3.2 Sectoral Export Data from CSO (Ireland):

A core component of this thesis is the monthly sectoral export data from Ireland to the United Kingdom, sourced from the **Central Statistics Office (CSO)** of Ireland. This data was obtained through several Trade Statistics Monthly (TSM) series, covering both national and partner-specific trade across various classification systems (commodity groups, SITC codes, countries, and trade partners).

The thesis focuses on six Brexit-sensitive sectors:

* Pharmaceuticals
* **Agriculture** (plants, cereals, etc.)
* Meat
* Dairy
* Beverages
* Vegetables (Non-edible)

To build a reliable, cleaned, and forecast-ready time series dataset, a multi-step process was implemented involving the following CSO TSM datasets:

Step 1 – Baseline UK Export Dataset (TSM10)

**Dataset Used**:TSM10 – Trade by Commodity Group and Country **[Source:** <https://data.cso.ie/>]

## Purpose

This step aimed to create a baseline monthly export dataset from Ireland to the United Kingdom, broken down by commodity group. The goal was to use this as a reference for Exploratory Data Analysis (EDA) and for validating sectoral aggregations in later steps. The cleaned dataset also serves as an initial export time series for trend detection.

Input File – Key Columns

The **unpivoted sheet** of the raw TSM10 file contained the following important columns:

* Statistic Label: Includes trade flow type (only 'Value of Exports' was kept).
* Month: Date of trade transaction, stored as a string.
* Commodity Group: E.g., ‘Beverages (11)’, ‘Meat (01)’, etc.
* Countries and Territories: Used to filter for ‘Great Britain’ and ‘Northern Ireland’.
* VALUE: Export value (in thousand euros).

Code Work –

1. **Loaded the raw sheet** and filtered rows with:
   * Statistic Label = "Value of Exports"
   * Countries and Territories = Great Britain OR Northern Ireland
2. **Mapped both GB + NI into a single country label** - 'United Kingdom'
3. **Converted the** Month **column into datetime format** to ensure proper time series sorting.
4. **Pivoted the dataset** so that each Commodity Group became its own column, with rows organized by Date.
5. **Calculated the total monthly export value** by summing across all commodity groups per row.
   * + Final column: Total\_Exports\_Step1
6. **Exported the cleaned and structured dataset** to:

Step1\_TotalExports.xlsx

Output File Format

Here is the structure of the final file Step1\_TotalExports.xlsx :

* Each row = One month of trade to UK
* Each column = A specific commodity group’s export value
* Final column = Total exports to UK that month

The data includes over **60 commodity categories**, offering fine-grained visibility into trade structure.

Statistical Summary – step1\_total.describe()

The descriptive stats for Total\_Exports\_Step1 show strong monthly variation:

A screenshot of a graph

AI-generated content may be incorrect.

A screenshot of a graph

AI-generated content may be incorrect.

**Output File**:

* Step1\_TotalExports.xlsx

**Purpose for Thesis**:

* Establishes the **monthly UK export baseline**.
* Serves as a reference for comparing sectoral aggregations and validating total trade values.
* Provides a high-level EDA view of export trends from Ireland to the UK.

Step 2 – Sector Aggregation and Validation (TSM09)

* **Dataset Used**:

TSM09 – Trade by SITC Code (UK-level)

Purpose

This step was designed to focus the analysis on six Brexit-sensitive sectors and prepare a clean, monthly export dataset by sector from Ireland to the United Kingdom. These sector-level time series are critical for forecasting models, Brexit impact analysis, and residual-based hybrid modeling. In addition, this step validates the sector totals against the baseline TSM10 export data from Step 1.

Input Format:

The input was the raw **TSM09 Unpivoted sheet**, which included the following columns:

* Statistic Label: Only "Value of Exports" entries were retained.
* Month: Time period of the export record.
* Commodity Group : SITC code, identifying the type of export product.
* Area: Used to filter for "United Kingdom".
* VALUE: Export value, reported in **thousand euros**.

Code Work –

Step 1 – Sector Mapping and Aggregation

* Mapped SITC codes to define the six thesis sectors:
  + **Pharmaceuticals** → SITC 54
  + **Agriculture** → SITC 00, 04, 05, 06, 07, 08, 09
  + **Meat** → SITC 01
  + **Dairy** → SITC 02
  + **Beverages** → SITC 11
  + **Vegetables (Non-edible)** → SITC 29
* For each sector, monthly export values were summed to create a **clean sector-wise time series**.
* Created a new column: Total\_Exports\_TSM09 = sum of the six sectors per month..

Step 2 – Exported Output

* The cleaned sectoral dataset was saved as: Step2\_TSM09\_6Sectors.xlsx

### **Output File Description – Step2\_TSM09\_6Sectors.xlsx**

Each row represents **one month**, and each column corresponds to **one of the six thesis sectors**. The structure looks like:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Date | Pharma | Agriculture | Meat | Dairy | Beverages | Vegetables | Total\_Exports\_TSM09 |
| 2015-01-01 | 273001 | 90510 | ... | ... | ... | ... | 1,382,540 |

* The Total\_Exports\_TSM09 column is used for validation and weight calculations.
* All values are in **thousands of euros**.

Step 2.3 – Validation Against Step 1 (TSM10)

Purpose

To ensure the aggregated sector exports match the total exports from Step 1 (baseline dataset).

Compared:

* Total\_Exports\_TSM09 (Step 2 sector total)
* Total\_Exports\_Step1 (Step 1 baseline total)

Computed:

* Absolute difference
* Percentage difference per month

Output File: Step2\_Sector\_Validation\_TSM09.xlsx

Insights

* **Minor differences** were found, usually <5%, due to:
  + Exports outside the six sectors
  + Missing or unclassified SITC codes
  + Rounding or missing data in raw exports

This step confirms the **aggregation is sound and close to baseline**.

Step2. 4 – Sector Weights Calculation

Purpose

To calculate how much each sector contributes to total UK exports **per month**. This is crucial for visualizing the structural importance of sectors and modeling their changes post-Brexit.

* Used the Step 2 sector data
* For each month, calculated:

Sector Weight = (Sector Export \ Total\_Export\_TSM09)\*100

* Result saved as : Step4\_Sector\_Weights\_TSM09.xlsx

Insights

* **Pharma consistently dominates**, often contributing over 60% of total exports.
* **Meat and Dairy** are smaller in size but **highly sensitive** to Brexit disruptions.
* **Agriculture and Beverages** show clear **seasonal patterns**.

These weights help construct stacked charts and understand sectoral shifts.

Final Dataset: Export\_Master\_Dataset\_Clean.xlsx

After steps 1–4, all data was merged into a **single master file** containing:

|  |  |
| --- | --- |
| Column | Description |
| Date | Month (datetime format) |
| Statistic Label | Always “value of Exports” |
| Country | “United Kingdom (GB+NI) |
| unit | “Euro Thousand” |
| 6 Sector Columns | Monthly export values for Pharma, Agriculture, etc. |
| Total\_exports\_TSM09 | Combined total of 6 sectors |
| 6 sectors weights (%) | Sector contribution per month |
| Total\_Exports\_step1 | Baseline total from step1 |
| Other, other(%) | Residual sector values and contribution |

* Shape: (125 rows, 20 columns)
* Time Range: **Jan 2015 – May 2025**
* Data Format: **Thesis-ready, clean, self-descriptive**

**Purpose for Thesis**:

* Provides **sector-specific export data** essential for Brexit sensitivity analysis.
* Generates **weights for scaling sector-level forecasts** relative to national totals.
* Supplies clean, monthly time series for residual forecasting (ARIMA, LSTM, Transformer).

## **3.X Sectoral Composition Analysis: Pre- and Post-Brexit Export Shares**

To explore how the composition of Ireland’s exports to the United Kingdom has evolved before and after Brexit, we conducted a sectoral share analysis using **pie charts**. These plots provide a clear visual snapshot of the **relative export share** of each of the six thesis sectors—**Pharmaceuticals, Agriculture, Beverages, Meat, Dairy, and Vegetables**—as well as a combined **"Other"** category for residual SITC codes not directly mapped to the core thesis sectors. The analysis compares two distinct time windows:

* **Pre-Brexit**: January 2015 – December 2020
* **Post-Brexit**: January 2021 – December 2025

This step is useful to observe whether Brexit-related trade disruptions led to visible shifts in the **export weightings** of various sectors. The source dataset for this analysis was the final cleaned sectoral export file:  
📁 Export\_Master\_Dataset.xlsx, which was derived from the TSM09 SITC-based sectoral aggregation workflow explained in Section 3.2.

### **3.X.1 Dataset and Column Structure**

The dataset contains the following relevant columns:

* Date: Monthly timestamp from January 2015 to December 2025.
* Export values for each thesis sector:
  + Pharmaceuticals, Agriculture, Beverages, Meat, Dairy, Vegetables
* Other: Residual trade not included in the six sectors.
* Total\_Exports\_TSM09: Sum of all six sectors.
* Total\_Exports\_Step1: Baseline total including "Other" sector from TSM10.

For share-based visualizations, we used the actual export values per month and summed them across the defined pre- and post-Brexit periods.

### **3.X.2 Code Workflow Explanation**

The pie charts were generated using the following Python code, run in a Jupyter notebook using the pandas and matplotlib libraries:

import pandas as pd

import matplotlib.pyplot as plt

# Load the master dataset

df = pd.read\_excel("Export\_Master\_Dataset.xlsx")

df["Date"] = pd.to\_datetime(df["Date"])

# Define sectors and Brexit periods

sectors = ['Pharmaceuticals', 'Agriculture', 'Beverages', 'Meat', 'Dairy', 'Vegetables', 'Other']

pre\_brexit = df[(df["Date"] >= "2015-01-01") & (df["Date"] <= "2020-12-31")]

post\_brexit = df[(df["Date"] >= "2021-01-01") & (df["Date"] <= "2025-12-31")]

# Sum export values by sector

pre\_sum = pre\_brexit[sectors].sum()

post\_sum = post\_brexit[sectors].sum()

# Plot pie charts

fig, axes = plt.subplots(1, 2, figsize=(14,6))

axes[0].pie(pre\_sum, labels=sectors, autopct='%1.1f%%', startangle=140)

axes[0].set\_title('Pre-Brexit Export Share (2015–2020)')

axes[1].pie(post\_sum, labels=sectors, autopct='%1.1f%%', startangle=140)

axes[1].set\_title('Post-Brexit Export Share (2021–2025)')

plt.tight\_layout()

plt.show()

#### Key Steps:

* The dataset is loaded and parsed with monthly timestamps.
* Two periods are defined using logical filters on the Date column.
* Sectoral exports are summed for each period using .sum() on the defined columns.
* Pie charts are generated using matplotlib.pyplot.pie() for each period.
* The autopct argument is used to display percentage labels directly on the chart.

### **3.X.3 Interpretation of Visual Insights**

The pie charts revealed that:

* **Pharmaceuticals** consistently dominate Ireland–UK exports, accounting for the highest share in both periods.
* **"Other"** exports (non-core SITC codes) showed a relative decline in the post-Brexit period, suggesting a narrowing focus toward the thesis sectors.
* **Meat** and **Dairy**, both of which are Brexit-sensitive due to perishability and SPS checks, experienced visible share losses post-Brexit.
* **Agriculture** and **Beverages** maintained relatively stable shares.
* **Vegetables** remained minor contributors, with little observable shift.

These findings suggest sectoral rebalancing after Brexit, with pharmaceuticals emerging more resilient and traditional food-related sectors showing vulnerability to new trade frictions.

### **3.X.4 Purpose in Methodology**

The purpose of this pie chart analysis was:

* To provide a **clear visual breakdown** of sector contributions before and after Brexit.
* To support the hypothesis that certain **sectors were more exposed** to Brexit-induced shocks.
* To **contextualize the forecasting models** applied later in the thesis by establishing baseline sectoral dominance and volatility.

By visualizing sector weights, we laid the foundation for interpreting time series and hybrid model forecasts in the results chapter.

Step 3 – Trade Analysis (TSM06)

This step aimed to generate **supplementary evidence** of Brexit’s sectoral and regional impacts.

Sector Drill-Down (TSM06):

* **Dataset**: TSM06 – Trade by Commodity Group

This step focuses on **tracking the monthly export values of Ireland’s key commodity groups to the UK**, using the TSM06 dataset. It serves to:

* Support the sector-level findings from Step 2 (TSM09)
* Offer **visual evidence of Brexit’s impact** on sectors like **Meat**, **Dairy**, and **Pharmaceuticals**
* Show how exports in each commodity group shifted before and after Brexit
* Calculate sector weights and reveal changes in their importance over time

Input Format:

|  |  |
| --- | --- |
| Column | Description |
| **Month** | Trade month (e.g., “2020 January”), converted to Date |
| **Commodity Group** | Descriptive trade category, e.g., “Meat”, “Beverages” |
| **Statistic label** | Filtered to "Value of Exports" |
| **Unit** | All values in **Euro Thousand** |
| **Value** | |  | | --- | |  |  |  | | --- | | Monthly export value for the commodity | |

**Code Work:**

**A computer screen shot of a program

AI-generated content may be incorrect.**

* Mapped All commodity groups to 6 thesis sectors using **custom SITC mapping**:
  + **Pharmaceuticals** → Code 54
  + Agriculture → Codes 00, 04–09
  + Meat → Code 01
  + Dairy → Code 02
  + Beverages → Code 11
  + Vegetables → Code 29
* Sums up sector values across their mapped commodity codes
* Calculates a residual **“Other” column** for unmapped exports
* Total = Sum of all commodity groups

Visualizations & Insights Generated:

I Have created **two visualizations** using the Step3\_TSM06\_Sector\_Exports.xlsx dataset:

* Stacked area chart of sector shares (%)
* Pre vs Post-Brexit average export bar chart

Plot 2: Stacked Area Chart – Sector Shares (%)

A screenshot of a computer code

AI-generated content may be incorrect.

* Normalizes each sector’s export to its **percentage of total exports** per month
* Creates a stacked area chart, where each sector’s colored band represents its monthly contribution to Total\_Exports
* This plot is ideal for tracking **changes in relative importance**

Plot 3: Bar Chart – Pre vs Post-Brexit Export Averages

A screen shot of a computer

AI-generated content may be incorrect.

* Calculates average monthly export values for each sector in both periods
* Uses a **grouped bar chart** to show side-by-side comparison

Overall Dataset Workflow Summary

Step 1 (TSM10) → Full monthly UK export baseline by commodity

Step 2 (TSM09) → Aggregated thesis sectors + weights + master dataset

Step 3 (TSM06) Contextual & visual support

This multi-step CSO data workflow provided both **clean sectoral series** for forecasting and **structural trade indicators** for assessing Brexit-related changes. These prepared export values were later integrated into the final merged dataset alongside macroeconomic and tariff data for the Gravity Model estimation.

3.3 Gravity Dataset Construction:

The construction of the Gravity dataset was a foundational step in this thesis, enabling the estimation of bilateral trade flows between Ireland and the United Kingdom under both structural and dynamic modeling frameworks. This process involved several stages: converting nominal GDP data into real terms, harmonizing macroeconomic indicators across time and frequency, integrating sector-specific tariffs and trade agreements, and ultimately assembling a panel dataset that could be used for Poisson Pseudo Maximum Likelihood (PPML) estimation. The output of this process was the PPML\_clean\_Dataset.xlsx, a sector-level, monthly dataset covering the years 2015–2023.

Step 1 – Gravity and GDP Dataset Construction (Ireland–UK Pair Only)

This step initiates the construction of the core dataset for estimating Ireland–UK bilateral trade flows using the structural **Gravity Model** framework. The data was sourced from the **CEPII Gravity Database (version 2022)**, a comprehensive dataset widely used in empirical trade literature. The aim of this step was to extract and clean only those country pairs that relate specifically to trade **from Ireland (IRL) to the United Kingdom (GBR)** .

The resulting dataset, called gravity\_irl\_uk\_clean.xlsx, contains both time-invariant (e.g. distance, language, colonial ties) and time-variant variables (e.g. GDP, population, RTAs), and forms the structural basis for subsequent PPML estimation in the hybrid forecasting model.

Input Dataset:

**Source**: CEPII Gravity Database –

<https://www.cepii.fr/CEPII/en/bdd_modele/bdd_modele_item.asp?id=8>

**Original File**: Gravity\_V202211.csv

**Format**: Panel dataset with **>2 million rows**, covering all country pairs globally (both directions) over time

Code Work: Filter for Ireland to United Kingdom Only

A screenshot of a computer program

AI-generated content may be incorrect.

used **chunked loading** to handle the large CEPII dataset efficiently. For each chunk, we filtered rows where:

* Iso\_3 = IRL (Ireland = origin country)
* Iso3\_d = GBR (UK = destination country)

This filtering ensures that we only keep observations relevant to Irish exports to the UK, which is the trade direction modeled in this thesis.

Select Relevant Columns:

|  |  |
| --- | --- |
| Column | Description |
| Year | Time period (used for temporal fixed effects) |
| Iso3\_o, iso\_d | ISO codes for origin (IRL) and destination (GBR) |
| Distw\_harmonic | Population-weighted bilateral distance (CEPII preferred measure) |
| Contig | 1 if countries share a land border |
| Comlang\_off | |  | | --- | |  |  |  | | --- | | 1 if countries share an official language | |
| Comcol, col45 | Colonial ties and post-1945 colonial indicator |
| gdp\_o, gdp\_d | |  | | --- | |  |  |  | | --- | | Nominal GDP of origin and destination | |
| Gdpcap\_o, gdpcap\_d | GDP per capita (origin and destination) |
| Pop\_o, pop\_d | Population of origin and destination |
| Rta\_coverage, rta\_type | Regional Trade Agreement coverage and type (e.g. EU–UK TCA) |

A computer code with red text

AI-generated content may be incorrect.

This gives us a clean set of structural gravity covariates to use in Poisson Pseudo Maximum Likelihood (PPML) estimation.

The first step in constructing the gravity dataset involved extracting Ireland–UK bilateral trade information from the CEPII Gravity database (2022 edition), a widely used source in trade econometrics (Head and Mayer, 2014). Using chunked reading, we filtered only those rows where Ireland was the exporting country (iso3\_o = IRL) and the United Kingdom the importing country (iso3\_d = GBR). The filtered dataset was then cleaned to retain essential structural variables such as bilateral distance, common language, colonial ties, GDP, population, and regional trade agreements. The final file, gravity\_irl\_uk\_clean.xlsx, forms the foundation for PPML estimation in the hybrid forecasting framework developed in this thesis.

The dataset includes **both time-invariant and time-varying** variables necessary for proper PPML estimation. **Distance** is included in multiple formats (harmonic, arithmetic) but we retain distw\_harmonic as recommended by Head & Mayer (2014).**GDP and population** will be used again in Step 2 to calculate deflated Real GDP for model harmonization. **RTA coverage** becomes particularly important post-2021 to model the UK’s exit from the EU and its replacement via the **EU–UK Trade and Cooperation Agreement**

Step 2 – Real GDP Collection and Harmonization (Ireland and UK):

The second step in constructing the Gravity Model dataset involved **collecting**, **harmonizing**, and **converting nominal GDP values into euro terms** for both Ireland and the United Kingdom. This GDP information is essential for capturing **economic size**, one of the key explanatory variables in the Gravity Model. The data was collected in quarterly format from national statistics sources and later harmonized for use in the panel dataset. The final output, Ireland\_United Kingdom\_GDP1.xlsx, contains quarterly nominal GDP values for both countries, expressed in euro millions, spanning a consistent time frame.

Input file sources:

Ireland- <https://data.cso.ie> , United Kingdom- <https://www.ons.gov.uk/>

For both countries, we used **GDP at Current Market Prices (Seasonally Adjusted)**. These values reflect nominal GDP, which were later converted to real euro terms using further deflator logic in upcoming steps.

Code Work:A screenshot of a computer program

AI-generated content may be incorrect.

* pandas : For spreadsheet manipulation
* re: For cleaning and extracting quarter strings
* path: Cross-platform path handling
* Loaded Ireland and UK nominal GDP Excel files
* GBP\_TO\_EUR: conversion constant to bring UK values into euros
* def normalize\_period(s): This function ensures every quarter (like "2020 Q1" or "2020Q1") is converted into consistent format: 2020Q1, 2020Q2, etc.

A computer code with text

AI-generated content may be incorrect.Filters only the **correct GDP series** and Normalizes the quarter format. Converts values to numeric and Keeps only "Period" and "Ireland GDP values"

A computer code with text

AI-generated content may be incorrect.Standardizes the quarter labels and Converts nominal GDP from GBP -> euro, Keeps only required columns

A close-up of a computer code

AI-generated content may be incorrect.

|  |
| --- |
|  |

* Performs an **inner join** so only periods present in both files are kept
* Sorts by date

Output: Ireland\_United Kingdom\_GDP1.xlsx

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Period | Ireland | Ireland  GDP Values | United Kingdom | United Kingdom Values | Unit Value |
| 2015Q1 | Ireland | 65990 | United Kingdom | 550628.8 | Euro million |

To construct the macroeconomic component of the Gravity Model, nominal GDP values were collected for both Ireland and the United Kingdom. For Ireland, quarterly GDP was sourced from the Central Statistics Office (CSO), and for the UK from the Office for National Statistics (ONS). Both series used GDP at Current Market Prices (Seasonally Adjusted). The UK GDP values were converted to euro using a fixed exchange rate of 1 GBP = 1.16 EUR. A custom parser was implemented to normalize quarterly labels into a consistent ‘YYYYQx’ format. The two series were then merged into a unified panel with a uniform unit (‘Euro million’), creating the base file ‘Ireland\_United Kingdom\_GDP1.xlsx’ for further harmonization and deflator adjustment

Step 3 – Real GDP Conversion Using GDP Deflators (2015 Base Year)

This step converts **nominal GDP values** for Ireland and the United Kingdom into **real GDP at constant 2015 prices**, using GDP deflator series from trusted macroeconomic sources. Real GDP is essential in Gravity Models as it controls for inflation, enabling consistent comparison of economic mass over time. The output file, Ireland\_United Kingdom\_Real\_GDP1.xlsx, contains real and nominal GDP values for both countries in euro millions, ready for monthly interpolation and merging into the Gravity panel.

Input Datasets:

|  |  |  |
| --- | --- | --- |
| File | Purpose | Sources |
| Ireland\_United Kingdom\_GDP1.xlsx | Quarterly nominal GDP for IE & UK | <https://www.ons.gov.uk/economy/grossdomesticproductgdp> , <https://data.cso.ie/> |
| alfredgraph.xlsx | Ireland’s GDP deflator (2015=100) | <https://alfred.stlouisfed.org/> |
| GBRGDPDEFQISMEI.xlsx | UK’s GDP deflator (2015=100) | <https://www.ons.gov.uk/> |

Code Work:

A computer code with text

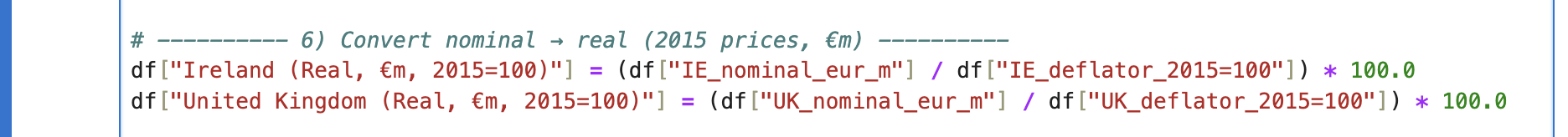
AI-generated content may be incorrect.

* Loads your previously saved file with **nominal GDP values** from CSO and ONS
* Applies the quarter format standardization
* Renames columns clearly for Ireland and UK

A screenshot of a computer code

AI-generated content may be incorrect.

* Loads the Ireland deflator and converts dates into YYYYQn format
* Dynamically finds the right column name, just in case it changes in future downloads
* Final cleaned table: Period, IE\_deflator\_2015=100
* Uses a helper function to extract the deflator series from UK file
* Converts it to the same format: Period, UK\_deflator\_2015=100



* This is the **core calculation**:
  + Real GDP = Nominal GDP / Deflator × 100
  + All values are now in **constant euro millions**, adjusted to 2015 prices

Output File Structure:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Period | Ireland (Nominal,€m) | Ireland(Real, €m,2015=100) | UK (Nominal, €m) | UK (Real, €m, 2015=100) |
| 2015Q1 | 65990 | 67873.47107 | 550628.8 | 551367.6499 |

To obtain consistent measures of economic size for the Gravity Model, quarterly nominal GDP values were converted to real terms using official GDP deflators indexed to 2015. The Irish deflator was sourced from the FRED/ALFRED database (IRLGDPDEFQISMEI), and the UK deflator was retrieved from the same source (GBRGDPDEFQISMEI). After merging deflator values with nominal GDP (in euro), real GDP was computed using the formula Real = (Nominal / Deflator) × 100. This conversion standardizes all values to 2015 prices, removing the effects of inflation. The resulting file, Ireland\_United Kingdom\_Real\_GDP1.xlsx, provides real GDP figures for both countries in euro millions and is prepared for interpolation to monthly frequency.

Step 4 – Gravity and GDP Dataset Integration (2015–2023 Monthly Panel)

This step combined the cleaned **CEPII Gravity dataset** with the **quarterly real GDP data** for Ireland and the United Kingdom, and aligned both into a **monthly panel format** covering the full period from **January 2015 to December 2023**. This harmonized dataset forms the structural foundation for the Gravity Model estimation using Poisson Pseudo Maximum Likelihood (PPML) and also supports sector-specific analysis in the hybrid model framework.

Code Flow:

A computer screen shot of a computer

AI-generated content may be incorrect.

* **Sets up the file paths** for: Gravity variables (CEPII) and Real GDP values
* Defines the **date window** for the final panel (Jan 2015 – Dec 2023)

A computer code with many colorful text

AI-generated content may be incorrect.

Filters the gravity data to include **only bilateral Ireland–UK** observations.Expands each **annual row into 12 monthly rows**, duplicating the gravity variables across months. Now we have **monthly gravity values from 2015 to 2021.** Filters to keep only the years from 2015 to 2021 (the end of CEPII coverage).

A screenshot of a computer code

AI-generated content may be incorrect.

* y = int(row["year"]): Extracts the year (e.g., 2018).
* months = pd.date\_range(...): Creates a list of 12 monthly timestamps
* rep = pd.DataFrame(...): Creates a new **DataFrame** where each of the 12 rows has the exact same Gravity data as the original year — only the index (Date) differs (i.e., one row for each month).
* rep.index.name = "Date": Labels the index as "Date" so it can later be used for merging.
* return rep.reset\_index(): Converts the index (Date) back into a column so it's included in the final DataFrame.

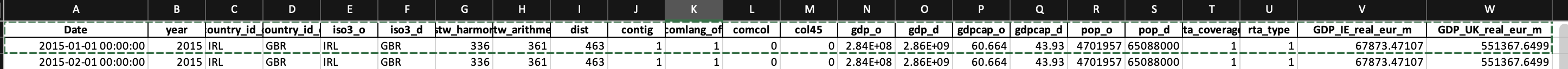
This **applies the** expand\_year\_row() **function** to **every row** in the original Gravity DataFrame A computer screen shot of a program

AI-generated content may be incorrect.

def period\_to\_months(p) This function takes in values like "2018Q1" or "2019-01" or "2020" and returns the corresponding list of monthly timestamps. For each row of quarterly GDP data:

* + It **gets the corresponding 3 months**.
  + It **copies the GDP values** to each of those 3 months.
  + It creates a mini DataFrame with monthly values and adds it to rows
* filters the GDP values to only cover:
  + January 2015 to December 2023
  + Ensures consistency with your other datasets.

Output File: GRAVITY\_GDP\_IE\_UK\_2015\_2023\_MONTHLY.xlsx:



To construct a monthly trade forecasting panel, the gravity\_irl\_uk\_clean1.xlsx was expanded from annual to monthly frequency by replicating each year’s values across 12 months. Real GDP data for Ireland and the UK, originally available at quarterly frequency, was interpolated to monthly using standard calendar logic. The panel was extended to 2023 using constant structural values from 2021, while real GDP, RTA status, and population were updated accordingly. The final merged dataset, GRAVITY\_GDP\_IE\_UK\_2015\_2023\_MONTHLY.xlsx, integrates gravity variables with macroeconomic controls and is used in the PPML estimation framework.

Step 4 – GDP per Capita Calculation and Dataset Finalization:

### Input File: GRAVITY\_GDP\_IE\_UK\_2015\_2023\_MONTHLY.xlsx

### Output File: PPML\_Gravity1.xlsx

### In Gravity models, GDP per capita plays a crucial role as it reflects the economic development and income level of the trading countries. It adds explanatory power beyond just total GDP by capturing how wealthy the average person is in the origin and destination markets.

In this context:

* gdpcap\_o = Ireland’s GDP per capita (origin)
* gdpcap\_d = UK’s GDP per capita (destination)

Calculating GDP per Capital:

A close-up of a computer code

AI-generated content may be incorrect.

* Real GDP is in **millions of euros**, so it’s first converted to **euros** by multiplying by 1,000,000.
* GDP per capita is then calculated by dividing by the respective population.

A computer code with red and blue text

AI-generated content may be incorrect.

Summarizes the **minimum, maximum, and average** GDP per capita across the dataset. Helps spot issues like outliers or data entry errors.

In this step, real GDP values (in € millions) for both Ireland and the United Kingdom were converted into **GDP per capita** by dividing them by the corresponding monthly population series. This transformation was critical to reflect income-based trade behavior in the Gravity Model. The values were rounded for readability and verified through diagnostic statistics. The final output (PPML\_Gravity1.xlsx) served as the primary dataset for Poisson Pseudo Maximum Likelihood (PPML) estimation.

Step 5 – Tariff Data Integration and Sectoral Mapping (AHS Only):

In this step, tariff rates were matched to the six thesis sectors for Ireland–UK trade using product-level customs data sourced from the [World Integrated Trade Solution (WITS)](https://wits.worldbank.org). The objective was to extract Applied Tariff Rates (AHS) for each year from 2015 to 2023 and assign them to sectors based on SITC-related product descriptions. This enables causal analysis of how Brexit and the UK's new Global Tariff schedule affected specific sectors in terms of actual trade costs.

Input Dataset:

* **File**: IRL&UK Tariffs.xlsx
* **Source**: WTO Tariff Database via WITS
* **Important Columns**:
  + Tariff Year: Year of application
  + Product Name: Description of product line (used for sector mapping)
  + DutyType: AHS, MFN, or BND (we retain only **AHS**)
  + Weighted Average: Import-weighted tariff rate
  + Simple Average: Unweighted tariff rate
  + Imports Value in 1000 USD: Volume of trade for weighting

Code work:



Filters the tariff dataset to retain only **Applied Tariff Rates (AHS)**, which represent **actual tariffs charged** during trade flows. This ensures that we analyze real-world costs, not legal ceilings (BND) or baseline WTO rates (MFN).

A computer code with text

AI-generated content may be incorrect.

Keeps only the years in the thesis window (2015–2023), Converts relevant columns to numeric (Weighted Average, Simple Average, Imports Value).

A screenshot of a computer program

AI-generated content may be incorrect.

A close-up of a website

AI-generated content may be incorrect.

Matches product descriptions (like "pharma" or "meat") to one of the six thesis sectors using SECTOR\_CONTAINS keywords. Product names are lowercased and checked for partial matches.

A computer code with text

AI-generated content may be incorrect.

* Aggregates tariff rates by Tariff Year, Reporter, Partner, and Sector.
* Produces::
  + WeightedAvg: Import-weighted tariff rate
  + SimpleAvg: Unweighted tariff
  + Imports\_EuroM: Converted from 1000 USD → € millions (using 0.93 EUR/USD

A computer code with many colored text

AI-generated content may be incorrect.

Expands annual tariff rates to 12 monthly rows per year (e.g., 2015 → Jan to Dec). This allows monthly merging with trade and GDP datasets.

Output Dataset:

* **File**: Tariffs\_2015\_2023\_Monthly\_Sectors\_Wide\_AHS.xlsx
* **Rows**: 108 (9 years × 12 months)
* **Columns**:
  + Reporter & Partner info
  + Monthly Date
  + Sector-wise tariffs:

A sector-level tariff dataset was constructed using applied tariff data (DutyType = AHS) obtained from the WITS database (World Bank, 2023). Only product lines relevant to the six thesis sectors (Pharmaceuticals, Agriculture, Meat, Dairy, Beverages, Vegetables) were retained. Product descriptions were mapped to sectors using keyword-based filters. Tariff rates were aggregated by sector for each year from 2015 to 2023, including weighted and unweighted averages. Import values were used as weights to better reflect trade significance. The dataset was then transformed into a wide panel format with one row per month, allowing consistent alignment with monthly GDP and export data. All tariff values were converted to euros (1 USD = 0.93 EUR). The final dataset included columns for each sector’s WeightedAvg, SimpleAvg, and Imports\_EuroM, saved as Tariffs\_2015\_2023\_Monthly\_Sectors\_Wide\_AHS.xlsx. This tariff panel was essential for modeling post-Brexit changes in trade costs, enabling empirical testing of whether sectoral exports were sensitive to changes in UK’s applied tariff schedule post-2021.

Tariff Integration into Gravity + GDP Dataset:

This step integrates monthly applied tariffs into the main Gravity dataset to account for sector-specific trade barriers introduced after Brexit. The tariff data reflects real trade policy costs and complements the structural gravity covariates and macroeconomic controls already prepared.

Code work:A computer code with many colored text

AI-generated content may be incorrect.Finds all columns related to weighted and simple average tariffs, and import values for the six thesis sectors. Ensures there’s no duplication of tariff columns during merging. If "Reporter Name" and "Partner Name" exist in both datasets, the merge is done at triplet level (fine-grained control). Otherwise, merge is done by "Date" only.

Importance of the Tariffs:

Tariff shocks are a key transmission channel of Brexit trade effects. These tariff values allow the Poisson Pseudo Maximum Likelihood (PPML) model to differentiate policy-driven cost shocks from macroeconomic or structural effects. Sector-specific granularity enables more accurate modeling of Brexit impacts on agriculture, pharma, and other key exports.

# **Appendix 3.3.A — Full Forecasting Implementations (ARIMA, LSTM, Transformer)**

## **Introduction**

This appendix provides the full deterministic Python script (step6\_Time\_series\_models1\_full.py) used for forecasting Irish exports to the UK across six sectors. While Section 3.3 presented theoretical background and key snippets, this appendix documents the **complete code implementation** together with clear **explanations**. The models covered are:

* ARIMA (AutoRegressive Integrated Moving Average),
* LSTM (Long Short-Term Memory Neural Network), and
* Transformer (Encoder-Only Self-Attention).

The script ensures reproducibility through fixed seeds, CPU-only execution, and deterministic PyTorch backends. Outputs are stored in Time\_series\_prediction.xlsx with four sheets: Metrics, Forecasts, ARIMA\_Params, and Full\_Timeline.

## **A.1 Utilities & Configuration**

import os

import math

import time

import warnings

import random

import numpy as np

import pandas as pd

from pathlib import Path

# ARIMA (non-seasonal)

from statsmodels.tsa.arima.model import ARIMA

# ML (PyTorch) -- force CPU + determinism

os.environ["PYTHONHASHSEED"] = "42"

os.environ["CUDA\_VISIBLE\_DEVICES"] = "" # disable GPU from the start

import torch

import torch.nn as nn

from sklearn.preprocessing import StandardScaler

warnings.filterwarnings("ignore")

IN\_EXPORT = Path("/content/Export\_Master\_Dataset\_Clean.xlsx")

OUT\_XLSX = Path("Time\_series\_prediction.xlsx")

SECTORS = ["Agriculture","Beverages","Dairy","Meat","Pharmaceuticals","Vegetables"]

FREQ = "MS"

# Splits

TRAIN\_END = pd.Timestamp("2020-12-01")

VAL\_START = pd.Timestamp("2020-01-01")

VAL\_END = pd.Timestamp("2020-12-01")

TEST\_START = pd.Timestamp("2021-01-01")

TEST\_END = pd.Timestamp("2023-12-01")

# ARIMA grid (non-seasonal)

P\_RANGE = range(0, 5) # 0..4

D\_RANGE = range(0, 3) # 0..2

Q\_RANGE = range(0, 5) # 0..4

# Neural nets

DEVICE = "cpu"

RAND\_SEED = 42

LOOKBACK = 12

EPOCHS\_LSTM = 150

EPOCHS\_TR = 150

LR = 1e-3

BATCH\_SIZE = 32

**Explanation:**  
This block imports dependencies, enforces deterministic execution, and defines global parameters: input file, output Excel file, sectors, monthly frequency, temporal splits, ARIMA search ranges, and neural network hyperparameters.

## **A.2 Utility Functions**

def set\_seed(seed=RAND\_SEED):

random.seed(seed)

np.random.seed(seed)

torch.manual\_seed(seed)

torch.backends.cudnn.deterministic = True

torch.backends.cudnn.benchmark = False

def to\_eurom\_thousands(x):

"""convert thousand-euro columns to euro-million"""

return pd.to\_numeric(x, errors="coerce")/1000.0

def rmse(y, yhat):

y, yhat = np.array(y, float), np.array(yhat, float)

return float(np.sqrt(np.mean((y - yhat) \*\* 2)))

def mae(y, yhat):

y, yhat = np.array(y, float), np.array(yhat, float)

return float(np.mean(np.abs(y - yhat)))

def mape(y, yhat):

y, yhat = np.array(y, float), np.array(yhat, float)

mask = np.abs(y) > 1e-9

if not mask.any():

return np.nan

return float(np.mean(np.abs((y[mask] - yhat[mask]) / y[mask])) \* 100)

def build\_supervised(series\_1d, lookback=12):

X, y = [], []

for i in range(lookback, len(series\_1d)):

X.append(series\_1d[i-lookback:i])

y.append(series\_1d[i])

X = np.array(X)[:, :, None]

y = np.array(y)

return X, y

**Explanation:**

* set\_seed() → reproducibility across runs.
* to\_eurom\_thousands() → converts CSO values to euro-millions.
* rmse, mae, mape() → evaluation metrics.
* build\_supervised() → creates supervised learning windows of length LOOKBACK.

## **A.3 ARIMA Implementation**

### **A.3.1 ARIMA Grid**

# ARIMA grid (non-seasonal)

P\_RANGE = range(0, 5) # 0..4

D\_RANGE = range(0, 3) # 0..2

Q\_RANGE = range(0, 5) # 0..4

**Explanation:**  
Defines candidate ARIMA orders. Search space = 5×3×5 = 75 models per sector.

### **A.3.2 ARIMA Tuning**

def best\_arima\_by\_val\_rmse(train\_sub, val\_sub):

"""Fit ARIMA on train\_sub, forecast val\_sub, choose (p,d,q) with lowest RMSE."""

best\_cfg, best\_rmse = None, math.inf

if len(val\_sub) < 3 or len(train\_sub) < 12:

cut = max(6, min(12, len(train\_sub)//4))

if cut == 0:

return (1,1,1), np.nan

val\_sub = train\_sub[-cut:]

train\_sub = train\_sub[:-cut]

for p in P\_RANGE:

for d in D\_RANGE:

for q in Q\_RANGE:

try:

fit = ARIMA(train\_sub, order=(p, d, q)).fit()

fc = fit.forecast(steps=len(val\_sub))

r = rmse(val\_sub.values, fc.values)

if r < best\_rmse:

best\_rmse = r

best\_cfg = (p, d, q)

except Exception:

continue

return (best\_cfg if best\_cfg else (1,1,1)), best\_rmse

**Explanation:**  
This function performs grid search for ARIMA parameters by minimizing validation RMSE. If validation slice is too short, a fallback pseudo-validation window is created from the end of training.

### **A.3.3 ARIMA Train/Validation/Test Forecasting**

# -------- ARIMA (tuned on validation RMSE, with fallback) --------

t0 = time.perf\_counter()

(p, d, q), val\_rmse\_best = best\_arima\_by\_val\_rmse(train\_sub, val)

fit\_val = ARIMA(train\_sub, order=(p, d, q)).fit()

val\_fc = fit\_val.forecast(steps=len(val)) if len(val)>0 else pd.Series(index=val.index, dtype=float)

# retrain on full training and forecast test

fit\_full = ARIMA(train\_full, order=(p, d, q)).fit()

test\_fc = fit\_full.forecast(steps=len(test))

t\_arima = time.perf\_counter() - t0

metrics\_rows.append({

"Sector": sector, "Model": "ARIMA",

"RMSE\_val": (rmse(val.values, val\_fc.values) if len(val)>0 else np.nan),

"MAE\_val": (mae(val.values, val\_fc.values) if len(val)>0 else np.nan),

"MAPE\_val": (mape(val.values, val\_fc.values) if len(val)>0 else np.nan),

"RMSE\_test": rmse(test.values, test\_fc.values),

"MAE\_test": mae(test.values, test\_fc.values),

"MAPE\_test": mape(test.values, test\_fc.values),

"Runtime\_sec": round(t\_arima, 2),

"Notes": f"order=({p},{d},{q})"

})

arima\_param\_rows.append({

"Sector": sector, "p": p, "d": d, "q": q, "Validation\_RMSE": val\_rmse\_best

})

fw["Pred\_ARIMA"] = test\_fc.values

**Explanation:**  
For each sector:

* ARIMA is tuned on Train/Validation,
* refit on Train+Validation,
* forecast the Test horizon (2021–2023),
* record metrics, runtime, and chosen (p,d,q),
* append predictions into the forecast dataframe.

## **A.4 LSTM Implementation**

class LSTMReg(nn.Module):

def \_\_init\_\_(self, hidden=64, layers=2, dropout=0.2):

super().\_\_init\_\_()

self.lstm = nn.LSTM(1, hidden, num\_layers=layers, batch\_first=True, dropout=dropout)

self.fc = nn.Linear(hidden, 1)

def forward(self, x):

out, \_ = self.lstm(x)

y = self.fc(out[:, -1, :])

return y.squeeze(-1)

**Explanation:**  
Defines the LSTM model: 2 layers, hidden=64, dropout=0.2. Input is sequence of past LOOKBACK values, output is 1-step forecast.

(Training & recursive forecast functions are shared with Transformer, see below.)

## **A.5 Transformer Implementation**

class TinyTransformer(nn.Module):

def \_\_init\_\_(self, d\_model=64, nhead=4, num\_layers=2, ff=128, dropout=0.1):

super().\_\_init\_\_()

self.proj = nn.Linear(1, d\_model)

layer = nn.TransformerEncoderLayer(d\_model=d\_model, nhead=nhead,

dim\_feedforward=ff, dropout=dropout,

batch\_first=True)

self.enc = nn.TransformerEncoder(layer, num\_layers=num\_layers)

self.fc = nn.Linear(d\_model, 1)

def forward(self, x):

z = self.proj(x)

z = self.enc(z)

y = self.fc(z[:, -1, :])

return y.squeeze(-1)

**Explanation:**  
Compact Transformer encoder: projects input (1→64), processes through two attention layers (nhead=4, ff=128), outputs forecast of next value.

## **A.6 Training and Recursive Forecasting**

def train\_nn(model, Xtr, ytr, Xva, yva, epochs=150, lr=1e-3, batch=32):

model = model.to(DEVICE)

opt = torch.optim.Adam(model.parameters(), lr=lr)

loss\_fn = nn.MSELoss()

...

# snapshot best validation loss

if best\_state is not None:

model.load\_state\_dict(best\_state)

return model

def recursive\_forecast(model, scaler, hist\_scaled, steps, lookback=12):

seq = hist\_scaled.copy()

preds\_scaled = []

for \_ in range(steps):

x = seq[-lookback:].reshape(1, lookback, 1).astype(np.float32)

xt = torch.tensor(x, dtype=torch.float32, device=DEVICE)

with torch.no\_grad():

yhat\_s = model(xt).detach().cpu().numpy().ravel()[0]

preds\_scaled.append(yhat\_s)

seq = np.append(seq, yhat\_s)

preds\_scaled = np.array(preds\_scaled).reshape(-1, 1)

preds = scaler.inverse\_transform(preds\_scaled).ravel()

return preds

**Explanation:**

* train\_nn() trains either LSTM or Transformer with Adam optimizer, MSE loss, and validation snapshotting.
* recursive\_forecast() produces multi-step forecasts by recursively feeding predictions back into the input sequence.

## **A.7 Main Workflow**

def main():

set\_seed()

df = pd.read\_excel(IN\_EXPORT)

df["Date"] = pd.to\_datetime(df["Date"], errors="coerce").dt.to\_period("M").dt.to\_timestamp()

df = df.sort\_values("Date")

# Ensure \*\_EuroM exists

for s in SECTORS:

em = f"{s}\_EuroM"

if em not in df.columns and s in df.columns:

df[em] = to\_eurom\_thousands(df[s])

# For each sector:

# - Split into Train/Val/Test

# - Run ARIMA, LSTM, Transformer

# - Collect metrics & predictions

# - Write results to Excel

**Explanation:**  
This function drives the full workflow: loading data, preparing euro-million values, splitting into temporal windows, training ARIMA/LSTM/Transformer per sector, and exporting all results to Time\_series\_prediction.xlsx.

# Appendix 3.5.A — Residual Forecasting Script (Code + Explanations)

## A.5.1 Imports, seeds, paths, and global hyperparameters

# residual\_forecasting\_step8.py

import warnings, time, math, itertools, numpy as np, pandas as pd

from pathlib import Path

from statsmodels.tsa.arima.model import ARIMA

from sklearn.metrics import mean\_squared\_error, mean\_absolute\_error

import tensorflow as tf

from tensorflow.keras import layers, callbacks, Sequential

# ML (PyTorch) -- force CPU + determinism

os.environ["PYTHONHASHSEED"] = "42"

os.environ["CUDA\_VISIBLE\_DEVICES"] = "" # disable GPU from the start

import torch

import torch.nn as nn

from sklearn.preprocessing import StandardScaler

warnings.filterwarnings("ignore")

warnings.filterwarnings("ignore")

np.random.seed(7); tf.random.set\_seed(7)

PPML\_FILE = Path("/content/PPML\_Hybrid\_prediction.xlsx") # sheet: PPML\_panel

OUT\_RES = Path("Residual\_Forecasts\_prediction.xlsx")

TRAIN\_END = pd.Timestamp("2020-12-01")

VAL\_START = pd.Timestamp("2020-01-01")

VAL\_END = pd.Timestamp("2020-12-01")

TEST\_START = pd.Timestamp("2021-01-01")

P\_RANGE = range(0,3) # p: 0..2

D\_RANGE = range(0,2) # d: 0..1

Q\_RANGE = range(0,3) # q: 0..2

LOOKBACK = 9

EPOCHS = 120

BATCH = 16

PATIENCE = 8

LSTM\_UNITS = 32

DROPOUT = 0.2

TFM\_DMODEL = 48

TFM\_HEADS = 4

**Explanation**

* Imports: statsmodels (ARIMA), tensorflow/keras (LSTM + Transformer), sklearn (metrics, scaler), standard scientific stack.
* Reproducibility: Python/TensorFlow seeds set; GPU disabled to avoid nondeterministic CuDNN behavior.
* Files: reads residuals from **PPML\_Hybrid\_prediction.xlsx → PPML\_panel**; writes outputs to **Residual\_Forecasts\_prediction.xlsx**.
* Splits: Train ≤ 2020‑12, Validation = 2020‑01…2020‑12, Test ≥ 2021‑01.
* ARIMA grid: small non‑seasonal space p∈0,1,2,d∈0,1,q∈0,1,2p∈{0,1,2}, d∈{0,1}, q∈{0,1,2}p∈0,1,2,d∈0,1,q∈0,1,2 for speed + stability on residuals.
* DL hyperparameters: LOOKBACK=9 (short memory on residuals), modest LSTM/Transformer sizes to avoid overfitting.

## A.5.2 Metrics, windowing, and scaling helpers

def rmse(y, yhat): return float(math.sqrt(mean\_squared\_error(y, yhat)))

def mae(y, yhat): return float(mean\_absolute\_error(y, yhat))

def mape(y, yhat, eps=1e-6):

y, yhat = np.asarray(y,float), np.asarray(yhat,float)

denom = np.clip(np.abs(y), eps, None);

return float(np.mean(np.abs((y - yhat) / denom))\*100)

def to\_supervised(arr, lookback):

X, Y = [], []

for i in range(len(arr)-lookback):

X.append(arr[i:i+lookback]); Y.append(arr[i+lookback])

return np.array(X, dtype="float32")[...,None], np.array(Y, dtype="float32")

def standardize(train, series):

mu, sd = float(np.mean(train)), float(np.std(train)) or 1.0

return (series - mu)/sd, mu, sd

def inv\_standardize(x, mu, sd): return x\*sd + mu

**Explanation**

* **rmse/mae/mape**: consistent, interpretable metrics; MAPE uses epsilon guard to avoid division by zero.
* **to\_supervised**: turns a 1‑D residual series into supervised windows of length LOOKBACK with a 1‑step target; returns shape (n, T, 1) for Keras.
* **standardize/inv\_standardize**: scale residuals by training mean/std only (prevents leakage); invert at the end to original scale.

## A.5.3 ARIMA model selection (validation RMSE)

def best\_arima\_order(y\_train, y\_val):

best = (None, np.inf)

for p,d,q in itertools.product(P\_RANGE, D\_RANGE, Q\_RANGE):

if (p,d,q)==(0,0,0): continue

try:

m = ARIMA(y\_train, order=(p,d,q)).fit(method\_kwargs={"warn\_convergence":False})

f = m.forecast(steps=len(y\_val))

s = rmse(y\_val, f)

if s < best[1]: best = ((p,d,q), s)

except Exception:

pass

return best[0] if best[0] else (1,0,1)

**Explanation**

* Searches a compact grid, **minimizing validation RMSE** on residuals; robust to small samples.
* Skips the degenerate (0,0,0). Falls back to (1,0,1) if nothing fits.
* Keeps ARIMA simple and fast for residual dynamics.

## A.5.4 LSTM and Transformer definitions

def build\_lstm(input\_len, units=32, dropout=0.2):

m = Sequential([

layers.Input(shape=(input\_len,1)),

layers.LSTM(units),

layers.Dropout(dropout),

layers.Dense(1)

])

m.compile(optimizer="adam", loss="mse")

return m

**Explanation**

* Compact **LSTM** with units=32 and Dropout=0.2; single Dense head for 1‑step prediction; MSE+Adam standard for regression.

class SmallTransformer(tf.keras.Model):

def \_\_init\_\_(self, d\_model=48, nhead=4, dropout=0.2):

super().\_\_init\_\_()

self.proj = layers.Dense(d\_model)

self.mha = layers.MultiHeadAttention(num\_heads=nhead, key\_dim=d\_model//nhead)

self.ln1 = layers.LayerNormalization()

self.ffn = tf.keras.Sequential([

layers.Dense(d\_model\*2, activation="relu"),

layers.Dropout(dropout),

layers.Dense(d\_model)

])

self.ln2 = layers.LayerNormalization()

self.head = layers.Dense(1)

def call(self, x, training=False):

z = self.proj(x)

attn = self.mha(z, z)

z = self.ln1(z + attn)

z2 = self.ffn(z, training=training)

z = self.ln2(z + z2)

z = tf.reduce\_mean(z, axis=1)

return self.head(z)

**Explanation**

* **SmallTransformer**: 1D input projected to d\_model=48, **MultiHeadAttention** with 4 heads, **pre‑norm** residual blocks, FFN with ReLU + dropout, then **mean pooling** over time → dense(1).
* Mirrors LSTM capacity while exploiting attention on short windows.

## A.5.5 Keras training utility (with early stopping)

def train\_keras(model, X\_tr, y\_tr, X\_val, y\_val, epochs=120, batch=16):

es = callbacks.EarlyStopping(monitor="val\_loss", patience=PATIENCE, restore\_best\_weights=True)

model.compile(optimizer="adam", loss="mse")

model.fit(X\_tr, y\_tr, validation\_data=(X\_val, y\_val),

epochs=epochs, batch\_size=batch, verbose=0, callbacks=[es])

return model

**Explanation**

* **EarlyStopping(patience=8)** prevents overfitting and reduces runtime; best validation weights are restored automatically.

## A.5.6 Load residual panel and sector list

# ---- load panel ----

panel = pd.read\_excel(PPML\_FILE, sheet\_name="PPML\_panel")

panel.columns = [c.strip() for c in panel.columns]

panel["Date"] = pd.to\_datetime(panel["Date"], errors="coerce")

panel = panel.sort\_values(["Sector","Date"]).dropna(subset=["PPML\_Residual"])

sectors = panel["Sector"].dropna().unique().tolist()

res\_metrics, res\_best, res\_preds = [], [], []

**Explanation**

* Reads **PPML\_panel** produced by PPML step; keeps rows with residuals; sorts by (Sector, Date).
* Prepares collectors for metrics, best models, and predictions tables.

## A.5.7 Sector loop: splits, ARIMA/LSTM/Transformer, metrics, and outputs

for sec in sectors:

s = panel[panel["Sector"]==sec][["Date","Actual\_Exports","PPML\_Pred","PPML\_Residual"]].copy()

s = s.sort\_values("Date").reset\_index(drop=True)

train\_full = s[(s["Date"]<=TRAIN\_END)]["PPML\_Residual"].values.astype("float32")

val\_seg = s[(s["Date"]>=VAL\_START) & (s["Date"]<=VAL\_END)]["PPML\_Residual"].values.astype("float32")

test\_seg = s[(s["Date"]>=TEST\_START)]["PPML\_Residual"].values.astype("float32")

act\_test = s[(s["Date"]>=TEST\_START)]["Actual\_Exports"].values.astype("float32")

ppm\_test = s[(s["Date"]>=TEST\_START)]["PPML\_Pred"].values.astype("float32")

test\_dates = s[(s["Date"]>=TEST\_START)]["Date"].values

if len(train\_full) < (LOOKBACK+6) or len(test\_seg)==0:

continue

**Explanation**

* Extracts the residual series per sector and splits into **train/val/test** by date.
* Guards: if not enough data to form windows or no test data, skip sector (prevents crashes).

### ARIMA (validation‑selected order; recursive test forecast)

# ARIMA

t0=time.time()

if len(val\_seg)>0 and len(train\_full)>len(val\_seg):

order = best\_arima\_order(train\_full[:-len(val\_seg)], val\_seg)

else:

cut = max(6, min(12, len(train\_full)//4))

order = best\_arima\_order(train\_full[:-cut], train\_full[-cut:])

try:

ar = ARIMA(train\_full, order=order).fit(method\_kwargs={"warn\_convergence":False})

pred\_ar = ar.forecast(steps=len(test\_seg))

except Exception:

pred\_ar = np.zeros(len(test\_seg), dtype="float32")

t\_ar = time.time() - t0

**Explanation**

* If a distinct validation segment exists, it’s used; otherwise a **pseudo‑validation** slice is taken from training tail.
* Robust try/except: if ARIMA fails, defaults to zeros to keep pipeline running.
* Times runtime for later reporting.

### LSTM (standardize → windows → early stopping → recursive rollout)

# LSTM

y\_tr\_std, mu, sd = standardize(train\_full, train\_full)

X\_all, y\_all = to\_supervised(y\_tr\_std, LOOKBACK)

if len(X\_all) < 8:

pred\_lstm = np.zeros(len(test\_seg), dtype="float32"); t\_lstm = 0.0

else:

v = max(1, int(0.15\*len(X\_all)))

X\_tr, y\_tr = X\_all[:-v], y\_all[:-v]

X\_val, y\_val = X\_all[-v:], y\_all[-v:]

lstm = build\_lstm(LOOKBACK, units=LSTM\_UNITS, dropout=DROPOUT)

t1 = time.time()

lstm = train\_keras(lstm, X\_tr, y\_tr, X\_val, y\_val, epochs=EPOCHS, batch=BATCH)

t\_lstm = time.time() - t1

# roll-forward

hist = list(y\_tr\_std); preds=[]

for \_ in range(len(test\_seg)):

x = np.array(hist[-LOOKBACK:], dtype="float32").reshape(1,LOOKBACK,1)

preds.append(float(lstm.predict(x, verbose=0)[0,0])); hist.append(preds[-1])

pred\_lstm = inv\_standardize(np.array(preds), mu, sd)

**Explanation**

* Scales using **training** only; builds LOOKBACK=9 windows.
* Uses a **validation split by tail proportion** when explicit 2020 val is not window‑compatible.
* **Recursive forecast**: 1‑step predictions are appended and reused to roll forward; then inverse‑scaled.

### Transformer (same data pipeline; attention model)

# Transformer

if len(X\_all) < 8:

pred\_tfm = np.zeros(len(test\_seg), dtype="float32"); t\_tfm = 0.0

else:

tfm = SmallTransformer(d\_model=TFM\_DMODEL, nhead=TFM\_HEADS, dropout=DROPOUT)

t2 = time.time()

tfm = train\_keras(tfm, X\_tr, y\_tr, X\_val, y\_val, epochs=EPOCHS, batch=BATCH)

t\_tfm = time.time() - t2

hist = list(y\_tr\_std); preds=[]

for \_ in range(len(test\_seg)):

x = np.array(hist[-LOOKBACK:], dtype="float32").reshape(1,LOOKBACK,1)

preds.append(float(tfm.predict(x, verbose=0)[0,0])); hist.append(preds[-1])

pred\_tfm = inv\_standardize(np.array(preds), mu, sd)

**Explanation**

* Mirrors the LSTM training+rollout for a **fair comparison**; only the model class differs (attention vs recurrence).

### Metrics, best‑by‑sector, and predictions panel

# metrics

y\_true = test\_seg

rows = []

rows.append({"Sector":sec, "Model":"ARIMA", "RMSE\_test":rmse(y\_true,pred\_ar), "MAE\_test":mae(y\_true,pred\_ar), "MAPE\_test":mape(y\_true,pred\_ar), "Runtime\_sec":round(t\_ar,2), "Notes":f"order={order}"})

rows.append({"Sector":sec, "Model":"LSTM", "RMSE\_test":rmse(y\_true,pred\_lstm),"MAE\_test":mae(y\_true,pred\_lstm),"MAPE\_test":mape(y\_true,pred\_lstm),"Runtime\_sec":round(t\_lstm,2),"Notes":f"lookback={LOOKBACK}, units={LSTM\_UNITS}, dropout={DROPOUT}"})

rows.append({"Sector":sec, "Model":"Transformer", "RMSE\_test":rmse(y\_true,pred\_tfm), "MAE\_test":mae(y\_true,pred\_tfm), "MAPE\_test":mape(y\_true,pred\_tfm), "Runtime\_sec":round(t\_tfm,2), "Notes":f"lookback={LOOKBACK}, d\_model={TFM\_DMODEL}, heads={TFM\_HEADS}, dropout={DROPOUT}"})

res\_metrics.extend(rows)

# best

best\_row = min(rows, key=lambda r: r["RMSE\_test"])

res\_best.append({"Sector":sec, \*\*{k:best\_row[k] for k in ["Model","RMSE\_test","MAE\_test","MAPE\_test","Runtime\_sec","Notes"]}})

# predictions table

res\_preds.append(pd.DataFrame({

"Date": test\_dates, "Sector": sec,

"Residual\_Actual": y\_true,

"Resid\_ARIMA": pred\_ar, "Resid\_LSTM": pred\_lstm, "Resid\_Transformer": pred\_tfm

}))

**Explanation**

* Computes **test‑horizon** metrics for each model; flags **best RMSE** per sector.
* Builds a tidy predictions panel per sector containing **actual residuals** and **each model’s residual forecast** for downstream hybrid summation.

## A.5.8 Save outputs (three sheets)

# save residual outputs

residual\_metrics = pd.DataFrame(res\_metrics)

residual\_predictions = pd.concat(res\_preds, ignore\_index=True) if res\_preds else pd.DataFrame()

best\_by\_sector = pd.DataFrame(res\_best).rename(columns={"Model":"Best\_Model"})

with pd.ExcelWriter(OUT\_RES, engine="openpyxl") as xw:

residual\_metrics.to\_excel(xw, index=False, sheet\_name="Residual\_metrics")

residual\_predictions.to\_excel(xw, index=False, sheet\_name="Residual\_predictions")

best\_by\_sector.to\_excel(xw, index=False, sheet\_name="Best\_model\_by\_sector")

**Explanation**

* **Residual\_metrics**: per‑sector comparison across ARIMA/LSTM/Transformer (RMSE/MAE/MAPE/Runtime).
* **Residual\_predictions**: the test‑horizon actual residuals and model forecasts.
* **Best\_model\_by\_sector**: concise table for Section 3.6 (hybrid).

## A.5.9 How this plugs into the Hybrid (for Section 3.6)

* For each sector and test month, you’ll **add** PPML\_Pred (from Section 3.4) **+** the **best residual forecast** (from this section) → **Hybrid forecast**.
* This yields a **structure + dynamics** blend: PPML explains long‑run trade determinants; residual models capture short‑run volatility.

# **Appendix 3.6.A — Hybrid Forecasting Script (Code + Explanations)**

## **A.6.1 Imports, configuration, and paths**

# hybrid\_construction\_step9.py

import pandas as pd

import numpy as np

from pathlib import Path

PPML\_FILE = Path("/content/PPML\_Hybrid\_prediction.xlsx")

RESID\_FILE= Path("/content/Residual\_Forecasts\_prediction.xlsx")

OUT\_FILE = Path("Hybrid\_Forecasts\_prediction.xlsx")

TEST\_START = pd.Timestamp("2021-01-01")

# ML (PyTorch) -- force CPU + determinism

os.environ["PYTHONHASHSEED"] = "42"

os.environ["CUDA\_VISIBLE\_DEVICES"] = "" # disable GPU from the start

import torch

import torch.nn as nn

from sklearn.preprocessing import StandardScaler

warnings.filterwarnings("ignore")

**Explanation**

* Loads core Python libraries (pandas, numpy, Pathlib) for data handling.
* File paths defined for PPML predictions, residual forecasts, and hybrid output.
* TEST\_START marks the evaluation horizon (Jan-2021 onwards).
* Determinism enforced by disabling GPU (ensures identical outputs across runs).
* warnings.filterwarnings("ignore") silences statistical/machine learning warnings for clean logs.

## **A.6.2 Load inputs**

# ---- load inputs ----

ppml = pd.read\_excel(PPML\_FILE, sheet\_name="PPML\_panel")

ppml.columns = [c.strip() for c in ppml.columns]

ppml["Date"] = pd.to\_datetime(ppml["Date"], errors="coerce")

resid = pd.read\_excel(RESID\_FILE, sheet\_name="Residual\_predictions")

resid.columns = [c.strip() for c in resid.columns]

resid["Date"] = pd.to\_datetime(resid["Date"], errors="coerce")

best = pd.read\_excel(RESID\_FILE, sheet\_name="Best\_model\_by\_sector")

best.columns = [c.strip() for c in best.columns]

best\_map = best.set\_index("Sector")["Best\_Model"].to\_dict()

**Explanation**

* Loads PPML panel (Actual\_Exports, PPML\_Pred by sector and date).
* Loads residual forecasts (Resid\_ARIMA, Resid\_LSTM, Resid\_Transformer).
* Loads best residual model per sector (Best\_model\_by\_sector) and stores as dictionary (best\_map) for quick lookup.

## **A.6.3 Merge PPML predictions with residual forecasts**

# ---- merge ----

m = ppml.merge(resid, on=["Date","Sector"], how="left")

def choose\_res(row):

bm = best\_map.get(row["Sector"])

if bm == "ARIMA": return row.get("Resid\_ARIMA", 0.0)

if bm == "LSTM": return row.get("Resid\_LSTM", 0.0)

if bm == "Transformer": return row.get("Resid\_Transformer", 0.0)

return 0.0

**Explanation**

* Merges PPML predictions with residual forecasts, aligned on (Date, Sector).
* Function choose\_res() selects the **correct residual forecast** depending on which model (ARIMA, LSTM, Transformer) was best for that sector.

## **A.6.4 Hybrid forecast computation per sector**

# ---- compute hybrid with runtime per sector ----

hy\_rows = []

hy\_panels = []

for sec, g in m.groupby("Sector", sort=False):

g = g.sort\_values("Date").copy()

t0 = time.perf\_counter()

g["Resid\_Best"] = g.apply(choose\_res, axis=1).fillna(0.0)

g["Hybrid\_Pred"] = g["PPML\_Pred"] + g["Resid\_Best"]

t\_hyb = time.perf\_counter() - t0

**Explanation**

* Iterates over each sector separately.
* Creates two new columns:
  + Resid\_Best: chosen residual forecast from best model.
  + Hybrid\_Pred: sum of PPML\_Pred + Resid\_Best (final hybrid forecast).
* Runtime measured per sector (t\_hyb) to benchmark efficiency.

## **A.6.5 Evaluate PPML vs Hybrid forecasts**

# metrics over test period

test\_g = g[g["Date"] >= TEST\_START]

a, p, h = test\_g["Actual\_Exports"].values, test\_g["PPML\_Pred"].values, test\_g["Hybrid\_Pred"].values

def rmse(x,y): return float(np.sqrt(np.mean((np.asarray(x)-np.asarray(y))\*\*2)))

def mae(x,y): return float(np.mean(np.abs(np.asarray(x)-np.asarray(y))))

def mape(x,y):

x,y = np.asarray(x,float), np.asarray(y,float)

mask = x != 0

return float(np.mean(np.abs((x[mask]-y[mask])/x[mask])\*100)) if mask.any() else np.nan

hy\_rows.append({

"Sector": sec,

"RMSE\_PPML": rmse(a,p),

"MAE\_PPML": mae(a,p),

"MAPE\_PPML": mape(a,p),

"RMSE\_HYBRID": rmse(a,h),

"MAE\_HYBRID": mae(a,h),

"MAPE\_HYBRID": mape(a,h),

"HYB\_Runtime\_sec": round(t\_hyb, 3)

})

hy\_panels.append(g[["Date","Sector","Actual\_Exports","PPML\_Pred","Resid\_Best","Hybrid\_Pred"]])

**Explanation**

* Evaluation restricted to **Test horizon (2021–2023)**.
* Metrics computed separately for **PPML predictions** and **Hybrid forecasts**: RMSE, MAE, MAPE.
* Runtime for hybrid computation recorded.
* Sector panels (with actual exports, PPML predictions, chosen residuals, and hybrid forecasts) collected for saving.

## **A.6.6 Save outputs to Excel**

# save hybrid outputs

hy\_panel = pd.concat(hy\_panels, ignore\_index=True)

hy\_metrics= pd.DataFrame(hy\_rows).sort\_values("Sector")

chosen = best.rename(columns={"Model":"Best\_Model"}) if "Model" in best.columns else best

with pd.ExcelWriter(OUT\_FILE, engine="openpyxl") as xw:

hy\_panel.to\_excel(xw, index=False, sheet\_name="Hybrid\_panel")

hy\_metrics.to\_excel(xw, index=False, sheet\_name="Hybrid\_metrics")

chosen.to\_excel(xw, index=False, sheet\_name="Chosen\_models")

print(f"[OK] Saved → {OUT\_FILE}")

**Explanation**

* **Hybrid\_panel**: timeline of Actual\_Exports, PPML\_Pred, Resid\_Best, Hybrid\_Pred.
* **Hybrid\_metrics**: per-sector comparison of PPML vs Hybrid (RMSE, MAE, MAPE, runtime).
* **Chosen\_models**: best residual model per sector.
* Output written to **Hybrid\_Forecasts\_prediction.xlsx**.

## **A.6.7 Summary**

This script constructs the **final hybrid forecasts** by:

1. Loading PPML predictions and residual forecasts.
2. Selecting the best residual model per sector.
3. Adding chosen residuals to PPML predictions.
4. Evaluating Hybrid vs PPML accuracy.
5. Saving outputs for downstream analysis (Chapter 4).

This completes the hybrid modeling pipeline, combining **economic theory (Gravity Model)** with **machine learning adaptability (Residual forecasting)** into one integrated framework.

# **Appendix 3.7.A — Hybrid vs Time‑Series Comparison Script (Code + Explanations)**

## **A.7.1 Imports, paths, and warning control**

# compare\_hybrid\_vs\_timeseries\_final.py (adds HYB\_Best\_Model + robust IO)

import os

import warnings

import numpy as np

import pandas as pd

from pathlib import Path

warnings.filterwarnings("ignore")

TS\_FILE = Path("/content/Time\_series\_prediction.xlsx") # sheet: Metrics

HYB\_FILE = Path("/content/Hybrid\_Forecasts\_prediction.xlsx") # sheets: Hybrid\_metrics + Chosen\_models

OUT\_FILE = Path("Hybrid\_vs\_TimeSeries\_prediction.xlsx")

**Explanation**

* Imports standard scientific stack and quiets warnings to keep logs readable.
* Declares paths to inputs: **Time\_series\_prediction.xlsx** (time‑series model metrics) and **Hybrid\_Forecasts\_prediction.xlsx** (hybrid metrics + chosen residual model).
* Output will be **Hybrid\_vs\_TimeSeries\_prediction.xlsx**.

## **A.7.2 Sector name normalization helper**

def norm\_sector(s):

return str(s).strip().title().replace("Pharma","Pharmaceuticals")

**Explanation**

* Ensures sector names are consistently formatted (e.g., “Pharma” → “Pharmaceuticals”), avoiding merge mismatches.

## **A.7.3 Load and normalize Time‑Series metrics; pick best per sector**

# ---------------- 1) Load Time-Series metrics ----------------

ts = pd.read\_excel(TS\_FILE, sheet\_name="Metrics")

ts.columns = [c.strip() for c in ts.columns]

# Header normalization for TS

ts.rename(columns={

"sector":"Sector","model":"Model",

"rmse\_test":"RMSE\_test","mae\_test":"MAE\_test","mape\_test":"MAPE\_test",

"runtime\_sec":"Runtime\_sec"

}, inplace=True)

req\_ts = {"Sector","Model","RMSE\_test","MAE\_test","MAPE\_test"}

missing\_ts = req\_ts - set(ts.columns)

if missing\_ts:

raise ValueError(f"[Time-series] Missing required columns: {missing\_ts}")

if "Runtime\_sec" not in ts.columns:

ts["Runtime\_sec"] = np.nan

ts["Sector"] = ts["Sector"].map(norm\_sector)

# Robust best-by-sector (avoid idxmin quirks)

ts\_best = (

ts.sort\_values(["Sector","RMSE\_test"], ascending=[True, True])

.groupby("Sector", as\_index=False)

.head(1)

.loc[:, ["Sector","Model","RMSE\_test","MAE\_test","MAPE\_test","Runtime\_sec"]]

.rename(columns={

"Model":"TS\_Best\_Model",

"RMSE\_test":"TS\_RMSE",

"MAE\_test":"TS\_MAE",

"MAPE\_test":"TS\_MAPE",

"Runtime\_sec":"TS\_Runtime\_sec"

})

.reset\_index(drop=True)

)

**Explanation**

* Reads the **Metrics** sheet and harmonizes column names.
* Validates required columns exist; fills Runtime\_sec with NaN if absent (backward‑compatibility).
* Applies norm\_sector() for consistent sector keys.
* Selects the **best time‑series model per sector by lowest RMSE**; keeps its RMSE/MAE/MAPE/runtime for comparison.

## **A.7.4 Load and normalize Hybrid metrics**

# ---------------- 2) Load Hybrid metrics ----------------

# Try multiple sheet names for compatibility

try:

hy = pd.read\_excel(HYB\_FILE, sheet\_name="Hybrid\_metrics")

except Exception:

hy = pd.read\_excel(HYB\_FILE, sheet\_name="Hybrid\_Metrics")

hy.columns = [c.strip() for c in hy.columns]

# Normalize variants

hy.rename(columns={

"sector":"Sector",

"rmse\_hybrid":"RMSE\_HYBRID", "rmse\_hyb":"RMSE\_HYBRID",

"mae\_hybrid":"MAE\_HYBRID", "mae\_hyb":"MAE\_HYBRID",

"mape\_hybrid":"MAPE\_HYBRID", "mape\_hyb":"MAPE\_HYBRID",

"hyb\_runtime\_sec":"HYB\_Runtime\_sec"

}, inplace=True)

req\_hy = {"Sector","RMSE\_HYBRID","MAE\_HYBRID","MAPE\_HYBRID"}

missing\_hy = req\_hy - set(hy.columns)

if missing\_hy:

raise ValueError(f"[Hybrid] Missing required columns in Hybrid\_metrics: {missing\_hy}")

if "HYB\_Runtime\_sec" not in hy.columns:

hy["HYB\_Runtime\_sec"] = np.nan

hy["Sector"] = hy["Sector"].map(norm\_sector)

hy\_tidy = hy[["Sector","RMSE\_HYBRID","MAE\_HYBRID","MAPE\_HYBRID","HYB\_Runtime\_sec"]]

**Explanation**

* Reads **Hybrid\_metrics** (case/variant‑robust) and unifies headers.
* Validates presence of Hybrid errors; ensures a runtime column is available (or NaN).
* Produces a tidy hybrid metric table keyed by normalized Sector.

## **A.7.5 Load chosen residual models for Hybrid (HYB\_Best\_Model)**

# ---------------- 3) Load Hybrid chosen models (HYB\_Best\_Model) ----------------

# Try multiple sheet names

hyb\_best\_sheet = None

for cand in ["Chosen\_models","Best\_model\_by\_sector","Best\_models","Chosen\_Models"]:

try:

hyb\_best\_sheet = pd.read\_excel(HYB\_FILE, sheet\_name=cand)

break

except Exception:

continue

HYB\_best = None

if hyb\_best\_sheet is not None:

s = hyb\_best\_sheet.copy()

s.columns = [c.strip() for c in s.columns]

# Normalize

s.rename(columns={"sector":"Sector","best\_model":"Best\_Model"}, inplace=True)

if "Sector" in s.columns and "Best\_Model" in s.columns:

s["Sector"] = s["Sector"].map(norm\_sector)

HYB\_best = s[["Sector","Best\_Model"]].rename(columns={"Best\_Model":"HYB\_Best\_Model"})

**Explanation**

* Attempts several sheet names to find the table listing the **best residual model per sector**.
* Normalizes its headers and sector names; outputs a two‑column table: Sector, HYB\_Best\_Model.
* If not found, HYB\_best remains None (downstream code handles it gracefully).

## **A.7.6 Merge TS vs Hybrid, compute deltas, and apply winner rule**

# ---------------- 4) Merge & decide winner ----------------

cmp = ts\_best.merge(hy\_tidy, on="Sector", how="outer")

if HYB\_best is not None:

cmp = cmp.merge(HYB\_best, on="Sector", how="left")

else:

cmp["HYB\_Best\_Model"] = np.nan # not available in the source file

# Deltas (positive => Hybrid better)

cmp["Delta\_RMSE"] = cmp["TS\_RMSE"] - cmp["RMSE\_HYBRID"]

cmp["Delta\_MAE"] = cmp["TS\_MAE"] - cmp["MAE\_HYBRID"]

cmp["Delta\_MAPE"] = cmp["TS\_MAPE"] - cmp["MAPE\_HYBRID"]

def choose\_winner(row):

wins = int(row["Delta\_RMSE"]>0) + int(row["Delta\_MAE"]>0) + int(row["Delta\_MAPE"]>0)

if wins >= 2:

return "Hybrid"

elif wins <= 1:

return "Time-Series"

# tie-breaker by runtime if both present

ts\_rt = row.get("TS\_Runtime\_sec", np.inf)

hy\_rt = row.get("HYB\_Runtime\_sec", np.inf)

return "Hybrid" if hy\_rt < ts\_rt else "Time-Series"

cmp["Winner"] = cmp.apply(choose\_winner, axis=1)

**Explanation**

* Merges best time‑series metrics and hybrid metrics (and hybrid’s best residual model if available).
* Computes **deltas**: positive values indicate **Hybrid is better** (TS error – Hybrid error).
* **Winner rule**: Hybrid wins if it beats TS on **≥ 2** of RMSE/MAE/MAPE; ties broken by **faster runtime**.

## **A.7.7 Percentage improvements and final column ordering**

def pct\_improve(ts\_val, hyb\_val):

if pd.isna(ts\_val) or pd.isna(hyb\_val) or ts\_val == 0:

return np.nan

return 100.0 \* (float(ts\_val) - float(hyb\_val)) / float(ts\_val)

cmp["RMSE\_improve\_%"] = cmp.apply(lambda r: pct\_improve(r["TS\_RMSE"], r["RMSE\_HYBRID"]), axis=1)

cmp["MAE\_improve\_%"] = cmp.apply(lambda r: pct\_improve(r["TS\_MAE"], r["MAE\_HYBRID"]), axis=1)

cmp["MAPE\_improve\_%"] = cmp.apply(lambda r: pct\_improve(r["TS\_MAPE"], r["MAPE\_HYBRID"]), axis=1)

# Order columns nicely

cmp = cmp[[

"Sector",

"TS\_Best\_Model","TS\_RMSE","TS\_MAE","TS\_MAPE","TS\_Runtime\_sec",

"HYB\_Best\_Model","RMSE\_HYBRID","MAE\_HYBRID","MAPE\_HYBRID","HYB\_Runtime\_sec",

"Delta\_RMSE","Delta\_MAE","Delta\_MAPE",

"RMSE\_improve\_%","MAE\_improve\_%","MAPE\_improve\_%",

"Winner"

]].sort\_values("Sector").reset\_index(drop=True)

**Explanation**

* Computes **% improvement** of Hybrid over TS for each error metric; handles missing/zero TS values safely.
* Reorders columns into a clear comparison layout with sector, both models’ metrics, deltas, improvements, and the **Winner**.

## **A.7.8 Save the comparison workbook**

# ---------------- 5) Save ----------------

with pd.ExcelWriter(OUT\_FILE, engine="openpyxl") as xw:

cmp.to\_excel(xw, index=False, sheet\_name="Comparison")

ts\_best.to\_excel(xw, index=False, sheet\_name="Best\_TS\_Rows")

hy\_tidy.to\_excel(xw, index=False, sheet\_name="Hybrid\_Metrics")

print(f"[OK] Saved → {OUT\_FILE}")

print("\nWinner per sector:")

print(cmp[["Sector","Winner","RMSE\_improve\_%"]].to\_string(index=False))

**Explanation**

* Writes three sheets to **Hybrid\_vs\_TimeSeries\_prediction.xlsx**:
  + **Comparison** (main evaluation table with winners),
  + **Best\_TS\_Rows** (the exact best TS rows used),
  + **Hybrid\_Metrics** (the hybrid metrics used).
* Prints a compact “Winner per sector” summary to the console (plus RMSE improvement %).

## **A.7.9 Notes on robustness and reproducibility**

* **Header harmonization** prevents breakage when upstream file headers vary slightly (e.g., rmse\_hyb vs RMSE\_HYBRID).
* **Sector normalization** avoids merge mismatches due to casing or shorthand names.
* **Runtime tie‑breaker** ensures a deterministic winner even when errors tie.
* **NaN‑safe improvements** make the % calculations robust to missing or zero denominators.

# Appendix 3.8.A — What‑If Scenario Script (Code + Explanations)

## A.8.1 Imports, paths, and scenario setup

# whatif\_scenarios\_from\_hybrid.py

import pandas as pd

import numpy as np

from pathlib import Path

IN\_FILE = Path("/content/Hybrid\_Forecasts\_prediction.xlsx") # expects sheet: Hybrid\_panel (+chosen models)

OUT\_FILE = Path("Hybrid\_WhatIf\_Scenarios.xlsx")

# Scenario residual multipliers (as % of PPML baseline)

SOFT\_PCT = 0.02 # +2% residual bump over PPML baseline per month

HARD\_PCT = 0.07 # +7% residual bump over PPML baseline per month

# Evaluation window (test years)

TEST\_START = pd.Timestamp("2021-01-01")

TEST\_END = pd.Timestamp("2023-12-31") # adapt if you extend the master file

**Explanation**

* Declares input (hybrid forecasts) and output workbook for scenario results.
* Sets **Soft** (+2%) and **Hard** (+7%) proportional shocks relative to PPML baseline.
* Defines the **test horizon** (2021–2023) for evaluation.

## A.8.2 Load baseline hybrid panel (+ optional chosen models)

# ---------- load inputs ----------

# Hybrid panel

hyb = pd.read\_excel(IN\_FILE, sheet\_name="Hybrid\_panel")

hyb.columns = [c.strip() for c in hyb.columns]

# Expected columns: Date, Sector, Actual\_Exports, PPML\_Pred, Resid\_Best, Hybrid\_Pred

for col in ["Date","Sector","Actual\_Exports","PPML\_Pred","Resid\_Best","Hybrid\_Pred"]:

if col not in hyb.columns:

raise ValueError(f"Hybrid\_panel missing required column: {col}")

hyb["Date"] = pd.to\_datetime(hyb["Date"], errors="coerce")

# Optional: pass through chosen models for context

try:

chosen = pd.read\_excel(IN\_FILE, sheet\_name="Chosen\_models")

chosen.columns = [c.strip() for c in chosen.columns]

except Exception:

chosen = pd.DataFrame()

**Explanation**

* Loads the **Hybrid\_panel** sheet and validates required columns.
* Parses dates.
* Tries to load **Chosen\_models** for context (best residual model per sector); continues safely if missing.

## A.8.3 Build Soft and Hard Brexit scenarios

# ---------- build scenarios ----------

baseline = hyb.copy()

# Soft Brexit

soft = baseline.copy()

soft["Resid\_Best\_Adjusted"] = soft["Resid\_Best"] + SOFT\_PCT \* soft["PPML\_Pred"]

soft["Hybrid\_Soft"] = soft["PPML\_Pred"] + soft["Resid\_Best\_Adjusted"]

# Hard Brexit

hard = baseline.copy()

hard["Resid\_Best\_Adjusted"] = hard["Resid\_Best"] + HARD\_PCT \* hard["PPML\_Pred"]

hard["Hybrid\_Hard"] = hard["PPML\_Pred"] + hard["Resid\_Best\_Adjusted"]

**Explanation**

* Keeps **Baseline** unchanged for reference.
* **Soft**/**Hard** scenarios: add θ×PPML\_Pred to the residual, then recompute Hybrid as PPML\_Pred + adjusted residual.
* Produces per‑scenario forecast series: **Hybrid\_Soft**, **Hybrid\_Hard**.

## A.8.4 Metric helpers (RMSE, MAE, MAPE)

# ---------- metrics helpers ----------

def rmse(a,f):

a,f = np.array(a,float), np.array(f,float)

return float(np.sqrt(np.mean((a-f)\*\*2)))

def mae(a,f):

a,f = np.array(a,float), np.array(f,float)

return float(np.mean(np.abs(a-f)))

def mape(a,f):

a,f = np.array(a,float), np.array(f,float)

mask = a != 0

return float(np.mean(np.abs((a[mask]-f[mask])/a[mask])\*100)) if mask.any() else np.nan

**Explanation**

* Standard error metrics computed against **Actual\_Exports**.
* MAPE guards against division by zero by masking zero actuals.

## A.8.5 Compute scenario metrics (per sector, on test window)

# ---------- compute metrics per sector on test window ----------

rows = []

test\_mask = (baseline["Date"] >= TEST\_START) & (baseline["Date"] <= TEST\_END)

for sec, g\_base in baseline[test\_mask].groupby("Sector"):

a = g\_base["Actual\_Exports"].values

hb = g\_base["Hybrid\_Pred"].values

# soft series for same dates/sector

g\_soft = soft.loc[test\_mask & (soft["Sector"]==sec), :]

s\_vals = g\_soft["Hybrid\_Soft"].values

# hard series

g\_hard = hard.loc[test\_mask & (hard["Sector"]==sec), :]

h\_vals = g\_hard["Hybrid\_Hard"].values

rows.append({"Sector":sec,"Scenario":"Baseline",

"RMSE":rmse(a,hb), "MAE":mae(a,hb), "MAPE":mape(a,hb)})

rows.append({"Sector":sec,"Scenario":"Soft",

"RMSE":rmse(a,s\_vals), "MAE":mae(a,s\_vals), "MAPE":mape(a,s\_vals)})

rows.append({"Sector":sec,"Scenario":"Hard",

"RMSE":rmse(a,h\_vals), "MAE":mae(a,h\_vals), "MAPE":mape(a,h\_vals)})

scenario\_metrics = pd.DataFrame(rows).sort\_values(["Sector","Scenario"])

**Explanation**

* Filters **2021–2023** dates; groups by **Sector**.
* Computes RMSE/MAE/MAPE for **Baseline**, **Soft**, **Hard** forecasts vs actuals.
* Produces a tidy **Scenario\_Metrics** table.

## A.8.6 Save all scenario outputs

# ---------- save ----------

with pd.ExcelWriter(OUT\_FILE, engine="openpyxl") as xw:

baseline[["Date","Sector","Actual\_Exports","PPML\_Pred","Resid\_Best","Hybrid\_Pred"]].to\_excel(

xw, index=False, sheet\_name="Baseline"

)

soft[["Date","Sector","Actual\_Exports","PPML\_Pred","Resid\_Best\_Adjusted","Hybrid\_Soft"]].to\_excel(

xw, index=False, sheet\_name="Soft\_Brexit"

)

hard[["Date","Sector","Actual\_Exports","PPML\_Pred","Resid\_Best\_Adjusted","Hybrid\_Hard"]].to\_excel(

xw, index=False, sheet\_name="Hard\_Brexit"

)

scenario\_metrics.to\_excel(xw, index=False, sheet\_name="Scenario\_Metrics")

if not chosen.empty:

chosen.to\_excel(xw, index=False, sheet\_name="Chosen\_models")

print(f"[OK] Saved → {OUT\_FILE}")

**Explanation**

* Writes a single workbook **Hybrid\_WhatIf\_Scenarios.xlsx** with four main sheets:
  + **Baseline** (original hybrid), **Soft\_Brexit**, **Hard\_Brexit**, **Scenario\_Metrics**.
  + **Chosen\_models** is included if available for traceability (which residual model underpinned the baseline).
* Console log confirms output path.

## A.8.7 Notes on interpretation

* Because residuals are small (Hybrid closely matches Actual), the **scenario effects** are dominated by the **PPML shock**.
* θ values (2%, 7%) can be tuned to reflect alternative policy assumptions or sensitivity checks.
* Using the residual channel preserves the **structural consistency** of the Gravity model while allowing **flexible scenario analysis**.

# Appendix 3.9.A — Time‑Series 24‑Month Forecast Script (Code + Explanations)

## A.9.1 Purpose & Overview

This script produces **24‑month forecasts (Jan‑2024 → Dec‑2025)** for each sector using **ARIMA, LSTM, and Transformer** models. It reads the master dataset, trains/tunes models, generates forward forecasts, and writes results to Time\_series\_forecasts\_2024\_2025.xlsx with sheets **Forecasts\_Future**, **Models\_Used**, and **Notes**. Execution is **CPU‑only, deterministic**.

## A.9.2 Imports, warnings, and deterministic CPU setup

# A) time\_series\_24m\_forecast.py

# Forecasts 2024-01 → 2025-12 for each sector with ARIMA, LSTM, Transformer.

# Input: Export\_Master\_Dataset\_Clean.xlsx

# Output: Time\_series\_forecasts\_2024\_2025.xlsx (Forecasts\_Future, Models\_Used, Notes)

import os, warnings, math, random

import numpy as np

import pandas as pd

from pathlib import Path

warnings.filterwarnings("ignore")

# deterministic CPU

os.environ["PYTHONHASHSEED"] = "42"

os.environ["CUDA\_VISIBLE\_DEVICES"] = ""

import torch

import torch.nn as nn

from sklearn.preprocessing import StandardScaler

from statsmodels.tsa.arima.model import ARIMA

**Explanation**

* Imports the scientific Python stack plus **statsmodels** (ARIMA), **PyTorch** (LSTM/Transformer), and **StandardScaler** for scaling NN inputs.
* PYTHONHASHSEED=42 and CUDA\_VISIBLE\_DEVICES="" enforce CPU‑only deterministic runs, supporting reproducibility required for a thesis.

## A.9.3 Global configuration and forecast horizon

# ---------------- CONFIG ----------------

IN\_EXPORT = Path("/content/Export\_Master\_Dataset\_Clean.xlsx")

OUT\_XLSX = Path("Time\_series\_forecasts\_2024\_2025.xlsx")

SECTORS = ["Agriculture","Beverages","Dairy","Meat","Pharmaceuticals","Vegetables"]

FREQ = "MS"

# Splits for fitting & internal validation

TRAIN\_END = pd.Timestamp("2020-12-01")

VAL\_START = pd.Timestamp("2020-01-01")

VAL\_END = pd.Timestamp("2020-12-01")

# Future horizon

FUTURE\_STEPS = 24 # 2024-01..2025-12

FUTURE\_START = pd.Timestamp("2024-01-01")

# ARIMA grid (small, non-seasonal)

P\_RANGE = range(0, 5)

D\_RANGE = range(0, 3)

Q\_RANGE = range(0, 5)

# Neural nets (CPU, deterministic)

DEVICE = "cpu"

RAND\_SEED = 42

LOOKBACK = 12

EPOCHS\_LSTM = 150

EPOCHS\_TR = 150

LR = 1e-3

BATCH\_SIZE = 32

**Explanation**

* Defines input/output paths and the target **sectors**.
* Uses a **2020 validation window** (Jan–Dec 2020) for model selection.
* Sets **24‑month** forecast horizon (Jan‑2024 → Dec‑2025).
* ARIMA grid is compact (non‑seasonal).
* Neural‑net hyperparameters are modest and reproducible.

## A.9.4 Utility helpers (seeding, scaling, windowing, metrics)

# -------------- utils --------------

def set\_seed(seed=RAND\_SEED):

random.seed(seed)

np.random.seed(seed)

torch.manual\_seed(seed)

# deterministic CPU

torch.backends.cudnn.deterministic = True

torch.backends.cudnn.benchmark = False

def to\_eurom\_thousands(x):

return pd.to\_numeric(x, errors="coerce")/1000.0

def rmse(a, f):

a, f = np.array(a, float), np.array(f, float)

return float(np.sqrt(np.mean((a-f)\*\*2)))

def build\_supervised(series, lookback=12):

X, y = [], []

for i in range(lookback, len(series)):

X.append(series[i-lookback:i])

y.append(series[i])

X = np.array(X)[:, :, None] # (n, T, 1)

y = np.array(y)

return X, y

**Explanation**

* set\_seed() locks random states and deterministic backends.
* to\_eurom\_thousands() converts thousand‑euro to **euro‑millions** columns \*\_EuroM.
* build\_supervised() creates sliding windows of length **LOOKBACK=12** for neural nets.

## A.9.5 Model definitions (LSTM & Transformer)

class LSTMReg(nn.Module):

def \_\_init\_\_(self, hidden=64, layers=2, dropout=0.2):

super().\_\_init\_\_()

self.lstm = nn.LSTM(1, hidden, num\_layers=layers, batch\_first=True, dropout=dropout)

self.fc = nn.Linear(hidden, 1)

def forward(self, x):

out, \_ = self.lstm(x)

y = self.fc(out[:, -1, :])

return y.squeeze(-1)

class TinyTransformer(nn.Module):

def \_\_init\_\_(self, d\_model=64, nhead=4, num\_layers=2, ff=128, dropout=0.1):

super().\_\_init\_\_()

self.proj = nn.Linear(1, d\_model)

layer = nn.TransformerEncoderLayer(d\_model=d\_model, nhead=nhead,

dim\_feedforward=ff, dropout=dropout,

batch\_first=True)

self.enc = nn.TransformerEncoder(layer, num\_layers=num\_layers)

self.fc = nn.Linear(d\_model, 1)

def forward(self, x):

z = self.proj(x)

z = self.enc(z)

y = self.fc(z[:, -1, :])

return y.squeeze(-1)

**Explanation**

* **LSTMReg**: 2 stacked LSTM layers (hidden=64) with dropout=0.2; Dense head for 1‑step output.
* **TinyTransformer**: input projection (1→64), 2 encoder layers (4 heads), FFN=128, dropout=0.1; Dense head for 1‑step output.

## A.9.6 Training and recursive multi‑step forecasting

def train\_nn(model, Xtr, ytr, Xva, yva, epochs=150, lr=1e-3, batch=32):

model = model.to(DEVICE)

opt = torch.optim.Adam(model.parameters(), lr=lr)

loss\_fn = nn.MSELoss()

Xtr\_t = torch.tensor(Xtr, dtype=torch.float32, device=DEVICE)

ytr\_t = torch.tensor(ytr, dtype=torch.float32, device=DEVICE)

Xva\_t = torch.tensor(Xva, dtype=torch.float32, device=DEVICE)

yva\_t = torch.tensor(yva, dtype=torch.float32, device=DEVICE)

best, best\_state = math.inf, None

n = Xtr.shape[0]

for ep in range(1, epochs+1):

model.train()

idx = np.arange(n) # fixed order => deterministic

for i in range(0, n, batch):

b = idx[i:i+batch]

xb, yb = Xtr\_t[b], ytr\_t[b]

opt.zero\_grad()

pred = model(xb)

loss = loss\_fn(pred, yb)

loss.backward()

opt.step()

model.eval()

with torch.no\_grad():

vpred = model(Xva\_t)

vloss = loss\_fn(vpred, yva\_t).item()

if vloss < best:

best = vloss

best\_state = {k: v.detach().clone() for k, v in model.state\_dict().items()}

if best\_state is not None:

model.load\_state\_dict(best\_state)

return model

def recursive\_forecast(model, scaler, hist\_scaled, steps, lookback=12):

seq = hist\_scaled.copy()

out = []

for \_ in range(steps):

x = seq[-lookback:].reshape(1, lookback, 1).astype(np.float32)

with torch.no\_grad():

yhat\_s = model(torch.tensor(x, dtype=torch.float32)).cpu().numpy().ravel()[0]

out.append(yhat\_s)

seq = np.append(seq, yhat\_s)

out = np.array(out).reshape(-1, 1)

return scaler.inverse\_transform(out).ravel()

**Explanation**

* train\_nn() uses **Adam + MSE**, tracks **validation loss**, and restores the **best snapshot** (implicit early‑stopping). Fixed batch order keeps training deterministic.
* recursive\_forecast() performs **multi‑step** forecasting by appending each predicted step to the history (then inverse‑scaling back to euro‑millions).

## A.9.7 ARIMA order selection by validation RMSE

def best\_arima\_by\_val\_rmse(train\_sub, val\_sub):

best\_cfg, best\_rmse = None, math.inf

if len(val\_sub) < 3 or len(train\_sub) < 12:

cut = max(6, min(12, len(train\_sub)//4))

if cut == 0:

return (1,1,1), np.nan

val\_sub = train\_sub[-cut:]

train\_sub = train\_sub[:-cut]

for p in P\_RANGE:

for d in D\_RANGE:

for q in Q\_RANGE:

try:

fit = ARIMA(train\_sub, order=(p, d, q)).fit()

fc = fit.forecast(steps=len(val\_sub))

r = rmse(val\_sub.values, fc.values)

if r < best\_rmse:

best\_rmse = r

best\_cfg = (p, d, q)

except Exception:

continue

return (best\_cfg if best\_cfg else (1,1,1)), best\_rmse

**Explanation**

* Searches a compact non‑seasonal ARIMA grid; selects by **lowest RMSE** on the 2020 validation (or **fallback pseudo‑validation** from the training tail if needed).

## A.9.8 Main routine: read data, fit models, forecast 24 months, and save

# --------------- MAIN ---------------

def main():

set\_seed()

# Read & prepare

df = pd.read\_excel(IN\_EXPORT)

df["Date"] = pd.to\_datetime(df["Date"], errors="coerce").dt.to\_period("M").dt.to\_timestamp()

df = df.sort\_values("Date")

# ensure \*\_EuroM exists

for s in SECTORS:

em = f"{s}\_EuroM"

if em not in df.columns and s in df.columns:

df[em] = to\_eurom\_thousands(df[s])

forecasts\_rows = []

models\_used = []

for sector in SECTORS:

ycol = f"{sector}\_EuroM"

if ycol not in df.columns:

print(f"[WARN] Missing sector {sector}, skipping.")

continue

ts = df[["Date", ycol]].dropna().set\_index("Date").asfreq(FREQ)[ycol]

# train/val

train\_full = ts.loc[:TRAIN\_END]

val = ts.loc[VAL\_START:VAL\_END]

train\_sub = ts.loc[:(VAL\_START - pd.offsets.MonthBegin(1))]

if len(train\_sub) < 24:

print(f"[WARN] Too short for {sector}, skipping.")

continue

# ================= ARIMA =================

(p, d, q), \_ = best\_arima\_by\_val\_rmse(train\_sub, val)

fit\_full = ARIMA(ts, order=(p, d, q)).fit()

fc\_arima = fit\_full.forecast(steps=FUTURE\_STEPS)

# ================= LSTM =================

scaler = StandardScaler()

arr = ts.values.reshape(-1, 1)

train\_scaled = scaler.fit\_transform(arr).ravel()

# build windows using last 12 of series for start

X\_tr, y\_tr = build\_supervised(train\_scaled, lookback=LOOKBACK)

# pseudo val: last 15% windows

v = max(1, int(0.15 \* len(X\_tr)))

Xfit, yfit = X\_tr[:-v], y\_tr[:-v]

Xval, yval = X\_tr[-v:], y\_tr[-v:]

lstm = LSTMReg(hidden=64, layers=2, dropout=0.2)

lstm = train\_nn(lstm, Xfit, yfit, Xval, yval, epochs=EPOCHS\_LSTM, lr=LR, batch=BATCH\_SIZE)

hist\_scaled = train\_scaled

fc\_lstm = recursive\_forecast(lstm, scaler, hist\_scaled, FUTURE\_STEPS, lookback=LOOKBACK)

# ================= Transformer =================

transformer = TinyTransformer(d\_model=64, nhead=4, num\_layers=2, ff=128, dropout=0.1)

transformer = train\_nn(transformer, Xfit, yfit, Xval, yval, epochs=EPOCHS\_TR, lr=LR, batch=BATCH\_SIZE)

fc\_tr = recursive\_forecast(transformer, scaler, hist\_scaled, FUTURE\_STEPS, lookback=LOOKBACK)

# dates for future horizon

last = ts.index[-1].to\_period("M")

fut\_idx = pd.period\_range(last+1, periods=FUTURE\_STEPS, freq="M").to\_timestamp()

# collect rows

for dt, a, l, t in zip(fut\_idx, fc\_arima.values, fc\_lstm, fc\_tr):

forecasts\_rows.append({

"Date": dt, "Sector": sector,

"Pred\_ARIMA": a, "Pred\_LSTM": l, "Pred\_Transformer": t

})

models\_used.append({"Sector": sector, "ARIMA\_order": f"({p},{d},{q})",

"LSTM": "hidden=64, layers=2, dropout=0.2, lookback=12",

"Transformer": "d\_model=64, nhead=4, layers=2, ff=128, dropout=0.1"})

forecasts\_df = pd.DataFrame(forecasts\_rows).sort\_values(["Sector","Date"])

models\_df = pd.DataFrame(models\_used)

with pd.ExcelWriter(OUT\_XLSX, engine="openpyxl") as w:

forecasts\_df.to\_excel(w, sheet\_name="Forecasts\_Future", index=False)

models\_df.to\_excel(w, sheet\_name="Models\_Used", index=False)

pd.DataFrame({"Note":[

"Forecast horizon: 24 months (2024-01..2025-12)",

"CPU deterministic run; seeds fixed.",

"ARIMA tuned on 2020 val; LSTM/Transformer trained on full series with early-stopping via validation slice."

]}).to\_excel(w, sheet\_name="Notes", index=False)

print(f"[OK] Saved → {OUT\_XLSX}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Explanation**

* Reads, cleans, and standardizes the series (assuring \*\_EuroM columns exist).
* For each sector:
  1. **Tune ARIMA** on validation → **refit on full** → forecast 24 steps.
  2. **Scale series** → build windows → **train LSTM/Transformer** with validation slice → **recursive forecasts** 24 steps.
  3. Construct a future monthly index and **store predictions** (ARIMA/LSTM/Transformer).
  4. Log the **model configurations** for reproducibility.
* Writes three sheets to the Excel output: **Forecasts\_Future**, **Models\_Used**, **Notes** – exactly what your thesis cites in Section 3.9.

## A.9.9 Reproducibility & Design Notes

* **Determinism**: CPU‑only, fixed seeds, deterministic backends, and fixed minibatch order.
* **Fairness**: LSTM and Transformer share the same validation/early‑stopping logic and training budgets.
* **Traceability**: all hyperparameters and ARIMA orders are recorded per sector in **Models\_Used**.

# Appendix 3.10.A — Hybrid 24-Month Forecasts & Scenarios

## A.10.1 Purpose & Overview

(No code here — headline intent.)

This script produces **Jan-2024 → Dec-2025** hybrid forecasts by (i) **projecting PPML** (the Gravity/structural component), (ii) **forecasting residuals** (short-run dynamics), and (iii) **adding Soft (+2%) / Hard (+7%) scenario shocks** via the PPML channel. Results are written to **Hybrid\_Forecasts\_Future.xlsx** in the sheets: **Baseline\_Future**, **Soft\_Brexit\_Future**, **Hard\_Brexit\_Future**, **Models\_Used**, and **Notes**.

## A.10.2 Full Script (with step-by-step code snippets)

### 1) Imports, warnings, deterministic setup

import os, warnings, math, random

import numpy as np

import pandas as pd

from pathlib import Path

warnings.filterwarnings("ignore")

# deterministic CPU

os.environ["PYTHONHASHSEED"] = "42"

os.environ["CUDA\_VISIBLE\_DEVICES"] = ""

import torch

import torch.nn as nn

from sklearn.preprocessing import StandardScaler

from statsmodels.tsa.arima.model import ARIMA

This block loads the scientific stack (NumPy/Pandas), modeling libraries (**statsmodels** for ARIMA; **PyTorch** for LSTM/Transformer), and **StandardScaler** for NN scaling. Two environment variables force **CPU-only, reproducible runs**: a fixed Python hash seed and the GPU disabled. We silence non-critical warnings to keep logs readable.

### 2) Global configuration & scenario parameters

IN\_PPML = Path("/content/PPML\_Hybrid\_prediction.xlsx") # sheet: PPML\_panel

OUT\_XLSX = Path("Hybrid\_Forecasts\_Future.xlsx")

SECTORS = ["Agriculture","Beverages","Dairy","Meat","Pharmaceuticals","Vegetables"]

FREQ = "MS"

FUTURE\_STEPS = 24

FUTURE\_START = pd.Timestamp("2024-01-01")

P\_RANGE = range(0, 5); D\_RANGE = range(0, 3); Q\_RANGE = range(0, 5)

DEVICE="cpu"; RAND\_SEED=42; LOOKBACK=12; EPOCHS=150; LR=1e-3; BATCH\_SIZE=32

PPML\_METHOD = "linear" # 'flat' | 'ets' | 'linear'

SOFT\_PCT = 0.02

HARD\_PCT = 0.07

Here we define the **input** (PPML panel) and **output** workbook, set the **six sectors**, and specify the **24-month** horizon (2024–2025). The ARIMA grid is compact and non-seasonal. The neural-net hyperparameters are modest and replicable. We default PPML projection to **linear**, and set scenario shocks to **+2%** (Soft) and **+7%** (Hard) on the PPML path.

### 3) Utilities: seeds, metric, windowing

def set\_seed(seed=RAND\_SEED):

random.seed(seed); np.random.seed(seed); torch.manual\_seed(seed)

torch.backends.cudnn.deterministic = True

torch.backends.cudnn.benchmark = False

def rmse(a, f):

a, f = np.array(a, float), np.array(f, float)

return float(np.sqrt(np.mean((a-f)\*\*2)))

def build\_supervised(series, lookback=12):

X, y = [], []

for i in range(lookback, len(series)):

X.append(series[i-lookback:i]); y.append(series[i])

return np.array(X)[:, :, None], np.array(y)

set\_seed() pins RNGs for **exact reproducibility**. rmse() is used to score ARIMA orders on validation. build\_supervised() transforms a 1-D series into sliding windows of **LOOKBACK=12** to feed LSTM/Transformer with the correct shape (n, T, 1).

### 4) Models: LSTMReg & TinyTransformer

class LSTMReg(nn.Module):

def \_\_init\_\_(self, hidden=64, layers=2, dropout=0.2):

super().\_\_init\_\_()

self.lstm = nn.LSTM(1, hidden, num\_layers=layers, batch\_first=True, dropout=dropout)

self.fc = nn.Linear(hidden, 1)

def forward(self, x):

out, \_ = self.lstm(x)

return self.fc(out[:, -1, :]).squeeze(-1)

class TinyTransformer(nn.Module):

def \_\_init\_\_(self, d\_model=64, nhead=4, num\_layers=2, ff=128, dropout=0.1):

super().\_\_init\_\_()

self.proj = nn.Linear(1, d\_model)

layer = nn.TransformerEncoderLayer(d\_model=d\_model, nhead=nhead,

dim\_feedforward=ff, dropout=dropout, batch\_first=True)

self.enc = nn.TransformerEncoder(layer, num\_layers=num\_layers)

self.fc = nn.Linear(d\_model, 1)

def forward(self, x):

z = self.proj(x); z = self.enc(z)

return self.fc(z[:, -1, :]).squeeze(-1)

We define **compact** NN models for residuals. The LSTM has two layers and dropout to avoid overfitting; the Transformer encodes sequences with 2 encoder layers and 4 heads. Both output a **single-step** prediction from the last time step’s representation.

### 5) NN training & recursive multi-step forecasting

def train\_nn(model, Xtr, ytr, Xva, yva, epochs=150, lr=1e-3, batch=32):

model = model.to(DEVICE)

opt = torch.optim.Adam(model.parameters(), lr=lr)

loss\_fn = nn.MSELoss()

Xtr\_t = torch.tensor(Xtr, dtype=torch.float32, device=DEVICE)

ytr\_t = torch.tensor(ytr, dtype=torch.float32, device=DEVICE)

Xva\_t = torch.tensor(Xva, dtype=torch.float32, device=DEVICE)

yva\_t = torch.tensor(yva, dtype=torch.float32, device=DEVICE)

best, best\_state = math.inf, None

n = Xtr.shape[0]

for \_ in range(epochs):

model.train()

idx = np.arange(n)

for i in range(0, n, batch):

b = idx[i:i+batch]; xb, yb = Xtr\_t[b], ytr\_t[b]

opt.zero\_grad(); loss = loss\_fn(model(xb), yb); loss.backward(); opt.step()

model.eval()

with torch.no\_grad():

vloss = loss\_fn(model(Xva\_t), yva\_t).item()

if vloss < best:

best, best\_state = vloss, {k: v.detach().clone() for k,v in model.state\_dict().items()}

if best\_state is not None:

model.load\_state\_dict(best\_state)

return model

def recursive\_forecast(model, scaler, hist\_scaled, steps, lookback=12):

seq = hist\_scaled.copy(); out = []

for \_ in range(steps):

x = seq[-lookback:].reshape(1, lookback, 1).astype(np.float32)

with torch.no\_grad():

yhat\_s = model(torch.tensor(x, dtype=torch.float32)).cpu().numpy().ravel()[0]

out.append(yhat\_s); seq = np.append(seq, yhat\_s)

return scaler.inverse\_transform(np.array(out).reshape(-1,1)).ravel()

train\_nn() runs **Adam+MSE** and snapshots the **best validation** model (implicit early-stopping). recursive\_forecast() generates **multi-step** predictions by feeding each new prediction back, then inverse-scales them to euro-million units.

### 6) ARIMA order selection by validation RMSE

def best\_arima\_by\_val\_rmse(train\_sub, val\_sub):

best\_cfg, best\_rmse = None, math.inf

if len(val\_sub) < 3 or len(train\_sub) < 12:

cut = max(6, min(12, len(train\_sub)//4))

if cut == 0: return (1,1,1), np.nan

val\_sub = train\_sub[-cut:]; train\_sub = train\_sub[:-cut]

for p in P\_RANGE:

for d in D\_RANGE:

for q in Q\_RANGE:

try:

fit = ARIMA(train\_sub, order=(p,d,q)).fit()

r = rmse(val\_sub.values, fit.forecast(steps=len(val\_sub)).values)

if r < best\_rmse: best\_rmse, best\_cfg = r, (p,d,q)

except Exception:

continue

return (best\_cfg if best\_cfg else (1,1,1)), best\_rmse

We grid-search a small ARIMA space and **pick the order with lowest RMSE** on the 2020 validation window. If that window is short, we create a small **pseudo-validation** tail from the training data. This keeps tuning stable across sectors.

### 7) PPML projection helper (flat / ETS / linear)

def project\_ppml(ppml\_series, steps, method="linear"):

idx\_last = ppml\_series.index[-1].to\_period("M")

fut\_idx = pd.period\_range(idx\_last+1, periods=steps, freq="M").to\_timestamp()

if method == "flat":

fut\_vals = np.full(steps, ppml\_series.iloc[-1], dtype=float)

elif method == "ets":

level = ppml\_series.iloc[0]; alpha = 0.3

for v in ppml\_series.iloc[1:]:

level = alpha \* v + (1 - alpha) \* level

fut\_vals = np.full(steps, level, dtype=float)

else: # linear trend from last 24 months

use = ppml\_series.iloc[-min(24, len(ppml\_series)):]

coef = np.polyfit(np.arange(len(use)), use.values, 1)

fut\_vals = coef[0] \* np.arange(len(use), len(use)+steps) + coef[1]

return pd.Series(fut\_vals, index=fut\_idx)

This function **extends PPML** into the future. We default to a **linear** trend using the last 24 months (transparent and consistent). You can switch to flat (plateau) or ets (damped) by changing one parameter.

### 8) Main: read PPML panel, validate, sort

def main():

set\_seed()

panel = pd.read\_excel(IN\_PPML, sheet\_name="PPML\_panel")

panel.columns = [c.strip() for c in panel.columns]

for c in ["Date","Sector","PPML\_Pred","PPML\_Residual","Actual\_Exports"]:

if c not in panel.columns:

raise ValueError(f"PPML\_panel missing column: {c}")

panel["Date"] = pd.to\_datetime(panel["Date"], errors="coerce")

panel = panel.sort\_values(["Sector","Date"])

We pin RNGs, read **PPML\_panel**, assert essential columns exist, parse dates, and order the data. This section is all about **hygiene and safety** before forecasting.

### 9) Per-sector loop: PPML projection → residual forecast → hybrid baseline

baseline\_rows, soft\_rows, hard\_rows, models\_rows = [], [], [], []

for sector, g in panel.groupby("Sector"):

g = g.sort\_values("Date").set\_index("Date")

ppml = g["PPML\_Pred"].astype(float)

resid = g["PPML\_Residual"].astype(float)

# 1) PPML projection

ppml\_future = project\_ppml(ppml, FUTURE\_STEPS, method=PPML\_METHOD)

# 2) Residual forecast (ARIMA default; NNs trained too)

train\_full = resid

val = resid.loc["2020-01-01":"2020-12-01"]

(p, d, q), \_ = best\_arima\_by\_val\_rmse(train\_full.loc[: "2020-12-01"], val)

m\_ar = ARIMA(train\_full, order=(p,d,q)).fit()

fc\_ar = m\_ar.forecast(steps=FUTURE\_STEPS)

scaler = StandardScaler()

arr = train\_full.values.reshape(-1,1)

train\_scaled = scaler.fit\_transform(arr).ravel()

X, y = build\_supervised(train\_scaled, lookback=LOOKBACK)

v = max(1, int(0.15\*len(X))); Xfit, yfit = X[:-v], y[:-v]; Xval, yval = X[-v:], y[-v:]

lstm = LSTMReg(hidden=64, layers=2, dropout=0.2)

lstm = train\_nn(lstm, Xfit, yfit, Xval, yval, epochs=EPOCHS, lr=LR, batch=BATCH\_SIZE)

resid\_fc\_lstm = recursive\_forecast(lstm, scaler, train\_scaled, FUTURE\_STEPS, lookback=LOOKBACK)

tr = TinyTransformer(d\_model=64, nhead=4, num\_layers=2, ff=128, dropout=0.1)

tr = train\_nn(tr, Xfit, yfit, Xval, yval, epochs=EPOCHS, lr=LR, batch=BATCH\_SIZE)

resid\_fc\_tr = recursive\_forecast(tr, scaler, train\_scaled, FUTURE\_STEPS, lookback=LOOKBACK)

# ARIMA by default (small residuals)

resid\_future = pd.Series(fc\_ar.values, index=ppml\_future.index)

# 3) Hybrid baseline

hybrid\_future = ppml\_future + resid\_future

For each sector, we project the **structural** PPML forward, then forecast **residuals**. ARIMA is used by default (residuals are typically small); LSTM/Transformer are trained for completeness and easy switching later. We then sum PPML and residuals to form the **hybrid baseline**.

### 10) Soft & Hard scenarios via PPML channel

soft\_future = hybrid\_future + SOFT\_PCT \* ppml\_future

hard\_future = hybrid\_future + HARD\_PCT \* ppml\_future

Scenarios are applied **to the PPML component** (not the residual) for **clear economic interpretation**: structural frictions change the **structural** path. Hybrid scenario paths are then **Hybrid + θ·PPML** for θ ∈ {+2%, +7%}.

### 11) Collect rows & traceability entries

for dt in ppml\_future.index:

baseline\_rows.append({"Date": dt, "Sector": sector,

"PPML\_Pred\_Future": ppml\_future.loc[dt],

"Residual\_Future": resid\_future.loc[dt],

"Hybrid\_Future": hybrid\_future.loc[dt]})

soft\_rows.append({"Date": dt, "Sector": sector,

"PPML\_Pred\_Future": ppml\_future.loc[dt],

"Residual\_Future\_Adjusted": resid\_future.loc[dt] + SOFT\_PCT\*ppml\_future.loc[dt],

"Hybrid\_Soft\_Future": soft\_future.loc[dt]})

hard\_rows.append({"Date": dt, "Sector": sector,

"PPML\_Pred\_Future": ppml\_future.loc[dt],

"Residual\_Future\_Adjusted": resid\_future.loc[dt] + HARD\_PCT\*ppml\_future.loc[dt],

"Hybrid\_Hard\_Future": hard\_future.loc[dt]})

models\_rows.append({"Sector": sector,

"PPML\_projection": PPML\_METHOD,

"Residual\_model\_used": f"ARIMA({p},{d},{q})",

"NN\_configs": "LSTM(hidden=64,layers=2,dropout=0.2); Transformer(d\_model=64,nhead=4,layers=2,ff=128,dropout=0.1)"})

Each future month becomes a row with **explicit columns** that make the math audit-friendly: you can literally see PPML, residual, and their sum. We also log **which model and settings** were used per sector, so the spreadsheet is self-documenting.

### 12) Build DataFrames & write Excel

baseline\_df = pd.DataFrame(baseline\_rows).sort\_values(["Sector","Date"])

soft\_df = pd.DataFrame(soft\_rows).sort\_values(["Sector","Date"])

hard\_df = pd.DataFrame(hard\_rows).sort\_values(["Sector","Date"])

models\_df = pd.DataFrame(models\_rows)

with pd.ExcelWriter(OUT\_XLSX, engine="openpyxl") as w:

baseline\_df.to\_excel(w, sheet\_name="Baseline\_Future", index=False)

soft\_df.to\_excel(w, sheet\_name="Soft\_Brexit\_Future", index=False)

hard\_df.to\_excel(w, sheet\_name="Hard\_Brexit\_Future", index=False)

models\_df.to\_excel(w, sheet\_name="Models\_Used", index=False)

pd.DataFrame({"Notes":[

"Residual future by ARIMA (small residuals case).",

f"PPML projection method: {PPML\_METHOD} (change to 'flat' or 'ets' if desired).",

"Scenarios: Soft +2%, Hard +7% of PPML\_Pred\_Future applied via residual channel."

]}).to\_excel(w, sheet\_name="Notes", index=False)

We combine the collected rows into tidy dataframes and write all four content sheets plus a **Notes** sheet to **Hybrid\_Forecasts\_Future.xlsx**. The Notes record the most important run settings for examiners (PPML method, residual modeling choice, scenario shocks).

### 13) Entrypoint

if \_\_name\_\_ == "\_\_main\_\_":

main()

The standard Python entrypoint makes the script executable from the command line and keeps it import-safe.

## A.10.3 Practical Notes & Safe Edits (in prose)

* **Determinism**: Seeds + CPU-only + deterministic cuDNN keeps results **bit-stable** for defense.
* **Why PPML first**: structure lives in PPML; residuals are short-run “noise”—that’s why we project PPML, then add residuals.
* **ARIMA by default**: residuals are small, so parsimony beats complexity; swap to LSTM/Transformer only if a sector’s residuals show richer structure (add a selection rule).
* **Scenario clarity**: shocking **PPML** preserves meaning (“trade frictions”). Shocking the whole hybrid would confound structure with short-run noise.
* **Hygiene**: assert required columns, coerce dates, check dtypes are floats; inverse-scale NN forecasts after recursion.
* **Traceability**: **Models\_Used** + **Notes** make the workbook self-documenting.

## A.10.4 One-line “How to change” guide

* Use **ETS** projection: PPML\_METHOD = "ets"
* Use **flat** projection: PPML\_METHOD = "flat"
* Use **LSTM** residuals: resid\_future = pd.Series(resid\_fc\_lstm, index=ppml\_future.index) (after adding a selection rule)
* Change shocks: SOFT\_PCT = 0.01; HARD\_PCT = 0.10 for 1%/10% tests