assignment_4

November 28, 2025

1 IND320 - Data to Decision

Student: Sofie Lauvås

Project: Machine Learning

GitHub Repository: https://github.com/sofielauvaas/IND320_sofielauvaas_project

Streamlit App: https://ind320sofielauvaasproject.streamlit.app/

1.0.1 1. Project Setup and Library Imports

[ ]:
```
import requests
import pandas as pd
from datetime import datetime
import calendar
import tomllib
import time

# Database and Spark Imports
from cassandra.cluster import Cluster
from pymongo import MongoClient
from pyspark.sql import SparkSession
from pyspark.sql.functions import to_timestamp, col
import time
from pyspark.sql import SparkSession
from cassandra.cluster import Cluster
```

1.0.2 2. Spark and Cassandra Connection Test

2.1 Spark set-up

[ ]:
```
spark = (
    SparkSession.builder
    .appName("IND320_A4_Data_Ingestion")
    .config("spark.jars.packages", "com.datastax.spark:
  ↪spark-cassandra-connector_2.12:3.5.1")
    .config("spark.cassandra.connection.host", "127.0.0.1")
    .getOrCreate()
)
```

```python
print("Spark session started successfully.")
```

```
25/11/27 13:33:08 WARN Utils: Your hostname, Sofies-Mac-Air.local resolves to a
loopback address: 127.0.0.1; using 192.168.1.3 instead (on interface en0)
25/11/27 13:33:08 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another
address
```

```
:: loading settings :: url =
jar:file:/opt/miniconda3/envs/D2D_env/lib/python3.12/site-packages/pyspark/jars/
ivy-2.5.1.jar!/org/apache/ivy/core/settings/ivysettings.xml
```

```
Ivy Default Cache set to: /Users/sofiesveindal/.ivy2/cache
The jars for the packages stored in: /Users/sofiesveindal/.ivy2/jars
com.datastax.spark#spark-cassandra-connector_2.12 added as a dependency
:: resolving dependencies :: org.apache.spark#spark-submit-
parent-5ce67939-76e9-4550-8510-1c8f24fae861;1.0
        confs: [default]
        found com.datastax.spark#spark-cassandra-connector_2.12;3.5.1 in central
        found com.datastax.spark#spark-cassandra-connector-driver_2.12;3.5.1 in
central
        found org.scala-lang.modules#scala-collection-compat_2.12;2.11.0 in
central
        found org.apache.cassandra#java-driver-core-shaded;4.18.1 in central
        found com.datastax.oss#native-protocol;1.5.1 in central
        found com.datastax.oss#java-driver-shaded-guava;25.1-jre-graal-sub-1 in
central
        found com.typesafe#config;1.4.1 in central
        found org.slf4j#slf4j-api;1.7.26 in central
        found io.dropwizard.metrics#metrics-core;4.1.18 in central
        found org.hdrhistogram#HdrHistogram;2.1.12 in central
        found org.reactivestreams#reactive-streams;1.0.3 in central
        found org.apache.cassandra#java-driver-mapper-runtime;4.18.1 in central
        found org.apache.cassandra#java-driver-query-builder;4.18.1 in central
        found org.apache.commons#commons-lang3;3.10 in central
        found com.thoughtworks.paranamer#paranamer;2.8 in central
        found org.scala-lang#scala-reflect;2.12.19 in central
:: resolution report :: resolve 2504ms :: artifacts dl 156ms
        :: modules in use:
        com.datastax.oss#java-driver-shaded-guava;25.1-jre-graal-sub-1 from
central in [default]
        com.datastax.oss#native-protocol;1.5.1 from central in [default]
        com.datastax.spark#spark-cassandra-connector-driver_2.12;3.5.1 from
central in [default]
        com.datastax.spark#spark-cassandra-connector_2.12;3.5.1 from central in
[default]
        com.thoughtworks.paranamer#paranamer;2.8 from central in [default]
        com.typesafe#config;1.4.1 from central in [default]
        io.dropwizard.metrics#metrics-core;4.1.18 from central in [default]
        org.apache.cassandra#java-driver-core-shaded;4.18.1 from central in
```

```
[default]
        org.apache.cassandra#java-driver-mapper-runtime;4.18.1 from central in
[default]
        org.apache.cassandra#java-driver-query-builder;4.18.1 from central in
[default]
        org.apache.commons#commons-lang3;3.10 from central in [default]
        org.hdrhistogram#HdrHistogram;2.1.12 from central in [default]
        org.reactivestreams#reactive-streams;1.0.3 from central in [default]
        org.scala-lang#scala-reflect;2.12.19 from central in [default]
        org.scala-lang.modules#scala-collection-compat_2.12;2.11.0 from central
in [default]
        org.slf4j#slf4j-api;1.7.26 from central in [default]
        ---------------------------------------------------------------------
        |                  |            modules            ||   artifacts   |
        |       conf       | number| search|dwnlded|evicted|| number|dwnlded|
        ---------------------------------------------------------------------
        |      default     |   16  |   0   |   0   |   0   ||   16  |   0   |
        ---------------------------------------------------------------------
:: retrieving :: org.apache.spark#spark-submit-
parent-5ce67939-76e9-4550-8510-1c8f24fae861
        confs: [default]
        0 artifacts copied, 16 already retrieved (0kB/20ms)
25/11/27 13:33:13 WARN NativeCodeLoader: Unable to load native-hadoop library
for your platform… using builtin-java classes where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).
```

```
Spark session started successfully.
```

2.2 Cassandra driver & Keyspace set-up

[ ]:
```
keyspace_name = "ind320_project"
test_table_name = "test_spark_connection"

# Connect to Cassandra using the PyCassandra driver
cluster = Cluster(["127.0.0.1"])
session = cluster.connect()

# Ensure the keyspace exists (idempotent operation)
session.execute(f"""
CREATE KEYSPACE IF NOT EXISTS {keyspace_name}
WITH replication = {{'class':'SimpleStrategy', 'replication_factor' : 1}};
""")
session.set_keyspace(keyspace_name)
print(f"Connected to Cassandra keyspace: {keyspace_name}")
```

```
Connected to Cassandra keyspace: ind320_project
```

2.3 Spark - Cassandra pipeline verification

[ ]:
```
# Create a temporary table for verification
session.execute(f"""
CREATE TABLE IF NOT EXISTS {test_table_name} (
    id int PRIMARY KEY,
    name text
);
""")

# Insert data using the Python driver
session.execute(f"INSERT INTO {test_table_name} (id, name) VALUES (1,␣
 ↪'verification_success');")

# Read data back using the Spark connector (The critical test for the full␣
 ↪pipeline)
df_test = (
    spark.read.format("org.apache.spark.sql.cassandra")
    .options(keyspace=keyspace_name, table=test_table_name)
    .load()
)

print("Verification Data Read by Spark:")
df_test.show()

# Drop the temporary table to keep the keyspace clean for A4 tables
session.execute(f"DROP TABLE {test_table_name};")
print(f"Cleaned up temporary table: {test_table_name}")
```

```
Verification Data Read by Spark:


+---+--------------------+
| id|                name|
+---+--------------------+
|  1|verification_success|
+---+--------------------+

Cleaned up temporary table: test_spark_connection
```

1.0.3 3. Configuration and API Utility Functions

3.1 Load Secrets and Define Constants

[ ]:
```
# Define constants based on the Assignment 4 requirements
PRODUCTION_DATASET = "PRODUCTION_PER_GROUP_MBA_HOUR"
CONSUMPTION_DATASET = "CONSUMPTION_PER_GROUP_MBA_HOUR"
ENTITY = "price-areas" # The API entity used for the Elhub.no API

# Years required by the assignment
# Production requires 2022, 2023, 2024 (new data to append)
PRODUCTION_YEARS = [2022, 2023, 2024]
# Consumption requires 2021, 2022, 2023, 2024 (full dataset)
CONSUMPTION_YEARS = [2021, 2022, 2023, 2024]


# Read secrets from secrets.toml (assuming it is in the project root or␣
↪accessible)
with open("../.streamlit/secrets.toml", "rb") as f:
    secrets = tomllib.load(f)

# Store MongoDB URI
MONGO_URI = secrets["mongodb"]["uri"]
MONGO_DB = secrets["mongodb"]["database"]
MONGO_COLLECTION = secrets["mongodb"]["collection"]
MONGO_CONSUMPTION_COLLECTION = "consumption_hourly"
```

3.2 API Fetcher Function

```
[ ]: def get_monthly_ranges(year):
         """Generates monthly date ranges for a given year in the required API␣
     ↪format."""
         monthly_ranges = []
         for month in range(1, 13):
             start = datetime(year, month, 1, 0, 0, 0)
             last_day = calendar.monthrange(year, month)[1]
             end = datetime(year, month, last_day, 23, 59, 59)

             # Format datetime strings for API, including the timezone offset (%2B01)
             start_str = start.strftime("%Y-%m-%dT%H:%M:%S") + "%2B01"
             end_str = end.strftime("%Y-%m-%dT%H:%M:%S") + "%2B01"

             monthly_ranges.append((start_str, end_str))
         return monthly_ranges

     def fetch_data_from_elhub(dataset_name: str, years: list):
         """
         Fetches hourly data from the Elhub API, converts column names from␣
     ↪camelCase
         to snake_case, and returns a Pandas DataFrame.
         """
         print(f"\n--- Starting fetch for Dataset: {dataset_name} (Years: {years})␣
     ↪---")

         all_data = []

         # Base URL for the Norwegian Elhub API
```

```
    BASE_URL = f"https://api.elhub.no/energy-data/v0/{ENTITY}?
↪dataset={dataset_name}&startDate={{}}&endDate={{}}"

    # Convert the snake_case constant to camelCase to find the nested list in␣
↪the response
    list_key = "".join(word.capitalize() for word in dataset_name.lower().
↪split('_'))
    list_key = list_key[0].lower() + list_key[1:]

    print(f"Expecting data under JSON key: '{list_key}'")

    for year in years:
        print(f"Fetching data for year {year}...")
        monthly_ranges = get_monthly_ranges(year)

        for start_date, end_date in monthly_ranges:
            url = BASE_URL.format(start_date, end_date)

            response = requests.get(url)

            if response.status_code == 200:
                data = response.json()

                # Extract the list from the deep, nested JSON structure
                for entry in data.get("data", []):

                    monthly_list = entry.get("attributes", {}).get(list_key, [])

                    # Add data to the main list and standardize to have the␣
↪'dataset' for tracking
                    for record in monthly_list:
                         record['dataset'] = dataset_name
                         all_data.append(record)

            else:
                print(f"  -> Failed to retrieve data for {start_date} to␣
↪{end_date}: HTTP {response.status_code}")

        print(f"  -> Finished processing {year}.")

    if all_data:
        # Define the mapping from camelCase (API fields) to snake_case (desired␣
↪schema)
        column_mapping = {
            'priceArea': 'pricearea',
            'productionGroup': 'productiongroup',
            'consumptionGroup': 'consumptiongroup',
            'meteringPointCount': 'meteringpointcount',
            'startTime': 'starttime',
            'endTime': 'endtime',
            'quantityKwh': 'quantitykwh',
            'lastUpdatedTime': 'lastupdatedtime'
        }

        # Convert the collected list of dictionaries to a Pandas DataFrame
        final_df = pd.DataFrame(all_data)

        # Rename columns directly to snake_case for consistency (This is the␣
↪key change)
        # The 'dataset' column, added earlier, remains as is.
        final_df.rename(columns=column_mapping, inplace=True)

        print(f"\nTotal data points fetched for {dataset_name}:␣
↪{len(final_df)}")
        return final_df

    print(f"\nNo data successfully fetched for {dataset_name}.")
    return pd.DataFrame()
```

3.3 Data Retrieval

```
[ ]: # Retrieve Production Data (2022-2024)
     production_df_raw = fetch_data_from_elhub(
         dataset_name=PRODUCTION_DATASET,
         years=PRODUCTION_YEARS
     )

     # Retrieve Consumption Data (2021-2024)
     consumption_df_raw = fetch_data_from_elhub(
         dataset_name=CONSUMPTION_DATASET,
         years=CONSUMPTION_YEARS
     )

     print("\nProduction Data Head:")
     print(production_df_raw.head())
     print("\nConsumption Data Head:")
     print(consumption_df_raw.head())
```

```
--- Starting fetch for Dataset: PRODUCTION_PER_GROUP_MBA_HOUR (Years: [2022,
2023, 2024]) ---
Expecting data under JSON key: 'productionPerGroupMbaHour'
Fetching data for year 2022…

  -> Finished processing 2022.
```

```
Fetching data for year 2023…
  -> Finished processing 2023.
Fetching data for year 2024…
  -> Finished processing 2024.

Total data points fetched for PRODUCTION_PER_GROUP_MBA_HOUR: 657600

--- Starting fetch for Dataset: CONSUMPTION_PER_GROUP_MBA_HOUR (Years: [2021,
2022, 2023, 2024]) ---
Expecting data under JSON key: 'consumptionPerGroupMbaHour'
Fetching data for year 2021…
  -> Finished processing 2021.
Fetching data for year 2022…
  -> Finished processing 2022.
Fetching data for year 2023…
  -> Finished processing 2023.
Fetching data for year 2024…
  -> Finished processing 2024.

Total data points fetched for CONSUMPTION_PER_GROUP_MBA_HOUR: 876600

Production Data Head:
                     endtime            lastupdatedtime pricearea  \
0  2022-01-01T01:00:00+01:00  2025-02-01T18:02:57+01:00       NO1
1  2022-01-01T02:00:00+01:00  2025-02-01T18:02:57+01:00       NO1
2  2022-01-01T03:00:00+01:00  2025-02-01T18:02:57+01:00       NO1
3  2022-01-01T04:00:00+01:00  2025-02-01T18:02:57+01:00       NO1
4  2022-01-01T05:00:00+01:00  2025-02-01T18:02:57+01:00       NO1

  productiongroup  quantitykwh                  starttime  \
0           hydro    1291422.4  2022-01-01T00:00:00+01:00
1           hydro    1246209.4  2022-01-01T01:00:00+01:00
2           hydro    1271757.0  2022-01-01T02:00:00+01:00
3           hydro    1204251.8  2022-01-01T03:00:00+01:00
4           hydro    1202086.9  2022-01-01T04:00:00+01:00

                         dataset
0  PRODUCTION_PER_GROUP_MBA_HOUR
1  PRODUCTION_PER_GROUP_MBA_HOUR
2  PRODUCTION_PER_GROUP_MBA_HOUR
3  PRODUCTION_PER_GROUP_MBA_HOUR
4  PRODUCTION_PER_GROUP_MBA_HOUR

Consumption Data Head:
  consumptiongroup                    endtime            lastupdatedtime  \
0            cabin  2021-01-01T01:00:00+01:00  2024-12-20T10:35:40+01:00
1            cabin  2021-01-01T02:00:00+01:00  2024-12-20T10:35:40+01:00
2            cabin  2021-01-01T03:00:00+01:00  2024-12-20T10:35:40+01:00
```

```
3           cabin  2021-01-01T04:00:00+01:00  2024-12-20T10:35:40+01:00
4           cabin  2021-01-01T05:00:00+01:00  2024-12-20T10:35:40+01:00

   meteringpointcount pricearea  quantitykwh                  starttime  \
0              100607       NO1    177071.56  2021-01-01T00:00:00+01:00
1              100607       NO1    171335.12  2021-01-01T01:00:00+01:00
2              100607       NO1    164912.02  2021-01-01T02:00:00+01:00
3              100607       NO1    160265.77  2021-01-01T03:00:00+01:00
4              100607       NO1    159828.69  2021-01-01T04:00:00+01:00

                       dataset
0  CONSUMPTION_PER_GROUP_MBA_HOUR
1  CONSUMPTION_PER_GROUP_MBA_HOUR
2  CONSUMPTION_PER_GROUP_MBA_HOUR
3  CONSUMPTION_PER_GROUP_MBA_HOUR
4  CONSUMPTION_PER_GROUP_MBA_HOUR
```

3.4 Restart/Refresh Spark Session

[ ]:
```python
print("\n--- Refreshing Spark Session for Data Transformation ---")
try:
    spark.stop()
    print("Previous Spark session stopped.")
except:
    pass # Ignore if it was already stopped

spark = (
    SparkSession.builder
    .appName("IND320_A4_Data_Ingestion")
    .config("spark.jars.packages", "com.datastax.spark:
  ↪spark-cassandra-connector_2.12:3.5.1")
    .config("spark.cassandra.connection.host", "127.0.0.1")
    .getOrCreate()
)
print("New Spark session started successfully for transformation.")
```

```
--- Refreshing Spark Session for Data Transformation ---
Previous Spark session stopped.
New Spark session started successfully for transformation.
```

1.0.4 4. Spark Transformation and Cassandra Ingestion

4.1 Spark Transformation Function

[ ]:
```python
def transform_pandas_to_spark_df(pandas_df, spark_session, data_type):
    """
    Converts Pandas (which is already clean from the fetch step) to Spark,
    and casts time columns to Spark Timestamp type.
    """
    if pandas_df.empty:
        print(f"Warning: Empty DataFrame provided for transformation␣
↪({data_type}). Returning empty Spark DF.")
        return spark_session.createDataFrame([], schema=None)

    spark_df = spark_session.createDataFrame(pandas_df)

    # Convert time columns using the timezone-aware format string
    time_columns = ['starttime', 'endtime', 'lastupdatedtime']
    for col_name in time_columns:
        if col_name in spark_df.columns:
             spark_df = spark_df.withColumn(
                 col_name,
                 to_timestamp(col(col_name), "yyyy-MM-dd'T'HH:mm:ssXXX")
               )

    # Select final columns based on data type for the respective Cassandra␣
↪tables
    base_cols = ['pricearea', 'starttime', 'endtime', 'quantitykwh',␣
↪'lastupdatedtime', 'meteringpointcount']

    if data_type == 'production':
        # Production data needs productiongroup
        final_cols = ['productiongroup'] + base_cols
    else: # Consumption data
        # Consumption data needs consumptiongroup
        final_cols = ['consumptiongroup'] + base_cols

    # Filter columns to only include those present in the DataFrame
    return spark_df.select([c for c in final_cols if c in spark_df.columns])

# 1. Transform the raw dataframes (New data: 2022-2024 Production, 2021-2024␣
↪Consumption)
production_spark_df_new = transform_pandas_to_spark_df(production_df_raw,␣
↪spark, 'production')
consumption_spark_df = transform_pandas_to_spark_df(consumption_df_raw, spark,␣
↪'consumption')



# 2. CONSOLIDATION: Read the old 2021 Production data from the staging table
old_production_table = "production_2021"
try:
    production_spark_df_2021 = (
        spark.read.format("org.apache.spark.sql.cassandra")
        .options(keyspace=keyspace_name, table=old_production_table)
        .load()
    )
    # Ensure columns match for union (e.g., lowercase names)
    production_spark_df_2021 = production_spark_df_2021.
↪select(production_spark_df_new.columns)
    print(f"\nRead {production_spark_df_2021.count()} 2021 records from␣
↪'{old_production_table}'.")

    # 3. UNION: Combine the old 2021 data with the new 2022-2024 data
    production_spark_df_full = production_spark_df_2021.
↪union(production_spark_df_new)
    print(f"Total Combined Production Records (2021-2024):␣
↪{production_spark_df_full.count()}")

except Exception as e:
    # If the old table doesn't exist, assume we're only writing the new data␣
↪(this shouldn't happen)
    print(f"\nWarning: Could not read 2021 data from '{old_production_table}'.␣
↪Using only new data.")
    production_spark_df_full = production_spark_df_new

print("Spark DataFrames created for production (full 2021-2024) and consumption␣
↪(2021-2024).")
print("\nProduction Full Spark DF Schema:")
production_spark_df_full.printSchema()
print("\nConsumption Spark DF Schema:")
consumption_spark_df.printSchema()
```

```
Warning: Could not read 2021 data from 'production_2021'. Using only new data.
Spark DataFrames created for production (full 2021-2024) and consumption
(2021-2024).

Production Full Spark DF Schema:
root
 |-- productiongroup: string (nullable = true)
 |-- pricearea: string (nullable = true)
 |-- starttime: timestamp (nullable = true)
 |-- endtime: timestamp (nullable = true)
 |-- quantitykwh: double (nullable = true)
 |-- lastupdatedtime: timestamp (nullable = true)


Consumption Spark DF Schema:
root
 |-- consumptiongroup: string (nullable = true)
 |-- pricearea: string (nullable = true)
 |-- starttime: timestamp (nullable = true)
 |-- endtime: timestamp (nullable = true)
 |-- quantitykwh: double (nullable = true)
 |-- lastupdatedtime: timestamp (nullable = true)
 |-- meteringpointcount: long (nullable = true)
```

4.2 Cassandra Table Creation (New/Updated Tables)

```
[ ]: #### 4.2 Cassandra Table Creation (New/Updated Tables)
production_table_name = "production_hourly"
consumption_table_name = "consumption_hourly"

# Explicitly drop old tables to ensure the new schema is used
session.execute(f"DROP TABLE IF EXISTS {keyspace_name}.production_2021;")
session.execute(f"DROP TABLE IF EXISTS {keyspace_name}.{production_table_name};
 ↪")
session.execute(f"DROP TABLE IF EXISTS {keyspace_name}.{consumption_table_name};
 ↪")



# 3. Production Table: Re-create with the all-lowercase schema (Includes␣
 ↪meteringpointcount)
session.execute(f"""
CREATE TABLE IF NOT EXISTS {keyspace_name}.{production_table_name} (
    pricearea text,
    productiongroup text,
    starttime timestamp,
    endtime timestamp,
    quantitykwh double,
    lastupdatedtime timestamp,
    meteringpointcount int,
    PRIMARY KEY ((pricearea, productiongroup), starttime)
);
""")

# 4. Consumption Table: Re-create with the all-lowercase schema (Includes␣
 ↪meteringpointcount)
session.execute(f"""
CREATE TABLE IF NOT EXISTS {keyspace_name}.{consumption_table_name} (
    pricearea text,
    consumptiongroup text,
    starttime timestamp,
    endtime timestamp,
    quantitykwh double,
    lastupdatedtime timestamp,
    meteringpointcount int,
    PRIMARY KEY ((pricearea, consumptiongroup), starttime)
);
""")
print(f"Target Cassandra tables '{production_table_name}' and␣
  ↪'{consumption_table_name}' explicitly dropped and re-created with the␣
  ↪correct schema, including all new fields.")
```

```
Target Cassandra tables 'production_hourly' and 'consumption_hourly' explicitly
dropped and re-created with the correct schema, including all new fields.
```

4.3 Ingestion into Cassandra (Append vs. New Table)

[ ]:
```
#### 4.3 Ingestion into Cassandra (Full Dataset Write)
# Ingest Production Data (2021-2024 Full Dataset)
# We use 'overwrite' because the full, combined dataset is ready and the table␣
  ↪was just cleared.
if 'production_spark_df_full' in locals() and not production_spark_df_full.
  ↪isEmpty():
    production_spark_df_full.write\
        .format("org.apache.spark.sql.cassandra") \
        .options(keyspace=keyspace_name, table=production_table_name,␣
  ↪**{'confirm.truncate': True}) \
        .mode("overwrite") \
        .save()
    print(f"\nSuccessfully OVERWROTE {production_spark_df_full.count()} records␣
  ↪(2021-2024) into '{production_table_name}'.")
else:
    print(f"\nSkipped Production Ingestion: No 2021-2024 production data to␣
  ↪insert.")


# Ingest Consumption Data (2021-2024) - INSERTING (New table)
# The consumption_spark_df contains the full 2021-2024 dataset, so we just␣
  ↪write it.
if not consumption_spark_df.isEmpty():
    consumption_spark_df.write\
        .format("org.apache.spark.sql.cassandra") \
        .options(keyspace=keyspace_name, table=consumption_table_name) \
        .mode("append") \
        .save()
    print(f"Successfully INSERTED {consumption_spark_df.count()} records␣
  ↪(2021-2024) into new table '{consumption_table_name}'.")
else:
    print(f"Skipped Consumption Ingestion: No 2021-2024 consumption data to␣
  ↪insert.")
```

```
25/11/27 13:37:11 WARN TaskSetManager: Stage 0 contains a task of very large
size (8991 KiB). The maximum recommended task size is 1000 KiB.
```

```
25/11/27 13:37:18 WARN PythonRunner: Detected deadlock while completing task 0.0
in stage 0 (TID 0): Attempting to kill Python Worker
25/11/27 13:37:18 WARN TaskSetManager: Stage 1 contains a task of very large
size (8991 KiB). The maximum recommended task size is 1000 KiB.
25/11/27 13:37:41 WARN TaskSetManager: Stage 2 contains a task of very large
size (8991 KiB). The maximum recommended task size is 1000 KiB.
25/11/27 13:37:44 WARN TaskSetManager: Stage 5 contains a task of very large
size (12677 KiB). The maximum recommended task size is 1000 KiB.


Successfully OVERWROTE 657600 records (2021-2024) into 'production_hourly'.

25/11/27 13:37:48 WARN PythonRunner: Detected deadlock while completing task 0.0
in stage 5 (TID 18): Attempting to kill Python Worker
25/11/27 13:37:49 WARN TaskSetManager: Stage 6 contains a task of very large
size (12677 KiB). The maximum recommended task size is 1000 KiB.
25/11/27 13:38:17 WARN TaskSetManager: Stage 7 contains a task of very large
size (12677 KiB). The maximum recommended task size is 1000 KiB.
[Stage 7:>                                                          (0 + 8) / 8]

Successfully INSERTED 876600 records (2021-2024) into new table
'consumption_hourly'.
```

1.0.5 5. MongoDB Ingestion and Final Cleanup

```
[ ]: ### 5. MongoDB Ingestion and Final Cleanup

     # 5.1 Connect to MongoDB
     if not MONGO_URI:
         print("\nMongoDB ingestion skipped (MONGO_URI not loaded).")
     else:
         client = MongoClient(MONGO_URI)
         db = client[MONGO_DB]
         print(f"\nConnected to MongoDB database '{MONGO_DB}'.")

         def ingest_to_mongo(spark_df, collection_name, data_type_label):
             """Helper function to convert Spark DF to Pandas and ingest into a␣
      ↪specific MongoDB collection."""
             if spark_df.isEmpty():
                 print(f"Skipped {data_type_label} Ingestion: Spark DataFrame is␣
      ↪empty.")
                 return

             # Get the target collection instance
             collection = db[collection_name]
             print(f"\n--- Starting Ingestion for {data_type_label} into collection␣
      ↪'{collection_name}' ---")
```

```
        # Convert Spark DataFrame back to Pandas
        mongo_pandas_df = spark_df.toPandas()

        # Convert time columns to Python datetime objects for BSON compliance
        for col_name in ['starttime', 'endtime', 'lastupdatedtime']:
            if col_name in mongo_pandas_df.columns:
                mongo_pandas_df[col_name] = mongo_pandas_df[col_name].apply(
                    lambda x: x.to_pydatetime() if pd.notna(x) else None
                )

        # Convert to a list of dictionaries (BSON documents)
        records = mongo_pandas_df.to_dict("records")

        # Clear existing data and insert the new data
        collection.delete_many({})
        print(f"Cleared existing documents in MongoDB collection␣
 ↪'{collection_name}'.")

        # Insert the new data
        result = collection.insert_many(records)
        print(f"Inserted {len(result.inserted_ids)} {data_type_label} records.")
        print(f"Total documents in collection: {collection.
 ↪count_documents({})}")

    # --- INGESTION 1: PRODUCTION DATA ---
    # Writes the production data to the MONGO_COLLECTION (e.g., "production")
    ingest_to_mongo(production_spark_df_full, MONGO_COLLECTION, "Production")

    # --- INGESTION 2: CONSUMPTION DATA ---
    # Writes the consumption data to the new MONGO_CONSUMPTION_COLLECTION (e.g.
 ↪, "consumption_hourly")
    ingest_to_mongo(consumption_spark_df, MONGO_CONSUMPTION_COLLECTION,␣
 ↪"Consumption")


### 6. Final Cleanup
# Stop Spark Session
try:
    spark.stop()
    print("\nSpark session stopped. All ingestion complete.")
except Exception as e:
    print(f"\nSpark session stop failed (was likely already stopped): {e}")
```

```
Connected to MongoDB database 'elhub_db'.
```

```
25/11/27 13:38:21 WARN TaskSetManager: Stage 10 contains a task of very large
```

```
size (8991 KiB). The maximum recommended task size is 1000 KiB.
25/11/27 13:38:25 WARN PythonRunner: Detected deadlock while completing task 0.0
in stage 10 (TID 36): Attempting to kill Python Worker


--- Starting Ingestion for Production into collection 'production' ---

25/11/27 13:38:27 WARN TaskSetManager: Stage 11 contains a task of very large
size (8991 KiB). The maximum recommended task size is 1000 KiB.

Cleared existing documents in MongoDB collection 'production'.
Inserted 657600 Production records.
Total documents in collection: 657600

25/11/27 13:46:08 WARN TaskSetManager: Stage 12 contains a task of very large
size (12677 KiB). The maximum recommended task size is 1000 KiB.
25/11/27 13:46:12 WARN PythonRunner: Detected deadlock while completing task 0.0
in stage 12 (TID 45): Attempting to kill Python Worker
25/11/27 13:46:12 WARN TaskSetManager: Stage 13 contains a task of very large
size (12677 KiB). The maximum recommended task size is 1000 KiB.


--- Starting Ingestion for Consumption into collection 'consumption_hourly' ---


Cleared existing documents in MongoDB collection 'consumption_hourly'.
Inserted 876600 Consumption records.
Total documents in collection: 876600

Spark session stopped. All ingestion complete.
```

1.0.6 Log

This final project has been very demanding. I usually expect a lot from myself, but this has truly been exhausting. I began working on it last week just by experimenting a little, and ended up spending all hours on it throughout the week.

The first major challenge was reading the data into MongoDB. When I finally thought I had finished the notebook, it ended up crashing my entire page before I had even modified anything. I spent two full days trying to find the root cause—getting closer, but never quite close enough. The issue turned out to be related to collections and incorrectly referenced groups, which the page was unable to handle properly.

As I continued developing, more issues appeared, most of them tracing back to small mistakes in the data ingestion step. These took an enormous amount of time to identify and correct. Even after fixing the main problem, I kept getting new errors, which I later discovered were caused simply by caching. I faced many obstacles, learned a lot, and often felt frustrated. I really tried to understand and solve things myself rather than relying on AI for everything—and that definitely takes more time. I also learned that AI doesn't always give the right answer, so I had to work things out on my own. I faced numerous errors while combining datasets and building the pipelines.

During the analysis phase, the SWC analysis showed that colder temperatures lead to higher energy consumption, higher wind speeds increase production, and using bigger time windows produces more stable forecasts.

If I had more time, I would revisit the SARIMAX model. I'm not convinced it's correct. I spent hours trying to get it to work without success and eventually received help from peers. I wanted to personalize that part more, but the logic wasn't easily transferable. No matter how many hours I put in, I still couldn't get it right.

With more time, I would also refine the aesthetics, clean up comments, and tidy the code. I would also spend more time doing the bonus assignments, as I only had time to do the chaching with spinners, and som error handling.

This project has been very challenging. But on the positive side, I have learned a great deal

1.0.7 AI Usage

Gemini, copilot and ChatGPT served as an essential codevelopment and strategic debugging partner to overcome major technical barriers. The AI's key role was resolving complex, recurring indexing and caching failures, debugging database problems, help with devolping code, rewriting my suggestions, ensuring the final analytical structure was robust, compliant, and accelerated the overall project delivery.