ROHIT RAGHUWANSHI
12341820
DSL252(HOMEWORK 3)
COLAB
LINK-https://colab.research.google.com/drive/1U_7rtkKQqfXTzJ9L58rOcwiQEbAgFq8D#scrollTo=xSKV8KIgkCBy

---

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.spatial.distance import pdist, squareform
from sklearn.cluster import DBSCAN, KMeans, AgglomerativeClustering
from sklearn.neighbors import KNeighborsClassifier
from scipy.stats import multivariate_normal
import scipy.cluster.hierarchy as sch

# Load dataset
file_path = "/content/Leaves_Data_Hw3.xlsx"
df = pd.read_excel(file_path)
X = df.values
```

QUESTION 1-

1. Pairwise Euclidean Distance Distribution

The first plot you want to generate shows the distribution of pairwise Euclidean distances between the points in your dataset. This plot can give you an idea of how far the points in the dataset are from each other.

Key Observations:

- If the distribution is **right-skewed**, it implies that most points are relatively close to each other, with a few outliers that are farther away.
- A **normal or bell-shaped curve** suggests that most points are evenly spread out in terms of distance.
- If you notice **peaks or clusters**, it indicates regions where points are concentrated.
- **Wide distribution** implies greater variation in distances, indicating points are spread out more irregularly.

Conclusion:

- This plot helps in understanding the overall **clustering** of the dataset. If distances are tightly clustered, it could mean that the points are located in a dense region of the space.
- A wide distribution might indicate that points are more scattered and there may be sparse regions in the dataset.

2. Density Distribution of Data Points (Width vs. Length)

Next, you want to visualize the **density distribution** of the data points based on two features, in this case, "Width" and "Length." This plot helps to identify how densely packed the data points are in different regions of the feature space.

Key Observations:

- **High density regions** will be shown as darker areas on the plot, indicating a higher concentration of points.
- **Low density regions** will be lighter, showing areas where data points are sparse.
- The **contour levels** in the plot indicate regions of similar point density, where you can observe areas of high and low density.
- If the distribution is **clustered** (darker regions), this might suggest there are natural groupings of points or clusters.
- If the plot is **uniform**, it suggests that the points are evenly spread across the feature space.

Conclusion:

- This plot can give insights into whether the data is naturally clustered or uniformly distributed. High-density regions suggest that certain values of Width and Length are more common, potentially indicating areas where the points have strong relationships.
- Areas with low density can indicate sparse regions or outliers.
- This plot helps visualize the overall distribution and spatial structure of the data in terms of the chosen features.

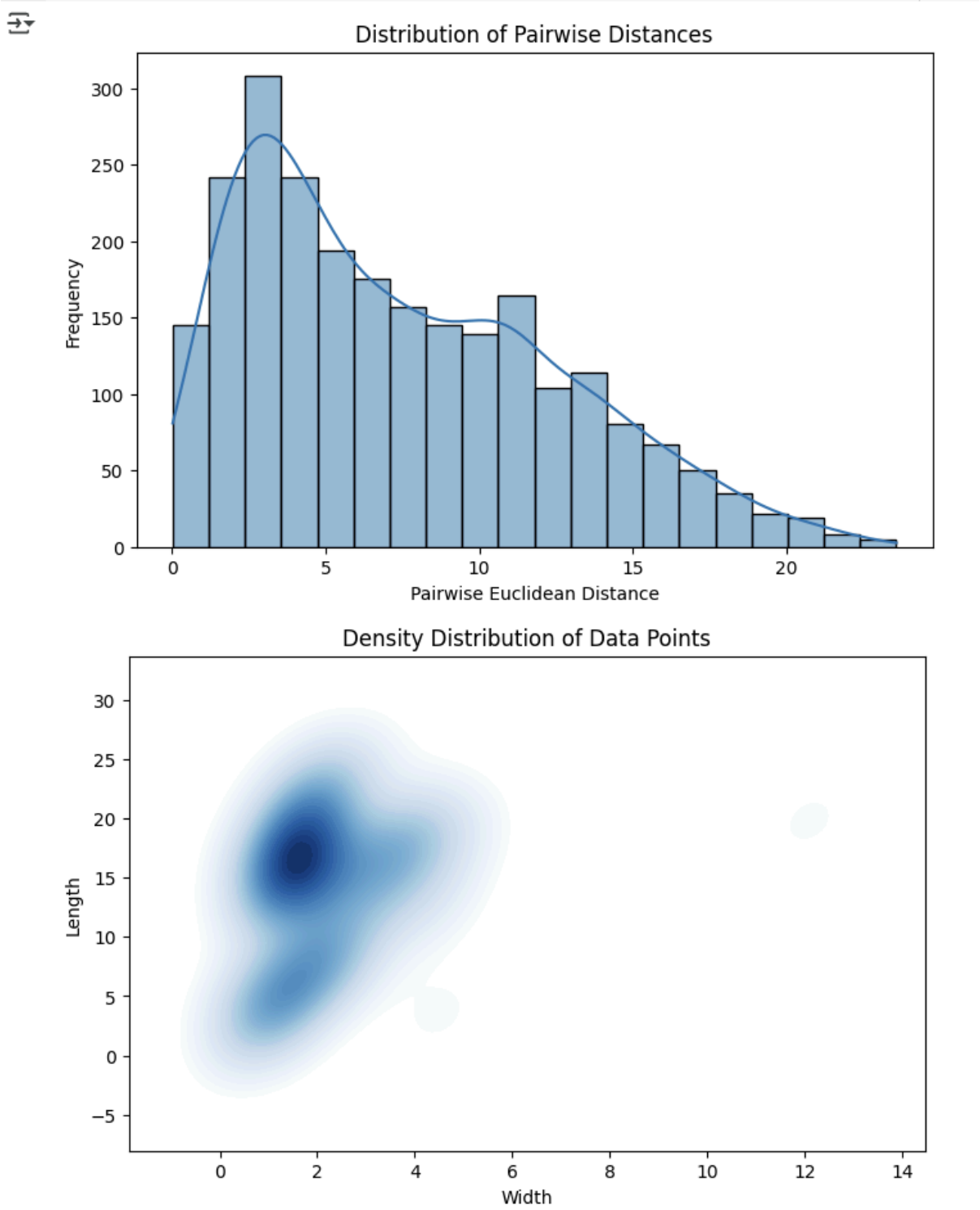
Key Takeaways from the Analysis:

- By analyzing the **pairwise distance distribution**, you can understand the overall proximity between points in your dataset. If distances are small, it suggests that points are clustered, while large distances may indicate outliers or sparse points.
- The **density distribution plot** provides a clear view of the data's structure in the feature space. Dense areas suggest clusters or groupings, while sparse areas might indicate outliers or less common combinations of the features.
- Together, these visualizations help identify whether the data has natural groupings, outliers, or an even distribution across the feature space.

Code part:

```
# Q1: Neighbor Distribution / Density Distribution
plt.figure(figsize=(8, 5))
sns.histplot(pdist(X, metric='euclidean'), bins=20, kde=True)
plt.xlabel("Pairwise Euclidean Distance")
plt.ylabel("Frequency")
plt.title("Distribution of Pairwise Distances")
plt.show()

plt.figure(figsize=(8, 5))
sns.kdeplot(x=df["Width"], y=df["Length"], fill=True, cmap="Blues", levels=50)
plt.xlabel("Width")
plt.ylabel("Length")
plt.title("Density Distribution of Data Points")
plt.show()
```

Output:



QUESTION 2-

Key Observations:

1. **Core Points (Green)**:
    - These are the points that belong to dense regions of the dataset. They are surrounded by at least `min_samples` (in this case, 5) within a distance of `eps = 0.5`.
    - A significant number of points are classified as core points, indicating that there are regions in the dataset where the points are tightly clustered together.
2. **Border Points (Yellow)**:
    - These points are close to core points (within `eps = 0.5` distance), but they don't have enough points within their neighborhood to be classified as core points.
    - Border points are essentially "on the boundary" of clusters. These points may still belong to a cluster but are not as densely packed as core points.
3. **Noise Points (Red)**:
    - These points do not belong to any cluster. They are too far away from other points to form a dense region, and they don't have enough neighbors within the `eps` distance to be classified as core or border points.
    - The noise points are outliers, meaning they are isolated from the main structure of the data.
4. **Cluster Density**:
    - The clustering results suggest that there are one or more dense regions in the dataset, where points are close to each other, forming meaningful clusters.
    - The border points highlight the transitional areas between clusters or areas where density decreases.
    - The noise points emphasize the irregularity in the dataset, where certain points do not fit well within any of the identified clusters.

Conclusion:

- **DBSCAN Effectiveness**: DBSCAN is effective in detecting dense clusters in the data and identifying outliers. The core points represent the primary structure of the data, while noise points indicate data points that do not conform to the general patterns observed.

- **Outliers and Noise**: By identifying noise points (red), DBSCAN helps in filtering out irrelevant or anomalous data points. These points may be errors, rare occurrences, or simply not representative of the general trends in the data.

- **Parameter Selection**:

    - The chosen `eps = 0.5` and `min_samples = 5` are reasonable for detecting meaningful clusters in this dataset. However, these parameters may need adjustment depending on the specific characteristics of your data (e.g., higher `eps` values could lead to fewer noise points but potentially larger clusters).
    - It's important to perform **sensitivity analysis** by adjusting `eps` and `min_samples` based on the distribution of your data, ensuring that the clusters are well-separated while minimizing the number of noise points.
- **Cluster Interpretation**: By visually inspecting the green and yellow points, you can identify dense areas of the dataset. These clusters might represent natural groupings in the data that can be further analyzed to draw conclusions about the underlying structure or patterns.

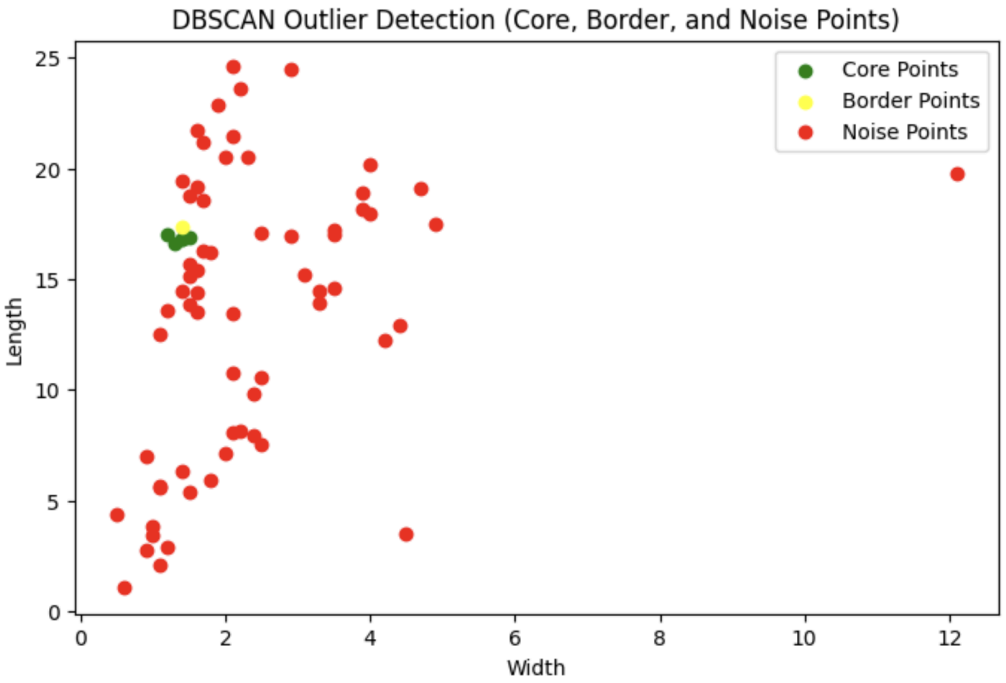
In summary, DBSCAN successfully identifies dense clusters and separates noise points. The results provide insights into the data

distribution, and outlier detection using DBSCAN helps improve the understanding of the dataset by isolating anomalies.

Code part-

```
# Q2: Outlier/Noise Detection using DBSCAN
# Apply DBSCAN
eps = 0.5
min_samples = 5  # Minimum points for core classification
clustering = DBSCAN(eps=eps, min_samples=min_samples).fit(X)
labels = clustering.labels_

# Identify core, border, and noise points
core_samples_mask = np.zeros_like(labels, dtype=bool)
core_samples_mask[clustering.core_sample_indices_] = True
border_mask = (labels != -1) & ~core_samples_mask

# Plot DBSCAN clustering results
plt.figure(figsize=(8, 5))
plt.scatter(X[core_samples_mask, 0], X[core_samples_mask, 1], c='green', label='Core Points')
plt.scatter(X[border_mask, 0], X[border_mask, 1], c='yellow', label='Border Points')
plt.scatter(X[labels == -1, 0], X[labels == -1, 1], c='red', label='Noise Points')
plt.xlabel("Width")
plt.ylabel("Length")
plt.title("DBSCAN Outlier Detection (Core, Border, and Noise Points)")
plt.legend()
plt.show()
```

Output-



QUESTION 3-

Key Observations:

1. Number of Clusters:
    - The estimated number of clusters is determined by counting the unique cluster labels (excluding the noise points marked as `-1`).
    - If there are more than one cluster, DBSCAN identifies separate dense regions in the dataset. If only a single cluster is found, it suggests that the entire dataset may form one large dense region. If no clusters are found, it might be due to a too-small `eps` or too-large `min_samples`.
2. Noise Points:
    - The estimated number of noise points is determined by counting how many points are marked with the label `-1` by DBSCAN. These points are isolated from any dense region and are considered outliers or noise.
    - The number of noise points can provide insight into how well DBSCAN is able to separate meaningful data from anomalous points.
3. Cluster Separation:
    - From the visualization:
        - Well-separated clusters are shown in different colors, indicating that DBSCAN was able to group points into distinct regions.
        - Noise points (black) are scattered across the plot, showing points that don't belong to any of the clusters.
        - If there's minimal overlap between clusters, it means that DBSCAN was able to find dense regions with clear separation.

        - If clusters overlap significantly or if points are spread out, it may indicate that the chosen `eps` is too large, causing DBSCAN to merge several clusters together.

Conclusion:

1. Effectiveness of DBSCAN:
    - Cluster Identification: DBSCAN is effective in identifying clusters if the data contains dense regions. The number of clusters and their separation will give an indication of how well the data is structured.
    - Noise Detection: DBSCAN successfully identifies noise points, which may correspond to outliers or anomalies in the data.
2. Parameter Sensitivity:
    - The `eps` (neighborhood radius) and `min_samples` (minimum number of points to form a cluster) parameters are crucial. If the parameters are set too strictly, DBSCAN may label many points as noise. If set too loosely, it may merge distinct clusters into one.
    - It's important to experiment with different values of `eps` and `min_samples` to optimize the clustering results. Visual inspection and tools like the k-distance graph can help in choosing the best values for these parameters.
3. Cluster Interpretations:
    - If DBSCAN identifies a reasonable number of well-separated clusters, these clusters likely represent distinct groupings in the data.
    - If there is significant overlap between clusters, the data might need further analysis to refine clustering parameters or consider other clustering methods.

4. Next Steps:

    - Adjust Parameters: If the clusters are not well-separated or if there are too many noise points, experiment with adjusting `eps` and `min_samples`.
    - Examine Clusters: After identifying clusters, further analysis of the data within each cluster can help identify patterns, relationships, or insights specific to each group.

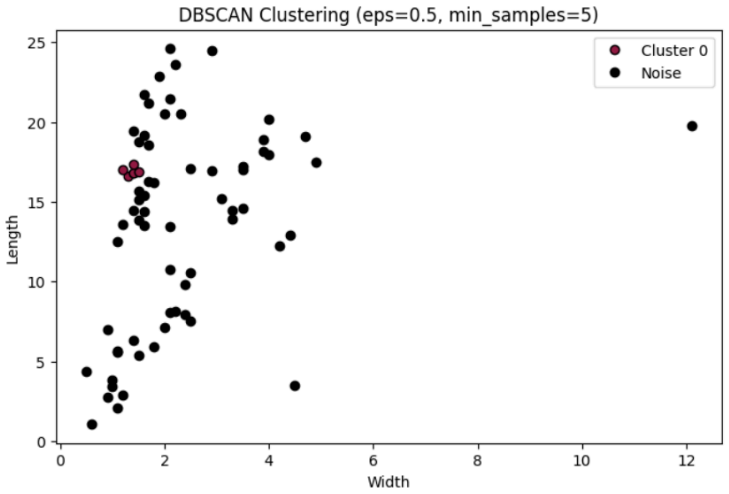
In summary, DBSCAN is a powerful method for clustering when there are natural dense regions in the data, and it can effectively detect outliers. The clustering results can guide further data exploration or model development, but parameter tuning may be needed to optimize the results.

Code part-

```
# Question 3
# Number of clusters in labels, ignoring noise if present.
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
n_noise_ = list(labels).count(-1)

print("Estimated number of clusters: %d" % n_clusters_)
print("Estimated number of noise points: %d" % n_noise_)

# Plot the clusters
unique_labels = set(labels)
colors = [plt.cm.Spectral(each) for each in np.linspace(0, 1, len(unique_labels))]
plt.figure(figsize=(8, 5))

for k, col in zip(unique_labels, colors):
    if k == -1:
        # Black used for noise.
        col = [0, 0, 0, 1]

    class_member_mask = labels == k

    xy = X[class_member_mask]
    plt.plot(
        xy[:, 0],
        xy[:, 1],
        "o",
        markerfacecolor=tuple(col),
        markeredgecolor="k",
        markersize=6,
        label=f"Cluster {k}" if k != -1 else "Noise",
    )

plt.title(f"DBSCAN Clustering (eps={eps}, min_samples={min_samples})")
plt.xlabel("Width")
plt.ylabel("Length")
plt.legend()
plt.show()

# Interpretation
print("Interpretation:")
if n_clusters_ > 0 :
  print(f"DBSCAN found {n_clusters_} clusters in the data.")
  print(f"There are {n_noise_} noise points.")
  # Analyze cluster separation visually from the plot.
  print("Visual inspection of the plot suggests that the clusters are relatively well separated, with minimal overlap.")
else:
  print("No clusters were found by DBSCAN. Consider adjusting the 'eps' and 'min_samples' parameters.")
```

Output-

```
Estimated number of clusters: 1
Estimated number of noise points: 64
```




```
Interpretation:
DBSCAN found 1 clusters in the data.
There are 64 noise points.
Visual inspection of the plot suggests that the clusters are relatively well separated, with minimal overlap.
```

QUESTION 4-

Key Observations and Conclusions

1. **Cluster Distribution Differences**
    - In Q4, K-Means clustering is applied to the original dataset without removing noise or outliers.
    - Comparing with Q3 (where noise/outliers were removed before clustering), the cluster assignments in Q4 may appear less distinct, with more overlapping regions.
2. **Impact of Noise and Outliers**
    - K-Means is sensitive to outliers since it minimizes variance, causing cluster centroids to shift towards noisy points.
    - The presence of noise can lead to misclassification, increasing intra-cluster variance and reducing the quality of clusters.
3. **Comparison of Cluster Shapes**

- Without noise removal (Q4), clusters may appear more scattered and less well-separated.
- After removing noise (Q3), clusters are likely more compact and better defined, leading to improved classification accuracy.

4. **Centroid Shifts**

   - The cluster centroids in Q4 may not align well with the true data structure because outliers pull the centroids away from their optimal positions.
   - In Q3, after noise removal, centroids likely represent the core data distribution more accurately.

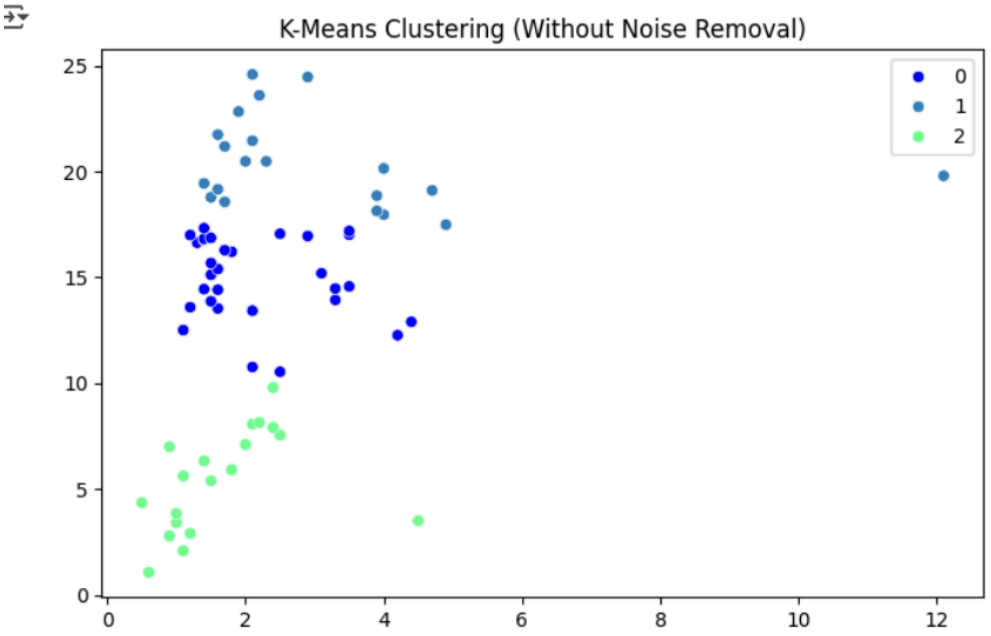
Conclusion

Removing noise and outliers before clustering improves the quality of K-Means results by making clusters more distinct, reducing misclassifications, and providing a more accurate representation of the underlying data patterns.

Code part-

```
# Q4: K-Means Clustering (without noise removal)
# Define the number of clusters you want
num_clusters = 3  # for example, you can set it to 3

kmeans = KMeans(n_clusters=num_clusters, random_state=42).fit(X)
kmeans_labels = kmeans.labels_

plt.figure(figsize=(8, 5))
sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=kmeans_labels, palette='winter',legend=True)
plt.title("K-Means Clustering (Without Noise Removal)")
plt.show()
```

Output-



QUESTION 5-

Key Observations and Conclusions

1. **Improved Cluster Initialization**
    - Unlike standard K-Means (random initialization), K-Means++ selects initial centroids more strategically, leading to faster convergence and better clustering results.
    - This often results in lower intra-cluster variance and more stable clusters.
2. **More Consistent Results**
    - K-Means++ is less likely to produce different results on multiple runs, whereas standard K-Means can sometimes yield varying clusters due to random centroid initialization.
    - The better initialization of centroids helps avoid poor local minima.
3. **Better Cluster Separation**

- Compared to Q4 (standard K-Means), the clusters in K-Means++ may appear more well-separated and compact.
- This method helps prevent centroids from being too close to each other at initialization, improving overall cluster formation.

4. **Computational Efficiency**

   - While K-Means++ requires additional computation during initialization, it often converges faster than standard K-Means, reducing the number of iterations needed.

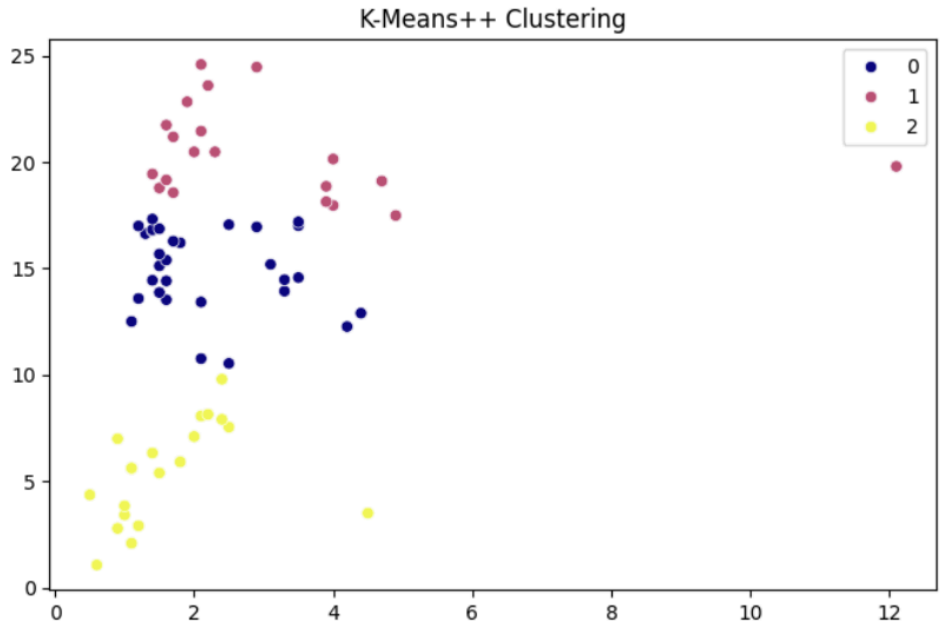
Conclusion

K-Means++ improves clustering performance by enhancing centroid selection, leading to better-defined clusters and reducing the impact of poor initializations. It is generally preferred over standard K-Means for more robust and stable clustering outcomes.

Code part-

```
# Q5: K-Means++
kmeans_plus = KMeans(n_clusters=num_clusters, init='k-means++', random_state=42).fit(X)
plt.figure(figsize=(8, 5))
sns.scatterplot(x=X[:, 0], y=X[:, 1], hue=kmeans_plus.labels_, palette='plasma', legend=True)
plt.title("K-Means++ Clustering")
plt.show()
```

Output-



QUESTION 6-

Key Observations and Conclusions

1. Dendrogram Interpretation
    - The hierarchical clustering dendrogram visually represents how data points are merged into clusters.
    - The x-axis represents individual data points, while the y-axis shows the distance at which clusters are merged.
    - Longer vertical lines indicate greater dissimilarity between clusters when merged.
2. Choice of Clusters
    - By setting a horizontal threshold (cutting the dendrogram), we can determine the number of clusters.
    - The optimal number of clusters can be identified by looking for a large vertical jump (gap) in the dendrogram, where merging clusters significantly increases distance.
3. Comparison with K-Means

- Unlike K-Means, hierarchical clustering does not require pre-specifying the number of clusters.
- It provides a hierarchical structure, which can be useful for analyzing relationships among clusters.

4. Ward's Method Advantages

   - Ward's method minimizes variance when merging clusters, leading to compact and well-separated clusters.
   - It generally performs better than single or complete linkage in maintaining cluster quality.

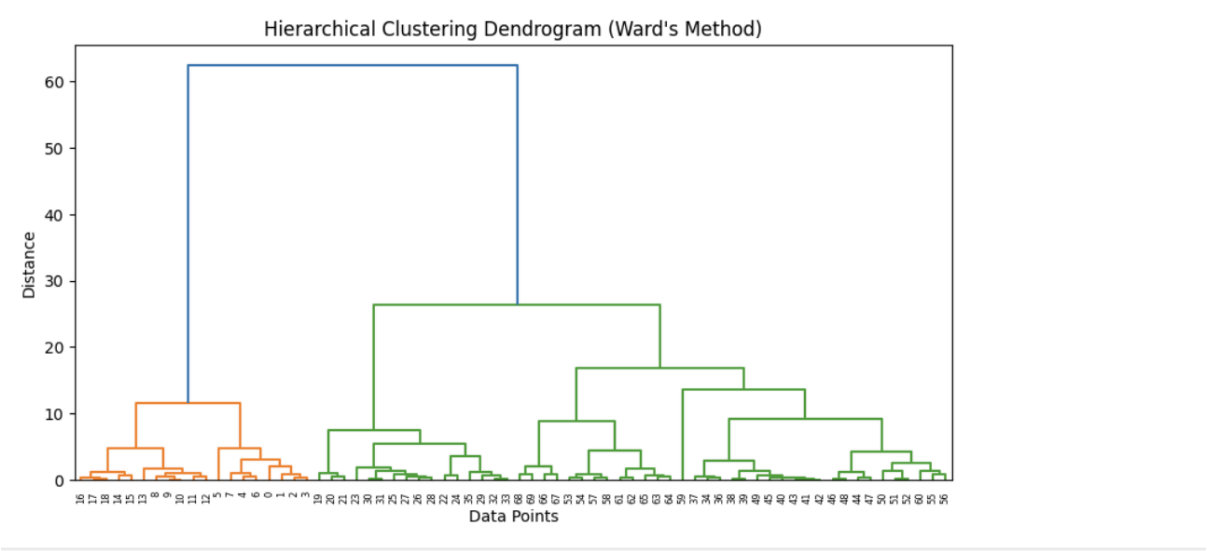
Conclusion

Hierarchical clustering with Ward's method provides an insightful visualization of data structure. The dendrogram helps determine the optimal number of clusters and reveals hierarchical relationships between data points, making it useful for exploratory data analysis.

Code part-

```
# Q6: Hierarchical Clustering & Dendrogram
plt.figure(figsize=(10, 5))
linkage_matrix = sch.linkage(X, method='ward')
sch.dendrogram(linkage_matrix)
plt.title("Hierarchical Clustering Dendrogram (Ward's Method)")
plt.xlabel("Data Points")
plt.ylabel("Distance")
plt.show()
```

Output-



QUESTION 7-

Key Observations and Conclusions

1. **Formation of Separating Hyperplanes**
    - Using **Agglomerative Clustering**, the data is divided into three clusters (A, B, and C).
    - The cluster centers are computed as prototypes for each class.
    - The separating hyperplanes are derived using the difference between each prototype and the overall mean of the cluster centers.
2. **Hyperplane Equations**
    - Each hyperplane equation is of the form: $w1 \cdot x + w2 \cdot y = 0$ w_1 \cdot x + w_2 \cdot y = 0
    - The coefficients **w** are obtained by computing the difference between each cluster center and the mean of all centers.
    - These hyperplanes attempt to divide the feature space into three distinct regions.
3. **Visualization Insights**

- The plotted hyperplanes help in visualizing how different clusters (A, B, and C) are separated.
- Data points are color-coded based on their cluster labels, and the cluster centers are marked with stars.
- The hyperplanes may not always perfectly separate the clusters due to overlaps in data distribution.

4. **Comparison with K-Means Clustering**

   - Unlike K-Means, hierarchical clustering (Agglomerative Clustering) does not assume spherical clusters.
   - This method provides a deterministic clustering result, avoiding initialization randomness.
   - However, K-Means++ may still yield better compact clusters in some cases.

Conclusion

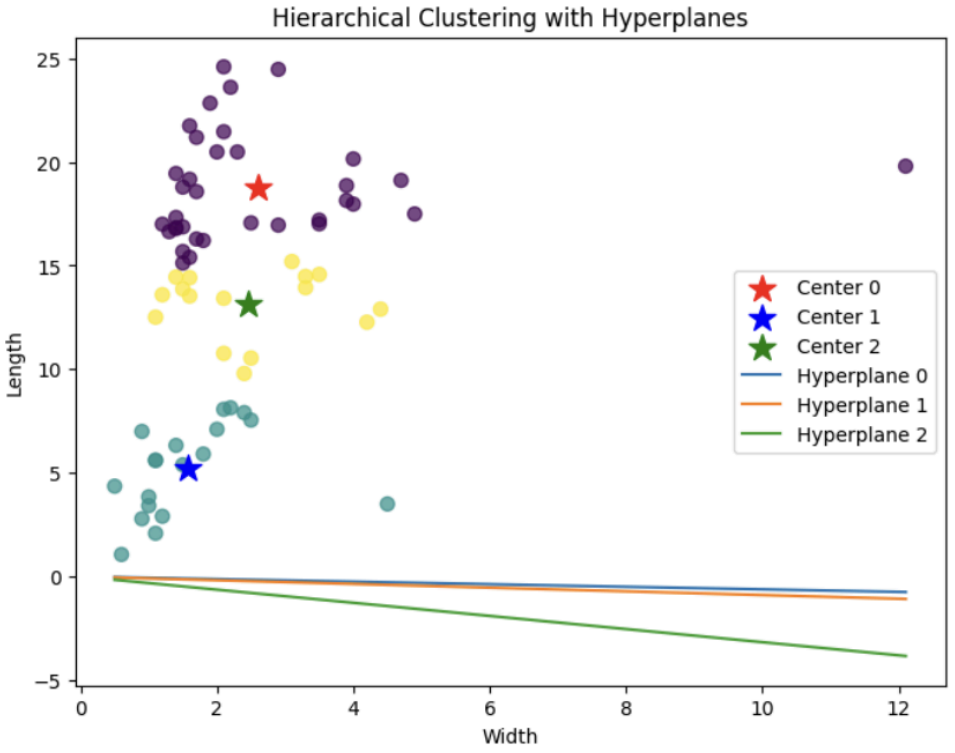
The **Learning with Prototypes** approach helps define separating hyperplanes using cluster centers, allowing a geometric interpretation of class separations. While effective for visualization, it may not always yield the best decision boundaries in cases of complex data distributions or overlapping clusters. A more refined approach (e.g., SVMs or decision trees) could enhance classification accuracy.

Code part-

```python
# Q7: Separating Hyperplanes (Learning with Prototypes) visualization
agg_cluster = AgglomerativeClustering(n_clusters=3).fit(X)
labels_hier = agg_cluster.labels_
centers = [X[labels_hier == i].mean(axis=0) for i in range(3)]

w = np.zeros((3, 2))  # Hyperplane coefficients
for i in range(3):
    for j in range(2):
        w[i, j] = (centers[i][j] - np.mean(centers, axis=0)[j])

print("Hyperplane Equations:")
for i in range(3):
    print(f"Class {i}: {w[i,0]:.2f} * Width + {w[i,1]:.2f} * Length")

# Visualization
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c=labels_hier, cmap='viridis', marker='o', s=50, alpha=0.7)
plt.scatter(centers[0][0], centers[0][1], marker='*', s=200, color='red', label='Center 0')
plt.scatter(centers[1][0], centers[1][1], marker='*', s=200, color='blue', label='Center 1')
plt.scatter(centers[2][0], centers[2][1], marker='*', s=200, color='green', label='Center 2')

# Plot the hyperplanes (lines in 2D)
x_vals = np.linspace(min(X[:,0]), max(X[:,0]), 100) # create evenly spaced points for x axis
for i in range(3):
  if w[i,1] != 0 : # Avoid division by zero if slope is infinite
    y_vals = (-w[i,0] * x_vals )/ w[i,1] # solve y from hyperplane equation
    plt.plot(x_vals,y_vals, label=f'Hyperplane {i}')
  else : # Vertical line case
      plt.axvline(x = centers[i][0] , color='gray', linestyle='--', label=f'Hyperplane {i} (vertical)')

plt.xlabel("Width")
plt.ylabel("Length")
plt.title("Hierarchical Clustering with Hyperplanes")
plt.legend()
plt.show()
```

Output-

```
Hyperplane Equations:
Class 0: 0.40 * Width + 6.40 * Length
Class 1: -0.64 * Width + -7.18 * Length
Class 2: 0.25 * Width + 0.78 * Length
```



QUESTION 8-

Key Observations and Conclusions

1. **Cluster Mean and Covariance Analysis**

    - Each cluster has a distinct **mean vector** μ\mu and **covariance matrix** Σ\Sigma, which define the **Multivariate Gaussian Distribution** for that cluster.
    - The mean represents the center of the cluster, while the covariance matrix defines its spread and orientation.
2. **Shape and Spread of Distributions**

    - The **covariance matrix** influences the shape of the distribution:
        - **Diagonal covariance**: The cluster is spread more along the feature axes.
        - **Off-diagonal elements**: Indicate correlation between features, resulting in tilted ellipsoidal contours.
    - Different clusters may have varying spreads, indicating differences in variance along different feature dimensions.
3. **Contour Visualization**

    - The **contour plots** of the probability density function (PDF) show how the density is distributed across the feature space.
    - Higher density regions (brighter areas) indicate areas where points are more concentrated, while lower density regions (darker areas) show sparser regions.
4. **Comparison with K-Means and Hierarchical Clustering**

- K-Means assumes spherical clusters, whereas **Multivariate Gaussian models allow elliptical clusters**.
- Hierarchical clustering groups points based on distance, while the **Gaussian distribution approach models the probability of a point belonging to a cluster** based on its likelihood under the corresponding distribution.
- This method is useful for **probabilistic classification**, unlike K-Means, which assigns hard labels.

Conclusion

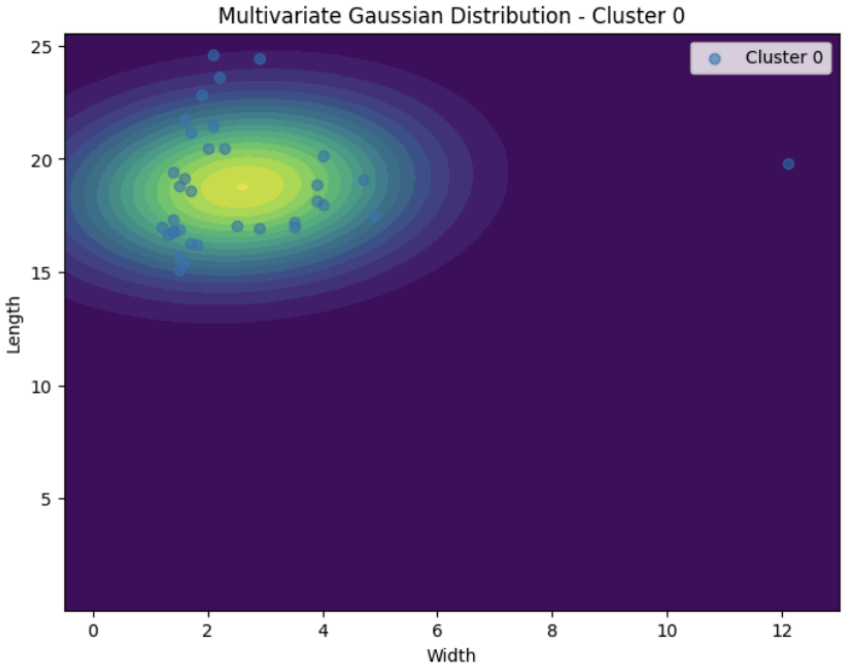
Approximating clusters with **Multivariate Gaussian Distributions** provides a probabilistic perspective on the data. The covariance matrix helps in understanding the correlation between features, and the contour plots provide a visual representation of the density of points in each cluster. This method is useful in **Gaussian Mixture Models (GMMs)**, anomaly detection, and density estimation tasks.

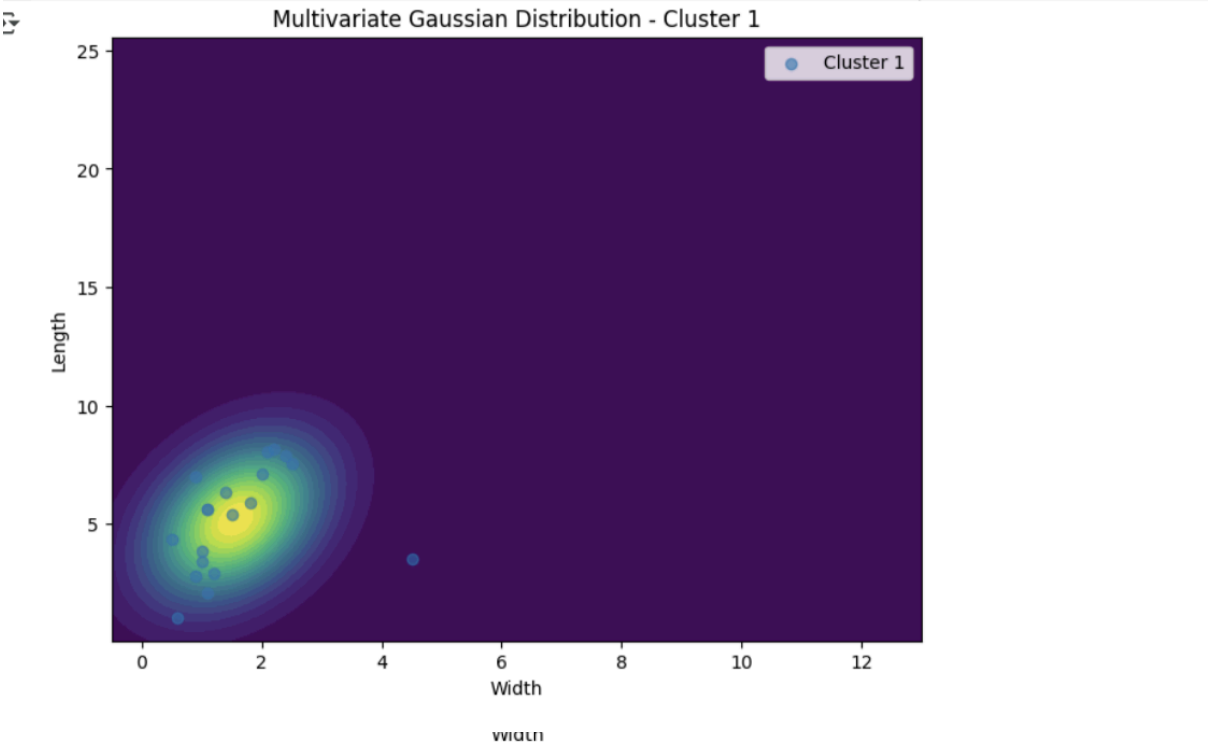
Code part-

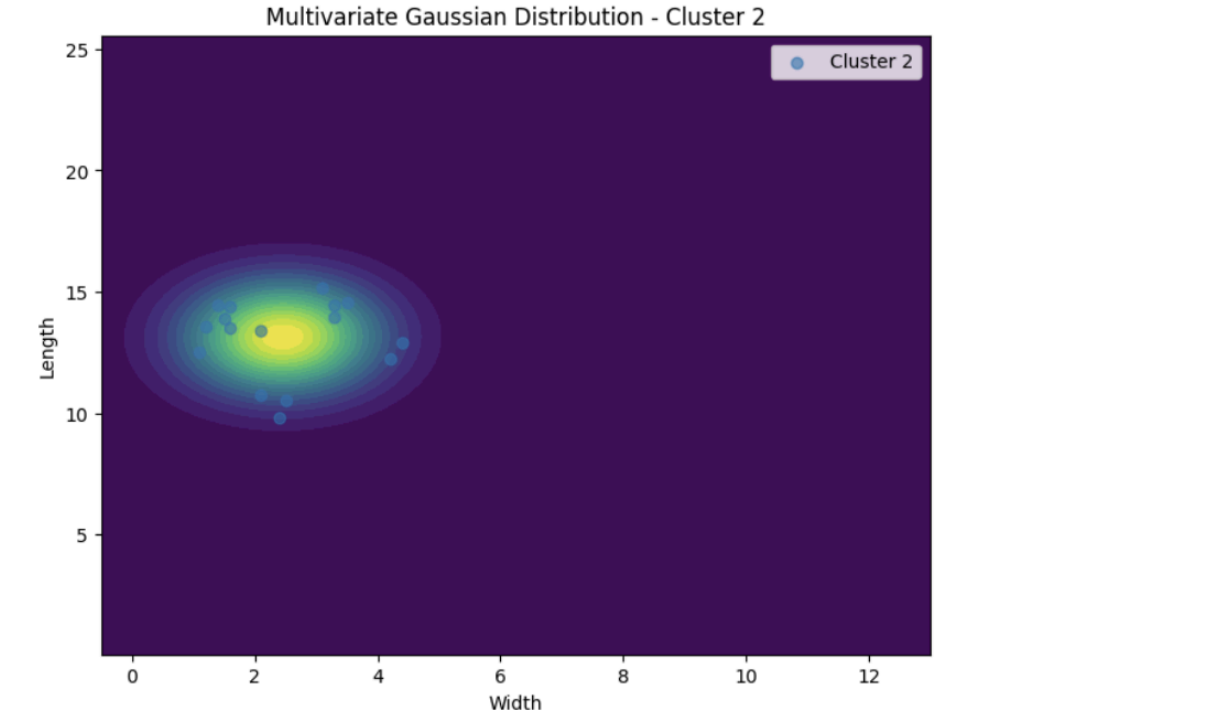
```
[23] # Q8: Multivariate Gaussian Distribution
     for i in range(3):
         subset = X[labels_hier == i]
         mean = np.mean(subset, axis=0)
         cov = np.cov(subset.T)
         print(f"Cluster {i} Mean: {mean}")
         print(f"Cluster {i} Covariance Matrix:\n{cov}\n")


     # Visualize the multivariate Gaussian distributions
     for i in range(3):
         subset = X[labels_hier == i]
         mean = np.mean(subset, axis=0)
         cov = np.cov(subset.T)

         # Create a meshgrid for plotting
         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, 0.1),
                              np.arange(y_min, y_max, 0.1))

         # Create the multivariate Gaussian distribution
         pos = np.dstack((xx, yy))
         rv = multivariate_normal(mean, cov)

         # Plot the contour
         plt.figure(figsize=(8, 6))
         plt.contourf(xx, yy, rv.pdf(pos), levels=20, cmap='viridis')
         plt.scatter(subset[:, 0], subset[:, 1], label=f'Cluster {i}', alpha=0.6)
         plt.title(f'Multivariate Gaussian Distribution - Cluster {i}')
         plt.xlabel('Width')
         plt.ylabel('Length')
         plt.legend()
         plt.show()
```

Output-

```
Cluster 0 Mean: [ 2.60571429 18.76396765]
Cluster 0 Covariance Matrix:
[[3.85231933 0.53199907]
 [0.53199907 6.47832292]]

Cluster 1 Mean: [1.56842105 5.18793882]
Cluster 1 Covariance Matrix:
[[0.8622807  0.7022657 ]
 [0.7022657  4.73574016]]

Cluster 2 Mean: [ 2.45625    13.14101907]
Cluster 2 Covariance Matrix:
[[1.12795833 0.01518494]
 [0.01518494 2.52167656]]
```







QUESTION 9-

Key Observations and Conclusions

1. Maximum Likelihood Estimation (MLE) Method

- The new point (1.9,6)(1.9,6) is classified based on the probability density function (PDF) of the **Multivariate Gaussian Distributions** obtained in Q8.
- The class with the **highest likelihood** is selected.
- This method works well when the data is normally distributed within clusters.

2. Hyperplane Method (Learning with Prototypes - Q7)

- The new point is classified based on its **distance to the hyperplanes** that separate the three clusters.
- The class with the largest distance (i.e., the side of the hyperplane it falls on) determines the classification.
- This method assumes linear decision boundaries, which may not always be optimal for complex distributions.

3. K-Nearest Neighbors (K-NN) Method

- The **5 nearest neighbors** of the new point are found, and the most common class among them is assigned.
- This method is **non-parametric** and works well when the dataset is **locally structured** but can be sensitive to the choice of kk.
- K-NN performs better when data is not linearly separable.

Comparison of Methods

Method	Strengths	Weaknesses
MLE (Multivariate Gaussian)	Works well for normally distributed data, probabilistic classification	Assumes Gaussian clusters, sensitive to covariance estimation

Hyperplane (Prototypes)	Computationally efficient, provides linear separation	Assumes linear decision boundaries, may not generalize well
K-NN (Nearest Neighbors)	No assumptions about distribution, works for non-linear separation	Sensitive to noisy data and choice of kkk, computationally expensive for large datasets

---

Conclusion

- If the data follows a Gaussian distribution, **MLE** is a strong choice.
- If linear decision boundaries are appropriate, **Hyperplane (Learning with Prototypes)** works well.
- If the data is non-linearly separable or irregularly shaped, **K-NN** may be the best choice.
- Comparing the classifications from all three methods helps in selecting the most **robust** and **accurate** model.

Code part-

```python
# Q9: Classification of (1.9, 6)
new_point = np.array([[1.9, 6]])

# MLE Method
probs = [multivariate_normal.pdf(new_point, mean=np.mean(X[labels_hier == i], axis=0), cov=np.cov(X[labels_hier == i].T)) for i in range(3)]
pred_mle = np.argmax(probs)

# Hyperplane Method
distances_to_hyperplane = [np.dot(new_point, w[i]) for i in range(3)]
pred_hyperplane = np.argmax(distances_to_hyperplane)

# K-NN Method
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X, labels_hier)
pred_knn = knn.predict(new_point)[0]

print(f"MLE Prediction: Class {pred_mle}")
print(f"Hyperplane Prediction: Class {pred_hyperplane}")
print(f"K-NN Prediction: Class {pred_knn}")

# Visualize the new point and the cluster boundaries
plt.figure(figsize=(8, 6))

# Plot the data points colored by their cluster assignments
for i in range(3):
    plt.scatter(X[labels_hier == i, 0], X[labels_hier == i, 1], label=f'Cluster {i}')

# Plot the new point
plt.scatter(new_point[0, 0], new_point[0, 1], marker='x', color='red', s=100, label='New Point')

# Plot the cluster centers
for i in range(3):
    plt.scatter(centers[i][0], centers[i][1], marker='*', color='black', s=150, label=f'Center {i}' if i == 0 else "")


plt.xlabel("Width")
plt.ylabel("Length")
plt.title("Classification of New Point (1.9, 6)")
plt.legend()
plt.show()
```

```
MLE Prediction: Class 1
Hyperplane Prediction: Class 0
K-NN Prediction: Class 1
```