This notebook aims to write the code necessary for all evaluation and analysis metrics. The metrics considred will be:

- Silhouette Score
- Davies-Bouldin Index
- Calinski-Harabasz Index
- Adjusted Rand Index
- Normalized Mutual Information
- Purity

```
import numpy as np
# Silhouette Score
# Measuring the accuracy of the cluster. Measure of how go
# Items in a cluster must be coherent (share similar chara
# To measure the silhouette coefficient, you must compute
# of all silhouette scores for each data point to find the
# Aim to find how different clusters are to one another.

# Cohesion --> how close the current data point is to its
# Separation --> how close the current data point is to po

# silhouette coefficient --> (separation - cohesion)/max(s
# The closer to one, the better the cluster generated (per

def silhouette_score(X, labels):
  clusters = np.unique(labels)
  samples = X.shape[0]

  # Compute distances matrix
  distances = np.zeros((samples, samples))
  for i in range(samples):
    for j in range(samples):
      distances[i, j] = np.linalg.norm(X[i] - X[j])

  # Compute scores
  scores = np.zeros(samples)
  for i in range(samples):
    cluster = labels[i]
    # Cohesion
    same_samples = np.where(labels == cluster)[0]
    same_samples = same_samples[same_samples != i]
```

```
        if len(same_samples) == 0:
          cohesion = 0
        else:
          cohesion = np.mean(distances[i, same_samples])

        # Separation
        separation = np.inf
        for j in clusters:
          if j != cluster:
            separation_j = np.mean(distances[i, labels == j])
            if separation_j < separation:
              separation = separation_j

        scores[i] = (separation - cohesion) / max(separation,

      return np.mean(scores)
```

```
# Davies-Bouldin Index
# Evaluates the quality of clustering reults.
# Provides a numerical score based on two main factors: compactness
# Compactness measures how close data points are within a single cl
# Separation assesses the distance between different clusters

# Begins by finding the centroid of each cluster. The intra-cluster
# from each point to its centroid
# Inter-cluster distance is measured between the centroids of pairs
# A lower index indicates better clustering with tight and well sep
# A higher index suggests potential overlaps/ineffective clustering

def davies_bouldin_index(X, labels):
    clusters = np.unique(labels)
    samples = X.shape[0]

    centroids = []
    dispersion = []

    # Intra-Cluster
    for cluster in clusters:
        cluster_points = X[labels == cluster]
        centroid = np.mean(cluster_points, axis=0)
        centroids.append(centroid)
        dispersion.append(np.mean(np.linalg.norm(cluster_points - c

    dispersion = np.array(dispersion)
    centroids = np.array(centroids)

    # Inter-Cluster
    distances = np.linalg.norm(centroids[:, np.newaxis] - centroids

    max_ratios =[]
    for i in range(len(clusters)):
      current_index_ratio = []
        for j in range(len(clusters)):
          if i != j:
            ratio = (dispersion[i]+dispersion[j]) / distances[i, j]
            current_index_ratio.append(ratio)
        max_ratios.append(np.max(current_index_ratio))

    return np.mean(max_ratios)
```

```
# Calinski-Harabasz Index (CH Index)
# Variance ratio criterion. Metric used to evaluate how well define
# The CH index looks at the data through the concept of variance. M
# within clusters.

# Its identifying parameters are the Between-Cluster Variance and t
# Between-Cluster Variance:
# How spread out the cluster centroids are from the global mean (of
# Within-Cluster Variance:
# Measures how spread out the points are within each cluster relati

# CH = ((Between-Cluster Variance)/ (Within-Cluster Variance))*((N-

# Can be any positive number. A higher index, is a better index (mo
# Function returns index and within_cluster_sum_of_squares
def calinski_harabasz_index(X, labels):
    clusters = np.unique(labels)
    samples = X.shape[0]

    mean_global = np.mean(X, axis=0)

    # Between-Cluster Variance
    between_cluster_sum_of_squares = 0
    within_cluster_sum_of_squares = 0

    for cluster in clusters:
        cluster_points = X[labels == cluster]
        mean_cluster = np.mean(cluster_points, axis=0)

        nSamples = cluster_points.shape[0]
        within_cluster_sum_of_squares += np.sum((cluster_points - m
        between_cluster_sum_of_squares += nSamples * np.sum((cluste

    index = ((between_cluster_sum_of_squares / (len(clusters) - 1))

    return within_cluster_sum_of_squares, index
```

```
# Adjusted Rand Index (ARI)
# Measures the similarity between two sets of clustering results.
# Measures the similarity between assignments. For example, if the
# this would result in an ARI of 1, a perfect assignment score. To
# TP: Points that are in the same cluster in the ground truth and t
# TN: Points that are in different clusters in both assignments
# FP: Points are in the same cluster in prediction but different in
# FN: Points are in different clusters in predictions but the same
```

```
# Rand Index = (TP+TN)/(TP+TN+FP+FN)

def adjusted_rand_index(labels_true, labels_pred):
  TP = 0
  TN = 0
  FP = 0
  FN = 0

  for i in range(0, len(labels_pred)-1):
    for j in range(i+1, len(labels_pred)):
      if labels_pred[i] == labels_pred[j]:
        # Positive Pair
        if labels_true[i] == labels_true[j]:
          # True
          TP += 1
        else: #False
          FP += 1
      else: # Negative Pair
        if labels_true[i] == labels_true[j]:
          # False
          FN += 1
        else: # True
          TN += 1

  total_pairs = TP+TN+FP+FN

  expected_index = ((TP+FP)*(TP+FN))/total_pairs
  max_index = ((TP+FP)+(TP+FN))/2

  # Prevent division by zero

  if (max_index-expected_index) == 0:
    return 0.0

  ari = (TP-expected_index)/(max_index-expected_index)

  return ari
```

```
# Normalized Mutual Information (NMI)
# Provides a quantitative assessment of a clustering algor
# distinct groups within data
# Given the knowledge of the ground truths class assignmen
# data output predicted labels, the NMI measures the agree
# Based on the concept of information theory introduced in
# is shared between the ground truth labels and the predic
# Looks at the reduction of uncertainity in one variable g
```

```
# Ranges from [0,1], with one being a perfect correlation

def normalized_mutual_information(labels_true, labels_pred
  clusters = np.unique(labels_pred)
  classes = np.unique(labels_true)

  matrix = np.zeros((len(classes), len(clusters)))

  for i, c in enumerate(classes):
    for j, k in enumerate(clusters):
      matrix[i, j] = np.sum((labels_true == c) & (labels_p

  # Entropy calculation
  pi_true = np.sum(matrix,axis=1)/len(labels_true)
  pi_pred = np.sum(matrix,axis=0)/len(labels_pred)

  pi_true = pi_true[pi_true > 0]
  pi_pred = pi_pred[pi_pred > 0]

  h_true = -np.sum(pi_true * np.log2(pi_true))
  h_pred = -np.sum(pi_pred * np.log2(pi_pred))

  mutual_information = 0

  for i in range(len(classes)):
    for j in range(len(clusters)):
      if matrix[i, j] > 0:
        # Probability of point being in class i and cluste
        p_ij = matrix[i, j] / len(labels_true)
        # Probability of point being in class i
        p_i  = np.sum(matrix[i, :]) / len(labels_true)
        # Probability of point being in cluster j
        p_j  = np.sum(matrix[:, j]) / len(labels_true)
        mutual_information += p_ij * np.log2(p_ij / (p_i *

  if (h_true + h_pred) == 0:
    return 0.0

  return 2* mutual_information / (h_true + h_pred)
```

```
# Purity
# Measures the extent to which a cluster has a single clas
# To measure it, we identify the most dominant class in a
# Ranges from [0,1] where 1 means perfect clustering and 0
# It is not enough on its own as a model may tend to overf

def purity(labels_true, labels_pred):
  clusters = np.unique(labels_pred)
  classes = np.unique(labels_true)

  matrix = np.zeros(len(classes), len(clusters))

  for i, c in enumerate(classes):
    for j, k in enumerate(clusters):
      matrix[i, j] = np.sum((labels_true == c) & (labels_p

  dominant_class = np.max(matrix, axis=0)
  return np.sum(dominant_class) / len(labels_true)
```