assignment_2

October 22, 2025

1 IND320 - Data to Decision

Student: Sofie Lauvås
Project: Integrating API, Spark, Cassandra, and MongoDB for Interactive Data Visualization

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

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

1.1 AI Usage

For this project, I used AI (ChatGPT, Gemini and Copilot) as a strategic coding and debugging partner. The AI's essential role was in overcoming initial technical barriers, such as resolving complex connection errors between Spark and Cassandra and decoding frustrating JSON errors from the Elhub API. Beyond troubleshooting, the AI ensured code compliance by guiding the implementation of required Streamlit widgets, like the selectors, and offered suggestions for clear, professional visualizations and complete project documentation.

1.1.1 Library Imports

[ ]:
```
import requests
import pandas as pd
from datetime import datetime
import calendar
import tomllib
from cassandra.cluster import Cluster
from pymongo import MongoClient
import plotly.express as px
from pyspark.sql import SparkSession
from pyspark.sql.functions import to_timestamp, col
import plotly.io as pio
pio.renderers.default = "notebook+pdf"

# mine worked without this, but if you run into issues, you might try setting␣
 ↪these eviorment variables for PySpark
# import os
# os.environ["JAVA_HOME"] = "/Library/Java/JavaVirtualMachines/microsoft-17.jdk/
 ↪Contents/Home"
# os.environ["PYSPARK_PYTHON"] = "python"
```

```
# os.environ["PYSPARK_DRIVER_PYTHON"] = "python"
```

1.1.2 Spark and Cassandra Connection Test

[ ]:
```
# Start Spark session with Cassandra connector configuration
spark = (
    SparkSession.builder
    .appName("IND320_Elhub_Project")
    .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("Spark session started successfully.")
```

```
Spark session started successfully.
```

[ ]:
```
# Connect to a local Cassandra instance
cluster = Cluster(["127.0.0.1"])
session = cluster.connect()
keyspace_name = "ind320_project"
test_table_name = "test_table"

# Create keyspace and test table
session.execute(f"""
CREATE KEYSPACE IF NOT EXISTS {keyspace_name}
WITH replication = {{'class':'SimpleStrategy', 'replication_factor' : 1}};
""")
session.set_keyspace(keyspace_name)

session.execute(f"""
CREATE TABLE IF NOT EXISTS {test_table_name} (
    id int PRIMARY KEY,
    name text
);
""")
session.execute(f"INSERT INTO {test_table_name} (id, name) VALUES (1,␣
 ↪'test_row_ok');")

# Load data into Spark to verify the full Spark-Cassandra connection pipeline
df_test = (
    spark.read.format("org.apache.spark.sql.cassandra")
    .options(keyspace=keyspace_name, table=test_table_name)
    .load()
)
```

```
print("\nCassandra keyspace and test table created. Spark-Cassandra connection␣
  ↪verified:")
df_test.show()
```

```
WARNING Task(Task-2) cassandra.cluster:cluster.py:__init__()- Cluster.__init__
called with contact_points specified, but no load_balancing_policy. In the next
major version, this will raise an error; please specify a load-balancing policy.
(contact_points = ['127.0.0.1'], lbp = None)
WARNING Task(Task-2) cassandra.cluster:cluster.py:protocol_downgrade()-
Downgrading core protocol version from 66 to 65 for 127.0.0.1:9042. To avoid
this, it is best practice to explicitly set Cluster(protocol_version) to the
version supported by your cluster. http://datastax.github.io/python-
driver/api/cassandra/cluster.html#cassandra.cluster.Cluster.protocol_version
WARNING Task(Task-2) cassandra.cluster:cluster.py:protocol_downgrade()-
Downgrading core protocol version from 65 to 5 for 127.0.0.1:9042. To avoid
this, it is best practice to explicitly set Cluster(protocol_version) to the
version supported by your cluster. http://datastax.github.io/python-
driver/api/cassandra/cluster.html#cassandra.cluster.Cluster.protocol_version


Cassandra keyspace and test table created. Spark-Cassandra connection verified:
+---+-----------+
| id|       name|
+---+-----------+
|  1|test_row_ok|
+---+-----------+
```

1.1.3 Elhub API Data Retrieval

[ ]:
```
# API configuration
entity = "price-areas"
dataset = "PRODUCTION_PER_GROUP_MBA_HOUR"

# Prepare monthly date ranges for 2021
monthly_ranges = []
for month in range(1, 13):
    start = datetime(2021, month, 1, 0, 0, 0)
    last_day = calendar.monthrange(2021, month)[1]
    end = datetime(2021, month, last_day, 23, 59, 59)

    # Format datetime strings for API
    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))

print(f"Generated {len(monthly_ranges)} monthly API ranges for 2021.")
```

```
Generated 12 monthly API ranges for 2021.
```

[ ]:
```python
# Collect all monthly data from the API
all_data = []

for start_date, end_date in monthly_ranges:
    url = f"https://api.elhub.no/energy-data/v0/{entity}?
 ↪dataset={dataset}&startDate={start_date}&endDate={end_date}"
    response = requests.get(url)

    # Check for successful response
    # We set status code 200 as success, other codes may indicate issues
    if response.status_code == 200:
        data = response.json()
        # Extract the list in 'productionPerGroupMbaHour' from each price area␣
 ↪entry
        for entry in data.get("data", []):
            monthly_list = entry.get("attributes", {}).
 ↪get("productionPerGroupMbaHour", [])
            all_data.extend(monthly_list)
    else:
        print(f"Failed to retrieve data for {start_date} to {end_date}:␣
 ↪{response.status_code}")

print(f"\nTotal records collected for 2021: {len(all_data)}")

# Convert the list of records to a Pandas DataFrame
df_production = pd.DataFrame(all_data)
# Standardize column names to lowercase because APIs can have inconsistent␣
 ↪casing
df_production.columns = [c.lower() for c in df_production.columns]

print("\nSample of API data (Pandas DataFrame):")
df_production.head()
```

```
Total records collected for 2021: 215353

Sample of API data (Pandas DataFrame):
```

[ ]:
```
                    endtime            lastupdatedtime pricearea  \
0  2021-01-01T01:00:00+01:00  2024-12-20T10:35:40+01:00       NO1
1  2021-01-01T02:00:00+01:00  2024-12-20T10:35:40+01:00       NO1
2  2021-01-01T03:00:00+01:00  2024-12-20T10:35:40+01:00       NO1
3  2021-01-01T04:00:00+01:00  2024-12-20T10:35:40+01:00       NO1
4  2021-01-01T05:00:00+01:00  2024-12-20T10:35:40+01:00       NO1

  productiongroup  quantitykwh                  starttime
```

```
0           hydro    2507716.8  2021-01-01T00:00:00+01:00
1           hydro    2494728.0  2021-01-01T01:00:00+01:00
2           hydro    2486777.5  2021-01-01T02:00:00+01:00
3           hydro    2461176.0  2021-01-01T03:00:00+01:00
4           hydro    2466969.2  2021-01-01T04:00:00+01:00
```

1.1.4 Spark Transformation and Cassandra Insertion

[ ]:
```
# Create Target Cassandra Table
target_table_name = "production_2021"

# Drop and re-create the table to ensure a clean insert
session.execute(f"DROP TABLE IF EXISTS {keyspace_name}.{target_table_name};")
session.execute(f"""
CREATE TABLE IF NOT EXISTS {keyspace_name}.{target_table_name} (
    pricearea text,
    productiongroup text,
    starttime timestamp,
    endtime timestamp,
    quantitykwh double,
    lastupdatedtime timestamp,
    PRIMARY KEY ((pricearea, productiongroup), starttime)
);
""")
print(f"Target Cassandra table '{target_table_name}' created.")
```

```
Target Cassandra table 'production_2021' created.
```

[ ]:
```
# Convert to Spark DataFrame and Transform Data Types
# Convert Pandas DataFrame to Spark DataFrame
sdf_production = spark.createDataFrame(df_production)

# Convert starttime, endtime, and lastupdatedtime to timestamp type
sdf_production = (
    sdf_production
    .withColumn("starttime", to_timestamp(col("starttime"), "yyyy-MM-dd'T'HH:mm:
 ↪ssXXX"))
    .withColumn("endtime", to_timestamp(col("endtime"), "yyyy-MM-dd'T'HH:mm:
 ↪ssXXX"))
    .withColumn("lastupdatedtime", to_timestamp(col("lastupdatedtime"),␣
 ↪"yyyy-MM-dd'T'HH:mm:ssXXX"))
)

#
print("Spark DataFrame schema after type casting:")
sdf_production.printSchema()
```

```
Spark DataFrame schema after type casting:
```

```
root
 |-- endtime: timestamp (nullable = true)
 |-- lastupdatedtime: timestamp (nullable = true)
 |-- pricearea: string (nullable = true)
 |-- productiongroup: string (nullable = true)
 |-- quantitykwh: double (nullable = true)
 |-- starttime: timestamp (nullable = true)
```

[ ]:
```
# Write to Cassandra using Spark
sdf_production.write\
    .format("org.apache.spark.sql.cassandra") \
    .options(keyspace=keyspace_name, table=target_table_name) \
    .mode("append") \
    .save()

print(f"Successfully inserted all records into Cassandra table␣
 ↪'{target_table_name}'.")
```

```
25/10/22 15:04:04 WARN TaskSetManager: Stage 3 contains a task of very large
size (2877 KiB). The maximum recommended task size is 1000 KiB.
[Stage 3:=============================================>              (6 + 2) / 8]
```

```
Successfully inserted all records into Cassandra table 'production_2021'.
```

1.1.5 Spark Extraction

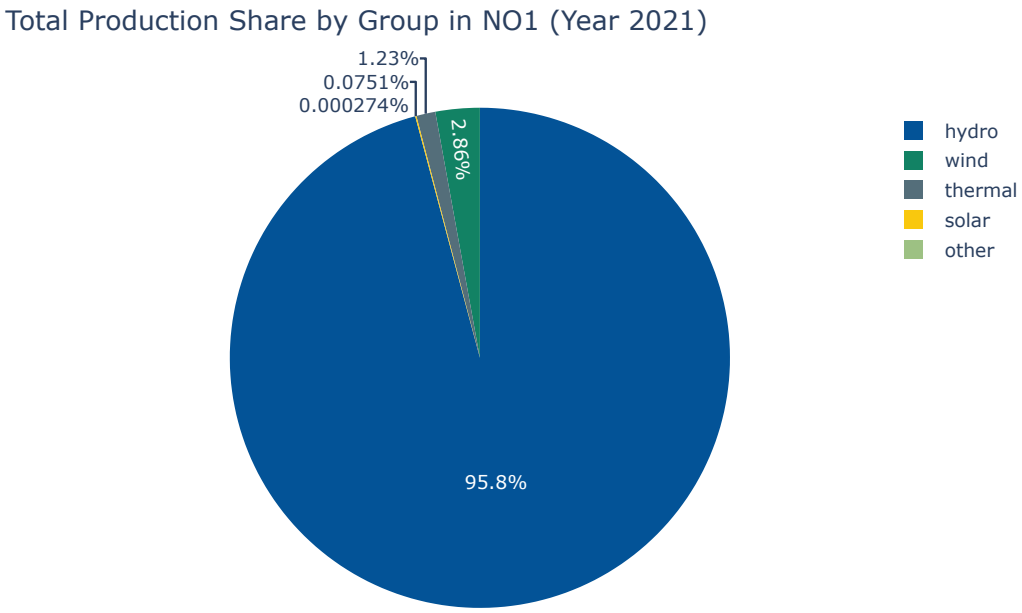
[ ]:
```
# Extract only the necessary columns from Cassandra
extracted_df = spark.read \
    .format("org.apache.spark.sql.cassandra") \
    .options(keyspace=keyspace_name, table=target_table_name) \
    .load() \
    .select("pricearea", "productiongroup", "starttime", "quantitykwh")

print("Extracted and curated Spark DataFrame head:")
extracted_df.show(5)

# Convert to Pandas DataFrame for plotting and MongoDB insertion
pdf_extracted = extracted_df.toPandas()
```

```
Extracted and curated Spark DataFrame head:
+---------+---------------+-------------------+-----------+
|pricearea|productiongroup|          starttime|quantitykwh|
+---------+---------------+-------------------+-----------+
|      NO3|        thermal|2021-01-01 00:00:00|        0.0|
|      NO3|        thermal|2021-01-01 01:00:00|        0.0|
|      NO3|        thermal|2021-01-01 02:00:00|        0.0|
```

```
|      NO3|        thermal|2021-01-01 03:00:00|        0.0|
|      NO3|        thermal|2021-01-01 04:00:00|        0.0|
+---------+---------------+-------------------+-----------+
only showing top 5 rows
```

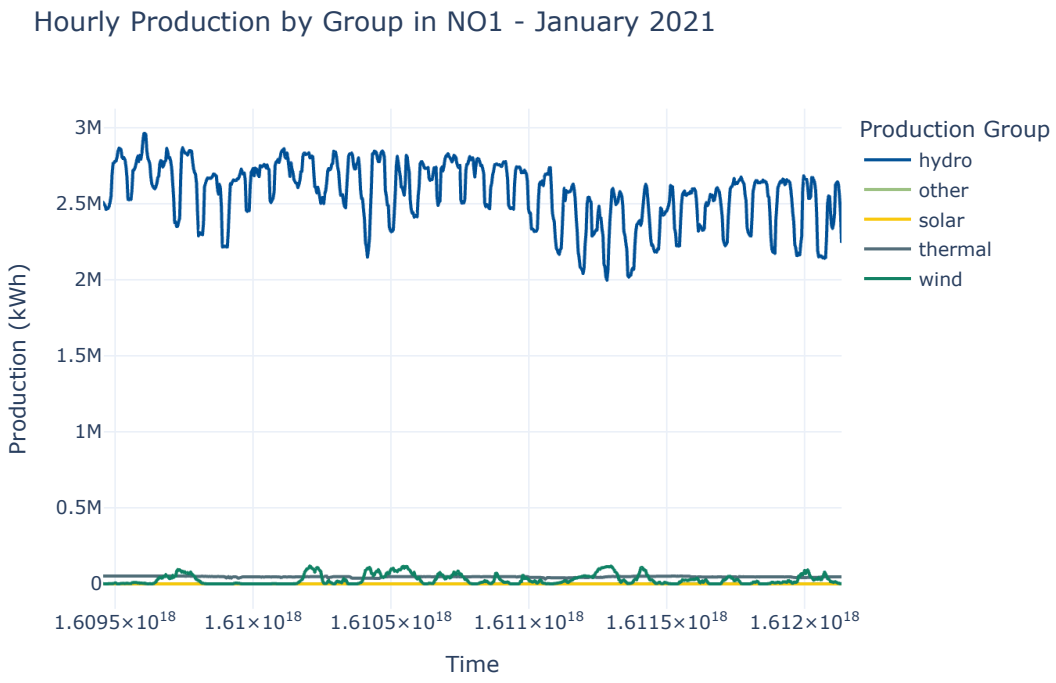
1.1.6 Plotting

[ ]:
```python
# Distinct color map for the production groups
COLOR_MAP = {
    'hydro': '#035397',
    'wind': '#128264',
    'solar': '#f9c80e',
    'thermal': '#546e7a',
    'other': '#9dc183'
}

# Choose a price area for the notebook visualization
chosen_area = "NO1"

# Filter data for the chosen area
area_data = pdf_extracted[pdf_extracted["pricearea"] == chosen_area]

# Group by production group and sum quantities for the entire year
production_by_group = area_data.groupby("productiongroup")["quantitykwh"].sum().
  ↪reset_index()

# Create Pie Chart, applying the custom color map to guarantee color.
fig_pie = px.pie(
    production_by_group,
    values='quantitykwh',
    names='productiongroup',
    color='productiongroup', # Assign the column for coloring
    color_discrete_map=COLOR_MAP, # Apply custom colors
    title=f"Total Production Share by Group in {chosen_area} (Year 2021)",
    template="plotly_white"
)

fig_pie.show()
```



[ ]:
```
# Filter data for the first month (January) of 2021
january_data = area_data[area_data["starttime"].dt.month == 1]

# Create a pivot table: time as index, production groups as columns
pivot_data = january_data.pivot_table(
    values="quantitykwh",
    index="starttime",
    columns="productiongroup",
    aggfunc="sum"
)

# Create line plot, applying the custom color map.
fig_line = px.line(
    pivot_data,
    x=pivot_data.index,
    y=pivot_data.columns,
    color_discrete_map=COLOR_MAP, # Apply custom colors here
    title=f"Hourly Production by Group in {chosen_area} - January 2021",
    labels={'value': 'Production (kWh)', 'starttime': 'Time', 'productiongroup':
 ↪ 'Group'},
    template="plotly_white"
```

```
)
fig_line.update_layout(
    xaxis_title="Time",
    yaxis_title="Production (kWh)",
    legend_title="Production Group"
)

fig_line.show()
```



1.1.7 MongoDB Insertion and Final Cleanup

[ ]:
```
# 1. Read the full URI and details from the secrets.toml file
with open("../.streamlit/secrets.toml", "rb") as f:
    cfg = tomllib.load(f)

# Access the URI, Database, and Collection from your TOML structure
uri = cfg["mongodb"]["uri"]
db_name = cfg["mongodb"]["database"]
col_name = cfg["mongodb"]["collection"]

# 2. Connect to MongoDB and select the collection
```

```
client = MongoClient(uri)
db = client[db_name]
collection = db[col_name]

print(f"Successfully connected to MongoDB and selected collection '{col_name}'!
  ↪")

# 3. Insert Data into MongoDB
records = pdf_extracted.to_dict("records")

# Clear existing data and insert the new 2021 data
collection.delete_many({})
result = collection.insert_many(records)

print(f"\nInserted {len(result.inserted_ids)} records.")
print(f"Total documents in collection: {collection.count_documents({})}")
print("\nSample document from MongoDB:")
print(collection.find_one())
```

```
Successfully connected to MongoDB and selected collection 'production'!

Inserted 215353 records.
Total documents in collection: 215353

Sample document from MongoDB:
{'_id': ObjectId('68f8d6ae1f518d8b209d458b'), 'pricearea': 'NO3',
'productiongroup': 'other', 'starttime': datetime.datetime(2021, 1, 1, 0, 0),
'quantitykwh': 0.0}
```

[ ]:
```
# Stop Spark Session
try:
    spark.stop()
    print("\nSpark session stopped.")
except:
    print("\nSpark session was already stopped.")
```

```
Spark session stopped.
```

1.2 Log

This project was a full end-to-end exercise, building a data pipeline from a raw API source to an interactive web application. This process significantly expanded on the ETL (Extract, Transform, Load) experience I gained this summer as a Data Engineer intern at Sparebank1 Utvikling.

The most challenging part of the project was the initial technical setup. Integrating Apache Spark with a local Cassandra database, managed via Docker, led to frustrating connection and initialization errors. I realized that my early difficulty was partly due to quickly asking the AI for solutions instead of taking the time to properly read and understand the underlying documentation for the

Spark-Cassandra-Docker setup. This self-correction strengthened my foundational knowledge and showed me that AI is best used after I've made a solid attempt to understand the problem myself. Separately, getting data from the Elhub API required careful work to fix JSON decoding errors and implement a systematic, month-by-month API calling strategy to collect the full year of hourly production data for all price areas without overloading the system.

Once the raw data was secure, the central Transform phase began. I used Spark DataFrames to efficiently clean the dataset, handle missing values, and standardize the time series format. The core transformation involved aggregating the hourly data to prepare it for two different visualizations: a single total for the yearly pie chart and the hourly quantities required for the monthly line plots. This required careful use of Spark functions to ensure accurate sums and groupings by both price area and production group. I then loaded the finalized, clean data into MongoDB, which now acts as the reliable and performant data source for the application, completely decoupling the real-time Streamlit app from the heavy lifting of the Spark cluster.

The final stage was the Streamlit application. I used a clear, two-column layout and implemented all the dynamic filters—st.radio for the price area, and the required st.pills widget for selecting multiple production groups. I styled the Plotly charts using a custom color map to ensure clarity and easy distinction between the different energy sources (hydro, wind, solar, etc.), and tried for it to match the config. The app is completed with an st.expander containing documentation, including the source API link (api.elhub.no). This project reinforced my practical skills in data engineering, from handling tricky APIs to creating interactive web presentations.