Part 4 — Machine Learning

Links

- **Live updated app:** https://ind320-dashboard-basics-newthing.streamlit.app
- **Repo:** https://github.com/TaoM29/IND320-dashboard-basics

AI Usage

I Used an ChatGPT 5 as a coding and troubleshooting partner:

```
•    Helped me with implementation of 20_Price_Areas_Map_Selector.py
with Folium + GeoJSON, click-to-store coordinates, and choropleth
coloring by mean energy values over user-selected intervals.

•    Modified Snow_Drift.py page using ERA5 (Open-Meteo) at the
clicked coordinate, defining season as Jul→Jun, plotting seasonal Qt
and a 16-sector wind rose, plus monthly-vs-yearly bonus.

•    Help with Building Sliding Correlation page (this task):
selectable meteorological + energy series, rolling window, lag
control, Plotly output, and notebook-friendly guidance.

•    Help with builing SARIMAX Forecast page with full parameter
control and dynamic forecasting, added exogenous weather variables and
strategies for future exog (bonus).

•    Refactored multiple static plots to Plotly; added helper links
between related pages.

•    tooltips, and side-bar improvements for clarity and consistency.
```

Work Log

Introduction.

This part focused on expanding the dataset window (Elhub 2021–2024 + ERA5), reshaping the Streamlit app for clearer navigation, and adding analysis pages that connect meteorology with energy. The app now has four logical sections (Exploration, Regional & Local, Modelling, Quality & Diagnostics), more descriptive page names, and cross-links for a smoother workflow. Heavy operations are cached and show progress indicators. Below is a concise record of what was implemented and refined.

Spark vs. Cassandra Driver

```
•    Followed the assignment scope:

•    Production: PRODUCTION_PER_GROUP_MBA_HOUR for 2022–2024 (all
price areas).
•    Consumption: CONSUMPTION_PER_GROUP_MBA_HOUR for 2021–2024 (all
price areas).
•    API fetching mirrors Part 2 (monthly paging), expanded across
years and areas.
•    Spark + Cassandra: despite working Spark sessions (connector
3.5.1 variants, shaded/non-shaded JARs, Java 11), the JVM stack
intermittently failed to create CqlSession / see the system keyspace
reliably in this environment.
•    Per course guidance, switched writes to the official Python
Cassandra driver (idempotent bulk upserts).
```

MongoDB writes use bulk upserts + compound unique indexes so the notebook is re-runnable without duplicates.

App structure, naming & navigation

```
•    Renamed pages to clearer, more descriptive titles.

•    Introduced four sidebar sections that match the app’s structure:
•    □ Exploration
•     Regional & Local
•    □ Modelling
•    □ Quality & Diagnostics

•    Home and About mirror these sections with concise descriptions
and links.
•    Cross-page links where it helps the flow (Price Areas Map → Snow
Drift after a map click).
•    Kept the global Price Area / Year context (set on Price Area
Selector) and tiny “Change area/year” links on analysis pages.
```

Exploration

```
•    Weather Overview – Stats & Sparklines
•    Whole-year summary table (min/mean/max) + first-month sparkline
per variable.
•    Added: monthly climatology by year for each weather variable.
•    Added: for wind direction, reused the wind-rose component from
Snow Drift so directionality is visualized.
•    Removed the old generic “interactive chart” (now covered—better—
by the Weather Explorer page).
```

```
•	Data from Open-Meteo (ERA5), cached.
•	Weather Explorer – Multi-Series & Resampling
•	Multi-select variables, interactive Plotly, and legend/hover.
•	Improved: resample (H/D/W), smooth (rolling mean), and a month
range filter for quick seasonal zoom-ins.
•	Data from Open-Meteo (ERA5), cached.
•	Energy Production / Energy Consumption
•	Overview line charts and basic summaries using MongoDB
aggregates, respects global Area/Year.
```

Regional & Local

```
•	Price Areas Map – Choropleth & Click-to-Select
•	Loads exported Elspot areas (GeoJSON), auto-detects the property
holding NO1–NO5, draws outlines, highlights the selected area, and
colors a choropleth by mean kWh over user-selected interval/group
(production or consumption).
•	Click-to-store coordinate in session state; shows a quick link to
Snow Drift for that point.
•	Snow Drift (Tabler)
•	Uses ERA5 at the clicked coordinate. Season defined as 1 July →
30 June.
•	Computes Qt per season, shows a 16-sector wind rose, and includes
the bonus: monthly Qt vs. yearly Qt overlay.

•	Added an optional fence-height calculator (Wyoming / Slat-and-
wire / Solid). To make the snow-drift results actionable, given the
estimated annual transport Q_t, the calculator converts that into the
required fence height for common designs (Wyoming, Slat-and-wire,
Solid) using their storage capacity factors. It shows how different
fence types change the needed height, linking the analysis to a
practical mitigation choice.
```

Modelling

```
•	Sliding Correlation (Weather ↔ Energy)
•	Select meteorological and energy series, lag (lead/lag
interpretation) and window length, frequency, and date span.
•	Plotly rolling correlation line with hover, used to explore
correlation shifts during normal periods vs. extreme events (see
below).
•	SARIMAX Forecast (Energy)
•	Full control of (p,d,q)(P,D,Q,s), training window, forecast
horizon, and dynamic in-sample forecasting (with a percentage dynamic
start).
•	Bonus completed: exogenous weather variables selectable; future
```

```
exog generated via “last” or “hour-of-day/day-of-week mean”
strategies.
•    Plots actuals, fitted (in-sample), forecast mean, and 95% CI.
```

Scenario 1 - Consumption → household vs temperature_2m (°C) (NO1, Jan-2024)

Setup • Consumption → household vs temperature_2m (°C) • Window length 168 h (1 week), lag 0 h.

What I saw:

```
•    The rolling correlation stays negative most of the month (≈ −0.6
to −0.3), dipping to around −0.8 mid/late January during the cold
spell.
•    On the overlay, consumption spikes when temperature drops, and
eases as temperatures rise—clear winter heating behavior.
•    Nudging the lag a few hours (±6 h) didn’t flip the sign and only
slightly changed magnitude, suggesting near-immediate response of
household demand to temperature (little delay).
```

Interpretation: In normal winter conditions, household demand is strongly inversely related to temperature—colder → more kWh. As the month warms toward the end, the correlation weakens toward zero, likely as temperature variance shrinks and weekday/behavioral patterns contribute more to demand.

Scenario 2 — Wind production vs. wind speed (NO1, Oct-2024)

Setup

```
•    Energy kind: Production → group wind
•    Weather variable: wind_speed_10m (m/s)
•    Window: 168 h (1 week)
•    Lag: +1 h (weather leads production by ~1 hour)
```

What I saw

```
•    The rolling correlation starts high and positive (~0.6–0.7) in
early October, consistent with wind speed driving wind output.
•    Around Oct 18–21, correlation drops toward 0 and briefly
negative, then rebounds to ~0.5–0.6.
•    The +1 h lag slightly improved the positive segments versus 0 h,
suggesting a short response delay from wind to generation.
```

Interpretation: Strong positive r is expected (more wind → more production). The mid-month dip likely reflects anomalous periods: calm spells or very high winds causing turbine cut-outs where speed no longer maps linearly to output Weekly window balances noise and responsiveness, shorter windows (72 h) made the drop sharper, while longer (336 h) smoothed it but hid the event.

Quality & Diagnostics

```
•	Time-Series Analysis — STL Decomposition & Spectrogram
•	STL seasonal/trend decomposition and STFT spectrogram (with clearer spacing/labels).
•	Supports both production and consumption after refactor.
•	Data Quality — SPC (Outliers) & LOF (Anomalies)
•	SPC-style control bands and Local Outlier Factor highlights: explanatory captions and consistent titles.
```

Caching & UX

```
•	Heavy data paths use @st.cache_data (ERA5 loaders, energy spans, some Mongo aggregates).
•	Spinners/progress on costly steps (data fetch, model fit) so waiting time is visible.
•	The result: pages stay responsive.
```

Bonus completion recap

```
•	Monthly snow drift overlaid with yearly Qt on Snow Drift.
•	Exogenous weather variables integrated into SARIMAX with simple future-exog strategies.

# Imports, paths, env
import os, sys
from pathlib import Path
from datetime import datetime, timezone
import os, requests, pandas as pd
from tqdm.auto import tqdm
import itertools
from pyspark.sql import functions as F

# Constants
BASE_V0 = "https://api.elhub.no/energy-data/v0"
ELHUB_API_TOKEN = os.getenv("ELHUB_API_TOKEN")
PRICE_AREAS = ["NO1","NO2","NO3","NO4","NO5"]


# Common headers for JSON:API
def headers_jsonapi():
    h = {"Accept": "application/vnd.api+json"}
    if ELHUB_API_TOKEN:
        h["Authorization"] = f"Bearer {ELHUB_API_TOKEN}"
    return h
```

```python
# ISO 8601 UTC offset formatting
def iso_utc_offset(dt: datetime) -> str:
    if dt.tzinfo is None:
        dt = dt.replace(tzinfo=timezone.utc)
    dt = dt.astimezone(timezone.utc)
    off = dt.strftime("%z")
    off = off[:-2] + ":" + off[-2:]
    return dt.strftime("%Y-%m-%dT%H:%M:%S") + off

# Groups (ids)
def list_groups(kind="production"):
    url = f"{BASE_V0}/{kind}-groups"
    r = requests.get(url, headers=headers_jsonapi(), timeout=30)
    r.raise_for_status()
    rows = []
    for item in r.json().get("data", []):
        attrs = item.get("attributes", {}) or {}
        rows.append({"id": item.get("id"), "name": attrs.get("name")})
    df = pd.DataFrame(rows)

    ids = [g for g in df["id"].tolist() if g != "*"]
    return ids, df

prod_group_ids, production_groups_df = list_groups("production")
cons_group_ids, consumption_groups_df = list_groups("consumption")

print("Production groups:", prod_group_ids)
print("Consumption groups:", cons_group_ids)
```

```
Production groups: ['solar', 'hydro', 'wind', 'thermal', 'nuclear',
'other']
Consumption groups: ['household', 'cabin', 'primary', 'secondary',
'tertiary', 'industry', 'private', 'business']
```

```python
# Generic monthly fetch
def fetch_month_generic(
    price_area: str,
    group_id: str,
    year: int,
    month: int,
    dataset: str,
    group_param_name: str,
    inner_key: str,
    group_col_out: str,
    verbose: bool = False,
) -> pd.DataFrame:
    start = datetime(year, month, 1, tzinfo=timezone.utc)
    end   = datetime(year + (month==12), (month % 12) + 1, 1,
tzinfo=timezone.utc)
```

```
    params = {
        "dataset": dataset,
        "priceArea": price_area,
        group_param_name: group_id,
        "startDate": iso_utc_offset(start),
        "endDate":   iso_utc_offset(end),
        "pageSize":  10000,
    }

    url = f"{BASE_V0}/price-areas"
    r = requests.get(url, headers=headers_jsonapi(), params=params,
timeout=90)

    if verbose:
        print("HTTP", r.status_code, "|", r.headers.get("Content-
Type"))
        print("URL:", r.url)

    if r.status_code != 200:
        if verbose: print("Body preview:", r.text[:400])
        return pd.DataFrame()

    data = r.json().get("data", [])
    if not data:
        return pd.DataFrame(columns=["priceArea", group_col_out,
"startTime", "quantityKwh"])

    rows = []
    for rec in data:
        attrs = rec.get("attributes", {}) or {}
        area  = attrs.get("name") or rec.get("id") or price_area
        inner = attrs.get(inner_key, []) or []
        for item in inner:
            rows.append({
                "priceArea": area,
                group_col_out: item.get(group_col_out),
                "startTime": item.get("startTime"),
                "quantityKwh": item.get("quantityKwh")
            })

    df = pd.DataFrame(rows)
    if df.empty:
        return df

    df["startTime"]   = pd.to_datetime(df["startTime"], utc=True,
errors="coerce")
    df["quantityKwh"] = pd.to_numeric(df["quantityKwh"],
errors="coerce")
    df =
```

```python
df.dropna(subset=["startTime","quantityKwh"]).reset_index(drop=True)
    return df


# Thin wrappers (so our code stays readable)
def fetch_month_prod(area, group_id, year, month, verbose=False):
    return fetch_month_generic(
        area, group_id, year, month,
        dataset="PRODUCTION_PER_GROUP_MBA_HOUR",
        group_param_name="productionGroup",
        inner_key="productionPerGroupMbaHour",
        group_col_out="productionGroup",
        verbose=verbose,
    )

def fetch_month_cons(area, group_id, year, month, verbose=False):
    return fetch_month_generic(
        area, group_id, year, month,
        dataset="CONSUMPTION_PER_GROUP_MBA_HOUR",
        group_param_name="consumptionGroup",
        inner_key="consumptionPerGroupMbaHour",
        group_col_out="consumptionGroup",
        verbose=verbose,
    )

# Fetch production data for 2022–2024
YEARS_PROD = [2022, 2023, 2024]

total_months = len(PRICE_AREAS) * len(prod_group_ids) *
len(YEARS_PROD) * 12
print(f"Planned requests (prod): {total_months} months "
      f"= {len(PRICE_AREAS)} areas × {len(prod_group_ids)} groups ×
{len(YEARS_PROD)} years × 12 months")

parts, runs, non_empty, row_count = [], 0, 0, 0
pbar = tqdm(total=total_months, desc="Production (2022–2024) months",
leave=True)

for area in PRICE_AREAS:
    for g in prod_group_ids:
        for y, m in itertools.product(YEARS_PROD, range(1, 13)):
            df_m = fetch_month_prod(area, g, y, m)
            runs += 1
            if not df_m.empty:
                parts.append(df_m)
                non_empty += 1
                row_count += len(df_m)
            # update progress every month
            pbar.set_postfix_str(f"rows_so_far={row_count:,}")
            pbar.update(1)
pbar.close()

prod_2224 = (pd.concat(parts, ignore_index=True)
             if parts else
pd.DataFrame(columns=["priceArea","productionGroup","startTime","quant
ityKwh"]))
prod_2224 =
prod_2224.drop_duplicates(subset=["priceArea","productionGroup","start
Time"]).reset_index(drop=True)

print("\n=== PRODUCTION 2022–2024 ===")
print(f"Requests run: {runs} | Non-empty months: {non_empty} | Rows:
{len(prod_2224):,}")
print("Span:", prod_2224["startTime"].min() if not prod_2224.empty
else None,
      "→",  prod_2224["startTime"].max() if not prod_2224.empty else
None)
display(prod_2224.head())
```

```
Planned requests (prod): 1080 months = 5 areas × 6 groups × 3 years ×
12 months
```

{"model_id":"fa60e161517444ba963b705e9829408b","version_major":2,"vers
ion_minor":0}

```
=== PRODUCTION 2022–2024 ===
Requests run: 1080 | Non-empty months: 900 | Rows: 657,600
Span: 2021-12-31 23:00:00+00:00 → 2024-12-31 22:00:00+00:00
```

```
  priceArea productionGroup                 startTime  quantityKwh
0       NO1           solar 2021-12-31 23:00:00+00:00        6.448
1       NO1           solar 2022-01-01 00:00:00+00:00        6.062
2       NO1           solar 2022-01-01 01:00:00+00:00        4.697
3       NO1           solar 2022-01-01 02:00:00+00:00       10.907
4       NO1           solar 2022-01-01 03:00:00+00:00        5.975
```

```
# Fetch consumption data for 2021–2024
YEARS_CONS = [2021, 2022, 2023, 2024]

total_months_c = len(PRICE_AREAS) * len(cons_group_ids) *
len(YEARS_CONS) * 12
print(f"\nPlanned requests (cons): {total_months_c} months "
      f"= {len(PRICE_AREAS)} areas × {len(cons_group_ids)} groups ×
{len(YEARS_CONS)} years × 12 months")

parts_c, runs_c, non_empty_c, row_count_c = [], 0, 0, 0
pbar_c = tqdm(total=total_months_c, desc="Consumption (2021–2024)
months", leave=True)
```

```python
for area in PRICE_AREAS:
    for g in cons_group_ids:
        for y, m in itertools.product(YEARS_CONS, range(1, 13)):
            df_m = fetch_month_cons(area, g, y, m)
            runs_c += 1
            if not df_m.empty:
                parts_c.append(df_m)
                non_empty_c += 1
                row_count_c += len(df_m)
            pbar_c.set_postfix_str(f"rows_so_far={row_count_c:,}")
            pbar_c.update(1)

pbar_c.close()

cons_2124 = (pd.concat(parts_c, ignore_index=True)
             if parts_c else
pd.DataFrame(columns=["priceArea","consumptionGroup","startTime","quan
tityKwh"]))
cons_2124 =
cons_2124.drop_duplicates(subset=["priceArea","consumptionGroup","star
tTime"]).reset_index(drop=True)

print("\n=== CONSUMPTION 2021–2024 ===")
print(f"Requests run: {runs_c} | Non-empty months: {non_empty_c} |
Rows: {len(cons_2124):,}")
print("Span:", cons_2124["startTime"].min() if not cons_2124.empty
else None,
      "→",  cons_2124["startTime"].max() if not cons_2124.empty else
None)
display(cons_2124.head())
```

```
Planned requests (cons): 1920 months = 5 areas × 8 groups × 4 years ×
12 months
```

{"model_id":"d5d1db6e494e4c93ae775afd0715bfa3","version_major":2,"vers
ion_minor":0}

```
=== CONSUMPTION 2021–2024 ===
Requests run: 1920 | Non-empty months: 1200 | Rows: 876,600
Span: 2020-12-31 23:00:00+00:00 → 2024-12-31 22:00:00+00:00

  priceArea consumptionGroup                 startTime  quantityKwh
0       NO1        household 2020-12-31 23:00:00+00:00    2366888.8
1       NO1        household 2021-01-01 00:00:00+00:00    2325218.2
2       NO1        household 2021-01-01 01:00:00+00:00    2273791.2
3       NO1        household 2021-01-01 02:00:00+00:00    2221311.8
4       NO1        household 2021-01-01 03:00:00+00:00    2188174.2
```

```python
# [CASSANDRA] Connect + ensure tables
from cassandra.cluster import Cluster
from cassandra.auth import PlainTextAuthProvider

CASS_HOST = "127.0.0.1"
CASS_PORT = 9042
KEYSPACE  = "ind320"

auth_provider = None

cluster = Cluster([CASS_HOST], port=CASS_PORT,
auth_provider=auth_provider)
session = cluster.connect()

# Create keyspace/tables idempotently
session.execute("""
CREATE KEYSPACE IF NOT EXISTS ind320
  WITH replication = {'class': 'SimpleStrategy', 'replication_factor':
1}
""")

session.set_keyspace(KEYSPACE)

session.execute("""
CREATE TABLE IF NOT EXISTS production_mba_hour (
  price_area text,
  production_group text,
  year int,
  start_time timestamp,
  quantity_kwh double,
  PRIMARY KEY ((price_area, production_group, year), start_time)
) WITH CLUSTERING ORDER BY (start_time ASC)
""")

session.execute("""
CREATE TABLE IF NOT EXISTS consumption_mba_hour (
  price_area text,
  consumption_group text,
  year int,
  start_time timestamp,
  quantity_kwh double,
  PRIMARY KEY ((price_area, consumption_group, year), start_time)
) WITH CLUSTERING ORDER BY (start_time ASC)
""")

# Quick check
rows = session.execute("SELECT table_name FROM system_schema.tables
WHERE keyspace_name=%s", [KEYSPACE])
print("Tables in ind320:", sorted([r.table_name for r in rows]))
```

```
Tables in ind320: ['consumption_mba_hour', 'production_mba_hour']
```

```
# [NORMALIZE] prod_2224 and cons_2124 to canonical columns
import pandas as pd
from pandas.errors import ParserError

def to_cassandra_prod(pdf: pd.DataFrame) -> pd.DataFrame:
    df = pdf.copy()
    df["price_area"]       = df["priceArea"]
    df["production_group"] = df["productionGroup"]
    df["start_time"]       = pd.to_datetime(df["startTime"], utc=True,
errors="coerce").dt.tz_localize(None)
    df["quantity_kwh"]     = pd.to_numeric(df["quantityKwh"],
errors="coerce")
    df =
df[["price_area","production_group","start_time","quantity_kwh"]].drop
na()
    df["year"]             = df["start_time"].dt.year.astype(int)
    return df

def to_cassandra_cons(pdf: pd.DataFrame) -> pd.DataFrame:
    df = pdf.copy()
    df["price_area"]        = df["priceArea"]
    df["consumption_group"] = df["consumptionGroup"]
    df["start_time"]        = pd.to_datetime(df["startTime"],
utc=True, errors="coerce").dt.tz_localize(None)
    df["quantity_kwh"]      = pd.to_numeric(df["quantityKwh"],
errors="coerce")
    df =
df[["price_area","consumption_group","start_time","quantity_kwh"]].dro
pna()
    df["year"]              = df["start_time"].dt.year.astype(int)
    return df

prod_cas = to_cassandra_prod(prod_2224)
cons_cas = to_cassandra_cons(cons_2124)

print("PROD rows:", len(prod_cas), "range:",
prod_cas["start_time"].min(), "→", prod_cas["start_time"].max())
print("CONS rows:", len(cons_cas), "range:",
cons_cas["start_time"].min(), "→", cons_cas["start_time"].max())
```

```
PROD rows: 657600 range: 2021-12-31 23:00:00 → 2024-12-31 22:00:00
CONS rows: 876600 range: 2020-12-31 23:00:00 → 2024-12-31 22:00:00
```

```
# QUIET Cassandra bulk upsert (minimal output, safe to rerun)
from cassandra.concurrent import execute_concurrent_with_args
import pandas as pd
import logging
```

```
# Silence noisy Python-side Cassandra logs for this cell
logging.getLogger("cassandra").setLevel(logging.ERROR)

# Ensure keyspace is active
try:
    session.set_keyspace("ind320")
except Exception:
    pass

# Prepared statements (idempotent to re-prepare)
prep_prod = session.prepare("""
INSERT INTO ind320.production_mba_hour
(price_area, production_group, year, start_time, quantity_kwh)
VALUES (?, ?, ?, ?, ?)
""")
prep_cons = session.prepare("""
INSERT INTO ind320.consumption_mba_hour
(price_area, consumption_group, year, start_time, quantity_kwh)
VALUES (?, ?, ?, ?, ?)
""")

def _to_native_args(df, cols):
    out = []
    for pa, grp, yr, ts, qty in df[cols].itertuples(index=False,
name=None):
        ts = pd.to_datetime(ts,
utc=True).tz_localize(None).to_pydatetime()  # naive UTC
        out.append((str(pa), str(grp), int(yr), ts, float(qty)))
    return out

prod_args = _to_native_args(
    prod_cas,
["price_area","production_group","year","start_time","quantity_kwh"]
)
cons_args = _to_native_args(
    cons_cas,
["price_area","consumption_group","year","start_time","quantity_kwh"]
)

# Execute quietly (no per-row logging, no COUNT(*))
_ = execute_concurrent_with_args(session, prep_prod, prod_args,
concurrency=256, raise_on_first_error=False)
_ = execute_concurrent_with_args(session, prep_cons, cons_args,
concurrency=256, raise_on_first_error=False)

print("Cassandra upserts completed.")
```

```
 Cassandra upserts completed.
```

```
# [MONGODB] bulk upsert to two collections
from pymongo import MongoClient, UpdateOne
import os
import streamlit as st
from pymongo import MongoClient

MONGO_URI = st.secrets["MONGO_URI"]
MDB_NAME  = st.secrets["MONGO_DB"]

COL_PROD  = "elhub_production_mba_hour"
COL_CONS  = "elhub_consumption_mba_hour"

client = MongoClient(MONGO_URI)
mdb = client[MDB_NAME]

prod_col = mdb[COL_PROD]
cons_col = mdb[COL_CONS]

# indexes (unique compound ensures idempotence)
prod_col.create_index(
    [("price_area",1), ("production_group",1), ("start_time",1)],
    unique=True, name="uniq_prod_area_group_time"
)
cons_col.create_index(
    [("price_area",1), ("consumption_group",1), ("start_time",1)],
    unique=True, name="uniq_cons_area_group_time"
)

def upsert_df(df, col, key_fields):
    ops = []
    for rec in df.to_dict(orient="records"):
        key = {k: rec[k] for k in key_fields}
        ops.append(UpdateOne(key, {"$set": rec}, upsert=True))
    if not ops:
        return (0,0)
    res = col.bulk_write(ops, ordered=False)
    return (res.upserted_count or 0, res.modified_count or 0)


# PRODUCTION: only years 2022–2024 (append to our existing 2021 in
Mongo)
prod_mongo = prod_cas[prod_cas["year"].between(2022, 2024)]
u, m = upsert_df(prod_mongo, prod_col,
["price_area","production_group","start_time"])
print(f"Mongo PRODUCTION upserts: {u}  |  updates: {m}  |  total docs
now: {prod_col.estimated_document_count()}")


# CONSUMPTION: 2021–2024 full load
u, m = upsert_df(cons_cas, cons_col,
["price_area","consumption_group","start_time"])
```

```
print(f"Mongo CONSUMPTION upserts: {u}  |  updates: {m}  |  total docs
now: {cons_col.estimated_document_count()}")
```

```
Mongo PRODUCTION upserts: 657575  |  updates: 0  |  total docs now:
657575
Mongo CONSUMPTION upserts: 876600  |  updates: 0  |  total docs now:
876600
```