## 

## Step 2: Rename Columns & Standardize Formats

*# Standardize column names*

hdi\_df**.**columns **=** hdi\_df**.**columns**.**str**.**lower()**.**str**.**strip()**.**str**.**replace(" ", "\_")**.**str**.**replace(r"[^\w\s]", "", regex**=True**)

wb\_df**.**columns **=** wb\_df**.**columns**.**str**.**lower()**.**str**.**strip()**.**str**.**replace(" ", "\_")**.**str**.**replace(r"[^\w\s]", "", regex**=True**)

#### **Purpose:**

To **clean and standardize** the column names of both datasets (hdi\_df and wb\_df) for easier processing and consistency in later analysis.

#### **What Each Part Does:**

* .str.lower(): Converts all column names to lowercase.  
  + Example: Country Name → country name
* .str.strip(): Removes any leading or trailing whitespace.
* .str.replace(" ", "\_"): Replaces spaces in column names with underscores.  
  + country name → country\_name
* .str.replace(r"[^\w\s]", "", regex=True): Removes any special characters (like %, (), ,, etc.) from the column names using a regular expression.  
  + Example: GDP ($) → gdp

*# Drop columns or rows with excessive nulls*

wb\_df **=** wb\_df**.**dropna(thresh**=**len(wb\_df**.**columns) **\*** 0.6)

hdi\_df **=** hdi\_df**.**dropna(thresh**=**len(hdi\_df**.**columns) **\*** 0.6)

#### **Purpose:**

To **clean the data** by removing rows that contain **too many missing values** (nulls).

#### **Explanation:**

* dropna(thresh=...) removes any **row** that doesn't have at least the specified number of **non-null values**.
* len(wb\_df.columns) \* 0.6 means: keep the row if it has **at least 60% of the columns filled**; otherwise, drop it.

**Result**:  
 You avoid working with rows that are mostly incomplete and improve data quality for analysis or modeling.

## explain to me how these codes work and what they do:

## Step 3: Merge the Datasets

To merge, both datasets need a common key — most likely country and year. Let’s align them:

*# Reshape hdi\_df from wide to long format for HDI values*

hdi\_long **=** hdi\_df**.**melt(

id\_vars**=**['iso3', 'country', 'hdicode', 'region', 'hdi\_rank\_2021'],

value\_vars**=**[col **for** col **in** hdi\_df**.**columns **if** col**.**startswith('hdi\_') **and** col[4:]**.**isdigit()],

var\_name**=**'year',

value\_name**=**'hdi'

)

The melt() function transforms the **wide format** HDI dataset into a **long format**. HDI was originally in **one column per year**, e.g.:

country hdi\_1990 hdi\_1991 hdi\_1992 ...

Rwanda 0.300 0.305 0.310 ...

After melting, each row represents one country-year:

country year hdi

Rwanda hdi\_1990 0.300

Rwanda hdi\_1991 0.305

#### **Arguments explained:**

* id\_vars=[...]: Columns you want to **keep as is** (non-HDIs).
* value\_vars=[...]: Columns you want to **melt into rows** — in this case, all columns that start with "hdi\_" and are followed by digits (like "hdi\_1990").
* var\_name='year': The new column name for the HDI year.
* value\_name='hdi': The new column name for the HDI value.

hdi\_long['year'] **=** hdi\_long['year']**.**str**.**replace('hdi\_', '')**.**astype(int)

#### **What It Does:**

* Ensures the year in World Bank data is also of integer type (same as in hdi\_long).
* Necessary for merging datasets on year.

*# Ensure year columns are in same format*

wb\_df['year'] **=** wb\_df['year']**.**astype(int)

*# Merge on 'country' and 'year'*

merged\_df **=** pd**.**merge(wb\_df, hdi\_long, left\_on**=**['country\_name', 'year'], right\_on**=**['country', 'year'], how**=**'inner')

#### **What It Does:**

Performs an **inner join** on:

* country\_name (from wb\_df)
* country (from hdi\_long)
* year (from both)

This combines rows that have **matching country and year values** in both datasets.

*# Inspect merged data*

print(merged\_df**.**shape)

print(merged\_df**.**columns)

merged\_df**.**head()

#### **What It Does:**

* merged\_df.shape: Shows the number of rows and columns in the merged dataset.
* merged\_df.columns: Displays the list of all column names.
* merged\_df.head(): Displays the first few rows of the merged data for inspection.

*# Fill missing numeric values with mean*

**for** col **in** merged\_df**.**select\_dtypes(include**=**'number')**.**columns:

merged\_df[col] **=** merged\_df[col]**.**fillna(merged\_df[col]**.**mean())

### **What This Code Does:**

#### **1. merged\_df.select\_dtypes(include='number')**

* This selects **only the numeric columns** in your dataset.
* Example: GDP, life expectancy, HDI, population, electricity consumption, etc.
* It ignores non-numeric columns like "country", "region", or "iso3".

#### **2. .columns**

* This retrieves a list of all **numeric column names**.

#### **3. for col in ...**

* This **loops through each numeric column** name in the dataset.

#### **4. merged\_df[col].fillna(merged\_df[col].mean())**

* This **fills in missing (NaN) values** in that column by replacing them with the **mean (average)** of that column.

#### **5. merged\_df[col] = ...**

* This **updates the original column** in the dataset with the filled version.

### EDA Question 1: Which countries have experienced the highest growth in population and GDP? Is there overlap?

#### **DP Growth Calculation**

gdp\_growth = merged\_df.groupby('country')['gdp\_per\_capita\_usd'].agg(['min', 'max']).reset\_index()

gdp\_growth['gdp\_growth'] = gdp\_growth['max'] - gdp\_growth['min']

**What it does:**

* Groups the data by **country**.
* For each country, it calculates the **minimum and maximum GDP per capita** across all years available.
* Then, it computes the **GDP growth** as the difference between max and min.

**Goal:** Measure how much each country’s GDP per capita has grown over time.

#### **2 Population Density Growth Calculation**

pop\_growth = merged\_df.groupby('country')['population\_density\_people\_per\_sq\_km\_of\_land\_area'].agg(['min', 'max']).reset\_index()

pop\_growth['population\_growth'] = pop\_growth['max'] - pop\_growth['min']

**What it does:**

* Similar to the GDP part, but this time for **population density**.
* Calculates **minimum and maximum** population density per country over time.
* Then computes **growth** as the difference.

**Goal:** Understand which countries have become more densely populated.

#### **3 Sort and Select Top 10 Countries**

python

CopyEdit

top\_gdp = gdp\_growth.sort\_values(by='gdp\_growth', ascending=False).head(10)

top\_pop = pop\_growth.sort\_values(by='population\_growth', ascending=False).head(10)

**What it does:**

* Sorts the GDP and population growth data in descending order.
* Selects the **top 10 countries** for each growth type.

This gives:

* top\_gdp: Countries with the **highest economic growth**.
* top\_pop: Countries with the **highest population growth**.

#### **4 Print Results**

print("Top 10 GDP Growth Countries:\n", top\_gdp[['country', 'gdp\_growth']])

print("\nTop 10 Population Growth Countries:\n", top\_pop[['country', 'population\_growth']])

**What it does:**

* Displays the top 10 countries for each metric, along with the growth values.

#### **5 Find Overlapping Countries**

overlap = set(top\_gdp['country']).intersection(set(top\_pop['country']))

print("\nCountries in both GDP and Population Growth Top 10:\n", list(overlap))

**What it does:**

* Converts the country lists to sets and finds their **intersection**.
* Displays any country that is in **both** top 10 lists.

**Why this matters:** Countries that are growing economically *and* demographically may be undergoing broader transformations — e.g., urbanization, industrialization, or population-driven economic growth.

### **What You Learn from This Code:**

* Countries with **massive economic expansion** (e.g., Luxembourg, Qatar).
* Countries experiencing **rapid population increases** (e.g., Singapore, Bangladesh).
* **Balanced growth countries** (e.g., Singapore) that appear in both lists — indicating strong development momentum.
* **Policy Insight**: These results help governments and organizations understand where infrastructure, education, or healthcare investments might be urgently needed.

### EDA Question 2: Where did HDI grow the most in the 21st century?

*# Filter 2000 onwards*

df\_21st **=** merged\_df[merged\_df['year'] **>=** 2000]

**What it does:**

* Filters the merged dataset to include only rows where the year is **2000 or later**.
* This creates a new DataFrame (df\_21st) that contains just 21st-century records.

**Why this is done:** The question specifically focuses on changes in HDI in the 21st century — so we exclude earlier years.

*# Calculate HDI change by country*

hdi\_change **=** df\_21st**.**groupby('country')['hdi']**.**agg(['min', 'max'])**.**reset\_index()

hdi\_change['hdi\_growth'] **=** hdi\_change['max'] **-** hdi\_change['min']

**What it does:**

* Groups the data by **country**.
* For each country, it calculates the **minimum** and **maximum HDI** from 2000 onwards.
* Then it creates a new column hdi\_growth representing the **difference** between max and min HDI values.

**Why this is done:** This tells us **how much HDI has increased** for each country over the time period. A larger difference means **greater improvement**.

*# Top 10 countries with highest HDI improvement*

top\_hdi **=** hdi\_change**.**sort\_values(by**=**'hdi\_growth', ascending**=False**)**.**head(10)

print("Top 10 HDI Growth Countries since 2000:\n", top\_hdi[['country', 'hdi\_growth']])

**What it does:**

* Sorts the countries in descending order based on their HDI growth.
* Selects the **top 10 countries** that had the **largest HDI improvements** since 2000.

**Why this is done:** We want to **highlight the top performers** — the countries that made the most progress in human development.

### EDA Question 3: Which factors are highly correlated with life expectancy?

*# Correlation matrix focused on life expectancy*

correlations **=** merged\_df**.**corr(numeric\_only**=True**)

life\_corr **=** correlations['life\_expectancy\_at\_birth\_years']**.**sort\_values(ascending**=False**)

**What it does:**

* Calculates the **Pearson correlation coefficients** between all numeric columns in the merged\_df DataFrame.
* This measures **how strongly each pair of numeric variables is linearly related**.
* numeric\_only=True ensures only numeric columns are included (avoids errors from strings/categorical data).

**Why this is done:** We want to explore relationships between numeric development indicators — e.g., HDI, GDP, infant mortality, etc.

print("Top Positive Correlations with Life Expectancy:\n", life\_corr**.**head(6))

print("\nTop Negative Correlations with Life Expectancy:\n", life\_corr**.**tail(5))

* Prints the top 6 variables that have the **strongest positive correlation** with life expectancy.
* Prints the **5 variables that are most negatively correlated** with life expectancy — i.e., as these values increase, life expectancy tends to decrease.

## 6.Build a regression model to predict Life Expectancy using features like GDP per capita, HDI, and other economic indicators.

#### Step 1: Feature Selection

*# Select features for modeling*

features **=** [

'gdp\_per\_capita\_usd',

'population\_density\_people\_per\_sq\_km\_of\_land\_area',

'electric\_power\_consumption\_kwh\_per\_capita',

'hdi'

]

**What it does:**

* Creates a Python list of the **feature column names** that will be used as predictors in the model.

**Why these were selected:**

* gdp\_per\_capita\_usd: Economic wealth and development
* population\_density\_people\_per\_sq\_km\_of\_land\_area: Urbanization or crowding
* electric\_power\_consumption\_kwh\_per\_capita: Infrastructure and access to services
* hdi: Composite indicator including education, income, and life expectancy

These variables are **numeric, continuous, and theoretically meaningful** predictors of life expectancy.

*# Drop NA rows for selected features + target*

model\_df **=** merged\_df[features **+** ['life\_expectancy\_at\_birth\_years']]**.**dropna()

**What it does:**

* Creates a new dataframe model\_df that contains:  
  + The selected feature columns
  + The **target variable**: 'life\_expectancy\_at\_birth\_years'
* Then drops any **rows with missing values** (NA/nulls) to avoid errors during modeling.

This ensures that the regression model only uses complete data — no blanks allowed.

*# Separate features and target*

X **=** model\_df[features]

y **=** model\_df['life\_expectancy\_at\_birth\_years']

**What it does:**

* X holds the **input features** (independent variables)
* y holds the **target variable** we’re trying to predict (dependent variable)

This split is required for model training:

* X → What we know (inputs)
* y → What we want to learn to predict (life expectancy)

### Step 2: Split Data

**from** sklearn.model\_selection **import** train\_test\_split

**What it does:**

* Imports the train\_test\_split function from **Scikit-learn**, a popular machine learning library in Python.
* This function will randomly split your dataset into **training and testing subsets**

*# Split the data: 80% training, 20% testing*

X\_train, X\_test, y\_train, y\_test **=** train\_test\_split(

X, y, test\_size**=**0.2, random\_state**=**42

)

**What it does:**

* Splits both your **features (X)** and **target (y)** into:  
  + X\_train: 80% of feature data for training
  + X\_test: 20% of feature data for testing
  + y\_train: 80% of labels (life expectancy) for training
  + y\_test: 20% of labels for evaluating model predictions

**Parameters explained:**

* test\_size=0.2: Reserve **20% of the data** for testing (common practice).
* random\_state=42: Sets a seed for **reproducibility** — ensures you get the same split every time you run the code.

### **Example Output:**

If your dataset has 1,000 rows, the result will be:

* X\_train: 800 rows of inputs
* X\_test: 200 rows of inputs
* y\_train: 800 true values (life expectancy)
* y\_test: 200 true values for testing prediction accuracy

### Step 3: Train Models

**1. Linear Regression**

**from** sklearn.linear\_model **import** LinearRegression

This imports the **LinearRegression** class from scikit-learn.

lr **=** LinearRegression()

This **creates a model instance** called lr (short for linear regression) with default settings.

lr**.**fit(X\_train, y\_train)

This **trains** the model using your training data:

* X\_train: feature variables (like GDP, HDI, etc.)
* y\_train: actual life expectancy values

y\_pred\_lr **=** lr**.**predict(X\_test)

This uses the **trained model** to **predict life expectancy** on new, unseen feature data (X\_test).

* Output: y\_pred\_lr — a list/array of predicted life expectancy values.

**2. Random Forest Regressor**

**from** sklearn.ensemble **import** RandomForestRegressor

This imports the **RandomForestRegressor** — a powerful ensemble model that builds multiple decision trees and averages their results.

rf **=** RandomForestRegressor(random\_state**=**42)

This creates a Random Forest model with:

* random\_state=42 for reproducibility (same results every time)

rf**.**fit(X\_train, y\_train)

Trains the Random Forest model on your training data, just like linear regression.

The key difference is:

* Instead of fitting a line, Random Forest fits **many decision trees** and averages their outputs.
* It can capture **non-linear relationships** and interactions between variables.

y\_pred\_rf **=** rf**.**predict(X\_test)

Predicts life expectancy values for the test set using the trained Random Forest model.

### Step 4: Model Evaluation

**from** sklearn.metrics **import** mean\_squared\_error, r2\_score

**import** numpy **as** np

* mean\_squared\_error: Measures how far predictions are from true values.
* r2\_score: Shows how well the model explains the variance in the data.
* np: Used to compute square root of MSE to get RMSE.

**def** evaluate\_model(name, y\_true, y\_pred):

print(f"--- {name} ---")

print(f"R² Score: {r2\_score(y\_true, y\_pred):.4f}")

rmse **=** np**.**sqrt(mean\_squared\_error(y\_true, y\_pred))

print(f"RMSE: {rmse:.4f}")

print()

**What it does:**

* Accepts model name, true values, and predicted values.
* Prints:  
  + **R² Score** – closer to 1.0 means better fit.
  + **RMSE (Root Mean Squared Error)** – smaller is better.

## **What’s R² Score?**

* Measures how much of the variance in the target variable (life expectancy) is explained by the model.
* Range:  
  + 1.0 = perfect prediction
  + 0.0 = model explains nothing
  + Negative = worse than just predicting the average

## **What’s RMSE?**

* Measures how far off the predictions are in the **same units as life expectancy** (years).
* Lower RMSE = more accurate predictions

*# Ensure rf and y\_pred\_rf are defined*

**if** 'rf' **not** **in** locals():

**from** sklearn.ensemble **import** RandomForestRegressor

rf **=** RandomForestRegressor(random\_state**=**42)

rf**.**fit(X\_train, y\_train)

Ensures that the RandomForestRegressor model is trained in case it wasn’t already.

**if** 'y\_pred\_rf' **not** **in** locals():

y\_pred\_rf **=** rf**.**predict(X\_test)

Makes sure predictions for the Random Forest model exist.

*# Evaluate both models*

evaluate\_model("Linear Regression", y\_test, y\_pred\_lr)

evaluate\_model("Random Forest Regressor", y\_test, y\_pred\_rf)

**What it does:**

* Runs the evaluate\_model() function twice.
* Compares results between the two models (Linear vs Random Forest).

## 7.Step-by-Step Clustering Plan

* Select relevant features
* Normalize the data
* Apply K-Means clustering
* Visualize clusters

**1. Feature Selection for Clustering**

*# Select one row per country for latest available year (e.g., 2018 or max year)*

latest\_year **=** merged\_df['year']**.**max()

* This retrieves the most recent year from the dataset (e.g., 2018 or 2021).
* This is done because clustering should compare **countries at the same time point**, not across years

df\_latest **=** merged\_df[merged\_df['year'] **==** latest\_year]

* Filters the dataset to include **only rows from the latest year**.
* This ensures fairness — comparing all countries based on the same time frame

*# Drop countries with missing values*

cluster\_features **=** [

'country', 'gdp\_per\_capita\_usd', 'population\_density\_people\_per\_sq\_km\_of\_land\_area', 'life\_expectancy\_at\_birth\_years', 'hdi', 'electric\_power\_consumption\_kwh\_per\_capita'

]

* This list defines the **features** you’ll use for clustering.
* These are **relevant development indicators** that capture economic and social well-being

df\_cluster **=** df\_latest[cluster\_features]**.**dropna()

* Sets the country name as the **index** for easier reference.
* This allows results (clusters) to be labeled clearly by country.

df\_cluster**.**set\_index('country', inplace**=True**)

* Sets the country name as the **index** for easier reference.
* This allows results (clusters) to be labeled clearly by country

*# Data to feed into the model*

X\_cluster **=** df\_cluster**.**copy()

* Makes a copy of the cleaned dataset (excluding the index).
* This is the actual data that will be used in the next step (scaling and clustering).

**2. Normalize the Data**

**from** sklearn.preprocessing **import** StandardScaler

* This imports the **StandardScaler** class from scikit-learn, which is used to **standardize features** by removing the mean and scaling to unit variance.

scaler **=** StandardScaler()

X\_scaled **=** scaler**.**fit\_transform(X\_cluster)

* This line does **two things**:

1. fit: Learn the mean and standard deviation of each column from your X\_cluster data.
2. transform: Applies the transformation to **standardize** each value.

1. After this, every column will:

* Have a **mean of 0**
* Have a **standard deviation of 1**

**3. Apply K-Means Clustering**

**from** sklearn.cluster **import** KMeans

* This imports the KMeans class from scikit-learn, a popular algorithm for unsupervised clustering.

*# Try 3 clusters (adjust based on elbow curve later)*

kmeans **=** KMeans(n\_clusters**=**3, random\_state**=**42)

* This **initializes** the KMeans model.

**Parameters**:

* n\_clusters=3: You're telling it to find **3 distinct groups** of countries. (You can experiment with different values later using an elbow curve.)
* random\_state=42: Ensures **reproducibility** of your results. Without this, the algorithm might assign clusters differently every time you run it.

df\_cluster['cluster'] **=** kmeans**.**fit\_predict(X\_scaled)

This line does two key things:

1. fit\_predict(X\_scaled):  
   * **Fits** the KMeans model to your normalized data (X\_scaled).
   * **Predicts** the cluster assignment (e.g., 0, 1, or 2) for each country.
2. df\_cluster['cluster'] = ...:  
   * Stores the predicted **cluster label** in a new column in the original dataset (df\_cluster), so now each country is assigned to a cluster.

**4. Visualize Clusters**

**from** sklearn.decomposition **import** PCA

**import** matplotlib.pyplot **as** plt

**import** seaborn **as** sns

*# Reduce to 2 dimensions for visualization*

pca **=** PCA(n\_components**=**2)

X\_pca **=** pca**.**fit\_transform(X\_scaled)

* PCA(n\_components=2): You're asking PCA to reduce all your numeric features into **2 principal components**, which capture the most variance (spread) in the data.
* fit\_transform(X\_scaled): Learns how to rotate and compress the data and then transforms it.

The output, X\_pca, is a 2D array with one row per country:

* pca1 = how far along the first principal axis (most variance)
* pca2 = how far along the second axis

df\_cluster['pca1'] **=** X\_pca[:, 0]

df\_cluster['pca2'] **=** X\_pca[:, 1]

* Add these new PCA values back to your original df\_cluster DataFrame so each country now has coordinates for plotting.

*# Scatter plot*

plt**.**figure(figsize**=**(10, 6))

sns**.**scatterplot(data**=**df\_cluster, x**=**'pca1', y**=**'pca2', hue**=**'cluster', palette**=**'Set2', s**=**100)

plt**.**title("Country Clusters Based on Development Indicators")

plt**.**xlabel("PCA Component 1")

plt**.**ylabel("PCA Component 2")

plt**.**legend(title**=**'Cluster')

plt**.**tight\_layout()

plt**.**show()