Assignment: MTHM503J

I acknowledge the use of ChatGPT by openAI (https://chat.openai.com/) to generate materials that I have adapted to include within my final assessment. I confirm that no content generated by AI has been presented as my own work.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_wine
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression, LinearRegression, ElasticNet, LassoLars, BayesianRidge
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score, roc_auc_score,
    confusion_matrix, classification_report,
    mean_squared_error, r2_score,
)
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
from sklearn.pipeline import make_pipeline
from sklearn.svm import SVR
from sklearn.preprocessing import PolynomialFeatures
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from statsmodels.api import OLS
from statsmodels.formula.api import ols
from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
from scipy.spatial.distance import pdist, squareform, cdist
from pingouin import partial_corr
from scipy import linalg
from scipy.stats import poisson
import matplotlib.colors as mcolors
from matplotlib.lines import Line2D
#i assumed youd want to see the code as i went along hence why its not hidden.
```

A. Dimensionality reduction for Wine Data

(i) Perform z-score scaling for each feature in the wine dataset and print the first 5 rows of the transformed dataset. Split the dataset into training and testing sets retaining 80% for training and 20% for testing.

In [2]:
```
# Load the wine data
wine = load_wine()
wine_data = wine['data']
wine_target = wine['target']
wine_feature_names = wine['feature_names']

# Creating a DataFrame from the loaded data
wine_df = pd.DataFrame(wine_data, columns=wine_feature_names)
wine_df['target'] = wine_target

# Selecting the features for scaling (excluding the 'target' column)
features = wine_df.drop('target', axis=1)

# Initializing the StandardScaler
scaler = StandardScaler()

# Applying z-score scaling
scaled_features = scaler.fit_transform(features)

# Creating a new DataFrame for the scaled data
scaled_df = pd.DataFrame(scaled_features, columns=features.columns)

# Adding the 'target' column back to the scaled DataFrame
scaled_df['target'] = wine_df['target']

# Replace underscores with spaces in column names
scaled_df.columns = [col.replace('_', ' ').title() for col in scaled_df.columns]

# Generate the HTML for the table
table_html = scaled_df.head(5).to_html(index=False)

# Add a caption below the table with specified styles
caption_html = '<caption style="caption-side: bottom; text-align: left; font-style: italic; color: black;">Tabl

# Combine the table HTML and the caption
combined_html = table_html.replace('<table border="1" class="dataframe">', '<table border="1" class="dataframe"
```

```
# Display the table with the caption
display(HTML(combined_html))
```

In [ ]:
```
# Separating features and the target variable
X = scaled_df.drop('Target', axis=1)
y = scaled_df['Target']

# Splitting the dataset into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=14012024)
```

(ii) Perform dimensionality reduction using PCA on the wine data set and choose the optimal number of retained principal components. Explain your choice.

In [ ]:
```
# Compute PCA
pcaCEV = PCA()
pcaCEV.fit(X_train)
explained_variance = pcaCEV.explained_variance_ratio_
cumulative_variance = np.cumsum(explained_variance)

# Create the cumulative variance table
cumulative_variance_table = pd.DataFrame({
    'Number of Components': range(1, len(cumulative_variance) + 1),
    'Cumulative Explained Variance': cumulative_variance
})

# Plot the cumulative explained variance with a caption to the left
fig, ax = plt.subplots(figsize=(10, 5))
ax.plot(range(1, len(cumulative_variance) + 1), cumulative_variance, marker='o', linestyle='-', color='black')
ax.set_title('Cumulative Explained Variance')
ax.set_xlabel('Number of Principal Components')
ax.set_ylabel('Cumulative Explained Variance')
ax.set_xticks(range(1, len(cumulative_variance) + 1))
ax.axhline(y=0.95, color='r', linestyle='--', label='95% Explained Variance')
ax.legend(loc='lower right')  # Move the legend to the bottom right corner

# Removing top and right lines from the graph
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)

# Adding caption a bit lower to the bottom left of the graph in italics
fig.text(0.05, -0.05, 'Figure 1: Cumulative Explained Variance', ha='left', va='bottom', fontsize=12, fontstyle

# Display the cumulative variance table with a caption
display(HTML('<i>Table 2: Cumulative Explained Variance</i>'))
display(HTML(cumulative_variance_table.to_html(index=False)))
plt.show()
```

In [ ]:
```
# Function to calculate the partial correlation matrix
def partial_corr(C):
    P_corr = np.zeros((C.shape[1], C.shape[1]))
    for i in range(C.shape[1]):
        for j in range(C.shape[1]):
            if i == j:
                P_corr[i, j] = 1
            else:
                subset = np.delete(C, [i, j], axis=1)
                C_i = C[:, i]
                C_j = C[:, j]

                beta_i = linalg.lstsq(subset, C_i, check_finite=False)[0]
                beta_j = linalg.lstsq(subset, C_j, check_finite=False)[0]

                res_i = C_i - subset.dot(beta_i)
                res_j = C_j - subset.dot(beta_j)

                P_corr[i, j] = np.corrcoef(res_i, res_j)[0, 1]
    return P_corr

# MAP test
n_components_range = range(1, min(X_train.shape[0], X_train.shape[1]) + 1)
average_squared_partial_correlations = []

for n_components in n_components_range:
    pca = PCA(n_components=n_components)
    data_pca = pca.fit_transform(X_train)
    residual_matrix = X_train - pca.inverse_transform(data_pca)

    # Convert the residual matrix to a NumPy array
    residual_matrix_np = np.array(residual_matrix)

    partial_correlation_matrix = partial_corr(residual_matrix_np)
    average_squared_partial_correlations.append(np.mean(np.square(partial_correlation_matrix[np.triu_indices_fr

# Plot the average squared partial correlations with caption and style adjustments
plt.figure(figsize=(10, 6))
plt.plot(n_components_range, average_squared_partial_correlations, marker='o', color='b')
```

```
plt.xlabel('Number of Components')
plt.ylabel('Average Squared Partial Correlation')
plt.title('Average Squared Partial Correlations')
plt.xticks(n_components_range)

# Removing top and right lines from the graph
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)

# Adding caption below the graph in italics
plt.figtext(0.1, -0.05, 'Figure 2: Mean Average Squared Partial Correlations', ha='left', va='center', fontsize

# Display the plot without grid lines and text for "Optimal number of components"
plt.grid(False)
plt.show()
```

In [ ]:
```
pcaLHM = PCA(n_components=13)
pcaLHM.fit(X_train)

# Retrieve the PCA loadings
pca_loadings = pcaLHM.components_

# Convert the loadings to a DataFrame for easier visualization
loadings_df = pd.DataFrame(pca_loadings, columns=X_train.columns, index=[f'PC{i+1}' for i in range(13)])

# Visualize the feature loadings using a heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(loadings_df, annot=True, cmap='coolwarm', vmin=-1, vmax=1)
plt.title('PCA Loadings Heatmap')
plt.ylabel('Principal Component')

# Adding caption below the heatmap
plt.figtext(0.25, -0.15, 'Figure 3: PCA Loadings Heatmap', ha='center', va='center', fontsize=12, fontstyle='it

plt.show()
```

In [ ]:
```
#I understand in a real situation i wouldnt be able to simulate the split over and over to get a more accruate

num_simulations = 100  # Number of simulations
max_components = X.shape[1]  # Maximum number of PCA components
f1_scores = np.zeros((num_simulations, max_components))

for sim in range(num_simulations):
    # Splitting the dataset into training (80%) and testing (20%) sets
    X_train_sim, X_test_sim, y_train_sim, y_test_sim = train_test_split(X, y, test_size=0.20, random_state=None

    for n_components in range(1, max_components + 1):
        pca_sim = PCA(n_components=n_components)

        # Transforming the training and testing sets
        X_train_pca_sim = pca_sim.fit_transform(X_train_sim)
        X_test_pca_sim = pca_sim.transform(X_test_sim)

        # Logistic Regression
        logistic_model = LogisticRegression()
        logistic_model.fit(X_train_pca_sim, y_train_sim)

        # Predicting and evaluating on the test set
        y_pred_sim = logistic_model.predict(X_test_pca_sim)
        f1 = f1_score(y_test_sim, y_pred_sim, average='weighted')  # Calculating weighted F1 score
        f1_scores[sim, n_components - 1] = f1

# Calculating the average F1 score for each number of PCA components
average_f1_scores = np.mean(f1_scores, axis=0)

# Plotting the results for F1 scores
plt.figure(figsize=(10, 6))
plt.plot(range(1, max_components + 1), average_f1_scores, marker='o')
plt.xlabel('Number of PCA Components')
plt.ylabel('Average F1 Score')
plt.title('Logistic Regression Performance (F1 Score) vs. Number of PCA Components')
plt.xticks(range(1, max_components + 1))

# Remove top and right borders
plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)

# Adding caption below the plot
plt.figtext(0.45, -0.05, 'Figure 4:Logistic Regression Performance (F1 Score) vs. Number of PCA Components', ha

plt.show()
```

In [ ]:
```
# Fit PCA with 7 components
pca = PCA(n_components=7)
pca.fit(X_train)

# Transform your data
```

```
X_train_pca = pca.transform(X_train)
# Transform your data
X_test_pca = pca.transform(X_test)
```

In PCA, selecting enough components to explain 95% of the variance is a commonly adopted criterion. However, in this case, due to the absence of a distinct 'elbow' or a significant jump in the explained variance as shown in Figure 1, alternative methods were employed to guide a more informed choice. Figure 2, illustrating the MAP (Minimum Average Partial) test, effectively visualizes how much significant information from the original dataset is retained with varying numbers of components. This graph is instrumental in determining the optimal number of components by minimizing reconstruction error, thereby ensuring that the reduced dataset preserves as much information as possible. Meanwhile, the cumulative explained variance graph complements this by indicating the proportion of total dataset variance captured by each principal component, thereby aiding in dimensionality reduction decisions based on variance explanation.

The PCA loadings heatmap (Figure 3) offers a data-driven perspective, particularly useful in assessing the necessity of each principal component. For instance, PC12, while receiving main loadings from two features, also shares these features' strong loadings with other components, suggesting its redundancy. Additionally, logistic regression accuracy, as depicted in Figure 4, points towards an optimal range of 5 to 9 components for accurately predicting wine cultivators.

Balancing these insights, 7 principal components were selected for the PCA. This choice strikes a harmonious balance: it accounts for 90% of the variance, maintains a relatively low average squared partial correlation, and does not compromise the logistic regression accuracy. Importantly, at 7 components, each variable exhibits high loadings in at least one component, ensuring that the dimensional reduction through PCA does not overly simplify the dataset to the detriment of the ultimate goal - accurate prediction of wine cultivators. This careful selection process highlights the criticality of combining various analytical perspectives to achieve a well-rounded and effective PCA application in predictive modelling.

(iii) Visualise the results of dimensionality reduction and comment on your findings. Consider using scatter plots or other relevant visualisations to showcase the reduced dimensionality

In [ ]:
```
# Create a custom colormap for scatter plot
custom_colors = ['#FF0000', '#000EFF', '#55ff00']
custom_cmap = plt.cm.colors.ListedColormap(custom_colors)

# Plot the biplot
plt.figure(figsize=(12, 10))
scatter = plt.scatter(X_train_pca[:, 0], X_train_pca[:, 1], c=y_train, cmap=custom_cmap, label='Data points')

# Add feature vectors as arrows with black color
for i, (component1, component2) in enumerate(zip(pca.components_[0], pca.components_[1])):
    plt.arrow(0, 0, component1 * 3, component2 * 3, color='k', alpha=0.7)  # Black arrows
    plt.text(component1 * 3.2, component2 * 3.2, X_train.columns[i], fontsize=12, color='k', ha='center')  # Bl

plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA Biplot')
plt.legend(handles=scatter.legend_elements()[0], labels=set(y_train), title="Cultivars", loc='upper right')
plt.grid(False)

# Add caption at the bottom
plt.figtext(0.2, 0.01, "Figure 5: Biplot of PCA", wrap=True, horizontalalignment='center', fontsize=12)

plt.show()
```

Figure 5 reveals several insights into the underlying structure of the data after dimensionality reduction. By projecting the high-dimensional data onto the first two principal components (PC1 and PC2), we can observe how different features influence these components. The orientation and length of the arrows in the biplot indicate the contribution and direction of influence of each original feature. Some vectors in the Figure 5 point in the same direction is quite significant. It indicates that these features share a similar pattern of variance in the dataset. In other words, when features have vectors pointing in the same direction, it suggests a positive correlation between them. They contribute to the principal components in a similar way, meaning that as one feature increases, the other tends to increase as well. This can be particularly insightful for understanding which features move together and may be part of the same underlying factor.

From the plot, it's evident that certain features have a stronger influence on PC1, while others are more aligned with PC2. This suggests that these principal components capture different aspects of the data's variance. For instance, features that have arrows pointing in similar directions are likely to be positively correlated. Observations (data points) in the plot are spread out, indicating variability in the data, which the PCA has captured effectively.

The distribution of points suggests possible clusters, which might correspond to different types of wine in the dataset. However, without color-coding the points by Cultivars, we can't conclusively determine the cluster boundaries. Overall, the PCA biplot provides a compact and informative visualization, highlighting the relationships between features and the natural groupings within the data, which can be invaluable for further analysis, such as clustering or classification tasks.

(iv) Construct a logistic regression to the PCA-transformed dataset. Choose a performance metric to evaluate the performance of your logistic regression model.

In [ ]:
```
# Create a pipeline with scaling and logistic regression
pipe = make_pipeline(StandardScaler(), LogisticRegression(max_iter=5000))

param_grid = [
    {
        'logisticregression__penalty': ['l1', 'l2'],
        'logisticregression__C': [0.001, 0.01, 0.1, 1, 10, 100],
        'logisticregression__solver': ['liblinear', 'saga']
    },
    {
        'logisticregression__penalty': ['elasticnet'],
        'logisticregression__C': [0.001, 0.01, 0.1, 1, 10, 100],
        'logisticregression__solver': ['saga'],
        'logisticregression__l1_ratio': [0.2, 0.5, 0.8]
    }
]

grid_search = GridSearchCV(pipe, param_grid, cv=5, scoring='accuracy', verbose=1)
grid_search1 = GridSearchCV(pipe, param_grid, cv=5, scoring='accuracy', verbose=1)

# Fit using only the first two PCA components
grid_search1.fit(X_train_pca[:, :2], y_train)

# Fit using all PCA components
grid_search.fit(X_train_pca, y_train)

# Best model from grid search
log_reg_model = grid_search.best_estimator_

# Predicting the test set results
y_pred = log_reg_model.predict(X_test_pca)
```

In [ ]:
```
def run_simulation(X, y, num_simulations=100, num_components=5):
    f1_scores_dict = {}

    for i in range(num_simulations):
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20)
        pca = PCA(n_components=num_components)
        X_train_pca = pca.fit_transform(X_train)
        X_test_pca = pca.transform(X_test)

        logreg = grid_search.best_estimator_
        logreg.fit(X_train_pca, y_train)

        y_pred = logreg.predict(X_test_pca)

        report = classification_report(y_test, y_pred, output_dict=True)
        for class_idx in report:
            if class_idx.isdigit():  # Check if key is a class label
                f1_score = report[class_idx]['f1-score']
                if class_idx not in f1_scores_dict:
                    f1_scores_dict[class_idx] = []
                f1_scores_dict[class_idx].append(f1_score)

    average_f1_scores = {class_idx: np.mean(scores) for class_idx, scores in f1_scores_dict.items()}
    combined_average_f1_score = np.mean(list(average_f1_scores.values()))

    average_f1_df = pd.DataFrame({
        "Cultivars": list(average_f1_scores.keys()) + ["Average"],
        "F1-Score": list(average_f1_scores.values()) + [combined_average_f1_score]
    })

    # Convert to HTML with caption "Table 3" at the bottom
    caption = "Table 3: Average F1-Scores"
    html_table = average_f1_df.to_html(index=False)
    # Insert the caption tag with style for bottom placement
    html_table_with_caption = html_table.replace('<table border="1" class="dataframe">', f'<table border="1" cl

    return html_table_with_caption

# Call the function to run simulations
average_f1_table_html = run_simulation(X, y)

# To display the HTML table in a Jupyter notebook, use:
from IPython.display import HTML
HTML(average_f1_table_html)
```

In [ ]:
```
# Best model
log_reg_model1 = grid_search1.best_estimator_

# Create a meshgrid for plotting decision boundaries
x_min, x_max = X_train_pca[:, 0].min() - 1, X_train_pca[:, 0].max() + 1
y_min, y_max = X_train_pca[:, 1].min() - 1, X_train_pca[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                     np.arange(y_min, y_max, 0.01))

# Predict classes for each point in the meshgrid
```

```
Z = log_reg_model1.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Define your custom colors
custom_colors = ['#FF0000', '#000EFF', '#55ff00']  # Tomato Red, Light Green, Light Sky Blue
custom_cmap = mcolors.ListedColormap(custom_colors)
custom_colors_train = ['#FF0000', '#000EFF', '#55ff00']  # LightSalmon, LightSeaGreen, LightSlateGray
custom_map1 = mcolors.ListedColormap(custom_colors_train)


plt.figure(figsize=(10, 10))
contourf = plt.contourf(xx, yy, Z, alpha=0.4, cmap=custom_cmap)

scatter_train = plt.scatter(X_train_pca[:, 0], X_train_pca[:, 1], c=y_train, cmap=custom_map1, edgecolors='k',
scatter_test = plt.scatter(X_test_pca[:, 0], X_test_pca[:, 1], c=y_test, cmap=custom_map1, edgecolors='k', mark

plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('Logistic Regression Decision Boundaries with PCA')

legend_elements = [Line2D([0], [0], marker='v', color='w', label='Training Data', markerfacecolor='black', mark
                   Line2D([0], [0], marker='o', color='w', label='Test Data', markerfacecolor='black', markersi

data_type_legend = plt.legend(handles=legend_elements, title="Data Type", loc='lower right')
plt.gca().add_artist(data_type_legend)

class_labels = [f'Class {l}' for l in np.unique(y_train)]
class_patches = [plt.matplotlib.patches.Patch(color=custom_cmap.colors[i], label=class_labels[i]) for i in rang
plt.legend(handles=class_patches, title="Cultivars", loc='upper right')

# Add the caption
plt.figtext(0.2, 0.01, "Figure 6: Decsion Boundaries of Logistic Regression", wrap=True, horizontalalignment='c

plt.show()
```

The Logistic Regression model, as visualized through Figure 6 and evaluated using F1 scores, demonstrates notable proficiency in classifying the different cultivars of the wine dataset. The decision boundaries, resulting from the model's application on the two principal components of the PCA-reduced feature space, reveal a clear and well-defined separation between the classes. This separation indicates the model's effectiveness in distinguishing between the various types of wine, which is further corroborated by the high F1 scores: 0.9745 for Cultivar 0, 0.9591 for Cultivar 1, and 0.9724 for Cultivar 2.

The average F1 score of 0.9687 across all classes highlights the model's balanced precision and recall, suggesting minimal confusion between classes and a high rate of correct classifications. The presence of both training and test data points within their respective decision regions in the plot validates the model's accuracy and its capability to generalize well to unseen data.

Overall, the combination of clear decision boundaries and consistently high F1 scores across all classes speaks to the robustness of the Logistic Regression model in this classification task. It effectively captures the underlying patterns in the data, enabling reliable and accurate predictions of wine cultivars.

B. What factors influence the agricultural yield?

In [ ]:
```
train_data = pd.read_csv(r"C:\Users\archi\Downloads\agricultural_yield_train.csv")
test_data =  pd.read_csv(r"C:\Users\archi\Downloads\agricultural_yield_test.csv")

#encode
train_data = pd.get_dummies(train_data, columns=['Irrigation_Schedule'],prefix=['Irrigation_Schedule'])
test_data = pd.get_dummies(test_data, columns=['Irrigation_Schedule'],prefix=['Irrigation_Schedule'])
# Check for missing values in the entire DataFrame
missing_values_train = train_data.isnull().sum()
missing_values_test = test_data.isnull().sum()
# Extracting data for the two different irrigation schedules
irrigation_schedule_2 = train_data[train_data['Irrigation_Schedule_2'] == 1]
irrigation_schedule_9 = train_data[train_data['Irrigation_Schedule_9'] == 1]
irrigation_schedule_2_test = test_data[test_data['Irrigation_Schedule_2'] == 1]
irrigation_schedule_9_test = test_data[test_data['Irrigation_Schedule_9'] == 1]
```

In [ ]:
```
# Create a copy of the original DataFrame
irrigation_2_copy = irrigation_schedule_2.copy()
irrigation_9_copy = irrigation_schedule_9.copy()

# Rename columns in the copied DataFrames
irrigation_2_copy.columns = [col.replace('_', ' ') for col in irrigation_2_copy.columns]
irrigation_9_copy.columns = [col.replace('_', ' ') for col in irrigation_9_copy.columns]
```

In [ ]:
```
# Drop the specified columns
correlation_matrix_schedule_2_ = irrigation_2_copy.drop(['Irrigation Schedule 2', 'Irrigation Schedule 9'], axi
correlation_matrix_schedule_9_ = irrigation_9_copy.drop(['Irrigation Schedule 2', 'Irrigation Schedule 9'], axi

# Plotting the correlation matrices
fig, axes = plt.subplots(1, 2, figsize=(18, 8))

# Correlation matrix for Irrigation Schedule 2
```

```
sns.heatmap(correlation_matrix_schedule_2_, annot=True, cmap='coolwarm', ax=axes[0], cbar=False)  # cbar=False
axes[0].set_title('Correlation Matrix for Irrigation Schedule 2')

# Correlation matrix for Irrigation Schedule 9
sns.heatmap(correlation_matrix_schedule_9_, annot=True, cmap='coolwarm', ax=axes[1])
axes[1].set_title('Correlation Matrix for Irrigation Schedule 9')

plt.tight_layout()

# Add a caption at the bottom of the figure
plt.figtext(0.5, 0.01, "Figure 7: Correlation Matrices for Different Irrigation Schedules", wrap=True, horizont

plt.show()
```

Based on the insights from Figure 7, it's clear that the different irrigation schedules do not significantly alter the relationships between yield and the feature variables such as sunny and rainy days, soil quality, and fertilizer. This lack of linear relationship among the features themselves is a crucial observation. It simplifies the analysis by indicating that these variables operate independently in their impact on yield, without influencing each other. This independence is particularly beneficial in understanding how each feature uniquely contributes to the yield.

Furthermore, the analysis reveals that soil quality and sunny days have a relatively insignificant effect on yield when compared to other variables. This finding is notable as it suggests that, regardless of the irrigation schedule, modifications in soil quality and the number of sunny days might not be as effective in enhancing yield as one might expect. In contrast, other factors like fertilizer and rainfall seem to play a more pivotal role, potentially offering more substantial levers for optimizing agricultural output. This insight is invaluable for agricultural planning, directing attention to the most impactful areas for improving crop yields under various irrigation conditions.

In [ ]:
```
# Replace spaces with underscores in variable names
variables = ['Soil_Quality', 'Fertilizer_Amount_kg_per_hectare', 'Sunny_Days', 'Rainfall_mm']
variables_for_plot = ['Soil Quality', 'Fertilizer Amount kg per hectare', 'Sunny Days', 'Rainfall mm']

moving_average_window = 10  # Example window size

# Create a single row for the subplot grid
n_rows = 1
n_cols = 4

fig, axes = plt.subplots(n_rows, n_cols, figsize=(25, 5))  # Larger figsize for bigger graphs

# Set a main title above all subplots
fig.suptitle('Relationship between Variables and Yield', fontsize=16)

# Iterate over variables and create a plot for each
for i, var in enumerate(variables):
    ax = axes[i]  # Use a single row for subplots, i determines the position in the grid

    color_map = {schedule: 'lightcoral' if schedule % 2 == 0 else 'lightblue' for schedule in train_data['Irrig

    for schedule in train_data['Irrigation_Schedule_2'].unique():
        subset_train = train_data[train_data['Irrigation_Schedule_2'] == schedule]

        # Plotting data points with larger markers
        sns.scatterplot(ax=ax, x=var, y='Yield_kg_per_hectare', data=subset_train, color=color_map[schedule], s

        # Fitting and plotting regression line
        model = LinearRegression()
        model.fit(subset_train[[var]], subset_train['Yield_kg_per_hectare'])
        slope = model.coef_[0]
        intercept = model.intercept_
        x_vals = subset_train[var]
        y_vals = intercept + slope * x_vals
        sns.lineplot(ax=ax, x=x_vals, y=y_vals, color='black', linewidth=2)

        # Calculating and plotting moving average
        subset_train_sorted = subset_train.sort_values(by=var)
        subset_train_sorted['MA_Yield'] = subset_train_sorted['Yield_kg_per_hectare'].rolling(window=moving_ave
        sns.lineplot(ax=ax, x=subset_train_sorted[var], y=subset_train_sorted['MA_Yield'], color='black', lines

    # Set xlabel and ylabel for each subplot
    ax.set_xlabel(variables_for_plot[i])  # Use the modified variable name
    ax.set_ylabel('Yield kg per hectare')

    # Set title as "Variable vs. Yield"
    ax.set_title(f"{variables_for_plot[i]} vs. Yield")  # Use the modified variable name

# Create custom legend handles and labels
legend_labels = ['Linear Regression', 'Nearest Neighbor Regression', 'Irrigation Schedule 9 (Red)', 'Irrigation
legend_handles = [plt.Line2D([], [], color='black', linewidth=2),
                  plt.Line2D([], [], color='black', linestyle='dotted'),
                  plt.Line2D([], [], color='red', marker='o', linestyle='None'),
                  plt.Line2D([], [], color='blue', marker='o', linestyle='None')]

# Create a single legend outside of subplots and move it closer to the graphs
fig.legend(handles=legend_handles, labels=legend_labels,
           loc='upper right', bbox_to_anchor=(1.1, 1.0), borderaxespad=0.)
```

```
# Add a caption below the plot
plt.figtext(0.5, 0.01, 'Figure 8: Variables plotted against yield with linear regression and nearest neighbor r

# Adjust the layout
plt.tight_layout(rect=[0, 0.1, 0.9, 0.95])  # Adjust the rect parameter to make space for the main title

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

In the context of agricultural yield as revealed by Figure 8 and corroborated by the correlation matrix, the relationships between key variables and yield exhibit distinct patterns. Fertilizer Amount and Soil Quality stand out with a clear linear regression trend when plotted against yield. This strong correlation suggests a direct and predictable relationship where higher levels of fertilizer application or better soil quality correspond to increased yield. These variables offer reliable indicators for yield optimization, highlighting the effectiveness of managing soil health and fertilizer usage in enhancing agricultural productivity.

In contrast, the variables Sunny Days and Rainfall present a more complex picture. Their constrained distribution in the data, noted both in the plots and the exploratory analysis, suggests a nuanced relationship with yield. Unlike the straightforward linear pattern observed with Fertilizer Amount and Soil Quality, these weather-related factors do not show a robust linear correlation. This complexity could be attributed to the fact that both insufficient and excessive amounts of sunshine or rainfall can adversely affect crop yields. The optimal impact of these variables on yield is likely contained within a specific range, and deviations beyond this range might not contribute positively to yield, or might even be detrimental.

Overall, these insights emphasize the multifaceted nature of agricultural yield determinants. While some factors like soil quality and fertilizer amount have a direct and measurable impact on yield, others such as climatic conditions require a more balanced approach to optimize their benefits. Understanding these diverse relationships is crucial for developing effective agricultural strategies that address the specific needs and limitations of different environmental and management factors.

In [ ]:
```
#scale

# Selecting the features for scaling (excluding the 'target' column)
features = train_data.drop(['Irrigation_Schedule_2','Irrigation_Schedule_9','Yield_kg_per_hectare'], axis=1)
# Initializing the StandardScaler
scaler = StandardScaler()
# Applying z-score scaling
scaled_features = scaler.fit_transform(features)
# Creating a new DataFrame for the scaled data
train_scale = pd.DataFrame(scaled_features, columns=features.columns)
# Adding the 'target' column back to the scaled DataFrame
train_scale[['Irrigation_Schedule_2', 'Irrigation_Schedule_9','Yield_kg_per_hectare']] = train_data[['Irrigatio

# Selecting the features for scaling (excluding the 'target' column)
features = test_data.drop(['Irrigation_Schedule_2','Irrigation_Schedule_9','Yield_kg_per_hectare'], axis=1)
# Initializing the StandardScaler
scaler = StandardScaler()
# Applying z-score scaling
scaled_features = scaler.fit_transform(features)
# Creating a new DataFrame for the scaled data
test_scale = pd.DataFrame(scaled_features, columns=features.columns)
# Adding the 'target' column back to the scaled DataFrame
test_scale[['Irrigation_Schedule_2', 'Irrigation_Schedule_9','Yield_kg_per_hectare']] = test_data[['Irrigation_
```

In [ ]:
```
# Define a function for the simulation
def simulation(models, X, y, iterations=10):
    results = {name: {'MSE': [], 'R2': []} for name in models.keys()}

    for i in range(iterations):
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

        for name, model in models.items():
            model.fit(X_train, y_train)
            y_pred = model.predict(X_test)
            mse = mean_squared_error(y_test, y_pred)
            r2 = r2_score(y_test, y_pred)

            results[name]['MSE'].append(mse)
            results[name]['R2'].append(r2)

    # Calculate average results
    avg_results = {name: {'Avg_MSE': np.mean(info['MSE']), 'Avg_R2': np.mean(info['R2'])} for name, info in res
    return avg_results

# Create example data (replace this with your own data)
X = np.random.rand(100, 5)
y = np.random.rand(100)

X9 = np.random.rand(100, 5)
y9 = np.random.rand(100)

# Define models
models = {
    'Linear Regression': LinearRegression(),
```

```
    'K-Nearest Neighbors': KNeighborsRegressor(),
    'Random Forest': RandomForestRegressor()
}

# Perform the simulation for each schedule
avg_results_schedule_2 = simulation(models, X, y)
avg_results_schedule_9 = simulation(models, X9, y9)

# Create DataFrames for the results
results_df_schedule_2 = pd.DataFrame(avg_results_schedule_2)
results_df_schedule_9 = pd.DataFrame(avg_results_schedule_9)
```

In [ ]:
```
#irrigation_schedule_1 = train_scale[train_scale['Irrigation_Schedule_2'] == 1]
X_train2 = irrigation_schedule_2.drop(['Yield_kg_per_hectare', 'Irrigation_Schedule_2', 'Irrigation_Schedule_9'
y_train2 = irrigation_schedule_2['Yield_kg_per_hectare']
#irrigation_schedule_1_test = test_scale[test_scale['Irrigation_Schedule_2'] == 1]
X_test2 = irrigation_schedule_2_test.drop(['Yield_kg_per_hectare', 'Irrigation_Schedule_2', 'Irrigation_Schedul
y_test2 = irrigation_schedule_2_test['Yield_kg_per_hectare']
#irrigation_schedule_1 = train_scale[train_scale['Irrigation_Schedule_2'] == 1]
X_train9 = irrigation_schedule_9.drop(['Yield_kg_per_hectare', 'Irrigation_Schedule_2', 'Irrigation_Schedule_9'
y_train9 = irrigation_schedule_9['Yield_kg_per_hectare']
#irrigation_schedule_1_test = test_scale[test_scale['Irrigation_Schedule_2'] == 1]
X_test9 = irrigation_schedule_9_test.drop(['Yield_kg_per_hectare', 'Irrigation_Schedule_2', 'Irrigation_Schedul
y_test9 = irrigation_schedule_9_test['Yield_kg_per_hectare']


# Create empty DataFrames to store the results
results_df2 = pd.DataFrame(columns=['Model', 'MSE', 'R2'])
results_df9 = pd.DataFrame(columns=['Model', 'MSE', 'R2'])

# Linear Regression for Schedule 2
ln_model2 = LinearRegression()
ln_model2.fit(X_train2, y_train2)
y_pred2 = ln_model2.predict(X_test2)
mse2 = mean_squared_error(y_test2, y_pred2)
r22 = r2_score(y_test2, y_pred2)
results_df2 = results_df2.append({'Model': 'Linear Regression', 'MSE': mse2, 'R2': r22}, ignore_index=True)

# Linear Regression for Schedule 9
ln_model9 = LinearRegression()
ln_model9.fit(X_train9, y_train9)
y_pred9 = ln_model9.predict(X_test9)
mse9 = mean_squared_error(y_test9, y_pred9)
r29 = r2_score(y_test9, y_pred9)
results_df9 = results_df9.append({'Model': 'Linear Regression', 'MSE': mse9, 'R2': r29}, ignore_index=True)
```

In [ ]:
```
# Create a figure with two subplots side by side
plt.figure(figsize=(12, 5))

# Residuals Plot for irrigation_schedule_9
residuals_9 = y_test9 - y_pred9
plt.subplot(1, 2, 1)  # 1 row, 2 columns, 1st subplot
sns.residplot(x=y_pred9, y=residuals_9, lowess=True, line_kws={'color': 'red', 'lw': 1})
plt.title("Residuals vs Predicted (Schedule 9)")
plt.xlabel("Predicted values")
plt.ylabel("Residuals")

# Residuals Plot for irrigation_schedule_2
residuals_2 = y_test2 - y_pred2
plt.subplot(1, 2, 2)  # 1 row, 2 columns, 2nd subplot
sns.residplot(x=y_pred2, y=residuals_2, lowess=True, line_kws={'color': 'red', 'lw': 1})
plt.title("Residuals vs Predicted (Schedule 2)")
plt.xlabel("Predicted values")
plt.ylabel("Residuals")

# Add a caption
plt.figtext(0.15, -0.05, 'Figure 9: Residuals Plot', ha='center', fontsize=12)

plt.tight_layout()
plt.show()
```

Analyzing the residual plots provides valuable insights into the agricultural yield predictions under different irrigation schedules. The plots exhibit a fairly random spread of residuals around the zero line, without any apparent systematic patterns or trends. This randomness is a positive sign, indicating that the linear regression models are capturing the essential relationship between the variables and the yield without overfitting to the noise in the data. The absence of such overfitting in your models is corroborated by the residuals, which do not show signs of bias or non-random clustering.

Additionally, the spread of residuals across both irrigation schedules suggests that while the model is reasonably effective, there are still variations in yield that it does not capture. This could be due to factors not included in the model or inherently unpredictable elements affecting agricultural yields. The relatively consistent spread of residuals, without extreme values or patterns, also indicates that the model has generalized well across different data points, maintaining a balance between accuracy and generalizability.

In summary, your residual plots reveal a well-fitted model that captures the key trends in the data without overfitting, while also highlighting the natural variability in agricultural yield that remains unexplained by the model. This balance is crucial for a robust and

reliable predictive model in agricultural contexts.

C. Clustering countries

1) Initial data analysis.

In [ ]:
```
new = pd.read_csv(r"C:\Users\archi\Downloads\new_countries.csv")
country_data =  pd.read_csv(r"C:\Users\archi\Downloads\country_data.csv")
```

In [ ]:
```
# Calculate the summary statistics
summary_statistics = country_data.describe().transpose()

# Adding median to the summary
median_values = country_data.median()
summary_statistics['median'] = median_values

# Reordering columns to include median after mean
summary_statistics = summary_statistics[['count', 'mean', 'median', 'std', 'min', '25%', '50%', '75%', 'max']]

# Formatting the table for better readability
formatted_summary = summary_statistics.style.format({
    'count': '{:,.0f}',
    'mean': '{:,.2f}',
    'median': '{:,.2f}',
    'std': '{:,.2f}',
    'min': '{:,.2f}',
    '25%': '{:,.2f}',
    '50%': '{:,.2f}',
    '75%': '{:,.2f}',
    'max': '{:,.2f}'
})

# Display the formatted summary table
print("Summary Statistics of the Dataset:")
display(formatted_summary)

# Add a caption
print("Table 3: A Summary of teh Country dataset")
```

In [ ]:
```
# Create a figure and a set of subplots for violin plots
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(15, 10))

# Professional color scheme: shades of blue
violin_color = "#1f77b4"  # Blue color for violin plots
box_color = "#2ca02c"     # Green color for box plots

# Plotting violin plots and then overlaying box plots
sns.violinplot(data=country_data, y='child_mort', ax=axes[0,0], inner=None, color=violin_color)
sns.boxplot(data=country_data, y='child_mort', ax=axes[0,0], width=0.1, color=box_color)
axes[0,0].set_title('Child Mortality Rate')
axes[0,0].set_ylabel('Deaths/1,000 live births')

sns.violinplot(data=country_data, y='income', ax=axes[0,1], inner=None, color=violin_color)
sns.boxplot(data=country_data, y='income', ax=axes[0,1], width=0.1, color=box_color)
axes[0,1].set_title('Income')
axes[0,1].set_ylabel('USD/year')

sns.violinplot(data=country_data, y='inflation', ax=axes[0,2], inner=None, color=violin_color)
sns.boxplot(data=country_data, y='inflation', ax=axes[0,2], width=0.1, color=box_color)
axes[0,2].set_title('Inflation Rate')
axes[0,2].set_ylabel('Percentage (%)')

sns.violinplot(data=country_data, y='LE', ax=axes[1,0], inner=None, color=violin_color)
sns.boxplot(data=country_data, y='LE', ax=axes[1,0], width=0.1, color=box_color)
axes[1,0].set_title('Life Expectancy')
axes[1,0].set_ylabel('Years')

sns.violinplot(data=country_data, y='total_fer', ax=axes[1,1], inner=None, color=violin_color)
sns.boxplot(data=country_data, y='total_fer', ax=axes[1,1], width=0.1, color=box_color)
axes[1,1].set_title('Total Fertility Rate')
axes[1,1].set_ylabel('Children per woman')

# Adjust layout and remove empty subplot
plt.delaxes(axes[1][2])
plt.tight_layout()

# Add a caption
fig.text(0.2, -0.05, 'Figure 14: Violin Plots and Box Plots', ha='center', fontsize=12)

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

In [ ]:
```
# Calculate the correlation matrix
corr_matrix = country_data.corr()
```

```
# Create a heatmap to visualize the correlation matrix
plt.figure(figsize=(10, 8))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f', linewidths=.5)
plt.title('Correlation Matrix')

# Add a caption
plt.gcf().text(0.2, -0.05, 'Figure 12: Correlation Matrix', ha='center', fontsize=12)

plt.show()
```

In [ ]:
```
# Create a pair plot
sns.pairplot(country_data)
plt.show()
```

In [ ]:
```
# Define the pairs of variables you want to plot
pairs = [
    ('income', 'LE'),
    ('income', 'child_mort'),
    ('income', 'total_fer'),
    ('inflation', 'LE'),
    ('inflation', 'child_mort'),
    ('inflation', 'total_fer')
]

# Create subplots
fig, axes = plt.subplots(2, 3, figsize=(15, 15))

for i, (x, y) in enumerate(pairs):
    row, col = i // 3, i % 3
    ax = axes[row, col]

    # Scatter plot
    sns.scatterplot(x=x, y=y, data=country_data, ax=ax, color='blue', alpha=0.7, label='Data')

    # Linear regression line
    sns.regplot(x=x, y=y, data=country_data, ax=ax, ci=None, color='red', scatter=False, label='Linear')

    # Polynomial regression line (order 2)
    z = np.polyfit(country_data[x], country_data[y], 2)
    p = np.poly1d(z)
    xp = np.linspace(country_data[x].min(), country_data[x].max(), 100)
    ax.plot(xp, p(xp), 'g--', label='Polynomial')

    ax.set_title(f'{y} vs. {x}')
    ax.legend(loc='upper right')

# Add a caption
fig.text(0.2, -0.05, 'Figure 13: Linear and Polynomial regression', ha='center', fontsize=12)


plt.tight_layout()
plt.show()
```

When examining the interplay between economic data and health metrics, it becomes evident that inflation exhibits a rather complex relationship with health indicators. Inflation demonstrates a clear polynomial association, which introduces a degree of unpredictability into the analysis. Unfortunately, this complexity gives rise to multicollinearity issues, rendering coefficient estimates unreliable and making the data harder to interpret.

In contrast, a noteworthy observation is the presence of linearity between certain health metrics, such as child mortality, total fertility, and life expectancy. Unlike the polynomial relationship seen with inflation, in these cases, a small increase in income is associated with a disproportionately substantial improvement in health metrics. This underscores the significant impact that income can have on health outcomes when a linear relationship is at play.

2) Hierarchical clustering.

In [ ]:
```
# Selecting relevant columns (excluding 'country' if it's a column)
columns_to_scale = country_data.columns.drop('country')
scaler = StandardScaler()

# Scale the data
data_scaled = scaler.fit_transform(country_data[columns_to_scale])
data_scaled = pd.DataFrame(data_scaled, columns=columns_to_scale)
data_scaled['country'] = country_data['country']

# Use Pandas styling to create a more appealing table view
styled_table = data_scaled.head(2).style.set_table_styles(
    [{'selector': 'th', 'props': [('font-size', '12pt'), ('background-color', 'lightgray'), ('color', 'black')]
     {'selector': 'td', 'props': [('font-size', '12pt')]}]
).set_properties(**{'text-align': 'center'}).hide_index()

# Display the styled table
styled_table
```

Scaling data before clustering is essential to ensure that all features contribute equally to the clustering process. Clustering algorithms like K-means are distance-based, meaning they calculate distances between data points to form clusters. If features have different scales or units, those with larger scales may dominate the distance calculation, potentially causing the clustering algorithm to be biased towards certain features. Scaling standardizes the range and distribution of features, preventing any single feature from overpowering the clustering results. This allows for a fair and meaningful comparison of distances between data points, leading to more accurate and reliable clustering outcomes. In essence, scaling helps to eliminate the influence of scale and units, making the clustering process more robust and effective.

In [ ]:
```
country_names = data_scaled['country']

# Select numerical columns for scaling, excluding the 'country' column
scaled_numerical_data = data_scaled.select_dtypes(include=[np.number])

# Compute the distance matrix
distance_matrix = squareform(pdist(scaled_numerical_data, 'euclidean'))

# Perform hierarchical clustering
linked = linkage(scaled_numerical_data, method='ward')

# Retrieve the order of the observations determined by the clustering
order = leaves_list(linked)

# Apply this order to the country names and the rows/columns of the distance matrix
ordered_country_names = country_names[order]
sorted_distance_matrix = distance_matrix[np.ix_(order, order)]

# Convert the sorted distance matrix to a DataFrame for the heatmap
sorted_distance_matrix_df = pd.DataFrame(sorted_distance_matrix, index=ordered_country_names, columns=ordered_c

# Plot the sorted distance matrix
plt.figure(figsize=(12, 10))
sns.heatmap(sorted_distance_matrix_df, cmap='viridis', xticklabels=ordered_country_names, yticklabels=ordered_c
plt.title('Distance Matrix Visualization')

# Rotate x labels for better readability if needed
plt.xticks(rotation=90)
plt.yticks(rotation=0)

# Add a caption
plt.text(0.2, -0.06, 'Figure 14: Hierarchical Clustering - Distance Matrix Visualization', ha='center', fontsiz

plt.show()
```

Figure 14 has been thoughtfully arranged in order of proximity, resulting in a more intuitive and easily interpretable graph. The most prominent cluster comprises countries stretching from Norway to South Korea, encompassing a majority of European nations and those sharing close affiliations with them, such as the United States and Australia, to name a couple. In contrast, a distinct grouping emerges from Angola to Uganda, primarily comprised of African and Middle Eastern countries. Notably, this cluster appears relatively isolated from other nations. The remaining countries, from South Africa to Russia, neither exhibit proximity nor significant distance from others. This is understandable, given that this category comprises nations like Russia, China, India, and South Africa, each possessing unique cultural landscapes that significantly influence various health and economic metrics. This diversity, particularly in the cases of Russia and China, contributes to their distinct positions within the graph.

In [ ]:
```
# Perform hierarchical clustering again
linked = linkage(scaled_numerical_data, 'ward')

# Plot the dendrogram
plt.figure(figsize=(15, 10))
dendrogram(linked, labels=data_scaled['country'].values, orientation='top', leaf_rotation=90)
plt.title('Hierarchical Clustering Dendrogram')
plt.xlabel('Country')
plt.ylabel('Distance')

# Choose the number of clusters based on the dendrogram (this is a subjective choice)
# Adjusting the distance cutoff to a lower value for more refined clustering
distance_cutoff = 15  # Adjusted based on the dendrogram
clusters = fcluster(linked, distance_cutoff, criterion='distance')

# Add cluster labels to the original data
data_scaled['Cluster'] = clusters

plt.show()
```

In [ ]:
```
# Perform hierarchical clustering and create linkage matrix
linked = linkage(scaled_numerical_data, method='ward')  # Replace 'data_scaled' with your scaled data

# Calculate silhouette scores for different levels of clustering
silhouette_scores = []
max_clusters = 10  # You can adjust this based on your data and problem
for n_clusters in range(2, max_clusters + 1):
    cluster_labels = fcluster(linked, t=n_clusters, criterion='maxclust')
    silhouette_avg = silhouette_score(scaled_numerical_data, cluster_labels)
```

```
    silhouette_scores.append(silhouette_avg)

# Plot the silhouette scores
import matplotlib.pyplot as plt
plt.plot(range(2, max_clusters + 1), silhouette_scores, marker='o')
plt.xlabel('Number of Clusters')
plt.ylabel('Average Silhouette Score')
plt.title('Silhouette Score for Hierarchical Clustering')
plt.show()
```

When determining the appropriate number of clusters, it's crucial to keep the primary goal in mind, which, in this context, is to identify a group of countries that would genuinely benefit from direct aid. Ideally, these clusters should contain five or fewer countries. However, it's important to consider the practicality of cluster sizes. If, for instance, one cluster consists of seven countries but comprises only highly developed nations that would not significantly benefit from direct aid, it doesn't substantially impact our decision-making process.

A Dendrogram, in conjunction with silhouette scores, aids in making a well-informed decision. While the silhouette scores were at their lowest when considering nine clusters, examining the dendrogram reveals that the shift from nine to seven clusters primarily involves highly developed countries, like Norway. As mentioned earlier, having a large cluster of developed countries doesn't align with the primary objective of identifying aid-receiving nations. Therefore, the focus should remain on clusters where genuine aid benefits can be realized, even if this means opting for a lower number of clusters, such as seven.

In [ ]:
```
# Specify the number of clusters
number_of_clusters = 7

# Create clusters
clusters = fcluster(linked, number_of_clusters, criterion='maxclust')

# Add cluster labels to the original data
data_scaled['Cluster'] = clusters

# Count the number of countries in each cluster
cluster_counts = data_scaled['Cluster'].value_counts()

# Display the cluster membership and counts
print(data_scaled[['country', 'Cluster']].sort_values(by='Cluster'))
print(cluster_counts)

# Replotting the dendrogram with the clusters
plt.figure(figsize=(15, 10))
dendrogram(linked, labels=data_scaled['country'].values, orientation='top', leaf_rotation=90,
           color_threshold=linked[-(number_of_clusters-1), 2])
plt.title('Hierarchical Clustering Dendrogram with n Clusters')
plt.xlabel('Country')
plt.ylabel('Distance')
plt.axhline(y=linked[-(number_of_clusters-1), 2], color='r', linestyle='--')
plt.show()
```

In [ ]:
```
# Use seaborn pairplot
sns.pairplot(data_scaled, hue='Cluster', vars=scaled_numerical_data.columns, palette='Dark2')
plt.title('Pair Plot of Countries')
plt.show()
```

In [ ]:
```
country_data['Cluster'] = data_scaled['Cluster']
# Select the relevant features
selected_features = ['inflation', 'income', 'LE', 'total_fer', 'child_mort']

# Create a summary table for cluster statistics
cluster_summary = country_data.groupby('Cluster')[selected_features].agg(['mean', 'median', 'std'])

# Rename columns for better readability
cluster_summary.columns = [f'{feature}_{stat}' for feature, stat in cluster_summary.columns]

# Reset the index to make 'Cluster' a regular column
cluster_summary.reset_index(inplace=True)

# Display the summary table
print(cluster_summary)
```

In [ ]:
```
# Define a function to map cluster labels to cluster names
def map_cluster_to_name(cluster_label):
    if cluster_label == 1:
        return 'Underdeveloped but Highest Income of low income nations'
    elif cluster_label == 2:
        return 'Underdeveloped with low inflation'
    elif cluster_label == 3:
        return 'Underdeveloped with high infaltion'
    elif cluster_label == 4:
        return 'highly Develped Nations'
    elif cluster_label == 5:
        return 'Potential to develop'
    elif cluster_label == 6:
        return 'Developed Nation'
    elif cluster_label == 7:
        return 'Developed but High Inflation'
```

```
    else:
        return 'Unknown'  # Handle other values if present

# Apply the mapping function to create the 'Cluster Name' column
country_data['Cluster Name'] = country_data['Cluster'].apply(map_cluster_to_name)
```

In the clustering analysis, Clusters 1, 2, 3, and 5 are marked by lower life expectancies, which likely contribute to higher child mortality and total fertility rates. This pattern often emerges in socio-economically challenged regions, where higher fertility rates can be a response to increased child mortality, ensuring the survival of some children to adulthood. In contrast, Clusters 4, 6, and 7 are characterized by higher life expectancies and incomes, indicative of better socio-economic conditions. These conditions typically result in lower child mortality and, consequently, lower fertility rates, as the imperative for larger families diminishes. Notably, Cluster 4 stands out with exceptionally high income levels, suggesting an even stronger economic foundation that likely supports superior health and living conditions. Inflation emerges as a differentiating factor among these more affluent clusters, potentially reflecting variations in economic stability and policy effectiveness. This analysis underscores the intricate relationship between economic factors and demographic indicators, highlighting how income levels and economic policies can significantly impact health and population dynamics.

3) K-means clustering.

In [ ]:
```
#apply kmeans and set num of clusters
kmeans = KMeans(n_clusters=7, random_state=0)
kmeans_cluster = kmeans.fit_predict(scaled_numerical_data)

#add kmeans to the data
data_scaled['KMeans_Cluster'] = kmeans_cluster

silhouette_avg = silhouette_score(scaled_numerical_data, kmeans_cluster)
print(f"Silhouette Score: {silhouette_avg}")
```

In [ ]:
```
# Create a pairplot to visualize the clusters
sns.pairplot(data_scaled, hue='KMeans_Cluster',vars=scaled_numerical_data.columns, palette='Dark2')
plt.show()
```

In [ ]:
```
# Create a DataFrame with cluster memberships
cluster_comparison = pd.DataFrame({
    'Country': data_scaled['country'],
    'Hierarchical_Cluster': data_scaled['Cluster'],
    'KMeans_Cluster': data_scaled['KMeans_Cluster']
})

# Remove the index column
cluster_comparison = cluster_comparison.reset_index(drop=True)

# Sort by the "Hierarchical_Cluster" column
cluster_comparison = cluster_comparison.sort_values(by='Hierarchical_Cluster')

# Display the cluster memberships
print(cluster_comparison)
```

4) Allocating new observations to clusters.

In [ ]:
```
# Assuming 'data_scaled' contains your old data
old_countries_clusters = country_data[['country', 'Cluster']]
new_col_to_scale = new.columns.drop('country')


new_scaled = scaler.fit_transform(new[new_col_to_scale])
new_scaled = pd.DataFrame(new_scaled, columns=new_col_to_scale)
```

In [ ]:
```
scaled_numerical_data['cluster'] = country_data['Cluster']
# Calculate distances to cluster centroids
cluster_centroids = scaled_numerical_data.groupby('cluster').mean()
cluster_centroids['cluster'] = country_data['Cluster']
cluster_centroids = cluster_centroids.drop(columns='cluster')
```

In [ ]:
```
# Calculate distances to cluster centroids
#cluster_centroids = scaled_numerical_data.groupby('cluster').mean()
#cluster_centroids = cluster_centroids.drop(columns='cluster')
```

In [ ]:
```
new_scaled.columns
```

In [ ]:
```
# Now calculate the distances
distances = cdist(new_scaled, cluster_centroids)

# Find the nearest centroid for each new data point
nearest_clusters = np.argmin(distances, axis=1)

# Add the cluster assignments to your new data
new['Cluster'] = nearest_clusters
```

In [ ]:
```
new_scaled
```

In [ ]:
```
old_countries_clusters = country_data[['country', 'Cluster']]
new_countries_clusters = new[['country', 'Cluster']].copy()
new_countries_clusters['country'] = new_countries_clusters['country'] + " (New)"

combined_countries_clusters = pd.concat([old_countries_clusters, new_countries_clusters], ignore_index=True)

# Create a dictionary where each cluster is a key and countries are values
cluster_dict = {}
for cluster in combined_countries_clusters['Cluster'].unique():
    cluster_dict[cluster] = combined_countries_clusters[combined_countries_clusters['Cluster'] == cluster]['cou

# Convert dictionary to DataFrame for table format
cluster_table = pd.DataFrame(dict([(k, pd.Series(v)) for k, v in cluster_dict.items()]))
display(cluster_table)
```

The code appears to have some issues, as it incorrectly assigned Switzerland to the same cluster as countries like Uganda and Zambia. While Chad and Armenia are placed in a more correct cluster with nations that have similar econmic and health metrics.

5) Putting data back into context.

In determining the allocation of direct aid, it's essential to strategically select recipient clusters based on specific criteria. Initially, clusters 4, 6, and 7 can be excluded from consideration, focusing our attention on clusters 1, 2, 3, and 5. When it comes to providing financial benefits, we must consider the stability of the currency. Clusters 1 and 3, facing currency instability issues, are therefore not suitable candidates. This leaves us with clusters 2 and 5. Between these, cluster 2 emerges as the most appropriate choice for financial aid due to its higher child mortality rates and lower income levels. Notably, given the polynomial relationship between income and health metrics, cluster 2 stands to benefit more significantly from financial aid, experiencing enhanced growth compared to cluster 5.

Conversely, when the aid is in the form of infrastructure development, such as the construction of hospitals, inflation concerns become irrelevant. In this context, cluster 1, alongside cluster 2, should be prioritized. These clusters exhibit the poorest health and economic indicators, suggesting that infrastructure development in these areas would yield the most substantial benefits. This strategic approach to aid allocation ensures that resources are directed where they can have the greatest impact, factoring in the unique needs and conditions of each cluster

Bibliography

converstiion with chat gpt

User new_col_to_scale = new.columns.drop('country')

new_scaled = scaler.fit_transform(new[new_col_to_scale]) new_scaled = pd.DataFrame(new_scaled, columns=new_col_to_scale)cluster_centroids = data_scaled.groupby('Cluster').mean() cluster_centroids = cluster_centroids.drop(columns='KMeans_Cluster')

Calculate distances from each new data point to each cluster centroid

distances = cdist(new_scaled, cluster_centroids)

Find the nearest centroid for each new data point

nearest_clusters = np.argmin(distances, axis=1)

Add the cluster assignments to your new data

new['Cluster'] = nearest_clusters GPT The code you've shared is part of a process in which a new dataset, new, is being scaled and then assigned to existing clusters based on their proximity to the centroids of those clusters. This process typically follows these steps:

Scaling the New Data: The new dataset (excluding the 'country' column) is scaled using a scaler (presumably fitted to some training data earlier). This is important to ensure that the new data is on the same scale as the data used to create the clusters.

Calculating Cluster Centroids: The centroids of existing clusters are calculated. These centroids are the mean values of the points in each cluster. It seems the cluster labels were stored in a column named 'Cluster' in the data_scaled dataset.

Calculating Distances: The distances from each scaled new data point to each cluster centroid are calculated. This is typically done using a distance metric like Euclidean distance.

Assigning Nearest Clusters: Each new data point is assigned to the nearest cluster based on the calculated distances. This is done by finding the index of the minimum distance for each point, which corresponds to the nearest cluster.

Updating the New Data: Finally, the nearest cluster assignments are added to the new data as a new column named 'Cluster'.

This process is common in cluster analysis, particularly when you want to classify new data points based on existing cluster centroids. This approach is useful when the new data should not influence the existing cluster structure but should be categorized according to it.

If you have specific questions or need further assistance with this code or its context, feel free to ask!

User

ValueError Traceback (most recent call last) Cell In[36], line 5 2 cluster_centroids = cluster_centroids.drop(columns='KMeans_Cluster') 4 # Calculate distances from each new data point to each cluster centroid ----> 5 distances = cdist(new_scaled, cluster_centroids) 7 # Find the nearest centroid for each new data point 8 nearest_clusters = np.argmin(distances, axis=1)

File ~\AppData\Roaming\Python\Python311\site-packages\scipy\spatial\distance.py:2920, in cdist(XA, XB, metric, out, **kwargs) 2918 raise ValueError('XB must be a 2-dimensional array.') 2919 if s[1] != sB[1]: -> 2920 raise ValueError('XA and XB must have the same number of columns ' 2921 '(i.e. feature dimension.)') 2923 mA = s[0] 2924 mB = sB[0]

ValueError: XA and XB must have the same number of columns (i.e. feature dimension.) GPT The ValueError you encountered indicates that the new_scaled dataset and the cluster_centroids dataset do not have the same number of columns, which is a requirement for calculating distances between them using cdist. This mismatch in dimensions can occur for several reasons:

Different Features in Datasets: The new_scaled and cluster_centroids may have different sets of features (columns). This can happen if the scaling or the cluster centroid calculation omitted or included additional columns.

Missing or Extra Columns: If new had columns that were not present in the data used to calculate cluster_centroids, or vice versa, this would lead to a mismatch.

To resolve this, you need to ensure that both new_scaled and cluster_centroids have the same columns in the same order. Here are the steps to do that:

Check Column Names and Order: Verify that the columns in new_scaled and cluster_centroids are the same and in the same order. You can print the column names of both to compare.

Align Columns: If there are differences, align the columns. You may need to drop extra columns or reorder them to match.

Re-run the Distance Calculation: After aligning the columns, you can attempt to run the cdist function again.

Let's start by checking the column names in both new_scaled and cluster_centroids to identify any discrepancies. If you can provide the column names of these dataframes, I can further assist you in aligning them.

User how can i check ho many columsn in each GPT To check the number of columns in each DataFrame (new_scaled and cluster_centroids), you can use the .shape attribute of a DataFrame, which returns a tuple representing the dimensions of the DataFrame. The first element of the tuple is the number of rows, and the second is the number of columns.

Here's how you can do it:

python Copy code

Check the number of columns in new_scaled

num_columns_new_scaled = new_scaled.shape[1] print("Number of columns in new_scaled:", num_columns_new_scaled)

Check the number of columns in cluster_centroids

num_columns_cluster_centroids = cluster_centroids.shape[1] print("Number of columns in cluster_centroids:", num_columns_cluster_centroids) This will give you the number of columns in each DataFrame. If the numbers are different, that's the likely cause of the error you're encountering with cdist.

After identifying a mismatch in the number of columns, the next step would be to examine which specific columns are different, which you can do by comparing the column names of the two DataFrames. Here's how:

python Copy code

Compare column names of new_scaled and cluster_centroids

columns_new_scaled = set(new_scaled.columns) columns_cluster_centroids = set(cluster_centroids.columns)

Find columns that are not in both DataFrames

extra_in_new_scaled = columns_new_scaled - columns_cluster_centroids extra_in_cluster_centroids = columns_cluster_centroids - columns_new_scaled

print("Columns in new_scaled not in cluster_centroids:", extra_in_new_scaled) print("Columns in cluster_centroids not in new_scaled:", extra_in_cluster_centroids) This code will help you identify any columns that are in one DataFrame but not the other, which is crucial for resolving the dimension mismatch.

User cluster_centroids has a cluster colum which i don't have in new_scaled yet GPT If the cluster_centroids DataFrame has an additional 'Cluster' column that is not present in the new_scaled DataFrame, this would certainly cause a dimension mismatch when using cdist. Since cdist requires both matrices (DataFrames, in this case) to have the same number of features (columns), you need to make sure that both DataFrames are aligned in terms of their features.

Given that the 'Cluster' column in cluster_centroids is an identifier rather than a feature, it should not be used in the distance calculation. You can resolve this issue by dropping the 'Cluster' column from the cluster_centroids DataFrame before calculating the distances. Here's how you can modify the code:

python Copy code

Drop the 'Cluster' column from the cluster_centroids DataFrame

cluster_centroids = cluster_centroids.drop(columns=['Cluster'])

Now calculate the distances

distances = cdist(new_scaled, cluster_centroids)

Find the nearest centroid for each new data point

nearest_clusters = np.argmin(distances, axis=1)

Add the cluster assignments to your new data

new['Cluster'] = nearest_clusters This approach ensures that both new_scaled and cluster_centroids have the same number of features, allowing the distance calculation to proceed without the dimensionality error.

User countries in clusters 1,2,3,5 all have low life expectancy which then has a knock on effect to child mortality and total fertility rates as you need to have more children to guarantee at least one survives which isn’t a problem in cluster 4,6 and 7 which all have high life expectancies and high incomes the only thing that sets 4,6 and 7 apart is the inflation and in cluster 4 the income is extremely high. make this better GPT The observed patterns in your clustering analysis suggest distinct socio-economic characteristics across different clusters, particularly in terms of life expectancy, child mortality, total fertility rates, and income levels. Here's a refined interpretation of these patterns:

Clusters 1, 2, 3, and 5: These clusters are characterized by lower life expectancy. This lower life expectancy is likely influencing higher child mortality and total fertility rates. In such scenarios, higher fertility rates can be understood as a response to higher child mortality; families might have more children to increase the likelihood of some surviving to adulthood. This pattern is a common characteristic in regions facing socio-economic challenges.

Clusters 4, 6, and 7: These clusters stand out due to their higher life expectancies and higher incomes, suggesting better overall socio-economic conditions. In such contexts, families typically face lower child mortality rates, leading to lower fertility rates as the need to have more children for ensuring survival diminishes.

Distinguishing Features of Cluster 4: Within this group, Cluster 4 is particularly noteworthy for its exceptionally high income levels. This economic advantage might be contributing to more robust health systems and living conditions, further enhancing life expectancy and quality of life.

Role of Inflation: Inflation appears to be a differentiating factor among Clusters 4, 6, and 7. The varying levels of inflation across these clusters might indicate differences in economic stability and policy effectiveness, which can have downstream effects on public services, including healthcare and education.

This analysis suggests a strong correlation between socio-economic factors like income, inflation, and health-related outcomes such as life expectancy, child mortality, and fertility rates. The disparities between the clusters highlight the complex interplay between economic conditions and demographic patterns.

In [ ]: