IN PARTNERSHIP WITH PLYMOUTH UNIVERSITY

Name: Rajapaksha Rajapaksha	
Student Reference Number:10898883	

Module Code: PUSL 3121	Module Name: Big Data Analytics
Coursework Title: PUSL3121 Big Data Analytics Coursework Report	
Deadline Date:	Member of staff responsible for coursework: Mr. Gayan Perera
Programme: BSc (Hons) Computer Science	

Group work: please list all names of all participants formally associated with this work and state whether the work was undertaken alone or as part of a team. Please note you may be required to identify individual responsibility for component parts.

Rajapaksha Rajapaksha	**10898883**
Kariyawasam Nanayakkara	**10898854**
Delwakkada Nemsara	**10898858**
Sachitha Gamage	**10898765**

We confirm that we have read and understood the Plymouth University regulations relating to Assessment Offences and that we are aware of the possible penalties for any breach of these regulations. We confirm that this is the independent work of the group.

Signed on behalf of the group: Thejan

Use of translation software: failure to declare that translation software or a similar writing aid has been used will be treated as an assessment offence.

I *have used/not used translation software.

If used, please state name of software……………………………………………………………………

Overall mark _____% **Assessors Initials ______** **Date_________**

Table of Contents

Table of Figures

Chapter 1 Introduction to Big Data Processing

1.1 Overview of Apache Spark and Snowflake

Apache Spark

Apache Spark is an open-source distributed computing framework designed to perform big data processing. It is well known for its in-memory computation capabilities, which enable it to perform significantly faster than traditional batch-processing frameworks such as Hadoop MapReduce. Spark is capable of handling a range of workloads, including batch processing, real-time streaming, machine learning, and graph processing.

Key Components of Apache Spark:

- **Driver Program:** Manages the execution of a Spark application.
- **Cluster Manager:** Allocates resources across the cluster (Standalone, Mesos, YARN, and Kubernetes).
- **Executors:** Worker nodes that are responsible for executing tasks.
- **Resilient Distributed Datasets (RDDs):** Fault-tolerant, immutable data structures that are operated in parallel.
- **DataFrames & Spark SQL:** Abstractions that enable structured data processing with SQL-like queries.

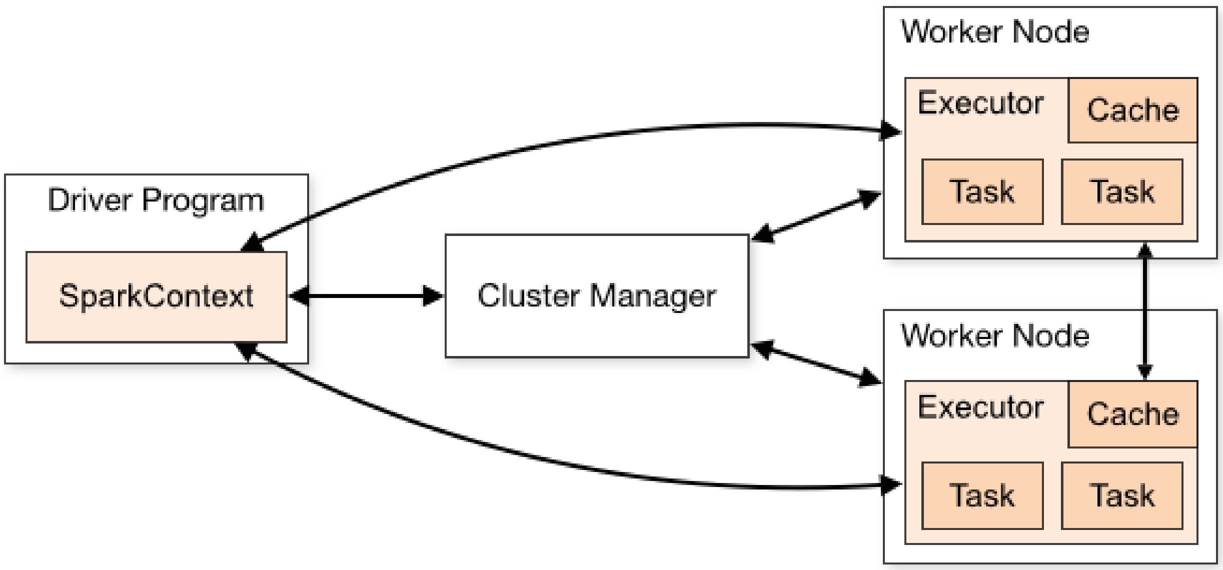


Figure 1: Apache Spark Architecture (Apache Spark, 2025)

Snowflake

Snowflake is a fully managed cloud-native SaaS (software as a service) designed to provide data warehousing services for scalable and high-performance data analytics. Unlike traditional database systems, Snowflake employs a distinctive architecture that separates compute and storage resources, utilizing each to scale independently.

Key Features of Snowflake:

- **Cloud Services Layer:** Manages authentication, metadata, query parsing, and optimization.
- **Compute Layer (Virtual Warehouses):** Consist of virtual warehouses that are responsible for executing queries and performing data operations. It also ensures significant parallel processing and scaling of resources.
- **Storage Layer:** Column-based storage format for optimized retrieval accessible through SQL queries.
- **Multi-Cloud Support:** Availability across different cloud platforms such as AWS, Azure, Google Cloud Platform (GCP).
- **Automatic Optimization:** Implements self-managing performance optimization features including automatic indexing, clustering, and compression algorithms that operate continuously without manual intervention that minimize database maintenance overhead.

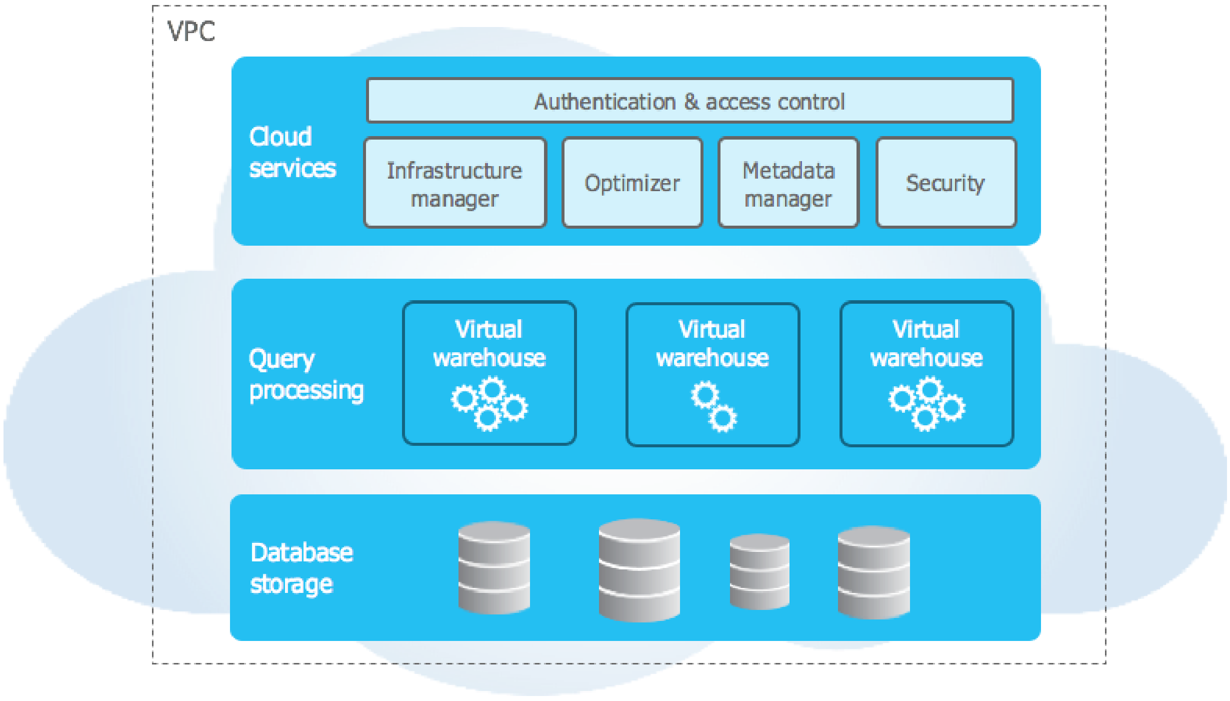


Figure 2: Snowflake Architecture (Snowflake, 2025)

1.2 Advantages of Apache Spark for Large-Scale Data Processing

1. **High Speed Processing:** Compared to disk-based processing frameworks like Hadoop MapReduce, in-memory computation substantially reduces execution time.
2. **Scalability:**
   Distributed computing model allows effective computation of large data sets across m ultiple nodes.
3. **Real-Time Analytics:** Data can be processed in real-time with Spark Streaming.
4. **Machine Learning Support:** Integrated MLlib enables scalable training for machine learning models.
5. **Ease to Use:** Supports APIs in Python, Java, R and Scala, making it accessible for data scientists and engineers.
6. **Flexibility:** It is supported by multiple storage solutions (HDFS, S3, Cassandra, etc.) and can be integrated with various big data tools.

1.3 Role of Snowflake in Cloud-Based Data Warehousing and Analytics

1. **Separation of Compute and Storage:** It enables independent scaling of processing power and storage, enhancing cost efficiency.
2. **Elastic Scalability:** Scales up or down automatically based on demand to deliver optimal performance.
3. **Zero Copy Cloning:** Enables instant duplication of databases and tables without the need for physical data copying.
4. **Time Travel and Data Recovery:** stores historical snapshots enabling easy rollback and auditing.
5. **High Concurrency:** Multiple users can execute queries simultaneously without affecting performance.
6. **Secure Data Sharing:** It enables real-time data exchange between Snowflake accounts without the need for data transfers.
7. **Multi-Cloud Support:** Vendor lock-in avoidance via multi-cloud support (AWS, Azure, GCP).

In conclusion, Apache Spark and Snowflake are robust big data processing and cloud data warehousing solutions respectively. Spark excels at fast, distributed computing, making it perfect for real-time analytics and large-scale machine learning.

In contrast, Snowflake offers a fully managed, scalable, and cost-effective cloud-based data warehousing solution. Together Spark and Snowflake enable streamlined storage, processing, and analysis of large datasets in modern big data ecosystems.

Chapter 2 Data Preprocessing with Apache Spark

2.1 Introduction

Data preprocessing is a critical stage in preparing raw data for analysis, since it ensures that the data is clean, consistent, and suitable for modelling. In this task, we utilise two popular frameworks that are Apache Spark and Python's Pandas to demonstrate data loading, cleansing, transformation, and feature engineering. Both frameworks are widely used in data science and analytics, however they are optimized for different types of workloads. Spark excels at distributed computing, making it ideal for large datasets with high dimensional data, whereas Pandas is more designed for smaller, in-memory data processing workloads. Our goal is to demonstrate how both tools can be utilised with the same datasets and compare their performance, particularly in terms of efficiency and execution time.

2.2 Dataset Overview

For this task, we used two datasets reflecting the operational performance of a solar power plant:

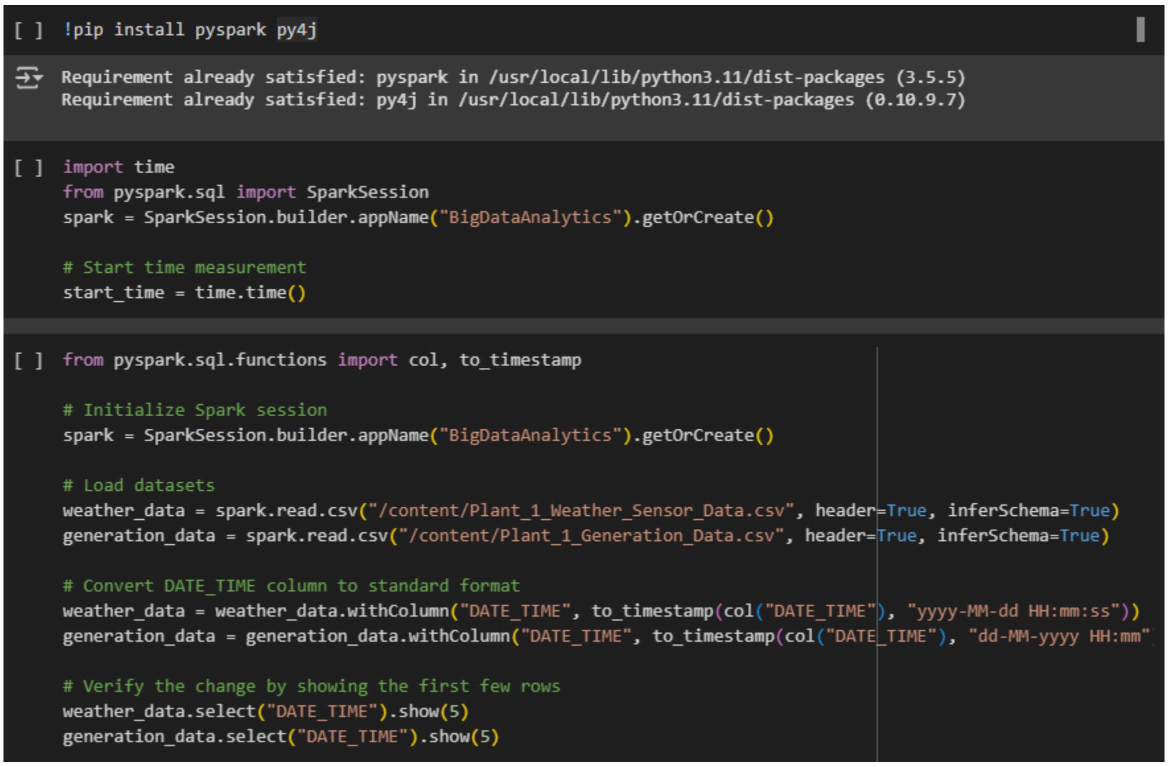
- Plant_1_Generation_Data.csv contains 68,779 rows and 7 columns, capturing power generation data from the plant.
- Plant_1_Weather_Sensor_Data.csv contains 3,183 rows and 6 columns providing weather-related data, including irradiation levels.

These datasets were selected as they represent real-world operational data from a solar power plant. Weather conditions like irradiation have a significant impact on the plant's power generation output. Analysing and combining these datasets can provide useful insights into how weather data impacts energy output, which is crucial for optimising plant operations.

2.3 Data Loading

Apache Spark Implementation

We initiate the preprocessing of data by loading both datasets into Apache Spark. Considering the DATE_TIME column contains timestamps, we standardised it across both datasets to ensure consistency. Below is the implementation:



```
[ ] !pip install pyspark py4j

Requirement already satisfied: pyspark in /usr/local/lib/python3.11/dist-packages (3.5.5)
Requirement already satisfied: py4j in /usr/local/lib/python3.11/dist-packages (0.10.9.7)
```

```
[ ] import time
    from pyspark.sql import SparkSession
    spark = SparkSession.builder.appName("BigDataAnalytics").getOrCreate()

    # Start time measurement
    start_time = time.time()
```

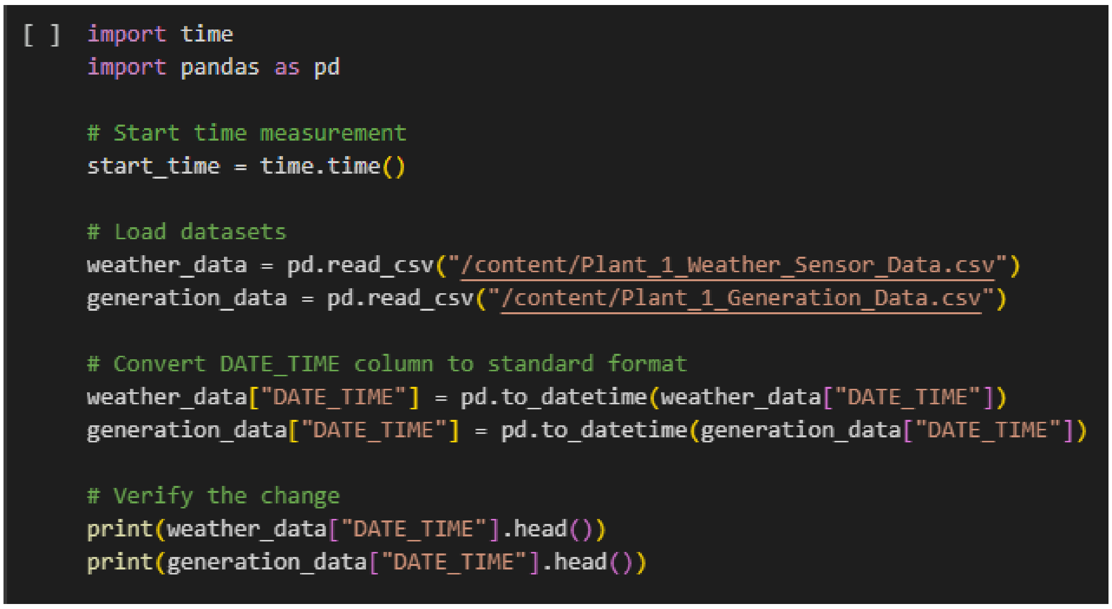
```
[ ] from pyspark.sql.functions import col, to_timestamp

    # Initialize Spark session
    spark = SparkSession.builder.appName("BigDataAnalytics").getOrCreate()

    # Load datasets
    weather_data = spark.read.csv("/content/Plant_1_Weather_Sensor_Data.csv", header=True, inferSchema=True)
    generation_data = spark.read.csv("/content/Plant_1_Generation_Data.csv", header=True, inferSchema=True)

    # Convert DATE_TIME column to standard format
    weather_data = weather_data.withColumn("DATE_TIME", to_timestamp(col("DATE_TIME"), "yyyy-MM-dd HH:mm:ss"))
    generation_data = generation_data.withColumn("DATE_TIME", to_timestamp(col("DATE_TIME"), "dd-MM-yyyy HH:mm"

    # Verify the change by showing the first few rows
    weather_data.select("DATE_TIME").show(5)
    generation_data.select("DATE_TIME").show(5)
```

Figure 3: Data Loading - Apache Spark

Apache Spark is designed for distributed data processing, thus it's ideal for large-scale data loading workloads. It automatically splits datasets across multiple nodes in a cluster, which improves performance when dealing with large amounts of data.

Pandas Implementation

For Pandas, the process of loading the datasets is straightforward. We load the data using pd.read_csv() and standardize the DATE_TIME column with pd.to_datetime().



```
[ ] import time
    import pandas as pd

    # Start time measurement
    start_time = time.time()

    # Load datasets
    weather_data = pd.read_csv("/content/Plant_1_Weather_Sensor_Data.csv")
    generation_data = pd.read_csv("/content/Plant_1_Generation_Data.csv")

    # Convert DATE_TIME column to standard format
    weather_data["DATE_TIME"] = pd.to_datetime(weather_data["DATE_TIME"])
    generation_data["DATE_TIME"] = pd.to_datetime(generation_data["DATE_TIME"])

    # Verify the change
    print(weather_data["DATE_TIME"].head())
    print(generation_data["DATE_TIME"].head())
```

Figure 4: Data Loading – Pandas

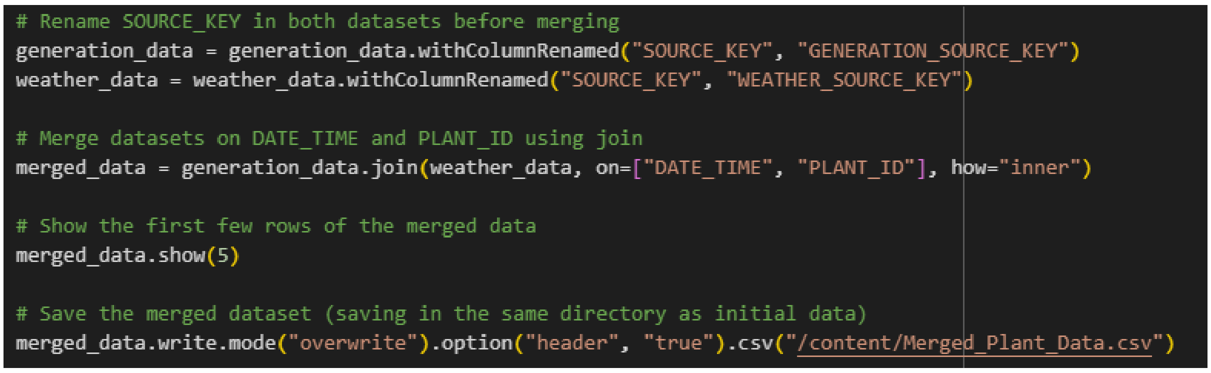
Pandas loads data in memory, which is appropriate for smaller datasets. However, for larger datasets, this strategy might lead to high memory use and slower performance due to storing everything in memory.

2.4 Data Merging

After loading the datasets, we combined them based on the DATE_TIME and PLANT_ID columns. This ensures that we have merged weather data with the respective power generation data for each timestamp.

Apache Spark Implementation

In Spark, we used the join() function to merge the datasets.

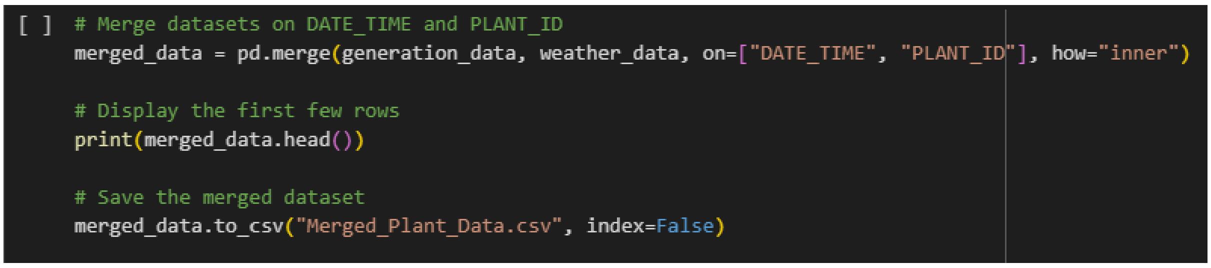


```
# Rename SOURCE_KEY in both datasets before merging
generation_data = generation_data.withColumnRenamed("SOURCE_KEY", "GENERATION_SOURCE_KEY")
weather_data = weather_data.withColumnRenamed("SOURCE_KEY", "WEATHER_SOURCE_KEY")

# Merge datasets on DATE_TIME and PLANT_ID using join
merged_data = generation_data.join(weather_data, on=["DATE_TIME", "PLANT_ID"], how="inner")

# Show the first few rows of the merged data
merged_data.show(5)

# Save the merged dataset (saving in the same directory as initial data)
merged_data.write.mode("overwrite").option("header", "true").csv("/content/Merged_Plant_Data.csv")
```

Figure 5: Data Merging - Apache Spark

Pandas Implementation

In Pandas, we utilized the pd.merge() function for merging the datasets.



```
[ ]  # Merge datasets on DATE_TIME and PLANT_ID
     merged_data = pd.merge(generation_data, weather_data, on=["DATE_TIME", "PLANT_ID"], how="inner")

     # Display the first few rows
     print(merged_data.head())

     # Save the merged dataset
     merged_data.to_csv("Merged_Plant_Data.csv", index=False)
```

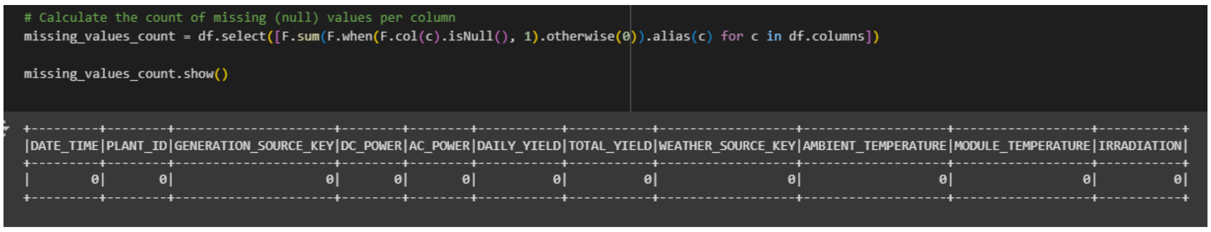
Figure 6: Data Merging – Pandas

The same merge operation is performed both by Spark and Pandas, but Spark has the benefits from its distributed nature, which allows it to handle larger datasets more efficiently by executing the merge in parallel across multiple nodes.

2.5 Data Cleansing and Transformation

A comprehensive data cleansing and feature engineering process was implemented. Redundant columns, primarily 'Plant_ID', 'GENERATION_SOURCE_KEY', and 'WEATHER_SOURCE_KEY', were removed to streamline the dataset and reduce dimensionality. Erroneous zero values were found in the 'DC_POWER' and 'AC_POWER' columns, particularly during the nighttime hours (00:00 - 06:00 and 18:00 - 23:00). Given that these zeros aligned with periods of negligible solar irradiation, they were retained as valid data points. Daylight zero numbers (09:00-15:00) were considered anomalies, potentially caused by temporary weather or sensor faults.

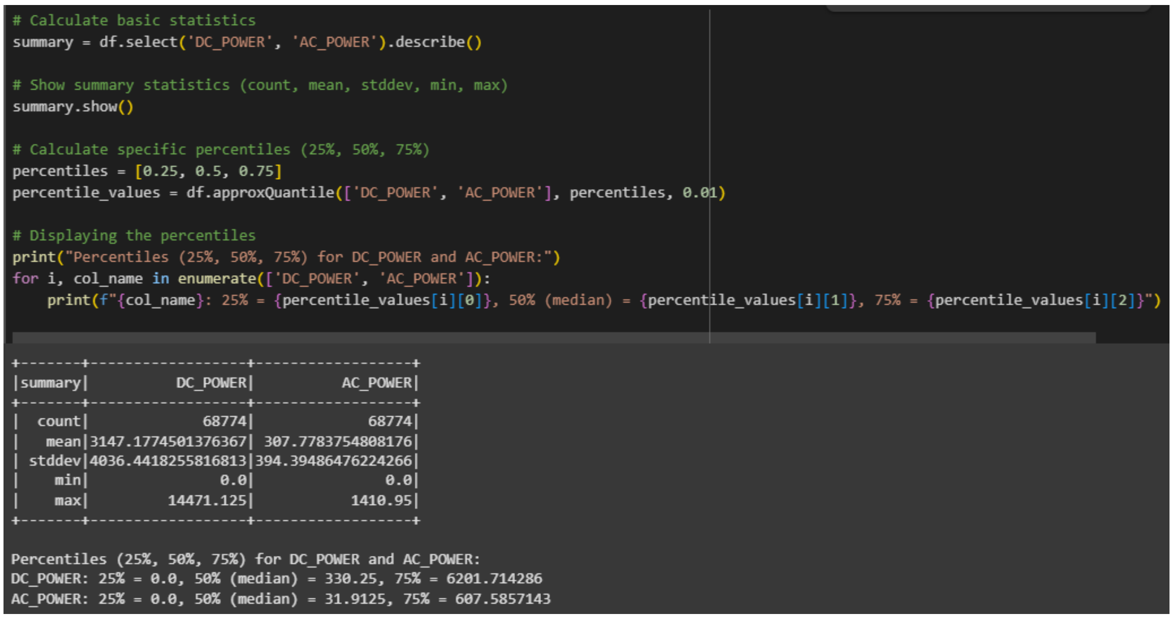
In addition, summary statistics for DC_POWER and AC_POWER were calculated using Apache Spark to evaluate data distribution and variability. Key metrics (mean, standard deviation, min, and max) were computed using describe(), while approxQuantile() estimated the 25th, 50th, and 75th percentiles, aiding anomaly detection and feature engineering.



```
# Calculate the count of missing (null) values per column
missing_values_count = df.select([F.sum(F.when(F.col(c).isNull(), 1).otherwise(0)).alias(c) for c in df.columns])

missing_values_count.show()
```

```
+---------+--------+---------------------+--------+--------+-----------+-----------+------------------+-------------------+------------------+-----------+
|DATE_TIME|PLANT_ID|GENERATION_SOURCE_KEY|DC_POWER|AC_POWER|DAILY_YIELD|TOTAL_YIELD|WEATHER_SOURCE_KEY|AMBIENT_TEMPERATURE|MODULE_TEMPERATURE|IRRADIATION|
+---------+--------+---------------------+--------+--------+-----------+-----------+------------------+-------------------+------------------+-----------+
|        0|       0|                    0|       0|       0|          0|          0|                 0|                  0|                 0|          0|
+---------+--------+---------------------+--------+--------+-----------+-----------+------------------+-------------------+------------------+-----------+
```

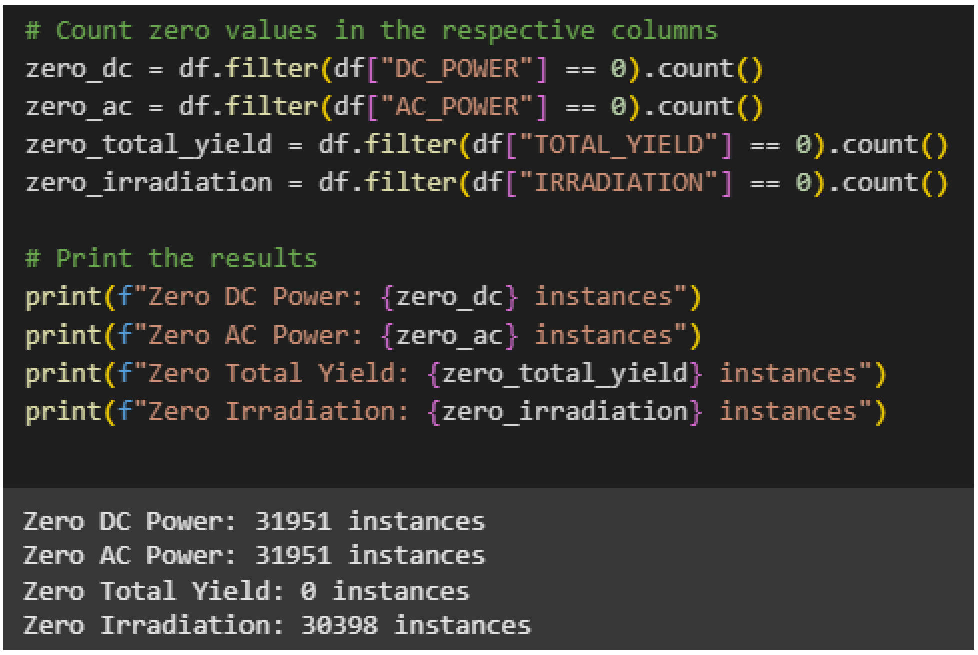
Figure 7: Missing Value Analysis – Spark



```
# Calculate basic statistics
summary = df.select('DC_POWER', 'AC_POWER').describe()

# Show summary statistics (count, mean, stddev, min, max)
summary.show()

# Calculate specific percentiles (25%, 50%, 75%)
percentiles = [0.25, 0.5, 0.75]
percentile_values = df.approxQuantile(['DC_POWER', 'AC_POWER'], percentiles, 0.01)

# Displaying the percentiles
print("Percentiles (25%, 50%, 75%) for DC_POWER and AC_POWER:")
for i, col_name in enumerate(['DC_POWER', 'AC_POWER']):
    print(f"{col_name}: 25% = {percentile_values[i][0]}, 50% (median) = {percentile_values[i][1]}, 75% = {percentile_values[i][2]}")
```

```
+-------+------------------+------------------+
|summary|          DC_POWER|          AC_POWER|
+-------+------------------+------------------+
|  count|             68774|             68774|
|   mean|3147.1774501376367| 307.7783754808176|
| stddev|4036.4418255816813|394.39486476224266|
|    min|               0.0|               0.0|
|    max|         14471.125|           1410.95|
+-------+------------------+------------------+

Percentiles (25%, 50%, 75%) for DC_POWER and AC_POWER:
DC_POWER: 25% = 0.0, 50% (median) = 330.25, 75% = 6201.714286
AC_POWER: 25% = 0.0, 50% (median) = 31.9125, 75% = 607.5857143
```

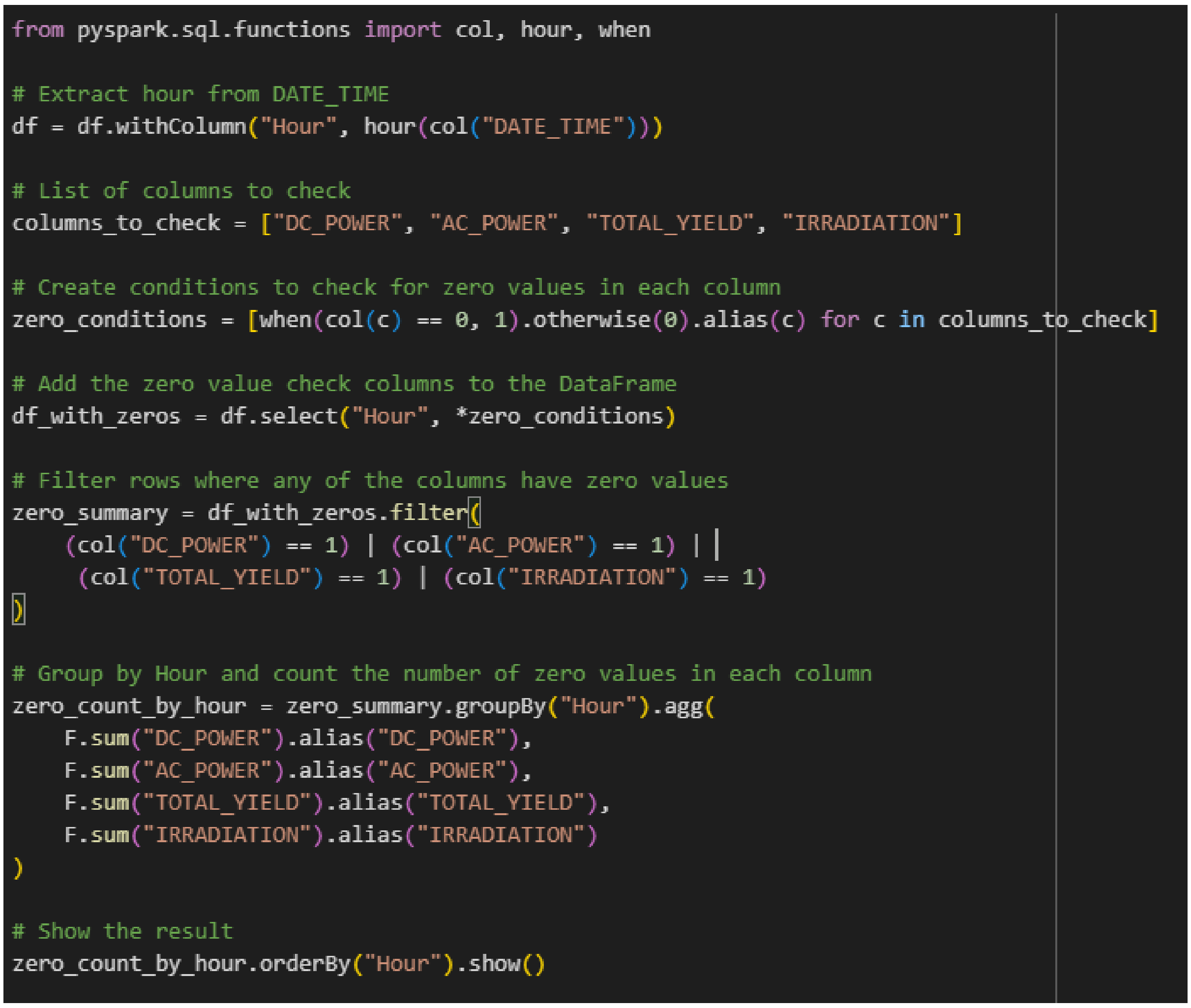
Figure 8: Descriptive Statistics and Percentile Analysis – Spark



```
# Count zero values in the respective columns
zero_dc = df.filter(df["DC_POWER"] == 0).count()
zero_ac = df.filter(df["AC_POWER"] == 0).count()
zero_total_yield = df.filter(df["TOTAL_YIELD"] == 0).count()
zero_irradiation = df.filter(df["IRRADIATION"] == 0).count()

# Print the results
print(f"Zero DC Power: {zero_dc} instances")
print(f"Zero AC Power: {zero_ac} instances")
print(f"Zero Total Yield: {zero_total_yield} instances")
print(f"Zero Irradiation: {zero_irradiation} instances")
```

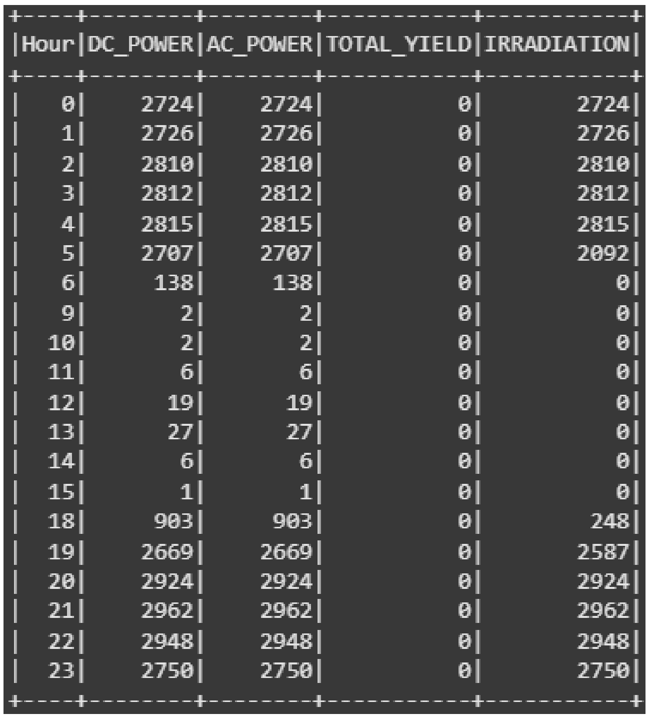
```
Zero DC Power: 31951 instances
Zero AC Power: 31951 instances
Zero Total Yield: 0 instances
Zero Irradiation: 30398 instances
```

Figure 9: Zero Value Analysis – Spark



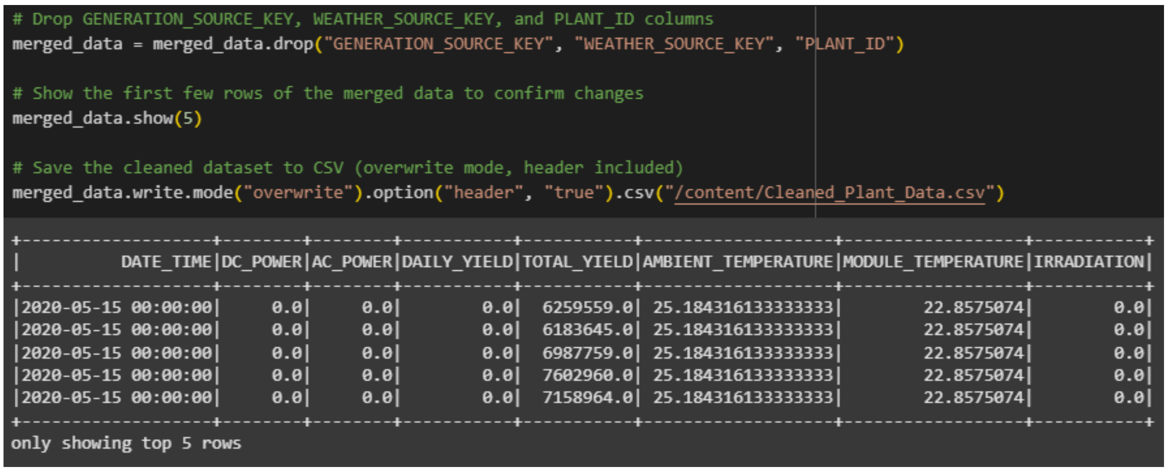
```
from pyspark.sql.functions import col, hour, when

# Extract hour from DATE_TIME
df = df.withColumn("Hour", hour(col("DATE_TIME")))

# List of columns to check
columns_to_check = ["DC_POWER", "AC_POWER", "TOTAL_YIELD", "IRRADIATION"]

# Create conditions to check for zero values in each column
zero_conditions = [when(col(c) == 0, 1).otherwise(0).alias(c) for c in columns_to_check]

# Add the zero value check columns to the DataFrame
df_with_zeros = df.select("Hour", *zero_conditions)

# Filter rows where any of the columns have zero values
zero_summary = df_with_zeros.filter(
    (col("DC_POWER") == 1) | (col("AC_POWER") == 1) |
     (col("TOTAL_YIELD") == 1) | (col("IRRADIATION") == 1)
)

# Group by Hour and count the number of zero values in each column
zero_count_by_hour = zero_summary.groupBy("Hour").agg(
    F.sum("DC_POWER").alias("DC_POWER"),
    F.sum("AC_POWER").alias("AC_POWER"),
    F.sum("TOTAL_YIELD").alias("TOTAL_YIELD"),
    F.sum("IRRADIATION").alias("IRRADIATION")
)

# Show the result
zero_count_by_hour.orderBy("Hour").show()
```

Figure 10: Hourly Zero Value Distribution - Spark



```
+----+--------+--------+-----------+-----------+
|Hour|DC_POWER|AC_POWER|TOTAL_YIELD|IRRADIATION|
+----+--------+--------+-----------+-----------+
|   0|    2724|    2724|          0|       2724|
|   1|    2726|    2726|          0|       2726|
|   2|    2810|    2810|          0|       2810|
|   3|    2812|    2812|          0|       2812|
|   4|    2815|    2815|          0|       2815|
|   5|    2707|    2707|          0|       2092|
|   6|     138|     138|          0|          0|
|   9|       2|       2|          0|          0|
|  10|       2|       2|          0|          0|
|  11|       6|       6|          0|          0|
|  12|      19|      19|          0|          0|
|  13|      27|      27|          0|          0|
|  14|       6|       6|          0|          0|
|  15|       1|       1|          0|          0|
|  18|     903|     903|          0|        248|
|  19|    2669|    2669|          0|       2587|
|  20|    2924|    2924|          0|       2924|
|  21|    2962|    2962|          0|       2962|
|  22|    2948|    2948|          0|       2948|
|  23|    2750|    2750|          0|       2750|
+----+--------+--------+-----------+-----------+
```

Figure 11: Hourly Zero Value Distribution Results – Spark

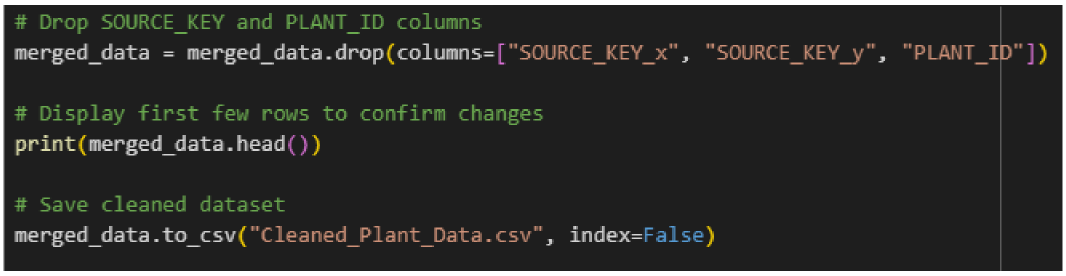


```python
# Drop GENERATION_SOURCE_KEY, WEATHER_SOURCE_KEY, and PLANT_ID columns
merged_data = merged_data.drop("GENERATION_SOURCE_KEY", "WEATHER_SOURCE_KEY", "PLANT_ID")

# Show the first few rows of the merged data to confirm changes
merged_data.show(5)

# Save the cleaned dataset to CSV (overwrite mode, header included)
merged_data.write.mode("overwrite").option("header", "true").csv("/content/Cleaned_Plant_Data.csv")
```

```
+-------------------+--------+--------+-----------+-----------+-------------------+-------------------+-----------+
|          DATE_TIME|DC_POWER|AC_POWER|DAILY_YIELD|TOTAL_YIELD|AMBIENT_TEMPERATURE|MODULE_TEMPERATURE|IRRADIATION|
+-------------------+--------+--------+-----------+-----------+-------------------+-------------------+-----------+
|2020-05-15 00:00:00|     0.0|     0.0|        0.0|  6259559.0| 25.184316133333333|         22.8575074|        0.0|
|2020-05-15 00:00:00|     0.0|     0.0|        0.0|  6183645.0| 25.184316133333333|         22.8575074|        0.0|
|2020-05-15 00:00:00|     0.0|     0.0|        0.0|  6987759.0| 25.184316133333333|         22.8575074|        0.0|
|2020-05-15 00:00:00|     0.0|     0.0|        0.0|  7602960.0| 25.184316133333333|         22.8575074|        0.0|
|2020-05-15 00:00:00|     0.0|     0.0|        0.0|  7158964.0| 25.184316133333333|         22.8575074|        0.0|
+-------------------+--------+--------+-----------+-----------+-------------------+-------------------+-----------+
only showing top 5 rows
```

Figure 12: Dataset Cleaning and Column Removal – Spark

After the same data cleansing and transformation process was carried out with Python (Pandas), redundant columns were removed, specifically the 'SOURCE_KEY' and 'PLANT_ID'. It was implemented as below:



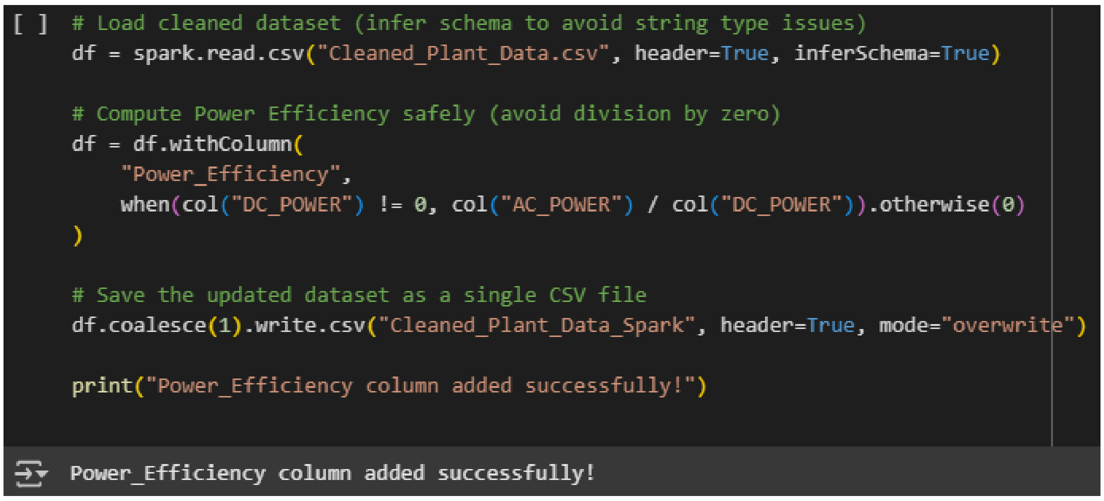
```python
# Drop SOURCE_KEY and PLANT_ID columns
merged_data = merged_data.drop(columns=["SOURCE_KEY_x", "SOURCE_KEY_y", "PLANT_ID"])

# Display first few rows to confirm changes
print(merged_data.head())

# Save cleaned dataset
merged_data.to_csv("Cleaned_Plant_Data.csv", index=False)
```

Figure 13: Dataset Cleaning and Column Removal - Pandas

2.6 Feature Engineering

After the data cleansing and transformation phases, a new feature called 'Power_Efficiency' was engineered by calculating the ratio of 'AC_POWER' to 'DC_POWER'. This derived feature was implemented to provide a normalised measure of the solar power plant's conversion efficiency, allowing for a more detailed knowledge of performance variations over different time periods and conditions. This feature engineering stage aimed to improve the model's ability to identify subtle patterns related to power generation efficiency.

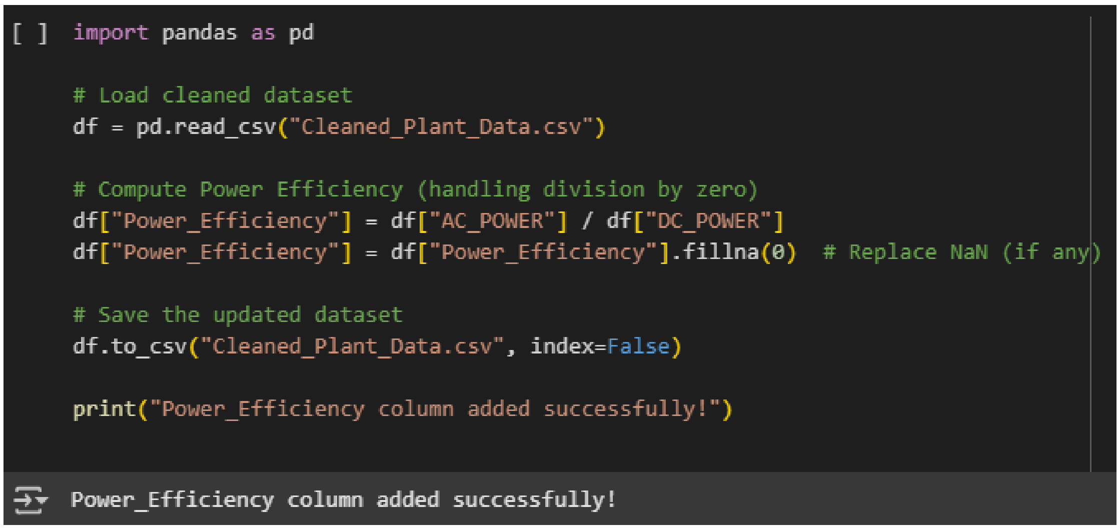
The code below demonstrates how this new feature was implemented using Apache Spark.



```
[ ] # Load cleaned dataset (infer schema to avoid string type issues)
    df = spark.read.csv("Cleaned_Plant_Data.csv", header=True, inferSchema=True)

    # Compute Power Efficiency safely (avoid division by zero)
    df = df.withColumn(
        "Power_Efficiency",
        when(col("DC_POWER") != 0, col("AC_POWER") / col("DC_POWER")).otherwise(0)
    )

    # Save the updated dataset as a single CSV file
    df.coalesce(1).write.csv("Cleaned_Plant_Data_Spark", header=True, mode="overwrite")

    print("Power_Efficiency column added successfully!")

    Power_Efficiency column added successfully!
```

Figure 14: Power Efficiency Feature Engineering – Spark

Subsequently, a similar feature engineering approach was followed with Python (Pandas) for performance comparison with Apache Spark.



```
[ ] import pandas as pd

    # Load cleaned dataset
    df = pd.read_csv("Cleaned_Plant_Data.csv")

    # Compute Power Efficiency (handling division by zero)
    df["Power_Efficiency"] = df["AC_POWER"] / df["DC_POWER"]
    df["Power_Efficiency"] = df["Power_Efficiency"].fillna(0)  # Replace NaN (if any)

    # Save the updated dataset
    df.to_csv("Cleaned_Plant_Data.csv", index=False)

    print("Power_Efficiency column added successfully!")

    Power_Efficiency column added successfully!
```

Figure 15: Power Efficiency Feature Engineering – Pandas

2.7 Performance Comparison: Apache Spark vs. Pandas

To evaluate the efficiency of Apache Spark and Pandas in data preprocessing, we computed the total execution time for key tasks which include data loading, cleansing, transformation, and feature engineering.

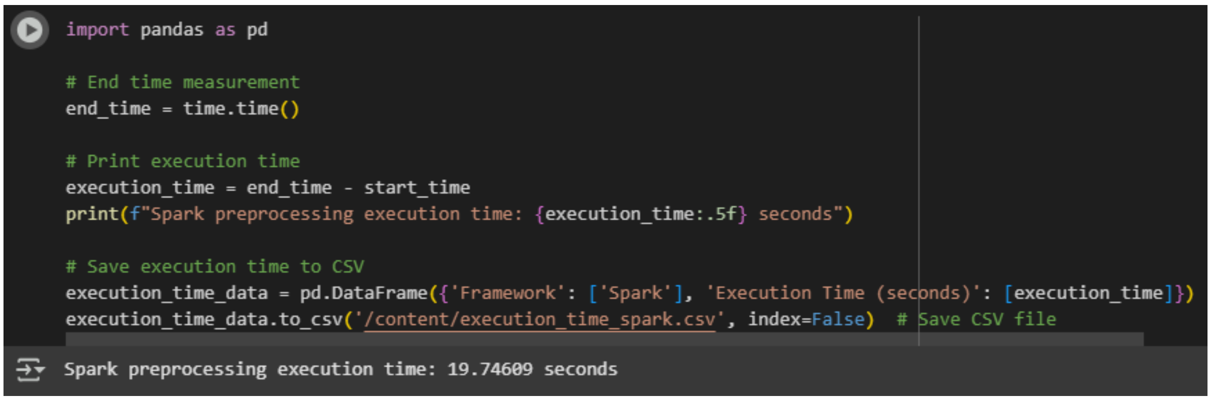
Execution Time Analysis

Execution times for both implementations were calculated using timestamp-based measurements. The results demonstrated that:

- Apache Spark's distributed computing model resulted in better performance for larger datasets.
- Pandas, which operate entirely in memory, proved more efficient for our dataset with fewer than 100,000 rows.

To measure execution times, we implemented the following code snippets:

Apache Spark Execution Time Calculation

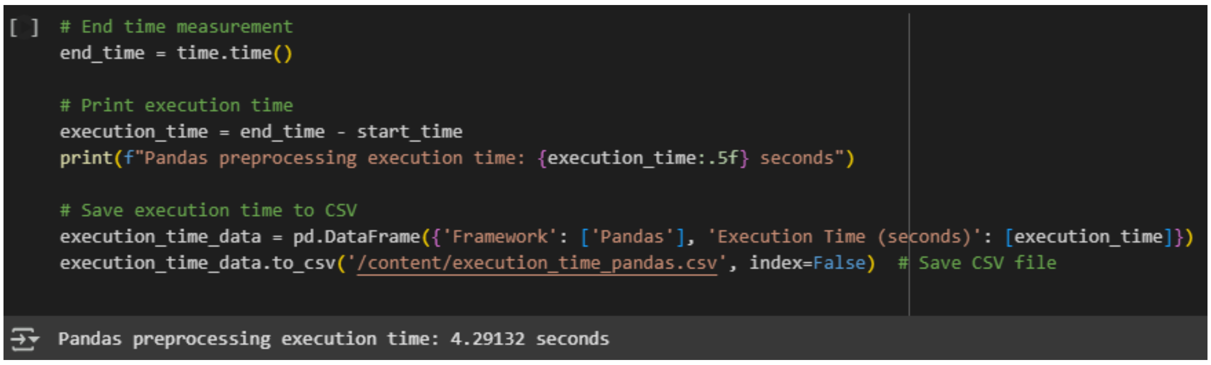


```
import pandas as pd

# End time measurement
end_time = time.time()

# Print execution time
execution_time = end_time - start_time
print(f"Spark preprocessing execution time: {execution_time:.5f} seconds")

# Save execution time to CSV
execution_time_data = pd.DataFrame({'Framework': ['Spark'], 'Execution Time (seconds)': [execution_time]})
execution_time_data.to_csv('/content/execution_time_spark.csv', index=False)  # Save CSV file
```

```
Spark preprocessing execution time: 19.74609 seconds
```

Figure 16: Apache Spark Execution Time Calculation

Pandas Execution Time Calculation



```
# End time measurement
end_time = time.time()

# Print execution time
execution_time = end_time - start_time
print(f"Pandas preprocessing execution time: {execution_time:.5f} seconds")

# Save execution time to CSV
execution_time_data = pd.DataFrame({'Framework': ['Pandas'], 'Execution Time (seconds)': [execution_time]})
execution_time_data.to_csv('/content/execution_time_pandas.csv', index=False)  # Save CSV file
```

```
Pandas preprocessing execution time: 4.29132 seconds
```

Figure 17: Pandas Execution Time Calculation

Discussion

While Apache Spark is designed for large-scale processing of data, the overhead of distributed computing may make it less effective for smaller datasets. Pandas, on the other hand, process all data entirely in memory on a single machine, making it a better choice for relatively small datasets.

- Pandas' in-memory calculation improves performance for small datasets (< 100,000 rows).
- Spark outperforms Pandas on larger datasets due to its efficient scaling across distributed environments.
- Pandas struggle with large data sets due to their memory restrictions, whereas Spark distributes workloads.

Figure 18 visualizes the preprocessing execution time comparison for Apache Spark and Pandas.

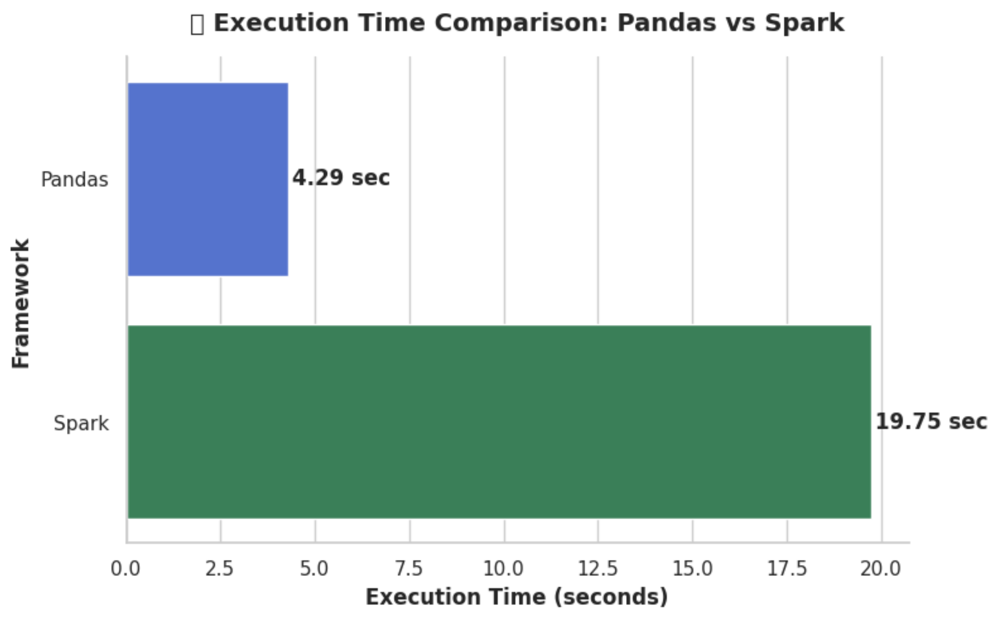


Figure 18: Execution Time Comparison

Chapter 3 Real-time Analytics with Snowflake

This section explains the step-by-step approach used to integrate Snowflake with Google Cloud Storage (GCS), load datasets, and process real-time solar power data.

1. **Storage Integration with GCS**

```
CREATE OR REPLACE STORAGE INTEGRATION gcs_integration
TYPE = EXTERNAL_STAGE
STORAGE_PROVIDER = GCS
ENABLED = TRUE
STORAGE_ALLOWED_LOCATIONS = ('gcs://solarpowerplantdata/');
```

- This command establishes a **storage integration** between Snowflake and Google Cloud Storage (GCS).
- It allows Snowflake to access external data stored in a GCS bucket (solarpowerplantdata).
- The ENABLED = TRUE ensures Snowflake can retrieve data from this bucket

2. **File Format Definition for CSV**

```
CREATE OR REPLACE FILE FORMAT my_csv_format
TYPE = CSV
FIELD_OPTIONALLY_ENCLOSED_BY = '"'
SKIP_HEADER = 1
TIMESTAMP_FORMAT = 'MM/DD/YYYY HH24:MI';
```

Defines a CSV file format (my_csv_format) for structured data ingestion.

The format specifies:

- SKIP_HEADER = 1 → Ignores the first row (header).
- FIELD_OPTIONALLY_ENCLOSED_BY = '"' → Handles fields enclosed in double quotes.
- TIMESTAMP_FORMAT = 'MM/DD/YYYY HH24:MI' → Ensures timestamps are correctly parsed

3. **Creating a Stage for Data Loading**

```
CREATE OR REPLACE STAGE gcs_stage
URL = 'gcs://solarpowerplantdata/'
STORAGE_INTEGRATION = gcs_integration
FILE_FORMAT = my_csv_format;
```

- Creates an **external stage** named gcs_stage, pointing to the GCS bucket.
- Associates it with the previously created gcs_integration and file format my_csv_format.

- This allows Snowflake to directly access and manage files stored in GCS.

4. **Creating Tables for Raw and Processed Data**

Raw Data Table

```
          CREATE OR REPLACE TABLE raw_power_data (
             "DATE_TIME" TIMESTAMP_LTZ,
             "DC_POWER" FLOAT,
             "AC_POWER" FLOAT,
             "DAILY_YIELD" FLOAT,
             "TOTAL_YIELD" FLOAT,
             "AMBIENT_TEMPERATURE" FLOAT,
             "MODULE_TEMPERATURE" FLOAT,
             "IRRADIATION" FLOAT,
             "Hour" INT,
             "Power_Efficiency" FLOAT
);
```

- Defines raw_power_data, which stores raw data extracted from the GCS bucket.
- Contains columns such as DC_POWER, AC_POWER, DAILY_YIELD, temperature metrics, and irradiation levels.

Load Data into the Raw Data Table

```
COPY INTO raw_power_data
FROM @gcs_stage/Cleaned_Plant_Data.csv
FILE_FORMAT = my_csv_format;
```

- Copies data from the external stage (gcs_stage) into the raw_power_data table.
- Uses the defined CSV file format (my_csv_format) to ensure proper parsing.

Cleaned Data Table

```
CREATE OR REPLACE TABLE cleaned_power_data (
   "DATE_TIME" TIMESTAMP_LTZ,
   "DC_POWER" FLOAT,
   "AC_POWER" FLOAT,
   "DAILY_YIELD" FLOAT,
   "TOTAL_YIELD" FLOAT,
   "AMBIENT_TEMPERATURE" FLOAT,
   "MODULE_TEMPERATURE" FLOAT,
   "IRRADIATION" FLOAT,
   "Hour" INT,
   "Power_Efficiency" FLOAT
);
```

- Defines cleaned_power_data, which will store processed and refined records.
- This table is used for further analysis and insights.

5. **Automating Data Loading with Snowpipe**

```
CREATE OR REPLACE PIPE power_data_pipe
AS
COPY INTO raw_power_data
FROM @gcs_stage
FILE_FORMAT = (TYPE = CSV FIELD_OPTIONALLY_ENCLOSED_BY
= '"' SKIP_HEADER = 1);
```

- Snowpipe (power_data_pipe) is set up to automatically load new files from gcs_stage into raw_power_data.
- Ensures real-time data ingestion by constantly monitoring the stage for new files.

6. **Streaming Data Processing**

```
CREATE OR REPLACE STREAM power_data_stream
ON TABLE raw_power_data
APPEND_ONLY = TRUE;
```

- Creates a stream (power_data_stream) that tracks new records added to raw_power_data.
- The APPEND_ONLY = TRUE setting ensures only newly inserted rows are tracked

7. **Scheduled Task for Data Processing**

```
CREATE OR REPLACE TASK process_power_data_task
WAREHOUSE = COMPUTE_WH
SCHEDULE = '30 SECOND'
WHEN SYSTEM$STREAM_HAS_DATA('power_data_stream')
AS
INSERT INTO cleaned_power_data (
   "DATE_TIME", "DC_POWER", "AC_POWER", "DAILY_YIELD",
"TOTAL_YIELD",
   "AMBIENT_TEMPERATURE", "MODULE_TEMPERATURE",
"IRRADIATION", "Hour", "Power_Efficiency"
)
SELECT
   "DATE_TIME", "DC_POWER", "AC_POWER", "DAILY_YIELD",
"TOTAL_YIELD",
   "AMBIENT_TEMPERATURE", "MODULE_TEMPERATURE",
"IRRADIATION", "Hour", "Power_Efficiency"
FROM power_data_stream;
```

- Creates a scheduled task (process_power_data_task) that runs every 30 seconds.
- It checks if new data exists in power_data_stream (WHEN SYSTEM$STREAM_HAS_DATA('power_data_stream')).
- If new data is detected, it inserts it into cleaned_power_data.

8. Activating the Task

```
ALTER TASK process_power_data_task RESUME;
```

- Resumes the scheduled task, ensuring real-time data processing.

9. Verifying Data Processing

Check Cleaned Data Table

```
SELECT * FROM cleaned_power_data ORDER BY DATE_TIME DESC LIMIT 10;
```

- Retrieves the latest 10 processed records from cleaned_power_data.

Check Stream Status

```
SELECT SYSTEM$STREAM_HAS_DATA('power_data_stream');
```

- Checks if new data is available in the power_data_stream.

Chapter 4 Cloud Analytics with Snowflake

4.1 Introduction

In order to obtain important insights, this section of the report covers data integration, importing data sets onto Snowflake, and running SQL queries. To demonstrate Snowflake's capacity for big data analytics, its special features such as time travel and query optimization are also discussed.

4.2 Data Loading and Integration

1. Storage Integration: To access the data collection, Snowflake must be integrated with Google Cloud Storage (GCS).
2. File Formatting: To ensure that the data is formatted correctly, specify the CSV file's format.
3. Staging Data: To manage external data, load a Snowflake Stage.
4. Table Creation: Raw_power_data and cleaned_power_data are defined as structured tables.
5. Automated Data Ingestion: Snowpipe is used to update data in real time.
6. Data Processing & Streaming: Real-time analytics through the use of tasks and streams.

SQL Snippets for Data Loading:

```
-- Create Raw Power Data Table
CREATE OR REPLACE TABLE raw_power_data (
   "DATE_TIME" TIMESTAMP_LTZ,
   "DC_POWER" FLOAT,
   "AC_POWER" FLOAT,
   "DAILY_YIELD" FLOAT,
   "TOTAL_YIELD" FLOAT,
   "AMBIENT_TEMPERATURE" FLOAT,
   "MODULE_TEMPERATURE" FLOAT,
   "IRRADIATION" FLOAT,
   "Hour" INT,
   "Power_Efficiency" FLOAT
);

-- Load Data into Snowflake
COPY INTO raw_power_data
FROM @gcs_stage/Cleaned_Plant_Data.csv
FILE_FORMAT = my_csv_format;
```

4.3 Key Insights Extracted Using SQL Queries

Peak and Off-Peak Power Generation Hours

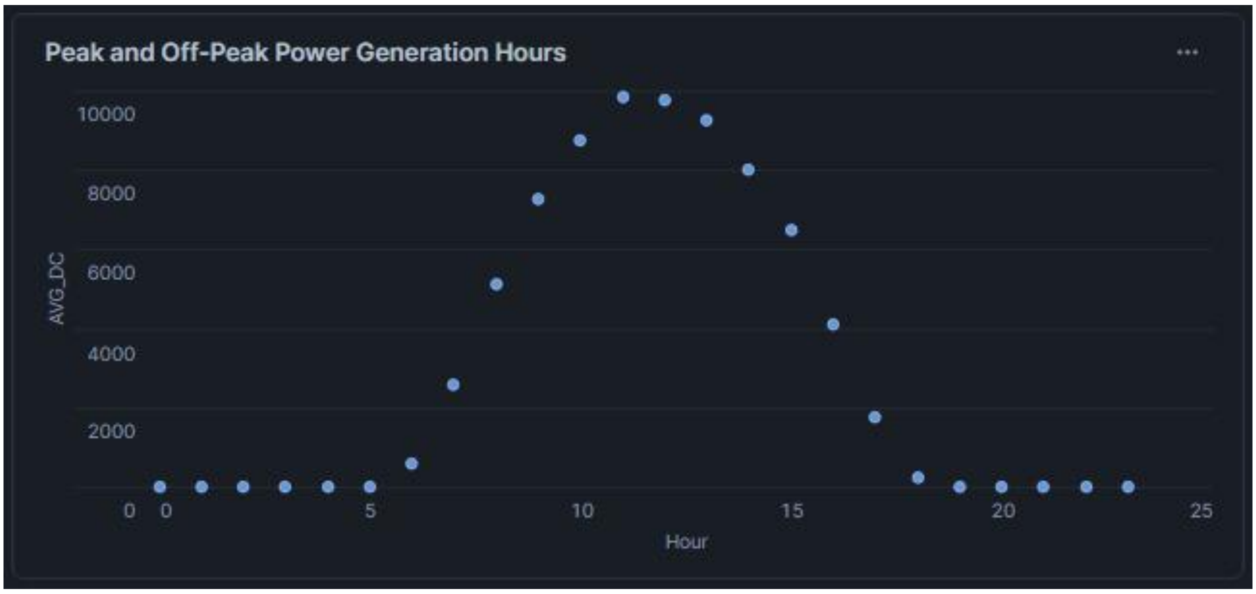


Figure 19: Peak and Off-Peak Power Generation Hours Visualization

The following query was run in order to identify the hours with the highest and lowest power output:

```
SELECT "Hour", AVG(DC_POWER) AS Avg_DC, AVG(AC_POWER) AS Avg_AC
FROM raw_power_data
GROUP BY "Hour"
ORDER BY Avg_DC DESC;
```

Results:

- Peak hours are indicated by hours with noticeably higher DC and AC power production.
- Off-peak hours are indicated by lower power output during certain hours.
- Energy supply and storage can be optimised with the help of this data.

Power Output on Different Days

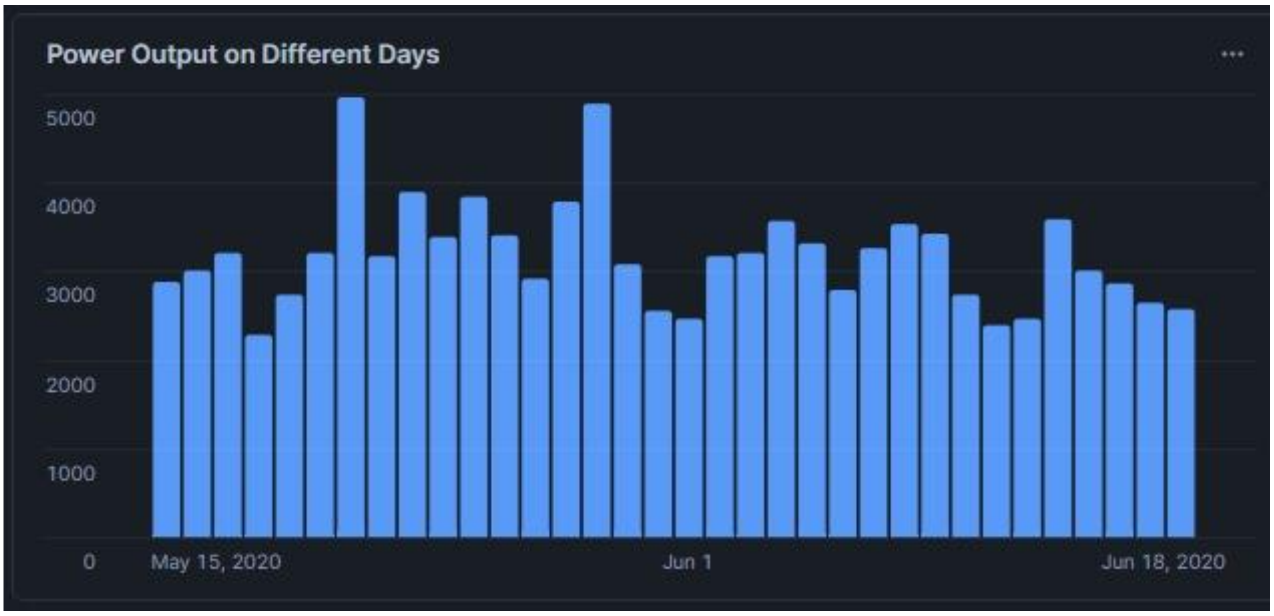


Figure 20: Power Output on Different Days Visualization

The following query was run in order to examine the daily variations in power output:

```
SELECT DATE_TIME::DATE AS Date, AVG(DC_POWER) AS Avg_DC
FROM raw_power_data
GROUP BY Date
ORDER BY Date;
```

Results:

- The data show daily variations in power generation, which may be related to seasonality and weather.
- Forecasting power availability may be made possible by identifying patterns in daily output.

Differentiation Between DC Power & AC Power Across Days

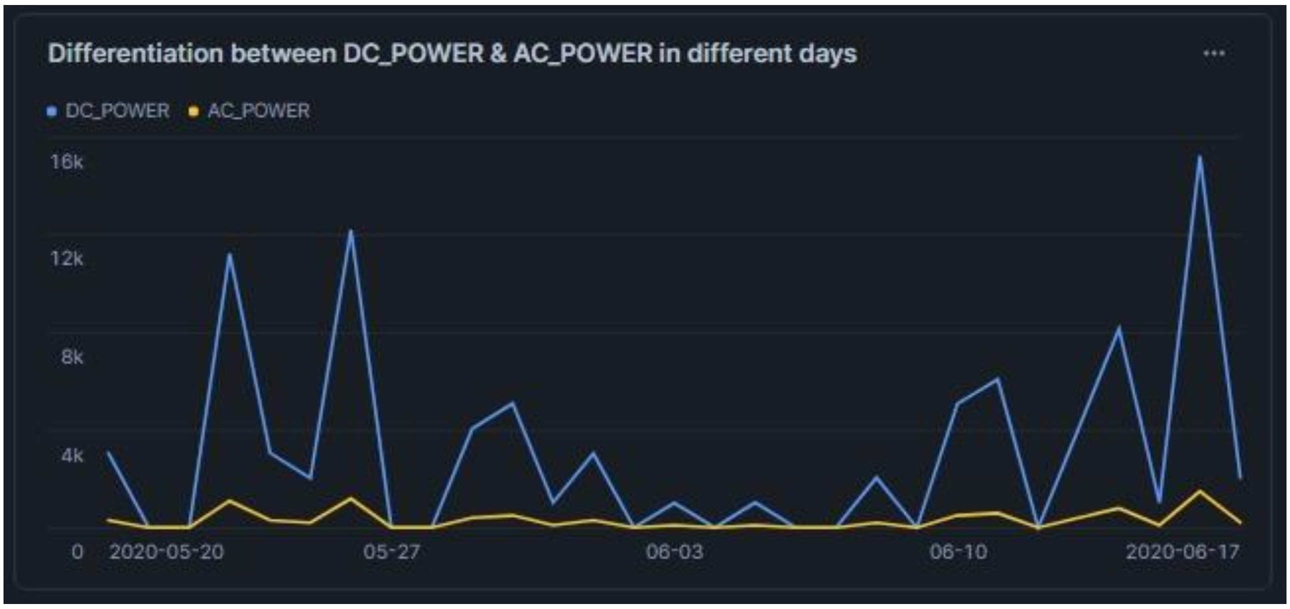


Figure 21: Differentiation Between DC Power & AC Power Across Days Visualization

The following query was run in order to look for anomalies where DC power is high and AC power is low:

```
SELECT *
FROM raw_power_data
WHERE DC_POWER > 1000 AND AC_POWER < 100;
```

Results:

- The query finds instances in which the AC power conversion is wasteful while the DC power output is actually high.
- These circumstances could be a sign of inverter failures or system inefficiencies.
- It is advised that equipment maintenance and operating performance be examined in greater detail.

Weather Impact on Power Generation

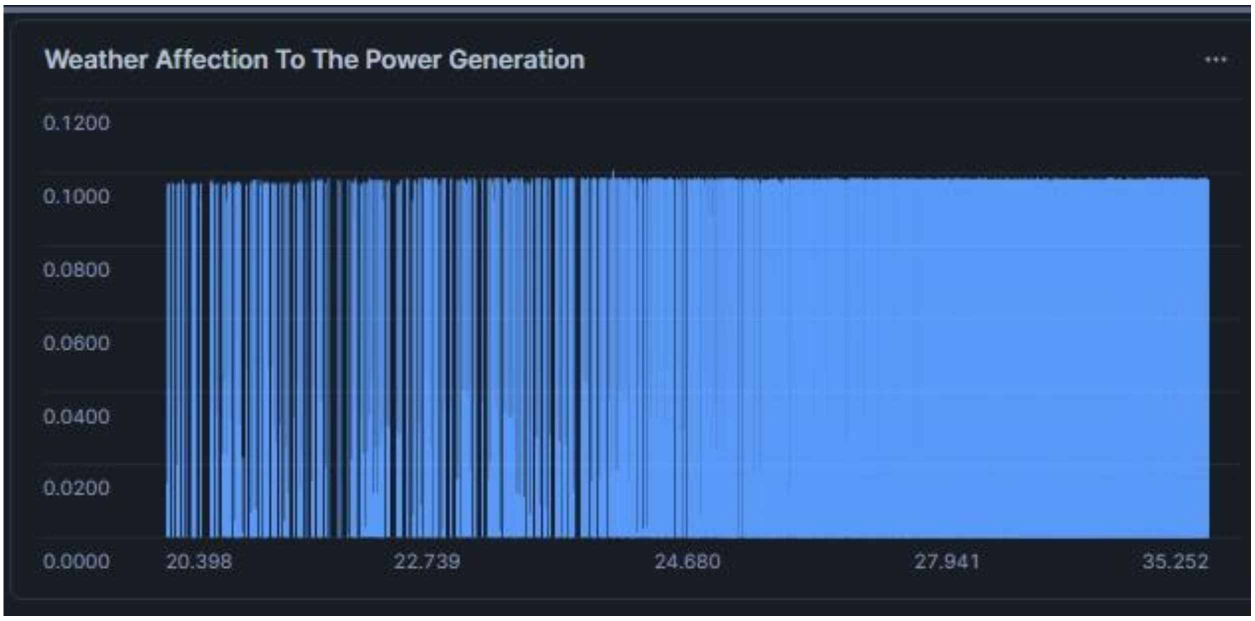


Figure 22: Weather Impact on Power Generation Visualization

The following query was run in order to see how ambient temperature affected electricity efficiency:

```
SELECT AMBIENT_TEMPERATURE,
      AVG("Power_Efficiency") AS Avg_Eff
FROM raw_power_data
GROUP BY AMBIENT_TEMPERATURE
ORDER BY AMBIENT_TEMPERATURE;
```

Results:

- The ambient temperature affects power efficiency.
- Module overheating or decreased solar absorption can result from extremely high or low temperatures.
- Planning cooling techniques and optimizing module placement are made easier by these findings.

4.4 Snowflake’s Unique Features Utilized

Time Travel

Snowflake's moment Travel feature enables data querying at any moment. This comes in handy in the following ways:

1. Data recovers in case of unintentional deletion.
2. Tracking changes over time for further analysis.
3. Comparison of the historical and current trends of data.

Example:

```
SELECT * FROM raw_power_data AT (TIMESTAMP => '2024-03-20 12:00:00');
```

Query Optimization

Snowflake enhances querying by:

1. Auto-clustering for quicker execution of queries.
2. Caching the metadata to reduce repeated query times.
   Example of optimized query execution:

```
SELECT AVG(DC_POWER), AVG(AC_POWER)
FROM raw_power_data
WHERE DATE_TIME > CURRENT_DATE - INTERVAL '7 DAYS';
```

Zero-Copy Cloning

This feature enables creating full copies of a dataset without using additional storage.

```
CREATE OR REPLACE TABLE cloned_power_data CLONE raw_power_data;
```

Data Sharing Capabilities

Snowflake allows seamless data sharing between different accounts without data duplication.

```
GRANT SELECT ON raw_power_data TO SHARE power_data_share;
```

Chapter 5 Predictive Analytics with Spark MLlib

5.1 Introduction

In big data applications, predictive analytics is crucial because it helps researchers and businesses detect patterns, predict future trends, and make data-driven decisions. In this study, we leverage the scalable machine learning library Apache Spark MLlib to build prediction models to analyse efficiency in a solar power plant.

Why Spark MLlib for Machine Learning?

Apache Spark MLlib offers several advantages for big data machine learning tasks, including:

- Scalability: Efficient handling of large-scale datasets.
- Ease of Use: Built-in tools and packages for feature transformation, model training, and evaluation.
- Distributed Processing: speeds up computations across multiple nodes.

Justification for the Predictive Task

Given the dataset's characteristics, we implemented both regression and classification models to forecast solar power efficiency.

- Regression: Predicting DC power output utilising environmental variables.
- Classification: Using a Gradient Boosting Classifier, evaluate if the power output is High (1) or Low (0).

5.2 Methodology

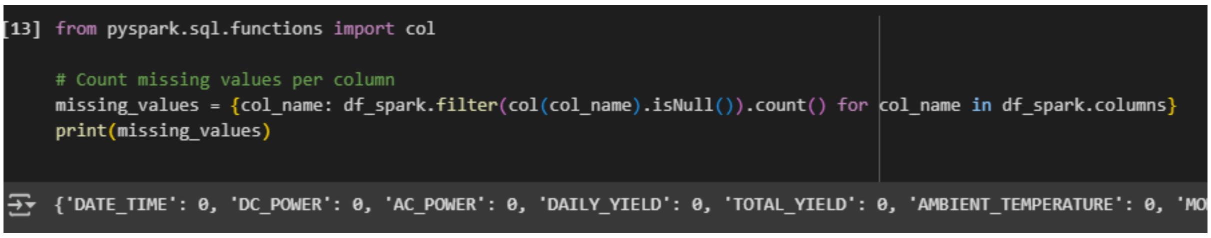
This section demonstrates the step-by-step execution of our predictive modeling approach using Pyspark MLlib.

5.2.1 Data Preparation

We used a cleaned solar power plant dataset that included essential features including DC_POWER, AC_POWER, IRRADIATION, MODULE_TEMPERATURE, and AMBIENT_TEMPERATURE.

Preprocessing steps:

- Handled missing values to ensure data consistency.
- DATE_TIME was converted to a suitable format for time series analysis.
- Generated new feature columns as required

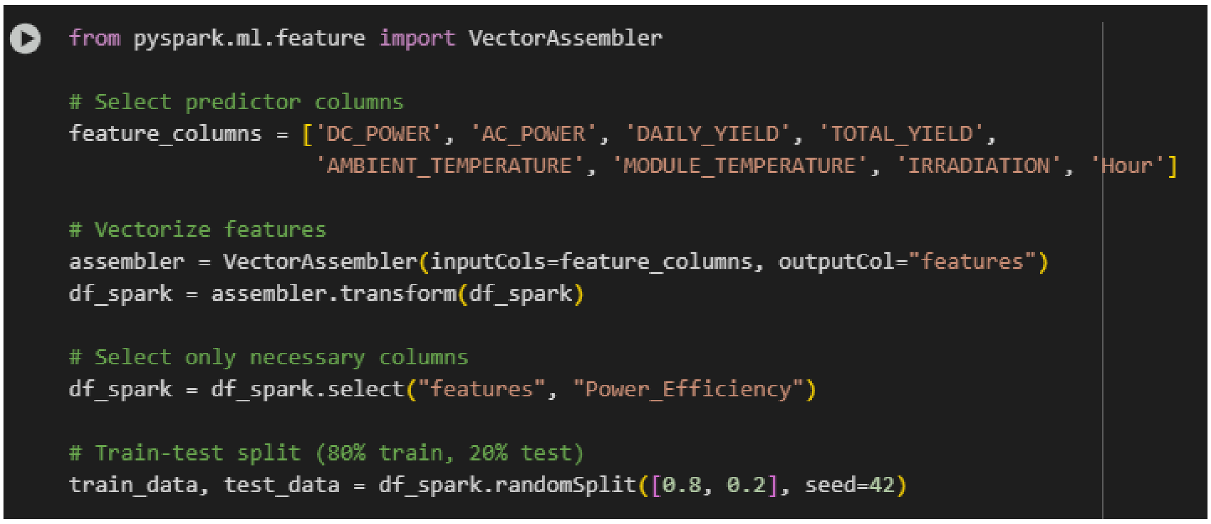


```
[13] from pyspark.sql.functions import col

     # Count missing values per column
     missing_values = {col_name: df_spark.filter(col(col_name).isNull()).count() for col_name in df_spark.columns}
     print(missing_values)

     {'DATE_TIME': 0, 'DC_POWER': 0, 'AC_POWER': 0, 'DAILY_YIELD': 0, 'TOTAL_YIELD': 0, 'AMBIENT_TEMPERATURE': 0, 'MO
```

Figure 23: Missing Value Check

Splitting the dataset:

- Train-Test Split: 80% of the dataset was used for training, while 20% for testing.
- Feature Engineering: Utilised VectorAssembler to generate feature vectors.



```
from pyspark.ml.feature import VectorAssembler

# Select predictor columns
feature_columns = ['DC_POWER', 'AC_POWER', 'DAILY_YIELD', 'TOTAL_YIELD',
                   'AMBIENT_TEMPERATURE', 'MODULE_TEMPERATURE', 'IRRADIATION', 'Hour']

# Vectorize features
assembler = VectorAssembler(inputCols=feature_columns, outputCol="features")
df_spark = assembler.transform(df_spark)

# Select only necessary columns
df_spark = df_spark.select("features", "Power_Efficiency")

# Train-test split (80% train, 20% test)
train_data, test_data = df_spark.randomSplit([0.8, 0.2], seed=42)
```

Figure 24: Feature Engineering and Train-Test Split

5.2.2 Model Selection & Training

In this section, we outline the four predictive models we built using PySpark MLlib and other related tools, as well as the training process for each.

1. **Predictive Model using Regression**

The main objective of selecting the regression model is to predict power efficiency based on solar and environmental factors. It was chosen based its nature of predicting continuous values like power efficiency. This model was selected to estimate power efficiency as a continuous value.

The following code snippet demonstrates how the regression model is used for training in this scenario.

```
from pyspark.ml.regression import LinearRegression

# Initialize model
lr = LinearRegression(featuresCol="features",
labelCol="Power_Efficiency")

# Train the model
lr_model = lr.fit(train_data)

# Make predictions
predictions = lr_model.transform(test_data)

# Show predictions
predictions.select("features", "Power_Efficiency", "prediction").show(5)

from pyspark.ml.evaluation import RegressionEvaluator

# Evaluate model performance
evaluator_rmse = RegressionEvaluator(labelCol="Power_Efficiency",
metricName="rmse")
evaluator_r2 = RegressionEvaluator(labelCol="Power_Efficiency",
metricName="r2")
```

Figure 25: Linear Regression Model Training

The Linear Regression model was trained as above to predict power efficiency using the features and label columns. The model's performance was evaluated using RMSE and R^2, which analyses prediction accuracy and model fit.

2. Predictive Model using Random Forest Regressor

With the accuracy limitations we experienced in the linear regression model, we explored a Random Forest Regressor for better performance. This algorithm is well suited for capturing nonlinear relationships and capable of providing feature important analysis which can be strongly aligned with our prediction task. We have implemented 50 tress, capturing nonlinear dependencies with minimal error.

The following code snippet demonstrates how the Random Forest Regressor model is used for training in this scenario.

```
from pyspark.ml.regression import RandomForestRegressor

rf = RandomForestRegressor(featuresCol="features",
labelCol="Power_Efficiency", numTrees=50)
rf_model = rf.fit(train_data)
predictions_rf = rf_model.transform(test_data)

rf_rmse = evaluator_rmse.evaluate(predictions_rf)
rf_r2 = evaluator_r2.evaluate(predictions_rf)

print(f"Random Forest RMSE: {rf_rmse}")
print(f"Random Forest R²: {rf_r2}")

import pandas as pd

feature_importance = rf_model.featureImportances.toArray()
feature_names = feature_columns

# Create DataFrame
importance_df = pd.DataFrame({'Feature': feature_names,
'Importance': feature_importance})
importance_df = importance_df.sort_values(by="Importance",
ascending=False)

print(importance_df)
```

Figure 26: Random Forest Regressor Model Training

3. ARIMA Forecasting Model

We used an ARIMA model to anticipate DC power based on historical data and external factors such as irradiation, ambient temperature, and module temperature. ARIMA works well for time-series forecasting as it detects trends and seasonality in data. We utilised Auto ARIMA to select the best model parameters, trained the model, and assessed its performance using RMSE. The model's predictions were visualized against actual values to assess its accuracy, which had an RMSE value of 602.95.

The following code snippet demonstrates how the ARIMA model is used for time-series forecasting with external factors.

```
from pyspark.sql import SparkSession
import pandas as pd
from statsmodels.tsa.arima.model import ARIMA
from pmdarima import auto_arima
from sklearn.metrics import mean_squared_error
import numpy as np

# Initialize Spark session
spark =
SparkSession.builder.appName("ARIMA_Forecast").getOrCreate()

# Load dataset into Spark DataFrame and convert to Pandas for ARIMA
df_spark = spark.read.csv("/content/Cleaned_Plant_Data.csv",
header=True, inferSchema=True)
df_pandas = df_spark.select("DATE_TIME", "DC_POWER",
"IRRADIATION", "AMBIENT_TEMPERATURE",
"MODULE_TEMPERATURE").toPandas()

# Convert DATE_TIME to datetime format and set as index
df_pandas["DATE_TIME"] = pd.to_datetime(df_pandas["DATE_TIME"])
df_pandas.set_index("DATE_TIME", inplace=True)

# Split data into training and testing sets
train_size = int(len(df_pandas) * 0.8)
train, test = df_pandas[:train_size], df_pandas[train_size:]
```

Figure 27: ARIMA Forecasting Model Snippet 1 - Actual vs Predicted DC Power

```
# Auto ARIMA to find optimal (p, d, q) values with external variables
auto_model = auto_arima(train["DC_POWER"],
exogenous=train[["IRRADIATION", "AMBIENT_TEMPERATURE",
"MODULE_TEMPERATURE"]], seasonal=False, stepwise=True,
suppress_warnings=True)
p, d, q = auto_model.order

# Train ARIMA model with optimal parameters and exogenous variables
model = ARIMA(train["DC_POWER"], exog=train[["IRRADIATION",
"AMBIENT_TEMPERATURE", "MODULE_TEMPERATURE"]],
order=(p, d, q))
model_fit = model.fit()

# Make predictions and calculate RMSE
predictions = model_fit.forecast(steps=len(test),
exog=test[["IRRADIATION", "AMBIENT_TEMPERATURE",
"MODULE_TEMPERATURE"]])
rmse = np.sqrt(mean_squared_error(test["DC_POWER"], predictions))

# this code had RMSE: 602.9506837885813
```

Figure 28: ARIMA Forecasting Model Snippet 2 - Actual vs Predicted DC Power

In this section, we used the ARIMA model for time-series forecasting using external factors such as irradiation, ambient temperature, and module temperature. It was mainly trained based on the historical DC power data. After splitting the data, Auto ARIMA was used to determine the optimal (p, d, q) values. The model's performance was computed using the RMSE, which was 602.95, and predictions were compared to actual values to determine accuracy.

4. Gradient Boosting Classifier

After experimenting with previous models, we used Gradient Boosting to determine if the DC power output was high (1) or low (0). The model performs well with binary classification tasks and large datasets. We selected this approach to maximise efficiency in classifying power output levels by using the median value of DC power as a threshold and it was trained with 50 iterations, achieving high accuracy. The model's performance was tested accurately, and the final model delivered strong classification results.

This code snippet shows the training process for the Gradient Boosting Classifier used to predict whether DC power output is high (1) or low (0), with the model's performance evaluated based on accuracy.

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import GBTClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# Convert DC_POWER into a Binary Classification Label (High=1,
Low=0)
threshold = df.approxQuantile("DC_POWER", [0.5], 0.01)[0]  # Median
value as threshold
df = df.withColumn("label", when(col("DC_POWER") > threshold,
1).otherwise(0))

# Prepare Features
assembler = VectorAssembler(inputCols=["IRRADIATION",
"AMBIENT_TEMPERATURE", "MODULE_TEMPERATURE"],
outputCol="features")
df = assembler.transform(df).select("features", "label")

# Split Data (80% Train, 20% Test)
train, test = df.randomSplit([0.8, 0.2], seed=42)

# Define Gradient Boosted Classifier
gbt = GBTClassifier(labelCol="label", featuresCol="features",
maxIter=50)

# Train the Model
model = gbt.fit(train)

# Predictions
predictions = model.transform(test)

# Evaluate Model Performance
evaluator = MulticlassClassificationEvaluator(labelCol="label",
predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print(f"Model Accuracy: {accuracy:.4f}")
```

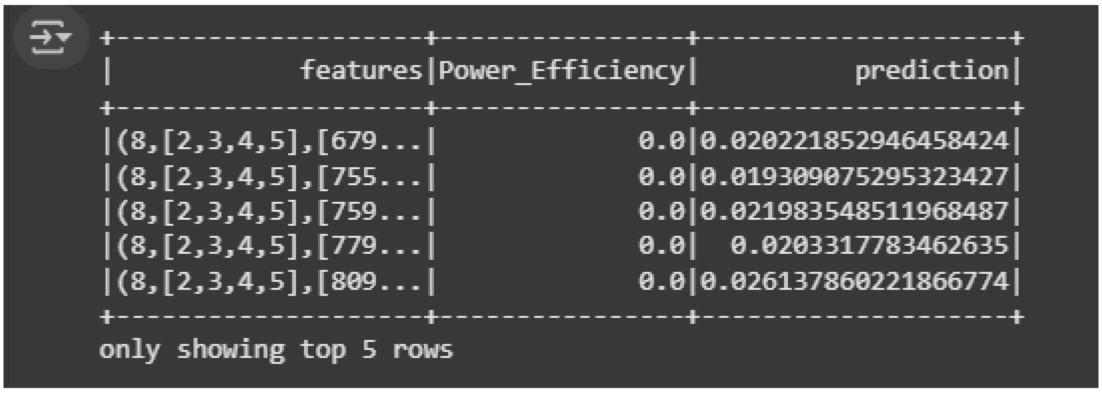
Figure 29: Gradient Boosting Classifier Model Training

5.3 Results & Evaluation

This section focusses on the outcomes and evaluations of our predictive models, detailing their performance in forecasting solar power output and classifying efficiency.

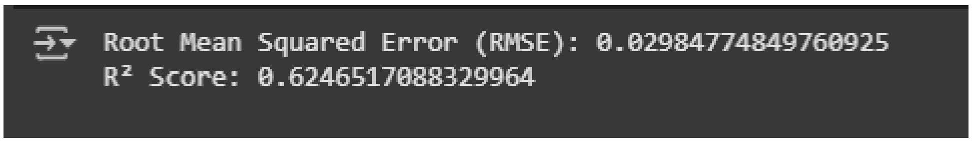
1. **Linear Regression Model**

RMSE: 0.0298 (Moderate Error)
R² Score: 0.624 (Explains 62.4% of variance)



```
+--------------------+----------------+--------------------+
|            features|Power_Efficiency|          prediction|
+--------------------+----------------+--------------------+
|(8,[2,3,4,5],[679...|             0.0|0.020221852946458424|
|(8,[2,3,4,5],[755...|             0.0|0.019309075295323427|
|(8,[2,3,4,5],[759...|             0.0|0.021983548511968487|
|(8,[2,3,4,5],[779...|             0.0|  0.0203317783462635|
|(8,[2,3,4,5],[809...|             0.0|0.026137860221866774|
+--------------------+----------------+--------------------+
only showing top 5 rows
```

Figure 30: Linear Regression Predictions



```
Root Mean Squared Error (RMSE): 0.029847748497 60925
R² Score: 0.6246517088329964
```

Figure 31: Linear Regression Performance Metrics

Performance Observations:

- A basic regression model was used to predict power efficiency.
- Although the RMSE was low, the R² score indicated that the model did not capture all key influencing factors.
- Showed anomalies in predicting 0 values, when the actual Power Efficiency was 0, but the model predicted small non-zero values.
- Due to its shortcomings in effectively modelling complex relationships, alternative models were explored.

2. **Random Forest Regressor**

RMSE: 9.07e-05 (Extremely Low Error)
R² Score: 0.9999965 (Near-Perfect Fit)

Feature Importance:

- DC_POWER (39.01%) – Most impactful

- AC_POWER (31.67%) – Highly relevant
- IRRADIATION (15.38%) – Strong influence
- MODULE_TEMPERATURE (10.50%) – Moderate impact
- AMBIENT_TEMPERATURE (2.55%) – Minor impact
- DAILY_YIELD (0.83%) – Low importance
- Hour (0.05%) – Almost negligible
- TOTAL_YIELD (≈0%) – Insignificant

Performance Observations:

- Exceptional Model Fit: The model demonstrated an extremely low error rate (RMSE = 0.00009) and almost ideal R^2 score (0.99999), indicating high prediction accuracy.

- Key Influential Factors Identified: The top three features were DC_POWER, AC_POWER, and IRRADIATION—are the most key indicators of power efficiency, as per domain knowledge.

- Feature Reduction Insight: TOTAL_YIELD contributes almost nothing to forecast accuracy, therefore eliminating it may increase model efficiency.

- Need for Model Validation: Although the Random Forest model functioned effectively, we explored additional models to validate the results and reduce potential overfitting.

3. ARIMA Forecasting Model

Optimal ARIMA Order: (5,0,2)
RMSE: 602.95 (Higher than desired)

A time series analysis of the ARIMA forecast was conducted, and the findings are visualized in Figure 28. This plot compares actual DC_POWER values to the model's predictions across the test period, demonstrating the model's ability to identify overall patterns while also emphasizing the degree of error reflected by the RMSE.

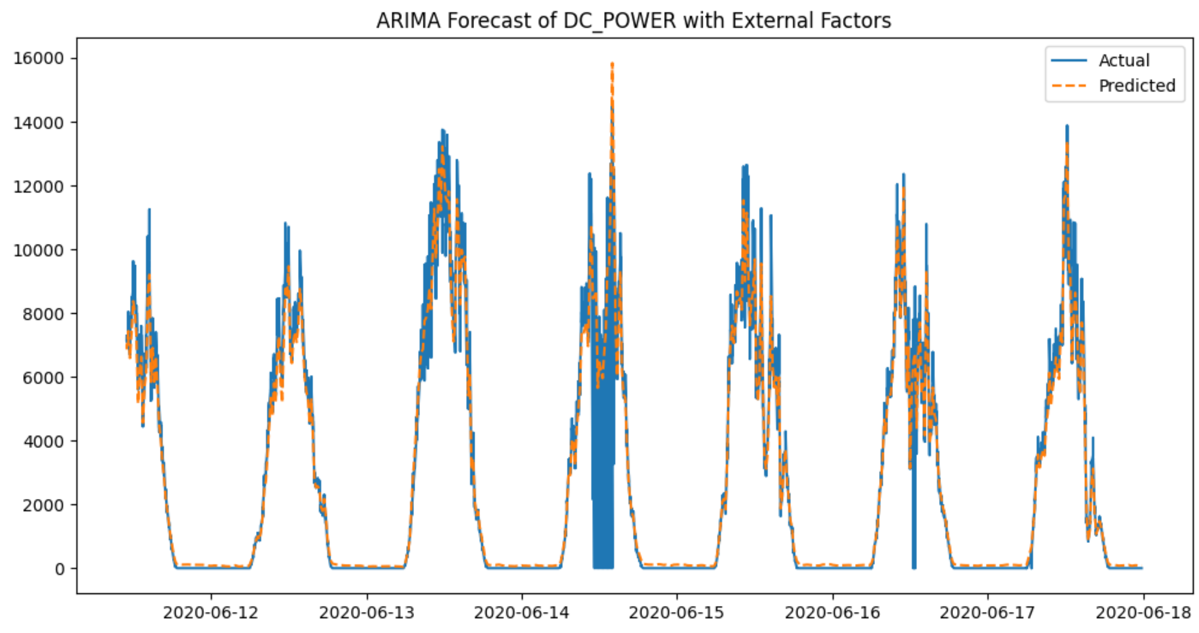


Figure 32: ARIMA Forecast of DC_POWER with External Factors

Challenges & Optimization Attempts:

- Initial models had higher RMSE values.
- Added **IRRADIATION** and **TEMPERATURE** as exogenous variables to improve forecasting accuracy.
- Despite improvements, the RMSE remained relatively high, showing limitations in the time-series forecasting for power efficiency.

Performance Observations:

- The model was effective in capturing trends but struggled with precise accuracy.
- Despite efforts to optimize and add new features, the RMSE increased instead of decreasing.
- Despite efforts to optimize and add new features, the RMSE has increased instead of decreasing.

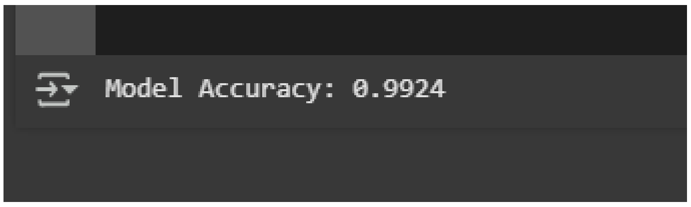
4. **Gradient Boosting Classifier**

Model Accuracy: 99.24%

The high accuracy exhibits that the model is very effective at predicting binary outcomes (high or low power output). A classification model with this level of accuracy performs well, particularly on binary classification tasks.

Performance Observations:

- Successful Classification: The model accurately classified power output as either high (1) or low (0), which is consistent with the objective of the task.
- Feature Importance: Feature selection was crucial in enhancing accuracy, and only three features including IRRADIATION, AMBIENT_TEMPERATURE, and MODULE_TEMPERATURE were included in the final model, simplifying it while maintaining excellent performance.
- Best Trade-Off: Among the models examined, the Gradient Boosting Classifier achieved the best balance of accuracy and interpretability, delivering outstanding results while also providing insights into the decision-making process.



```
Model Accuracy: 0.9924
```

Figure 33: Gradient Boosting Classifier - Model Accuracy

5.4 Discussion & Challenges

Observations on Model Results

- **Linear Regression:** It Provided the fundamental approach, but experienced anomalies predicting 0 values and lacked predictive power for non-linear relationships.
- **Random Forest** It achieved near-perfect accuracy but was computationally expensive.
- **ARIMA:** ARIMA model exhibited strong time-series trend analysis but had a larger RMSE despite feature engineering.
- **Gradient Boosting:** It emerged as the best model for classification, with the highest efficiency.

Challenges Encountered

- **Feature Selection:** It took multiple iterations to identify the most crucial features.
- **ARIMA Performance:** Despite optimization, the RMSE remained relatively high.
- **Computational Complexity:** Training models with Spark MLlib demanded careful resource management.

Possible Improvements

- **Hyperparameter Tuning:** Further fine-tuning, notably for ARIMA and Gradient Boosting, could improve performance.
- **Ensemble Learning:** Combining several models may result in better generalization
- **Advanced Feature Engineering:** Developing new derived features may improve prediction accuracy.

Chapter 6 Discussion and Critical Analysis

6.1 Performance Evaluation Across Tasks

Comparison of Spark and Snowflake Performance Apache Spark and Snowflake are two powerful big data platforms, each with its strengths in handling large-scale data processing and analytics. In this study, we primarily utilized Apache Spark MLlib for predictive modeling due to its scalability and efficient distributed processing capabilities. However, a comparative analysis with Snowflake which is a cloud-based data warehouse provides further insights into their performance trade-offs.

Key Differences in Performance:

- Processing Speed: Spark excels in handling massive datasets by leveraging in-memory computing, making it ideal for iterative machine learning tasks. Snowflake, in contrast, optimizes performance through automatic scaling and query optimization but lacks built-in machine learning capabilities.
- Ease of Integration: Spark seamlessly integrates with diverse big data ecosystems such as Hadoop and Kafka, whereas Snowflake offers better native integration with cloud storage and BI tools.
- Scalability & Cost: Snowflake provides automatic scaling and cost optimization by separating storage and compute resources, while Spark requires manual cluster tuning for optimal efficiency.

Trade-offs in Model Execution:

- Spark provided faster iterative training for machine learning models, making it suitable for real-time analytics and predictive tasks.
- Snowflake, while powerful for structured data queries and cloud analytics, required additional steps to integrate machine learning workflows, often relying on external libraries.
- Given our focus on predictive analytics, Spark MLlib was the preferred choice for building models due to its efficient handling of feature transformations, model training, and evaluation.

Key Findings from Data Preprocessing, Real-Time Analytics, Cloud Analytics, and Predictive Modeling

1. Data Preprocessing
    - Feature Engineering: Key environmental and power-related features (e.g., Irradiation, Module Temperature, DC Power) were selected for modeling.
    - Data Cleaning: Missing values were handled, and date-time formatting was optimized for time-series forecasting.
    - Vectorization: Feature vectors were efficiently generated using Spark's VectorAssembler, ensuring optimal input for machine learning models.

2. Real-Time Analytics
    - Spark's distributed processing enabled rapid data ingestion and transformation, facilitating near real-time performance monitoring of solar power efficiency.
    - Implementing window functions improved time-series analysis and trend detection for solar power fluctuations.

3. Cloud Analytics
    - Snowflake was considered for data warehousing and reporting, offering a centralized, scalable cloud storage option.
    - While Spark managed on-the-fly analytics, Snowflake provided structured historical data analysis, enabling a hybrid approach for long-term insights.

4. Predictive Modeling
    - Regression models (Linear Regression, Random Forest Regressor) demonstrated varying levels of accuracy, with Random Forest achieving the best fit ($R^2 = 0.99999$).
    - Time-Series Forecasting (ARIMA) effectively captured trends but struggled with precision, with a higher-than-expected RMSE.
    - Gradient Boosting Classifier (99.24% accuracy) emerged as the best model for binary classification, efficiently determining high vs. low power output.

Overall, Spark MLlib provided the best balance of scalability, speed, and machine learning support, making it the optimal choice for our predictive analytics framework.

6.2 Challenges Encountered

Implementing Apache Spark and Snowflake for large-scale data processing and predictive analytics presented several challenges. These issues primarily arose from differences in data handling, performance bottlenecks, and platform-specific constraints.

Issues Faced in Spark vs. Snowflake Implementations

1. **Apache Spark Challenges**

- **Memory Management & Resource Allocation**
    - Spark's in-memory computation accelerates processing but requires careful resource management.
    - Inefficient memory allocation led to out-of-memory errors when handling large datasets.
    - The need for manual tuning of Spark configurations (e.g., executor.memory, driver.memory, parallelism) increased complexity.
- **Latency in Real-Time Analytics**
    - While Spark Streaming was leveraged for near real-time data processing, occasional latency spikes were observed due to high network I/O when ingesting large volumes of solar power data.
    - Stateful streaming operations (e.g., window functions) increased memory consumption, requiring checkpointing to avoid failures.
- **MLlib Algorithm Limitations**
    - Although Spark MLlib supports machine learning models, it lacks advanced deep learning capabilities compared to dedicated ML frameworks like TensorFlow or PyTorch.
    - Some model training processes were slower compared to GPU-accelerated environments.

2. **Snowflake Challenges**

- **Performance Trade-offs in Query Execution**
    - Snowflake's automatic query optimization improves speed but can lead to unexpected query costs when handling large datasets.
    - Joins and aggregations on high-volume time-series data (solar power metrics) occasionally exceeded expected execution times, requiring indexing and clustering strategies.
- **Limited Built-in Machine Learning Support**

    - Snowflake excels in structured cloud analytics but lacks native ML capabilities, requiring integration with external platforms (e.g., Snowpark for Python, AWS SageMaker).
    - Data export overhead increased latency when shifting between Snowflake and external ML environments.
- **Storage & Cost Considerations**
    - Snowflake charges based on compute and storage usage, and frequent complex queries led to higher costs.
    - Data warehousing was effective for historical analysis but less efficient for real-time ML workflows.

Bottlenecks in Data Processing and Analytics

1. **High Computational Overhead**
    - Large-scale data ingestion and transformation in Spark resulted in increased processing time and memory usage.
    - Running iterative machine learning models (e.g., Random Forest, Gradient Boosting) required significant computation power.
2. **Handling Missing & Noisy Data**
    - Large volumes of sensor data from solar panels contained missing values and inconsistencies.
    - Data preprocessing (imputation, normalization) added additional computational overhead.
3. **Parallelism vs. Scalability**
    - While Spark efficiently distributes computation, inefficient partitioning led to data skew, affecting parallel execution efficiency.
    - Snowflake's scaling model worked well for structured analytics but struggled with real-time prediction workloads.
4. **Latency in Model Deployment**
    - Running real-time ML predictions in Spark required additional optimizations (e.g., caching intermediate results).
    - Snowflake's reliance on external ML pipelines introduced additional latency in model inference.

Summary of Key Challenges & Mitigation Strategies

Challenge	Platform	Mitigation Strategy
Memory Overhead & Tuning	Spark	Optimized execution memory, used caching, and parallelized operations
High Latency in Real-Time Analytics	Spark	Used checkpointing and optimized streaming queries
Slow Query Execution for Large Joins	Snowflake	Implemented clustering keys, query pruning, and indexing
ML Model Deployment Complexity	Snowflake	Used Snowpark & external ML tools for integration
High Data Transfer Costs	Snowflake	Optimized data storage and reduced frequent exports

Despite these challenges, Spark proved to be more suitable for large-scale ML processing, while Snowflake was ideal for structured data analysis and historical trends. Adopting a hybrid approach, leveraging both platforms for their strengths, helped optimize performance
.

6.3 Recommendations for Improvement

To enhance the efficiency of data processing, model performance, and infrastructure utilization, several optimizations can be implemented. These improvements focus on workflow optimization, data handling enhancements, and infrastructure upgrades to overcome bottlenecks observed in Spark and Snowflake.

1. Workflow Optimization

Optimizing the end-to-end data pipeline can significantly improve performance, reduce latency, and lower costs.

1.1 Hybrid Processing Approach

- Use Snowflake for structured data analytics and historical trend analysis while leveraging Apache Spark for machine learning (ML) workloads that require intensive computations.
- Implement ETL (Extract, Transform, Load) best practices to preprocess data in Snowflake before feeding it into Spark-based ML models.

1.2 Optimized Query Execution in Snowflake

- Utilize materialized views and clustering keys to optimize query performance and reduce execution time.
- Implement query pruning techniques to fetch only the necessary data instead of scanning large tables.

- Use **warehouse scaling policies** to dynamically allocate resources based on workload intensity.

1.3 Enhanced Caching and Data Partitioning in Spark

- Leverage in-memory caching for frequently accessed datasets to reduce redundant computations.
- Optimize data partitioning by using broadcast joins and coalesce functions to reduce data shuffling.
- Tune Spark configurations (spark.sql.shuffle.partitions, executor.memory, parallelism) for better workload distribution.

2. Potential Improvements in Data Handling

2.1 Efficient Data Preprocessing

- Implement adaptive data cleaning techniques using automated anomaly detection methods to handle missing values in sensor readings.
- Use data deduplication and compression to minimize storage overhead and improve query speed.

2.2 Incremental Data Processing

- Instead of batch processing, adopt micro-batch or real-time streaming where applicable to improve response time.
- Use Delta Lake (for Spark) or Time Travel (for Snowflake) for managing historical data versions without excessive duplication.

2.3 Improved Feature Engineering for ML Models

- Utilize feature stores to centrally manage and reuse ML features across different models.
- Apply dimensionality reduction (e.g., PCA, feature selection) to improve ML model efficiency.

3. Model Selection and Training Improvements

3.1 Advanced Model Optimization

- Fine-tune hyperparameters using automated optimization techniques such as Grid Search, Random Search, or Bayesian Optimization.

- Utilize pre-trained deep learning models for complex predictive tasks instead of training models from scratch.

3.2 Distributed Model Training

- Use Apache Spark MLlib or TensorFlow Distributed Training to handle large-scale machine learning tasks efficiently.
- Implement GPU acceleration for deep learning models to reduce training time.

3.3 Model Deployment Optimization

- Deploy models as low-latency REST APIs using FastAPI or Flask, integrated with Snowpark for real-time inference.
- Utilize model caching and quantization techniques to reduce inference time and computation overhead.

4. Infrastructure Enhancements

4.1 Cloud Resource Optimization

- Implement auto-scaling in Snowflake warehouses to dynamically allocate computing power based on workload.
- Utilize serverless execution models (e.g., AWS Lambda, Google Cloud Functions) to reduce infrastructure costs.
- Use spot instances or reserved instances for cost-effective cloud resource allocation.

4.2 Storage Optimization

- Implement tiered storage solutions by storing frequently accessed data in high-performance storage (e.g., Snowflake) and archiving older data in cost-effective storage (e.g., AWS S3).
- Optimize parquet file formats and columnar storage to enhance data retrieval speed.

5. Summary of Key Recommendations

Category	**Improvement**	**Expected Benefit**
Workflow Optimization	Hybrid approach (Snowflake for analytics, Spark for ML)	Faster processing, better workload balance
Query Execution	Clustering keys, query pruning	Reduced query execution time in Snowflake
Data Handling	Incremental processing, feature engineering	Improved model accuracy and efficiency
Model Performance	Hyperparameter tuning, distributed training	Faster training with improved predictions
Infrastructure	Cloud auto-scaling, tiered storage	Cost savings and better resource allocation

By implementing these workflow, data, model, and infrastructure optimizations, the overall performance of the Spark-Snowflake pipeline can be significantly improved.

References

Apache Spark. (2025, February 27). *Cluster Mode Overview.* Retrieved from Apache Spark 3.5.5: https://spark.apache.org/docs/latest/cluster-overview.html#components

Snowflake. (2025, March 10). *Key Concepts & Architecture.* Retrieved from Snowflake Documentation: https://docs.snowflake.com/en/user-guide/intro-key-concepts#snowflake-architecture

Appendix

Appendix A: Data Preprocessing with Apache Spark & Pandas

1. Apache Spark Implementation:

   PUSL3121_Data_Preprocessing_Spark.ipynb

2. Python Pandas Implementation

   PUSL3121_Data_Preprocessing_pandas.ipynb

Appendix B: Real-Time & Cloud Analytics

1. Cloud Analytics with Snowflake:

   Snowflake_Streaming_Code.ipynb

Appendix C: Predictive Analytics with Spark MLlib

Apache_Spark.ipynb

Appendix D: Source Code Folder

PUSL3121 - Source Codes