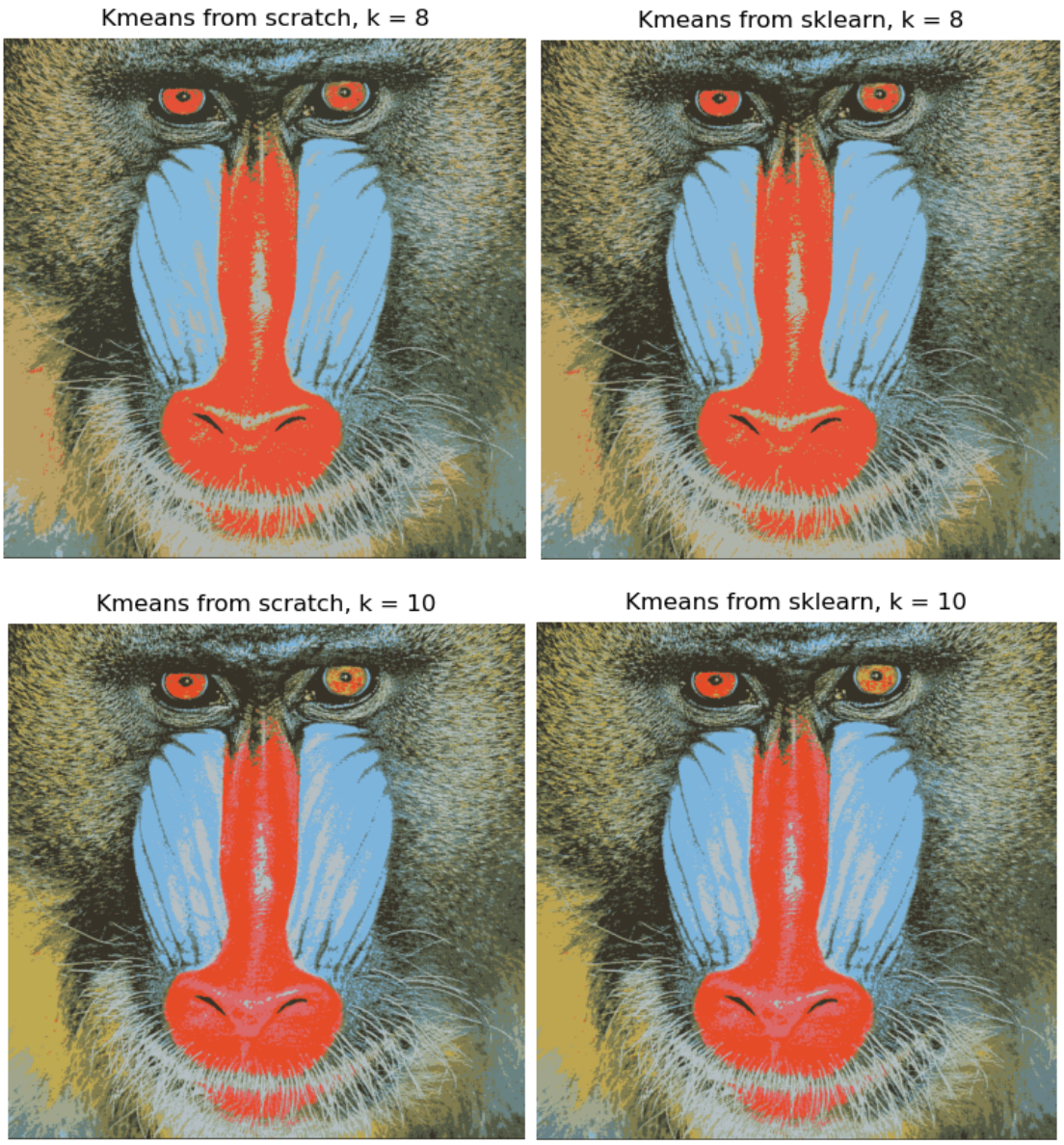
Kmeans from sklearn, k = 2

Kmeans from scratch, k = 4

Kmeans from sklearn, k = 4

Kmeans from scratch, k = 6

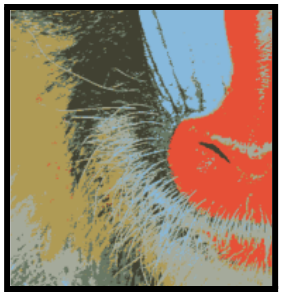
Kmeans from sklearn, k = 6



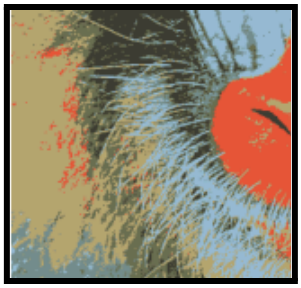
Differences:

There are very subtle differences visually, when comparing my implementation with that of sklearn's.

For example, for **k = 6**, in sklearn's image, it can be observed that there is a **slight red patch towards the bottom left**, which is not present for my implementation.



My implementation



Scikit-Learn

Differences observed from data:

Subtle changes in the colours (colour channel values) used for K-Means.

Note: The columns are ordered as **R G B** and **order of rows may vary**.

Values of k (no of clusters)	My Implementation	Scikit-Learn
2	[[124 96 70] [155 173 171]]	[[124 96 70] [155 173 171]]
4	[[227 95 60] [140 145 113] [152 184 202] [77 80 63]]	[[227 93 60] [152 184 203] [78 81 64] [142 147 114]]
6	[[108 121 103] [168 173 157] [232 83 57] [69 69 54] [178 156 86] [139 186 220]]	[[151 188 211] [106 110 81] [231 85 56] [183 166 114] [119 143 140] [63 62 50]]
8	[[232 83 56] [87 96 78] [58 55 45] [179 184 176] [132 125 80] [134 186 224] [187 164 101] [118 142 139]]	[[187 163 99] [135 187 224] [86 95 77] [232 83 56] [179 183 174] [129 125 82] [58 54 44] [117 142 141]]
10	[[57 53 44] [127 183 223] [132 126 80] [160 168 143] [234 78 44] [106 131 133] [222 106 107] [87 93 73] [181 193 197] [191 170 82]]	[[162 169 142] [130 126 82] [234 78 44] [181 193 197] [87 93 73] [106 132 136] [191 169 80] [222 106 107] [57 53 44] [127 184 223]]

Incorporating Spatial Coherence:

Idea reference: k-Means Segmentation | Image Segmentation

- Spatial coherence has been incorporated by adding 2 features to each datapoint, which are the **x and y coordinates**, as present in the original image array.
- Previously, distance was calculated by considering 3 features (3 colour channels). Now, **3 + 2 = 5 features** will be considered to calculate Euclidean distance. Rest of the algorithm will remain the same.
- I have added a scaling feature which basically sets the importance given to a feature while calculating the distance.
  Example: In my code, there are the following lines:

```
self.coord[i][j][0] = self.coord[i][j][0] * np.sqrt(0.25)
self.coord[i][j][1] = self.coord[i][j][1] * np.sqrt(0.25)
```

```
x = x * np.sqrt(0.75)
```

I assign a **sqrt(0.25)** weight to **coordinate features** and **sqrt(0.75)** to the **colour features**. So, when Euclidean distance is calculated, the square roots are squared, thus I get 0.25 scaling for the coordinate features, and 0.75 scaling for the colour features. After trial and error, this combination gives an acceptable image.

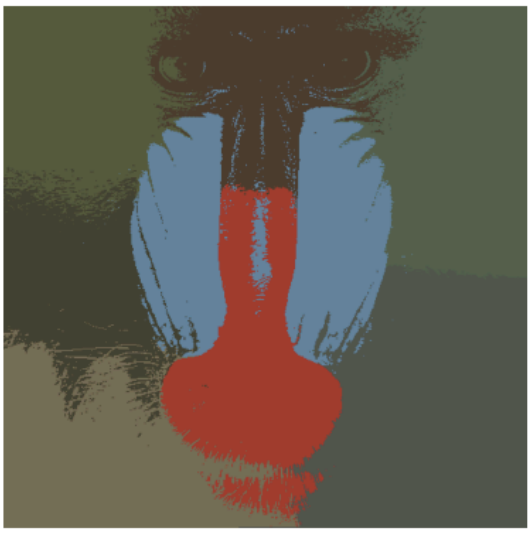
Kmeans Spatial Coherence (Balanced importance), k = 8

Kmeans from scratch, k = 8

It can be clearly seen that the **bottom right part is blacker** and filled more evenly with a single colour, which is not the case for K-means without spatial coherence. Hence, having coordinates play a part in assigning similar colours to close coordinates.

Giving too much importance to distance features (increases to extent of assigning same colours to neighbours) and too low importance to colour features produces **‘segments’** or **‘blocks’** of colours and we lose the image structure.

Kmeans Spatial Coherence (High Coordinate importance), k = 8

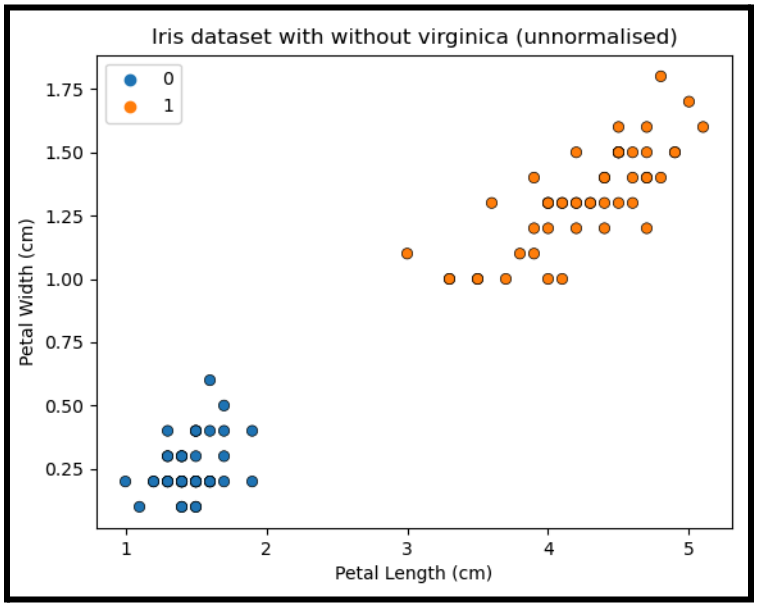


On the other hand, giving too much importance to colour features and negligible importance to coordinate features results in **almost similar image** as obtained without spatial coherence.

Question 2:

Task 1

- Firstly, the iris dataset is loaded and **80-20 splitting** is done for training and test set.
- Only the features 'petal length' and 'petal width', and classes 'setosa' and 'versicolor' have been used.

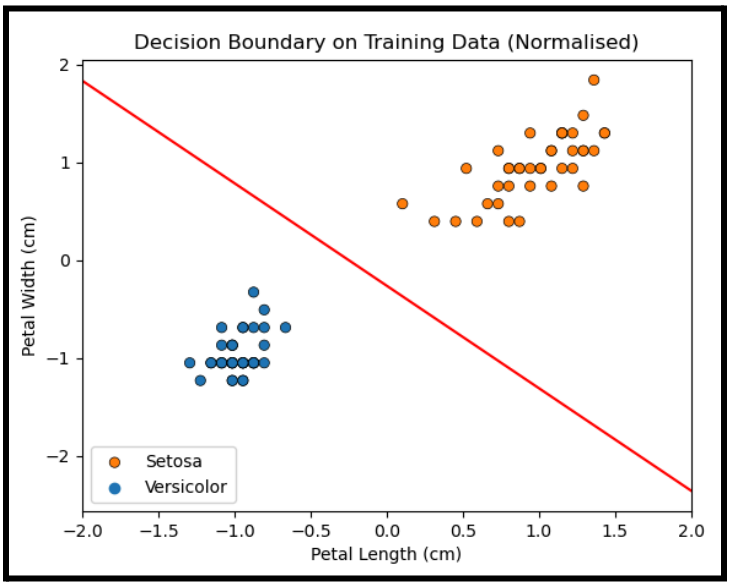


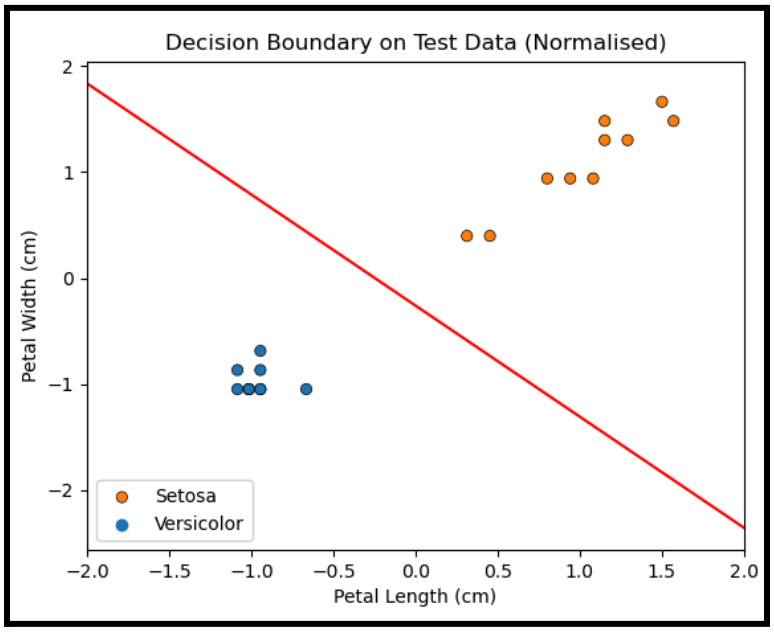
- After splitting, the dataset has been normalised using **StandardScaler()**.
- **LinearSVC()** model is initialised with default hyperparameter values **(C = 1.0)**, and this model is trained with the normalised training set.
- The decision boundaries are obtained by a function implemented by me, named **decision_boundary_plotter()**.
  For linear SVM having 2 features, the equation of the decision boundary is as follows:

  $$x_2 = -(w_1 / w_2) * x_1 - (b / w_2)$$

  x_2 = value of feature 2
  x_1 = value of feature 1
  w_i = coefficients in SVM model **(i = {1, 2})**
  b = intercept in SVM model

- The following is the decision boundary obtained:

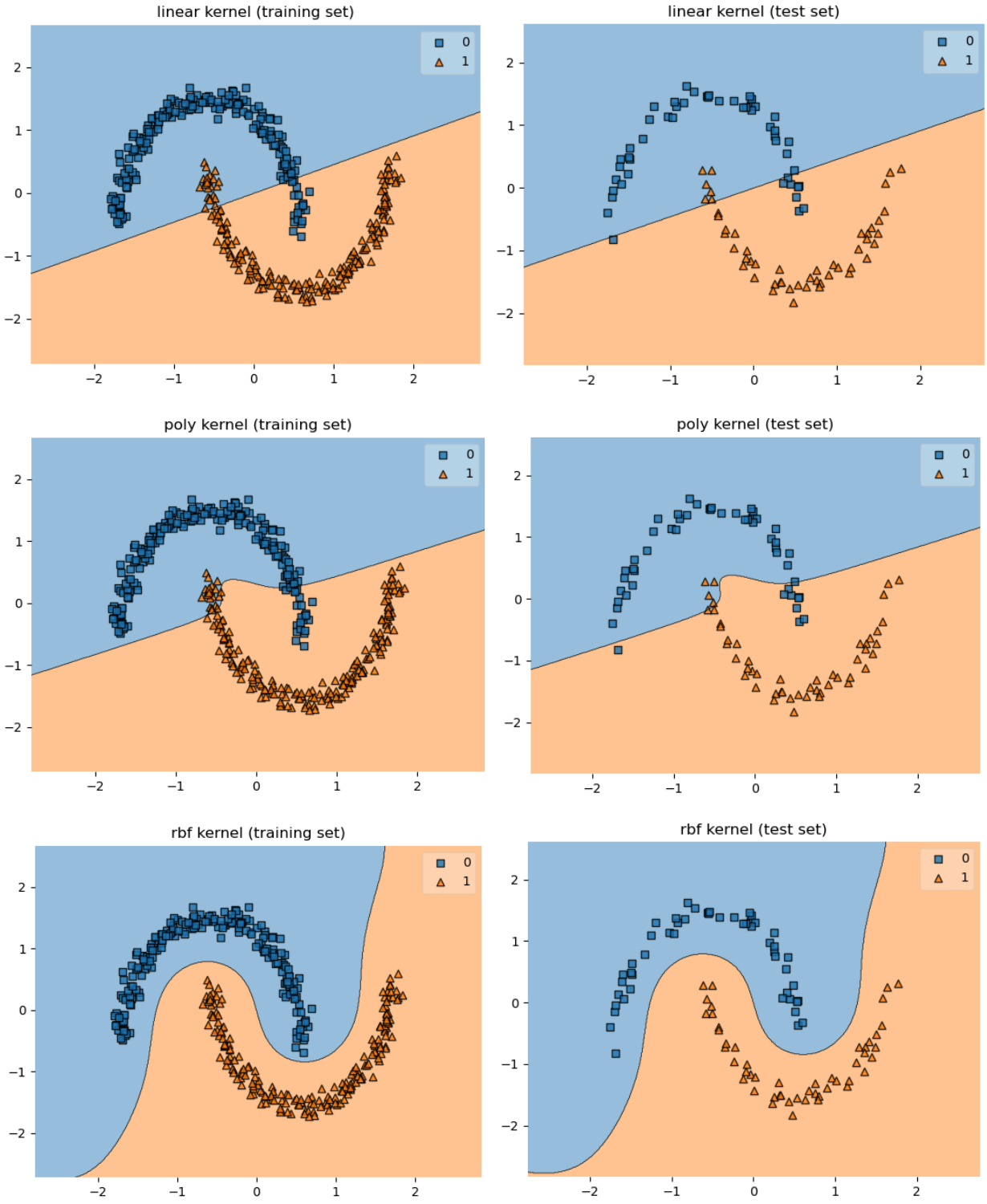




Task 2

- A synthetic dataset is generated using **make_moons()**, having **5% noise**.
- The training set contains **400** data points and the test set contains **100** data points **(80-20 splitting).**
- The training and the test set have been normalised using **StandardScaler().**
- **SVC()** model with 3 kernels have been used respectively, with default hyperparameter values initially.
- **Kernels used:** Linear, Polynomial, Radial Basis Function (RBF)

The following are the decision boundaries:



Accuracies obtained:

Kernel	Training set (fraction)	Test set (fraction)
Linear	0.89	0.85
Polynomial	0.8875	0.87
RBF	1.0	1.0

Types of decision boundaries:

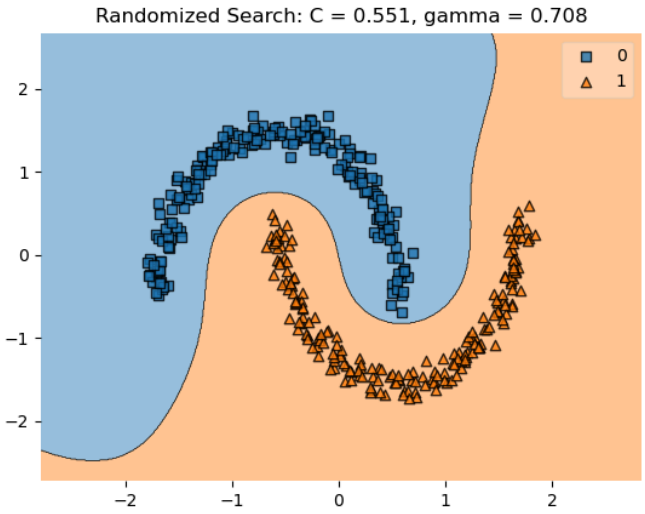
- Linear: Straight line
- Polynomial: Can be non-linear and take the form of curves. It depends on the **degree of the polynomial** (default = 3). If degree = 1, it is basically a linear kernel.
- RBF: Non-linear can is more complex compared to linear and polynomial kernels. Depending on the values of **C** and **gamma**, it can take any shape.

Performing hyperparameter tuning of RBF kernel SVM using RandomizedSearchCV:

- RandomizedSearchCV has been used to perform hyperparameter tuning of this classifier.
- **5 fold cross validation** is used in this case. This means that while training, the training set of 400 samples will be divided into **5 equal sets**, each having size of **80**. 4 sets will be used for training, and 1 for validation. This will be done for every possible combination. The one with the best validation score will be considered.
- Value of C is sampled from a **uniform distribution ranging from 0 to 1**, and so is gamma.
- This range of values produces a great possible decision boundary, i.e, the decision boundary does not overfit the training data.

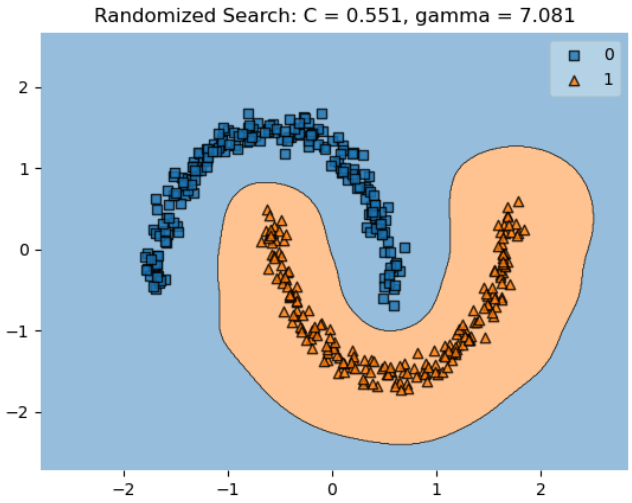
Obtained hyperparameters:

```
{'C': 0.5507979025745755, 'gamma': 0.7081478226181048}
```



- However, unconstrained sampling range for hyperparameters may lead to overfitting of the training data.
  For example:
  C is sampled from a **uniform distribution ranging from 0 to 1**.
  gamma is sampled from a **uniform distribution ranging from 0 to 10**.

  The obtained decision boundary clearly overfits the training data.

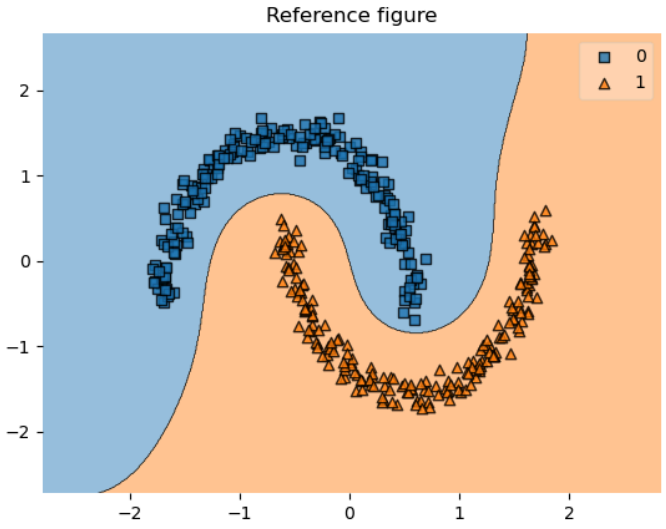


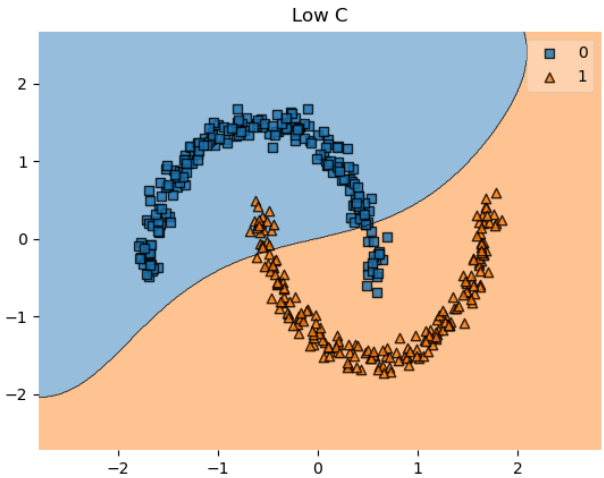
Impact of C and gamma:

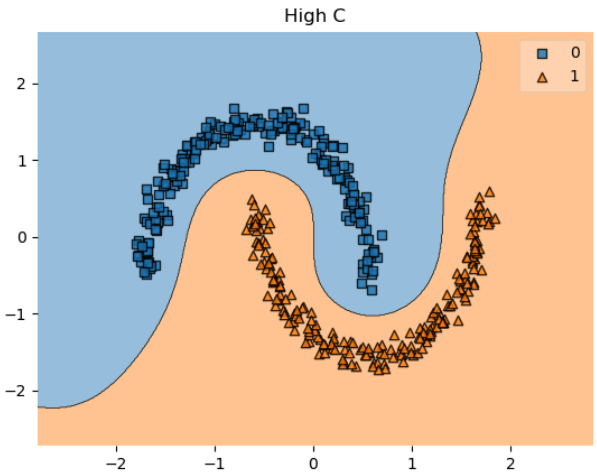
- C is a **regularisation parameter**, which is **inversely proportional to regularisation strength**.
- **Smaller values of C** encourages **larger margins** and allows for **more misclassifications**.

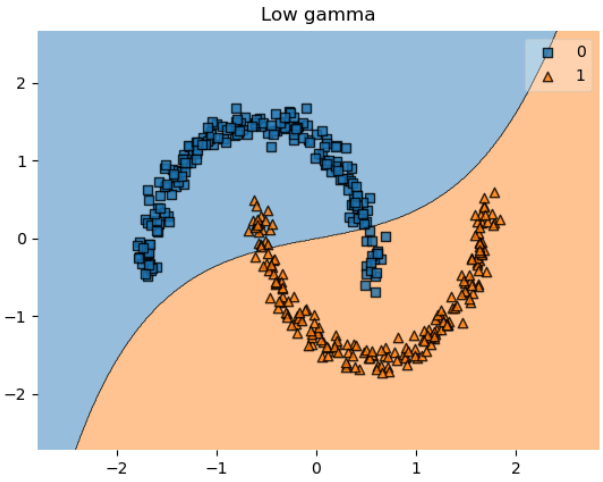
- **Higher values of C** penalises misclassifications more, thereby having **smaller margins**, but too high can result in **overfitting**.
- **Gamma** controls the **influence of individual training samples on the decision boundary**.
- **Smaller gamma** value has a **larger kernel radius**, indicating that each training example has a **wider influence** on the decision boundary.
- **Higher gamma** values make the decision boundary **more dependent on the points closest to it**, which may result in **overfitting**.
- Higher gamma values lead to more **complex** and **wiggly** decision boundaries.

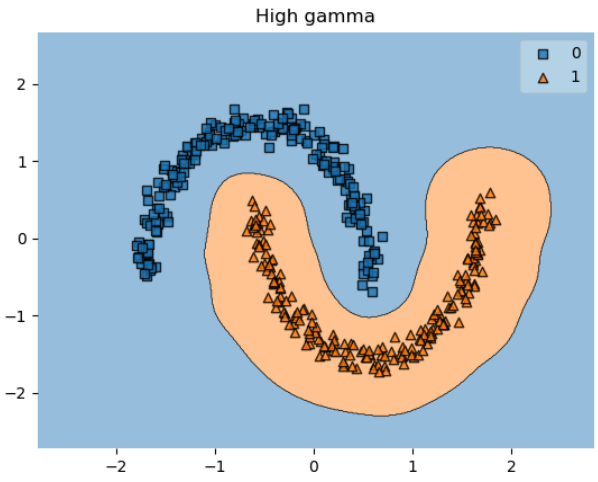
Showing using figures:











Even higher gamma showing, how decision boundary is more dependent on points closer to it, and is more complex and wiggly