Open Access

Exploiting clustering algorithms for spatial analysis of geophysical data

Dave O'Leary[1]

[1]Teagasc

Abstract

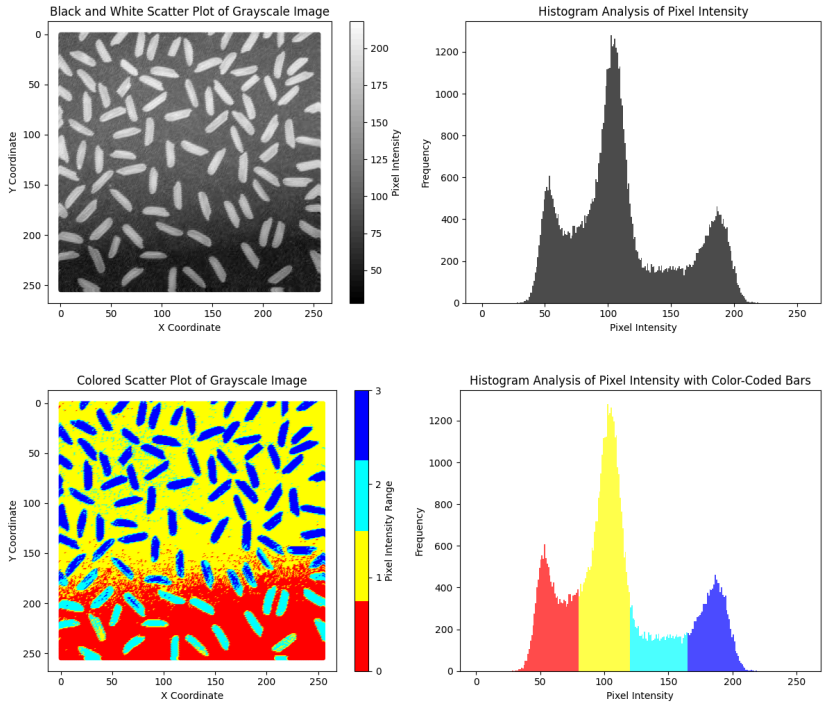
In this workshop, you will learn the tips and tricks on how to use centroid based clustering algorithms to analyse your spatial data. Using examples you will see how clustering can help to understand spatial patterns in several example datasets, including synthetic and real geophysical data. You will learn how clustering can be used to give insight into temporal changes in the landscape and the final example will show how clustering can combine data from different remotely sensed platforms. With the help of an online Google Colab notebook, you will also have the opportunity to try clustering on your data. Bring along some example dataset to try!

Keywords K-Means, Unsupervised Machine Learning, Spatial Analysis, MCASD

Prerequisites

Access to a Google Account and email (all code is run in Google Colab, but no coding experience necessary). If you want to try clustering on your own dataset, please bring it along. Prepare it in .CSV format with columns organised as `X_Coordinate, Y_Coordinate, Data1, Data 2, …, Data N`.

Below are examples of clustering in action! A grey scale image is grouped into 4 classes.



Workshop

Published Feb 02, 2024

Correspondence to
Dave O'Leary
Daveolearyphd@gmail.com

1. Introduction to Google Colab and Clustering

In this part of the workshop you will be generating some randomised 3D data, which have distinct locations in the dataspace. These will aid in the conceptulisation of clustering which will help with understanding the rest of the workshop.

This Part will also act as a basic introduction to Google Colab and work out any problems before we move on to more complex code and worksheets.

The aim of this workshop is to become familiar with Google Colab and running code in this environment and to cluster these randomised data and using the provided code.

Please following along with the workshop leader in the first instance until you are familiar with using Google Colab environment.

No coding experience is required to run this code. All the code contains comments describing what each line does. Please click “Show Code” on any section to view the code and click “View” and “Show/Hide code” to hide the code again.

Please feel free to ask questions if you don’t understand any parts.

1.1. Section 0

This section sets up the Python environment for the clustering analysis.

It imports essential libraries such as Pandas for data manipulation, NumPy for numerical operations, Matplotlib for data visualization, scikit-learn for machine learning tools, and other supporting libraries.

Additionally, it configures the display.

The code also imports specific functions and modules required for the clustering analysis, such as KMeans.

Finally, it sets up tools for working with images, zip files, and file uploads in Google Colab. This preparation ensures that the subsequent code can efficiently perform clustering analysis and handle related tasks.

```
# @title
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
import imageio.v2 as imageio
import os
from zipfile import ZipFile

from sklearn.cluster import KMeans
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_score, pairwise_distances_argmin_min
from collections import Counter
from matplotlib.ticker import MaxNLocator
from matplotlib.colors import ListedColormap
from PIL import Image
from google.colab import files

# Set display options to show all rows and columns
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
```

Great!

1.2. Section 1

Now we are going to generate the data!

These are randomly generated 3D data point, with an x, y, and z value being assigned to each. They are generated in 3 “groups” in order to simulate a dataset that has clear groups. Once the data are create their order is randomised to highlight the power of clustering.

This section will help with visualising and conceptulising the centroid clustering technique we are using today.

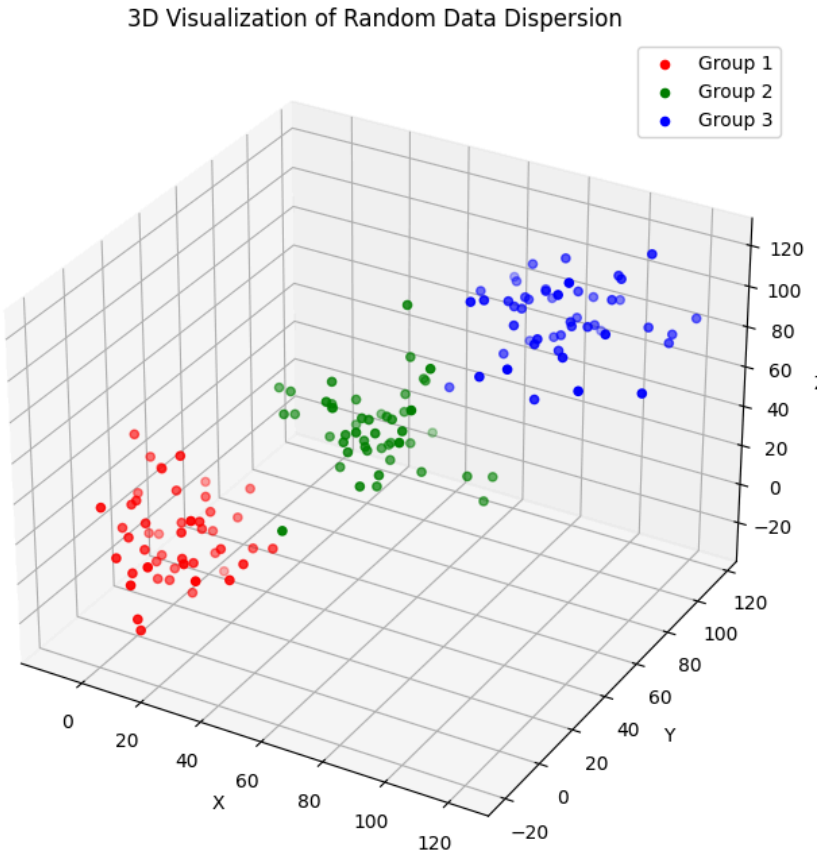
Press the play button below to generate and view the data. You will be asked to enter the number of datapoint per group and a value for how disperse they are.

Run this code a few times with different values to see the outputs change.

```
# @title
# Set random seed for reproducibility
np.random.seed(42)

# Number of data points in each group
num_points = int(input("Enter the number of points within each group (i.e., 100): "))

# Set the standard deviation for each group (controls dispersion)
std_deviation = int(input("Enter a value for how disperse the data should be (i.e., low = 1,
high = 20): "))

# Generate random data for three groups with increased dispersion
group1_center = np.array([10, 10, 10])
group2_center = np.array([50, 50, 50])
group3_center = np.array([90, 90, 90])

group1_data = np.round(group1_center + std_deviation * np.random.normal(size=(num_points,
len(group1_center))),2)
group2_data = np.round(group2_center + std_deviation * np.random.normal(size=(num_points,
len(group2_center))),2)
group3_data = np.round(group3_center + std_deviation * np.random.normal(size=(num_points,
len(group3_center))),2)

# Merge the data from all three groups
all_data = np.vstack([group1_data, group2_data, group3_data])

# Shuffle the merged data
np.random.shuffle(all_data)

# Create a DataFrame
columns = ['X', 'Y', 'Z']
remaining_data = pd.DataFrame(all_data, columns=columns)

# Display the first few rows of the DataFrame
print(remaining_data.head(10))

# Visualize the data in 3D
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection='3d')
```

```
# Scatter plot for each group
ax.scatter(group1_data[:, 0], group1_data[:, 1], group1_data[:, 2], label='Group 1',
c='red', marker='o')
ax.scatter(group2_data[:, 0], group2_data[:, 1], group2_data[:, 2], label='Group 2',
c='green', marker='o')
ax.scatter(group3_data[:, 0], group3_data[:, 1], group3_data[:, 2], label='Group 3',
c='blue', marker='o')

# Set plot properties
ax.set_title('3D Visualization of Random Data Dispersion')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.legend()

# Show the plot
plt.show()
```

```
Enter the number of points within each group (i.e., 100): 50
Enter a value for how disperse the data should be (i.e., low = 1, high = 20): 15
       X      Y       Z
0  60.29  25.81   42.92
1  81.54  77.67   93.66
2  75.90  82.29   74.11
3  86.97  86.73  106.48
4  11.38 -19.81    6.70
5  67.38  37.69   64.45
6  57.24  46.65   60.71
7  78.40  90.37   97.47
8  -2.13   2.47   23.73
9  93.48  68.28   68.89
```



1.3. Section 2

This section performs data normalization, a crucial step before applying clustering algorithms.

The resulting normalized data is displayed, providing an insight into the standardized values across the dataset.

Normalization enhances the accuracy of clustering algorithms, ensuring that features with different scales contribute equally to the clustering process.

```
# @title
#### Section 2 Normalization ####

# Custom normalization using MinMaxScaler
min_vals = remaining_data.min()
max_vals = remaining_data.max()

normalized_data = (remaining_data - min_vals) / (max_vals - min_vals)

# Display the normalized data
print("\nNormalized Data:")
print(normalized_data.head(10).to_string(index=False))  # Use to_string to prevent
truncation
```

```
Normalized Data:
          X         Y         Z
0.532716 0.335096 0.476385
0.691440 0.716028 0.811082
0.649313 0.749963 0.682124
0.731999 0.782577 0.895646
0.167389 0.000000 0.237467
0.585674 0.422359 0.618404
0.509934 0.488174 0.593734
0.667986 0.809314 0.836214
0.066477 0.163655 0.349802
0.780624 0.647055 0.647691
```

1.4. Section 3

This section guides the user in determining the optimal number of clusters (k) for the K-Means algorithm by utilizing both the Elbow Method and Silhouette Scores.

After specifying the maximum number of clusters to consider, the code calculates the Within-Cluster Sum of Squares (WCSS) distance using the Elbow Method. The Elbow Method graph illustrates the trade-off between clustering complexity and WCSS reduction, helping identify an optimal k value.

Simultaneously, Silhouette Scores, a measure of how well-separated clusters are, are computed and presented on the same graph. Silhouette Scores range from −1 to 1, where higher scores indicate better-defined clusters.

When assessing the graph, users should look for the “elbow” point where WCSS plateaus, suggesting diminishing returns with additional clusters.

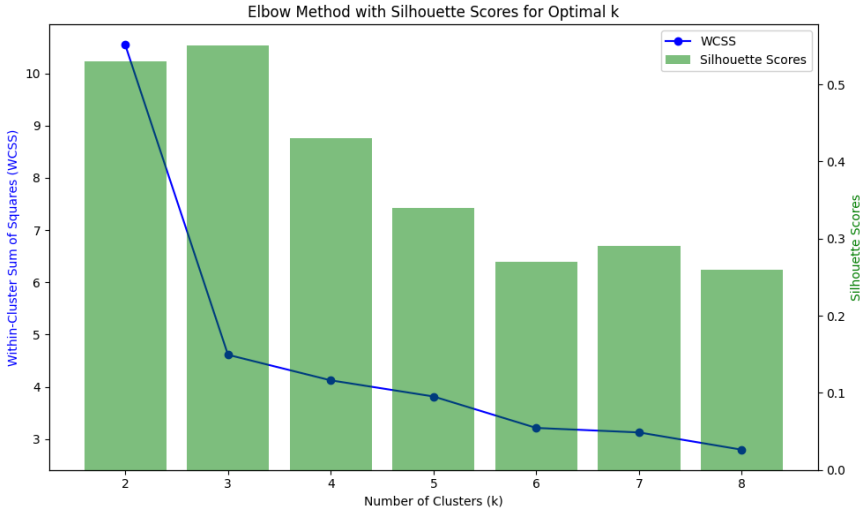
Additionally, a high Silhouette Score at the “elbow” reinforces the choice, ensuring a balance between compact clusters and distinct cluster boundaries for effective clustering.

When running the cell, you will be prompted to enter the max number of clusters. (i.e., 10).

```
# @title
#### Section 3 Elbow Method with Silhouette Scores ####

# Prompt the user to choose the maximum number of clusters for the Elbow Method
max_clusters_elbow = int(input("Enter the maximum number of clusters for the Elbow Method:
"))

# Calculate the within-cluster sum of squares (WCSS) and Silhouette Scores for different
values of k
wcss = []
silhouette = []
for k in range(2, max_clusters_elbow + 1):
    kmeans = KMeans(n_clusters=k, n_init=1, init='k-means++')
    kmeans.fit(normalized_data)
    sscore = round(silhouette_score(normalized_data, kmeans.labels_), 2)
    silhouette.append(sscore)
    wcss.append(kmeans.inertia_)

# Plot the Elbow Method graph with Silhouette Scores as a bar graph
fig, ax1 = plt.subplots(figsize=(10, 6))

# Plot WCSS on the left y-axis
ax1.plot(range(2, max_clusters_elbow + 1), wcss, marker='o', color='blue', label='WCSS')
```

```
ax1.set_xlabel('Number of Clusters (k)')
ax1.set_ylabel('Within-Cluster Sum of Squares (WCSS)', color='blue')

# Set the x-axis ticks to show only integers
plt.xticks(range(2, max_clusters_elbow + 1))

# Create a second y-axis for Silhouette Scores
ax2 = ax1.twinx()
ax2.bar(range(2, max_clusters_elbow + 1), silhouette, color='green', alpha=0.5,
label='Silhouette Scores')
ax2.set_ylabel('Silhouette Scores', color='green')

# Add legend
lines, labels = ax1.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
ax2.legend(lines + lines2, labels + labels2, loc='upper right')

plt.title('Elbow Method with Silhouette Scores for Optimal k')
fig.tight_layout()
plt.show()
```

```
Enter the maximum number of clusters for the Elbow Method: 8
```



1.5. Section 4

In this section, users will perform K-Means clustering on the data. The optimal number of clusters (k) can be determined by referencing the results from the previous Elbow Method and Silhouette Scores analysis (Section 3). After entering the desired number of clusters, the code applies K-Means clustering and displays key information.

The results include a count of occurrences for each cluster label as a QC.

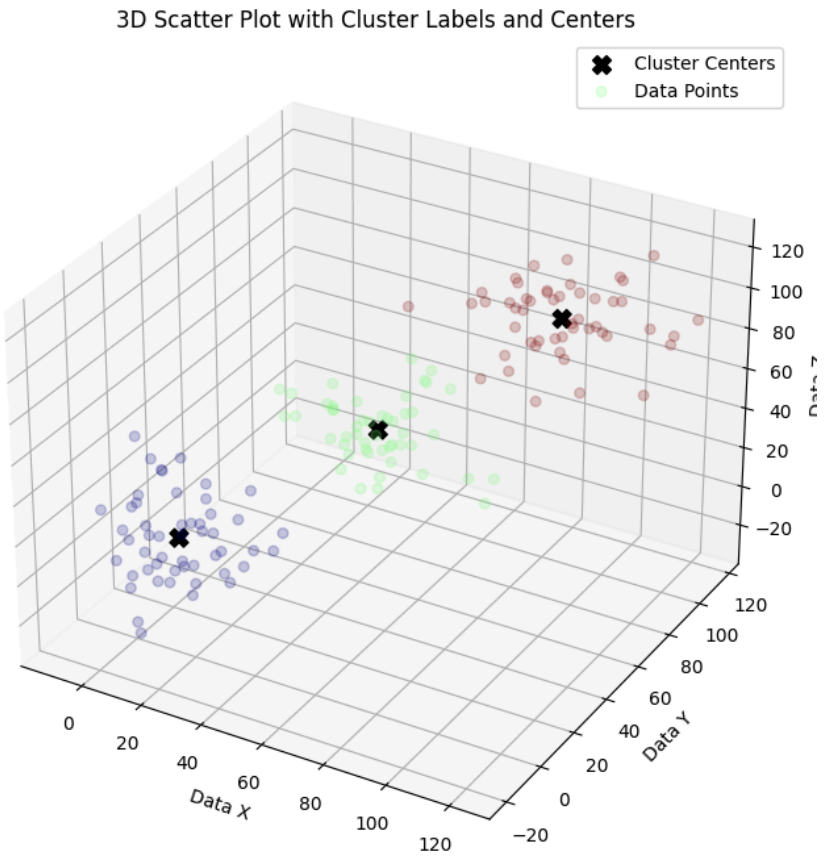
To visually assess the clustering, a 3D scatter plot with labeled data and unnormalized cluster centers are generated.

When running the cell, you will be prompted to enter the number of clusters, which should be based on the Elbow and Silhouette results, but feel free to run this code a few times for various number of clusters and take a look at the results!

```
# @title
##### Section 4 K Means Clustering ####

### Section 4.1 Perform K Means Clustering ###

# Prompt the user to choose the number of clusters for K-Means
num_clusters = int(input("Enter the number of clusters for K-Means: "))

# Initialize the K-Means model
kmeans = KMeans(n_clusters=num_clusters, init='k-means++', n_init=1)

# Fit the K-Means model to the normalized data
kmeans.fit(normalized_data)

# Get the cluster labels for each data point
cluster_labels = kmeans.labels_

# Get the cluster centers
cluster_centers = kmeans.cluster_centers_

### Section 4.2 Sort the Cluster centers ###

# Calculate the distances of cluster centers from the origin (0, 0)
distances_from_origin = np.sqrt(np.sum(cluster_centers ** 2, axis=1))

# Sort cluster centers based on distances from the origin
sorted_indices = np.argsort(distances_from_origin)

# Sort cluster centers and labels
sorted_cluster_centers = cluster_centers[sorted_indices]
sorted_cluster_labels = np.zeros_like(cluster_labels)

# Relabel the cluster labels based on the sorted order
for new_label, old_label in enumerate(sorted_indices):
    sorted_cluster_labels[cluster_labels == old_label] = new_label

# Calculate the count of each cluster label
cluster_labels_count = dict(zip(*np.unique(sorted_cluster_labels, return_counts=True)))

### Section 4.3 Denormalize the cluster centers ###

# Denormalize the data using the inverse transformation
cluster_centers_original_scale = sorted_cluster_centers * (max_vals.values -
min_vals.values) + min_vals.values

### Section 4.4 Display Clustering counts for visual QC ###

# Display the count of each cluster label
print("\nCount of Each Cluster Label:")
for label, count in cluster_labels_count.items():
    print(f"Cluster {label}: {count} occurrences")
```

```
# Create a DataFrame with Cluster and Remaining Data
clustered_data_df = pd.DataFrame({
    'Cluster Number': sorted_cluster_labels,
    **{f'{col}': remaining_data[col] for col in remaining_data.columns}
})
clustered_data_df = clustered_data_df.round(4)

# Create a DataFrame with Cluster center data
center_data_df = pd.DataFrame(cluster_centers_original_scale,
columns=remaining_data.columns)

# Add a new column 'Cluster Number' to indicate the cluster number for each row
center_data_df.insert(0, 'Cluster Number', range(num_clusters))
center_data_df = center_data_df.round(4)

### Section 4.5 Plot KMeans Clustering results ###
fig = plt.figure(figsize=(18, 18))

# Plot 1: 3D Scatter Plot with Cluster Labels and Centers
ax1 = fig.add_subplot(121, projection='3d')
centers_scatter = ax1.scatter(center_data_df['X'], center_data_df['Y'], center_data_df['Z'],
c='black', marker='X',
                              s=100, alpha=1.0, label='Cluster Centers')
scatter = ax1.scatter(clustered_data_df['X'],clustered_data_df['Y'], clustered_data_df['Z'],
c=sorted_cluster_labels, cmap='jet',
                      marker='o', s=30, alpha=0.2, label='Data Points')

ax1.set_title('3D Scatter Plot with Cluster Labels and Centers')
ax1.set_xlabel('Data X')
ax1.set_ylabel('Data Y')
ax1.set_zlabel('Data Z')
ax1.legend()

# Show the plots
plt.show()
```

```
Enter the number of clusters for K-Means: 3

Count of Each Cluster Label:
Cluster 0: 51 occurrences
Cluster 1: 49 occurrences
Cluster 2: 50 occurrences
```



1.6. Section 5

This section introduces the MCASD (Multiple Cluster Average Standard Deviation) method, designed to aid you in identifying the optimal number of clusters for their dataset.

MCASD was first published as a method by O'Leary *et al* 2023.

MCASD evaluates the stability of cluster centers across multiple attempts and cluster numbers. Participants will input the maximum number of clusters and the maximum number of attempts per cluster.

The code loops through different cluster numbers, applying K-Means clustering multiple times to analyze stability.

The results include GIFs illustrating scatter plots and line plots for each attempt. Additionally, MCASD metrics are calculated, providing insights into the stability and consistency of the clustering results for various numbers of clusters. A line plot visualizes the MCASD metric across various cluster numbers, aiding participants in selecting the optimal cluster count.

All results, including csv files, plots and gifs, are compressed into a zip file for easy download.

Please save this ZIP file to "Part 1" of the data directory you were provided with and unzip it to view the outputs from MCASD analysis.

The ultimate goal is to assist participants in making informed decisions about the optimal number of clusters for their specific dataset.

Press the play button to run the code. You will be prompted to enter the number of max number of clusters (i.e., 10) and max attempts (i.e., 10).

Note this might take some time depending on the max number of clusters and max attempts chosen.

```
# @title
#### Section 8 MCASD Method ####
### Section 8.1 Get information from the user ###

# Prompt the user for the maximum number of clusters for MCASD Method
max_num_clusters = int(input("Enter the maximum number of clusters for MCASD Method: "))
# Prompt the user for the maximum number of attempts for MCASD Method
max_attempts = int(input("Enter the maximum number of attempts for MCASD Method: "))

### Section 8.2 Loop for MCASD Method ###
# Create a DataFrame to store MCASD Metrics
mcasd_metrics_df = pd.DataFrame(index=['MCASD Metric', 'MCASD Error'], columns=range(1,
max_num_clusters + 1))

print(f"\nCalculating MCASD Metrics...")

# Create a zip file to store all results
zip_filename = f'Agrogeo24_WS_Part_1_kmeans_plots.zip'
with ZipFile(zip_filename, 'w') as zip_file:

    # Loop through the various number of clusters
    for num_clusters in range(1, max_num_clusters + 1):
        images_attempt = []  # List to store images for the current attempt
        distances_df = pd.DataFrame()  # Initialize distances DataFrame

        # Cluster the data a user specified number of times (Attempts)
        for attempt in range(1, max_attempts + 1):
            #print(f"\nNumber of Clusters: {num_clusters}: Attempt {attempt} of
{max_attempts}")

            # Initialize the K-Means model
            kmeans = KMeans(n_clusters=num_clusters, n_init=1, init='k-means++')

            # Fit the K-Means model to the normalized data
            kmeans.fit(normalized_data)

            # Get the cluster labels for each data point
            cluster_labels = kmeans.labels_

            # Get the cluster centers
            cluster_centers = kmeans.cluster_centers_

            ### Section 8.2.1 Sort the Cluster centers ###

            # Calculate the distances of cluster centers from the origin (0, 0)
            distances_from_origin = np.sqrt(np.sum(cluster_centers ** 2, axis=1))

            # Sort cluster centers based on distances from the origin
            sorted_indices = np.argsort(distances_from_origin)

            # Sort cluster centers and labels
            sorted_cluster_centers = cluster_centers[sorted_indices]
            sorted_cluster_labels = np.zeros_like(cluster_labels)
```

```
            # Relabel the cluster labels based on the sorted order
            for i, new_label in enumerate(np.arange(num_clusters)):
                old_label = sorted_indices[i]
                sorted_cluster_labels[cluster_labels == old_label] = new_label

            ### Section 8.2.2 Calculate the distance (in the dataspace) between each
datapoint and its closest cluster center ###

            # Calculate the distances between cluster centers and data
            distances = np.linalg.norm(normalized_data.values[:, np.newaxis, :] -
cluster_centers, axis=-1)

            # Get the smallest distance for each data point
            min_distances = np.min(distances, axis=1)

            # Create a DataFrame for distances with only the smallest distances
            new_column = pd.DataFrame(min_distances, columns=[f'Attempt_{attempt}'])

            # Append the new column to the existing distances_df
            distances_df = pd.concat([distances_df, new_column], axis=1)

            # Calculate the count of each cluster label
            cluster_labels_count = dict(zip(*np.unique(sorted_cluster_labels,
return_counts=True)))

            # Create a DataFrame with Cluster and Remaining Data
            clustered_data_df = pd.DataFrame({
                'Cluster Number': sorted_cluster_labels,
                **{f'{col}': remaining_data[col] for col in remaining_data.columns}
            })
            clustered_data_df = clustered_data_df.round(4)

            ### Section 8.2.3 Denormalize the cluster centers ###

            cluster_centers_original_scale = sorted_cluster_centers * (max_vals.values -
min_vals.values) + min_vals.values

            # Create a DataFrame with Cluster center data
            center_data_df = pd.DataFrame(cluster_centers_original_scale,
columns=remaining_data.columns)

            # Add a new column 'Cluster Number' to indicate the cluster number for each row
            center_data_df.insert(0, 'Cluster Number', range(num_clusters))
            center_data_df = center_data_df.round(4)

            ### Section 8.2.4 Create and save plots for later GIF creation ###

            fig = plt.figure(figsize=(18, 18))

            # Plot 1: 3D Scatter Plot with Cluster Labels and Centers
            ax1 = fig.add_subplot(121, projection='3d')
            centers_scatter = ax1.scatter(center_data_df['X'], center_data_df['Y'],
center_data_df['Z'], c='black', marker='X',
                                          s=100, alpha=1.0, label='Cluster Centers')
            scatter = ax1.scatter(clustered_data_df['X'],clustered_data_df['Y'],
```

```
clustered_data_df['Z'], c=sorted_cluster_labels, cmap='jet',
                                marker='o', s=30, alpha=0.3, label='Data Points')

            ax1.set_title(f'3D Scatter Plot with Cluster Labels and Centers: {num_clusters}
Clusters, Attempt {attempt}')
            ax1.set_xlabel('Data X')
            ax1.set_ylabel('Data Y')
            ax1.set_zlabel('Data Z')
            ax1.legend()

            # Save the plots
            plot_filename = f'kmeans_plots_Attempt_{attempt}
_Num_Clusters_{num_clusters}.png'
            plot_filepath = os.path.join(plot_filename)
            plt.tight_layout()
            plt.savefig(plot_filepath)
            #plt.show()  # Display the plot
            images_attempt.append(plot_filepath)  # Append the plot to the list
            plt.close()

        # Convert the images for the current number of clusters to a GIF
        gif_filename = f'kmeans_plots_Num_Clusters_{num_clusters}.gif'
        with imageio.get_writer(gif_filename, mode='I', fps=1, loop=0) as writer_attempt:
            for image_filename in images_attempt:
                # Adjust the image filename to include the subfolder
                image = imageio.imread(image_filename)
                writer_attempt.append_data(image)

                # Remove individual plot files after adding to GIF
                os.remove(image_filename)

        # Save the GIF to the current cluster folder
        zip_file.write(gif_filename)

        # Remove the GIF file after adding to the zip file
        os.remove(gif_filename)

        ### Section 8.3 Calculate MCASD metrics ###

        # Calculate Standard Deviation along each row
        row_std_dev = distances_df.std(axis=1)

        # Calculate Average of Standard Deviation for all Rows
        avg_std_dev = row_std_dev.mean()

        # Calculate Standard Deviation of the first Standard Deviation for all rows
        error = row_std_dev.std(axis=0)

        # Save values in the mcasd_metrics_df DataFrame
        mcasd_metrics_df.at['MCASD Metric', num_clusters] = avg_std_dev
        mcasd_metrics_df.at['MCASD Error', num_clusters] = error

        ### Section 8.4 Save results for final attempt at Cluster number to a CSV ###

        # Output file names
```

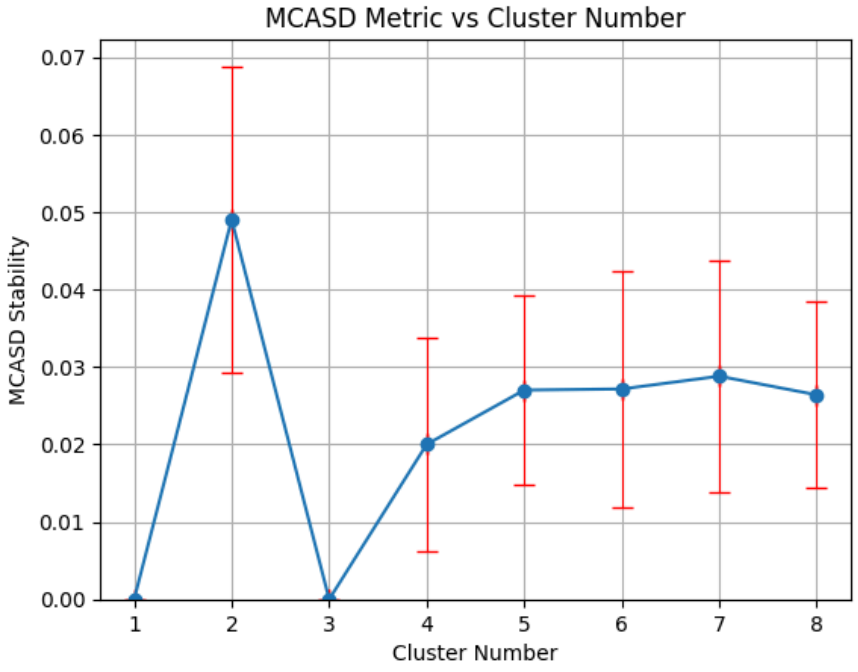
```
        output_cluster_filename = f'AgroGeo24_WS_Part_1_kmeans_{num_clusters}
_cluster_data.csv'
        output_center_filename = f'AgroGeo24_WS_Part_1_kmeans_{num_clusters}
_cluster_centers.csv'

        # Create a DataFrame with X, Y, Cluster, and Remaining Data
        clustered_data_df = pd.DataFrame({
            'Cluster Number': sorted_cluster_labels,
            'MCASD Metric': row_std_dev,
            **{f'{col}': remaining_data[col] for col in remaining_data.columns}
        })
        clustered_data_df = clustered_data_df.round(4)

        # Save the DataFrame to a CSV file
        clustered_data_df.to_csv(output_cluster_filename, index=False)

        # Create a DataFrame with Cluster center data
        center_data_df = pd.DataFrame(cluster_centers_original_scale,
columns=remaining_data.columns)

        # Add a new column 'Cluster Number' to indicate the cluster number for each row
        center_data_df.insert(0, 'Cluster Number', range(num_clusters))
        center_data_df = center_data_df.round(4)

        # Save the DataFrame to a CSV file
        center_data_df.to_csv(output_center_filename, index=False)

        # Save the csv to the current cluster folder
        zip_file.write(output_cluster_filename)
        zip_file.write(output_center_filename)

        # Remove the csv file after adding to the zip file
        os.remove(output_cluster_filename)
        os.remove(output_center_filename)

    # Save mcasd_metrics_df to a CSV file
    mcasd_metrics_csv_filename = 'mcasd_metrics.csv'
    mcasd_metrics_df.to_csv(mcasd_metrics_csv_filename)

    # Add mcasd_metrics CSV file to the zip file
    zip_file.write(mcasd_metrics_csv_filename)

    # Remove the mcasd_metrics CSV file after adding to the zip file
    os.remove(mcasd_metrics_csv_filename)

    ### Section 8.5 Create MCASD metric plot for visual QC ###

    # Make a 2D Line plot
    plt.errorbar(mcasd_metrics_df.columns, mcasd_metrics_df.loc['MCASD Metric'],
                 yerr=mcasd_metrics_df.loc['MCASD Error'], xerr=0, fmt='-o', capsize=5,
ecolor='red', errorevery=1,
                 elinewidth=0.8)
    plt.xlabel('Cluster Number')
    plt.ylabel('MCASD Stability')
    plt.title('MCASD Metric vs Cluster Number')
```

```
    plt.ylim(0)  # Set Y Axis starting at 0
    plt.xticks(np.arange(1, max_num_clusters + 1, 1))  # Set X Tick marks at all integers
    plt.grid(True)

    # Save the 2D line plot to the zip file
    line_plot_filename = 'mcasd_line_plot.png'
    plt.savefig(line_plot_filename)
    zip_file.write(line_plot_filename)
    os.remove(line_plot_filename)  # Remove the saved file after adding to the zip file

# Inform the user that the MCASD Method clustering is complete
print("\nMCASD Method clustering complete.")

### Section 8.6 Save the final zip file to the user's local machine ###

# Move the zip file to the user's local machine
files.download(zip_filename)
```

```
Enter the maximum number of clusters for MCASD Method: 8
Enter the maximum number of attempts for MCASD Method: 5

Calculating MCASD Metrics...

MCASD Method clustering complete.
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```



2. Introduction to Centroid Clustering using Rice Grain Model

In this part of the workshop you will be provided with a Black and White image of Rice.

1. The Rice Grains are randomly orientated
2. The background changes from light to dark in intensity

The image can be considered analogous to a remote sensing image, with a variable background and anomalies superimposed within this changing background. Here the “data” are called Pixel Intensity and range from 1 - 256 defining the colour from black to white.

The aim of this workshop is to determine the appropriate number of clusters for this image by considering the Elbow and Silhouette scores and MCASD metrics.KMeans clustering is the clustering methodology.

The input file name is: **“AgroGeo24_WS_Part_2_Rice_Grain_Model.jpg”** and should be located in the folder named “Part 2” of the data directory provided.

Please following along with the workshop leader in the first instance until you are familiar with using Google Colab environment.

No coding experience is required to run this code. All the code contains comments describing what each line does. Please click “Show Code” on any section to view the code.

Please feel free to ask questions if you don’t understand any parts.

2.1. Section 0

This section sets up the Python environment for the clustering analysis.

It imports essential libraries such as Pandas for data manipulation, NumPy for numerical operations, Matplotlib for data visualization, scikit-learn for machine learning tools, and other supporting libraries.

Additionally, it configures the display.

The code also imports specific functions and modules required for the clustering analysis, such as KMeans.

Finally, it sets up tools for working with images, zip files, and file uploads in Google Colab. This preparation ensures that the subsequent code can efficiently perform clustering analysis and handle related tasks.

```
# @title
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
import imageio.v2 as imageio
import matplotlib
import imageio.v2 as imageio
import io
import os
import re

from sklearn.cluster import KMeans
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import normalize
from sklearn.preprocessing import scale
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_score
from sklearn.metrics import pairwise_distances_argmin_min
```

```
from collections import Counter
from sklearn.preprocessing import MinMaxScaler
from matplotlib.ticker import MaxNLocator
from matplotlib.colors import ListedColormap
from IPython.display import display, clear_output
from PIL import Image
from zipfile import ZipFile
from google.colab import files


# Set display options to show all rows and columns
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
```

2.2. Section 1

Now we are going to read in the data from your local device. If you have not yet downloaded the necessary data, please do so from here.

The image is in greyscale format.

Press the play button below to select the correct file. It should be called:
"AgroGeo24_WS_Part_2_Rice_Grain_Mode.jpg"

You will see a button "Choose files". Click this and navigate to where this file is stored on your machine. Then highlight it and click "open".

You will see a progress message as the file is uploaded to this Google Colab environment.

```
# @title
#### Section 1: Import Grayscale .jpg image  ####

# Prompt the user to upload a file
uploaded = files.upload()

# Get the uploaded file name
image_file_name = list(uploaded.keys())[0]

# Extract the filename without the extension
image_file_name_without_extension = os.path.splitext(image_file_name)[0]

# Remove numerical suffixes from the filename
image_file_name_without_extension = re.sub(r'\(\d+\)', '',
image_file_name_without_extension)
image_file_name_without_extension = re.sub(r'\(\ \)', '', image_file_name_without_extension)

# Read the image into a DataFrame
image = Image.open(io.BytesIO(uploaded[image_file_name])) # Convert to grayscale
image_array = np.array(image)[:,:,0]
height, width = image_array.shape
coordinates = np.array(np.meshgrid(np.arange(width), np.arange(height))).T.reshape(-1, 2)
pixel_values = image_array.flatten()

# Create DataFrame with X Coordinate, Y Coordinate, and pixel intensity
df_image = pd.DataFrame({
    'X': coordinates[:, 1],
```

```
    'Y': coordinates[:, 0],
    'Pixel Intensity': pixel_values
})
```

Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving AgroGeo24_WS_Part_2_Rice_Grain_Model.jpg to AgroGeo24_WS_Part_2_Rice_Grain_Model.jpg
```

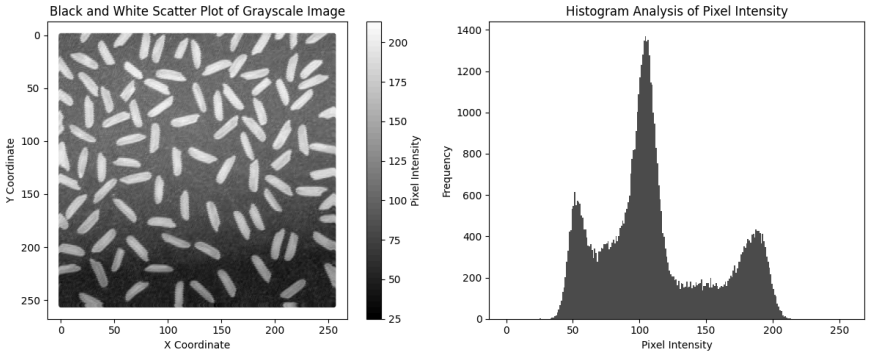
Excellent. You have now loaded the image into the Google Colab environment and are ready to take a look at the data.

2.3. Section 2

We now visualize the image. The color intensity in each plot reflects the values of the corresponding data column.

This visualization provides an initial exploration of the image and how the pixel intensities are distributed in the “geographical” space, setting the stage for further analysis and insights.

```
# @title
#### Section 2: Plot Grayscale Image as Black and White Scatter Plot with Histogram ####

# Set up a 1x2 subplot grid
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# Plot 1: Black and White Scatter Plot
scatter_plot = axes[0].scatter(df_image['X'], df_image['Y'], c=df_image['Pixel Intensity'],
cmap='gray', marker='o', s=10)
axes[0].set_title('Black and White Scatter Plot of Grayscale Image')
axes[0].set_xlabel('X Coordinate')
axes[0].set_ylabel('Y Coordinate')
axes[0].invert_yaxis()  # Invert the y-axis to match the image orientation
cbar = plt.colorbar(scatter_plot, ax=axes[0], orientation='vertical')
cbar.set_label('Pixel Intensity')

# Plot 2: Histogram Analysis of Pixel Intensity
axes[1].hist(df_image['Pixel Intensity'], bins=256, range=(0, 256), color='black',
alpha=0.7)
axes[1].set_title('Histogram Analysis of Pixel Intensity')
axes[1].set_xlabel('Pixel Intensity')
axes[1].set_ylabel('Frequency')

# Adjust layout for better spacing
plt.tight_layout()

# Show the plots
plt.show()
```



2.4. Section 3

We now visualize the image again but this time we're going to let you try to cluster this data using your intutition!

Using the data above (image and histogram) in Section 2, try to decide the number and location of the cluster centers.

The code will ask you for the number of clusters, and then for their cluster center position.

It will then calculate the quantizitation error, which is the mean of the distance between each data point (pixel intensity) and it's closest cluster center that you have picked.

One of the aims of clustering is to reduce this error.

Run this code a few times and try to get a low quantization error as you can with as few clusters as you can. Remember, we are trying to describe this full dataset with just a few data points (cluster centers).

Hopefully you will see that the human brain is already quite good at clustering!

How many clusters do you think there are?? And what are their cluster centers.

Once you are happy with your selection, please move on to the next section, where we will use traditional Elbow and Silhoutte and more modern MCASD to see how close you can get!

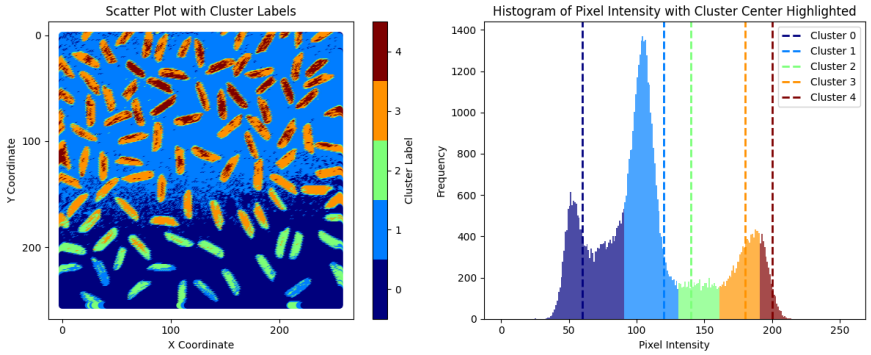
```
# @title
#### Section 3: KMeans Clustering and Visualization ####

# Extract the Pixel Intensity data
coordinates = df_image.iloc[:, :2]

# Extract the Pixel Intensity data
remaining_data = df_image.iloc[:, 2:3]
clustering_data = df_image['Pixel Intensity'].values.reshape(-1, 1)


num_clusters = []
# Get the number of clusters from the user
num_clusters = int(input("Enter the number of clusters: "))

# Initialize variables to store user input for cluster centers
cluster_centers = []

# Use a while loop to repeatedly prompt the user for input until valid values are entered
while True:
```

```
    try:
        # Get input from the user for each cluster center
        for i in range(num_clusters):
            center = int(input(f"Enter value for center of cluster {i} (between 0 and 256):
"))
            cluster_centers.append(center)

        # Break out of the loop if the input is valid
        break

    except ValueError as e:
        print(f"Error: {e}. Please enter valid numeric values.")

# Convert cluster_centers to a numpy array
cluster_centers = np.array(cluster_centers).reshape(-1, 1)

# Calculate distances from each data point to each cluster center
distances = np.abs(clustering_data - cluster_centers.T)

# Calculate the mean of minimum distances
min_distance_indices = np.argmin(distances, axis=1)
min_distances = distances[np.arange(len(distances)), min_distance_indices]
mean_min_distance = np.mean(min_distances).round(4)

# Display the mean of minimum distances
print(f"\nThe quantization error for this attempt is: {mean_min_distance}\n")

# Store original inputs for repeated attempts
original_cluster_centers = cluster_centers.copy()
original_mean_min_distance = mean_min_distance

# Assign each data point to the cluster with the nearest cluster center
cluster_labels = np.argmin(distances, axis=1)

### Section 3.2 Sort the Cluster centers ###

# Calculate the distances of cluster centers from the origin (0)
distances_from_origin = np.sqrt(np.sum(cluster_centers ** 2, axis=1))

# Sort cluster centers based on distances from the origin
sorted_indices = np.argsort(distances_from_origin)

# Sort cluster centers and labels
sorted_cluster_centers = cluster_centers[sorted_indices]
sorted_cluster_labels = np.zeros_like(cluster_labels)

# Relabel the cluster labels based on the sorted order
for new_label, old_label in enumerate(sorted_indices):
    sorted_cluster_labels[cluster_labels == old_label] = new_label

# Calculate the count of each cluster label
cluster_labels_count = dict(zip(*np.unique(sorted_cluster_labels, return_counts=True)))

# Create a DataFrame with X, Y, Cluster, and Remaining Data
clustered_data_df = pd.DataFrame({
```

```
    'X': coordinates['X'],
    'Y': coordinates['Y'],
    'Cluster Number': sorted_cluster_labels,
    **{f'{col}': remaining_data[col] for col in remaining_data.columns}
})

### Section 3.3 Plot Human Clustering results ###
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Plot 1: Scatter Plot with Cluster Labels
scatter = ax1.scatter(coordinates['X'], coordinates['Y'], c=sorted_cluster_labels,
cmap='jet', marker='o', s=30)

# Set plot properties for Plot 1
ax1.set_title(f'Scatter Plot with Cluster Labels')
ax1.set_xlabel('X Coordinate')
ax1.set_ylabel('Y Coordinate')
ax1.invert_yaxis()  # Invert the y-axis to match the image orientation

# Create a discrete color map with the number of clusters for Plot 1
cmap_discrete = matplotlib.colormaps.get_cmap('jet')

# Define boundaries for the discrete color map for Plot 1
boundaries = np.arange(-0.5, num_clusters, 1)

# Create a BoundaryNorm for the color map for Plot 1
norm_discrete = mcolors.BoundaryNorm(boundaries, cmap_discrete.N, clip=True)

# Add a discrete color bar with integer cluster labels for Plot 1
cbar = plt.colorbar(scatter, ax=ax1, ticks=np.arange(num_clusters), cmap=cmap_discrete,
norm=norm_discrete,
                    boundaries=boundaries)
cbar.set_label('Cluster Label')

# Set the number of tick marks on the X and Y axes for Plot 1
ax1.xaxis.set_major_locator(MaxNLocator(nbins=3))
ax1.yaxis.set_major_locator(MaxNLocator(nbins=3))

for cluster_label, cluster_center in zip(cluster_labels_count.keys(),
sorted_cluster_centers):
    color = cmap_discrete(
        cluster_label / (num_clusters - 1) if num_clusters - 1 != 0 else 0.5)  # Use the
same cmap for pixel intensities
    # Get the pixel intensities assigned to the current cluster
    cluster_pixel_intensity = clustered_data_df[
        clustered_data_df['Cluster Number'] == cluster_label]['Pixel Intensity']
    # Calculate color bins based on the minimum and maximum pixel intensity values
    color_bins = [cluster_pixel_intensity.min(), cluster_pixel_intensity.max()]
    # Plot the histogram with specific bin edges and color
    hist, bin_edges, _ = ax2.hist(cluster_pixel_intensity, bins=256, range=(0, 256),
alpha=0.7, color=color)
    ax2.axvline(x=cluster_center, color=color, linestyle='dashed', linewidth=2,
                label=f'Cluster {cluster_label}')

# Set plot properties for Plot 2
```

```
ax2.set_title('Histogram of Pixel Intensity with Cluster Center Highlighted')
ax2.set_xlabel('Pixel Intensity')
ax2.set_ylabel('Frequency')

# Add a legend to distinguish cluster center lines
ax2.legend()

# Adjust layout for better spacing
plt.tight_layout()

# Show the plots
plt.show()
```

```
Enter the number of clusters: 5
Enter value for center of cluster 0 (between 0 and 256): 60
Enter value for center of cluster 1 (between 0 and 256): 120
Enter value for center of cluster 2 (between 0 and 256): 140
Enter value for center of cluster 3 (between 0 and 256): 180
Enter value for center of cluster 4 (between 0 and 256): 200

The quantization error for this attempt is: 12.2601
```



2.5. Section 4

In this section, we divide our dataset into two key components: coordinates and remaining data. The first two columns, containing coordinate information, are isolated to construct the “Coordinates” DataFrame. Simultaneously, the remaining data, excluding the initial two columns forms the “Remaining Data” DataFrame.

This separation serves a pivotal purpose—clustering analysis will solely operate on the remaining data. Subsequently, the cluster labels obtained can be associated with their respective X and Y coordinates. This distinction is fundamental for generating geographical cluster maps, allowing us to visually interpret and understand the spatial distribution of clusters across the dataset.

This section also performs data normalization, a crucial step before applying clustering algorithms.

The resulting normalized data is displayed, providing an insight into the standardized values across the dataset.

Normalization enhances the accuracy of clustering algorithms, ensuring that features with different scales contribute equally to the clustering process.

```
# @title
#### Section 4 Separate the Data for clustering ####

# Extract coordinate information (assuming it's in the first two columns)
coordinates = df_image.iloc[:, :2]

# Extract the Pixel Intensity data
remaining_data = df_image.iloc[:, 2:3]

# Display the Coordinates DataFrame
print("\nCoordinates:")
print(coordinates.head(10).to_string(index=False))  # Use to_string to prevent truncation
print(f"Number of rows: {df_image.shape[0]}")

# Display the Remaining Data DataFrame
print("\nRemaining Data:")
print(remaining_data.head(10).to_string(index=False))  # Use to_string to prevent truncation
print(f"Number of rows: {df_image.shape[0]}")

# Custom normalization using MinMaxScaler
min_vals = remaining_data.min()
max_vals = remaining_data.max()
remaining_data_column = ['Pixel Intensity']

normalized_data = (remaining_data - min_vals) / (max_vals - min_vals)

# Round the normalized data to 4 decimal places
normalized_data = normalized_data.round(4)

# Convert the rounded normalized data back to a DataFrame and set column names
normalized_df = pd.DataFrame(normalized_data, columns=remaining_data_column)

# Display the rounded normalized data
print("\nNormalized Data:")
print(normalized_df.head(10).to_string(index=False))  # Use to_string to prevent truncation
print(f"Number of rows: {df_image.shape[0]}")
```

```
Coordinates:
 X  Y
 0  0
 1  0
 2  0
 3  0
 4  0
 5  0
 6  0
 7  0
 8  0
 9  0
Number of rows: 65536

Remaining Data:
 Pixel Intensity
```

```
             116
              96
             101
              96
             108
             108
              79
              91
              91
             112
Number of rows: 65536

Normalized Data:
 Pixel Intensity
          0.4840
          0.3777
          0.4043
          0.3777
          0.4415
          0.4415
          0.2872
          0.3511
          0.3511
          0.4628
Number of rows: 65536
```

2.6. Section 5

This section guides the user in determining the optimal number of clusters (k) for the K-Means algorithm by utilizing both the Elbow Method and Silhouette Scores.

After specifying the maximum number of clusters to consider, the code calculates the Within-Cluster Sum of Squares (WCSS) distance using the Elbow Method. The Elbow Method graph illustrates the trade-off between clustering complexity and WCSS reduction, helping identify an optimal k value.

Simultaneously, Silhouette Scores, a measure of how well-separated clusters are, are computed and presented on the same graph. Silhouette Scores range from −1 to 1, where higher scores indicate better-defined clusters.

When assessing the graph, users should look for the “elbow” point where WCSS plateaus, suggesting diminishing returns with additional clusters.

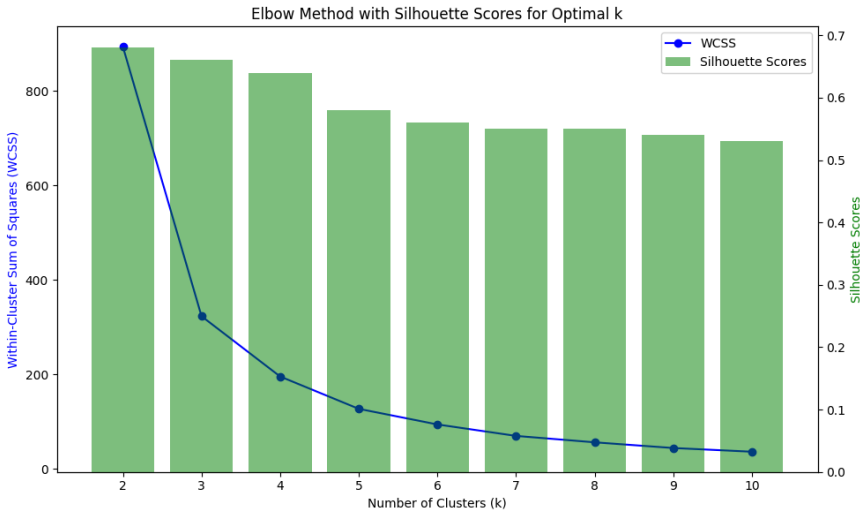
Additionally, a high Silhouette Score at the “elbow” reinforces the choice, ensuring a balance between compact clusters and distinct cluster boundaries for effective clustering.

When running the cell, you will be prompted to enter the max number of clusters. (i.e., 10).

```
# @title
#### Section 5 Elbow Method with Silhouette Scores ####

# Prompt the user to choose the maximum number of clusters for the Elbow Method
max_clusters_elbow = int(input("Enter the maximum number of clusters for the Elbow Method:
"))

# Calculate the within-cluster sum of squares (WCSS) and Silhouette Scores for different
values of k
wcss = []
```

```
silhouette = []
for k in range(2, max_clusters_elbow + 1):
    kmeans = KMeans(n_clusters=k, n_init=1, init='k-means++')
    kmeans.fit(normalized_data)
    sscore = round(silhouette_score(normalized_data, kmeans.labels_), 2)
    silhouette.append(sscore)
    wcss.append(kmeans.inertia_)

# Plot the Elbow Method graph with Silhouette Scores as a bar graph
fig, ax1 = plt.subplots(figsize=(10, 6))

# Plot WCSS on the left y-axis
ax1.plot(range(2, max_clusters_elbow + 1), wcss, marker='o', color='blue', label='WCSS')
ax1.set_xlabel('Number of Clusters (k)')
ax1.set_ylabel('Within-Cluster Sum of Squares (WCSS)', color='blue')

# Set the x-axis ticks to show only integers
plt.xticks(range(2, max_clusters_elbow + 1))

# Create a second y-axis for Silhouette Scores
ax2 = ax1.twinx()
ax2.bar(range(2, max_clusters_elbow + 1), silhouette, color='green', alpha=0.5,
label='Silhouette Scores')
ax2.set_ylabel('Silhouette Scores', color='green')

# Add legend
lines, labels = ax1.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
ax2.legend(lines + lines2, labels + labels2, loc='upper right')

plt.title('Elbow Method with Silhouette Scores for Optimal k')
fig.tight_layout()
plt.show()
```

```
Enter the maximum number of clusters for the Elbow Method: 10
```



2.7. Section 6

In this section, users will perform K-Means clustering on the data. The optimal number of clusters (k) can be determined by referencing the results from the previous Elbow Method and Silhouette Scores analysis (Section 5). After entering the desired number of clusters, the code applies K-Means clustering and displays key information.

The results include a count of occurrences for each cluster label via a plot of the classified image and histograph.

You will be prompted to enter the number of clusters, which should be based on the Elbow and Silhouette results, but feel free to run this code a few times for various number of clusters and take a look at the results!

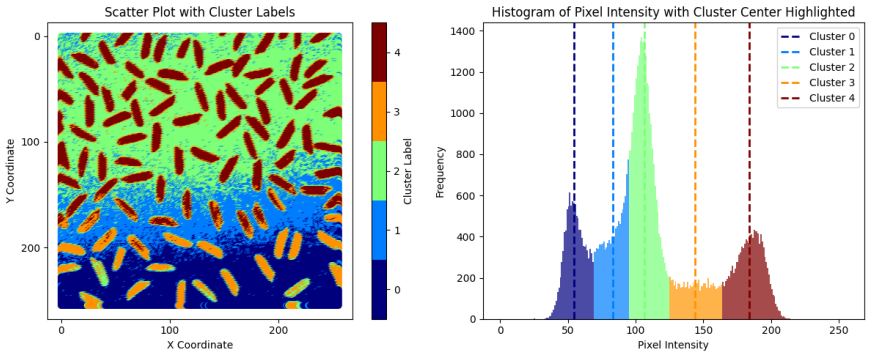
```
# @title
##### Section 6 K Means Clustering ####

### Section 6.1 Perform K Means Clustering ###

# Prompt the user to choose the number of clusters for K-Means
num_clusters = int(input("Enter the number of clusters for K-Means: "))

# Initialize the K-Means model
kmeans = KMeans(n_clusters=num_clusters, init = 'k-means++', n_init=1)

# Fit the K-Means model to the normalized data
kmeans.fit(normalized_data)

# Get the cluster labels for each data point
cluster_labels = kmeans.labels_

# Get the cluster centers
cluster_centers = kmeans.cluster_centers_

### Section 6.2 Sort the Cluster centers ###

# Calculate the distances of cluster centers from the origin (0, 0)
distances_from_origin = np.sqrt(np.sum(cluster_centers ** 2, axis=1))

# Sort cluster centers based on distances from the origin
sorted_indices = np.argsort(distances_from_origin)

# Sort cluster centers and labels
sorted_cluster_centers = cluster_centers[sorted_indices]
sorted_cluster_labels = np.zeros_like(cluster_labels)

# Relabel the cluster labels based on the sorted order
for new_label, old_label in enumerate(sorted_indices):
    sorted_cluster_labels[cluster_labels == old_label] = new_label

# Calculate the count of each cluster label
cluster_labels_count = dict(zip(*np.unique(sorted_cluster_labels, return_counts=True)))

### Section 6.3 Denormalize the cluster centers ###

# Denormalize the data using the inverse transformation
cluster_centers_original_scale = sorted_cluster_centers * (max_vals.values -
min_vals.values) + min_vals.values
```

```
clustering_data = df_image['Pixel Intensity'].values.reshape(-1, 1)
# Calculate distances from each data point to each cluster center
distances = np.abs(clustering_data - cluster_centers_original_scale.T)

# Calculate the mean of minimum distances
min_distance_indices = np.argmin(distances, axis=1)
min_distances = distances[np.arange(len(distances)), min_distance_indices]
mean_min_distance = np.mean(min_distances).round(4)

### Section 6.4 Display Clustering counts for visual QC ###

# Display the count of each cluster label
print("\nCount of Each Cluster Label:")
for label, count in cluster_labels_count.items():
    print(f"Cluster {label}: {count} occurrences")

print(f"\nCluster Centers: {cluster_centers_original_scale.round(2).flatten()}")

# Display the mean of minimum distances
print(f"\nThe quantization error for this attempt is: {mean_min_distance}\n")

### Section 6.5 Save results to Data Frame ###

# Create a DataFrame with X, Y, Cluster, and Remaining Data
clustered_data_df = pd.DataFrame({
    'X': coordinates['X'],
    'Y': coordinates['Y'],
    'Cluster Number': sorted_cluster_labels,
    **{f'{col}': remaining_data[col] for col in remaining_data.columns}
})
clustered_data_df = clustered_data_df.round(4)

# Create a DataFrame with Cluster center data
center_data_df = pd.DataFrame(cluster_centers_original_scale,
columns=remaining_data.columns)

# Add a new column 'Cluster Number' to indicate the cluster number for each row
center_data_df.insert(0, 'Cluster Number', range(num_clusters))
center_data_df = center_data_df.round(4)

### Section 6.6 Plot KMeans Clustering results ###
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Plot 1: Scatter Plot with Cluster Labels
scatter = ax1.scatter(coordinates['X'], coordinates['Y'], c=sorted_cluster_labels,
cmap='jet', marker='o', s=30)

# Set plot properties for Plot 1
ax1.set_title(f'Scatter Plot with Cluster Labels')
ax1.set_xlabel('X Coordinate')
ax1.set_ylabel('Y Coordinate')
ax1.invert_yaxis()  # Invert the y-axis to match the image orientation

# Create a discrete color map with the number of clusters for Plot 1
cmap_discrete = matplotlib.colormaps.get_cmap('jet')
```

```
# Define boundaries for the discrete color map for Plot 1
boundaries = np.arange(-0.5, num_clusters, 1)

# Create a BoundaryNorm for the color map for Plot 1
norm_discrete = mcolors.BoundaryNorm(boundaries, cmap_discrete.N, clip=True)

# Add a discrete color bar with integer cluster labels for Plot 1
cbar = plt.colorbar(scatter, ax=ax1, ticks=np.arange(num_clusters), cmap=cmap_discrete,
norm=norm_discrete, boundaries=boundaries)
cbar.set_label('Cluster Label')

# Set the number of tick marks on the X and Y axes for Plot 1
ax1.xaxis.set_major_locator(MaxNLocator(nbins=3))
ax1.yaxis.set_major_locator(MaxNLocator(nbins=3))

for cluster_label, cluster_center in zip(cluster_labels_count.keys(),
cluster_centers_original_scale):
    color = cmap_discrete(cluster_label / (num_clusters - 1) if num_clusters - 1 != 0 else
0.5)  # Use the same cmap for pixel intensities
    # Get the pixel intensities assigned to the current cluster
    cluster_pixel_intensity = clustered_data_df[clustered_data_df['Cluster Number'] ==
cluster_label]['Pixel Intensity']
    # Calculate color bins based on the minimum and maximum pixel intensity values
    color_bins = [cluster_pixel_intensity.min(), cluster_pixel_intensity.max()]
    # Plot the histogram with specific bin edges and color
    hist, bin_edges, _ = ax2.hist(cluster_pixel_intensity, bins=256, range=(0, 256),
alpha=0.7, color = color)
    ax2.axvline(x=cluster_center, color=color, linestyle='dashed', linewidth=2,
label=f'Cluster {cluster_label}')

# Set plot properties for Plot 2
ax2.set_title('Histogram of Pixel Intensity with Cluster Center Highlighted')
ax2.set_xlabel('Pixel Intensity')
ax2.set_ylabel('Frequency')

# Add a legend to distinguish cluster center lines
ax2.legend()

# Adjust layout for better spacing
plt.tight_layout()

# Show the plots
plt.show()
```

```
Enter the number of clusters for K-Means: 5

Count of Each Cluster Label:
Cluster 0: 11044 occurrences
Cluster 1: 11304 occurrences
Cluster 2: 24961 occurrences
Cluster 3: 6485 occurrences
Cluster 4: 11742 occurrences
```

```
Cluster Centers: [ 54.75  83.22 106.29 143.68 183.97]

The quantization error for this attempt is: 6.7965
```



2.8. Section 7

This section applies the MCASD (Multiple Cluster Average Standard Deviation) method, designed to aid participants in identifying the optimal number of clusters for their dataset.

MCASD was first published as a method by O'Leary et al 2023.

MCASD evaluates the stability of cluster centers across multiple attempts and cluster numbers. Participants will input the maximum number of clusters and the maximum number of attempts per cluster.

The code loops through different cluster numbers, applying K-Means clustering multiple times to analyze stability.

The results include GIFs illustrating scatter plots and line plots for each attempt. Additionally, MCASD metrics are calculated, providing insights into the stability and consistency of clusters. A line plot visualizes the MCASD metric across various cluster numbers, aiding participants in selecting the optimal cluster count.

All results, including csv files, plots and gifs, are compressed into a zip file for easy download

Please save this ZIP file to "Part 2" of the data directory you were provided with and unzip it to view the outputs from MCASD analysis.

The ultimate goal is to assist participants in making informed decisions about the optimal number of clusters for their specific dataset.

When running the cell, you will be prompted to enter the number of max number of clusters (i.e., 10) and max attempts (i.e., 10). Note this might take some time depending on the max number of clusters and max attempts chosen.

```
# @title
#### Section 7 MCASD Method ####
### Section 7.1 Get information from the user ###

# Prompt the user for the maximum number of clusters for MCASD Method
max_num_clusters = int(input("Enter the maximum number of clusters for MCASD Method: "))
# Prompt the user for the maximum number of attempts for MCASD Method
max_attempts = int(input("Enter the maximum number of attempts for MCASD Method: "))
```

```
### Section 7.2 Loop for MCASD Method ###
# Create a DataFrame to store MCASD Metrics
mcasd_metrics_df = pd.DataFrame(index=['MCASD Metric', 'MCASD Error'], columns=range(1,
max_num_clusters + 1))

print(f"\nCalculating MCASD Metrics...")

# Create a zip file to store all results
zip_filename = f'AgroGeo24_WS_Part_2_kmeans_plots.zip'
with ZipFile(zip_filename, 'w') as zip_file:

    # Loop through the various number of clusters
    for num_clusters in range(1, max_num_clusters + 1):
        images_attempt = []  # List to store images for the current attempt
        distances_df = pd.DataFrame()  # Initialize distances DataFrame

        # Cluster the data a user specified number of times (Attempts)
        for attempt in range(1, max_attempts + 1):
            #print(f"\nNumber of Clusters: {num_clusters}: Attempt {attempt} of
{max_attempts}")

            # Initialize the K-Means model
            kmeans = KMeans(n_clusters=num_clusters, n_init=1, init='k-means++')

            # Fit the K-Means model to the normalized data
            kmeans.fit(normalized_data)

            # Get the cluster labels for each data point
            cluster_labels = kmeans.labels_

            # Get the cluster centers
            cluster_centers = kmeans.cluster_centers_

            ### Section 7.2.1 Sort the Cluster centers ###

            # Calculate the distances of cluster centers from the origin (0, 0)
            distances_from_origin = np.sqrt(np.sum(cluster_centers ** 2, axis=1))

            # Sort cluster centers based on distances from the origin
            sorted_indices = np.argsort(distances_from_origin)

            # Sort cluster centers and labels
            sorted_cluster_centers = cluster_centers[sorted_indices]
            sorted_cluster_labels = np.zeros_like(cluster_labels)

            # Relabel the cluster labels based on the sorted order
            for i, new_label in enumerate(np.arange(num_clusters)):
                old_label = sorted_indices[i]
                sorted_cluster_labels[cluster_labels == old_label] = new_label

            ### Section 7.2.2 Calculate the distance (in the dataspace) between each
datapoint and its closest cluster center ###

            # Calculate the distances between cluster centers and data
            distances = np.linalg.norm(normalized_data.values[:, np.newaxis, :] -
```

```
cluster_centers, axis=-1)

            # Get the smallest distance for each data point
            min_distances = np.min(distances, axis=1)

            # Create a DataFrame for distances with only the smallest distances
            new_column = pd.DataFrame(min_distances, columns=[f'Attempt_{attempt}'])

            # Append the new column to the existing distances_df
            distances_df = pd.concat([distances_df, new_column], axis=1)

            # Calculate the count of each cluster label
            cluster_labels_count = dict(zip(*np.unique(sorted_cluster_labels,
return_counts=True)))

            # Create a DataFrame with X, Y, Cluster, and Remaining Data
            clustered_data_df = pd.DataFrame({
                'X': coordinates['X'],
                'Y': coordinates['Y'],
                'Cluster Number': sorted_cluster_labels,
                **{f'{col}': remaining_data[col] for col in remaining_data.columns}
            })
            clustered_data_df = clustered_data_df.round(4)

            ### Section 7.2.3 Denormalize the cluster centers ###

            cluster_centers_original_scale = sorted_cluster_centers * (max_vals.values -
min_vals.values) + min_vals.values

            ### Section 8.2.4 Create and save plots for later GIF creation ###

            fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

            # Plot 1: Scatter Plot with Cluster Labels
            scatter = ax1.scatter(coordinates['X'], coordinates['Y'],
c=sorted_cluster_labels, cmap='jet', marker='o', s=30)

            # Set plot properties for Plot 1
            ax1.set_title(f'Scatter Plot with Cluster Labels: {num_clusters} Clusters,
Attempt {attempt}')
            ax1.set_xlabel('X Coordinate')
            ax1.set_ylabel('Y Coordinate')
            ax1.invert_yaxis()  # Invert the y-axis to match the image orientation

            # Create a discrete color map with the number of clusters for Plot 1
            cmap_discrete = matplotlib.colormaps.get_cmap('jet')

            # Define boundaries for the discrete color map for Plot 1
            boundaries = np.arange(-0.5, num_clusters, 1)

            # Create a BoundaryNorm for the color map for Plot 1
            norm_discrete = mcolors.BoundaryNorm(boundaries, cmap_discrete.N, clip=True)

            # Add a discrete color bar with integer cluster labels for Plot 1
            cbar = plt.colorbar(scatter, ax=ax1, ticks=np.arange(num_clusters),
```

```
cmap=cmap_discrete, norm=norm_discrete, boundaries=boundaries)
            cbar.set_label('Cluster Label')

            # Set the number of tick marks on the X and Y axes for Plot 1
            ax1.xaxis.set_major_locator(MaxNLocator(nbins=3))
            ax1.yaxis.set_major_locator(MaxNLocator(nbins=3))

            for cluster_label, cluster_center in zip(cluster_labels_count.keys(),
cluster_centers_original_scale):
                color = cmap_discrete(cluster_label / (num_clusters - 1) if num_clusters -
1 != 0 else 0.5)  # Use the same cmap for pixel intensities
                # Get the pixel intensities assigned to the current cluster
                cluster_pixel_intensity = clustered_data_df[clustered_data_df['Cluster
Number'] == cluster_label]['Pixel Intensity']
                # Calculate color bins based on the minimum and maximum pixel intensity
values
                color_bins = [cluster_pixel_intensity.min(), cluster_pixel_intensity.max()]
                # Plot the histogram with specific bin edges and color
                hist, bin_edges, _ = ax2.hist(cluster_pixel_intensity, bins=256, range=(0,
256), alpha=0.7, color = color)
                ax2.axvline(x=cluster_center, color=color, linestyle='dashed', linewidth=2,
label=f'Cluster {cluster_label}')

            # Set plot properties for Plot 2
            ax2.set_title('Histogram of Pixel Intensity with Cluster Center Highlighted')
            ax2.set_xlabel('Pixel Intensity')
            ax2.set_ylabel('Frequency')

            # Add a legend to distinguish cluster center lines
            ax2.legend()

            # Save the plots
            plot_filename = f'kmeans_plots_Attempt_{attempt}
_Num_Clusters_{num_clusters}.png'
            plot_filepath = os.path.join(plot_filename)
            plt.tight_layout()
            plt.savefig(plot_filepath)
            #plt.show()  # Display the plot
            images_attempt.append(plot_filepath)  # Append the plot to the list
            plt.close()

        # Convert the images for the current number of clusters to a GIF
        gif_filename = f'kmeans_plots_Num_Clusters_{num_clusters}.gif'
        with imageio.get_writer(gif_filename, mode='I', fps=1, loop=0) as writer_attempt:
            for image_filename in images_attempt:
                # Adjust the image filename to include the subfolder
                image = imageio.imread(image_filename)
                writer_attempt.append_data(image)

                # Remove individual plot files after adding to GIF
                os.remove(image_filename)

        # Save the GIF to the current cluster folder
        zip_file.write(gif_filename)
```

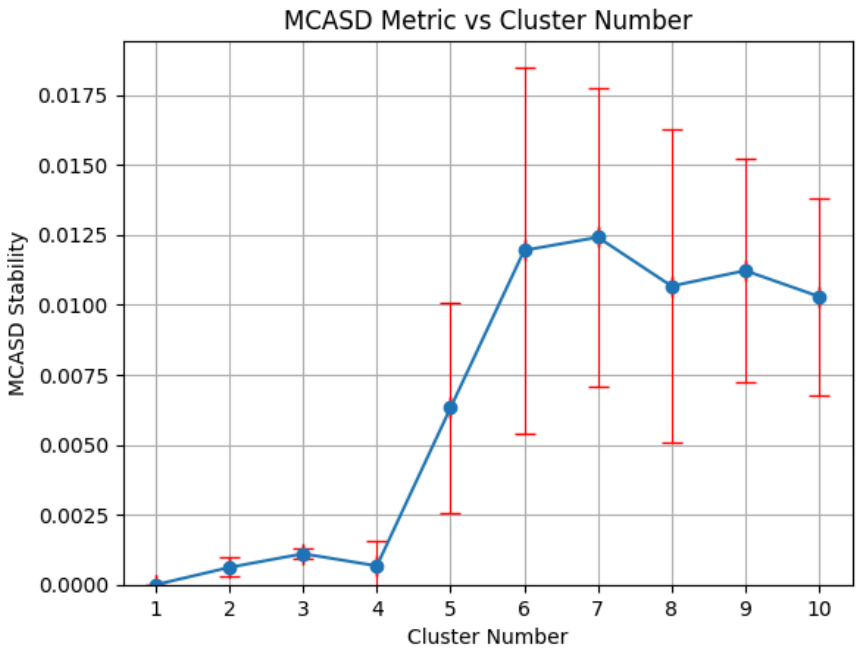
```
        # Remove the GIF file after adding to the zip file
        os.remove(gif_filename)

        ### Section 7.3 Calculate MCASD metrics ###

        # Calculate Standard Deviation along each row
        row_std_dev = distances_df.std(axis=1)

        # Calculate Average of Standard Deviation for all Rows
        avg_std_dev = row_std_dev.mean()

        # Calculate Standard Deviation of the first Standard Deviation for all rows
        error = row_std_dev.std(axis=0)

        # Save values in the mcasd_metrics_df DataFrame
        mcasd_metrics_df.at['MCASD Metric', num_clusters] = avg_std_dev
        mcasd_metrics_df.at['MCASD Error', num_clusters] = error

        ### Section 7.4 Save results for final attempt at Cluster number to a CSV ###

        # Output file names
        output_cluster_filename = f'{image_file_name_without_extension}
_kmeans_{num_clusters}_cluster_data.csv'
        output_center_filename = f'{image_file_name_without_extension}_kmeans_{num_clusters}
_cluster_centers.csv'

        # Create a DataFrame with X, Y, Cluster, and Remaining Data
        clustered_data_df = pd.DataFrame({
            'X': coordinates['X'],
            'Y': coordinates['Y'],
            'Cluster Number': sorted_cluster_labels,
            'MCASD Metric': row_std_dev,
            **{f'{col}': remaining_data[col] for col in remaining_data.columns}
        })
        clustered_data_df = clustered_data_df.round(4)

        # Save the DataFrame to a CSV file
        clustered_data_df.to_csv(output_cluster_filename, index=False)

        # Create a DataFrame with Cluster center data
        center_data_df = pd.DataFrame(cluster_centers_original_scale,
columns=remaining_data.columns)

        # Add a new column 'Cluster Number' to indicate the cluster number for each row
        center_data_df.insert(0, 'Cluster Number', range(num_clusters))
        center_data_df = center_data_df.round(4)

        # Save the DataFrame to a CSV file
        center_data_df.to_csv(output_center_filename, index=False)

        # Save the csv to the current cluster folder
        zip_file.write(output_cluster_filename)
        zip_file.write(output_center_filename)

        # Remove the csv file after adding to the zip file
```

```
        os.remove(output_cluster_filename)
        os.remove(output_center_filename)

    # Save mcasd_metrics_df to a CSV file
    mcasd_metrics_csv_filename = 'mcasd_metrics.csv'
    mcasd_metrics_df.to_csv(mcasd_metrics_csv_filename)

    # Add mcasd_metrics CSV file to the zip file
    zip_file.write(mcasd_metrics_csv_filename)

    # Remove the mcasd_metrics CSV file after adding to the zip file
    os.remove(mcasd_metrics_csv_filename)

    ### Section 7.5 Create MCASD metric plot for visual QC ###

    # Make a 2D Line plot
    plt.errorbar(mcasd_metrics_df.columns, mcasd_metrics_df.loc['MCASD Metric'],
                 yerr=mcasd_metrics_df.loc['MCASD Error'], xerr=0, fmt='-o', capsize=5,
ecolor='red', errorevery=1,
                 elinewidth=0.8)
    plt.xlabel('Cluster Number')
    plt.ylabel('MCASD Stability')
    plt.title('MCASD Metric vs Cluster Number')
    plt.ylim(0)  # Set Y Axis starting at 0
    plt.xticks(np.arange(1, max_num_clusters + 1, 1))  # Set X Tick marks at all integers
    plt.grid(True)

    # Save the 2D line plot to the zip file
    line_plot_filename = 'mcasd_line_plot.png'
    plt.savefig(line_plot_filename)
    zip_file.write(line_plot_filename)
    os.remove(line_plot_filename)  # Remove the saved file after adding to the zip file

# Inform the user that the MCASD Method clustering is complete
print("\nMCASD Method clustering complete.")

### Section 7.6 Save the final zip file to the user's local machine ###

# Move the zip file to the user's local machine
files.download(zip_filename)
```

```
Enter the maximum number of clusters for MCASD Method: 10
Enter the maximum number of attempts for MCASD Method: 10

Calculating MCASD Metrics...

MCASD Method clustering complete.
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```



3. Using Centroid Clustering on real geophysical data (EMI data)

In this part of the workshop you will be provided with data from an Electromagnetic Induction Survey.

1. The data were acquired with a CMD Mini Explorer 6L
2. The data were processed using a noise reduction flow
3. The data were gridded to a 1 x 1 m grid
4. All coordinate information has been altered to keep the privacy of the land owner

The aim of this workshop is to determine the appropriate number of clusters for the EMI data using KMeans clustering. We will compare some tradition methods with a recently proposed MCASD method.

The input file name is: “AgroGeo24_WS_Part_3_EMI.csv” and should be located in the folder named “Part 3” of the data directory provided.

Please following along with the workshop leader in the first instance until you are familiar with using Google Colab environment.

No coding experience is required to run this code. All the code contains comments describing what each line does. Please click “Show Code” on any section to view the code.

Please feel free to ask questions if you don’t understand any parts.

3.1. Section 0

This section sets up the Python environment for the clustering analysis.

It imports essential libraries such as Pandas for data manipulation, NumPy for numerical operations, Matplotlib for data visualization, scikit-learn for machine learning tools, and other supporting libraries.

Additionally, it configures the display.

The code also imports specific functions and modules required for the clustering analysis, such as KMeans.

Finally, it sets up tools for working with images, zip files, and file uploads in Google Colab. This preparation ensures that the subsequent code can efficiently perform clustering analysis and handle related tasks.

```
# @title
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
import imageio.v2 as imageio
import matplotlib
import imageio.v2 as imageio
import os
import re

from sklearn.cluster import KMeans
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import normalize
from sklearn.preprocessing import scale
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_score
from sklearn.metrics import pairwise_distances_argmin_min
from collections import Counter
from sklearn.preprocessing import MinMaxScaler
from matplotlib.ticker import MaxNLocator
from PIL import Image
from zipfile import ZipFile
from google.colab import files


# Set display options to show all rows and columns
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
```

3.2. Section 1

Now we are going to read in the data from your local device. If you have not yet downloaded the necessary data, please do so from here.

The Data are in CSV format with the following header descriptors:

1. Column 1 = X Coordinate
2. Column 2 = Y Coordinate
3. Columns 3..N = Data

Press the play button below to select the correct file. It should be called: `AgroGeo24_WS_Part_3.csv`

You will see a button “Choose files”. Click this and navigate to where this file is stored on your machine. Then highlight it and click “open”.

When running the cell, you will see a progress message as the file is uploaded to this Google Colab environment.

```
# @title
#### Section 1: Import CSV file  ####

# Prompt the user to upload a file
uploaded = files.upload()
```

```
# Get the uploaded file name
file_name = list(uploaded.keys())[0]

# Extract the filename without the extension
file_name_without_extension = os.path.splitext(file_name)[0]

# Remove numerical suffixes from the filename
file_name_without_extension = re.sub(r'\(\d+\)', '', file_name_without_extension)
file_name_without_extension = re.sub(r'\(\ \)', '', file_name_without_extension)

# Read the CSV file into a DataFrame
df = pd.read_csv(file_name).round(2)
```

Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving AgroGeo24_WS_Part_3_EMI.csv to AgroGeo24_WS_Part_3_EMI.csv
```

Excellent. You have now loaded the CSV file into the Google Colab environment and are ready to take a look at the data.

3.3. Section 2

In this section, we're inspecting and refining the dataset. Initially, we showcase a snippet of the original dataset, including its structure and the number of rows.

Next, we perform data cleaning by eliminating rows containing NaN (Not a Number) and blank values. The cleaned dataset is then displayed, again with a snippet and the updated row count.

This process ensures that the dataset is well-prepared for subsequent analyses by removing any instances of missing or empty data.

Note that this dataset contained no blank values, so the number of rows doesn't change.

```
# @title
#### Section 2: Data Cleaning ####

# Display the original DataFrame and its shape
print("Original Data:")
print(df.head(5).to_string(index=False))
print(f"Number of rows before cleaning: {df.shape[0]}")

# Remove rows with NaN and blank values
df_cleaned = df.dropna().replace('', np.nan).dropna()

# Display the cleaned DataFrame and its shape
print("\nData after removing NaN and blank values:")
print(df_cleaned.head(5).to_string(index=False))
print(f"Number of rows after cleaning: {df_cleaned.shape[0]}")
```

```
Original Data:
    X      Y  COND1  COND2  COND3  COND4  COND5  COND6
52.82 154.26   4.06  18.75  18.59  15.24  13.77  12.04
```

```
53.82 154.26   3.93  17.75  17.79  14.74  13.40  11.80
54.82 154.26   3.80  16.74  16.99  14.23  13.02  11.57
55.82 154.26   3.67  15.74  16.19  13.73  12.65  11.33
56.81 154.26   3.33  14.69  15.32  13.14  12.18  11.02
Number of rows before cleaning: 12865

Data after removing NaN and blank values:
    X      Y  COND1  COND2  COND3  COND4  COND5  COND6
52.82 154.26   4.06  18.75  18.59  15.24  13.77  12.04
53.82 154.26   3.93  17.75  17.79  14.74  13.40  11.80
54.82 154.26   3.80  16.74  16.99  14.23  13.02  11.57
55.82 154.26   3.67  15.74  16.19  13.73  12.65  11.33
56.81 154.26   3.33  14.69  15.32  13.14  12.18  11.02
Number of rows after cleaning: 12865
```

3.4. Section 3

Building on the cleaned dataset from the previous section, we now visualize the data columns through scatter plots.

Each subplot represents a specific data column, showcasing its spatial distribution across the X and Y coordinates. The color intensity in each plot reflects the values of the corresponding data column.

The number of rows and columns for the subplot grid is dynamically calculated based on the available data columns.

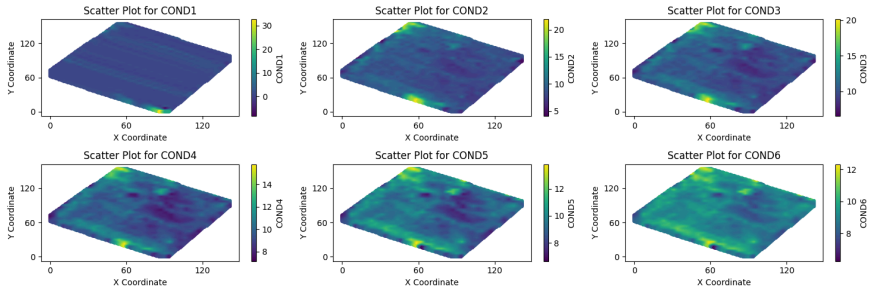
Typically with EMI data acquired by a CMD Mini-Explorer, increase coil separation = increase sampling depth.

```
COND1 = 0.20 m coil separation
COND2 = 0.33 m coil separation
COND3 = 0.50 m coil separation
COND4 = 0.72 m coil separation
COND5 = 1.03 m coil separation
COND6 = 1.50 m coil separation
```

This visualization provides an initial exploration of how different data layers are distributed in geographical space, setting the stage for further analysis and insights.

```
# @title
#### Section 3 Data Viewing ###

# Get the list of non-coordinate column names
data_column_names = df_cleaned.columns[2:]

# Get the list of non-coordinate column names
data_column_names = df_cleaned.columns[2:]

# Calculate the number of rows and columns for subplots
num_plots = len(data_column_names)
num_plots_per_row = 3
num_rows = (num_plots + num_plots_per_row - 1) // num_plots_per_row
num_cols = min(num_plots, num_plots_per_row)

# Create subplots with the specified number of rows and columns
```

```
fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, 5))

# Flatten the axes to simplify indexing
axes = axes.flatten()

# Loop through each data column and create scatter plots
for i, column_name in enumerate(data_column_names):
    # Extract the data for the current column
    column_data = df_cleaned[column_name]

    # Calculate the position in the subplot grid
    row_index = i // num_cols
    col_index = i % num_cols

    # Create a scatter plot
    scatter = axes[i].scatter(df_cleaned['X'], df_cleaned['Y'], c=column_data,
cmap='viridis', marker='o', s=10)

    # Set plot properties
    axes[i].set_title(f'Scatter Plot for {column_name}')
    axes[i].set_xlabel('X Coordinate')
    axes[i].set_ylabel('Y Coordinate')

    # Set the number of tick marks on the X and Y axes
    axes[i].xaxis.set_major_locator(MaxNLocator(nbins=3))
    axes[i].yaxis.set_major_locator(MaxNLocator(nbins=3))

    # Add a color bar scaled to the min and max of the current column
    cbar = plt.colorbar(scatter, ax=axes[i])
    cbar.set_label(column_name)

# Adjust layout for better spacing
plt.tight_layout()

# Show the plots
plt.show()
```



3.5. Section 4

In this section, we strategically divide our dataset into two key components: coordinates and remaining data. The first two columns, containing coordinate information, are isolated to construct the “Coordinates” DataFrame.

Simultaneously, the remaining data, excluding the initial two columns, forms the “Remaining Data” DataFrame.

COND1 is removed in this instance as there is strong acquisition footprint, negative values and high noise content. (See Plots in Section 3)

This separation serves a pivotal purpose—clustering analysis will solely operate on the remaining data. Subsequently, the cluster labels obtained can be associated with their respective X and Y coordinates. This distinction is fundamental for generating geographical cluster maps, allowing us to visually interpret and understand the spatial distribution of clusters across the dataset.

This section also focuses on data normalization, a crucial step before applying clustering algorithms.

The resulting normalized data is displayed, providing an insight into the standardized values across the dataset.

Normalization enhances the accuracy of clustering algorithms, ensuring that features with different scales contribute equally to the clustering process.

```
# @title
#### Section 4 Separate the Data for clustering ####

# Extract coordinate information (assuming it's in the first two columns)
coordinates = df_cleaned.iloc[:, :2]

# Extract the remaining data (excluding the first two columns) (Note COND1 is also removed
here as it is bad data)
remaining_data = df_cleaned.iloc[:, 3:]
remaining_data = remaining_data.round(2)

# Display the Coordinates DataFrame
print("\nCoordinates:")
print(coordinates.head(5).to_string(index=False))  # Use to_string to prevent truncation
print(f"Number of rows: {df_cleaned.shape[0]}")

# Display the Remaining Data DataFrame
print("\nRemaining Data:")
print(remaining_data.head(5).to_string(index=False))  # Use to_string to prevent truncation
print(f"Number of rows: {df_cleaned.shape[0]}")

# Custom normalization using MinMaxScaler
min_vals = remaining_data.min().min()
max_vals = remaining_data.max().max()

normalized_data = (remaining_data - min_vals) / (max_vals - min_vals)

# Round the normalized data to 4 decimal places
normalized_data = normalized_data.round(4)

# Convert the rounded normalized data back to a DataFrame and set column names
normalized_df = pd.DataFrame(normalized_data, columns=remaining_data.columns)

# Display the normalized data
print("\nNormalized Data:")
print(normalized_data.head(5).to_string(index=False))  # Use to_string to prevent truncation
print(f"Number of rows: {df_cleaned.shape[0]}")
```

```
Coordinates:
    X      Y
52.82 154.26
53.82 154.26
54.82 154.26
55.82 154.26
56.81 154.26
Number of rows: 12865

Remaining Data:
 COND2  COND3  COND4  COND5  COND6
 18.75  18.59  15.24  13.77  12.04
 17.75  17.79  14.74  13.40  11.80
 16.74  16.99  14.23  13.02  11.57
 15.74  16.19  13.73  12.65  11.33
 14.69  15.32  13.14  12.18  11.02
Number of rows: 12865

Normalized Data:
 COND2  COND3  COND4  COND5  COND6
0.8233 0.8144 0.6265 0.5440 0.4470
0.7672 0.7695 0.5984 0.5233 0.4335
0.7106 0.7246 0.5698 0.5020 0.4206
0.6545 0.6798 0.5418 0.4812 0.4072
0.5956 0.6310 0.5087 0.4549 0.3898
Number of rows: 12865
```

3.6. Section 5

This section guides the user in determining the optimal number of clusters (k) for the K-Means algorithm by utilizing both the Elbow Method and Silhouette Scores.

After specifying the maximum number of clusters to consider, the code calculates the Within-Cluster Sum of Squares (WCSS) distance using the Elbow Method. The Elbow Method graph illustrates the trade-off between clustering complexity and WCSS reduction, helping identify an optimal k value.

Simultaneously, Silhouette Scores, a measure of how well-separated clusters are, are computed and presented on the same graph. Silhouette Scores range from −1 to 1, where higher scores indicate better-defined clusters.

When assessing the graph, users should look for the “elbow” point where WCSS plateaus, suggesting diminishing returns with additional clusters.

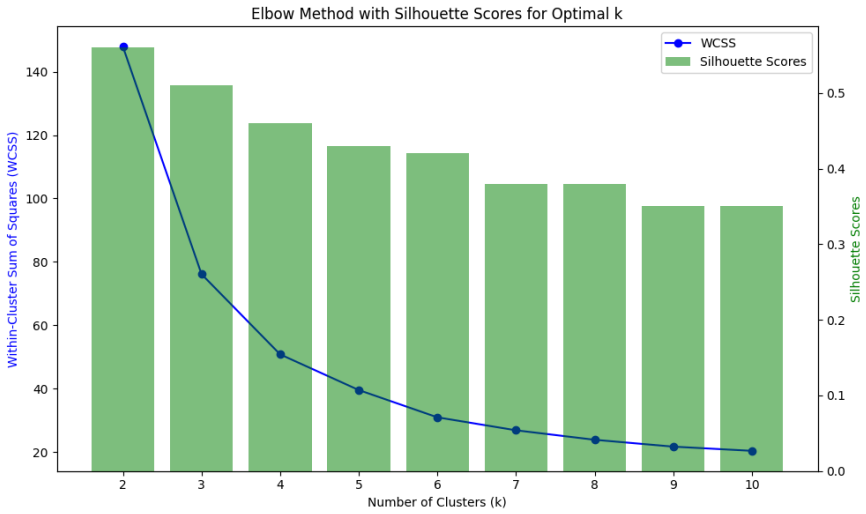
Additionally, a high Silhouette Score at the “elbow” reinforces the choice, ensuring a balance between compact clusters and distinct cluster boundaries for effective clustering.

When running the cell, you will be prompted to enter the max number of clusters. (i.e., 10).

```
# @title
#### Section 6 Elbow Method with Silhouette Scores ####

# Prompt the user to choose the maximum number of clusters for the Elbow Method
max_clusters_elbow = int(input("Enter the maximum number of clusters for the Elbow Method:
"))

# Calculate the within-cluster sum of squares (WCSS) and Silhouette Scores for different
```

```
values of k
wcss = []
silhouette = []
for k in range(2, max_clusters_elbow + 1):
    kmeans = KMeans(n_clusters=k, n_init=1, init='k-means++')
    kmeans.fit(normalized_data)
    sscore = round(silhouette_score(normalized_data, kmeans.labels_), 2)
    silhouette.append(sscore)
    wcss.append(kmeans.inertia_)

# Plot the Elbow Method graph with Silhouette Scores as a bar graph
fig, ax1 = plt.subplots(figsize=(10, 6))

# Plot WCSS on the left y-axis
ax1.plot(range(2, max_clusters_elbow + 1), wcss, marker='o', color='blue', label='WCSS')
ax1.set_xlabel('Number of Clusters (k)')
ax1.set_ylabel('Within-Cluster Sum of Squares (WCSS)', color='blue')

# Set the x-axis ticks to show only integers
plt.xticks(range(2, max_clusters_elbow + 1))

# Create a second y-axis for Silhouette Scores
ax2 = ax1.twinx()
ax2.bar(range(2, max_clusters_elbow + 1), silhouette, color='green', alpha=0.5,
label='Silhouette Scores')
ax2.set_ylabel('Silhouette Scores', color='green')

# Add legend
lines, labels = ax1.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
ax2.legend(lines + lines2, labels + labels2, loc='upper right')

plt.title('Elbow Method with Silhouette Scores for Optimal k')
fig.tight_layout()
plt.show()
```

```
Enter the maximum number of clusters for the Elbow Method: 10
```



3.7. Section 6

In this section, users will perform K-Means clustering on the preprocessed EMI data.

The optimal number of clusters (k) can be determined by referencing the results from the previous Elbow Method and Silhouette Scores analysis (Section 5). After entering the desired number of clusters, the code applies K-Means clustering and displays key information.

The results include a count of occurrences for each cluster label via a plot of the classified image and “spectral” graph showing the cluster centers.

The scatter plot provides an overview of spatial distribution of the clusters, while the line plot illustrates how cluster center values vary across the survey.

You will be prompted to enter the number of clusters, which should be based on the Elbow and Silhouette results, but feel free to run this code a few times for various number of clusters and take a look at the results!

Press the play button to run the code.

Even without knowing the most appropriate number of clusters what does this analysis tell you about the EMI data across this survey?

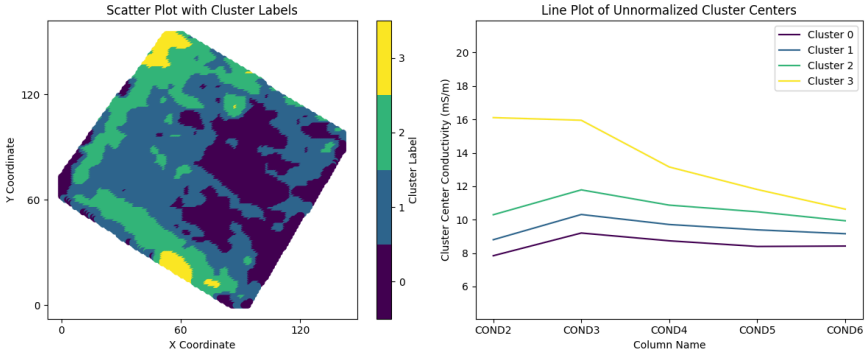
```
# @title
##### Section 6 K Means Clustering ####

### Section 6.1 Perform K Means Clustering ###

# Prompt the user to choose the number of clusters for K-Means
num_clusters = int(input("Enter the number of clusters for K-Means: "))

# Initialize the K-Means model
kmeans = KMeans(n_clusters=num_clusters, init = 'k-means++', n_init=1)

# Fit the K-Means model to the normalized data
kmeans.fit(normalized_data)

# Get the cluster labels for each data point
cluster_labels = kmeans.labels_
```

```
# Get the cluster centers
cluster_centers = kmeans.cluster_centers_

### Section 6.2 Sort the Cluster centers ###

# Calculate the distances of cluster centers from the origin (0, 0)
distances_from_origin = np.sqrt(np.sum(cluster_centers ** 2, axis=1))

# Sort cluster centers based on distances from the origin
sorted_indices = np.argsort(distances_from_origin)

# Sort cluster centers and labels
sorted_cluster_centers = cluster_centers[sorted_indices]
sorted_cluster_labels = np.zeros_like(cluster_labels)

# Relabel the cluster labels based on the sorted order
for new_label, old_label in enumerate(sorted_indices):
    sorted_cluster_labels[cluster_labels == old_label] = new_label

# Calculate the count of each cluster label
cluster_labels_count = dict(zip(*np.unique(sorted_cluster_labels, return_counts=True)))

### Section 6.3 Denormalize the cluster centers ###

# Denormalize the data using the inverse transformation
cluster_centers_original_scale = sorted_cluster_centers * (max_vals - min_vals) + min_vals

### Section 6.4 Display Clustering counts for visual QC ###

# Display the count of each cluster label
print("\nCount of Each Cluster Label:")
for label, count in cluster_labels_count.items():
    print(f"Cluster {label}: {count} occurrences")

### Section 6.5 Save results to CSV ###

# Create a DataFrame with X, Y, Cluster, and Remaining Data
clustered_data_df = pd.DataFrame({
    'X': coordinates['X'],
    'Y': coordinates['Y'],
    'Cluster Number': sorted_cluster_labels,
    **{f'{col}': remaining_data[col] for col in remaining_data.columns}
})
clustered_data_df = clustered_data_df.round(4)

# Create a DataFrame with Cluster center data
center_data_df = pd.DataFrame(cluster_centers_original_scale,
columns=remaining_data.columns)

# Add a new column 'Cluster Number' to indicate the cluster number for each row
center_data_df.insert(0, 'Cluster Number', range(num_clusters))
center_data_df = center_data_df.round(4)

### Section 6.6 Plot KMeans Clustering results ###
```

```
# Create a scatter plot of X, Y, and final cluster label
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Plot 1: Scatter Plot with Cluster Labels
scatter = ax1.scatter(coordinates['X'], coordinates['Y'], c=sorted_cluster_labels,
cmap='viridis', marker='o', s=30)

# Set plot properties for Plot 1
ax1.set_title('Scatter Plot with Cluster Labels')
ax1.set_xlabel('X Coordinate')
ax1.set_ylabel('Y Coordinate')

# Create a discrete color map with the number of clusters for Plot 1
cmap_discrete = matplotlib.colormaps.get_cmap('viridis')

# Define boundaries for the discrete color map for Plot 1
boundaries = np.arange(-0.5, num_clusters, 1)

# Create a BoundaryNorm for the color map for Plot 1
norm_discrete = mcolors.BoundaryNorm(boundaries, cmap_discrete.N, clip=True)

# Add a discrete color bar with integer cluster labels for Plot 1
cbar = plt.colorbar(scatter, ax=ax1, ticks=np.arange(num_clusters), cmap=cmap_discrete,
norm=norm_discrete, boundaries=boundaries)
cbar.set_label('Cluster Label')

# Set the number of tick marks on the X and Y axes for Plot 1
ax1.xaxis.set_major_locator(MaxNLocator(nbins=3))
ax1.yaxis.set_major_locator(MaxNLocator(nbins=3))

# Plot 2: Line Plot of Unnormalized Cluster Centers
for cluster_label in range(num_clusters):
    color = cmap_discrete(cluster_label / (num_clusters - 1))  # Match color from scatter
plot
    ax2.plot(remaining_data.columns, cluster_centers_original_scale[cluster_label],
label=f'Cluster {cluster_label}', color=color)
    ax2.set_ylim(remaining_data.min().min(), remaining_data.max().max())

# Set plot properties for Plot 2
ax2.set_title('Line Plot of Unnormalized Cluster Centers')
ax2.set_xlabel('Column Name')
ax2.set_ylabel('Cluster Center Conductivity (mS/m)')
ax2.legend()

# Show the plots
plt.tight_layout()
plt.show()
```

```
Enter the number of clusters for K-Means: 4

Count of Each Cluster Label:
Cluster 0: 3543 occurrences
Cluster 1: 5980 occurrences
```

```
Cluster 2: 2944 occurrences
Cluster 3: 398 occurrences
```



3.8. Section 7

This section introduces the MCASD (Multiple Cluster Average Standard Deviation) method, designed to aid participants in identifying the optimal number of clusters for their dataset.

MCASD was first published as a method by O'Leary et al 2023.

MCASD evaluates the stability of cluster centers across multiple attempts and cluster numbers. Participants will input the maximum number of clusters and the maximum number of attempts per cluster.

The code loops through different cluster numbers, applying K-Means clustering multiple times to analyze stability.

The results include GIFs illustrating scatter plots and line plots for each attempt. Additionally, MCASD metrics are calculated, providing insights into the stability and consistency of clusters. A line plot visualizes the MCASD metric across various cluster numbers, aiding participants in selecting the optimal cluster count.

All results, including plots and metrics, are compressed into a zip file for easy download.

Please save this ZIP file to "Part 3" of the data directory you were provided with.

The ultimate goal is to assist participants in making informed decisions about the optimal number of clusters for their specific dataset.

When running the cell, you will be prompted to enter the number of max number of clusters (i.e., 10) and max attempts (i.e., 10). Note this might take some time depending on the max number of clusters and max attempts chosen.

```
# @title
#### Section 7 MCASD Method ####
### Section 7.1 Get information from the user ###

# Prompt the user for the maximum number of clusters for MCASD Method
max_num_clusters = int(input("Enter the maximum number of clusters for MCASD Method: "))
# Prompt the user for the maximum number of attempts for MCASD Method
max_attempts = int(input("Enter the maximum number of attempts for MCASD Method: "))

### Section 7.2 Loop for MCASD Method ###
# Create a DataFrame to store MCASD Metrics
mcasd_metrics_df = pd.DataFrame(index=['MCASD Metric', 'MCASD Error'], columns=range(1,
```

```
max_num_clusters + 1))

print(f"\nCalculating MCASD Metrics...")

# Create a zip file to store all results
zip_filename = f'AgroGeo24_WS_Part_3_kmeans_plots.zip'
with ZipFile(zip_filename, 'w') as zip_file:

    # Loop through the various number of clusters
    for num_clusters in range(1, max_num_clusters + 1):
        images_attempt = []  # List to store images for the current attempt
        distances_df = pd.DataFrame()  # Initialize distances DataFrame

        # Cluster the data a user specified number of times (Attempts)
        for attempt in range(1, max_attempts + 1):
            #print(f"\nNumber of Clusters: {num_clusters}: Attempt {attempt} of
{max_attempts}")

            # Initialize the K-Means model
            kmeans = KMeans(n_clusters=num_clusters, n_init=1, init='k-means++')

            # Fit the K-Means model to the normalized data
            kmeans.fit(normalized_data)

            # Get the cluster labels for each data point
            cluster_labels = kmeans.labels_

            # Get the cluster centers
            cluster_centers = kmeans.cluster_centers_

            ### Section 7.2.1 Sort the Cluster centers ###

            # Calculate the distances of cluster centers from the origin (0, 0)
            distances_from_origin = np.sqrt(np.sum(cluster_centers ** 2, axis=1))

            # Sort cluster centers based on distances from the origin
            sorted_indices = np.argsort(distances_from_origin)

            # Sort cluster centers and labels
            sorted_cluster_centers = cluster_centers[sorted_indices]
            sorted_cluster_labels = np.zeros_like(cluster_labels)

            # Relabel the cluster labels based on the sorted order
            for i, new_label in enumerate(np.arange(num_clusters)):
                old_label = sorted_indices[i]
                sorted_cluster_labels[cluster_labels == old_label] = new_label

            ### Section 7.2.2 Calculate the distance (in the dataspace) between each
datapoint and its closest cluster center ###

            # Calculate the distances between cluster centers and data
            distances = np.linalg.norm(normalized_data.values[:, np.newaxis, :] -
cluster_centers, axis=-1)

            # Get the smallest distance for each data point
```

```
            min_distances = np.min(distances, axis=1)

            # Create a DataFrame for distances with only the smallest distances
            new_column = pd.DataFrame(min_distances, columns=[f'Attempt_{attempt}'])

            # Append the new column to the existing distances_df
            distances_df = pd.concat([distances_df, new_column], axis=1)

            ### Section 7.2.3 Denormalize the cluster centers ###

            # Denormalize the data using the inverse transformation
            cluster_centers_original_scale = sorted_cluster_centers * (max_vals - min_vals)
+ min_vals

            ### Section 7.2.4 Create and save plots for later GIF creation ###

            # Create a scatter plot of X, Y, and final cluster label
            fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

            # Plot 1: Scatter Plot with Cluster Labels
            scatter = ax1.scatter(coordinates['X'], coordinates['Y'],
c=sorted_cluster_labels, cmap='viridis',
                                  marker='o', s=30)

            # Set plot properties for Plot 1
            ax1.set_title(f'Scatter Plot with Cluster Labels (Attempt {attempt}, Clusters
{num_clusters})')
            ax1.set_xlabel('X Coordinate')
            ax1.set_ylabel('Y Coordinate')

            # Create a discrete color map with the number of clusters for Plot 1
            cmap_discrete = matplotlib.colormaps.get_cmap('viridis')

            # Define boundaries for the discrete color map for Plot 1
            boundaries = np.arange(-0.5, num_clusters, 1)

            # Create a BoundaryNorm for the color map for Plot 1
            norm_discrete = mcolors.BoundaryNorm(boundaries, cmap_discrete.N, clip=True)

            # Add a discrete color bar with integer cluster labels for Plot 1
            cbar = plt.colorbar(scatter, ax=ax1, ticks=np.arange(num_clusters),
cmap=cmap_discrete, norm=norm_discrete,
                                boundaries=boundaries)
            cbar.set_label('Cluster Label')

            # Set the number of tick marks on the X and Y axes for Plot 1
            ax1.xaxis.set_major_locator(MaxNLocator(nbins=3))
            ax1.yaxis.set_major_locator(MaxNLocator(nbins=3))

            # Plot 2: Line Plot of Unnormalized Cluster Centers
            for cluster_label in range(num_clusters):
                color = cmap_discrete(cluster_label / (num_clusters))  # Match color from
scatter plot
                ax2.plot(remaining_data.columns,
cluster_centers_original_scale[cluster_label],
```

```
                        label=f'Cluster {cluster_label}', color=color)
                ax2.set_ylim(remaining_data.min().min(), remaining_data.max().max())

            # Set plot properties for Plot 2
            ax2.set_title(f'Line Plot of Cluster Centers (Attempt {attempt}, Clusters
{num_clusters})')
            ax2.set_xlabel('Column Name')
            ax2.set_ylabel('Cluster Center Value')
            ax2.legend()

            # Save the plots
            plot_filename = f'kmeans_plots_Attempt_{attempt}
_Num_Clusters_{num_clusters}.png'
            plot_filepath = os.path.join(plot_filename)
            plt.tight_layout()
            plt.savefig(plot_filepath)
            #plt.show()  # Display the plot
            images_attempt.append(plot_filepath)  # Append the plot to the list
            plt.close()

        # Convert the images for the current number of clusters to a GIF
        gif_filename = f'kmeans_plots_Num_Clusters_{num_clusters}.gif'
        with imageio.get_writer(gif_filename, mode='I', fps=1, loop=0) as writer_attempt:
            for image_filename in images_attempt:
                # Adjust the image filename to include the subfolder
                image = imageio.imread(image_filename)
                writer_attempt.append_data(image)

                # Remove individual plot files after adding to GIF
                os.remove(image_filename)

        # Save the GIF to the current cluster folder
        zip_file.write(gif_filename)

        # Remove the GIF file after adding to the zip file
        os.remove(gif_filename)

        ### Section 7.3 Calculate MCASD metrics ###

        # Calculate Standard Deviation along each row
        row_std_dev = distances_df.std(axis=1)

        # Calculate Average of Standard Deviation for all Rows
        avg_std_dev = row_std_dev.mean()

        # Calculate Standard Deviation of the first Standard Deviation for all rows
        error = row_std_dev.std(axis=0)

        # Save values in the mcasd_metrics_df DataFrame
        mcasd_metrics_df.at['MCASD Metric', num_clusters] = avg_std_dev
        mcasd_metrics_df.at['MCASD Error', num_clusters] = error

        ### Section 7.4 Save results for Cluster number to a CSV ###

        # Output file names
```

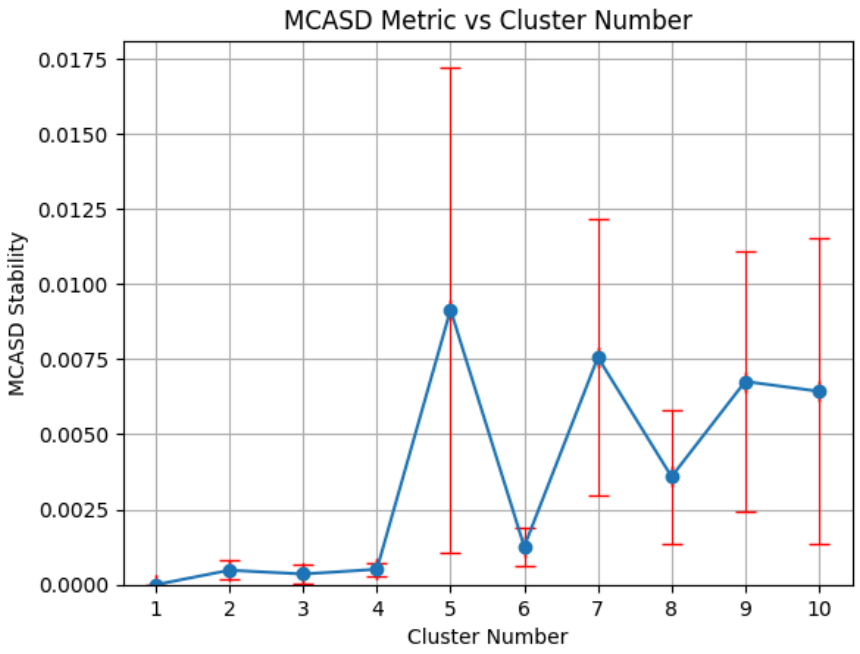
```
        output_cluster_filename = f'{file_name_without_extension}_kmeans_{num_clusters}
_cluster_data.csv'
        output_center_filename = f'{file_name_without_extension}_kmeans_{num_clusters}
_cluster_centers.csv'

        # Create a DataFrame with X, Y, Cluster, and Remaining Data
        clustered_data_df = pd.DataFrame({
            'X': coordinates['X'],
            'Y': coordinates['Y'],
            'Cluster Number': sorted_cluster_labels,
            'MCASD Metric': row_std_dev,
            **{f'{col}': remaining_data[col] for col in remaining_data.columns}
        })
        clustered_data_df = clustered_data_df.round(4)

        # Save the DataFrame to a CSV file
        clustered_data_df.to_csv(output_cluster_filename, index=False)

        # Create a DataFrame with Cluster center data
        center_data_df = pd.DataFrame(cluster_centers_original_scale,
columns=remaining_data.columns)

        # Add a new column 'Cluster Number' to indicate the cluster number for each row
        center_data_df.insert(0, 'Cluster Number', range(num_clusters))
        center_data_df = center_data_df.round(4)

        # Save the DataFrame to a CSV file
        center_data_df.to_csv(output_center_filename, index=False)

        # Save the csv to the current cluster folder
        zip_file.write(output_cluster_filename)
        zip_file.write(output_center_filename)

        # Remove the csv file after adding to the zip file
        os.remove(output_cluster_filename)
        os.remove(output_center_filename)

    # Save mcasd_metrics_df to a CSV file
    mcasd_metrics_csv_filename = 'mcasd_metrics.csv'
    mcasd_metrics_df.to_csv(mcasd_metrics_csv_filename)

    # Add mcasd_metrics CSV file to the zip file
    zip_file.write(mcasd_metrics_csv_filename)

    # Remove the mcasd_metrics CSV file after adding to the zip file
    os.remove(mcasd_metrics_csv_filename)

    ### Section 7.5 Create MCASD metric plot for visual QC ###

    # Make a 2D Line plot
    plt.errorbar(mcasd_metrics_df.columns, mcasd_metrics_df.loc['MCASD Metric'],
                 yerr=mcasd_metrics_df.loc['MCASD Error'], xerr=0, fmt='-o', capsize=5,
ecolor='red', errorevery=1,
                 elinewidth=0.8)
    plt.xlabel('Cluster Number')
```

```
    plt.ylabel('MCASD Stability')
    plt.title('MCASD Metric vs Cluster Number')
    plt.ylim(0)  # Set Y Axis starting at 0
    plt.xticks(np.arange(1, max_num_clusters + 1, 1))  # Set X Tick marks at all integers
    plt.grid(True)

    # Save the 2D line plot to the zip file
    line_plot_filename = 'mcasd_line_plot.png'
    plt.savefig(line_plot_filename)
    zip_file.write(line_plot_filename)
    os.remove(line_plot_filename)  # Remove the saved file after adding to the zip file

# Inform the user that the MCASD Method clustering is complete
print("\nMCASD Method clustering complete.")

### Section 7.6 Save the final zip file to the user's local machine ###

# Move the zip file to the user's local machine
files.download(zip_filename)
```

```
Enter the maximum number of clusters for MCASD Method: 10
Enter the maximum number of attempts for MCASD Method: 10

Calculating MCASD Metrics...

MCASD Method clustering complete.
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```



4. Using Centroid Clustering to compare data from two surveys (Satellite S2 data).

In this part of the workshop you will be provided with data from 2 fly overs of ESA Sentinel 2 Optical Satellite data.

1. The data were acquired in June 2017 and June 2018 over a peatland site in Ireland
2. The data were processed Semi-Automatic Classification plugin in QGIS
3. The data were gridded to a 50 x 50 m grid (See Part 5 for reasoning)
4. Only Pixels that have been identified as being "Peat" are presented
5. All coordinate information has been altered to keep the privacy of the land owner

The aim of this workshop is to determine the appropriate number of clusters for each of the two datasets using KMeans clustering and MCASD and then determine the physical reason behind the temporal difference in the clustering solutions.

The input file name is: `AgroGeo24_WS_Part_4_20170620_Sentinel2.csv` and `AgroGeo24_WS_Part_4_20170628_Sentinel2.csv` and should be located in the folder named "Part 4" of the data directory provided.

Please following along with the workshop leader in the first instance until you are familiar with using Google Colab environment.

No coding experience is required to run this code. All the code contains comments describing what each line does. Please click "Show Code" on any section to view the code.

Please feel free to ask questions if you don't understand any parts.

4.1. Section 0

This section sets up the Python environment for the clustering analysis.

It imports essential libraries such as Pandas for data manipulation, NumPy for numerical operations, Matplotlib for data visualization, scikit-learn for machine learning tools, and other supporting libraries.

Additionally, it configures the display.

The code also imports specific functions and modules required for the clustering analysis, such as KMeans

Finally, it sets up tools for working with images, zip files, and file uploads in Google Colab. This preparation ensures that the subsequent code can efficiently perform clustering analysis and handle related tasks.

```
# @title
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
import imageio.v2 as imageio
import matplotlib
import imageio.v2 as imageio
import os
import re

from sklearn.cluster import KMeans
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import normalize
from sklearn.preprocessing import scale
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_score
from sklearn.metrics import pairwise_distances_argmin_min
from collections import Counter
from sklearn.preprocessing import MinMaxScaler
from matplotlib.ticker import MaxNLocator
from PIL import Image
from zipfile import ZipFile
from google.colab import files


# Set display options to show all rows and columns
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
```

4.2. Section 1

Now we are going to read in the data from your local device. If you have not yet downloaded the necessary data, please do so from here.

The Data are in CSV format with the following header descriptors:

1. Column 1 = X Coordinate
2. Column 2 = Y Coordinate
3. Columns 3..N = Data

Press the play button below to select the correct files. They should be called:
`AgroGeo24_WS_Part_4_20170620_Sentinel2.csv` and `AgroGeo24_WS_Part_4_20180628_Sentinel2.csv`

You will see a button “Choose files”. Click this and navigate to where this file is stored on your machine. Then highlight it and click “open”.

Please upload the 2017 dataset first and then the 2018 dataset.

When running the cell, you will see a progress message as the file is uploaded to this Google Colab environment.

```
# @title
#### Section 1: Import CSV file  ####

# Prompt the user to upload a file
uploaded_1 = files.upload()
uploaded_2 = files.upload()

# Get the uploaded file name
file_name_1 = list(uploaded_1.keys())[0]
file_name_2 = list(uploaded_2.keys())[0]

# Extract the filename without the extension
file_name_without_extension_1 = os.path.splitext(file_name_1)[0]
file_name_without_extension_2 = os.path.splitext(file_name_2)[0]

# Remove numerical suffixes from the filename
file_name_without_extension_1 = re.sub(r'\(\d+\)', '', file_name_without_extension_1)
file_name_without_extension_2 = re.sub(r'\(\d+\)', '', file_name_without_extension_2)
file_name_without_extension_1 = re.sub(r'\(\ \)', '', file_name_without_extension_1)
file_name_without_extension_2 = re.sub(r'\(\ \)', '', file_name_without_extension_2)

# Read the CSV file into a DataFrame
df_1 = pd.read_csv(file_name_1).round(4)
df_2 = pd.read_csv(file_name_2).round(4)
```

Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving AgroGeo24_WS_Part_4_20170620_Sentinel2.csv to
AgroGeo24_WS_Part_4_20170620_Sentinel2.csv
```

Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving AgroGeo24_WS_Part_4_20180628_Sentinel2.csv to
AgroGeo24_WS_Part_4_20180628_Sentinel2.csv
```

Excellent. You have now loaded the CSV file into the Google Colab environment and are ready to take a look at the data.

4.3. Section 2

In this section, we're inspecting and refining the dataset. Initially, we showcase a snippet of the original dataset, including its structure and the number of rows.

Next, we perform data cleaning by eliminating rows containing NaN (Not a Number) and blank values. The cleaned dataset is then displayed, again with a snippet and the updated row count.

This process ensures that the dataset is well-prepared for subsequent analyses by removing any instances of missing or empty data.

Note that this dataset contained no blank values, so the number of rows doesn't change

```
# @title
#### Section 2: Data Cleaning ####

# Display the original DataFrame and its shape
print("File 1: Original Data:")
print(df_1.head(5).to_string(index=False))
print(f"Number of rows before cleaning: {df_1.shape[0]}")

print("\nFile 2: Original Data:")
print(df_2.head(5).to_string(index=False))
print(f"Number of rows before cleaning: {df_2.shape[0]}")

# Remove rows with NaN and blank values
df_cleaned_1 = df_1.dropna().replace('', np.nan).dropna()
df_cleaned_2 = df_2.dropna().replace('', np.nan).dropna()

# Display the cleaned DataFrame and its shape
print("\nFile 1: Data after removing NaN and blank values:")
print(df_cleaned_1.head(5).to_string(index=False))
print(f"Number of rows after cleaning: {df_cleaned_1.shape[0]}")

print("\nFile 2: Data after removing NaN and blank values:")
print(df_cleaned_2.head(5).to_string(index=False))
print(f"Number of rows after cleaning: {df_cleaned_2.shape[0]}")
```

```
File 1: Original Data:
     X      Y     B1     B2     B3     B4     B5     B6     B7     B8    B8A     B9    B10
B11    B12
9750.0 9000.0 0.0307 0.0361 0.0655 0.0467 0.1144 0.3056 0.3908 0.3858 0.4550 0.0959 0.0113
0.2055 0.0984
9800.0 9000.0 0.0304 0.0347 0.0639 0.0449 0.1124 0.3135 0.3988 0.3915 0.4604 0.0946 0.0113
0.2065 0.0962
9850.0 9000.0 0.0293 0.0323 0.0592 0.0405 0.1056 0.3236 0.4176 0.4080 0.4769 0.0945 0.0112
0.1987 0.0897
9900.0 9000.0 0.0290 0.0316 0.0563 0.0389 0.0997 0.3150 0.4117 0.4017 0.4677 0.0915 0.0111
0.1950 0.0898
9550.0 8950.0 0.0295 0.0300 0.0502 0.0378 0.0877 0.2845 0.3799 0.3697 0.4327 0.0884 0.0114
0.1930 0.0877
Number of rows before cleaning: 16794

File 2: Original Data:
     X      Y     B1     B2     B3     B4     B5     B6     B7     B8    B8A     B9    B10
B11    B12
9750.0 9000.0 0.0326 0.0475 0.0705 0.1039 0.1995 0.1995 0.2407 0.2498 0.2980 0.0828 0.0110
0.3392 0.2006
9800.0 9000.0 0.0332 0.0501 0.0757 0.1095 0.2201 0.2201 0.2653 0.2745 0.3260 0.0869 0.0110
0.3396 0.1994
9850.0 9000.0 0.0321 0.0488 0.0739 0.1065 0.2216 0.2216 0.2694 0.2798 0.3351 0.0894 0.0109
0.3455 0.2005
9900.0 9000.0 0.0303 0.0457 0.0690 0.0953 0.2199 0.2199 0.2687 0.2783 0.3306 0.0876 0.0109
0.3244 0.1878
9550.0 8950.0 0.0218 0.0234 0.0393 0.0405 0.2283 0.2283 0.3043 0.3067 0.3560 0.0903 0.0109
0.1932 0.0936
Number of rows before cleaning: 16794
```

```
File 1: Data after removing NaN and blank values:
     X      Y     B1     B2     B3     B4     B5     B6     B7     B8    B8A     B9    B10
B11    B12
9750.0 9000.0 0.0307 0.0361 0.0655 0.0467 0.1144 0.3056 0.3908 0.3858 0.4550 0.0959 0.0113
0.2055 0.0984
9800.0 9000.0 0.0304 0.0347 0.0639 0.0449 0.1124 0.3135 0.3988 0.3915 0.4604 0.0946 0.0113
0.2065 0.0962
9850.0 9000.0 0.0293 0.0323 0.0592 0.0405 0.1056 0.3236 0.4176 0.4080 0.4769 0.0945 0.0112
0.1987 0.0897
9900.0 9000.0 0.0290 0.0316 0.0563 0.0389 0.0997 0.3150 0.4117 0.4017 0.4677 0.0915 0.0111
0.1950 0.0898
9550.0 8950.0 0.0295 0.0300 0.0502 0.0378 0.0877 0.2845 0.3799 0.3697 0.4327 0.0884 0.0114
0.1930 0.0877
Number of rows after cleaning: 16793

File 2: Data after removing NaN and blank values:
     X      Y     B1     B2     B3     B4     B5     B6     B7     B8    B8A     B9    B10
B11    B12
9750.0 9000.0 0.0326 0.0475 0.0705 0.1039 0.1995 0.1995 0.2407 0.2498 0.2980 0.0828 0.0110
0.3392 0.2006
9800.0 9000.0 0.0332 0.0501 0.0757 0.1095 0.2201 0.2201 0.2653 0.2745 0.3260 0.0869 0.0110
0.3396 0.1994
9850.0 9000.0 0.0321 0.0488 0.0739 0.1065 0.2216 0.2216 0.2694 0.2798 0.3351 0.0894 0.0109
0.3455 0.2005
9900.0 9000.0 0.0303 0.0457 0.0690 0.0953 0.2199 0.2199 0.2687 0.2783 0.3306 0.0876 0.0109
0.3244 0.1878
9550.0 8950.0 0.0218 0.0234 0.0393 0.0405 0.2283 0.2283 0.3043 0.3067 0.3560 0.0903 0.0109
0.1932 0.0936
Number of rows after cleaning: 16793
```

4.4. Section 3

Building on the cleaned dataset from the previous section, we now visualize the non-coordinate columns through scatter plots for both inputs.

Data from File 1 (2017 S2) is placed alongside data from File 2 (2018 S2)

For information on each band see this link.

Each subplot represents a specific data column, showcasing its spatial distribution across the X and Y coordinates. The color intensity in each plot reflects the values of the corresponding data column.

The number of rows and columns for the subplot grid is dynamically calculated based on the available data columns.

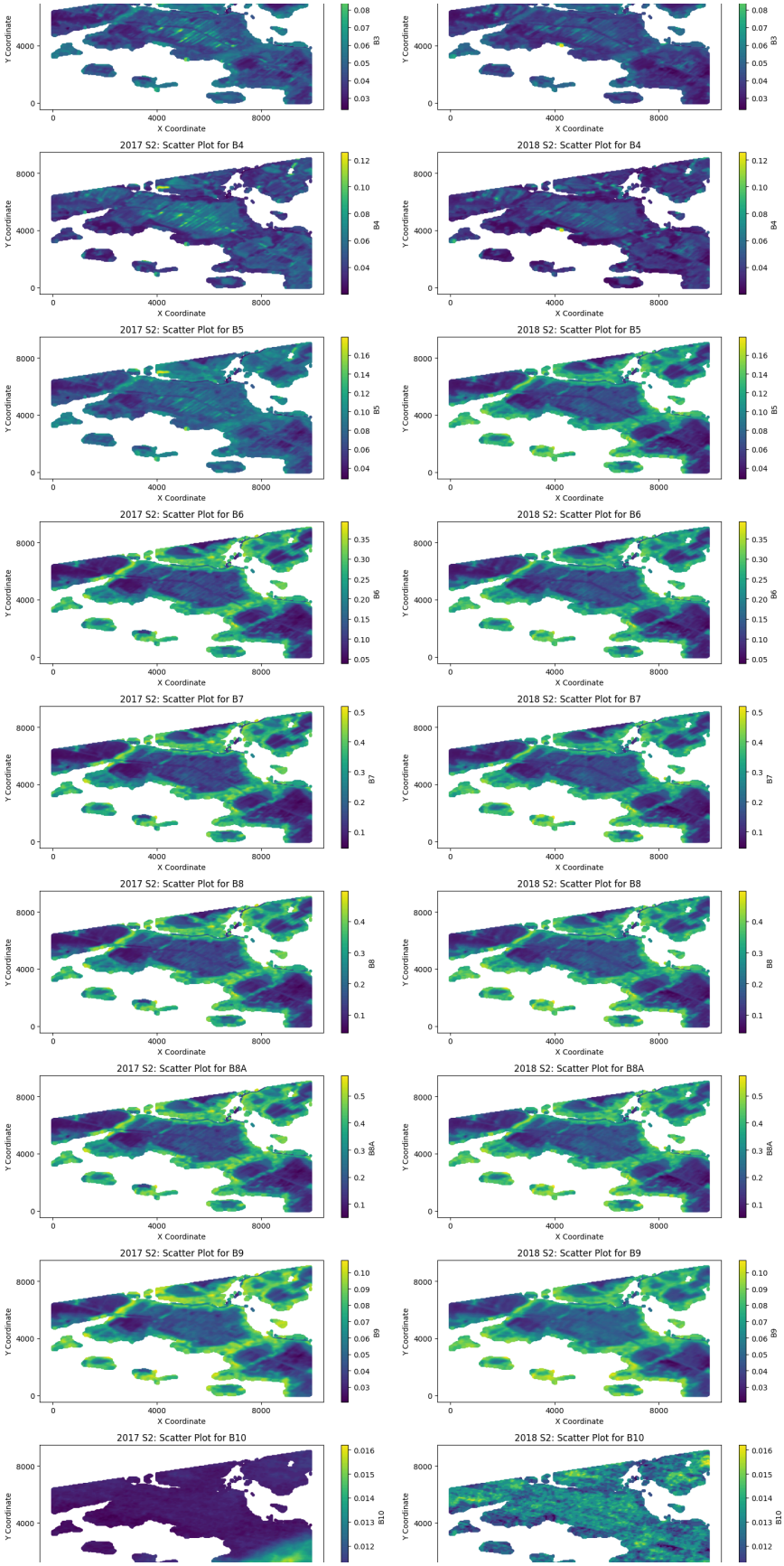
This visualization provides an initial exploration of how different data variables are distributed in the geographical space, setting the stage for further analysis and insights.

Note that only pixels that have been identified as being part of a peatland are included in this analysis (See accompanying presentation).

```
# @title
#### Section 3 Data Viewing ###

# Assuming df_cleaned_1 and df_cleaned_2 have the same number of columns
```

```
data_column_names_1 = df_cleaned_1.columns[2:]
data_column_names_2 = df_cleaned_2.columns[2:]

# Calculate the number of rows and columns for subplots
num_plots = len(data_column_names_1) + len(data_column_names_2)
num_plots_per_row = 2
num_rows = (num_plots + num_plots_per_row - 1) // num_plots_per_row
num_cols = min(num_plots_per_row, 2)

# Create subplots with the specified number of rows and columns
fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, 45))

# Flatten the axes to simplify indexing
axes = axes.flatten()

# Loop through each data column and create scatter plots
for i, (column_name_1, column_name_2) in enumerate(zip(data_column_names_1,
data_column_names_2)):
    # Extract the data for the current column
    column_data_1 = df_cleaned_1[column_name_1]
    column_data_2 = df_cleaned_2[column_name_2]

    # Calculate the position in the subplot grid
    row_index = i // num_cols
    col_index = i % num_cols

    # Create scatter plots for each input
    scatter_1 = axes[i * 2].scatter(df_cleaned_1['X'], df_cleaned_1['Y'], c=column_data_1,
cmap='viridis', marker='o', s=10)
    scatter_2 = axes[i * 2 + 1].scatter(df_cleaned_2['X'], df_cleaned_2['Y'],
c=column_data_2, cmap='viridis', marker='o', s=10)

    # Set plot properties for the first column
    axes[i * 2].set_title(f'2017 S2: Scatter Plot for {column_name_1}')
    axes[i * 2].set_xlabel('X Coordinate')
    axes[i * 2].set_ylabel('Y Coordinate')

    # Set the number of tick marks on the X and Y axes
    axes[i * 2].xaxis.set_major_locator(MaxNLocator(nbins=3))
    axes[i * 2].yaxis.set_major_locator(MaxNLocator(nbins=3))

    # Add a color bar scaled to the min and max of the current columns
    cbar_1 = plt.colorbar(scatter_1, ax=axes[i * 2], orientation='vertical')
    cbar_1.set_label(column_name_1)

    # Set plot properties for the second column
    axes[i * 2 + 1].set_title(f'2018 S2: Scatter Plot for {column_name_2}')
    axes[i * 2 + 1].set_xlabel('X Coordinate')
    axes[i * 2 + 1].set_ylabel('Y Coordinate')

    # Set the number of tick marks on the X and Y axes
    axes[i * 2 + 1].xaxis.set_major_locator(MaxNLocator(nbins=3))
    axes[i * 2 + 1].yaxis.set_major_locator(MaxNLocator(nbins=3))

    # Add a color bar scaled to the min and max of the current columns
```

```
    cbar_2 = plt.colorbar(scatter_1, ax=axes[i * 2 + 1], orientation='vertical')
    cbar_2.set_label(column_name_2)

# Adjust layout for better spacing
plt.tight_layout()

# Show the plots
plt.show()
```


2017 S2: Scatter Plot for B3
2018 S2: Scatter Plot for B3
2017 S2: Scatter Plot for B4
2018 S2: Scatter Plot for B4
2017 S2: Scatter Plot for B5
2018 S2: Scatter Plot for B5
2017 S2: Scatter Plot for B6
2018 S2: Scatter Plot for B6
2017 S2: Scatter Plot for B7
2018 S2: Scatter Plot for B7
2017 S2: Scatter Plot for B8
2018 S2: Scatter Plot for B8
2017 S2: Scatter Plot for B8A
2018 S2: Scatter Plot for B8A
2017 S2: Scatter Plot for B9
2018 S2: Scatter Plot for B9
2017 S2: Scatter Plot for B10
2018 S2: Scatter Plot for B10
X Coordinate
Y Coordinate

4.5. Section 4

In this section, we strategically divide our dataset into two key components: coordinates and remaining data. The first two columns, containing coordinate information, are isolated to construct the “Coordinates” DataFrame.

Simultaneously, the remaining data, excluding the initial two columns, forms the “Remaining Data” DataFrame.

B10 is removed in this instance as it is a measurement of cloud. As these images were acquired on cloud free days, this layer is considered as noise and skews the clustering results

This separation serves a pivotal purpose—clustering analysis will solely operate on the remaining data. Subsequently, the cluster labels obtained can be associated with their respective X and Y coordinates. This distinction is fundamental for generating geographical cluster maps, allowing us to visually interpret and understand the spatial distribution of clusters across the dataset.

This section also focuses on data normalization, a crucial step before applying clustering algorithms.

The resulting normalized data is displayed, providing an insight into the standardized values across the dataset.

Normalization enhances the accuracy of clustering algorithms, ensuring that features with different scales contribute equally to the clustering process.

```
# @title
#### Section 4 Separate the Data for clustering ####

# Extract coordinate information (assuming it's in the first two columns)
coordinates_1 = df_cleaned_1.iloc[:, :2]
coordinates_2 = df_cleaned_2.iloc[:, :2]

# Extract the remaining data (excluding the first two columns)
remaining_data_1 = df_cleaned_1.iloc[:, 2:]
remaining_data_1 = remaining_data_1.drop(columns = ['B10'])  # Assuming rows are zero-
indexed
remaining_data_1 = remaining_data_1.round(4)
remaining_data_2 = df_cleaned_2.iloc[:, 2:]
remaining_data_2 = remaining_data_2.drop(columns = ['B10'])  # Assuming rows are zero-
indexed
remaining_data_2 = remaining_data_2.round(4)

# Display the Coordinates DataFrame
print("\n2017 S2 Coordinates:")
print(coordinates_1.head(3).to_string(index=False))  # Use to_string to prevent truncation
print(f"Number of rows: {df_cleaned_1.shape[0]}")

print("\n2018 S2 Coordinates:")
print(coordinates_2.head(3).to_string(index=False))  # Use to_string to prevent truncation
print(f"Number of rows: {df_cleaned_2.shape[0]}")

# Display the Remaining Data DataFrame
print("\n2017 S2 Remaining Data:")
print(remaining_data_1.head(3).to_string(index=False))  # Use to_string to prevent
truncation
print(f"Number of rows: {df_cleaned_1.shape[0]}")
```

```
print("\n2018 S2 Remaining Data:")
print(remaining_data_2.head(3).to_string(index=False))  # Use to_string to prevent
truncation
print(f"Number of rows: {df_cleaned_2.shape[0]}")

# Custom normalization using MinMaxScaler
min_vals_1 = remaining_data_1.min()
max_vals_1 = remaining_data_1.max()
min_vals_2 = remaining_data_2.min()
max_vals_2 = remaining_data_2.max()

normalized_data_1 = (remaining_data_1 - min_vals_1.values) / (max_vals_1.values -
min_vals_1.values)
normalized_data_2 = (remaining_data_2 - min_vals_2.values) / (max_vals_2.values -
min_vals_2.values)

# Round the normalized data to 4 decimal places
#normalized_data_1 = normalized_data_1.round(4)
#normalized_data_2 = normalized_data_2.round(4)

# Convert the rounded normalized data back to a DataFrame and set column names
normalized_df_1 = pd.DataFrame(normalized_data_1, columns=remaining_data_1.columns)
normalized_df_2 = pd.DataFrame(normalized_data_2, columns=remaining_data_2.columns)


# Display the normalized data
print("\n2017 S2 Normalized Data:")
print(normalized_data_1.head(3).to_string(index=False))  # Use to_string to prevent
truncation
print(f"Number of rows: {df_cleaned_1.shape[0]}")

print("\n2018 S2 Normalized Data:")
print(normalized_data_2.head(3).to_string(index=False))  # Use to_string to prevent
truncation
print(f"Number of rows: {df_cleaned_2.shape[0]}")
```

```
2017 S2 Coordinates:
     X      Y
9750.0 9000.0
9800.0 9000.0
9850.0 9000.0
Number of rows: 16793

2018 S2 Coordinates:
     X      Y
9750.0 9000.0
9800.0 9000.0
9850.0 9000.0
Number of rows: 16793

2017 S2 Remaining Data:
    B1     B2     B3     B4     B5     B6     B7     B8    B8A     B9    B11    B12
0.0307 0.0361 0.0655 0.0467 0.1144 0.3056 0.3908 0.3858 0.4550 0.0959 0.2055 0.0984
```

```
0.0304 0.0347 0.0639 0.0449 0.1124 0.3135 0.3988 0.3915 0.4604 0.0946 0.2065 0.0962
0.0293 0.0323 0.0592 0.0405 0.1056 0.3236 0.4176 0.4080 0.4769 0.0945 0.1987 0.0897
Number of rows: 16793

2018 S2 Remaining Data:
    B1     B2     B3     B4     B5     B6     B7     B8    B8A     B9    B11    B12
0.0326 0.0475 0.0705 0.1039 0.1995 0.1995 0.2407 0.2498 0.2980 0.0828 0.3392 0.2006
0.0332 0.0501 0.0757 0.1095 0.2201 0.2201 0.2653 0.2745 0.3260 0.0869 0.3396 0.1994
0.0321 0.0488 0.0739 0.1065 0.2216 0.2216 0.2694 0.2798 0.3351 0.0894 0.3455 0.2005
Number of rows: 16793

2017 S2 Normalized Data:
      B1       B2       B3       B4       B5       B6       B7       B8      B8A       B9
B11      B12
0.224839 0.239003 0.522444 0.252372 0.570100 0.753894 0.731299 0.754738 0.771953 0.868056
0.413927 0.261135
0.218415 0.218475 0.502494 0.235294 0.556811 0.776267 0.748252 0.767298 0.782284 0.853009
0.416516 0.253233
0.194861 0.183284 0.443890 0.193548 0.511628 0.804871 0.788091 0.803658 0.813851 0.851852
0.396324 0.229885
Number of rows: 16793

2018 S2 Normalized Data:
      B1       B2       B3       B4       B5       B6       B7       B8      B8A       B9
B11      B12
0.466357 0.533742 0.550296 0.648485 0.486540 0.486540 0.459672 0.492162 0.516486 0.581960
0.775362 0.614420
0.480278 0.573620 0.601578 0.690909 0.544914 0.544914 0.513443 0.546699 0.570791 0.617520
0.776362 0.610240
0.454756 0.553681 0.583826 0.668182 0.549164 0.549164 0.522404 0.558401 0.588441 0.639202
0.791104 0.614072
Number of rows: 16793
```

4.6. Section 5

This section guides the user in determining the optimal number of clusters (k) for the K-Means algorithm by utilizing both the Elbow Method and Silhouette Scores.

After specifying the maximum number of clusters to consider, the code calculates the Within-Cluster Sum of Squares (WCSS) distance using the Elbow Method. The Elbow Method graph illustrates the trade-off between clustering complexity and WCSS reduction, helping identify an optimal k value.

Simultaneously, Silhouette Scores, a measure of how well-separated clusters are, are computed and presented on the same graph. Silhouette Scores range from −1 to 1, where higher scores indicate better-defined clusters.

When assessing the graph, users should look for the “elbow” point where WCSS plateaus, suggesting diminishing returns with additional clusters.

Additionally, a high Silhouette Score at the “elbow” reinforces the choice, ensuring a balance between compact clusters and distinct cluster boundaries for effective clustering

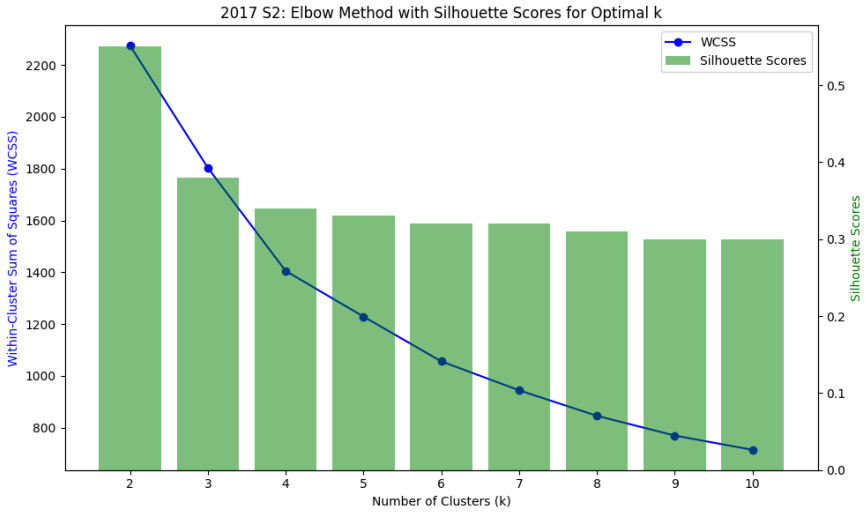
When running the cell, you will be prompted to enter the max number of clusters. (i.e., 10).

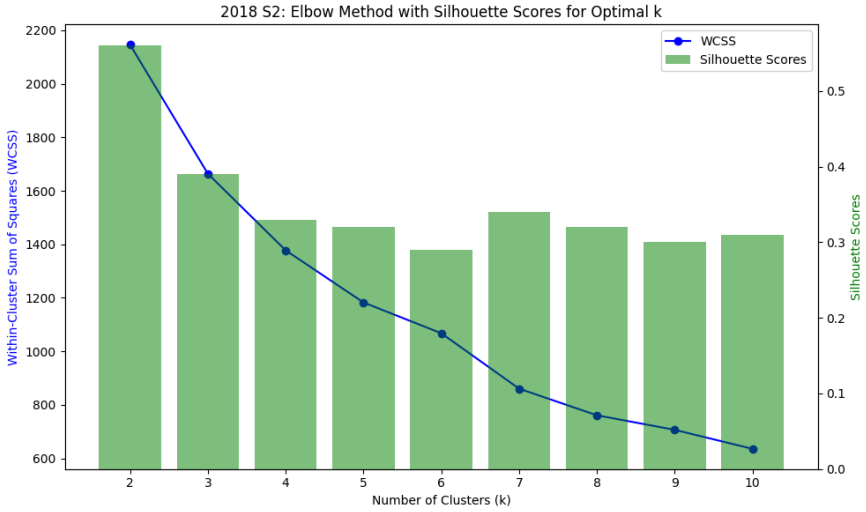
```
# @title
#### Section 5 Elbow Method with Silhouette Scores ####
```

```
# Prompt the user to choose the maximum number of clusters for the Elbow Method
max_clusters_elbow = int(input("Enter the maximum number of clusters for the Elbow Method:
"))

# Calculate the within-cluster sum of squares (WCSS) and Silhouette Scores for different
values of k
wcss_1 = []
silhouette_1 = []
wcss_2 = []
silhouette_2 = []
for k in range(2, max_clusters_elbow + 1):
    kmeans_1 = KMeans(n_clusters=k, n_init=1, init='k-means++')
    kmeans_2 = KMeans(n_clusters=k, n_init=1, init='k-means++')
    kmeans_1.fit(normalized_data_1)
    kmeans_2.fit(normalized_data_2)
    sscore_1 = round(silhouette_score(normalized_data_1, kmeans_1.labels_), 2)
    sscore_2 = round(silhouette_score(normalized_data_2, kmeans_2.labels_), 2)
    silhouette_1.append(sscore_1)
    silhouette_2.append(sscore_2)
    wcss_1.append(kmeans_1.inertia_)
    wcss_2.append(kmeans_2.inertia_)

# Plot the Elbow Method graph with Silhouette Scores as a bar graph for 2017 S2
fig, ax1 = plt.subplots(figsize=(10, 6))

# Plot WCSS on the left y-axis
ax1.plot(range(2, max_clusters_elbow + 1), wcss_1, marker='o', color='blue', label='WCSS')
ax1.set_xlabel('Number of Clusters (k)')
ax1.set_ylabel('Within-Cluster Sum of Squares (WCSS)', color='blue')

# Set the x-axis ticks to show only integers
plt.xticks(range(2, max_clusters_elbow + 1))

# Create a second y-axis for Silhouette Scores
ax2 = ax1.twinx()
ax2.bar(range(2, max_clusters_elbow + 1), silhouette_1, color='green', alpha=0.5,
label='Silhouette Scores')
ax2.set_ylabel('Silhouette Scores', color='green')

# Add legend
lines, labels = ax1.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
ax2.legend(lines + lines2, labels + labels2, loc='upper right')

plt.title('2017 S2: Elbow Method with Silhouette Scores for Optimal k')
fig.tight_layout()
plt.show()

# Plot the Elbow Method graph with Silhouette Scores as a bar graph for 2018 S2
fig, ax1 = plt.subplots(figsize=(10, 6))

# Plot WCSS on the left y-axis
ax1.plot(range(2, max_clusters_elbow + 1), wcss_2, marker='o', color='blue', label='WCSS')
ax1.set_xlabel('Number of Clusters (k)')
ax1.set_ylabel('Within-Cluster Sum of Squares (WCSS)', color='blue')
```

```
# Set the x-axis ticks to show only integers
plt.xticks(range(2, max_clusters_elbow + 1))

# Create a second y-axis for Silhouette Scores
ax2 = ax1.twinx()
ax2.bar(range(2, max_clusters_elbow + 1), silhouette_2, color='green', alpha=0.5,
label='Silhouette Scores')
ax2.set_ylabel('Silhouette Scores', color='green')

# Add legend
lines, labels = ax1.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
ax2.legend(lines + lines2, labels + labels2, loc='upper right')

plt.title('2018 S2: Elbow Method with Silhouette Scores for Optimal k')
fig.tight_layout()
plt.show()
```

```
Enter the maximum number of clusters for the Elbow Method: 10
```





4.7. Section 6

In this section, users will perform K-Means clustering on the preprocessed Satellite data.

The optimal number of clusters (k) can be determined by referencing the results from the previous Elbow Method and Silhouette Scores analysis (Section 5). After entering the desired number of clusters, the code applies K-Means clustering and displays key information.

The results include a count of occurrences for each cluster label via a plot of the classified image and “spectral” graph showing the cluster centers.

The scatter plot provides an overview of spatial distribution of the clusters, while the line plot illustrates how cluster center values vary across the survey

You will be prompted to enter the number of clusters for both 2017 and 2018 S2 datasets, which should be based on the Elbow and Silhouette results.

Even without knowing the most appropriate number of clusters what does this analysis tell you about the Satellite data across this survey and also differences in data with time?

```
# @title
##### Section 6 K Means Clustering ####

### Section 6.1 Perform K Means Clustering ###

# Prompt the user to choose the number of clusters for K-Means
num_clusters_1 = int(input("Enter the number of clusters for 2017 S2 K-Means: "))
num_clusters_2 = int(input("Enter the number of clusters for 2018 S2 K-Means: "))

# Initialize the K-Means model
kmeans_1 = KMeans(n_clusters=num_clusters_1, init = 'k-means++', n_init=1)
kmeans_2 = KMeans(n_clusters=num_clusters_2, init = 'k-means++', n_init=1)

# Fit the K-Means model to the normalized data
kmeans_1.fit(normalized_data_1)
kmeans_2.fit(normalized_data_2)

# Get the cluster labels for each data point
cluster_labels_1 = kmeans_1.labels_
```

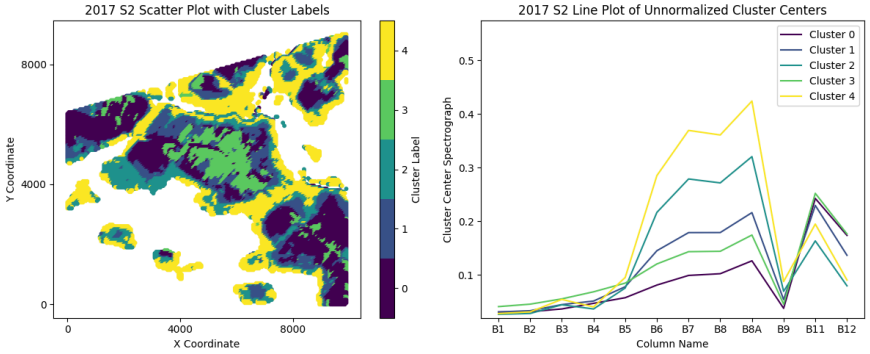
```
cluster_labels_2 = kmeans_2.labels_

# Get the cluster centers
cluster_centers_1 = kmeans_1.cluster_centers_
cluster_centers_2 = kmeans_2.cluster_centers_

### Section 6.2 Sort the Cluster centers ###

# Calculate the distances of cluster centers from the origin (0, 0)
distances_from_origin_1 = np.sqrt(np.sum(cluster_centers_1 ** 2, axis=1))
distances_from_origin_2 = np.sqrt(np.sum(cluster_centers_2 ** 2, axis=1))

# Sort cluster centers based on distances from the origin
sorted_indices_1 = np.argsort(distances_from_origin_1)
sorted_indices_2 = np.argsort(distances_from_origin_2)

# Sort cluster centers and labels
sorted_cluster_centers_1 = cluster_centers_1[sorted_indices_1]
sorted_cluster_centers_2 = cluster_centers_2[sorted_indices_2]
sorted_cluster_labels_1 = np.zeros_like(cluster_labels_1)
sorted_cluster_labels_2 = np.zeros_like(cluster_labels_2)

# Relabel the cluster labels based on the sorted order
for new_label_1, old_label_1 in enumerate(sorted_indices_1):
    sorted_cluster_labels_1[cluster_labels_1 == old_label_1] = new_label_1

for new_label_2, old_label_2 in enumerate(sorted_indices_2):
    sorted_cluster_labels_2[cluster_labels_2 == old_label_2] = new_label_2

# Calculate the count of each cluster label
cluster_labels_count_1 = dict(zip(*np.unique(sorted_cluster_labels_1, return_counts=True)))
cluster_labels_count_2 = dict(zip(*np.unique(sorted_cluster_labels_2, return_counts=True)))

### Section 6.3 Denormalize the cluster centers ###

# Denormalize the data using the inverse transformation
cluster_centers_original_scale_1 = sorted_cluster_centers_1 * (max_vals_1.values -
min_vals_1.values) + min_vals_1.values
cluster_centers_original_scale_2 = sorted_cluster_centers_2 * (max_vals_2.values -
min_vals_2.values) + min_vals_2.values

### Section 6.4 Display Clustering counts for visual QC ###

# Display the count of each cluster label
print("\n2017 S2 Count of Each Cluster Label:")
for label_1, count_1 in cluster_labels_count_1.items():
    print(f"Cluster {label_1}: {count_1} occurrences")

print("\n2018 S2 Count of Each Cluster Label:")
for label_2, count_2 in cluster_labels_count_2.items():
    print(f"Cluster {label_2}: {count_2} occurrences")

### Section 6.5 Save results to CSV ###

# Create a DataFrame with X, Y, Cluster, and Remaining Data
```

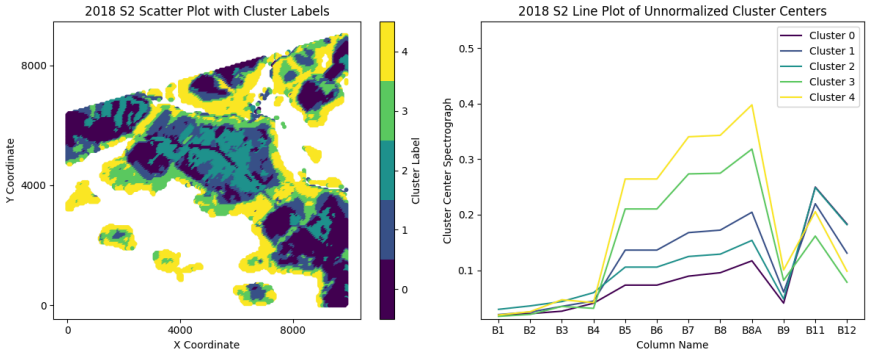
```
clustered_data_df_1 = pd.DataFrame({
    'X': coordinates_1['X'],
    'Y': coordinates_1['Y'],
    'Cluster Number': sorted_cluster_labels_1,
    **{f'{col}': remaining_data_1[col] for col in remaining_data_1.columns}
})
clustered_data_df_2 = pd.DataFrame({
    'X': coordinates_2['X'],
    'Y': coordinates_2['Y'],
    'Cluster Number': sorted_cluster_labels_2,
    **{f'{col}': remaining_data_2[col] for col in remaining_data_2.columns}
})
clustered_data_df_1 = clustered_data_df_1.round(4)
clustered_data_df_2 = clustered_data_df_2.round(4)

# Create a DataFrame with Cluster center data
center_data_df_1 = pd.DataFrame(cluster_centers_original_scale_1,
columns=remaining_data_1.columns)
center_data_df_2 = pd.DataFrame(cluster_centers_original_scale_2,
columns=remaining_data_2.columns)

# Add a new column 'Cluster Number' to indicate the cluster number for each row
center_data_df_1.insert(0, 'Cluster Number', range(num_clusters_1))
center_data_df_2.insert(0, 'Cluster Number', range(num_clusters_2))
center_data_df_1 = center_data_df_1.round(4)
center_data_df_2 = center_data_df_2.round(4)

### Section 6.6 Plot KMeans Clustering results ###

# 2017 S2 Create a scatter plot of X, Y, and final cluster label
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Plot 1: Scatter Plot with Cluster Labels
scatter_1 = ax1.scatter(coordinates_1['X'], coordinates_1['Y'], c=sorted_cluster_labels_1,
cmap='viridis', marker='o', s=10)

# Set plot properties for Plot 1
ax1.set_title('2017 S2 Scatter Plot with Cluster Labels')
ax1.set_xlabel('X Coordinate')
ax1.set_ylabel('Y Coordinate')

# Create a discrete color map with the number of clusters for Plot 1
cmap_discrete_1 = matplotlib.colormaps.get_cmap('viridis')

# Define boundaries for the discrete color map for Plot 1
boundaries_1 = np.arange(-0.5, num_clusters_1, 1)

# Create a BoundaryNorm for the color map for Plot 1
norm_discrete_1 = mcolors.BoundaryNorm(boundaries_1, cmap_discrete_1.N, clip=True)

# Add a discrete color bar with integer cluster labels for Plot 1
cbar_1 = plt.colorbar(scatter_1, ax=ax1, ticks=np.arange(num_clusters_1),
cmap=cmap_discrete_1, norm=norm_discrete_1, boundaries=boundaries_1)
cbar_1.set_label('Cluster Label')
```

```
# Set the number of tick marks on the X and Y axes for Plot 1
ax1.xaxis.set_major_locator(MaxNLocator(nbins=3))
ax1.yaxis.set_major_locator(MaxNLocator(nbins=3))

# Plot 2: Line Plot of Unnormalized Cluster Centers
for cluster_label_1 in range(num_clusters_1):
    color_1 = cmap_discrete_1(cluster_label_1 / (num_clusters_1 - 1))  # Match color from
scatter plot
    ax2.plot(remaining_data_1.columns, cluster_centers_original_scale_1[cluster_label_1],
label=f'Cluster {cluster_label_1}', color=color_1)
    ax2.set_ylim(remaining_data_1.min().min(), remaining_data_1.max().max())

# Set plot properties for Plot 2
ax2.set_title('2017 S2 Line Plot of Unnormalized Cluster Centers')
ax2.set_xlabel('Column Name')
ax2.set_ylabel('Cluster Center Spectrograph')
ax2.legend()

# Show the plots
plt.tight_layout()
plt.show()

# 2018 S2 Create a scatter plot of X, Y, and final cluster label
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Plot 1: Scatter Plot with Cluster Labels
scatter_2 = ax1.scatter(coordinates_2['X'], coordinates_2['Y'], c=sorted_cluster_labels_2,
cmap='viridis', marker='o', s=10)

# Set plot properties for Plot 1
ax1.set_title('2018 S2 Scatter Plot with Cluster Labels')
ax1.set_xlabel('X Coordinate')
ax1.set_ylabel('Y Coordinate')

# Create a discrete color map with the number of clusters for Plot 1
cmap_discrete_2 = matplotlib.colormaps.get_cmap('viridis')

# Define boundaries for the discrete color map for Plot 1
boundaries_2 = np.arange(-0.5, num_clusters_2, 1)

# Create a BoundaryNorm for the color map for Plot 1
norm_discrete_2 = mcolors.BoundaryNorm(boundaries_2, cmap_discrete_2.N, clip=True)

# Add a discrete color bar with integer cluster labels for Plot 1
cbar_2 = plt.colorbar(scatter_2, ax=ax1, ticks=np.arange(num_clusters_2),
cmap=cmap_discrete_2, norm=norm_discrete_2, boundaries=boundaries_2)
cbar_2.set_label('Cluster Label')

# Set the number of tick marks on the X and Y axes for Plot 1
ax1.xaxis.set_major_locator(MaxNLocator(nbins=3))
ax1.yaxis.set_major_locator(MaxNLocator(nbins=3))

# Plot 2: Line Plot of Unnormalized Cluster Centers
for cluster_label_2 in range(num_clusters_2):
    color_2 = cmap_discrete_2(cluster_label_2 / (num_clusters_2 - 1))  # Match color from
```

```
scatter plot
    ax2.plot(remaining_data_2.columns, cluster_centers_original_scale_2[cluster_label_2],
label=f'Cluster {cluster_label_2}', color=color_2)
    ax2.set_ylim(remaining_data_2.min().min(), remaining_data_2.max().max())

# Set plot properties for Plot 2
ax2.set_title('2018 S2 Line Plot of Unnormalized Cluster Centers')
ax2.set_xlabel('Column Name')
ax2.set_ylabel('Cluster Center Spectrograph')
ax2.legend()

# Show the plots
plt.tight_layout()
plt.show()
```

```
Enter the number of clusters for 2017 S2 K-Means: 5
Enter the number of clusters for 2018 S2 K-Means: 5

2017 S2 Count of Each Cluster Label:
Cluster 0: 4629 occurrences
Cluster 1: 3269 occurrences
Cluster 2: 2877 occurrences
Cluster 3: 1900 occurrences
Cluster 4: 4118 occurrences

2018 S2 Count of Each Cluster Label:
Cluster 0: 4221 occurrences
Cluster 1: 3524 occurrences
Cluster 2: 2309 occurrences
Cluster 3: 3006 occurrences
Cluster 4: 3733 occurrences
```





4.8. Section 7

This section applies the MCASD (Multiple Cluster Average Standard Deviation) method, designed to aid participants in identifying the optimal number of clusters for each of the datasets.

MCASD was first published as a method by O'Leary et al 2023.

MCASD evaluates the stability of cluster centers across multiple attempts and cluster numbers. Participants will input the maximum number of clusters and the maximum number of attempts per cluster.

The code loops through different cluster numbers, applying K-Means clustering multiple times to analyze stability.

The results include GIFs illustrating scatter plots and line plots for each attempt. Additionally, MCASD metrics are calculated, providing insights into the stability and consistency of clusters. A line plot visualizes the MCASD metric across various cluster numbers, aiding participants in selecting the optimal cluster count.

All results, including plots and metrics, are compressed into a zip file for easy download.

Please save this ZIP file to "Part 4" of the data directory you were provided with.

The ultimate goal is to assist participants in making informed decisions about the optimal number of clusters for their specific datasets.

When running the cell, you will be prompted to enter the number of max number of clusters (i.e., 10) and max attempts (i.e., 10).

MCASD will be applied to both 2017 and 2018 S2 datasets separately and MCASD plots will be displayed at the end.

Note this might take some time depending on the max number of clusters and max attempts chosen.

```
# @title
#### Section 7 MCASD Method ####
### Section 7.1 Get information from the user ###

# Prompt the user for the maximum number of clusters for MCASD Method
max_num_clusters = int(input("Enter the maximum number of clusters for MCASD Method: "))
# Prompt the user for the maximum number of attempts for MCASD Method
max_attempts = int(input("Enter the maximum number of attempts for MCASD Method: "))

### Section 7.2 Loop for MCASD Method ###
# Create a DataFrame to store MCASD Metrics
mcasd_metrics_df_1 = pd.DataFrame(index=['MCASD Metric', 'MCASD Error'], columns=range(1,
max_num_clusters + 1))
```

```
mcasd_metrics_df_2 = pd.DataFrame(index=['MCASD Metric', 'MCASD Error'], columns=range(1,
max_num_clusters + 1))

print(f"\nCalculating MCASD Metrics...")

# Create a zip file to store all results
zip_filename = f'AgroGeo24_WS_Part_4_kmeans_plots.zip'
with ZipFile(zip_filename, 'w') as zip_file:

    # Loop through the various number of clusters
    for num_clusters in range(2, max_num_clusters + 1):
        images_attempt_1 = []  # List to store images for the current attempt
        images_attempt_2 = []  # List to store images for the current attempt
        distances_df_1 = pd.DataFrame()  # Initialize distances DataFrame
        distances_df_2 = pd.DataFrame()  # Initialize distances DataFrame

        # Cluster the data a user specified number of times (Attempts)
        for attempt in range(1, max_attempts + 1):
            #print(f"\nNumber of Clusters: {num_clusters}: Attempt {attempt} of
{max_attempts}")

            # Initialize the K-Means model
            kmeans_1 = KMeans(n_clusters=num_clusters, n_init=1, init='k-means++')
            kmeans_2 = KMeans(n_clusters=num_clusters, n_init=1, init='k-means++')

            # Fit the K-Means model to the normalized data
            kmeans_1.fit(normalized_data_1)
            kmeans_2.fit(normalized_data_2)

            # Get the cluster labels for each data point
            cluster_labels_1 = kmeans_1.labels_
            cluster_labels_2 = kmeans_2.labels_

            # Get the cluster centers
            cluster_centers_1 = kmeans_1.cluster_centers_
            cluster_centers_2 = kmeans_2.cluster_centers_

            ### Section 7.2.1 Sort the Cluster centers ###

            # Calculate the distances of cluster centers from the origin (0, 0)
            distances_from_origin_1 = np.sqrt(np.sum(cluster_centers_1 ** 2, axis=1))
            distances_from_origin_2 = np.sqrt(np.sum(cluster_centers_2 ** 2, axis=1))

            # Sort cluster centers based on distances from the origin
            sorted_indices_1 = np.argsort(distances_from_origin_1)
            sorted_indices_2 = np.argsort(distances_from_origin_2)

            # Sort cluster centers and labels
            sorted_cluster_centers_1 = cluster_centers_1[sorted_indices_1]
            sorted_cluster_centers_2 = cluster_centers_2[sorted_indices_2]
            sorted_cluster_labels_1 = np.zeros_like(cluster_labels_1)
            sorted_cluster_labels_2 = np.zeros_like(cluster_labels_2)

            # Relabel the cluster labels based on the sorted order
            for new_label_1, old_label_1 in enumerate(sorted_indices_1):
```

```
                sorted_cluster_labels_1[cluster_labels_1 == old_label_1] = new_label_1

            for new_label_2, old_label_2 in enumerate(sorted_indices_2):
                sorted_cluster_labels_2[cluster_labels_2 == old_label_2] = new_label_2

            ### Section 7.2.2 Calculate the distance (in the dataspace) between each
datapoint and its closest cluster center ###

            # Calculate the distances between cluster centers and data
            distances_1 = np.linalg.norm(normalized_data_1.values[:, np.newaxis, :] -
cluster_centers_1, axis=-1)
            distances_2 = np.linalg.norm(normalized_data_2.values[:, np.newaxis, :] -
cluster_centers_2, axis=-1)

            # Get the smallest distance for each data point
            min_distances_1 = np.min(distances_1, axis=1)
            min_distances_2 = np.min(distances_2, axis=1)

            # Create a DataFrame for distances with only the smallest distances
            new_column_1 = pd.DataFrame(min_distances_1, columns=[f'Attempt_{attempt}'])
            new_column_2 = pd.DataFrame(min_distances_2, columns=[f'Attempt_{attempt}'])

            # Append the new column to the existing distances_df
            distances_df_1 = pd.concat([distances_df_1, new_column_1], axis=1)
            distances_df_2 = pd.concat([distances_df_2, new_column_2], axis=1)

            ### Section 7.2.3 Denormalize the cluster centers ###

            # Denormalize the data using the inverse transformation
            cluster_centers_original_scale_1 = sorted_cluster_centers_1 * (max_vals_1.values
- min_vals_1.values) + min_vals_1.values
            cluster_centers_original_scale_2 = sorted_cluster_centers_2 * (max_vals_2.values
- min_vals_2.values) + min_vals_2.values

            ### Section 7.2.4 Create and save plots for later GIF creation ###

            # 2017 S2 Create a scatter plot of X, Y, and final cluster label
            fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

            # Plot 1: Scatter Plot with Cluster Labels
            scatter_1 = ax1.scatter(coordinates_1['X'], coordinates_1['Y'],
c=sorted_cluster_labels_1, cmap='viridis', marker='o', s=10)

            # Set plot properties for Plot 1
            ax1.set_title(f'2017 S2 Scatter Plot with Cluster Labels: Attempt {attempt},
Clusters {num_clusters}')
            ax1.set_xlabel('X Coordinate')
            ax1.set_ylabel('Y Coordinate')

            # Create a discrete color map with the number of clusters for Plot 1
            cmap_discrete_1 = matplotlib.colormaps.get_cmap('viridis')

            # Define boundaries for the discrete color map for Plot 1
            boundaries_1 = np.arange(-0.5, num_clusters_1, 1)
```

```
            # Create a BoundaryNorm for the color map for Plot 1
            norm_discrete_1 = mcolors.BoundaryNorm(boundaries_1, cmap_discrete_1.N,
clip=True)

            # Add a discrete color bar with integer cluster labels for Plot 1
            cbar_1 = plt.colorbar(scatter_1, ax=ax1, ticks=np.arange(num_clusters_1),
cmap=cmap_discrete_1, norm=norm_discrete_1, boundaries=boundaries_1)
            cbar_1.set_label('Cluster Label')

            # Set the number of tick marks on the X and Y axes for Plot 1
            ax1.xaxis.set_major_locator(MaxNLocator(nbins=3))
            ax1.yaxis.set_major_locator(MaxNLocator(nbins=3))

            # Plot 2: Line Plot of Unnormalized Cluster Centers
            for cluster_label_1 in range(num_clusters):
                color_1 = cmap_discrete_1(cluster_label_1 / (num_clusters - 1))  # Match
color from scatter plot
                ax2.plot(remaining_data_1.columns,
cluster_centers_original_scale_1[cluster_label_1], label=f'Cluster {cluster_label_1}',
color=color_1)
                ax2.set_ylim(remaining_data_1.min().min(), remaining_data_1.max().max())

            # Set plot properties for Plot 2
            ax2.set_title('2017 S2 Line Plot of Unnormalized Cluster Centers')
            ax2.set_xlabel('Column Name')
            ax2.set_ylabel('Cluster Center Spectrograph')
            ax2.legend()

            # Save the plots
            plot_filename_1 = f'2017 S2 kmeans_plots_Attempt_{attempt}
_Num_Clusters_{num_clusters}.png'
            plot_filepath_1 = os.path.join(plot_filename_1)
            plt.tight_layout()
            plt.savefig(plot_filepath_1)
            #plt.show()  # Display the plot
            images_attempt_1.append(plot_filepath_1)  # Append the plot to the list
            plt.close()

            # 2018 S2 Create a scatter plot of X, Y, and final cluster label
            fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

            # Plot 1: Scatter Plot with Cluster Labels
            scatter_2 = ax1.scatter(coordinates_2['X'], coordinates_2['Y'],
c=sorted_cluster_labels_2, cmap='viridis', marker='o', s=10)

            # Set plot properties for Plot 1
            ax1.set_title(f'2018 S2 Scatter Plot with Cluster Labels: Attempt {attempt},
Clusters {num_clusters}')
            ax1.set_xlabel('X Coordinate')
            ax1.set_ylabel('Y Coordinate')

            # Create a discrete color map with the number of clusters for Plot 1
            cmap_discrete_2 = matplotlib.colormaps.get_cmap('viridis')

            # Define boundaries for the discrete color map for Plot 1
```

```
            boundaries_2 = np.arange(-0.5, num_clusters_2, 1)

            # Create a BoundaryNorm for the color map for Plot 1
            norm_discrete_2 = mcolors.BoundaryNorm(boundaries_2, cmap_discrete_2.N,
clip=True)

            # Add a discrete color bar with integer cluster labels for Plot 1
            cbar_2 = plt.colorbar(scatter_2, ax=ax1, ticks=np.arange(num_clusters_2),
cmap=cmap_discrete_2, norm=norm_discrete_2, boundaries=boundaries_2)
            cbar_2.set_label('Cluster Label')

            # Set the number of tick marks on the X and Y axes for Plot 1
            ax1.xaxis.set_major_locator(MaxNLocator(nbins=3))
            ax1.yaxis.set_major_locator(MaxNLocator(nbins=3))

            # Plot 2: Line Plot of Unnormalized Cluster Centers
            for cluster_label_2 in range(num_clusters):
                color_2 = cmap_discrete_2(cluster_label_2 / (num_clusters - 1))  # Match
color from scatter plot
                ax2.plot(remaining_data_2.columns,
cluster_centers_original_scale_2[cluster_label_2], label=f'Cluster {cluster_label_2}',
color=color_2)
                ax2.set_ylim(remaining_data_2.min().min(), remaining_data_2.max().max())

            # Set plot properties for Plot 2
            ax2.set_title('2018 S2 Line Plot of Unnormalized Cluster Centers')
            ax2.set_xlabel('Column Name')
            ax2.set_ylabel('Cluster Center Spectrograph')
            ax2.legend()

            # Save the plots
            plot_filename_2 = f'2018 S2 kmeans_plots_Attempt_{attempt}
_Num_Clusters_{num_clusters}.png'
            plot_filepath_2 = os.path.join(plot_filename_2)
            plt.tight_layout()
            plt.savefig(plot_filepath_2)
            #plt.show()  # Display the plot
            images_attempt_2.append(plot_filepath_2)  # Append the plot to the list
            plt.close()

         # Convert the images for the current number of clusters to a GIF
        gif_filename_1 = f'2017 S2 kmeans_plots_Num_Clusters_{num_clusters}.gif'
        with imageio.get_writer(gif_filename_1, mode='I', fps=1, loop=0) as writer_attempt:
            for image_filename_1 in images_attempt_1:
                # Adjust the image filename to include the subfolder
                image = imageio.imread(image_filename_1)
                writer_attempt.append_data(image)

                # Remove individual plot files after adding to GIF
                os.remove(image_filename_1)

        # Save the GIF to the current cluster folder
        zip_file.write(gif_filename_1)

        # Remove the GIF file after adding to the zip file
```

```
        os.remove(gif_filename_1)

         # Convert the images for the current number of clusters to a GIF
        gif_filename_2 = f'2018 S2 kmeans_plots_Num_Clusters_{num_clusters}.gif'
        with imageio.get_writer(gif_filename_2, mode='I', fps=1, loop=0) as writer_attempt:
            for image_filename_2 in images_attempt_2:
                # Adjust the image filename to include the subfolder
                image = imageio.imread(image_filename_2)
                writer_attempt.append_data(image)

                # Remove individual plot files after adding to GIF
                os.remove(image_filename_2)

        # Save the GIF to the current cluster folder
        zip_file.write(gif_filename_2)

        # Remove the GIF file after adding to the zip file
        os.remove(gif_filename_2)

        ### Section 7.3 Calculate MCASD metrics ###

        # Calculate Standard Deviation along each row
        row_std_dev_1 = distances_df_1.std(axis=1)
        row_std_dev_2 = distances_df_2.std(axis=1)

        # Calculate Average of Standard Deviation for all Rows
        avg_std_dev_1 = row_std_dev_1.mean()
        avg_std_dev_2 = row_std_dev_2.mean()

        # Calculate Standard Deviation of the first Standard Deviation for all rows
        error_1 = row_std_dev_1.std(axis=0)
        error_2 = row_std_dev_2.std(axis=0)

        # Save values in the mcasd_metrics_df DataFrame
        mcasd_metrics_df_1.at['MCASD Metric', num_clusters] = avg_std_dev_1
        mcasd_metrics_df_2.at['MCASD Metric', num_clusters] = avg_std_dev_2
        mcasd_metrics_df_1.at['MCASD Error', num_clusters] = error_1
        mcasd_metrics_df_2.at['MCASD Error', num_clusters] = error_2

        ### Section 7.4 Save results for Cluster number to a CSV ###

        # Output file names
        output_cluster_filename_1 = f'{file_name_without_extension_1}_kmeans_{num_clusters}
_cluster_data.csv'
        output_cluster_filename_2 = f'{file_name_without_extension_2}_kmeans_{num_clusters}
_cluster_data.csv'
        output_center_filename_1 = f'{file_name_without_extension_1}_kmeans_{num_clusters}
_cluster_centers.csv'
        output_center_filename_2 = f'{file_name_without_extension_2}_kmeans_{num_clusters}
_cluster_centers.csv'

        # Create a DataFrame with X, Y, Cluster, and Remaining Data
        clustered_data_df_1 = pd.DataFrame({
            'X': coordinates_1['X'],
            'Y': coordinates_1['Y'],
```

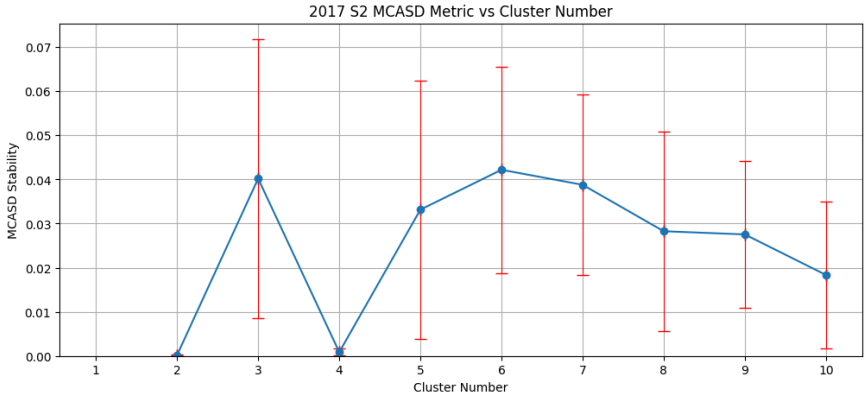
```
            'Cluster Number': sorted_cluster_labels_1,
            'MCASD Metric': row_std_dev_1,
            **{f'{col}': remaining_data_1[col] for col in remaining_data_1.columns}
        })

        clustered_data_df_2 = pd.DataFrame({
            'X': coordinates_2['X'],
            'Y': coordinates_2['Y'],
            'Cluster Number': sorted_cluster_labels_2,
            'MCASD Metric': row_std_dev_2,
            **{f'{col}': remaining_data_2[col] for col in remaining_data_2.columns}
        })
        clustered_data_df_1 = clustered_data_df_1.round(4)
        clustered_data_df_2 = clustered_data_df_2.round(4)

        # Save the DataFrame to a CSV file
        clustered_data_df_1.to_csv(output_cluster_filename_1, index=False)
        clustered_data_df_2.to_csv(output_cluster_filename_2, index=False)

        # Create a DataFrame with Cluster center data
        center_data_df_1 = pd.DataFrame(cluster_centers_original_scale_1,
columns=remaining_data_1.columns)
        center_data_df_2 = pd.DataFrame(cluster_centers_original_scale_2,
columns=remaining_data_2.columns)

        # Add a new column 'Cluster Number' to indicate the cluster number for each row
        center_data_df_1.insert(0, 'Cluster Number', range(num_clusters))
        center_data_df_2.insert(0, 'Cluster Number', range(num_clusters))
        center_data_df_1 = center_data_df_1.round(4)
        center_data_df_2 = center_data_df_2.round(4)

        # Save the DataFrame to a CSV file
        center_data_df_1.to_csv(output_center_filename_1, index=False)
        center_data_df_2.to_csv(output_center_filename_2, index=False)

        # Save the csv to the current cluster folder
        zip_file.write(output_cluster_filename_1)
        zip_file.write(output_cluster_filename_2)
        zip_file.write(output_center_filename_1)
        zip_file.write(output_center_filename_2)

        # Remove the csv file after adding to the zip file
        os.remove(output_cluster_filename_1)
        os.remove(output_cluster_filename_2)
        os.remove(output_center_filename_1)
        os.remove(output_center_filename_2)

    # Save mcasd_metrics_df to a CSV file
    mcasd_metrics_csv_filename_1 = '2017_S2_mcasd_metrics.csv'
    mcasd_metrics_csv_filename_2 = '2018_S2_mcasd_metrics.csv'
    mcasd_metrics_df_1.to_csv(mcasd_metrics_csv_filename_1)
    mcasd_metrics_df_2.to_csv(mcasd_metrics_csv_filename_2)

    # Add mcasd_metrics CSV file to the zip file
    zip_file.write(mcasd_metrics_csv_filename_1)
```

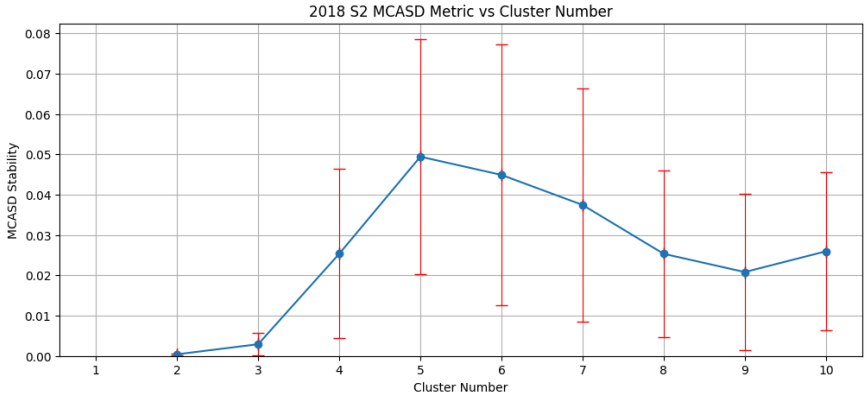
```
    zip_file.write(mcasd_metrics_csv_filename_2)

    # Remove the mcasd_metrics CSV file after adding to the zip file
    os.remove(mcasd_metrics_csv_filename_1)
    os.remove(mcasd_metrics_csv_filename_2)

    ### Section 7.5 Create MCASD metric plot for visual QC ###
    fig = plt.subplots(1, 1, figsize=(12, 5))
    # Make a 2D Line plot for 2017 S2 Data
    plt.errorbar(mcasd_metrics_df_1.columns, mcasd_metrics_df_1.loc['MCASD Metric'],
                 yerr=mcasd_metrics_df_1.loc['MCASD Error'], xerr=0, fmt='-o', capsize=5,
ecolor='red', errorevery=1,
                 elinewidth=0.8)
    plt.xlabel('Cluster Number')
    plt.ylabel('MCASD Stability')
    plt.title('2017 S2 MCASD Metric vs Cluster Number')
    plt.ylim(0)  # Set Y Axis starting at 0
    plt.xticks(np.arange(1, max_num_clusters + 1, 1))  # Set X Tick marks at all integers
    plt.grid(True)

    # Save the 2D line plot to the zip file
    line_plot_filename_1 = '2017_S2_mcasd_line_plot.png'
    plt.savefig(line_plot_filename_1)
    zip_file.write(line_plot_filename_1)
    os.remove(line_plot_filename_1)  # Remove the saved file after adding to the zip file

    fig = plt.subplots(1, 1, figsize=(12, 5))
    # Make a 2D Line plot for 2018 S2 Data
    plt.errorbar(mcasd_metrics_df_2.columns, mcasd_metrics_df_2.loc['MCASD Metric'],
                 yerr=mcasd_metrics_df_2.loc['MCASD Error'], xerr=0, fmt='-o', capsize=5,
ecolor='red', errorevery=1,
                 elinewidth=0.8)
    plt.xlabel('Cluster Number')
    plt.ylabel('MCASD Stability')
    plt.title('2018 S2 MCASD Metric vs Cluster Number')
    plt.ylim(0)  # Set Y Axis starting at 0
    plt.xticks(np.arange(1, max_num_clusters + 1, 1))  # Set X Tick marks at all integers
    plt.grid(True)

    # Save the 2D line plot to the zip file
    line_plot_filename_2 = '2018_S2_mcasd_line_plot.png'
    plt.savefig(line_plot_filename_2)
    zip_file.write(line_plot_filename_2)
    os.remove(line_plot_filename_2)  # Remove the saved file after adding to the zip file

# Inform the user that the MCASD Method clustering is complete
print("\nMCASD Method clustering complete.")

### Section 7.6 Save the final zip file to the user's local machine ###

# Move the zip file to the user's local machine
files.download(zip_filename)
```

```
Enter the maximum number of clusters for MCASD Method: 10
Enter the maximum number of attempts for MCASD Method: 10

Calculating MCASD Metrics...

MCASD Method clustering complete.
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```





4.9. Section 8

In this section, users will perform K-Means clustering on the preprocessed Satellite data.

The optimal number of clusters (k) are now known by referencing the results from the previous MCASD analysis (Section 7). After entering the desired number of clusters, the code applies K-Means clustering and displays key information.

The results include a count of occurrences for each cluster label via a plot of the classified image and “spectral” graph showing the cluster centers.

The scatter plot provides an overview of spatial distribution of the clusters, while the line plot illustrates how cluster center values vary across the survey

In this instance the Spectral plots are overlain in order to allow for combined interpretation.

The task for this section is to think about the physical reason for the differences in the clustering results.

Hint: Pay particular attention to the Spectral plot and what each band is measuring, and how these differ from year to year. What might cause the differences from one year to the next??

When running the cell, you will be prompted to enter the number of clusters for both 2017 and 2018 S2 datasets.

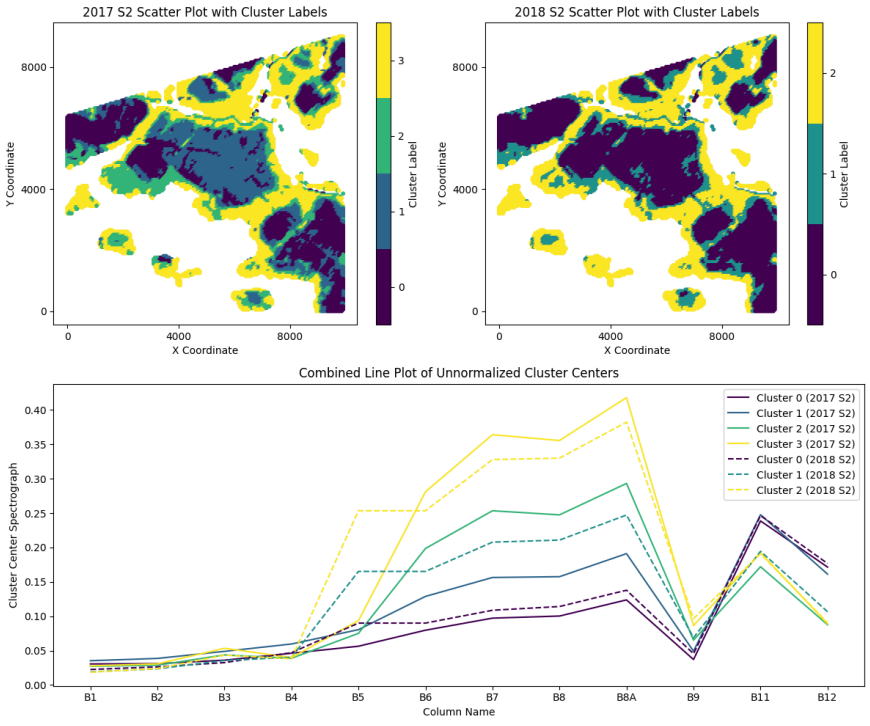
```
# @title
##### Section 8 K Means Clustering ####

### Section 8.1 Perform K Means Clustering ###

# Prompt the user to choose the number of clusters for K-Means
num_clusters_1 = int(input("Enter the number of clusters for 2017 S2 K-Means: "))
num_clusters_2 = int(input("Enter the number of clusters for 2018 S2 K-Means: "))

# Initialize the K-Means model
kmeans_1 = KMeans(n_clusters=num_clusters_1, init = 'k-means++', n_init=1)
kmeans_2 = KMeans(n_clusters=num_clusters_2, init = 'k-means++', n_init=1)

# Fit the K-Means model to the normalized data
kmeans_1.fit(normalized_data_1)
kmeans_2.fit(normalized_data_2)

# Get the cluster labels for each data point
cluster_labels_1 = kmeans_1.labels_
cluster_labels_2 = kmeans_2.labels_

# Get the cluster centers
cluster_centers_1 = kmeans_1.cluster_centers_
cluster_centers_2 = kmeans_2.cluster_centers_

### Section 8.2 Sort the Cluster centers ###

# Calculate the distances of cluster centers from the origin (0, 0)
distances_from_origin_1 = np.sqrt(np.sum(cluster_centers_1 ** 2, axis=1))
distances_from_origin_2 = np.sqrt(np.sum(cluster_centers_2 ** 2, axis=1))

# Sort cluster centers based on distances from the origin
sorted_indices_1 = np.argsort(distances_from_origin_1)
sorted_indices_2 = np.argsort(distances_from_origin_2)

# Sort cluster centers and labels
sorted_cluster_centers_1 = cluster_centers_1[sorted_indices_1]
sorted_cluster_centers_2 = cluster_centers_2[sorted_indices_2]
sorted_cluster_labels_1 = np.zeros_like(cluster_labels_1)
sorted_cluster_labels_2 = np.zeros_like(cluster_labels_2)

# Relabel the cluster labels based on the sorted order
for new_label_1, old_label_1 in enumerate(sorted_indices_1):
    sorted_cluster_labels_1[cluster_labels_1 == old_label_1] = new_label_1

for new_label_2, old_label_2 in enumerate(sorted_indices_2):
    sorted_cluster_labels_2[cluster_labels_2 == old_label_2] = new_label_2
```

```
# Calculate the count of each cluster label
cluster_labels_count_1 = dict(zip(*np.unique(sorted_cluster_labels_1, return_counts=True)))
cluster_labels_count_2 = dict(zip(*np.unique(sorted_cluster_labels_2, return_counts=True)))

### Section 8.3 Denormalize the cluster centers ###

# Denormalize the data using the inverse transformation
cluster_centers_original_scale_1 = sorted_cluster_centers_1 * (max_vals_1.values -
min_vals_1.values) + min_vals_1.values
cluster_centers_original_scale_2 = sorted_cluster_centers_2 * (max_vals_2.values -
min_vals_2.values) + min_vals_2.values

### Section 8.4 Save results to CSV ###

# Output file names

# Create a DataFrame with X, Y, Cluster, and Remaining Data
clustered_data_df_1 = pd.DataFrame({
    'X': coordinates_1['X'],
    'Y': coordinates_1['Y'],
    'Cluster Number': sorted_cluster_labels_1,
    **{f'{col}': remaining_data_1[col] for col in remaining_data_1.columns}
})
clustered_data_df_2 = pd.DataFrame({
    'X': coordinates_2['X'],
    'Y': coordinates_2['Y'],
    'Cluster Number': sorted_cluster_labels_2,
    **{f'{col}': remaining_data_2[col] for col in remaining_data_2.columns}
})
clustered_data_df_1 = clustered_data_df_1.round(4)
clustered_data_df_2 = clustered_data_df_2.round(4)

# Create a DataFrame with Cluster center data
center_data_df_1 = pd.DataFrame(cluster_centers_original_scale_1,
columns=remaining_data_1.columns)
center_data_df_2 = pd.DataFrame(cluster_centers_original_scale_2,
columns=remaining_data_2.columns)

# Add a new column 'Cluster Number' to indicate the cluster number for each row
center_data_df_1.insert(0, 'Cluster Number', range(num_clusters_1))
center_data_df_2.insert(0, 'Cluster Number', range(num_clusters_2))
center_data_df_1 = center_data_df_1.round(4)
center_data_df_2 = center_data_df_2.round(4)

### Section 8.5 Plot KMeans Clustering results ###

# Create a single figure with three subplots
fig = plt.figure(figsize=(12, 10))

gs = fig.add_gridspec(2,2)
ax1 = fig.add_subplot(gs[0, 0])
ax2 = fig.add_subplot(gs[0, 1])
ax3 = fig.add_subplot(gs[1, :])
```

```
# Plot 1: Scatter Plot with Cluster Labels (2017 S2)
scatter_1 = ax1.scatter(coordinates_1['X'], coordinates_1['Y'], c=sorted_cluster_labels_1,
cmap='viridis', marker='o', s=10)

# Set plot properties for Plot 1
ax1.set_title('2017 S2 Scatter Plot with Cluster Labels')
ax1.set_xlabel('X Coordinate')
ax1.set_ylabel('Y Coordinate')

# Create a discrete color map with the number of clusters for Plot 1
cmap_discrete_1 = matplotlib.colormaps.get_cmap('viridis')

# Define boundaries for the discrete color map for Plot 1
boundaries_1 = np.arange(-0.5, num_clusters_1, 1)

# Create a BoundaryNorm for the color map for Plot 1
norm_discrete_1 = mcolors.BoundaryNorm(boundaries_1, cmap_discrete_1.N, clip=True)

# Add a discrete color bar with integer cluster labels for Plot 1
cbar_1 = plt.colorbar(scatter_1, ax=ax1, ticks=np.arange(num_clusters_1),
cmap=cmap_discrete_1, norm=norm_discrete_1, boundaries=boundaries_1)
cbar_1.set_label('Cluster Label')

# Set the number of tick marks on the X and Y axes for Plot 1
ax1.xaxis.set_major_locator(MaxNLocator(nbins=3))
ax1.yaxis.set_major_locator(MaxNLocator(nbins=3))

# Plot 2: Scatter Plot with Cluster Labels (2018 S2)
scatter_2 = ax2.scatter(coordinates_2['X'], coordinates_2['Y'], c=sorted_cluster_labels_2,
cmap='viridis', marker='o', s=10)

# Set plot properties for Plot 2
ax2.set_title('2018 S2 Scatter Plot with Cluster Labels')
ax2.set_xlabel('X Coordinate')
ax2.set_ylabel('Y Coordinate')

# Create a discrete color map with the number of clusters for Plot 2
cmap_discrete_2 = matplotlib.colormaps.get_cmap('viridis')

# Define boundaries for the discrete color map for Plot 2
boundaries_2 = np.arange(-0.5, num_clusters_2, 1)

# Create a BoundaryNorm for the color map for Plot 2
norm_discrete_2 = mcolors.BoundaryNorm(boundaries_2, cmap_discrete_2.N, clip=True)

# Add a discrete color bar with integer cluster labels for Plot 2
cbar_2 = plt.colorbar(scatter_2, ax=ax2, ticks=np.arange(num_clusters_2),
cmap=cmap_discrete_2, norm=norm_discrete_2, boundaries=boundaries_2)
cbar_2.set_label('Cluster Label')

# Set the number of tick marks on the X and Y axes for Plot 2
ax2.xaxis.set_major_locator(MaxNLocator(nbins=3))
ax2.yaxis.set_major_locator(MaxNLocator(nbins=3))

# Plot 3: Line Plot of Combined Unnormalized Cluster Centers
```

```
for cluster_label_1 in range(num_clusters_1):
    color_1 = cmap_discrete_1(cluster_label_1 / (num_clusters_1 - 1))  # Match color from
scatter plot (2017 S2)
    ax3.plot(remaining_data_1.columns, cluster_centers_original_scale_1[cluster_label_1],
label=f'Cluster {cluster_label_1} (2017 S2)', color=color_1)

for cluster_label_2 in range(num_clusters_2):
    color_2 = cmap_discrete_2(cluster_label_2 / (num_clusters_2 - 1))  # Match color from
scatter plot (2018 S2)
    ax3.plot(remaining_data_2.columns, cluster_centers_original_scale_2[cluster_label_2],
'--', label=f'Cluster {cluster_label_2} (2018 S2)', color=color_2)

# Set plot properties for Plot 3
ax3.set_title('Combined Line Plot of Unnormalized Cluster Centers')
ax3.set_xlabel('Column Name')
ax3.set_ylabel('Cluster Center Spectrograph')
ax3.legend()

# Adjust layout for better spacing
plt.tight_layout()

# Show the plots
plt.show()
```

```
Enter the number of clusters for 2017 S2 K-Means: 4
Enter the number of clusters for 2018 S2 K-Means: 3
```



5. Using Centroid Clustering to combine two remotely sensed datasets (Satellite S2 and Airborne Radiometric data)

In this part of the workshop you will be provided with data from ESA Sentinel 2 Optical Satellite data and the Tellus Radiometric Survey

1. The satellite data were acquired in June 2017 over a peatland site in Ireland
2. The airborne radiometric data were acquired in June 2016, over the same peatland site in Ireland
3. The Satellite data were processed Semi-Automatic Classification plugin in QGIS
4. The Radiometric data were processed by the acquisiton contracter and converted to Counts per Second following O'Leary et al 2022
5. Both datasets were gridded to a 50 x 50 m grid (Lowest resolution of the two datasets) so they could be matched
6. Only Pixels that have been identified as being "Peat" are presented
7. All coordinate information has been altered to keep the privacy of the land owner

The aim of this workshop is to determine the appropriate number of clusters for each of the two datasets using KMeans clustering and MCASD and then combine the datasets in a final clustering analysis and perform a joint interpretation

The input file name is: `AgroGeo24_WS_Part_5_20170620_Sentinel2.csv` and `AgroGeo24_WS_Part_5_Tellus_RM.csv` and should be located in the folder named "Part 5" of the data directory provided.

Please following along with the workshop leader in the first instance until you are familiar with using Google Colab environment.

No coding experience is required to run this code. All the code contains comments describing what each line does. Please click "Show Code" on any section to view the code.

Please feel free to ask questions if you don't understand any parts.

5.1. Section 0

This section sets up the Python environment for the clustering analysis.

It imports essential libraries such as Pandas for data manipulation, NumPy for numerical operations, Matplotlib for data visualization, scikit-learn for machine learning tools, and other supporting libraries.

Additionally, it configures the display.

The code also imports specific functions and modules required for the clustering analysis, such as KMeans

Finally, it sets up tools for working with images, zip files, and file uploads in Google Colab. This preparation ensures that the subsequent code can efficiently perform clustering analysis and handle related tasks.

```
# @title
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
import imageio.v2 as imageio
import matplotlib
import imageio.v2 as imageio
import os
import re

from sklearn.cluster import KMeans
```

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import normalize
from sklearn.preprocessing import scale
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_score
from sklearn.metrics import pairwise_distances_argmin_min
from collections import Counter
from sklearn.preprocessing import MinMaxScaler
from matplotlib.ticker import MaxNLocator
from PIL import Image
from zipfile import ZipFile
from google.colab import files


# Set display options to show all rows and columns
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
```

5.2. Section 1

Now we are going to read in the data from your local device. If you have not yet downloaded the necessary data, please do so from here.

The Data are in CSV format with the following header descriptors:

1. Column 1 = X Coordinate
2. Column 2 = Y Coordinate
3. Columns 3..N = Data

Press the play button below to select the correct files. They should be called:
`AgroGeo24_WS_Part_5_20170620_Sentinel2.csv` and `AgroGeo24_WS_Part_5_Tellus_RM.csv`

You will see a button “Choose files”. Click this and navigate to where this file is stored on your machine. Then highlight it and click “open”.

Please upload the Satellite dataset first and then the Tellus RM dataset

When running the cell, you will see a progress message as the file is uploaded to this Google Colab environment.

```
# @title
#### Section 1: Import CSV file  ####

# Prompt the user to upload a file
uploaded_1 = files.upload()
uploaded_2 = files.upload()

# Get the uploaded file name
file_name_1 = list(uploaded_1.keys())[0]
file_name_2 = list(uploaded_2.keys())[0]

# Extract the filename without the extension
file_name_without_extension_1 = os.path.splitext(file_name_1)[0]
file_name_without_extension_2 = os.path.splitext(file_name_2)[0]
```

```
# Remove numerical suffixes from the filename
file_name_without_extension_1 = re.sub(r'\(\d+\)', '', file_name_without_extension_1)
file_name_without_extension_2 = re.sub(r'\(\d+\)', '', file_name_without_extension_2)
file_name_without_extension_1 = re.sub(r'\(\ \)', '', file_name_without_extension_1)
file_name_without_extension_2 = re.sub(r'\(\ \)', '', file_name_without_extension_2)

# Read the CSV file into a DataFrame
df_1 = pd.read_csv(file_name_1).round(4)
df_2 = pd.read_csv(file_name_2).round(2)
```

Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving AgroGeo24_WS_Part_5_20170620_Sentinel2.csv to AgroGeo24_WS_Part_5_20170620_Sentinel2
(3).csv
```

Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving AgroGeo24_WS_Part_5_Tellus_RM.csv to AgroGeo24_WS_Part_5_Tellus_RM (3).csv
```

Excellent. You have now loaded the CSV file into the Google Colab environment and are ready to take a look at the data.

5.3. Section 2

In this section, we're inspecting and refining the dataset. Initially, we showcase a snippet of the original dataset, including its structure and the number of rows.

Next, we perform data cleaning by eliminating rows containing NaN (Not a Number) and blank values. The cleaned dataset is then displayed, again with a snippet and the updated row count.

B10 is removed in this instance as it is a measurement of cloud. As these images were acquired on cloud free days, this layer is considered as noise and skews the clustering results

This process ensures that the dataset is well-prepared for subsequent analyses by removing any instances of missing or empty data.

Note that this dataset contained no blank values, so the number of rows doesn't change

```
# @title
#### Section 2: Data Cleaning ####

# Display the original DataFrame and its shape
print("File 1: Original Data:")
print(df_1.head(5).to_string(index=False))
print(f"Number of rows before cleaning: {df_1.shape[0]}")

print("\nFile 2: Original Data:")
print(df_2.head(5).to_string(index=False))
print(f"Number of rows before cleaning: {df_2.shape[0]}")

# Remove rows with NaN and blank values
```

```
df_cleaned_1 = df_1.dropna().replace('', np.nan).dropna()
df_cleaned_1 = df_cleaned_1.drop(columns = ['B10'])  # Assuming rows are zero-indexed
df_cleaned_2 = df_2.dropna().replace('', np.nan).dropna()

# Display the cleaned DataFrame and its shape
print("\nFile 1: Data after removing NaN and blank values:")
print(df_cleaned_1.head(5).to_string(index=False))
print(f"Number of rows after cleaning: {df_cleaned_1.shape[0]}")

print("\nFile 2: Data after removing NaN and blank values:")
print(df_cleaned_2.head(5).to_string(index=False))
print(f"Number of rows after cleaning: {df_cleaned_2.shape[0]}")
```

```
File 1: Original Data:
   X    Y     B1     B2     B3     B4     B5     B6     B7     B8    B8A     B9    B10
B11    B12
9750 9000 0.0307 0.0361 0.0655 0.0467 0.1144 0.3056 0.3908 0.3858 0.4550 0.0959 0.0113
0.2055 0.0984
9800 9000 0.0304 0.0347 0.0639 0.0449 0.1124 0.3135 0.3988 0.3915 0.4604 0.0946 0.0113
0.2065 0.0962
9850 9000 0.0293 0.0323 0.0592 0.0405 0.1056 0.3236 0.4176 0.4080 0.4769 0.0945 0.0112
0.1987 0.0897
9900 9000 0.0290 0.0316 0.0563 0.0389 0.0997 0.3150 0.4117 0.4017 0.4677 0.0915 0.0111
0.1950 0.0898
9550 8950 0.0295 0.0300 0.0502 0.0378 0.0877 0.2845 0.3799 0.3697 0.4327 0.0884 0.0114
0.1930 0.0877
Number of rows before cleaning: 16794

File 2: Original Data:
   X    Y     K     U   Th     TC
9750 9000 29.45 16.01 7.11 524.36
9800 9000 27.33 15.18 6.84 484.47
9850 9000 24.38 13.61 6.39 430.22
9900 9000 21.32 11.34 5.74 367.71
9550 8950 23.72 11.38 5.34 494.51
Number of rows before cleaning: 16794

File 1: Data after removing NaN and blank values:
   X    Y     B1     B2     B3     B4     B5     B6     B7     B8    B8A     B9    B11
B12
9750 9000 0.0307 0.0361 0.0655 0.0467 0.1144 0.3056 0.3908 0.3858 0.4550 0.0959 0.2055
0.0984
9800 9000 0.0304 0.0347 0.0639 0.0449 0.1124 0.3135 0.3988 0.3915 0.4604 0.0946 0.2065
0.0962
9850 9000 0.0293 0.0323 0.0592 0.0405 0.1056 0.3236 0.4176 0.4080 0.4769 0.0945 0.1987
0.0897
9900 9000 0.0290 0.0316 0.0563 0.0389 0.0997 0.3150 0.4117 0.4017 0.4677 0.0915 0.1950
0.0898
9550 8950 0.0295 0.0300 0.0502 0.0378 0.0877 0.2845 0.3799 0.3697 0.4327 0.0884 0.1930
0.0877
Number of rows after cleaning: 16794

File 2: Data after removing NaN and blank values:
   X    Y     K     U   Th     TC
```

```
9750 9000 29.45 16.01 7.11 524.36
9800 9000 27.33 15.18 6.84 484.47
9850 9000 24.38 13.61 6.39 430.22
9900 9000 21.32 11.34 5.74 367.71
9550 8950 23.72 11.38 5.34 494.51
Number of rows after cleaning: 16794
```

5.4. Section 3

Building on the cleaned dataset from the previous section, we now visualize the non-coordinate columns through scatter plots for both inputs.

Data from File 1 (2017 S2) is placed alongside data from File 2 (Tellus Radiometric data)

For information on each Satellite band see this link.

For information on Radiometric data see this link (O'Leary et al 2022) which also outlines how the pixels used in this workshop were chosen.

Each subplot represents a specific data column, showcasing its spatial distribution across the X and Y coordinates. The color intensity in each plot reflects the values of the corresponding data column.

The number of rows and columns for the subplot grid is dynamically calculated based on the available data columns.

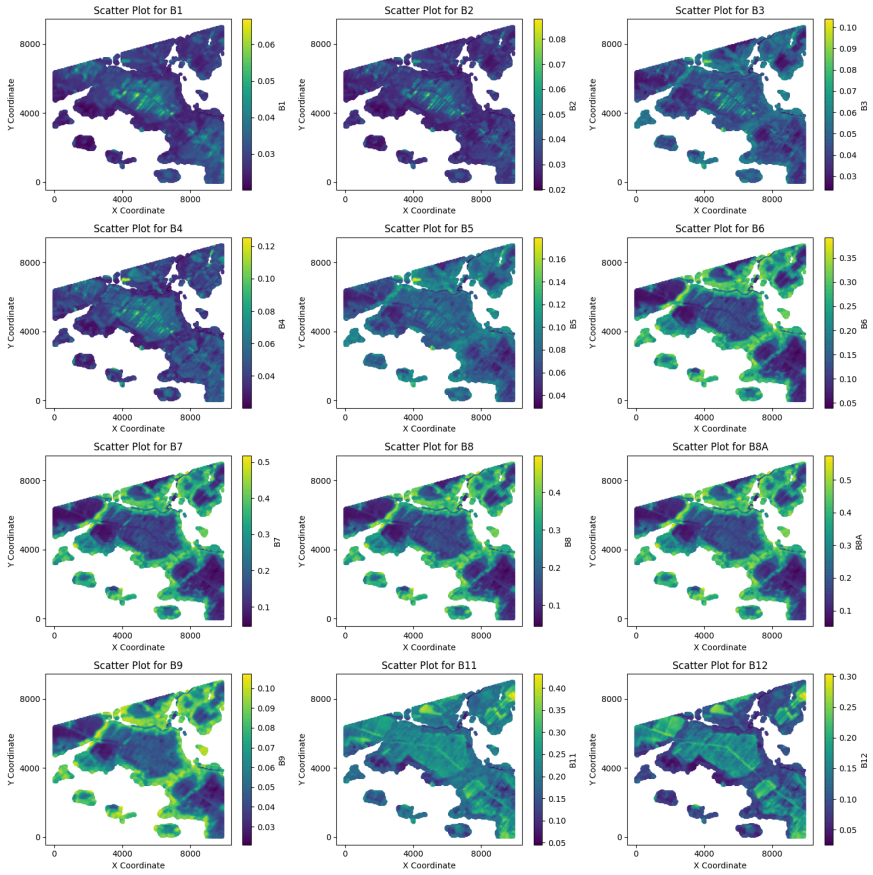
This visualization provides an initial exploration of how different data variables are distributed in the geographical space, setting the stage for further analysis and insights.

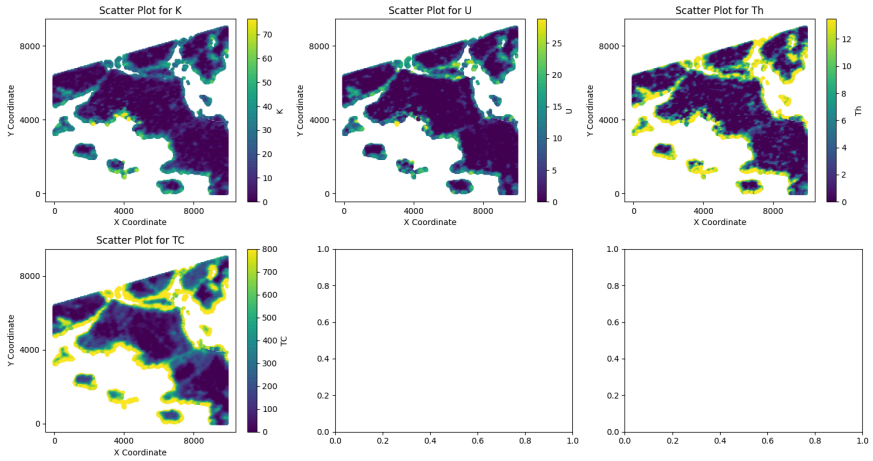
Note that only pixels that have been identified as being part of a peatland are included in this analysis (See accompanying presentation)

```
# @title
#### Section 3 Data Viewing ###

# Get the list of non-coordinate column names
data_column_names_1 = df_cleaned_1.columns[2:]
data_column_names_2 = df_cleaned_2.columns[2:]

# Get the list of non-coordinate column names
data_column_names_1 = df_cleaned_1.columns[2:]
data_column_names_2 = df_cleaned_2.columns[2:]

# Calculate the number of rows and columns for subplots
num_plots_1 = len(data_column_names_1)
num_plots_2 = len(data_column_names_2)
num_plots_per_row_1 = 3
num_plots_per_row_2 = 3
num_rows_1 = (num_plots_1 + num_plots_per_row_1 - 1) // num_plots_per_row_1
num_rows_2 = (num_plots_2 + num_plots_per_row_2 - 1) // num_plots_per_row_2
num_cols_1 = min(num_plots_1, num_plots_per_row_1)
num_cols_2 = min(num_plots_2, num_plots_per_row_2)

### Section 3.1 Plot Satellite Data ###

# Create subplots with the specified number of rows and columns
fig, axes = plt.subplots(num_rows_1, num_cols_1, figsize=(15, 15))
```

```
# Flatten the axes to simplify indexing
axes = axes.flatten()

# Loop through each data column and create scatter plots
for i, column_name in enumerate(data_column_names_1):
    # Extract the data for the current column
    column_data = df_cleaned_1[column_name]

    # Calculate the position in the subplot grid
    row_index = i // num_cols_1
    col_index = i % num_cols_1

    # Create a scatter plot
    scatter = axes[i].scatter(df_cleaned_1['X'], df_cleaned_1['Y'], c=column_data,
cmap='viridis', marker='o', s=10)

    # Set plot properties
    axes[i].set_title(f'Scatter Plot for {column_name}')
    axes[i].set_xlabel('X Coordinate')
    axes[i].set_ylabel('Y Coordinate')

    # Set the number of tick marks on the X and Y axes
    axes[i].xaxis.set_major_locator(MaxNLocator(nbins=3))
    axes[i].yaxis.set_major_locator(MaxNLocator(nbins=3))

    # Add a color bar scaled to the min and max of the current column
    cbar = plt.colorbar(scatter, ax=axes[i])
    cbar.set_label(column_name)

# Adjust layout for better spacing
plt.tight_layout()

# Show the plots
plt.show()

### Section 3.2 Plot Airborne Radiometrics Data ###

# Create subplots with the specified number of rows and columns
fig, axes = plt.subplots(num_rows_2, num_cols_2, figsize=(15, 8))

# Flatten the axes to simplify indexing
axes = axes.flatten()

# Loop through each data column and create scatter plots
for i, column_name in enumerate(data_column_names_2):
    # Extract the data for the current column
    column_data = df_cleaned_2[column_name]

    # Calculate the position in the subplot grid
    row_index = i // num_cols_2
    col_index = i % num_cols_2

    # Set min and max for colour bar for RM data
    vmin = 0
```

```
    vmax = column_data.max()/1.5

    # Create a scatter plot
    scatter = axes[i].scatter(df_cleaned_2['X'], df_cleaned_2['Y'], c=column_data,
cmap='viridis', marker='o', s=10, vmin=vmin, vmax=vmax)

    # Set plot properties
    axes[i].set_title(f'Scatter Plot for {column_name}')
    axes[i].set_xlabel('X Coordinate')
    axes[i].set_ylabel('Y Coordinate')

    # Set the number of tick marks on the X and Y axes
    axes[i].xaxis.set_major_locator(MaxNLocator(nbins=3))
    axes[i].yaxis.set_major_locator(MaxNLocator(nbins=3))

    # Add a color bar scaled to the min and max of the current column
    cbar = plt.colorbar(scatter, ax=axes[i])
    cbar.set_label(column_name)


# Adjust layout for better spacing
plt.tight_layout()

# Show the plots
plt.show()
```





5.5. Section 4

In this section, we strategically divide our dataset into two key components: coordinates and remaining data. The first two columns, containing coordinate information, are isolated to construct the “Coordinates” DataFrame.

Simultaneously, the remaining data, excluding the initial two columns, forms the “Remaining Data” DataFrame.

This separation serves a pivotal purpose—clustering analysis will solely operate on the remaining data. Subsequently, the cluster labels obtained can be associated with their respective X and Y coordinates. This distinction is fundamental for generating geographical cluster maps, allowing us to visually interpret and understand the spatial distribution of clusters across the dataset.

This section also focuses on data normalization, a crucial step before applying clustering algorithms.

The resulting normalized data is displayed, providing an insight into the standardized values across the dataset.

Normalization enhances the accuracy of clustering algorithms, ensuring that features with different scales contribute equally to the clustering process.

This code also combined these datasets together (appends columns) into a single dataset for use later.

```
# @title
#### Section 4 Separate the Data for clustering ####

# Extract coordinate information (assuming it's in the first two columns)
coordinates_1 = df_cleaned_1.iloc[:, :2]
coordinates_2 = df_cleaned_2.iloc[:, :2]

# Extract the remaining data (excluding the first two columns)
remaining_data_1 = df_cleaned_1.iloc[:, 2:]
remaining_data_1 = remaining_data_1.round(4)
remaining_data_2 = df_cleaned_2.iloc[:, 2:]
remaining_data_2 = remaining_data_2.round(4)

combined_remaining_data = pd.concat([remaining_data_1, remaining_data_2], axis=1)

# Display the Coordinates DataFrame
print("\n2017 S2 Coordinates:")
```

```
print(coordinates_1.head(3).to_string(index=False))  # Use to_string to prevent truncation
print(f"Number of rows: {df_cleaned_1.shape[0]}")

print("\nTellus RM Coordinates:")
print(coordinates_2.head(3).to_string(index=False))  # Use to_string to prevent truncation
print(f"Number of rows: {df_cleaned_2.shape[0]}")

# Display the Remaining Data DataFrame
print("\n2017 S2 Remaining Data:")
print(remaining_data_1.head(3).to_string(index=False))  # Use to_string to prevent
truncation
print(f"Number of rows: {df_cleaned_1.shape[0]}")

print("\nTellus RM Remaining Data:")
print(remaining_data_2.head(3).to_string(index=False))  # Use to_string to prevent
truncation
print(f"Number of rows: {df_cleaned_2.shape[0]}")

# Custom normalization using MinMaxScaler
min_vals_1 = remaining_data_1.min()
max_vals_1 = remaining_data_1.max()
min_vals_2 = remaining_data_2.min()
max_vals_2 = remaining_data_2.max()
min_vals_3 = combined_remaining_data.min()
max_vals_3 = combined_remaining_data.max()

normalized_data_1 = (remaining_data_1 - min_vals_1.values) / (max_vals_1.values -
min_vals_1.values)
normalized_data_2 = (remaining_data_2 - min_vals_2.values) / (max_vals_2.values -
min_vals_2.values)
combined_normalized_data = (combined_remaining_data - min_vals_3.values) /
(max_vals_3.values - min_vals_3.values)

# Round the normalized data to 4 decimal places
#normalized_data_1 = normalized_data_1.round(4)
#normalized_data_2 = normalized_data_2.round(4)

# Convert the rounded normalized data back to a DataFrame and set column names
normalized_df_1 = pd.DataFrame(normalized_data_1, columns=remaining_data_1.columns)
normalized_df_2 = pd.DataFrame(normalized_data_2, columns=remaining_data_2.columns)


# Display the normalized data
print("\n2017 S2 Normalized Data:")
print(normalized_data_1.head(3).to_string(index=False))  # Use to_string to prevent
truncation
print(f"Number of rows: {df_cleaned_1.shape[0]}")

print("\nTellus RM Normalized Data:")
print(normalized_data_2.head(3).to_string(index=False))  # Use to_string to prevent
truncation
print(f"Number of rows: {df_cleaned_2.shape[0]}")
```

```
2017 S2 Coordinates:
   X    Y
9750 9000
9800 9000
9850 9000
Number of rows: 16794

Tellus RM Coordinates:
   X    Y
9750 9000
9800 9000
9850 9000
Number of rows: 16794

2017 S2 Remaining Data:
    B1     B2     B3     B4     B5     B6     B7     B8    B8A     B9    B11    B12
0.0307 0.0361 0.0655 0.0467 0.1144 0.3056 0.3908 0.3858 0.4550 0.0959 0.2055 0.0984
0.0304 0.0347 0.0639 0.0449 0.1124 0.3135 0.3988 0.3915 0.4604 0.0946 0.2065 0.0962
0.0293 0.0323 0.0592 0.0405 0.1056 0.3236 0.4176 0.4080 0.4769 0.0945 0.1987 0.0897
Number of rows: 16794

Tellus RM Remaining Data:
    K     U   Th     TC
29.45 16.01 7.11 524.36
27.33 15.18 6.84 484.47
24.38 13.61 6.39 430.22
Number of rows: 16794

2017 S2 Normalized Data:
      B1       B2       B3       B4       B5       B6       B7       B8      B8A       B9
B11      B12
0.224839 0.239003 0.522444 0.252372 0.570100 0.753894 0.731299 0.754738 0.771953 0.868056
0.413927 0.261135
0.218415 0.218475 0.502494 0.235294 0.556811 0.776267 0.748252 0.767298 0.782284 0.853009
0.416516 0.253233
0.194861 0.183284 0.443890 0.193548 0.511628 0.804871 0.788091 0.803658 0.813851 0.851852
0.396324 0.229885
Number of rows: 16794

Tellus RM Normalized Data:
       K        U       Th       TC
0.466612 0.646668 0.596178 0.480646
0.453345 0.635867 0.587855 0.449985
0.434883 0.615435 0.573983 0.408286
Number of rows: 16794
```

5.6. Section 5

This section guides the user in determining the optimal number of clusters (k) for the K-Means algorithm by utilizing both the Elbow Method and Silhouette Scores.

After specifying the maximum number of clusters to consider, the code calculates the Within-Cluster Sum of Squares (WCSS) distance using the Elbow Method. The Elbow Method graph illustrates the trade-off between clustering complexity and WCSS reduction, helping identify an optimal k value.

Simultaneously, Silhouette Scores, a measure of how well-separated clusters are, are computed and presented on the same graph. Silhouette Scores range from −1 to 1, where higher scores indicate better-defined clusters.

When assessing the graph, users should look for the “elbow” point where WCSS plateaus, suggesting diminishing returns with additional clusters.

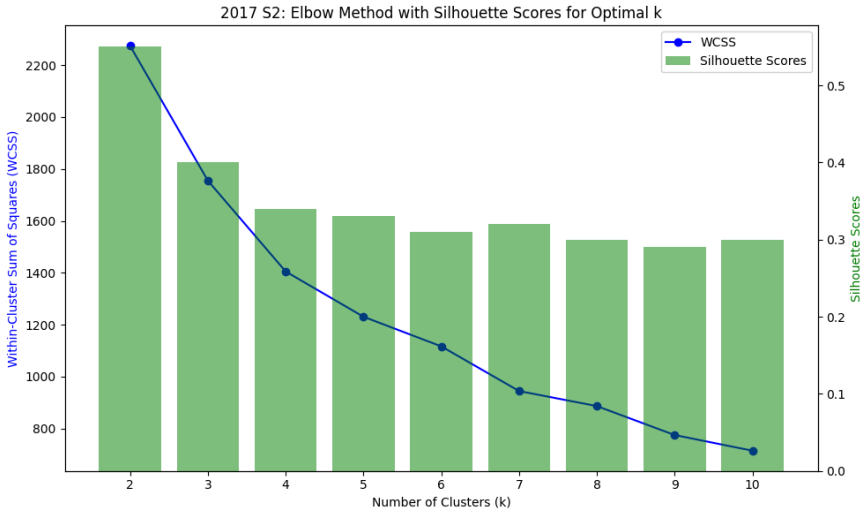
Additionally, a high Silhouette Score at the “elbow” reinforces the choice, ensuring a balance between compact clusters and distinct cluster boundaries for effective clustering

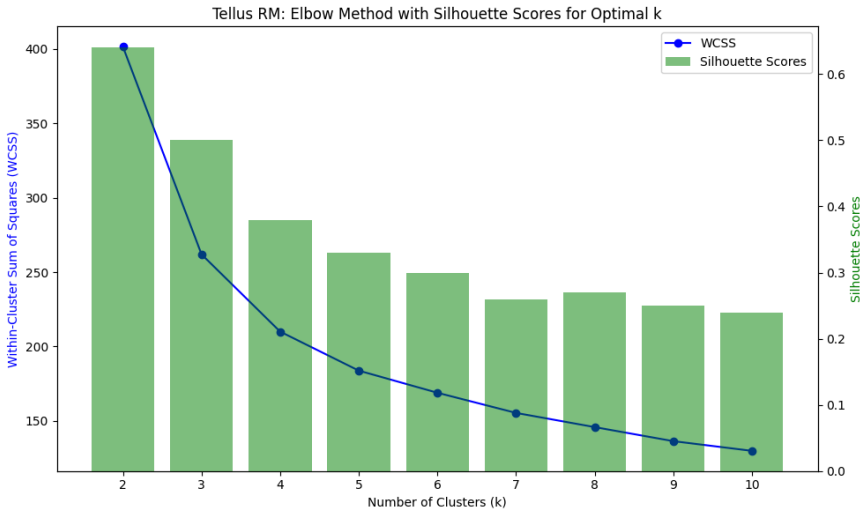
When running the cell, you will be prompted to enter the max number of clusters. (i.e., 10)

```
# @title
#### Section 5 Elbow Method with Silhouette Scores ####

# Prompt the user to choose the maximum number of clusters for the Elbow Method
max_clusters_elbow = int(input("Enter the maximum number of clusters for the Elbow Method:
"))

# Calculate the within-cluster sum of squares (WCSS) and Silhouette Scores for different
values of k
wcss_1 = []
silhouette_1 = []
wcss_2 = []
silhouette_2 = []
for k in range(2, max_clusters_elbow + 1):
    kmeans_1 = KMeans(n_clusters=k, n_init=1, init='k-means++')
    kmeans_2 = KMeans(n_clusters=k, n_init=1, init='k-means++')
    kmeans_1.fit(normalized_data_1)
    kmeans_2.fit(normalized_data_2)
    sscore_1 = round(silhouette_score(normalized_data_1, kmeans_1.labels_), 2)
    sscore_2 = round(silhouette_score(normalized_data_2, kmeans_2.labels_), 2)
    silhouette_1.append(sscore_1)
    silhouette_2.append(sscore_2)
    wcss_1.append(kmeans_1.inertia_)
    wcss_2.append(kmeans_2.inertia_)

# Plot the Elbow Method graph with Silhouette Scores as a bar graph for 2017 S2
fig, ax1 = plt.subplots(figsize=(10, 6))

# Plot WCSS on the left y-axis
ax1.plot(range(2, max_clusters_elbow + 1), wcss_1, marker='o', color='blue', label='WCSS')
ax1.set_xlabel('Number of Clusters (k)')
ax1.set_ylabel('Within-Cluster Sum of Squares (WCSS)', color='blue')

# Set the x-axis ticks to show only integers
plt.xticks(range(2, max_clusters_elbow + 1))

# Create a second y-axis for Silhouette Scores
ax2 = ax1.twinx()
ax2.bar(range(2, max_clusters_elbow + 1), silhouette_1, color='green', alpha=0.5,
label='Silhouette Scores')
ax2.set_ylabel('Silhouette Scores', color='green')

# Add legend
lines, labels = ax1.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
```

```
ax2.legend(lines + lines2, labels + labels2, loc='upper right')

plt.title('2017 S2: Elbow Method with Silhouette Scores for Optimal k')
fig.tight_layout()
plt.show()

# Plot the Elbow Method graph with Silhouette Scores as a bar graph for 2018 S2
fig, ax1 = plt.subplots(figsize=(10, 6))

# Plot WCSS on the left y-axis
ax1.plot(range(2, max_clusters_elbow + 1), wcss_2, marker='o', color='blue', label='WCSS')
ax1.set_xlabel('Number of Clusters (k)')
ax1.set_ylabel('Within-Cluster Sum of Squares (WCSS)', color='blue')

# Set the x-axis ticks to show only integers
plt.xticks(range(2, max_clusters_elbow + 1))

# Create a second y-axis for Silhouette Scores
ax2 = ax1.twinx()
ax2.bar(range(2, max_clusters_elbow + 1), silhouette_2, color='green', alpha=0.5,
label='Silhouette Scores')
ax2.set_ylabel('Silhouette Scores', color='green')

# Add legend
lines, labels = ax1.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
ax2.legend(lines + lines2, labels + labels2, loc='upper right')

plt.title('Tellus RM: Elbow Method with Silhouette Scores for Optimal k')
fig.tight_layout()
plt.show()
```

```
Enter the maximum number of clusters for the Elbow Method: 10
```





5.7. Section 6

In this section, users will perform K-Means clustering on the preprocessed Satellite and Tellus RM data.

The optimal number of clusters (k) can be determined by referencing the results from the previous Elbow Method and Silhouette Scores analysis (Section 5). After entering the desired number of clusters, the code applies K-Means clustering and displays key information.

The results include a count of occurrences for each cluster label via a plot of the classified image and “spectral” graph showing the cluster centers.

The scatter plot provides an overview of spatial distribution of the clusters, while the line plot illustrates how cluster center values vary across the survey

> **Note**
>
> The Tellus Radiometric data are plotted as normalised as the Total Count data have significantly different dynamic range to the other data layers.

You will be prompted to enter the number of clusters for both 2017 and Tellus RM datasets, which should be based on the Elbow and Silhouette results.

Even without knowing the most appropriate number of clusters what does this analysis tell you about the Satellite data across this survey and also differences in data with time?

```
# @title
##### Section 6 K Means Clustering ####

### Section 6.1 Perform K Means Clustering ###

# Prompt the user to choose the number of clusters for K-Means
num_clusters_1 = int(input("Enter the number of clusters for 2017 S2 K-Means: "))
num_clusters_2 = int(input("Enter the number of clusters for Tellus RM K-Means: "))

# Initialize the K-Means model
kmeans_1 = KMeans(n_clusters=num_clusters_1, init = 'k-means++', n_init=1)
kmeans_2 = KMeans(n_clusters=num_clusters_2, init = 'k-means++', n_init=1)
```

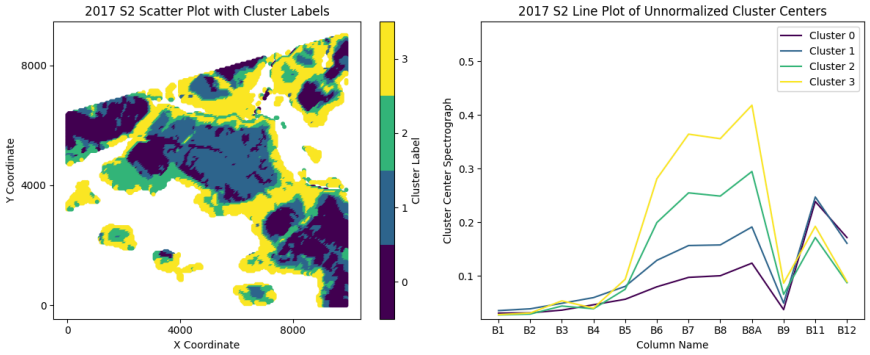
```
# Fit the K-Means model to the normalized data
kmeans_1.fit(normalized_data_1)
kmeans_2.fit(normalized_data_2)

# Get the cluster labels for each data point
cluster_labels_1 = kmeans_1.labels_
cluster_labels_2 = kmeans_2.labels_

# Get the cluster centers
cluster_centers_1 = kmeans_1.cluster_centers_
cluster_centers_2 = kmeans_2.cluster_centers_

### Section 6.2 Sort the Cluster centers ###

# Calculate the distances of cluster centers from the origin (0, 0)
distances_from_origin_1 = np.sqrt(np.sum(cluster_centers_1 ** 2, axis=1))
distances_from_origin_2 = np.sqrt(np.sum(cluster_centers_2 ** 2, axis=1))

# Sort cluster centers based on distances from the origin
sorted_indices_1 = np.argsort(distances_from_origin_1)
sorted_indices_2 = np.argsort(distances_from_origin_2)

# Sort cluster centers and labels
sorted_cluster_centers_1 = cluster_centers_1[sorted_indices_1]
sorted_cluster_centers_2 = cluster_centers_2[sorted_indices_2]
sorted_cluster_labels_1 = np.zeros_like(cluster_labels_1)
sorted_cluster_labels_2 = np.zeros_like(cluster_labels_2)

# Relabel the cluster labels based on the sorted order
for new_label_1, old_label_1 in enumerate(sorted_indices_1):
    sorted_cluster_labels_1[cluster_labels_1 == old_label_1] = new_label_1

for new_label_2, old_label_2 in enumerate(sorted_indices_2):
    sorted_cluster_labels_2[cluster_labels_2 == old_label_2] = new_label_2

# Calculate the count of each cluster label
cluster_labels_count_1 = dict(zip(*np.unique(sorted_cluster_labels_1, return_counts=True)))
cluster_labels_count_2 = dict(zip(*np.unique(sorted_cluster_labels_2, return_counts=True)))

### Section 6.3 Denormalize the cluster centers ###

# Denormalize the data using the inverse transformation
cluster_centers_original_scale_1 = sorted_cluster_centers_1 * (max_vals_1.values -
min_vals_1.values) + min_vals_1.values
cluster_centers_original_scale_2 = sorted_cluster_centers_2 #* (max_vals_2.values -
min_vals_2.values) + min_vals_2.values

### Section 6.4 Display Clustering counts for visual QC ###

# Display the count of each cluster label
print("\n2017 S2 Count of Each Cluster Label:")
for label_1, count_1 in cluster_labels_count_1.items():
    print(f"Cluster {label_1}: {count_1} occurrences")

print("\nTellus RM Count of Each Cluster Label:")
```

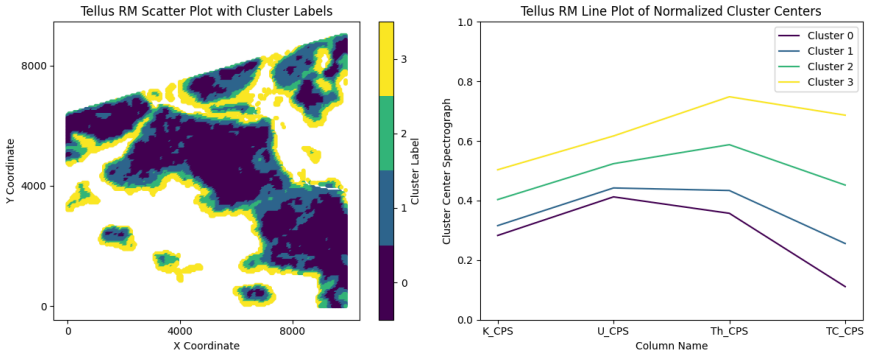
```
for label_2, count_2 in cluster_labels_count_2.items():
    print(f"Cluster {label_2}: {count_2} occurrences")

### Section 6.5 Save results to CSV ###

# Create a DataFrame with X, Y, Cluster, and Remaining Data
clustered_data_df_1 = pd.DataFrame({
    'X': coordinates_1['X'],
    'Y': coordinates_1['Y'],
    'Cluster Number': sorted_cluster_labels_1,
    **{f'{col}': remaining_data_1[col] for col in remaining_data_1.columns}
})
clustered_data_df_2 = pd.DataFrame({
    'X': coordinates_2['X'],
    'Y': coordinates_2['Y'],
    'Cluster Number': sorted_cluster_labels_2,
    **{f'{col}': remaining_data_2[col] for col in remaining_data_2.columns}
})
clustered_data_df_1 = clustered_data_df_1.round(4)
clustered_data_df_2 = clustered_data_df_2.round(4)

# Create a DataFrame with Cluster center data
center_data_df_1 = pd.DataFrame(cluster_centers_original_scale_1,
columns=remaining_data_1.columns)
center_data_df_2 = pd.DataFrame(cluster_centers_original_scale_2,
columns=remaining_data_2.columns)

# Add a new column 'Cluster Number' to indicate the cluster number for each row
center_data_df_1.insert(0, 'Cluster Number', range(num_clusters_1))
center_data_df_2.insert(0, 'Cluster Number', range(num_clusters_2))
center_data_df_1 = center_data_df_1.round(4)
center_data_df_2 = center_data_df_2.round(4)

### Section 6.6 Plot KMeans Clustering results ###

# 2017 S2 Create a scatter plot of X, Y, and final cluster label
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Plot 1: Scatter Plot with Cluster Labels
scatter_1 = ax1.scatter(coordinates_1['X'], coordinates_1['Y'], c=sorted_cluster_labels_1,
cmap='viridis', marker='o', s=10)

# Set plot properties for Plot 1
ax1.set_title('2017 S2 Scatter Plot with Cluster Labels')
ax1.set_xlabel('X Coordinate')
ax1.set_ylabel('Y Coordinate')

# Create a discrete color map with the number of clusters for Plot 1
cmap_discrete_1 = matplotlib.colormaps.get_cmap('viridis')

# Define boundaries for the discrete color map for Plot 1
boundaries_1 = np.arange(-0.5, num_clusters_1, 1)

# Create a BoundaryNorm for the color map for Plot 1
norm_discrete_1 = mcolors.BoundaryNorm(boundaries_1, cmap_discrete_1.N, clip=True)
```

```
# Add a discrete color bar with integer cluster labels for Plot 1
cbar_1 = plt.colorbar(scatter_1, ax=ax1, ticks=np.arange(num_clusters_1),
cmap=cmap_discrete_1, norm=norm_discrete_1, boundaries=boundaries_1)
cbar_1.set_label('Cluster Label')

# Set the number of tick marks on the X and Y axes for Plot 1
ax1.xaxis.set_major_locator(MaxNLocator(nbins=3))
ax1.yaxis.set_major_locator(MaxNLocator(nbins=3))

# Plot 2: Line Plot of Unnormalized Cluster Centers
for cluster_label_1 in range(num_clusters_1):
    color_1 = cmap_discrete_1(cluster_label_1 / (num_clusters_1 - 1))  # Match color from
scatter plot
    ax2.plot(remaining_data_1.columns, cluster_centers_original_scale_1[cluster_label_1],
label=f'Cluster {cluster_label_1}', color=color_1)
    ax2.set_ylim(remaining_data_1.min().min(), remaining_data_1.max().max())

# Set plot properties for Plot 2
ax2.set_title('2017 S2 Line Plot of Unnormalized Cluster Centers')
ax2.set_xlabel('Column Name')
ax2.set_ylabel('Cluster Center Spectrograph')
ax2.legend()

# Show the plots
plt.tight_layout()
plt.show()

# Tellus RM Create a scatter plot of X, Y, and final cluster label
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Plot 1: Scatter Plot with Cluster Labels
scatter_2 = ax1.scatter(coordinates_2['X'], coordinates_2['Y'], c=sorted_cluster_labels_2,
cmap='viridis', marker='o', s=10)

# Set plot properties for Plot 1
ax1.set_title('Tellus RM Scatter Plot with Cluster Labels')
ax1.set_xlabel('X Coordinate')
ax1.set_ylabel('Y Coordinate')

# Create a discrete color map with the number of clusters for Plot 1
cmap_discrete_2 = matplotlib.colormaps.get_cmap('viridis')

# Define boundaries for the discrete color map for Plot 1
boundaries_2 = np.arange(-0.5, num_clusters_2, 1)

# Create a BoundaryNorm for the color map for Plot 1
norm_discrete_2 = mcolors.BoundaryNorm(boundaries_2, cmap_discrete_2.N, clip=True)

# Add a discrete color bar with integer cluster labels for Plot 1
cbar_2 = plt.colorbar(scatter_2, ax=ax1, ticks=np.arange(num_clusters_2),
cmap=cmap_discrete_2, norm=norm_discrete_2, boundaries=boundaries_2)
cbar_2.set_label('Cluster Label')

# Set the number of tick marks on the X and Y axes for Plot 1
```

```
ax1.xaxis.set_major_locator(MaxNLocator(nbins=3))
ax1.yaxis.set_major_locator(MaxNLocator(nbins=3))

# Plot 2: Line Plot of Unnormalized Cluster Centers
for cluster_label_2 in range(num_clusters_2):
    color_2 = cmap_discrete_2(cluster_label_2 / (num_clusters_2 - 1))  # Match color from
scatter plot
    ax2.plot(remaining_data_2.columns, cluster_centers_original_scale_2[cluster_label_2],
label=f'Cluster {cluster_label_2}', color=color_2)
    ax2.set_ylim(normalized_data_2.min().min(), normalized_data_2.max().max())

# Set plot properties for Plot 2
ax2.set_title('Tellus RM Line Plot of Normalized Cluster Centers')
ax2.set_xlabel('Column Name')
ax2.set_ylabel('Cluster Center Spectrograph')
ax2.legend()

# Show the plots
plt.tight_layout()
plt.show()
```

```
Enter the number of clusters for 2017 S2 K-Means: 4
Enter the number of clusters for Tellus RM K-Means: 4

2017 S2 Count of Each Cluster Label:
Cluster 0: 4374 occurrences
Cluster 1: 4680 occurrences
Cluster 2: 3099 occurrences
Cluster 3: 4641 occurrences

Tellus RM Count of Each Cluster Label:
Cluster 0: 7681 occurrences
Cluster 1: 4472 occurrences
Cluster 2: 2358 occurrences
Cluster 3: 2283 occurrences
```





5.8. Section 7

This section applies the MCASD (Multiple Cluster Average Standard Deviation) method, designed to aid participants in identifying the optimal number of clusters for each of the datasets.

This data analysis and MCASD was first published as a method by O'Leary et al 2023 here.

MCASD evaluates the stability of cluster centers across multiple attempts and cluster numbers. Participants will input the maximum number of clusters and the maximum number of attempts per cluster.

The code loops through different cluster numbers, applying K-Means clustering multiple times to analyze stability.

The results include GIFs illustrating scatter plots and line plots for each attempt. Additionally, MCASD metrics are calculated, providing insights into the stability and consistency of clusters. A line plot visualizes the MCASD metric across various cluster numbers, aiding participants in selecting the optimal cluster count.

All results, including plots and metrics, are compressed into a zip file for easy download.

Please save this ZIP file to "Part 5" of the data directory you were provided with.

The ultimate goal is to assist participants in making informed decisions about the optimal number of clusters for their specific datasets.

When running the cell, you will be prompted to enter the number of max number of clusters (i.e., 10) and max attempts (i.e., 10).

MCASD will be applied to both 2017 S2 and Tellus RM datasets seperately and MCASD plots will be displayed at the end.

Note this might take some time depending on the max number of clusters and max attempts chosen.

Once this is complete you can go back to Section 6 and input the most approprate number of clusters for each dataset, before moving on to a combined analysis in Section 8.

```
# @title
#### Section 7 MCASD Method ####
### Section 7.1 Get information from the user ###

# Prompt the user for the maximum number of clusters for MCASD Method
max_num_clusters = int(input("Enter the maximum number of clusters for MCASD Method: "))
# Prompt the user for the maximum number of attempts for MCASD Method
max_attempts = int(input("Enter the maximum number of attempts for MCASD Method: "))

### Section 7.2 Loop for MCASD Method ###
```

```
# Create a DataFrame to store MCASD Metrics
mcasd_metrics_df_1 = pd.DataFrame(index=['MCASD Metric', 'MCASD Error'], columns=range(1,
max_num_clusters + 1))
mcasd_metrics_df_2 = pd.DataFrame(index=['MCASD Metric', 'MCASD Error'], columns=range(1,
max_num_clusters + 1))

print(f"\nCalculating MCASD Metrics...")

# Create a zip file to store all results
zip_filename = f'AgroGeo24_WS_Part_5_kmeans_plots.zip'
with ZipFile(zip_filename, 'w') as zip_file:

    # Loop through the various number of clusters
    for num_clusters in range(2, max_num_clusters + 1):
        images_attempt_1 = []  # List to store images for the current attempt
        images_attempt_2 = []  # List to store images for the current attempt
        distances_df_1 = pd.DataFrame()  # Initialize distances DataFrame
        distances_df_2 = pd.DataFrame()  # Initialize distances DataFrame

        # Cluster the data a user specified number of times (Attempts)
        for attempt in range(1, max_attempts + 1):
            #print(f"\nNumber of Clusters: {num_clusters}: Attempt {attempt} of
{max_attempts}")

            # Initialize the K-Means model
            kmeans_1 = KMeans(n_clusters=num_clusters, n_init=1, init='k-means++')
            kmeans_2 = KMeans(n_clusters=num_clusters, n_init=1, init='k-means++')

            # Fit the K-Means model to the normalized data
            kmeans_1.fit(normalized_data_1)
            kmeans_2.fit(normalized_data_2)

            # Get the cluster labels for each data point
            cluster_labels_1 = kmeans_1.labels_
            cluster_labels_2 = kmeans_2.labels_

            # Get the cluster centers
            cluster_centers_1 = kmeans_1.cluster_centers_
            cluster_centers_2 = kmeans_2.cluster_centers_

            ### Section 7.2.1 Sort the Cluster centers ###

            # Calculate the distances of cluster centers from the origin (0, 0)
            distances_from_origin_1 = np.sqrt(np.sum(cluster_centers_1 ** 2, axis=1))
            distances_from_origin_2 = np.sqrt(np.sum(cluster_centers_2 ** 2, axis=1))

            # Sort cluster centers based on distances from the origin
            sorted_indices_1 = np.argsort(distances_from_origin_1)
            sorted_indices_2 = np.argsort(distances_from_origin_2)

            # Sort cluster centers and labels
            sorted_cluster_centers_1 = cluster_centers_1[sorted_indices_1]
            sorted_cluster_centers_2 = cluster_centers_2[sorted_indices_2]
            sorted_cluster_labels_1 = np.zeros_like(cluster_labels_1)
            sorted_cluster_labels_2 = np.zeros_like(cluster_labels_2)
```

```
            # Relabel the cluster labels based on the sorted order
            for new_label_1, old_label_1 in enumerate(sorted_indices_1):
                sorted_cluster_labels_1[cluster_labels_1 == old_label_1] = new_label_1

            for new_label_2, old_label_2 in enumerate(sorted_indices_2):
                sorted_cluster_labels_2[cluster_labels_2 == old_label_2] = new_label_2

            ### Section 7.2.2 Calculate the distance (in the dataspace) between each
datapoint and its closest cluster center ###

            # Calculate the distances between cluster centers and data
            distances_1 = np.linalg.norm(normalized_data_1.values[:, np.newaxis, :] -
cluster_centers_1, axis=-1)
            distances_2 = np.linalg.norm(normalized_data_2.values[:, np.newaxis, :] -
cluster_centers_2, axis=-1)

            # Get the smallest distance for each data point
            min_distances_1 = np.min(distances_1, axis=1)
            min_distances_2 = np.min(distances_2, axis=1)

            # Create a DataFrame for distances with only the smallest distances
            new_column_1 = pd.DataFrame(min_distances_1, columns=[f'Attempt_{attempt}'])
            new_column_2 = pd.DataFrame(min_distances_2, columns=[f'Attempt_{attempt}'])

            # Append the new column to the existing distances_df
            distances_df_1 = pd.concat([distances_df_1, new_column_1], axis=1)
            distances_df_2 = pd.concat([distances_df_2, new_column_2], axis=1)

            ### Section 7.2.3 Denormalize the cluster centers ###

            # Denormalize the data using the inverse transformation
            cluster_centers_original_scale_1 = sorted_cluster_centers_1 * (max_vals_1.values
- min_vals_1.values) + min_vals_1.values
            cluster_centers_original_scale_2 = sorted_cluster_centers_2 #*
(max_vals_2.values - min_vals_2.values) + min_vals_2.values

            ### Section 7.2.4 Create and save plots for later GIF creation ###

            # 2017 S2 Create a scatter plot of X, Y, and final cluster label
            fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

            # Plot 1: Scatter Plot with Cluster Labels
            scatter_1 = ax1.scatter(coordinates_1['X'], coordinates_1['Y'],
c=sorted_cluster_labels_1, cmap='viridis', marker='o', s=10)

            # Set plot properties for Plot 1
            ax1.set_title(f'2017 S2 Scatter Plot with Cluster Labels: Attempt {attempt},
Clusters {num_clusters}')
            ax1.set_xlabel('X Coordinate')
            ax1.set_ylabel('Y Coordinate')

            # Create a discrete color map with the number of clusters for Plot 1
            cmap_discrete_1 = matplotlib.colormaps.get_cmap('viridis')
```

```
            # Define boundaries for the discrete color map for Plot 1
            boundaries_1 = np.arange(-0.5, num_clusters_1, 1)

            # Create a BoundaryNorm for the color map for Plot 1
            norm_discrete_1 = mcolors.BoundaryNorm(boundaries_1, cmap_discrete_1.N,
clip=True)

            # Add a discrete color bar with integer cluster labels for Plot 1
            cbar_1 = plt.colorbar(scatter_1, ax=ax1, ticks=np.arange(num_clusters_1),
cmap=cmap_discrete_1, norm=norm_discrete_1, boundaries=boundaries_1)
            cbar_1.set_label('Cluster Label')

            # Set the number of tick marks on the X and Y axes for Plot 1
            ax1.xaxis.set_major_locator(MaxNLocator(nbins=3))
            ax1.yaxis.set_major_locator(MaxNLocator(nbins=3))

            # Plot 2: Line Plot of Unnormalized Cluster Centers
            for cluster_label_1 in range(num_clusters):
                color_1 = cmap_discrete_1(cluster_label_1 / (num_clusters - 1))  # Match
color from scatter plot
                ax2.plot(remaining_data_1.columns,
cluster_centers_original_scale_1[cluster_label_1], label=f'Cluster {cluster_label_1}',
color=color_1)
                ax2.set_ylim(remaining_data_1.min().min(), remaining_data_1.max().max())

            # Set plot properties for Plot 2
            ax2.set_title('2017 S2 Line Plot of Unnormalized Cluster Centers')
            ax2.set_xlabel('Column Name')
            ax2.set_ylabel('Cluster Center Spectrograph')
            ax2.legend()

            # Save the plots
            plot_filename_1 = f'2017 S2 kmeans_plots_Attempt_{attempt}
_Num_Clusters_{num_clusters}.png'
            plot_filepath_1 = os.path.join(plot_filename_1)
            plt.tight_layout()
            plt.savefig(plot_filepath_1)
            #plt.show()  # Display the plot
            images_attempt_1.append(plot_filepath_1)  # Append the plot to the list
            plt.close()

            # Tellus RM Create a scatter plot of X, Y, and final cluster label
            fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

            # Plot 1: Scatter Plot with Cluster Labels
            scatter_2 = ax1.scatter(coordinates_2['X'], coordinates_2['Y'],
c=sorted_cluster_labels_2, cmap='viridis', marker='o', s=10)

            # Set plot properties for Plot 1
            ax1.set_title(f'Tellus RM Scatter Plot with Cluster Labels: Attempt {attempt},
Clusters {num_clusters}')
            ax1.set_xlabel('X Coordinate')
            ax1.set_ylabel('Y Coordinate')

            # Create a discrete color map with the number of clusters for Plot 1
```

```
            cmap_discrete_2 = matplotlib.colormaps.get_cmap('viridis')

            # Define boundaries for the discrete color map for Plot 1
            boundaries_2 = np.arange(-0.5, num_clusters_2, 1)

            # Create a BoundaryNorm for the color map for Plot 1
            norm_discrete_2 = mcolors.BoundaryNorm(boundaries_2, cmap_discrete_2.N,
clip=True)

            # Add a discrete color bar with integer cluster labels for Plot 1
            cbar_2 = plt.colorbar(scatter_2, ax=ax1, ticks=np.arange(num_clusters_2),
cmap=cmap_discrete_2, norm=norm_discrete_2, boundaries=boundaries_2)
            cbar_2.set_label('Cluster Label')

            # Set the number of tick marks on the X and Y axes for Plot 1
            ax1.xaxis.set_major_locator(MaxNLocator(nbins=3))
            ax1.yaxis.set_major_locator(MaxNLocator(nbins=3))

            # Plot 2: Line Plot of Unnormalized Cluster Centers
            for cluster_label_2 in range(num_clusters):
                color_2 = cmap_discrete_2(cluster_label_2 / (num_clusters - 1))  # Match
color from scatter plot
                ax2.plot(remaining_data_2.columns,
cluster_centers_original_scale_2[cluster_label_2], label=f'Cluster {cluster_label_2}',
color=color_2)
                ax2.set_ylim(normalized_data_2.min().min(), normalized_data_2.max().max())

            # Set plot properties for Plot 2
            ax2.set_title('Tellus Line Plot of Normalized Cluster Centers')
            ax2.set_xlabel('Column Name')
            ax2.set_ylabel('Cluster Center Spectrograph')
            ax2.legend()

            # Save the plots
            plot_filename_2 = f'Tellus RM kmeans_plots_Attempt_{attempt}
_Num_Clusters_{num_clusters}.png'
            plot_filepath_2 = os.path.join(plot_filename_2)
            plt.tight_layout()
            plt.savefig(plot_filepath_2)
            #plt.show()  # Display the plot
            images_attempt_2.append(plot_filepath_2)  # Append the plot to the list
            plt.close()

         # Convert the images for the current number of clusters to a GIF
        gif_filename_1 = f'2017 S2 kmeans_plots_Num_Clusters_{num_clusters}.gif'
        with imageio.get_writer(gif_filename_1, mode='I', fps=1, loop=0) as writer_attempt:
            for image_filename_1 in images_attempt_1:
                # Adjust the image filename to include the subfolder
                image = imageio.imread(image_filename_1)
                writer_attempt.append_data(image)

                # Remove individual plot files after adding to GIF
                os.remove(image_filename_1)

        # Save the GIF to the current cluster folder
```

```
        zip_file.write(gif_filename_1)

        # Remove the GIF file after adding to the zip file
        os.remove(gif_filename_1)

         # Convert the images for the current number of clusters to a GIF
        gif_filename_2 = f'Tellus RM kmeans_plots_Num_Clusters_{num_clusters}.gif'
        with imageio.get_writer(gif_filename_2, mode='I', fps=1, loop=0) as writer_attempt:
            for image_filename_2 in images_attempt_2:
                # Adjust the image filename to include the subfolder
                image = imageio.imread(image_filename_2)
                writer_attempt.append_data(image)

                # Remove individual plot files after adding to GIF
                os.remove(image_filename_2)

        # Save the GIF to the current cluster folder
        zip_file.write(gif_filename_2)

        # Remove the GIF file after adding to the zip file
        os.remove(gif_filename_2)

        ### Section 7.3 Calculate MCASD metrics ###

        # Calculate Standard Deviation along each row
        row_std_dev_1 = distances_df_1.std(axis=1)
        row_std_dev_2 = distances_df_2.std(axis=1)

        # Calculate Average of Standard Deviation for all Rows
        avg_std_dev_1 = row_std_dev_1.mean()
        avg_std_dev_2 = row_std_dev_2.mean()

        # Calculate Standard Deviation of the first Standard Deviation for all rows
        error_1 = row_std_dev_1.std(axis=0)
        error_2 = row_std_dev_2.std(axis=0)

        # Save values in the mcasd_metrics_df DataFrame
        mcasd_metrics_df_1.at['MCASD Metric', num_clusters] = avg_std_dev_1
        mcasd_metrics_df_2.at['MCASD Metric', num_clusters] = avg_std_dev_2
        mcasd_metrics_df_1.at['MCASD Error', num_clusters] = error_1
        mcasd_metrics_df_2.at['MCASD Error', num_clusters] = error_2

        ### Section 7.4 Save results for Cluster number to a CSV ###

        # Output file names
        output_cluster_filename_1 = f'{file_name_without_extension_1}_kmeans_{num_clusters}
_cluster_data.csv'
        output_cluster_filename_2 = f'{file_name_without_extension_2}_kmeans_{num_clusters}
_cluster_data.csv'
        output_center_filename_1 = f'{file_name_without_extension_1}_kmeans_{num_clusters}
_cluster_centers.csv'
        output_center_filename_2 = f'{file_name_without_extension_2}_kmeans_{num_clusters}
_cluster_centers.csv'

        # Create a DataFrame with X, Y, Cluster, and Remaining Data
```

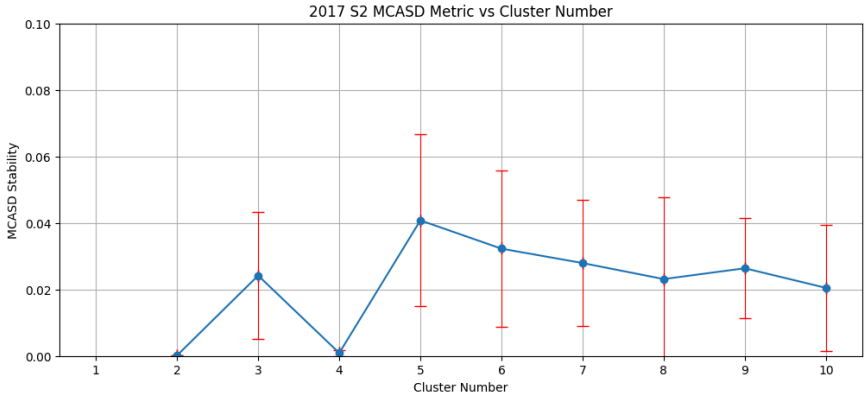
```
        clustered_data_df_1 = pd.DataFrame({
            'X': coordinates_1['X'],
            'Y': coordinates_1['Y'],
            'Cluster Number': sorted_cluster_labels_1,
            'MCASD Metric': row_std_dev_1,
            **{f'{col}': remaining_data_1[col] for col in remaining_data_1.columns}
        })

        clustered_data_df_2 = pd.DataFrame({
            'X': coordinates_2['X'],
            'Y': coordinates_2['Y'],
            'Cluster Number': sorted_cluster_labels_2,
            'MCASD Metric': row_std_dev_2,
            **{f'{col}': remaining_data_2[col] for col in remaining_data_2.columns}
        })
        clustered_data_df_1 = clustered_data_df_1.round(4)
        clustered_data_df_2 = clustered_data_df_2.round(4)

        # Save the DataFrame to a CSV file
        clustered_data_df_1.to_csv(output_cluster_filename_1, index=False)
        clustered_data_df_2.to_csv(output_cluster_filename_2, index=False)

        # Create a DataFrame with Cluster center data
        center_data_df_1 = pd.DataFrame(cluster_centers_original_scale_1,
columns=remaining_data_1.columns)
        center_data_df_2 = pd.DataFrame(cluster_centers_original_scale_2,
columns=remaining_data_2.columns)

        # Add a new column 'Cluster Number' to indicate the cluster number for each row
        center_data_df_1.insert(0, 'Cluster Number', range(num_clusters))
        center_data_df_2.insert(0, 'Cluster Number', range(num_clusters))
        center_data_df_1 = center_data_df_1.round(4)
        center_data_df_2 = center_data_df_2.round(4)

        # Save the DataFrame to a CSV file
        center_data_df_1.to_csv(output_center_filename_1, index=False)
        center_data_df_2.to_csv(output_center_filename_2, index=False)

        # Save the csv to the current cluster folder
        zip_file.write(output_cluster_filename_1)
        zip_file.write(output_cluster_filename_2)
        zip_file.write(output_center_filename_1)
        zip_file.write(output_center_filename_2)

        # Remove the csv file after adding to the zip file
        os.remove(output_cluster_filename_1)
        os.remove(output_cluster_filename_2)
        os.remove(output_center_filename_1)
        os.remove(output_center_filename_2)

    # Save mcasd_metrics_df to a CSV file
    mcasd_metrics_csv_filename_1 = '2017_S2_mcasd_metrics.csv'
    mcasd_metrics_csv_filename_2 = 'Tellus_RM_mcasd_metrics.csv'
    mcasd_metrics_df_1.to_csv(mcasd_metrics_csv_filename_1)
    mcasd_metrics_df_2.to_csv(mcasd_metrics_csv_filename_2)
```

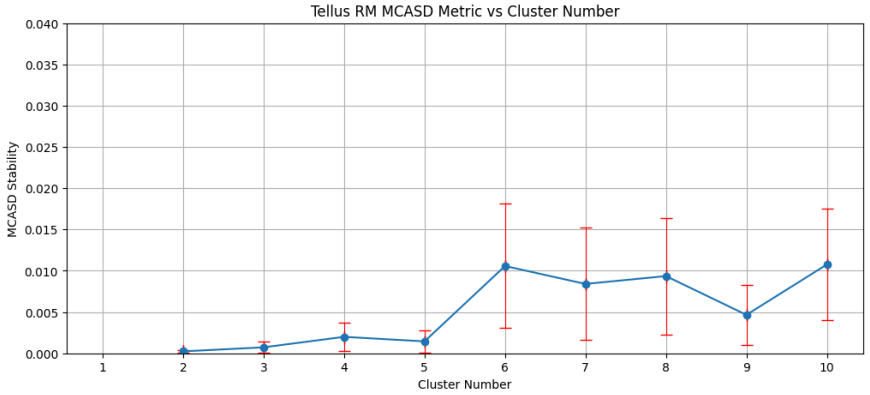
```
    # Add mcasd_metrics CSV file to the zip file
    zip_file.write(mcasd_metrics_csv_filename_1)
    zip_file.write(mcasd_metrics_csv_filename_2)

    # Remove the mcasd_metrics CSV file after adding to the zip file
    os.remove(mcasd_metrics_csv_filename_1)
    os.remove(mcasd_metrics_csv_filename_2)

    ### Section 7.5 Create MCASD metric plot for visual QC ###
    fig = plt.subplots(1, 1, figsize=(12, 5))
    # Make a 2D Line plot for 2017 S2 Data
    plt.errorbar(mcasd_metrics_df_1.columns, mcasd_metrics_df_1.loc['MCASD Metric'],
                 yerr=mcasd_metrics_df_1.loc['MCASD Error'], xerr=0, fmt='-o', capsize=5,
ecolor='red', errorevery=1,
                 elinewidth=0.8)
    plt.xlabel('Cluster Number')
    plt.ylabel('MCASD Stability')
    plt.title('2017 S2 MCASD Metric vs Cluster Number')
    plt.ylim(0, 0.1)  # Set Y Axis starting at 0
    plt.xticks(np.arange(1, max_num_clusters + 1, 1))  # Set X Tick marks at all integers
    plt.grid(True)

    # Save the 2D line plot to the zip file
    line_plot_filename_1 = '2017_S2_mcasd_line_plot.png'
    plt.savefig(line_plot_filename_1)
    zip_file.write(line_plot_filename_1)
    os.remove(line_plot_filename_1)  # Remove the saved file after adding to the zip file

    fig = plt.subplots(1, 1, figsize=(12, 5))
    # Make a 2D Line plot for 2018 S2 Data
    plt.errorbar(mcasd_metrics_df_2.columns, mcasd_metrics_df_2.loc['MCASD Metric'],
                 yerr=mcasd_metrics_df_2.loc['MCASD Error'], xerr=0, fmt='-o', capsize=5,
ecolor='red', errorevery=1,
                 elinewidth=0.8)
    plt.xlabel('Cluster Number')
    plt.ylabel('MCASD Stability')
    plt.title('Tellus RM MCASD Metric vs Cluster Number')
    plt.ylim(0, 0.04)  # Set Y Axis starting at 0
    plt.xticks(np.arange(1, max_num_clusters + 1, 1))  # Set X Tick marks at all integers
    plt.grid(True)

    # Save the 2D line plot to the zip file
    line_plot_filename_2 = '2018_S2_mcasd_line_plot.png'
    plt.savefig(line_plot_filename_2)
    zip_file.write(line_plot_filename_2)
    os.remove(line_plot_filename_2)  # Remove the saved file after adding to the zip file

# Inform the user that the MCASD Method clustering is complete
print("\nMCASD Method clustering complete.")

### Section 7.6 Save the final zip file to the user's local machine ###

# Move the zip file to the user's local machine
files.download(zip_filename)
```

```
Enter the maximum number of clusters for MCASD Method: 10
Enter the maximum number of attempts for MCASD Method: 10

Calculating MCASD Metrics...

MCASD Method clustering complete.
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```





5.9. Section 8

In this section, users will look at a combined analysis of Satellite S2 and Tellus Radiometric data.

As these data measure difference properties and different depths, an integrated analysis can provide a combined surface/subsurface view of this study site.

This code will perform MCASD analysis on the combined preprocessed Satellite and Tellus Radiometric data.

The code loops through different cluster numbers, applying K-Means clustering multiple times to analyze stability.

The results include GIFs illustrating scatter plots and line plots for each attempt. Additionally, MCASD metrics are calculated, providing insights into the stability and consistency of clusters. A line plot visualizes the MCASD metric across various cluster numbers, aiding participants in selecting the optimal cluster count.

Note that the Tellus Radiometric data are plotted as normalised as the Total Count data have significantly different dynamic range to the other data layers.

All results, including plots and metrics, are compressed into a zip file for easy download.

Please save this ZIP file to “Part 5” of the data directory you were provided with.

The ultimate goal is to assist participants in making informed decisions about the optimal number of clusters for their specific datasets.

When running the cell, you will be prompted to enter the number of max number of clusters (i.e., 10) and max attempts (i.e., 10).

MCASD will be applied to both 2017 S2 and Tellus RM datasets separately and MCASD plots will be displayed at the end.

Note this might take some time depending on the max number of clusters and max attempts chosen.

```
# @title
#### Section 8 MCASD Method ####
### Section 8.1 Get information from the user ###

# Display the Remaining Data DataFrame
print("\n2017 S2 + Tellus RM Remaining Data:")
print(combined_remaining_data.head(3).to_string(index=False))  # Use to_string to prevent
truncation

# Display the normalized data
print("\n2017 S2 + Tellus RM Normalized Data:")
print(combined_normalized_data.head(3).to_string(index=False))  # Use to_string to prevent
truncation

# Prompt the user for the maximum number of clusters for MCASD Method
max_num_clusters = int(input("\nEnter the maximum number of clusters for MCASD Method: "))
# Prompt the user for the maximum number of attempts for MCASD Method
max_attempts = int(input("Enter the maximum number of attempts for MCASD Method: "))


### Section 8.2 Loop for MCASD Method ###
# Create a DataFrame to store MCASD Metrics
mcasd_metrics_df_1 = pd.DataFrame(index=['MCASD Metric', 'MCASD Error'], columns=range(1,
max_num_clusters + 1))

print(f"\nCalculating MCASD Metrics...")

# Create a zip file to store all results
zip_filename = f'AgroGeo24_WS_Part_5_kmeans_combined_plots.zip'
with ZipFile(zip_filename, 'w') as zip_file:

    # Loop through the various number of clusters
    for num_clusters in range(2, max_num_clusters + 1):
        images_attempt_1 = []  # List to store images for the current attempt
        distances_df_1 = pd.DataFrame()  # Initialize distances DataFrame

        # Cluster the data a user specified number of times (Attempts)
        for attempt in range(1, max_attempts + 1):
            #print(f"\nNumber of Clusters: {num_clusters}: Attempt {attempt} of
```

```
{max_attempts}")

            # Initialize the K-Means model
            kmeans_1 = KMeans(n_clusters=num_clusters, n_init=1, init='k-means++')

            # Fit the K-Means model to the normalized data
            kmeans_1.fit(combined_normalized_data)

            # Get the cluster labels for each data point
            cluster_labels_1 = kmeans_1.labels_

            # Get the cluster centers
            cluster_centers_1 = kmeans_1.cluster_centers_

            ### Section 8.2.1 Sort the Cluster centers ###

            # Calculate the distances of cluster centers from the origin (0, 0)
            distances_from_origin_1 = np.sqrt(np.sum(cluster_centers_1 ** 2, axis=1))

            # Sort cluster centers based on distances from the origin
            sorted_indices_1 = np.argsort(distances_from_origin_1)

            # Sort cluster centers and labels
            sorted_cluster_centers_1 = cluster_centers_1[sorted_indices_1]
            sorted_cluster_labels_1 = np.zeros_like(cluster_labels_1)

            # Relabel the cluster labels based on the sorted order
            for new_label_1, old_label_1 in enumerate(sorted_indices_1):
                sorted_cluster_labels_1[cluster_labels_1 == old_label_1] = new_label_1

            ### Section 78.2.2 Calculate the distance (in the dataspace) between each
datapoint and its closest cluster center ###

            # Calculate the distances between cluster centers and data
            distances_1 = np.linalg.norm(combined_normalized_data.values[:, np.newaxis, :] -
cluster_centers_1, axis=-1)

            # Get the smallest distance for each data point
            min_distances_1 = np.min(distances_1, axis=1)

            # Create a DataFrame for distances with only the smallest distances
            new_column_1 = pd.DataFrame(min_distances_1, columns=[f'Attempt_{attempt}'])

            # Append the new column to the existing distances_df
            distances_df_1 = pd.concat([distances_df_1, new_column_1], axis=1)

            ### Section 8.2.3 Denormalize the cluster centers ###

            # Denormalize the Satellite data using the inverse transformation. Does not
denormalise the Tellus RM data
            num_columns_to_denormalize = 12

            # Denormalize the first 12 columns
            cluster_centers_original_scale_1 = sorted_cluster_centers_1.copy()
```

```
            # Extract relevant columns from min_vals_3
            min_vals_subset = min_vals_3.iloc[:num_columns_to_denormalize]
            max_vals_subset = max_vals_3.iloc[:num_columns_to_denormalize]

            # Denormalize using broadcasting
            cluster_centers_original_scale_1[:, :num_columns_to_denormalize] = (
                sorted_cluster_centers_1[:, :num_columns_to_denormalize] *
(max_vals_subset.values - min_vals_subset.values) + min_vals_subset.values
            )

            ### Section 8.2.4 Create and save plots for later GIF creation ###

            # 2017 S2 Create a scatter plot of X, Y, and final cluster label
            fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

            # Plot 1: Scatter Plot with Cluster Labels
            scatter_1 = ax1.scatter(coordinates_1['X'], coordinates_1['Y'],
c=sorted_cluster_labels_1, cmap='viridis', marker='o', s=10)

            # Set plot properties for Plot 1
            ax1.set_title(f'2017 S2 + Tellus RM Scatter Plot with Cluster Labels: Attempt
{attempt}, Clusters {num_clusters}')
            ax1.set_xlabel('X Coordinate')
            ax1.set_ylabel('Y Coordinate')

            # Create a discrete color map with the number of clusters for Plot 1
            cmap_discrete_1 = matplotlib.colormaps.get_cmap('viridis')

            # Define boundaries for the discrete color map for Plot 1
            boundaries_1 = np.arange(-0.5, num_clusters_1, 1)

            # Create a BoundaryNorm for the color map for Plot 1
            norm_discrete_1 = mcolors.BoundaryNorm(boundaries_1, cmap_discrete_1.N,
clip=True)

            # Add a discrete color bar with integer cluster labels for Plot 1
            cbar_1 = plt.colorbar(scatter_1, ax=ax1, ticks=np.arange(num_clusters_1),
cmap=cmap_discrete_1, norm=norm_discrete_1, boundaries=boundaries_1)
            cbar_1.set_label('Cluster Label')

            # Set the number of tick marks on the X and Y axes for Plot 1
            ax1.xaxis.set_major_locator(MaxNLocator(nbins=3))
            ax1.yaxis.set_major_locator(MaxNLocator(nbins=3))

            # Plot 2: Line Plot of Unnormalized Cluster Centers
            for cluster_label_1 in range(num_clusters):
                color_1 = cmap_discrete_1(cluster_label_1 / (num_clusters - 1))  # Match
color from scatter plot
                ax2.plot(combined_remaining_data.columns,
cluster_centers_original_scale_1[cluster_label_1], label=f'Cluster {cluster_label_1}',
color=color_1)
                ax2.set_ylim(0.0, 1.0)

            # Set plot properties for Plot 2
            ax2.set_title('2017 S2 + Tellus RM Line Plot of Normalized Cluster Centers')
```

```
            ax2.set_xlabel('Column Name')
            ax2.set_ylabel('Cluster Center Spectrograph')
            ax2.legend()

            # Save the plots
            plot_filename_1 = f'2017 S2 + Tellus RM kmeans_plots_Attempt_{attempt}
_Num_Clusters_{num_clusters}.png'
            plot_filepath_1 = os.path.join(plot_filename_1)
            plt.tight_layout()
            plt.savefig(plot_filepath_1)
            #plt.show()  # Display the plot
            images_attempt_1.append(plot_filepath_1)  # Append the plot to the list
            plt.close()

         # Convert the images for the current number of clusters to a GIF
        gif_filename_1 = f'2017 S2 + Tellus RM kmeans_plots_Num_Clusters_{num_clusters}.gif'
        with imageio.get_writer(gif_filename_1, mode='I', fps=1, loop=0) as writer_attempt:
            for image_filename_1 in images_attempt_1:
                # Adjust the image filename to include the subfolder
                image = imageio.imread(image_filename_1)
                writer_attempt.append_data(image)

                # Remove individual plot files after adding to GIF
                os.remove(image_filename_1)

        # Save the GIF to the current cluster folder
        zip_file.write(gif_filename_1)

        # Remove the GIF file after adding to the zip file
        os.remove(gif_filename_1)

        ### Section 8.3 Calculate MCASD metrics ###

        # Calculate Standard Deviation along each row
        row_std_dev_1 = distances_df_1.std(axis=1)

        # Calculate Average of Standard Deviation for all Rows
        avg_std_dev_1 = row_std_dev_1.mean()

        # Calculate Standard Deviation of the first Standard Deviation for all rows
        error_1 = row_std_dev_1.std(axis=0)

        # Save values in the mcasd_metrics_df DataFrame
        mcasd_metrics_df_1.at['MCASD Metric', num_clusters] = avg_std_dev_1
        mcasd_metrics_df_1.at['MCASD Error', num_clusters] = error_1

        ### Section 8.4 Save results for Cluster number to a CSV ###

        # Output file names
        output_cluster_filename_1 = f'{file_name_without_extension_1}_kmeans_{num_clusters}
_cluster_data.csv'
        output_center_filename_1 = f'{file_name_without_extension_1}_kmeans_{num_clusters}
_cluster_centers.csv'

        # Create a DataFrame with X, Y, Cluster, and Remaining Data
```

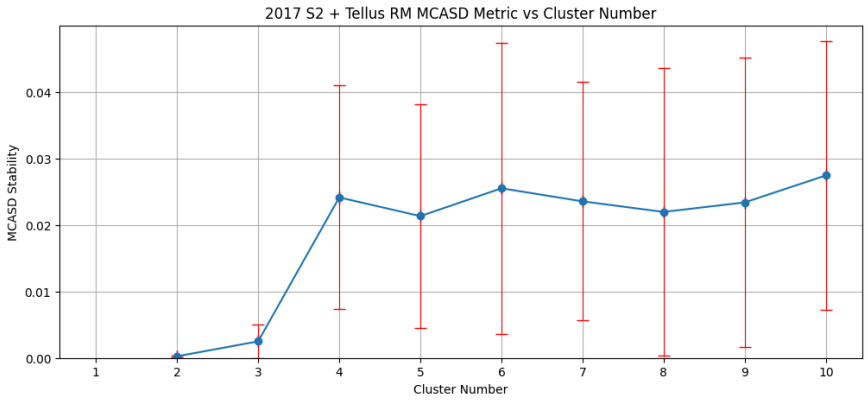
```
        clustered_data_df_1 = pd.DataFrame({
            'X': coordinates_1['X'],
            'Y': coordinates_1['Y'],
            'Cluster Number': sorted_cluster_labels_1,
            'MCASD Metric': row_std_dev_1,
            **{f'{col}': remaining_data_1[col] for col in remaining_data_1.columns}
        })

        clustered_data_df_1 = clustered_data_df_1.round(4)

        # Save the DataFrame to a CSV file
        clustered_data_df_1.to_csv(output_cluster_filename_1, index=False)

        # Create a DataFrame with Cluster center data
        center_data_df_1 = pd.DataFrame(cluster_centers_original_scale_1,
columns=combined_remaining_data.columns)

        # Add a new column 'Cluster Number' to indicate the cluster number for each row
        center_data_df_1.insert(0, 'Cluster Number', range(num_clusters))
        center_data_df_1 = center_data_df_1.round(4)

        # Save the DataFrame to a CSV file
        center_data_df_1.to_csv(output_center_filename_1, index=False)

        # Save the csv to the current cluster folder
        zip_file.write(output_cluster_filename_1)
        zip_file.write(output_center_filename_1)

        # Remove the csv file after adding to the zip file
        os.remove(output_cluster_filename_1)
        os.remove(output_center_filename_1)

    # Save mcasd_metrics_df to a CSV file
    mcasd_metrics_csv_filename_1 = '2017_S2_+_Tellus_RM_mcasd_metrics.csv'
    mcasd_metrics_df_1.to_csv(mcasd_metrics_csv_filename_1)

    # Add mcasd_metrics CSV file to the zip file
    zip_file.write(mcasd_metrics_csv_filename_1)

    # Remove the mcasd_metrics CSV file after adding to the zip file
    os.remove(mcasd_metrics_csv_filename_1)

    ### Section 8.5 Create MCASD metric plot for visual QC ###
    fig = plt.subplots(1, 1, figsize=(12, 5))
    # Make a 2D Line plot for 2017 S2 + Tellus RM Data
    plt.errorbar(mcasd_metrics_df_1.columns, mcasd_metrics_df_1.loc['MCASD Metric'],
                 yerr=mcasd_metrics_df_1.loc['MCASD Error'], xerr=0, fmt='-o', capsize=5,
ecolor='red', errorevery=1,
                 elinewidth=0.8)
    plt.xlabel('Cluster Number')
    plt.ylabel('MCASD Stability')
    plt.title('2017 S2 + Tellus RM MCASD Metric vs Cluster Number')
    plt.ylim(0)  # Set Y Axis starting at 0
    plt.xticks(np.arange(1, max_num_clusters + 1, 1))  # Set X Tick marks at all integers
    plt.grid(True)
```

```
    # Save the 2D line plot to the zip file
    line_plot_filename_1 = '2017_S2_+_Tellus_RM_mcasd_line_plot.png'
    plt.savefig(line_plot_filename_1)
    zip_file.write(line_plot_filename_1)
    os.remove(line_plot_filename_1)  # Remove the saved file after adding to the zip file

# Inform the user that the MCASD Method clustering is complete
print("\nMCASD Method clustering complete.")

### Section 8.6 Save the final zip file to the user's local machine ###

# Move the zip file to the user's local machine
files.download(zip_filename)
```

```
2017 S2 + Tellus RM Remaining Data:
    B1     B2     B3     B4     B5     B6     B7     B8    B8A     B9    B11    B12   K_CPS
U_CPS  Th_CPS   TC_CPS
0.0307 0.0361 0.0655 0.0467 0.1144 0.3056 0.3908 0.3858 0.4550 0.0959 0.2055 0.0984 29.4484
16.0089  7.1108 524.3576
0.0304 0.0347 0.0639 0.0449 0.1124 0.3135 0.3988 0.3915 0.4604 0.0946 0.2065 0.0962 27.3273
15.1769  6.8440 484.4671
0.0293 0.0323 0.0592 0.0405 0.1056 0.3236 0.4176 0.4080 0.4769 0.0945 0.1987 0.0897 24.3830
13.6135  6.3852 430.2153

2017 S2 + Tellus RM Normalized Data:
      B1       B2       B3       B4       B5       B6       B7       B8      B8A       B9
B11      B12    K_CPS    U_CPS   Th_CPS   TC_CPS
0.224839 0.239003 0.522444 0.252372 0.570100 0.753894 0.731299 0.754738 0.771953 0.868056
0.413927 0.261135 0.466632 0.646639 0.596157 0.480646
0.218415 0.218475 0.502494 0.235294 0.556811 0.776267 0.748252 0.767298 0.782284 0.853009
0.416516 0.253233 0.453358 0.635811 0.587933 0.449984
0.194861 0.183284 0.443890 0.193548 0.511628 0.804871 0.788091 0.803658 0.813851 0.851852
0.396324 0.229885 0.434932 0.615464 0.573791 0.408284

Enter the maximum number of clusters for MCASD Method: 10
Enter the maximum number of attempts for MCASD Method: 10

Calculating MCASD Metrics...

MCASD Method clustering complete.
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```



5.10. Section 9

In this section, users will perform K-Means clustering on the preprocessed Satellite and Tellus Radiometric data.

The optimal number of clusters (k) are now known by referencing the results from the previous MCASD analysis (Section 8). After entering the desired number of clusters, the code applies K-Means clustering and displays key information.

The results include a count of occurrences for each cluster label via a plot of the classified image and "spectral" graph showing the cluster centers.

The scatter plot provides an overview of spatial distribution of the clusters, while the line plot illustrates how cluster center values vary across the survey

In this instance the Spectral plots are overlain in order to allow for combined interpretation.

The task for this section is to think about the physical meaning of the clustering results.

Hint: Pay particular attention to the Spectral plot and what each band is measuring, and how the attenuation of Radiometric signal is affected by peat.

When running the cell, you will be prompted to enter the number of clusters.

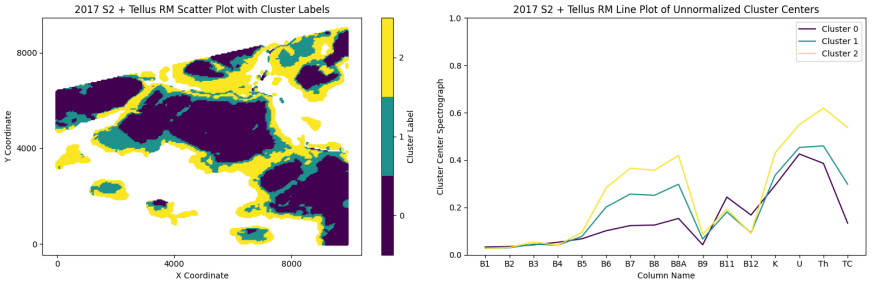
```
# @title
##### Section 9 K Means Clustering ####

### Section 9.1 Perform K Means Clustering ###

# Prompt the user to choose the number of clusters for K-Means
num_clusters_1 = int(input("Enter the number of clusters for combined 2017 S2 and Tellus RM
K-Means: "))

# Initialize the K-Means model
kmeans_1 = KMeans(n_clusters=num_clusters_1, init = 'k-means++', n_init=1)

# Fit the K-Means model to the normalized data
kmeans_1.fit(combined_normalized_data)

# Get the cluster labels for each data point
cluster_labels_1 = kmeans_1.labels_

# Get the cluster centers
```

```
cluster_centers_1 = kmeans_1.cluster_centers_

### Section 9.2 Sort the Cluster centers ###

# Calculate the distances of cluster centers from the origin (0, 0)
distances_from_origin_1 = np.sqrt(np.sum(cluster_centers_1 ** 2, axis=1))

# Sort cluster centers based on distances from the origin
sorted_indices_1 = np.argsort(distances_from_origin_1)

# Sort cluster centers and labels
sorted_cluster_centers_1 = cluster_centers_1[sorted_indices_1]
sorted_cluster_labels_1 = np.zeros_like(cluster_labels_1)

# Relabel the cluster labels based on the sorted order
for new_label_1, old_label_1 in enumerate(sorted_indices_1):
    sorted_cluster_labels_1[cluster_labels_1 == old_label_1] = new_label_1

# Calculate the count of each cluster label
cluster_labels_count_1 = dict(zip(*np.unique(sorted_cluster_labels_1, return_counts=True)))

### Section 9.3 Denormalize the cluster centers ###

# Denormalize the Satellite data using the inverse transformation. Does not denormalise the
Tellus RM data
num_columns_to_denormalize = 12

# Denormalize the first 12 columns
cluster_centers_original_scale_1 = sorted_cluster_centers_1.copy()

# Extract relevant columns from min_vals_3
min_vals_subset = min_vals_3.iloc[:num_columns_to_denormalize]
max_vals_subset = max_vals_3.iloc[:num_columns_to_denormalize]

# Denormalize using broadcasting
cluster_centers_original_scale_1[:, :num_columns_to_denormalize] = (
    sorted_cluster_centers_1[:, :num_columns_to_denormalize] * (max_vals_subset.values -
min_vals_subset.values) + min_vals_subset.values
)

### Section 9.4 Display Clustering counts for visual QC ###

# Display the count of each cluster label
print("\n2017 S2 + Tellus RM Count of Each Cluster Label:")
for label_1, count_1 in cluster_labels_count_1.items():
    print(f"Cluster {label_1}: {count_1} occurrences")

### Section 9.5 Save results to CSV ###

# Create a DataFrame with X, Y, Cluster, and Remaining Data
clustered_data_df_1 = pd.DataFrame({
    'X': coordinates_1['X'],
    'Y': coordinates_1['Y'],
    'Cluster Number': sorted_cluster_labels_1,
    **{f'{col}': combined_remaining_data[col] for col in combined_remaining_data.columns}
```

```
})

clustered_data_df_1 = clustered_data_df_1.round(4)

# Create a DataFrame with Cluster center data
center_data_df_1 = pd.DataFrame(cluster_centers_original_scale_1,
columns=combined_remaining_data.columns)


# Add a new column 'Cluster Number' to indicate the cluster number for each row
center_data_df_1.insert(0, 'Cluster Number', range(num_clusters_1))
center_data_df_1 = center_data_df_1.round(4)

### Section 9.6 Plot KMeans Clustering results ###

# 2017 S2 + Tellus RM Create a scatter plot of X, Y, and final cluster label
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))

# Plot 1: Scatter Plot with Cluster Labels
scatter_1 = ax1.scatter(coordinates_1['X'], coordinates_1['Y'], c=sorted_cluster_labels_1,
cmap='viridis', marker='o', s=10)

# Set plot properties for Plot 1
ax1.set_title('2017 S2 + Tellus RM Scatter Plot with Cluster Labels')
ax1.set_xlabel('X Coordinate')
ax1.set_ylabel('Y Coordinate')

# Create a discrete color map with the number of clusters for Plot 1
cmap_discrete_1 = matplotlib.colormaps.get_cmap('viridis')

# Define boundaries for the discrete color map for Plot 1
boundaries_1 = np.arange(-0.5, num_clusters_1, 1)

# Create a BoundaryNorm for the color map for Plot 1
norm_discrete_1 = mcolors.BoundaryNorm(boundaries_1, cmap_discrete_1.N, clip=True)

# Add a discrete color bar with integer cluster labels for Plot 1
cbar_1 = plt.colorbar(scatter_1, ax=ax1, ticks=np.arange(num_clusters_1),
cmap=cmap_discrete_1, norm=norm_discrete_1, boundaries=boundaries_1)
cbar_1.set_label('Cluster Label')

# Set the number of tick marks on the X and Y axes for Plot 1
ax1.xaxis.set_major_locator(MaxNLocator(nbins=3))
ax1.yaxis.set_major_locator(MaxNLocator(nbins=3))

# Plot 2: Line Plot of Unnormalized Cluster Centers
for cluster_label_1 in range(num_clusters_1):
    color_1 = cmap_discrete_1(cluster_label_1 / (num_clusters_1 - 1))  # Match color from
scatter plot
    ax2.plot(combined_remaining_data.columns,
cluster_centers_original_scale_1[cluster_label_1], label=f'Cluster {cluster_label_1}',
color=color_1)
    ax2.set_ylim(0.0, 1.0)

# Set plot properties for Plot 2
```

```
ax2.set_title('2017 S2 + Tellus RM Line Plot of Unnormalized Cluster Centers')
ax2.set_xlabel('Column Name')
ax2.set_ylabel('Cluster Center Spectrograph')
ax2.legend()

# Show the plots
plt.tight_layout()
plt.show()
```

```
Enter the number of clusters for combined 2017 S2 and Tellus RM K-Means: 3

2017 S2 + Tellus RM Count of Each Cluster Label:
Cluster 0: 8562 occurrences
Cluster 1: 3925 occurrences
Cluster 2: 4307 occurrences
```



6. Using Centroid Clustering on your own data!

This part of the workshop allows participants to try clustering on their own data, if they have brought some.

It should be a csv file, with format:

```
Column 1.   X Coordinate (X)
Column 2.   Y Coordinate (Y)
Column 3.   Data Layer 1
.
.
.
Column N.   Data Layer N
```

The aim of this workshop is to determine the appropriate number of clusters for your data using KMeans clustering. We will compare some tradition methods with a recently proposed MCASD method.

No coding experience is required to run this code. All the code contains comments describing what each line does. Please click “Show Code” on any section to view the code.

Please feel free to ask questions if you don’t understand any parts.

6.1. Section 0

This section sets up the Python environment for the clustering analysis.

It imports essential libraries such as Pandas for data manipulation, NumPy for numerical operations, Matplotlib for data visualization, scikit-learn for machine learning tools, and other supporting libraries.

Additionally, it configures the display.

The code also imports specific functions and modules required for the clustering analysis, such as KMeans.

Finally, it sets up tools for working with images, zip files, and file uploads in Google Colab. This preparation ensures that the subsequent code can efficiently perform clustering analysis and handle related tasks.

```
# @title
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
import imageio.v2 as imageio
import matplotlib
import imageio.v2 as imageio
import os
import re

from sklearn.cluster import KMeans
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import normalize
from sklearn.preprocessing import scale
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_score
from sklearn.metrics import pairwise_distances_argmin_min
from collections import Counter
from sklearn.preprocessing import MinMaxScaler
from matplotlib.ticker import MaxNLocator
from PIL import Image
from zipfile import ZipFile
from google.colab import files


# Set display options to show all rows and columns
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)
```

6.2. Section 1

Now we are going to read in the data from your local device.

The Data should be in CSV format with the following header descriptors:

1. Column 1 = X Coordinate
2. Column 2 = Y Coordinate
3. Columns 3..N = Data

Data will be rounded to 4 decimal places. (Change this in the code if necessary)

Press the play button below to select the correct file from your computer.

You will see a button “Choose files”. Click this and navigate to where this file is stored on your machine. Then highlight it and click “open”.

You will see a progress message as the file is uploaded to this Google Colab environment.

```
# @title
#### Section 1: Import CSV file  ####

# Prompt the user to upload a file
uploaded = files.upload()

# Get the uploaded file name
file_name = list(uploaded.keys())[0]

# Extract the filename without the extension
file_name_without_extension = os.path.splitext(file_name)[0]

# Remove numerical suffixes from the filename
file_name_without_extension = re.sub(r'\(\d+\)', '', file_name_without_extension)
file_name_without_extension = re.sub(r'\(\ \)', '', file_name_without_extension)

# Read the CSV file into a DataFrame
df = pd.read_csv(file_name).round(4)
```

Excellent. You have now loaded the CSV file into the Google Colab environment and are ready to take a look at the data.

6.3. Section 2

In this section, we're inspecting and refining the dataset. Initially, we showcase a snippet of the original dataset, including its structure and the number of rows.

Next, we perform data cleaning by eliminating rows containing NaN (Not a Number) and blank values. The cleaned dataset is then displayed, again with a snippet and the updated row count.

This process ensures that the dataset is well-prepared for subsequent analyses by removing any instances of missing or empty data.

Note that this dataset contained no blank values, so the number of rows doesn't change

```
# @title
#### Section 2: Data Cleaning ####

# Display the original DataFrame and its shape
print("Original Data:")
print(df.head(5).to_string(index=False))
print(f"Number of rows before cleaning: {df.shape[0]}")

# Remove rows with NaN and blank values
df_cleaned = df.dropna().replace('', np.nan).dropna()

# Display the cleaned DataFrame and its shape
print("\nData after removing NaN and blank values:")
print(df_cleaned.head(5).to_string(index=False))
print(f"Number of rows after cleaning: {df_cleaned.shape[0]}")
```

6.4. Section 3

Building on the cleaned dataset from the previous section, we now visualize the data columns through scatter plots.

Each subplot represents a specific data column, showcasing its spatial distribution across the X and Y coordinates. The color intensity in each plot reflects the values of the corresponding data column.

The number of rows and columns for the subplot grid is dynamically calculated based on the available data columns.

This visualization provides an initial exploration of how different data layers are distributed in geographical space, setting the stage for further analysis and insights.

```
# @title
#### Section 3 Data Viewing ###

# Get the list of non-coordinate column names
data_column_names = df_cleaned.columns[2:]

# Get the list of non-coordinate column names
data_column_names = df_cleaned.columns[2:]

# Calculate the number of rows and columns for subplots
num_plots = len(data_column_names)
num_plots_per_row = 3
num_rows = (num_plots + num_plots_per_row - 1) // num_plots_per_row
num_cols = min(num_plots, num_plots_per_row)

# Create subplots with the specified number of rows and columns
fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, 5))

# Flatten the axes to simplify indexing
axes = axes.flatten()

# Loop through each data column and create scatter plots
for i, column_name in enumerate(data_column_names):
    # Extract the data for the current column
    column_data = df_cleaned[column_name]

    # Calculate the position in the subplot grid
    row_index = i // num_cols
    col_index = i % num_cols

    # Create a scatter plot
    scatter = axes[i].scatter(df_cleaned['X'], df_cleaned['Y'], c=column_data,
cmap='viridis', marker='o', s=10)

    # Set plot properties
    axes[i].set_title(f'Scatter Plot for {column_name}')
    axes[i].set_xlabel('X Coordinate')
    axes[i].set_ylabel('Y Coordinate')

    # Set the number of tick marks on the X and Y axes
    axes[i].xaxis.set_major_locator(MaxNLocator(nbins=3))
    axes[i].yaxis.set_major_locator(MaxNLocator(nbins=3))

    # Add a color bar scaled to the min and max of the current column
    cbar = plt.colorbar(scatter, ax=axes[i])
    cbar.set_label(column_name)
```

```
# Adjust layout for better spacing
plt.tight_layout()

# Show the plots
plt.show()
```

6.5. Section 4

In this section, we strategically divide our dataset into two key components: coordinates and remaining data. The first two columns, containing coordinate information, are isolated to construct the “Coordinates” DataFrame.

Simultaneously, the remaining data, excluding the initial two columns, forms the “Remaining Data” DataFrame.

This separation serves a pivotal purpose—clustering analysis will solely operate on the remaining data. Subsequently, the cluster labels obtained can be associated with their respective X and Y coordinates. This distinction is fundamental for generating geographical cluster maps, allowing us to visually interpret and understand the spatial distribution of clusters across the dataset.

This section also focuses on data normalization, a crucial step before applying clustering algorithms.

You can choose the normalisation type here, column wise, or dataset wise.

The resulting normalized data is displayed, providing an insight into the standardized values across the dataset.

Normalization enhances the accuracy of clustering algorithms, ensuring that features with different scales contribute equally to the clustering process.

```
# @title
#### Section 4 Separate the Data for clustering ####

# Extract coordinate information (assuming it's in the first two columns)
coordinates = df_cleaned.iloc[:, :2]

# Extract the remaining data (excluding the first two columns)
remaining_data = df_cleaned.iloc[:, 2:]
remaining_data = remaining_data.round(2)

# Display the Coordinates DataFrame
print("\nCoordinates:")
print(coordinates.head(5).to_string(index=False))  # Use to_string to prevent truncation
print(f"Number of rows: {df_cleaned.shape[0]}")

# Display the Remaining Data DataFrame
print("\nRemaining Data:")
print(remaining_data.head(5).to_string(index=False))  # Use to_string to prevent truncation
print(f"Number of rows: {df_cleaned.shape[0]}")

# Prompt the user to choose the maximum number of clusters for the Elbow Method
norm_type = int(input("Select normalization type. 1 = Column-wise normalization. 2 =
Dataset-wise normalization: "))

if norm_type == 1:
    # Custom normalization using MinMaxScaler
```

```
    min_vals = remaining_data.min()
    max_vals = remaining_data.max()

    normalized_data = (remaining_data - min_vals) / (max_vals - min_vals)
elif norm_type == 2:
    # Custom normalization using MinMaxScaler
    min_vals = remaining_data.min().min()
    max_vals = remaining_data.max().max()

    normalized_data = (remaining_data - min_vals) / (max_vals - min_vals)
else:
    print("Invalid norm_type. Please use 1 or 2.")

# Round the normalized data to 4 decimal places
normalized_data = normalized_data.round(4)

# Convert the rounded normalized data back to a DataFrame and set column names
normalized_df = pd.DataFrame(normalized_data, columns=remaining_data.columns)

# Display the normalized data
print("\nNormalized Data:")
print(normalized_data.head(5).to_string(index=False))  # Use to_string to prevent truncation
print(f"Number of rows: {df_cleaned.shape[0]}")
```

6.6. Section 5

This section guides the user in determining the optimal number of clusters (k) for the K-Means algorithm by utilizing both the Elbow Method and Silhouette Scores.

After specifying the maximum number of clusters to consider, the code calculates the Within-Cluster Sum of Squares (WCSS) distance using the Elbow Method. The Elbow Method graph illustrates the trade-off between clustering complexity and WCSS reduction, helping identify an optimal k value.

Simultaneously, Silhouette Scores, a measure of how well-separated clusters are, are computed and presented on the same graph. Silhouette Scores range from −1 to 1, where higher scores indicate better-defined clusters.

When assessing the graph, users should look for the “elbow” point where WCSS plateaus, suggesting diminishing returns with additional clusters.

Additionally, a high Silhouette Score at the “elbow” reinforces the choice, ensuring a balance between compact clusters and distinct cluster boundaries for effective clustering

When running the cell, you will be prompted to enter the max number of clusters. (i.e., 10)

```
# @title
#### Section 6 Elbow Method with Silhouette Scores ####

# Prompt the user to choose the maximum number of clusters for the Elbow Method
max_clusters_elbow = int(input("Enter the maximum number of clusters for the Elbow Method:
"))

# Calculate the within-cluster sum of squares (WCSS) and Silhouette Scores for different
values of k
wcss = []
silhouette = []
for k in range(2, max_clusters_elbow + 1):
```

```
    kmeans = KMeans(n_clusters=k, n_init=1, init='k-means++')
    kmeans.fit(normalized_data)
    sscore = round(silhouette_score(normalized_data, kmeans.labels_), 2)
    silhouette.append(sscore)
    wcss.append(kmeans.inertia_)

# Plot the Elbow Method graph with Silhouette Scores as a bar graph
fig, ax1 = plt.subplots(figsize=(10, 6))

# Plot WCSS on the left y-axis
ax1.plot(range(2, max_clusters_elbow + 1), wcss, marker='o', color='blue', label='WCSS')
ax1.set_xlabel('Number of Clusters (k)')
ax1.set_ylabel('Within-Cluster Sum of Squares (WCSS)', color='blue')

# Set the x-axis ticks to show only integers
plt.xticks(range(2, max_clusters_elbow + 1))

# Create a second y-axis for Silhouette Scores
ax2 = ax1.twinx()
ax2.bar(range(2, max_clusters_elbow + 1), silhouette, color='green', alpha=0.5,
label='Silhouette Scores')
ax2.set_ylabel('Silhouette Scores', color='green')

# Add legend
lines, labels = ax1.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
ax2.legend(lines + lines2, labels + labels2, loc='upper right')

plt.title('Elbow Method with Silhouette Scores for Optimal k')
fig.tight_layout()
plt.show()
```

6.7. Section 6

In this section, users will perform K-Means clustering on the preprocessed EMI data.

The optimal number of clusters (k) can be determined by referencing the results from the previous Elbow Method and Silhouette Scores analysis (Section 5). After entering the desired number of clusters, the code applies K-Means clustering and displays key information.

The results include a count of occurrences for each cluster label via a plot of the classified image and “spectral” graph showing the cluster centers.

The scatter plot provides an overview of spatial distribution of the clusters, while the line plot illustrates how cluster center values vary across the survey.

You will be prompted to enter the number of clusters, which should be based on the Elbow and Silhouette results, but feel free to run this code a few times for various number of clusters and take a look at the results!

```
# @title
##### Section 6 K Means Clustering ####

### Section 6.1 Perform K Means Clustering ###

# Prompt the user to choose the number of clusters for K-Means
```

```
num_clusters = int(input("Enter the number of clusters for K-Means: "))

# Initialize the K-Means model
kmeans = KMeans(n_clusters=num_clusters, init = 'k-means++', n_init=1)

# Fit the K-Means model to the normalized data
kmeans.fit(normalized_data)

# Get the cluster labels for each data point
cluster_labels = kmeans.labels_

# Get the cluster centers
cluster_centers = kmeans.cluster_centers_

### Section 6.2 Sort the Cluster centers ###

# Calculate the distances of cluster centers from the origin (0, 0)
distances_from_origin = np.sqrt(np.sum(cluster_centers ** 2, axis=1))

# Sort cluster centers based on distances from the origin
sorted_indices = np.argsort(distances_from_origin)

# Sort cluster centers and labels
sorted_cluster_centers = cluster_centers[sorted_indices]
sorted_cluster_labels = np.zeros_like(cluster_labels)

# Relabel the cluster labels based on the sorted order
for new_label, old_label in enumerate(sorted_indices):
    sorted_cluster_labels[cluster_labels == old_label] = new_label

# Calculate the count of each cluster label
cluster_labels_count = dict(zip(*np.unique(sorted_cluster_labels, return_counts=True)))

### Section 6.3 Denormalize the cluster centers ###

# Denormalize the data using the inverse transformation
if norm_type == 1:
    cluster_centers_original_scale = sorted_cluster_centers * (max_vals.values -
min_vals.values) + min_vals.values
elif norm_type == 2:
    cluster_centers_original_scale = sorted_cluster_centers * (max_vals - min_vals) +
min_vals

### Section 6.4 Display Clustering counts for visual QC ###

# Display the count of each cluster label
print("\nCount of Each Cluster Label:")
for label, count in cluster_labels_count.items():
    print(f"Cluster {label}: {count} occurrences")

### Section 6.5 Save results to CSV ###

# Create a DataFrame with X, Y, Cluster, and Remaining Data
clustered_data_df = pd.DataFrame({
    'X': coordinates['X'],
```

```
    'Y': coordinates['Y'],
    'Cluster Number': sorted_cluster_labels,
    **{f'{col}': remaining_data[col] for col in remaining_data.columns}
})
clustered_data_df = clustered_data_df.round(4)

# Create a DataFrame with Cluster center data
center_data_df = pd.DataFrame(cluster_centers_original_scale,
columns=remaining_data.columns)

# Add a new column 'Cluster Number' to indicate the cluster number for each row
center_data_df.insert(0, 'Cluster Number', range(num_clusters))
center_data_df = center_data_df.round(4)

### Section 6.6 Plot KMeans Clustering results ###

# Create a scatter plot of X, Y, and final cluster label
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Plot 1: Scatter Plot with Cluster Labels
scatter = ax1.scatter(coordinates['X'], coordinates['Y'], c=sorted_cluster_labels,
cmap='viridis', marker='o', s=30)

# Set plot properties for Plot 1
ax1.set_title('Scatter Plot with Cluster Labels')
ax1.set_xlabel('X Coordinate')
ax1.set_ylabel('Y Coordinate')

# Create a discrete color map with the number of clusters for Plot 1
cmap_discrete = matplotlib.colormaps.get_cmap('viridis')

# Define boundaries for the discrete color map for Plot 1
boundaries = np.arange(-0.5, num_clusters, 1)

# Create a BoundaryNorm for the color map for Plot 1
norm_discrete = mcolors.BoundaryNorm(boundaries, cmap_discrete.N, clip=True)

# Add a discrete color bar with integer cluster labels for Plot 1
cbar = plt.colorbar(scatter, ax=ax1, ticks=np.arange(num_clusters), cmap=cmap_discrete,
norm=norm_discrete, boundaries=boundaries)
cbar.set_label('Cluster Label')

# Set the number of tick marks on the X and Y axes for Plot 1
ax1.xaxis.set_major_locator(MaxNLocator(nbins=3))
ax1.yaxis.set_major_locator(MaxNLocator(nbins=3))

# Plot 2: Line Plot of Unnormalized Cluster Centers
for cluster_label in range(num_clusters):
    color = cmap_discrete(cluster_label / (num_clusters - 1))  # Match color from scatter
plot
    ax2.plot(remaining_data.columns, cluster_centers_original_scale[cluster_label],
label=f'Cluster {cluster_label}', color=color)
    ax2.set_ylim(remaining_data.min().min(), remaining_data.max().max())

# Set plot properties for Plot 2
```

```
ax2.set_title('Line Plot of Unnormalized Cluster Centers')
ax2.set_xlabel('Column Name')
ax2.set_ylabel('Cluster Center Values)')
ax2.legend()

# Show the plots
plt.tight_layout()
plt.show()
```

6.8. Section 7

This section introduces the MCASD (Multiple Cluster Average Standard Deviation) method, designed to aid participants in identifying the optimal number of clusters for their dataset.

MCASD was first published as a method by O'Leary et al 2023.

MCASD evaluates the stability of cluster centers across multiple attempts and cluster numbers. Participants will input the maximum number of clusters and the maximum number of attempts per cluster.

The code loops through different cluster numbers, applying K-Means clustering multiple times to analyze stability.

The results include GIFs illustrating scatter plots and line plots for each attempt. Additionally, MCASD metrics are calculated, providing insights into the stability and consistency of clusters. A line plot visualizes the MCASD metric across various cluster numbers, aiding participants in selecting the optimal cluster count.

All results, including plots and metrics, are compressed into a zip file for easy download.

Please save this ZIP file to your computer.

The ultimate goal is to assist participants in making informed decisions about the optimal number of clusters for their specific dataset.

When running the cell, you will be prompted to enter the number of max number of clusters (i.e., 10) and max attempts (i.e., 10).

Note this might take some time depending on the max number of clusters and max attempts chosen.

```
# @title
#### Section 7 MCASD Method ####
### Section 7.1 Get information from the user ###

# Prompt the user for the maximum number of clusters for MCASD Method
max_num_clusters = int(input("Enter the maximum number of clusters for MCASD Method: "))
# Prompt the user for the maximum number of attempts for MCASD Method
max_attempts = int(input("Enter the maximum number of attempts for MCASD Method: "))

### Section 7.2 Loop for MCASD Method ###
# Create a DataFrame to store MCASD Metrics
mcasd_metrics_df = pd.DataFrame(index=['MCASD Metric', 'MCASD Error'], columns=range(1,
max_num_clusters + 1))

print(f"\nCalculating MCASD Metrics...")

# Create a zip file to store all results
zip_filename = f'AgroGeo24_WS_Part_3_kmeans_plots.zip'
```

```
with ZipFile(zip_filename, 'w') as zip_file:

    # Loop through the various number of clusters
    for num_clusters in range(1, max_num_clusters + 1):
        images_attempt = []  # List to store images for the current attempt
        distances_df = pd.DataFrame()  # Initialize distances DataFrame

        # Cluster the data a user specified number of times (Attempts)
        for attempt in range(1, max_attempts + 1):
            #print(f"\nNumber of Clusters: {num_clusters}: Attempt {attempt} of
{max_attempts}")

            # Initialize the K-Means model
            kmeans = KMeans(n_clusters=num_clusters, n_init=1, init='k-means++')

            # Fit the K-Means model to the normalized data
            kmeans.fit(normalized_data)

            # Get the cluster labels for each data point
            cluster_labels = kmeans.labels_

            # Get the cluster centers
            cluster_centers = kmeans.cluster_centers_

            ### Section 7.2.1 Sort the Cluster centers ###

            # Calculate the distances of cluster centers from the origin (0, 0)
            distances_from_origin = np.sqrt(np.sum(cluster_centers ** 2, axis=1))

            # Sort cluster centers based on distances from the origin
            sorted_indices = np.argsort(distances_from_origin)

            # Sort cluster centers and labels
            sorted_cluster_centers = cluster_centers[sorted_indices]
            sorted_cluster_labels = np.zeros_like(cluster_labels)

            # Relabel the cluster labels based on the sorted order
            for i, new_label in enumerate(np.arange(num_clusters)):
                old_label = sorted_indices[i]
                sorted_cluster_labels[cluster_labels == old_label] = new_label

            ### Section 7.2.2 Calculate the distance (in the dataspace) between each
datapoint and its closest cluster center ###

            # Calculate the distances between cluster centers and data
            distances = np.linalg.norm(normalized_data.values[:, np.newaxis, :] -
cluster_centers, axis=-1)

            # Get the smallest distance for each data point
            min_distances = np.min(distances, axis=1)

            # Create a DataFrame for distances with only the smallest distances
            new_column = pd.DataFrame(min_distances, columns=[f'Attempt_{attempt}'])

            # Append the new column to the existing distances_df
```

```
            distances_df = pd.concat([distances_df, new_column], axis=1)

            ### Section 7.2.3 Denormalize the cluster centers ###

            # Denormalize the data using the inverse transformation
            if norm_type == 1:
                cluster_centers_original_scale = sorted_cluster_centers * (max_vals.values -
min_vals.values) + min_vals.values
            elif norm_type == 2:
                cluster_centers_original_scale = sorted_cluster_centers * (max_vals -
min_vals) + min_vals

            ### Section 7.2.4 Create and save plots for later GIF creation ###

            # Create a scatter plot of X, Y, and final cluster label
            fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

            # Plot 1: Scatter Plot with Cluster Labels
            scatter = ax1.scatter(coordinates['X'], coordinates['Y'],
c=sorted_cluster_labels, cmap='viridis',
                                  marker='o', s=30)

            # Set plot properties for Plot 1
            ax1.set_title(f'Scatter Plot with Cluster Labels (Attempt {attempt}, Clusters
{num_clusters})')
            ax1.set_xlabel('X Coordinate')
            ax1.set_ylabel('Y Coordinate')

            # Create a discrete color map with the number of clusters for Plot 1
            cmap_discrete = matplotlib.colormaps.get_cmap('viridis')

            # Define boundaries for the discrete color map for Plot 1
            boundaries = np.arange(-0.5, num_clusters, 1)

            # Create a BoundaryNorm for the color map for Plot 1
            norm_discrete = mcolors.BoundaryNorm(boundaries, cmap_discrete.N, clip=True)

            # Add a discrete color bar with integer cluster labels for Plot 1
            cbar = plt.colorbar(scatter, ax=ax1, ticks=np.arange(num_clusters),
cmap=cmap_discrete, norm=norm_discrete,
                                boundaries=boundaries)
            cbar.set_label('Cluster Label')

            # Set the number of tick marks on the X and Y axes for Plot 1
            ax1.xaxis.set_major_locator(MaxNLocator(nbins=3))
            ax1.yaxis.set_major_locator(MaxNLocator(nbins=3))

            # Plot 2: Line Plot of Unnormalized Cluster Centers
            for cluster_label in range(num_clusters):
                color = cmap_discrete(cluster_label / (num_clusters))  # Match color from
scatter plot
                ax2.plot(remaining_data.columns,
cluster_centers_original_scale[cluster_label],
                         label=f'Cluster {cluster_label}', color=color)
                ax2.set_ylim(remaining_data.min().min(), remaining_data.max().max())
```

```
            # Set plot properties for Plot 2
            ax2.set_title(f'Line Plot of Cluster Centers (Attempt {attempt}, Clusters
{num_clusters})')
            ax2.set_xlabel('Column Name')
            ax2.set_ylabel('Cluster Center Value')
            ax2.legend()

            # Save the plots
            plot_filename = f'kmeans_plots_Attempt_{attempt}
_Num_Clusters_{num_clusters}.png'
            plot_filepath = os.path.join(plot_filename)
            plt.tight_layout()
            plt.savefig(plot_filepath)
            #plt.show()  # Display the plot
            images_attempt.append(plot_filepath)  # Append the plot to the list
            plt.close()

        # Convert the images for the current number of clusters to a GIF
        gif_filename = f'kmeans_plots_Num_Clusters_{num_clusters}.gif'
        with imageio.get_writer(gif_filename, mode='I', fps=1, loop=0) as writer_attempt:
            for image_filename in images_attempt:
                # Adjust the image filename to include the subfolder
                image = imageio.imread(image_filename)
                writer_attempt.append_data(image)

                # Remove individual plot files after adding to GIF
                os.remove(image_filename)

        # Save the GIF to the current cluster folder
        zip_file.write(gif_filename)

        # Remove the GIF file after adding to the zip file
        os.remove(gif_filename)

        ### Section 7.3 Calculate MCASD metrics ###

        # Calculate Standard Deviation along each row
        row_std_dev = distances_df.std(axis=1)

        # Calculate Average of Standard Deviation for all Rows
        avg_std_dev = row_std_dev.mean()

        # Calculate Standard Deviation of the first Standard Deviation for all rows
        error = row_std_dev.std(axis=0)

        # Save values in the mcasd_metrics_df DataFrame
        mcasd_metrics_df.at['MCASD Metric', num_clusters] = avg_std_dev
        mcasd_metrics_df.at['MCASD Error', num_clusters] = error

        ### Section 7.4 Save results for Cluster number to a CSV ###

        # Output file names
        output_cluster_filename = f'{file_name_without_extension}_kmeans_{num_clusters}
_cluster_data.csv'
```

```
        output_center_filename = f'{file_name_without_extension}_kmeans_{num_clusters}
_cluster_centers.csv'

        # Create a DataFrame with X, Y, Cluster, and Remaining Data
        clustered_data_df = pd.DataFrame({
            'X': coordinates['X'],
            'Y': coordinates['Y'],
            'Cluster Number': sorted_cluster_labels,
            'MCASD Metric': row_std_dev,
            **{f'{col}': remaining_data[col] for col in remaining_data.columns}
        })
        clustered_data_df = clustered_data_df.round(4)

        # Save the DataFrame to a CSV file
        clustered_data_df.to_csv(output_cluster_filename, index=False)

        # Create a DataFrame with Cluster center data
        center_data_df = pd.DataFrame(cluster_centers_original_scale,
columns=remaining_data.columns)

        # Add a new column 'Cluster Number' to indicate the cluster number for each row
        center_data_df.insert(0, 'Cluster Number', range(num_clusters))
        center_data_df = center_data_df.round(4)

        # Save the DataFrame to a CSV file
        center_data_df.to_csv(output_center_filename, index=False)

        # Save the csv to the current cluster folder
        zip_file.write(output_cluster_filename)
        zip_file.write(output_center_filename)

        # Remove the csv file after adding to the zip file
        os.remove(output_cluster_filename)
        os.remove(output_center_filename)

    # Save mcasd_metrics_df to a CSV file
    mcasd_metrics_csv_filename = 'mcasd_metrics.csv'
    mcasd_metrics_df.to_csv(mcasd_metrics_csv_filename)

    # Add mcasd_metrics CSV file to the zip file
    zip_file.write(mcasd_metrics_csv_filename)

    # Remove the mcasd_metrics CSV file after adding to the zip file
    os.remove(mcasd_metrics_csv_filename)

    ### Section 7.5 Create MCASD metric plot for visual QC ###

    # Make a 2D Line plot
    plt.errorbar(mcasd_metrics_df.columns, mcasd_metrics_df.loc['MCASD Metric'],
                 yerr=mcasd_metrics_df.loc['MCASD Error'], xerr=0, fmt='-o', capsize=5,
ecolor='red', errorevery=1,
                 elinewidth=0.8)
    plt.xlabel('Cluster Number')
    plt.ylabel('MCASD Stability')
    plt.title('MCASD Metric vs Cluster Number')
```

```
    plt.ylim(0)  # Set Y Axis starting at 0
    plt.xticks(np.arange(1, max_num_clusters + 1, 1))  # Set X Tick marks at all integers
    plt.grid(True)

    # Save the 2D line plot to the zip file
    line_plot_filename = 'mcasd_line_plot.png'
    plt.savefig(line_plot_filename)
    zip_file.write(line_plot_filename)
    os.remove(line_plot_filename)  # Remove the saved file after adding to the zip file

# Inform the user that the MCASD Method clustering is complete
print("\nMCASD Method clustering complete.")

### Section 7.6 Save the final zip file to the user's local machine ###

# Move the zip file to the user's local machine
files.download(zip_filename)
```