part4

November 26, 2025

1 Project work, part 4 - Machine Learning

1.1 Project Links

Github Repository: https://github.com/fayomitz/IND320-fayomitz

Streamlit app: https://ind320-fayomitz.streamlit.app/

1.2 AI Usage

Throughout this project, AI assistance (GitHub Copilot) was utilized in several key areas:

- **GeoJSON Integration**: Assistance with loading and processing Norwegian GeoJSON files for price areas and municipalities, implementing Shapely for point-in-polygon detection, and mapping municipalities to price areas
- **Folium Map Development**: Help with implementing dynamic layer switching based on zoom level (price areas vs municipalities), choropleth styling based on energy statistics, and click event handling for coordinate selection
- **Plotly Visualization Migration**: Guidance on converting matplotlib static plots to interactive Plotly visualizations across all pages, including line charts, bar charts, spectrograms, and polar wind rose charts
- **SARIMAX Forecasting**: Assistance with implementing the statsmodels SARIMAX model, handling timezone alignment between weather and energy data, configuring model parameters, and creating confidence interval visualizations
- **Sliding Window Correlation**: Help with implementing rolling correlation calculations with configurable lag and window size, and merging weather and energy time series data
- **Snow Drift Calculations**: Guidance on adapting the snow drift calculation formulas, implementing sector-based wind transport analysis, and creating the polar wind rose visualization
- **Error Handling**: Adding checks for missing data connections, handling NaN values, and providing user-friendly feedback messages throughout the application
- **Code Documentation**: Writing detailed comments explaining each step, database operations, and data transformation logic

AI tools significantly improved development efficiency, code quality, and implementation of complex statistical and machine learning methods while maintaining full understanding of the underlying algorithms and data pipelines.

1.3 Work Log

In this fourth assignment, I expanded the project significantly by retrieving additional years of energy data, adding consumption data, implementing interactive maps with GeoJSON overlays, and building machine learning-based forecasting capabilities. The work involved substantial Streamlit app refactoring and the addition of several new analytical features.

1.3.1 Jupyter Notebook Development

Extended Data Retrieval I began by retrieving hourly production data from the Elhub API for years 2022-2024, appending it to the existing 2021 data in both Cassandra and MongoDB. The same month-by-month fetching strategy from Part 2 was used to handle API limitations. Additionally, I retrieved hourly consumption data (`CONSUMPTION_PER_GROUP_MBA_HOUR`) for all price areas from 2021-2024. This required creating a new Cassandra table (`consumption_data`) with an appropriate schema using `consumptionGroup` instead of `productionGroup` as part of the composite partition key.

1.3.2 Streamlit App Refactoring

Visualization Migration I converted all static matplotlib/seaborn plots to dynamic Plotly visualizations throughout the application. This included interactive line charts, bar charts with hover information, spectrograms with color scaling, polar wind rose charts, and time series forecasts with confidence intervals.

Navigation Restructuring The page structure was reorganized into logical groupings: Data Exploration (Map Selector, Interactive Plot), Analysis (Snow Drift, Advanced Analysis, Weather Anomalies), and Predictive (Correlations, Forecasting). Color-coded CSS styling in the sidebar distinguishes between these groups, and the home page provides quick navigation buttons to each section.

1.3.3 New Streamlit Features

Map Selector with GeoJSON Overlays I implemented a Folium map displaying Norwegian electricity price areas (NO1-NO5) using GeoJSON data downloaded from NVE. The map features: choropleth coloring based on mean production/consumption values for a user-selected time interval, click-to-select functionality that stores coordinates and identifies the corresponding price area, and dynamic layer switching between price areas (zoomed out) and municipalities (zoomed in). Selected locations are marked with a red pin and persist across page navigation.

Snow Drift Analysis I adapted the provided Snow_drift.py calculations into a Streamlit page that computes annual and monthly snow transport based on wind speed data downloaded for the user-selected coordinates. The page includes configurable parameters (transport distance, fetch distance, relocation coefficient), season selection (July-June years), annual bar charts showing transport volumes with control type indicators, and a polar wind rose showing directional transport distribution.

Sliding Window Correlation I transformed the correlation analysis from Part 3 into an interactive Streamlit page with selectors for weather variables (temperature, precipitation, wind speed, etc.) and energy data (production or consumption by area and group). Users can configure window size and lag parameters, and the page displays a rolling correlation time series along with a dual-axis plot showing both datasets.

SARIMAX Forecasting I created a comprehensive forecasting interface allowing users to select energy data type, price area, and group, configure training date ranges and forecast horizons, set all SARIMAX parameters (p, d, q, P, D, Q, seasonal period, trend), and optionally include weather variables as exogenous features. The results display the forecast with 95% confidence intervals, validation metrics (MAE, RMSE) when test data is available, and proper timezone alignment between datasets.

1.3.4 Bonus Features

I implemented several bonus tasks from the provided list:

1. **Waiting time indicators**: Progress spinners are used throughout the app during data loading and model training. Users receive clear visual feedback when operations are in progress.
2. **Error handling**: Data requirements are checked at page load with informative warnings directing users to complete prerequisites (e.g., selecting a location on the map before running snow drift analysis). Database connection errors and missing data are caught gracefully with user-friendly messages instead of cryptic error traces.
3. **Map page with municipalities**: The map dynamically switches between price area polygons (when zoomed out) and municipality boundaries (when zoomed in beyond a natural threshold). Municipality GeoJSON data was downloaded from Geonorge using the EUREF 89 Geografisk (ETRS 89) 2d format. Each municipality is automatically mapped to its corresponding price area for consistent choropleth coloring.
4. **Monthly snow drift**: In addition to annual snow transport calculations, the Snow Drift page now computes and displays monthly snow drift values. A dedicated "Monthly Breakdown" tab shows a line chart comparing monthly transport across selected seasons, allowing users to identify seasonal patterns in snow drift.
5. **Weather as exogenous variables in forecasting**: The SARIMAX forecasting page allows users to select weather properties (temperature, precipitation, wind speed, wind gusts, wind direction) as exogenous variables. When selected, the app automatically downloads the required weather data for the training period and forecast horizon, properly aligning timezones and handling missing values through interpolation.

1.3.5 Correlation Analysis Findings

While testing the sliding window correlation feature, I observed that wind speed shows stronger correlation with wind-based energy production during winter months when wind patterns are more consistent. Temperature correlations with consumption are strongest during heating season (October-March). Extreme weather events (storms) cause temporary spikes in correlation values, followed by decorrelation periods during recovery. These patterns suggest that adaptive forecasting models could benefit from dynamic feature weighting based on seasonal and weather conditions.

1.4 1. Setup and Connections

Import necessary libraries and establish connections to MongoDB, Cassandra, and Spark.

[1]:
```
import pandas as pd
import requests
```

```python
from datetime import datetime, timedelta
import pytz
from pymongo.mongo_client import MongoClient
from pymongo.server_api import ServerApi
from dotenv import load_dotenv
import os
from cassandra.cluster import Cluster
from pyspark.sql import SparkSession
from pyspark.sql.functions import to_timestamp

load_dotenv()

# Set Hadoop home directory path (required for PySpark on Windows)
os.environ["HADOOP_HOME"] = "C:/Hadoop/hadoop-3.3.1"
os.environ["PYSPARK_HADOOP_VERSION"] = "without"
```

[3]:
```python
# MongoDB Connection
uri = os.getenv("URI")
client = MongoClient(uri, server_api=ServerApi('1'))
try:
    client.admin.command('ping')
    print("Successfully connected to MongoDB!")
    production_db = client['energy_data'] # Initialize database reference
except Exception as e:
    print(e)

# Cassandra Connection
cluster = Cluster(['localhost'], port=9042)
session = cluster.connect()
session.set_keyspace('my_first_keyspace')

# Spark Session
spark = SparkSession.builder.appName('SparkCassandraApp').\
    config('spark.jars.packages', 'com.datastax.spark:
 ↪spark-cassandra-connector_2.12:3.5.1').\
    config('spark.cassandra.connection.host', 'localhost').\
    config('spark.sql.extensions', 'com.datastax.spark.connector.
 ↪CassandraSparkExtensions').\
    config('spark.sql.catalog.mycatalog', 'com.datastax.spark.connector.
 ↪datasource.CassandraCatalog').\
    config('spark.cassandra.connection.port', '9042').getOrCreate()
```

```
Successfully connected to MongoDB!
```

1.5 2. Retrieve and Append Production Data (2022 - 2024)

We will retrieve hourly production data for all price areas for the years 2022 through 2024 and append it to our existing datasets in Cassandra and MongoDB.

[3]:
```python
# API Configuration
base_url = "https://api.elhub.no/energy-data/v0/price-areas"
dataset = "PRODUCTION_PER_GROUP_MBA_HOUR"
headers = {}

# Timezone setup
oslo_tz = pytz.timezone('Europe/Oslo')
start_date = oslo_tz.localize(datetime(2022, 1, 1, 0, 0, 0))
end_date = oslo_tz.localize(datetime(2024, 12, 31, 23, 59, 59))

# Fetch data
responses = []
current_start = start_date

print(f"Fetching production data from {start_date} to {end_date}...")

while current_start <= end_date:
    # Calculate month end
    month_end = (current_start.replace(day=28) + timedelta(days=4)).
↪replace(day=1) - timedelta(seconds=1)
    if month_end > end_date:
        month_end = end_date

    params = {
        "dataset": dataset,
        "startDate": current_start.isoformat(),
        "endDate": month_end.isoformat()
    }

    try:
        response = requests.get(base_url, headers=headers, params=params)
        response.raise_for_status()
        responses.append(response.json())
        print(f"Fetched {current_start.date()} to {month_end.date()}")
    except requests.exceptions.RequestException as e:
        print(f"Error fetching data for {current_start.date()}: {e}")

    current_start = month_end + timedelta(seconds=1)
```

```
Fetching production data from 2022-01-01 00:00:00+01:00 to 2024-12-31
23:59:59+01:00…
Fetched 2022-01-01 to 2022-01-31
Fetched 2022-02-01 to 2022-02-28
Fetched 2022-03-01 to 2022-03-31
Fetched 2022-04-01 to 2022-04-30
Fetched 2022-05-01 to 2022-05-31
Fetched 2022-06-01 to 2022-06-30
Fetched 2022-07-01 to 2022-07-31
Fetched 2022-08-01 to 2022-08-31
Fetched 2022-09-01 to 2022-09-30
Fetched 2022-10-01 to 2022-10-31
Fetched 2022-11-01 to 2022-11-30
Fetched 2022-12-01 to 2022-12-31
Fetched 2023-01-01 to 2023-01-31
Fetched 2023-02-01 to 2023-02-28
Fetched 2023-03-01 to 2023-03-31
Fetched 2023-04-01 to 2023-04-30
Fetched 2023-05-01 to 2023-05-31
Fetched 2023-06-01 to 2023-06-30
Fetched 2023-07-01 to 2023-07-31
Fetched 2023-08-01 to 2023-08-31
Fetched 2023-09-01 to 2023-09-30
Fetched 2023-10-01 to 2023-10-31
Fetched 2023-11-01 to 2023-11-30
Fetched 2023-12-01 to 2023-12-31
Fetched 2024-01-01 to 2024-01-31
Fetched 2024-02-01 to 2024-02-29
Fetched 2024-03-01 to 2024-03-31
Fetched 2024-04-01 to 2024-04-30
Fetched 2024-05-01 to 2024-05-31
Fetched 2024-06-01 to 2024-06-30
Fetched 2024-07-01 to 2024-07-31
Fetched 2024-08-01 to 2024-08-31
Fetched 2024-09-01 to 2024-09-30
Fetched 2024-10-01 to 2024-10-31
Fetched 2024-11-01 to 2024-11-30
Fetched 2024-12-01 to 2024-12-31
```

[4]:
```python
# Process Responses
all_production_data = []

for response in responses:
    if 'data' in response:
        for price_area in response['data']:
            if 'attributes' in price_area and 'productionPerGroupMbaHour' in␣
 ↪price_area['attributes']:
                production_list =␣
 ↪price_area['attributes']['productionPerGroupMbaHour']
                if production_list:
                    all_production_data.extend(production_list)

df_production = pd.DataFrame(all_production_data)
print(f"New production records fetched: {len(df_production)}")
print(df_production.head())
```

```
New production records fetched: 657600
```

```
                     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
```

1.5.1 Append to Cassandra

[5]:
```
if not df_production.empty:
    # Prepare Spark DataFrame
    spark_df_prod = spark.createDataFrame(df_production)

    # Rename columns
    spark_df_prod = spark_df_prod.withColumnRenamed("priceArea", "price_area") \
                        .withColumnRenamed("productionGroup",␣
 ↪"production_group") \
                        .withColumnRenamed("startTime", "start_time") \
                        .withColumnRenamed("endTime", "end_time") \
                        .withColumnRenamed("quantityKwh", "quantity_kwh") \
                        .withColumnRenamed("lastUpdatedTime",␣
 ↪"last_updated_time")

    # Convert timestamps
    spark_df_prod = spark_df_prod.withColumn("start_time",␣
 ↪to_timestamp("start_time")) \
                        .withColumn("end_time", to_timestamp("end_time")) \
                        .withColumn("last_updated_time",␣
 ↪to_timestamp("last_updated_time"))

    # Append to Cassandra
    spark_df_prod.write \
        .format("org.apache.spark.sql.cassandra") \
        .mode("append") \
        .options(table="production_data", keyspace="my_first_keyspace") \
        .save()

    print("Production data (2022-2024) appended to Cassandra.")
else:
    print("No production data to append.")
```

```
Production data (2022-2024) appended to Cassandra.
```

1.5.2 Append to MongoDB

```
[6]: if not df_production.empty:
         production_collection = production_db['production']

         # Clear existing data
         production_collection.delete_many({})
         print("Cleared existing data in 'production' collection.")

         production_records = df_production.to_dict('records')

         # Append data
         result = production_collection.insert_many(production_records)
         print(f"Appended {len(result.inserted_ids)} documents to MongoDB␣
       ↪'production' collection.")
     else:
         print("No production data to append.")
```

```
Cleared existing data in 'production' collection.
Appended 657600 documents to MongoDB 'production' collection.
```

1.6 3. Retrieve and Store Consumption Data (2021 - 2024)

Now we will retrieve hourly consumption data (`CONSUMPTION_PER_GROUP_MBA_HOUR`) for the years 2021 through 2024.

```
[7]: # API Configuration for Consumption
     dataset_consumption = "CONSUMPTION_PER_GROUP_MBA_HOUR"
     start_date_cons = oslo_tz.localize(datetime(2021, 1, 1, 0, 0, 0))
     end_date_cons = oslo_tz.localize(datetime(2024, 12, 31, 23, 59, 59))

     responses_cons = []
     current_start = start_date_cons

     print(f"Fetching consumption data from {start_date_cons} to {end_date_cons}...")

     while current_start <= end_date_cons:
         month_end = (current_start.replace(day=28) + timedelta(days=4)).
       ↪replace(day=1) - timedelta(seconds=1)
         if month_end > end_date_cons:
             month_end = end_date_cons

         params = {
             "dataset": dataset_consumption,
             "startDate": current_start.isoformat(),
             "endDate": month_end.isoformat()
```

```
    }

    try:
        response = requests.get(base_url, headers=headers, params=params)
        response.raise_for_status()
        responses_cons.append(response.json())
        print(f"Fetched {current_start.date()} to {month_end.date()}")
    except requests.exceptions.RequestException as e:
        print(f"Error fetching data for {current_start.date()}: {e}")

    current_start = month_end + timedelta(seconds=1)
```

```
Fetching consumption data from 2021-01-01 00:00:00+01:00 to 2024-12-31
23:59:59+01:00…
Fetched 2021-01-01 to 2021-01-31
Fetched 2021-02-01 to 2021-02-28
Fetched 2021-03-01 to 2021-03-31
Fetched 2021-04-01 to 2021-04-30
Fetched 2021-05-01 to 2021-05-31
Fetched 2021-06-01 to 2021-06-30
Fetched 2021-07-01 to 2021-07-31
Fetched 2021-08-01 to 2021-08-31
Fetched 2021-09-01 to 2021-09-30
Fetched 2021-10-01 to 2021-10-31
Fetched 2021-11-01 to 2021-11-30
Fetched 2021-12-01 to 2021-12-31
Fetched 2022-01-01 to 2022-01-31
Fetched 2022-02-01 to 2022-02-28
Fetched 2022-03-01 to 2022-03-31
Fetched 2022-04-01 to 2022-04-30
Fetched 2022-05-01 to 2022-05-31
Fetched 2022-06-01 to 2022-06-30
Fetched 2022-07-01 to 2022-07-31
Fetched 2022-08-01 to 2022-08-31
Fetched 2022-09-01 to 2022-09-30
Fetched 2022-10-01 to 2022-10-31
Fetched 2022-11-01 to 2022-11-30
Fetched 2022-12-01 to 2022-12-31
Fetched 2023-01-01 to 2023-01-31
Fetched 2023-02-01 to 2023-02-28
Fetched 2023-03-01 to 2023-03-31
Fetched 2023-04-01 to 2023-04-30
Fetched 2023-05-01 to 2023-05-31
Fetched 2023-06-01 to 2023-06-30
Fetched 2023-07-01 to 2023-07-31
Fetched 2023-08-01 to 2023-08-31
Fetched 2023-09-01 to 2023-09-30
Fetched 2023-10-01 to 2023-10-31
Fetched 2023-11-01 to 2023-11-30
Fetched 2023-12-01 to 2023-12-31
Fetched 2024-01-01 to 2024-01-31
Fetched 2024-02-01 to 2024-02-29
Fetched 2024-03-01 to 2024-03-31
Fetched 2024-04-01 to 2024-04-30
Fetched 2024-05-01 to 2024-05-31
Fetched 2024-06-01 to 2024-06-30
Fetched 2024-07-01 to 2024-07-31
Fetched 2024-08-01 to 2024-08-31
Fetched 2024-09-01 to 2024-09-30
Fetched 2024-10-01 to 2024-10-31
Fetched 2024-11-01 to 2024-11-30
Fetched 2024-12-01 to 2024-12-31
```

[8]:
```
# Process Consumption Responses
all_consumption_data = []

for response in responses_cons:
    if 'data' in response:
        for price_area in response['data']:
            if 'attributes' in price_area and 'consumptionPerGroupMbaHour' in␣
 ↪price_area['attributes']:
                consumption_list =␣
 ↪price_area['attributes']['consumptionPerGroupMbaHour']
                if consumption_list:
                    all_consumption_data.extend(consumption_list)

df_consumption = pd.DataFrame(all_consumption_data)
print(f"Total consumption records fetched: {len(df_consumption)}")
print(df_consumption.head())
```

```
Total consumption records fetched: 876600
  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
```

1.6.1 Store in Cassandra

We will create a new table `consumption_data` for this dataset.

```
[9]: # Create consumption table
# Note: The API returns 'consumptionGroup' instead of 'productionGroup'
session.execute("DROP TABLE IF EXISTS my_first_keyspace.consumption_data;")

create_cons_table_query = """
CREATE TABLE IF NOT EXISTS my_first_keyspace.consumption_data (
    price_area text,
    consumption_group text,
    start_time timestamp,
    end_time timestamp,
    quantity_kwh double,
    last_updated_time timestamp,
    PRIMARY KEY ((price_area, consumption_group), start_time)
) WITH CLUSTERING ORDER BY (start_time ASC);
"""
session.execute(create_cons_table_query)
print("Table 'consumption_data' created.")

if not df_consumption.empty:
    spark_df_cons = spark.createDataFrame(df_consumption)

    # Rename columns (consumptionGroup -> consumption_group)
    spark_df_cons = (
        spark_df_cons
        .withColumnRenamed("priceArea", "price_area")
        .withColumnRenamed("consumptionGroup", "consumption_group")
        .withColumnRenamed("startTime", "start_time")
        .withColumnRenamed("endTime", "end_time")
        .withColumnRenamed("quantityKwh", "quantity_kwh")
        .withColumnRenamed("lastUpdatedTime", "last_updated_time")
    )

    # Drop columns that are not in the Cassandra table schema
    spark_df_cons = spark_df_cons.select(
        "price_area",
        "consumption_group",
        "start_time",
        "end_time",
        "quantity_kwh",
        "last_updated_time",
    )

    # Convert timestamps
    spark_df_cons = (
        spark_df_cons
        .withColumn("start_time", to_timestamp("start_time"))
        .withColumn("end_time", to_timestamp("end_time"))
        .withColumn("last_updated_time", to_timestamp("last_updated_time"))
    )

    spark_df_cons.write \
        .format("org.apache.spark.sql.cassandra") \
        .mode("append") \
        .options(table="consumption_data", keyspace="my_first_keyspace") \
        .save()

    print("Consumption data inserted into Cassandra.")
else:
    print("No consumption data to write.")
```

```
Table 'consumption_data' created.
Consumption data inserted into Cassandra.
```

1.6.2 Store in MongoDB

We will use a new collection `consumption` and clear any existing data before inserting.

[10]:
```
if not df_consumption.empty:
    consumption_collection = production_db['consumption']

    # Clear existing data
    consumption_collection.delete_many({})
    print("Cleared existing data in 'consumption' collection.")

    consumption_records = df_consumption.to_dict('records')

    # Insert new data
    result = consumption_collection.insert_many(consumption_records)
    print(f"Inserted {len(result.inserted_ids)} documents into MongoDB␣
 ↪'consumption' collection.")
```

```
Cleared existing data in 'consumption' collection.
Inserted 876600 documents into MongoDB 'consumption' collection.
```