CLOUD COMPUTING PROJECT REPORT - GROUP 10

- **Group Members :** Rakesh (801337430) , Deepiga (801370629), Sai Pavan (801377971)
- **Characteristics of Host** : Task is hosted on MAC M2 with 16 GB RAM with Latest MacOS.
- **Cluster Configuration** :
    - Cluster consisted of **1 namenode, 1 datanodes, 1 nodemanagers, 1 ResourceManager and 1 historyserver**.

Summary

Security is off.

Safemode is off.

26 files and directories, 19 blocks (19 replicated blocks, 0 erasure coded block groups) = 45 total filesystem object(s).

Heap Memory used 246.99 MB of 589.5 MB Heap Memory. Max Heap Memory is 3.56 GB.

Non Heap Memory used 68.89 MB of 70.44 MB Commited Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	228.27 GB
Configured Remote Capacity:	0 B
DFS Used:	1.26 GB (0.55%)
Non DFS Used:	191.76 GB
DFS Remaining:	35.25 GB (15.44%)
Block Pool Used:	1.26 GB (0.55%)
DataNodes usages% (Min/Median/Max/stdDev):	0.55% / 0.55% / 0.55% / 0.00%
Live Nodes	1 (Decommissioned: 0, In Maintenance: 0)
Dead Nodes	0 (Decommissioned: 0, In Maintenance: 0)
Decommissioning Nodes	0
Entering Maintenance Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	0
Number of Blocks Pending Deletion (including replicas)	0
Block Deletion Start Time	Tue Feb 27 07:47:01 -0500 2024
Last Checkpoint Time	Tue Feb 27 07:47:02 -0500 2024
Enabled Erasure Coding Policies	RS-6-3-1024k

- **Data** :
    - For this project we have considered data from 1910 - 1935 due to storage constraints and computation constraints.

- **Step 1 - Quality of Data :**
  - **Filtering and Augmenting data** :

  Our data looks like this **before** this step. It is spread across 25 files based on year.

```
AGE00135039,19100101,TMAX,150,,,E,
AGE00135039,19100101,TMIN,50,,,E,
AGE00135039,19100101,PRCP,0,,,E,
AGE00147704,19100101,TMAX,140,,,E,
AGE00147704,19100101,TMIN,100,,,E,
AGE00147704,19100101,PRCP,82,,,E,
AGE00147705,19100101,TMAX,180,,,E,
```

We need to process it according to the question.

In question it is mentioned that we need to use data for North America so we are also filtering the data for US, CA AND MX so that we will only process relevant data for upcoming steps.

After this we are pivoting the data so that all the datapoints for same StationID and Date are in same line.

Then we are calculating the TAVG based on TMIN AND TMAX.

```python


from pyspark.sql import SparkSession
from pyspark.sql.functions import *

spark = SparkSession.builder.appName("Q1").getOrCreate()

country_codes = spark.read.text("country_codes.txt")
country_codes = country_codes.withColumn('country_code', split(col('value'), ' ')[0])\
                              .withColumn('country_name', concat_ws(' ', slice(split(col('value'), ' '), 2, 999)))\
                              .drop('value')

df = spark.read.csv("Dataset/", header=False)

#rename the columns
df = df.withColumnRenamed('_c0', 'Station ID')\
       .withColumnRenamed('_c1', 'Date')\
       .withColumnRenamed('_c2', 'Type')\
       .withColumnRenamed('_c3', 'Value1')\
       .withColumnRenamed('_c4', 'Value2')\
       .withColumnRenamed('_c5', 'Value3')\
       .withColumnRenamed('_c6', 'Value4')\
       .withColumnRenamed('_c7', 'Value5')

#ignore the columns Value1, Value2, Value3

df = df.drop('Value2', 'Value3', 'Value4', 'Value5')

#filer the data based on the first two letters of the Station ID which represent the country code and the country code should be US, CA or MX
df = df.filter(df['Station ID'].substr(1, 2).isin('US', 'CA', 'MX'))
# Add country name to this dataframe by joining with the country_codes dataframe
df = df.join(country_codes, df['Station ID'].substr(1, 2) == country_codes['country_code'], 'inner')

#pivot the data based on the Type column
df = df.groupBy('Station ID', 'country_name', 'Date').pivot('Type').agg(first('Value1'))

#calculate TAVG
df = df.withColumn('TAVG', (df['TMAX'] + df['TMIN'])/2)

#select the required columns

df = df.select('Station ID', 'country_name', 'Date', 'TMIN', 'TMAX','TAVG', 'PRCP')

#sort the data based on Station ID and Date
df = df.orderBy('Station ID', 'Date')

df.printSchema()

df.show()

#save the dataframe as a single csv file
df.coalesce(1).write.csv("output", header=True)

spark.stop()

```

After this step our data looks like this below given screenshot where TAVG is calculated for each station according to TMIN AND TMAX and the data is sorted based on the station number. PRCP is also included in the same row itself.

```csv
Station ID,country_name,Date,TMIN,TMAX,TAVG,PRCP
USW00094967,United States,19351112,-139,-28,-83.5,0
USW00094967,United States,19351113,-56,0,-28.0,0
USW00094967,United States,19351114,-94,0,-47.0,3
USW00094967,United States,19351115,-161,-39,-100.0,0
USW00094967,United States,19351116,-56,6,-25.0,0
USW00094967,United States,19351117,-11,17,3.0,0
USW00094967,United States,19351118,-11,17,3.0,10
USW00094967,United States,19351119,-139,0,-69.5,5
USW00094967,United States,19351120,-122,-56,-89.0,0
```

- **Handling missing or incomplete data**: Based on the question if the dataset has None values for some values of temperature or precipitation data for more than 10 consecutive days, they have been removed from the dataset. Dataset from previous step has been given to this step as input. All the remaining null values are filled 0.

```python

from pyspark.sql import SparkSession
from pyspark.sql import SQLContext
from pyspark.sql.functions import *
from pyspark.sql.window import Window
from pyspark.sql.types import *
import sys

spark = SparkSession.builder.appName("q1_part2")\
    .config("spark.executor.memory", "4g") \
    .config("spark.driver.memory", "4g") \
    .getOrCreate()
sc = spark.sparkContext
sqlContext = SQLContext(sc)

#read the input file
df = spark.read.csv("q1_part1_output.csv", header=True, inferSchema=True)


# exclude from the dataset stations that have missing or incomplete temperature or precipitation data for more than 10 consecutive days at any period

#fill in missing values with 0
df = df.fillna(0)

#calculate the number of consecutive days with missing or incomplete data
df = df.withColumn("row_num", row_number().over(Window.partitionBy("Station ID").orderBy("Date")))

df = df.withColumn("row_num", col("row_num") - 1)

df = df.withColumn("row_num", col("row_num") - row_number().over(Window.partitionBy("Station ID", "TMIN", "TMAX", "TAVG", "PRCP").orderBy("Date")))

df = df.withColumn("row_num", abs(col("row_num")))

df = df.withColumn("row_num", row_number().over(Window.partitionBy("Station ID", "TMIN", "TMAX", "TAVG", "PRCP", "row_num").orderBy("Date")))

df = df.withColumn("row_num", col("row_num") - 1)

df = df.withColumn("row_num", col("row_num") - row_number().over(Window.partitionBy("Station ID", "TMIN", "TMAX", "TAVG", "PRCP", "row_num").orderBy("Date")))

#filter out stations with more than 10 consecutive days of missing or incomplete data

df = df.filter(col("row_num") <= 10)

df = df.drop("row_num")

#sort the data by station id and date
df = df.orderBy("Station ID", "Date")


#write the output to a csv file
df.coalesce(1).write.csv("output", header=True)

spark.stop()
```

After this step our data looks like

```csv
Station ID,country_name,Date,TMIN,TMAX,TAVG,PRCP
USW00094967,United States,19351112,-139,-28,-83.5,0
USW00094967,United States,19351113,-56,0,-28.0,0
USW00094967,United States,19351114,-94,0,-47.0,3
USW00094967,United States,19351115,-161,-39,-100.0,0
USW00094967,United States,19351116,-56,6,-25.0,0
USW00094967,United States,19351117,-11,17,3.0,0
USW00094967,United States,19351118,-11,17,3.0,10
USW00094967,United States,19351119,-139,0,-69.5,5
USW00094967,United States,19351120,-122,-56,-89.0,0
```

- **Imputing missing temperature data:** Based on the question if any of the element is missing even after first part of the question, then they are calculated in this step based on the TAVG formula given in the question. Dataset from previous step has been given to this step as input.

```python
from pyspark.sql import SparkSession
from pyspark.sql import SQLContext
from pyspark.sql.functions import *
from pyspark.sql.window import Window
from pyspark.sql.types import *
import sys


spark = SparkSession.builder.appName("q1_part3")\
    .config("spark.executor.memory", "4g") \
    .config("spark.driver.memory", "4g") \
    .getOrCreate()

sc = spark.sparkContext
sqlContext = SQLContext(sc)

#read the input file
df = spark.read.csv("q1_part2_output.csv", header=True, inferSchema=True)

df = df.withColumn("TAVG", when(col("TAVG") == 0, (col("TMIN") + col("TMAX")) / 2).otherwise(col("TAVG")))

#if TMIN is missing, it can be approximated as 2 * TAVG - TMAX
df = df.withColumn("TMIN", when(col("TMIN") == 0, 2 * col("TAVG") - col("TMAX")).otherwise(col("TMIN")))

#if TMAX is missing, it can be approximated as 2 * TAVG - TMIN
df = df.withColumn("TMAX", when(col("TMAX") == 0, 2 * col("TAVG") - col("TMIN")).otherwise(col("TMAX")))

df = df.orderBy("Station ID", "Date")

df.show()

df.coalesce(1).write.csv("output", header = "true")
```

Now our data looks like this

```csv
Station ID,country_name,Date,TMIN,TMAX,TAVG,PRCP
USW00094967,United States,19351112,-139.0,-28.0,-83.5,0
USW00094967,United States,19351113,-56.0,0.0,-28.0,0
USW00094967,United States,19351114,-94.0,0.0,-47.0,3
USW00094967,United States,19351115,-161.0,-39.0,-100.0,0
USW00094967,United States,19351116,-56.0,6.0,-25.0,0
```

- **Handling temperature outliers**: Our code calculates rolling averages and identifies temperature outliers where the absolute difference from the mean temperature exceeds 10 degrees. It utilizes a window specification to compute and replace outliers with the average of prior and subsequent values. The temperatures (TMIN, TMAX, TAVG) are rounded to one decimal place. After cleaning the data, it is saved back to a CSV file, ensuring a more consistent and smooth dataset for further analysis.
    - **Outlier** : Outlier criteria for temperatures is defined as values where the absolute difference from the station's mean temperature exceeds a threshold of 10.0 degrees.
    - **Output** : Output consists of TAVG's data smoothened across all datapoints. We dont have any irregularities in our data.

Output :

```
Station ID,country_name,Date,TMIN,TMAX,TAVG,PRCP
USW00025325,United States,19351112,-44.0,22.0,25.8,0
USW00025325,United States,19351113,0.0,44.0,23.1,196
USW00025325,United States,19351114,33.0,72.0,16.2,302
USW00025325,United States,19351115,17.0,67.0,16.1,13
USW00025325,United States,19351116,-11.0,33.0,26.3,10
USW00025325,United States,19351117,-11.0,33.0,30.1,15
USW00025325,United States,19351118,-11.0,33.0,29.2,0
```

Code :

```python

from pyspark.sql import SparkSession
from pyspark.sql import SQLContext
from pyspark.sql.functions import *
from pyspark.sql.window import Window
from pyspark.sql.types import *
import sys

spark = SparkSession.builder.appName("q1_part4")\
    .config("spark.executor.memory", "4g") \
    .config("spark.driver.memory", "4g") \
    .getOrCreate()

sc = spark.sparkContext
sqlContext = SQLContext(sc)


# Load your data into a DataFrame
df = spark.read.csv("q1_part3_output.csv", header=True, inferSchema=True)

# Define window specification for rolling averages
windowSpec = Window.partitionBy("Station ID").orderBy("Date").rowsBetween(-3, 3)

# Calculate mean temperature for each station
mean_df = df.groupBy("Station ID").agg(avg(col("TAVG")).alias("mean_temp"))

# Join mean temperature back to the original DataFrame
df = df.join(mean_df, "Station ID")

# Define the threshold for outliers
threshold = 10.0

# Calculate rolling averages for imputation
df = df.withColumn("prev_avg", avg("TAVG").over(
    windowSpec.rowsBetween(-3, -1)))
df = df.withColumn("next_avg", avg("TAVG").over(windowSpec.rowsBetween(1, 3)))

# Calculate the final average using both previous and next averages
df = df.withColumn("final_avg", (col("prev_avg") + col("next_avg")) / 2)

# Replace outliers by checking if the absolute difference from the mean is greater than the threshold
df = df.withColumn(
    "TAVG",
    when(
        expr("abs(TAVG - mean_temp) > {}".format(threshold)),
        col("final_avg")
    ).otherwise(col("TAVG"))
)
# Drop the extra columns used for calculations
df = df.drop("mean_temp", "prev_avg", "next_avg", "final_avg")

# Round off TMIN, TMAX, and TAVG to one decimal place
df = df.withColumn("TMIN", round(col("TMIN"), 1))
df = df.withColumn("TMAX", round(col("TMAX"), 1))
df = df.withColumn("TAVG", round(col("TAVG"), 1))

#
# Optionally, save the DataFrame back to a CSV file
df.coalesce(1).write.csv("output", header="true")

```

- **Imputing missing precipitation data** : In this step all the None values present in the precipitation will be replaced with average of the previous three days precipitation values. Output from the previous step is given as input to this step.

Code :

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, when, avg, coalesce
from pyspark.sql.window import Window

# Initialize Spark session
spark = SparkSession.builder.appName("q1_part5")\
    .config("spark.executor.memory", "4g") \
    .config("spark.driver.memory", "4g") \
    .getOrCreate()

# Load data from CSV file
df = spark.read.csv("q1_part4_output.csv", header=True, inferSchema=True)

# Define the window specification for rolling average
windowSpec = Window.partitionBy("Station ID").orderBy("Date").rowsBetween(-3, -1)

# Calculate the rolling average of the last three valid precipitation values (ignoring zeros)
# Using conditional averaging only where PRCP > 0
avg_prcp = avg(when(col("PRCP") > 0, col("PRCP"))).over(windowSpec)

# Update PRCP column where values are zero with the calculated average
df = df.withColumn("PRCP", when((col("PRCP").isNull() | (col("PRCP") == 0)), coalesce(avg_prcp, col("PRCP"))).otherwise(col("PRCP")))
# Show the updated DataFrame to verify results
df.show()
df.filter(((col("TMAX") > 300) | (col("TMIN") > 300)| (col("TAVG") > 300))).show()

df.coalesce(1).write.csv("output", header="true")
```

Now all the None values in the dataset are replaced accordingly. Our dataset looks like this

```csv
Station ID,country_name,Date,TMIN,TMAX,TAVG,PRCP
USW00025325,United States,19351112,-44.0,22.0,25.8,0
USW00025325,United States,19351113,0.0,44.0,23.1,196
USW00025325,United States,19351114,33.0,72.0,16.2,302
USW00025325,United States,19351115,17.0,67.0,16.1,13
USW00025325,United States,19351116,-11.0,33.0,26.3,10
USW00025325,United States,19351117,-11.0,33.0,30.1,15
```

- **Handling precipitation outliers** : Our Spark code computes lagged precipitation values for the previous three days using a defined window specification by station and date. It calculates a moving average of these values and replaces precipitation outliers defined as values exceeding a set threshold of 1000 with the computed average. Finally, the cleaned data is saved to a CSV file.
    - **Outlier**: Outlier in precipitation data is defined as any value that exceeds 1000 units, significantly higher than the moving average of the previous three days' values.

Code :

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, when, avg, lag, round
from pyspark.sql.window import Window

# Create a Spark session
spark = SparkSession.builder \
    .config("spark.executor.memory", "4g") \
    .config("spark.driver.memory", "4g") \
    .appName("q1_part6") \
    .getOrCreate()

# Load the data
df = spark.read.csv("q1_part5_output.csv", header=True, inferSchema=True)


# Define a window specification ordered by Date for each station
windowSpec = Window.partitionBy("Station ID").orderBy("Date")

# Add columns for lagged precipitation values
df = df.withColumn("lag1", lag("PRCP", 1).over(windowSpec))
df = df.withColumn("lag2", lag("PRCP", 2).over(windowSpec))
df = df.withColumn("lag3", lag("PRCP", 3).over(windowSpec))

# Compute the moving average of the previous 3 days
df = df.withColumn("avg_3days", round((col("lag1") + col("lag2") + col("lag3")) / 3, 1))

# Here, an outlier is simplistically defined as a value much higher than recent averages
outlier_threshold = 1000
df = df.withColumn("PRCP", when(col("PRCP") > outlier_threshold,col("avg_3days")).otherwise(col("PRCP")))

df = df.drop("lag1", "lag2", "lag3", "avg_3days")



df.coalesce(1).write.csv("output", header="true")


# Stop the Spark session
spark.stop()

```

Our final output after the above two steps would look something like this

```
Station ID,country_name,Date,TMIN,TMAX,TAVG,PRCP
USW00025325,United States,19351112,-44.0,22.0,25.8,145.0
USW00025325,United States,19351113,0.0,44.0,23.1,196.0
USW00025325,United States,19351114,33.0,72.0,16.2,302.0
USW00025325,United States,19351115,17.0,67.0,16.1,13.0
USW00025325,United States,19351116,-11.0,33.0,26.3,10.0
USW00025325,United States,19351117,-11.0,33.0,30.1,15.0
```

This is our final input to Step 2

- **Step 2 - Temperature Analysis:** In this we were asked to write two Spark applications

Part 2: Part 1

The goal is to develop a Spark Application job to perform temperature analysis for a given geographic location, period of time (between two dates), and unit of time. In practice, the job receives several parameters - location (station, state, or country), start date and end date, and aggregation unit (year, month, or day), and outputs a file with a time reference, the average minimum temperature, the average maximum temperature, and the average of the average temperatures.

Input :

```
Station ID,country_name,Date,TMIN,TMAX,TAVG,PRCP
USW00025325,United States,19351112,-44.0,22.0,25.8,145.0
USW00025325,United States,19351113,0.0,44.0,23.1,196.0
USW00025325,United States,19351114,33.0,72.0,16.2,302.0
USW00025325,United States,19351115,17.0,67.0,16.1,13.0
USW00025325,United States,19351116,-11.0,33.0,26.3,10.0
USW00025325,United States,19351117,-11.0,33.0,30.1,15.0
```

Code

```python
from pyspark import SparkContext
import sys

# Initialize SparkContext
sc = SparkContext(appName="TemperatureAnalysis")

# Load input data
input_data = sc.textFile("q1_part6_output.csv")  # Assuming the input file is named "input.csv"

# Print first few lines of input data
print("Sample of input data:")
for line in input_data.take(5):
    print(line)

# Read input parameters
location_filter = sys.argv[1]
start_date = int(sys.argv[2])
end_date = int(sys.argv[3])
aggregation_unit = sys.argv[4]

def parse_date(date, unit):
    if unit == "year":
        return date[:4]
    elif unit == "month":
        return date[:6]
    elif unit == "day":
        return date
    else:
        raise ValueError("Invalid aggregation unit")

# Define a function to parse and filter data
def parse_and_filter(line):
    try:
        parts = line.strip().split(',')
        station = parts[0]
        country = parts[1]
        date = parts[2]
        tmin = float(parts[3])
        tmax = float(parts[4])
        tavg = float(parts[5])

        if country == location_filter and start_date <= int(date) <= end_date:
            return True, (parse_date(date, aggregation_unit), tmin, tmax, tavg)
        else:
            return False, None
    except ValueError:
        return False, None

# Apply parsing and filtering function
filtered_data = input_data.map(parse_and_filter).filter(lambda x: x[0])

# Display filtered data
print("Filtered data:")
for data in filtered_data.take(5):
    print(data)

# Aggregate data by key (aggregation time frame) and compute average temperatures
averaged_data = filtered_data.map(lambda x: (x[1][0], (x[1][1], x[1][2], x[1][3], 1))) \
                              .reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1], a[2] + b[2], a[3] + b[3])) \
                              .map(lambda x: (x[0], (x[1][0] / x[1][3], x[1][1] / x[1][3], x[1][2] / x[1][3])))

# Display averaged data
print("Averaged data:")
for data in averaged_data.take(5):
    print(data)

# Arrange data in descending order of the aggregation unit
sorted_data = averaged_data.sortByKey(ascending=True)

# Format output with tab-separated values
formatted_data = sorted_data.map(lambda x: '\t'.join(map(str, [x[0], "{:.3f}".format(x[1][0]), "{:.3f}".format(x[1][1]), "{:.3f}".format(x[1][2])])))

# Save formatted data to a single file
formatted_data.coalesce(1).saveAsTextFile("FinalOutput")

# Stop SparkContext
sc.stop()
```

Execution

```
deepu@Deepigas-MacBook-Air Project2 % python3 Question2Part1.py "United States" 19290101 19291201 "month"
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
24/05/01 20:21:27 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java
 classes where applicable
Sample of input data:
Station ID,country_name,Date,TMIN,TMAX,TAVG,PRCP
CA001021230,Canada,19240101,0.0,0.0,0.0,0.0
CA001021230,Canada,19240102,0.0,0.0,0.0,0.0
CA001021230,Canada,19240103,0.0,0.0,0.0,0.0
CA001021230,Canada,19240104,0.0,0.0,0.0,0.0
Filtered data:
(True, ('192901', 100.0, 0.0, -5.1))
(True, ('192901', -33.0, 0.0, 9.7))
(True, ('192901', -50.0, 22.0, 24.1))
(True, ('192901', 0.0, 0.0, 43.6))
(True, ('192901', -28.0, 72.0, 55.7))
Averaged data:
('192906', (90.36051245396771, 195.16481501623755, 142.70942787886258))
('192907', (112.65122796944996, 220.50897841704878, 166.33133493990385))
('192908', (106.30159769213789, 220.69535002321564, 163.6915662558888))
('192902', (-57.5416925583192, 29.23941521253548, -13.95995750969889))
('192905', (58.94139230089765, 161.78568894056184, 108.06979623575918))
24/05/01 20:21:44 WARN GarbageCollectionMetrics: To enable non-built-in garbage collector(s) List(G1 Concurrent GC),
users should configure it(them) to spark.eventLog.gcMetrics.youngGenerationGarbageCollectors or spark.eventLog.gcMetr
ics.oldGenerationGarbageCollectors
```

Output generated is the data over the given aggregation time frame, average of TMAX, average of TMIN, average of TAVG

Output

```
192901  -60.474 25.528  -15.881
192902  -57.542 29.239  -13.960
192903  5.444   97.262  53.323
192904  32.925  127.646 79.764
192905  58.941  161.786 108.070
192906  90.361  195.165 142.709
192907  112.651 220.509 166.331
192908  106.302 220.695 163.692
192909  79.253  181.437 128.659
192910  39.978  141.117 90.372
192911  -8.508  78.026  35.240
192912  -55.206 31.245  -5.240
```

Part 2: Part 2

Another standalone Spark application that compares the temperatures between two different regions. A similar approach to the above is followed to generate a file with data for both regions and the difference between the two regions.

Code

```python
from pyspark import SparkContext
import sys

# Initialize SparkContext
sc = SparkContext(appName="TemperatureComparison")

# Load input data
input_data = sc.textFile("q1_part6_output.csv")  # Assuming the input file is named "input.csv"

# Read input parameters
location1_filter = sys.argv[1]
location2_filter = sys.argv[2]
start_date = int(sys.argv[3])
end_date = int(sys.argv[4])
aggregation_unit = sys.argv[5]

def parse_date(date, unit):
    if unit == "year":
        return date[:4]
    elif unit == "month":
        return date[:6]
    elif unit == "day":
        return date
    else:
        raise ValueError("Invalid aggregation unit")

# Define a function to parse and filter data
def parse_and_filter(line, location_filter):
    try:
        parts = line.strip().split(',')
        station, country, date, tmin, tmax, tavg, prcp = parts
        if country.strip() == location_filter and start_date <= int(date) <= end_date:
            return True, (parse_date(date, aggregation_unit), float(tmin), float(tmax), float(tavg))
        else:
            return False, None
    except ValueError:
        return False, None

# Parse and filter data for location 1
filtered_data1 = input_data.map(lambda line: parse_and_filter(line, location1_filter)).filter(lambda x: x[0])

# Parse and filter data for location 2
filtered_data2 = input_data.map(lambda line: parse_and_filter(line, location2_filter)).filter(lambda x: x[0])

# Aggregate data by key (aggregation time frame) and compute average temperatures for location 1
averaged_data1 = filtered_data1.map(lambda x: (x[1][0], (x[1][1], x[1][2], x[1][3], 1))) \
                                .reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1], a[2] + b[2], a[3] + b[3])) \
                                .map(lambda x: (x[0], (x[1][0] / x[1][3], x[1][1] / x[1][3], x[1][2] / x[1][3])))

# Aggregate data by key (aggregation time frame) and compute average temperatures for location 2
averaged_data2 = filtered_data2.map(lambda x: (x[1][0], (x[1][1], x[1][2], x[1][3], 1))) \
                                .reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1], a[2] + b[2], a[3] + b[3])) \
                                .map(lambda x: (x[0], (x[1][0] / x[1][3], x[1][1] / x[1][3], x[1][2] / x[1][3])))

# Calculate the difference between the two locations for each temperature component
difference_data = averaged_data1.join(averaged_data2) \
                                 .map(lambda x: (x[0], (x[1][0][0] - x[1][1][0], x[1][0][1] - x[1][1][1], x[1][0][2] - x[1][1][2])))

# Format output for location 1
formatted_data1 = averaged_data1.map(lambda x: '{:<15} {:<8} {:<10.2f} {:<10.2f} {:<10.2f}'.format(location1_filter, x[0], x[1][0], x[1][1], x[1][2]))

# Format output for location 2
formatted_data2 = averaged_data2.map(lambda x: '{:<15} {:<8} {:<10.2f} {:<10.2f} {:<10.2f}'.format(location2_filter, x[0], x[1][0], x[1][1], x[1][2]))

# Format output for the difference
formatted_difference = difference_data.map(lambda x: '{:<15} {:<8} {:<10.2f} {:<10.2f} {:<10.2f}'.format("Difference", x[0], x[1][0], x[1][1], x[1][2]))

# Save formatted data to a single file
formatted_data1.union(formatted_data2).union(formatted_difference).coalesce(1).saveAsTextFile("FinalOutput2")

# Stop SparkContext
sc.stop()
```

Execution

```
deepu@Deepigas-MacBook-Air Project2 % python3 Question2Part2.py "Canada" "United States" 19350101 19351201 year
24/05/01 20:32:13 WARN Utils: Your hostname, Deepigas-MacBook-Air.local resolves to a loopback address: 127.0.0.1; us
ing 172.20.5.108 instead (on interface en0)
24/05/01 20:32:13 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
24/05/01 20:32:13 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java
 classes where applicable
24/05/01 20:32:30 WARN GarbageCollectionMetrics: To enable non-built-in garbage collector(s) List(G1 Concurrent GC),
users should configure it(them) to spark.eventLog.gcMetrics.youngGenerationGarbageCollectors or spark.eventLog.gcMetr
ics.oldGenerationGarbageCollectors
deepu@Deepigas-MacBook-Air Project2 %
```

Output

```
Canada          1935     -11.89     77.57      32.79
United States   1935     42.62      138.54     90.70
Difference      1935     -54.51     -60.97     -57.91
```

- **Step 3 - Extreme weather events :**

Data Retrieval: Begin by retrieving historical temperature data for a specified geographic location (station, state, country, or continent).
Defining Extreme Events: Establish clear criteria to define heatwaves and cold spells. Examples could include:

Heatwave: A period with consecutive days exceeding a specific high-temperature threshold.

Cold Spell: A sequence of consecutive days falling below a defined low-temperature threshold.

Identifying Event Periods: Utilize the defined criteria to scan through daily temperature data and pinpoint sequences that meet the heatwave or cold spell thresholds. Flag these periods within the dataset.
Event Frequency and Intensity Analysis:

- Frequency: Calculate how often extreme events occur within the chosen time frame.
- Intensity: Analyze the severity of each event by considering factors like:
    - Duration (number of consecutive days)
    - Maximum temperature during the heatwave
    - Minimum temperature during the cold spell

Comparative Analysis Across Locations: Develop a solution to compare the frequency and intensity of extreme weather events across various geographic locations. This will reveal potential variations and trends in extreme weather patterns.

Code

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, lag, lead, when
from pyspark.sql.window import Window

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

# Define the new schema to match the CSV data format
schema = "station_id STRING, country STRING, dateYYYYMMDD STRING, tmin DOUBLE, tmax DOUBLE, tavg DOUBLE, prcp DOUBLE"

# Create DataFrame from the provided data with the new schema
weather_data = spark.read \
    .format("csv") \
    .schema(schema) \
    .option("header", "true") \
    .load("q1_part6_output.csv")

# Register the DataFrame as a temporary view
weather_data.createOrReplaceTempView("weather_data")
```

```python
# Capture data for heat waves
heat_wave_data = spark.sql("""
    SELECT station_id, country, dateYYYYMMDD, tmax
    FROM weather_data
    WHERE tmax > 300
    ORDER BY station_id, country, dateYYYYMMDD
    LIMIT 10000
""")
print("Count of heat_wave_data:", heat_wave_data.count())
print("Sample of heat_wave_data:")
heat_wave_data.show()
# Capture data for cold storms
cold_storm_data = spark.sql("""
    SELECT station_id, country, dateYYYYMMDD, tmin
    FROM weather_data
    WHERE tmin < -50
    ORDER BY station_id, country, dateYYYYMMDD
    LIMIT 10000
""")
print("Count of cold_storm_data:", cold_storm_data.count())
print("Sample of cold_storm_data:")
cold_storm_data.show()
```

```python
# Flagging heat waves
heat_wave_window = Window.partitionBy("station_id", "country").orderBy("dateYYYYMMDD")
heat_wave_data = heat_wave_data.withColumn("prev_tmax", lag("tmax", 1).over(heat_wave_window)) \
    .withColumn("next_tmax", lead("tmax", 1).over(heat_wave_window)) \
    .withColumn("next_next_tmax", lead("tmax", 2).over(heat_wave_window)) \
    .withColumn("next_next_next_tmax", lead("tmax", 3).over(heat_wave_window)) \
    .withColumn("heat_wave",
                when((col("tmax") > 300) &
                     (col("next_tmax") > 300) &
                     (col("next_next_tmax") > 300) &
                     (col("next_next_next_tmax") > 300), "heat_wave")
                .otherwise(None)) \
    .na.fill({"heat_wave": "heat_wave"}) \
    .drop("prev_tmax", "next_tmax", "next_next_tmax", "next_next_next_tmax")
```

```python
# Flagging cold storms
cold_storm_window = Window.partitionBy("station_id", "country").orderBy("dateYYYYMMDD")
cold_storm_data = cold_storm_data.withColumn("prev_tmin", lag("tmin", 1).over(cold_storm_window)) \
    .withColumn("next_tmin", lead("tmin", 1).over(cold_storm_window)) \
    .withColumn("next_next_tmin", lead("tmin", 2).over(cold_storm_window)) \
    .withColumn("next_next_next_tmin", lead("tmin", 3).over(cold_storm_window)) \
    .withColumn("cold_storm",
                when((col("tmin") < -50) &
                     (col("next_tmin") < -50) &
                     (col("next_next_tmin") < -50) &
                     (col("next_next_next_tmin") < -50), "cold_storm")
                .otherwise(None)) \
    .na.fill({"cold_storm": "cold_storm"}) \
    .drop("prev_tmin", "next_tmin", "next_next_tmin", "next_next_next_tmin")
```

```python
# Combine heat wave and cold storm data
processed_data = heat_wave_data.union(cold_storm_data)

# Write the processed data to a file
processed_data.write \
    .format("csv") \
    .option("header", "true") \
    .mode("overwrite") \
    .save("processed_data_output")

# Stop the Spark session
spark.stop()
```

Output is as following

```
station_id,country,dateYYYYMMDD,tmax,heat_wave
CA001012008,Canada,19100710,328.0,heat_wave
CA001012008,Canada,19100720,311.0,heat_wave
CA001012008,Canada,19110713,322.0,heat_wave
CA001012008,Canada,19110714,306.0,heat_wave
CA001012008,Canada,19110715,333.0,heat_wave
CA001012008,Canada,19110716,317.0,heat_wave
CA001012008,Canada,19110717,333.0,heat_wave
```

```
station_id,country,dateYYYYMMDD,tmax,heat_wave
CA001012008,Canada,19100102,-100.0,cold_storm
CA001012008,Canada,19100103,-56.0,cold_storm
CA001012008,Canada,19100105,-56.0,cold_storm
CA001012008,Canada,19100219,-61.0,cold_storm
CA001012008,Canada,19100221,-56.0,cold_storm
CA001012008,Canada,19100222,-100.0,cold_storm
CA001012008,Canada,19110111,-78.0,cold_storm
```

Reading Weather Data:

- Loads the weather data from a CSV file named "small.csv" using `spark.read`.
- Specifies the format as "csv" and provides the defined schema.
- Sets `header=True` to indicate the first row contains column names.

Identifying Extreme Events (Function):

- Defines a function `identify_extreme_events` that takes weather data as input.
- Uses `defaultdict(lambda: {'heat_wave': [], 'cold_storm': []})` to create a dictionary where keys are station and country tuples, and values are dictionaries containing lists for heatwaves and cold storms.
- Iterates through each row (data point) in the weather data:
    - Extracts information like station ID, country, date string, maximum temperature (tmax), and minimum temperature (tmin).
    - Converts the date string to a Python datetime object using `datetime.strptime`.
    - Checks for heatwaves: If tmax is not null and greater than or equal to 30°C (assumed threshold), it appends a tuple of (date, tmax) to the heat_wave list for the corresponding station and country.
    - Checks for cold spells: If tmin is not null and less than -50°C (assumed threshold), it appends a tuple of (date, tmin) to the cold_storm list for the corresponding station and country.
- Returns the dictionary containing extreme events data categorized by station and country.

Calculating Duration and End Dates (Function - Incomplete):

- Defines a function `calculate_duration_end_dates` that takes station ID, country, event type (heatwave or cold storm), and event data as input (currently commented out).
- This function is intended to process the event data (list of date-temperature tuples) and calculate properties like duration, end date, and intensity.
- However, the current implementation within the provided code is commented out.

Processing and Analyzing Data:

- Collects all data from the DataFrame into a list named `processed_data` using `collect()`.
- Calls the `identify_extreme_events` function with the processed data to identify extreme events and stores the result in `extreme_events_data`.

- Initializes an empty list `summary_data` to store summarized extreme event information.
- Iterates through the `extreme_events_data` dictionary:
    - For each station-country key and its corresponding event data (heatwave/cold storm lists):
        - Calls the (commented out) `calculate_duration_end_dates` function to potentially calculate additional details like duration and end dates. Note: This functionality seems to be missing from the provided code.
        - The results (presumably including station ID, country, event type, start date, end date, duration, intensity, and frequency) are then appended to the `summary_data` list.

Code :

```python
from pyspark.sql import SparkSession
from datetime import datetime
import csv
from collections import defaultdict

# Initialize Spark session
spark = SparkSession.builder \
    .config("spark.executor.memory", "8g")\
    .config("spark.driver.memory", "8g")\
    .appName("ExtremeEventsAnalysis") \
    .getOrCreate()

# Define the schema to match the CSV data format
schema = "station_id STRING, country STRING, dateYYYYMMDD STRING, tmin DOUBLE, tmax DOUBLE, tavg DOUBLE, prcp DOUBLE"

# Create DataFrame from the provided data with the new schema
weather_data = spark.read \
    .format("csv") \
    .schema(schema) \
    .option("header", "true") \
    .load("q1_part6_output.csv")

```

```python
# Function to identify extreme events
def identify_extreme_events(data):
    extreme_events_by_location = defaultdict(lambda: {'heat_wave': [], 'cold_storm': []})

    for row in data:
        station_id = row.station_id
        country = row.country
        date_str = row.dateYYYYMMDD
        tmax = row.tmax
        tmin = row.tmin

        date_obj = datetime.strptime(date_str, "%Y%m%d")

        if tmax is not None and tmax >= 300:
            extreme_events_by_location[(station_id, country)]['heat_wave'].append((date_obj, tmax))

        if tmin is not None and tmin < -50:
            extreme_events_by_location[(station_id, country)]['cold_storm'].append((date_obj, tmin))

    return extreme_events_by_location
```

```python
# Function to calculate duration and end dates
def calculate_duration_end_dates(station_id, country, event_type, event_data):
    result = []
    for loc, data in event_data.items():
        if data:
            data.sort(key=lambda x: x[0])
            start_date = data[0][0]
            end_date = data[-1][0]
            duration = (end_date - start_date).days + 1
            intensity = max([x[1] for x in data])
            event_type = "heat_wave" if intensity >= 300 else "cold_storm"

            result.append((station_id, country, event_type, start_date.strftime("%Y-%m-%d"),
                           end_date.strftime("%Y-%m-%d"), duration, intensity, len(data)))
    return result
```

```python
# Process the weather data and identify extreme events
processed_data = weather_data.collect()
extreme_events_data = identify_extreme_events(processed_data)

# Analyze the extreme events
summary_data = []
for (station_id, country), event_data in extreme_events_data.items():
    for event_type, data in event_data.items():
        summary_data.extend(calculate_duration_end_dates(station_id, country, event_type, {event_type: data}))

# Write summary to a CSV file
output_filepath = 'extreme_events_summary_with_dates.csv'

with open(output_filepath, 'w', newline='') as csvfile:
    fieldnames = ['Station ID', 'Country', 'Event Type', 'Start Date', 'End Date', 'Duration (days)', 'Intensity (°C)', 'Frequency']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    for row in summary_data:
        writer.writerow({
            'Station ID': row[0],
            'Country': row[1],
            'Event Type': row[2],
            'Start Date': row[3],
            'End Date': row[4],
            'Duration (days)': row[5],
            'Intensity (°C)': row[6],
            'Frequency': row[7]
        })

print(f"Summary of extreme events with start and end dates written to {output_filepath}")

# Stop the Spark session
spark.stop()
```

Output :

```csv
Station ID,Country,Event Type,Start Date,End Date,Duration (days),Intensity (°C),Frequency
CA001025340,Canada,heat_wave,1910-07-10,1935-08-28,9181,339.0,80
CA001025340,Canada,cold_storm,1910-01-02,1935-11-02,9436,-56.0,159
CA001086090,Canada,heat_wave,1912-08-07,1935-08-12,8406,367.0,455
CA001086090,Canada,cold_storm,1912-10-20,1935-12-05,8447,-56.0,1265
CA001088970,Canada,heat_wave,1926-07-09,1935-07-21,3300,356.0,97
CA001088970,Canada,cold_storm,1926-01-11,1935-12-31,3642,-56.0,1227
CA001092970,Canada,heat_wave,1914-08-21,1935-07-21,7640,339.0,71
CA001092970,Canada,cold_storm,1910-01-01,1935-12-31,9496,-56.0,3567
CA001108465,Canada,heat_wave,1910-07-10,1934-08-24,8812,333.0,33
CA001108465,Canada,cold_storm,1910-01-01,1935-11-02,9437,-56.0,208
CA001108910,Canada,heat_wave,1929-09-05,1935-08-27,2183,333.0,12
CA001108910,Canada,cold_storm,1929-01-19,1935-11-03,2480,-56.0,68
CA001125585,Canada,heat_wave,1910-07-12,1918-08-25,2967,378.0,73
CA001125585,Canada,cold_storm,1910-01-01,1918-04-03,3015,-56.0,642
CA001133275,Canada,heat_wave,1911-05-31,1935-09-09,8868,428.0,1125
CA001133275,Canada,cold_storm,1911-03-14,1935-12-29,9057,-56.0,1730
CA001143900,Canada,heat_wave,1912-06-20,1935-08-12,8454,378.0,141
CA001143900,Canada,cold_storm,1912-03-01,1935-12-22,8697,-56.0,1259
CA001153655,Canada,heat_wave,1913-06-07,1935-07-23,8082,372.0,267
```

- **Step 4 -**

Our PySpark script designed for temperature and precipitation forecasting using various machine learning models.

Spark Session Initialization

- The Spark session is configured with specific settings for memory allocation, parallelism, and adaptive query execution, which are crucial for optimizing performance in processing large datasets.

Data Loading and Preprocessing

- Data is loaded from a CSV file. The 'Date' column is converted to a proper date format. Missing values are fled with zeros, which may impact model accuracy if zero is not a reasonable placeholder.

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import to_date, col, year, month, dayofmonth, dayofweek, udf,round
from pyspark.sql.types import StringType
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler, StringIndexer, OneHotEncoder
from pyspark.ml.regression import RandomForestRegressor, GBTRegressor, LinearRegression
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

# Initialize Spark Session
spark = SparkSession.builder\
    .config("spark.executor.memory", "6g")\
    .config("spark.driver.memory", "6g")\
    .config("spark.sql.shuffle.partitions", "200")\
    .config("spark.default.parallelism", "200")\
    .config("spark.sql.adaptive.enabled", "true")\
    .config("spark.sql.adaptive.skewJoin.enabled", "true")\
    .config("spark.sql.adaptive.skewJoin.skewedPartitionFactor", "5")\
    .config("spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes", "256000000")\
    .config("spark.sql.adaptive.skewJoin.skewedPartitionSizeInBytes", "134217728")\
    .appName("Temperature Forecasting").getOrCreate()

# Load data
data_path = "q1_part6_output.csv"
df = spark.read.csv(data_path, header=True, inferSchema=True)

# Convert 'Date' to date format
df = df.withColumn("Date", to_date(col("Date"), 'yyyyMMdd'))

# Fill or drop missing values
df = df.na.fill({"TMIN": 0, "TMAX": 0, "TAVG": 0, "PRCP": 0})

```

- A user-defined function (UDF) is employed to categorize data into seasons based on the month, which is a significant feature for temperature and precipitation modeling.

```python
# Define UDF for season calculation
def get_season(month):
    if month in [12, 1, 2]:
        return "Winter"
    elif month in [3, 4, 5]:
        return "Spring"
    elif month in [6, 7, 8]:
        return "Summer"
    elif month in [9, 10, 11]:
        return "Fall"


season_udf = udf(get_season, StringType())
df = df.withColumn("Month", month("Date"))
df = df.withColumn("Season", season_udf(col("Month")))

```

Feature Engineering

- StringIndexer and OneHotEncoder are used to convert the 'Season' categorical data into numeric format, which is necessary for machine learning algorithms that require numerical input.

```python
 Convert "Season" to numerical format using StringIndexer and OneHotEncoder
season_indexer = StringIndexer(
    inputCol="Season", outputCol="SeasonIndex", handleInvalid="skip")
season_encoder = OneHotEncoder(
    inputCols=["SeasonIndex"], outputCols=["SeasonEncoded"])

# Additional date-related features
df = df.withColumn("Year", year("Date"))
df = df.withColumn("Day", dayofmonth("Date"))
df = df.withColumn("Weekday", dayofweek("Date"))

#round prcp to 1 decimal place using round
df = df.withColumn("PRCP", round(col("PRCP"), 1))
#show 50 rows
df.show(50)

```

- New date-related features like 'Year', 'Day', and 'Weekday' are created to capture more temporal patterns, improving model predictions.

Features used :

In the our script, several features are utilized to predict maximum temperature (TMAX) and precipitation (PRCP). Here's a concise overview of each feature used:

1. **Year, Month, Day, and Weekday**: Extracted from the 'Date' column, these temporal features capture the cyclical nature of weather patterns over different timescales. Including these allows the model to detect and utilize patterns based on time, such as seasonal variations and day-of-the-week effects.

2. **TMIN**: This is the minimum temperature for the day. It is used directly as a predictor for TMAX, assuming a strong correlation between the day's minimum and maximum temperatures.

3. **PRCP**: Represents the precipitation amount. When predicting TMAX, it can be an indicative feature, as weather conditions affecting precipitation can also influence temperature extremes.

4. **SeasonEncoded**: Derived from the 'Season' column, which categorizes the date into one of four seasons: Winter, Spring, Summer, or Fall. The 'Season' is first transformed into a numerical index using `StringIndexer`, and then `OneHotEncoder` is applied to convert these indices into a binary vector format. This is crucial for capturing seasonal effects on temperature and precipitation, as these can vary significantly with seasons.

These features are assembled into a single vector using `**VectorAssembler**`, which is a common preprocessing step required by ML algorithms in Spark to handle input features collectively during model training and predictions. This feature set aims to provide a comprehensive representation of the factors influencing weather conditions, enhancing the model's ability to make accurate predictions.

```
# Assemble features
feature_cols = ['Year', 'Month', 'Day',
                'Weekday', 'TMIN', 'PRCP', 'SeasonEncoded']
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")

```

Station ID	country_name	Date	TMIN	TMAX	TAVG	PRCP	Month	Season	Year	Day	Weekday
CA001021230	Canada	1924-01-01	0.0	0.0	0.0	0.0	1	Winter	1924	1	3
CA001021230	Canada	1924-01-02	0.0	0.0	0.0	0.0	1	Winter	1924	2	4
CA001021230	Canada	1924-01-03	0.0	0.0	0.0	0.0	1	Winter	1924	3	5
CA001021230	Canada	1924-01-04	0.0	0.0	0.0	0.0	1	Winter	1924	4	6
CA001021230	Canada	1924-01-05	0.0	0.0	0.0	0.0	1	Winter	1924	5	7
CA001021230	Canada	1924-01-06	0.0	0.0	0.0	117.0	1	Winter	1924	6	1
CA001021230	Canada	1924-01-07	0.0	0.0	0.0	201.0	1	Winter	1924	7	2
CA001021230	Canada	1924-01-08	0.0	0.0	0.0	155.0	1	Winter	1924	8	3
CA001021230	Canada	1924-01-09	0.0	0.0	0.0	81.0	1	Winter	1924	9	4
CA001021230	Canada	1924-01-10	0.0	0.0	0.0	91.0	1	Winter	1924	10	5
CA001021230	Canada	1924-01-11	0.0	0.0	0.0	5.0	1	Winter	1924	11	6
CA001021230	Canada	1924-01-12	0.0	0.0	0.0	59.0	1	Winter	1924	12	7
CA001021230	Canada	1924-01-13	0.0	0.0	0.0	48.0	1	Winter	1924	13	1
CA001021230	Canada	1924-01-14	0.0	0.0	0.0	5.0	1	Winter	1924	14	2
CA001021230	Canada	1924-01-15	0.0	0.0	0.0	0.0	1	Winter	1924	15	3
CA001021230	Canada	1924-01-16	0.0	0.0	0.0	0.0	1	Winter	1924	16	4
CA001021230	Canada	1924-01-17	0.0	0.0	0.0	0.0	1	Winter	1924	17	5
CA001021230	Canada	1924-01-18	0.0	0.0	0.0	76.0	1	Winter	1924	18	6
CA001021230	Canada	1924-01-19	0.0	0.0	0.0	76.0	1	Winter	1924	19	7
CA001021230	Canada	1924-01-20	0.0	0.0	0.0	76.0	1	Winter	1924	20	1
CA001021230	Canada	1924-01-21	0.0	0.0	0.0	51.0	1	Winter	1924	21	2
CA001021230	Canada	1924-01-22	0.0	0.0	0.0	15.0	1	Winter	1924	22	3
CA001021230	Canada	1924-01-23	0.0	0.0	0.0	107.0	1	Winter	1924	23	4
CA001021230	Canada	1924-01-24	0.0	0.0	0.0	57.666666666666664	1	Winter	1924	24	5
CA001021230	Canada	1924-01-25	0.0	0.0	0.0	3.0	1	Winter	1924	25	6
CA001021230	Canada	1924-01-26	0.0	0.0	0.0	229.0	1	Winter	1924	26	7
CA001021230	Canada	1924-01-27	0.0	0.0	0.0	178.0	1	Winter	1924	27	1
CA001021230	Canada	1924-01-28	0.0	0.0	0.0	157.0	1	Winter	1924	28	2
CA001021230	Canada	1924-01-29	0.0	0.0	0.0	315.0	1	Winter	1924	29	3
CA001021230	Canada	1924-01-30	0.0	0.0	0.0	119.0	1	Winter	1924	30	4
CA001021230	Canada	1924-01-31	0.0	0.0	0.0	197.0	1	Winter	1924	31	5
CA001021230	Canada	1924-02-01	0.0	0.0	0.0	104.0	2	Winter	1924	1	6
CA001021230	Canada	1924-02-02	0.0	0.0	0.0	135.0	2	Winter	1924	2	7
CA001021230	Canada	1924-02-03	0.0	0.0	0.0	198.0	2	Winter	1924	3	1
CA001021230	Canada	1924-02-04	0.0	0.0	0.0	218.0	2	Winter	1924	4	2
CA001021230	Canada	1924-02-05	0.0	0.0	0.0	10.0	2	Winter	1924	5	3
CA001021230	Canada	1924-02-06	0.0	0.0	0.0	132.0	2	Winter	1924	6	4
CA001021230	Canada	1924-02-07	0.0	0.0	0.0	120.0	2	Winter	1924	7	5
CA001021230	Canada	1924-02-08	0.0	0.0	0.0	3.0	2	Winter	1924	8	6
CA001021230	Canada	1924-02-09	0.0	0.0	0.0	310.0	2	Winter	1924	9	7
CA001021230	Canada	1924-02-10	0.0	0.0	0.0	859.0	2	Winter	1924	10	1
CA001021230	Canada	1924-02-11	0.0	0.0	0.0	183.0	2	Winter	1924	11	2
CA001021230	Canada	1924-02-12	0.0	0.0	0.0	259.0	2	Winter	1924	12	3
CA001021230	Canada	1924-02-13	0.0	0.0	0.0	140.0	2	Winter	1924	13	4

Our data looks like this after adding all the above things

Model Setup and Evaluation

- Multiple regression models (Linear Regression, Random Forest, and Gradient Boosting Tree) are defined for predicting maximum temperature ('TMAX') and precipitation ('PRCP'). Each model type has different assumptions and characteristics, offering a broad evaluation spectrum.

```python
# Define the models for predictions
lr_temp_max = LinearRegression(
    featuresCol="features", labelCol="TMAX", regParam=0.1)
lr_precip = LinearRegression(
    featuresCol="features", labelCol="PRCP", regParam=0.1)


rf_temp_max = RandomForestRegressor(featuresCol="features", labelCol="TMAX")
rf_precip = RandomForestRegressor(featuresCol="features", labelCol="PRCP")

gbt_temp_max = GBTRegressor(featuresCol="features", labelCol="TMAX")
gbt_precip = GBTRegressor(featuresCol="features", labelCol="PRCP")

```

- Models are wrapped in pipelines, which streamline the execution of multiple preprocessing steps followed by model training.

```python
# Set up the pipelines
pipeline_lr_temp_max = Pipeline(
    stages=[season_indexer, season_encoder, assembler, lr_temp_max])

pipeline_lr_precip = Pipeline(
    stages=[season_indexer, season_encoder, assembler, lr_precip])

pipeline_temp_max = Pipeline(
    stages=[season_indexer, season_encoder, assembler, rf_temp_max])

pipeline_precip = Pipeline(
    stages=[season_indexer, season_encoder, assembler, rf_precip])
pipeline_gbt_temp_max = Pipeline(
    stages=[season_indexer, season_encoder, assembler, gbt_temp_max])

pipeline_gbt_precip = Pipeline(
    stages=[season_indexer, season_encoder, assembler, gbt_precip])

```

```python
# Split the data
train_data, test_data = df.randomSplit([0.8, 0.2], seed=42)

# Train the models
model_lr_temp_max = pipeline_lr_temp_max.fit(train_data)
model_lr_precip = pipeline_lr_precip.fit(train_data)
model_rf_temp_max = pipeline_temp_max.fit(train_data)
model_rf_precip = pipeline_precip.fit(train_data)
model_gbt_temp_max = pipeline_gbt_temp_max.fit(train_data)
model_gbt_precip = pipeline_gbt_precip.fit(train_data)

# Make predictions
predictions_lr_temp_max = model_lr_temp_max.transform(test_data)
predictions_lr_precip = model_lr_precip.transform(test_data)
predictions_rf_temp_max = model_rf_temp_max.transform(test_data)
predictions_rf_precip = model_rf_precip.transform(test_data)
predictions_gbt_temp_max = model_gbt_temp_max.transform(test_data)
predictions_gbt_precip = model_gbt_precip.transform(test_data)
```

- The models are evaluated using Root Mean Square Error (RMSE), Mean Absolute Error (MAE), and R-squared (R2) metrics, providing a comprehensive view of model performance.

```python
# Evaluate the models
evaluator = RegressionEvaluator()

print("RMSE for LR Max Temp:", evaluator.evaluate(
    predictions_lr_temp_max, {evaluator.labelCol: "TMAX"}))
print("RMSE for LR Precipitation:", evaluator.evaluate(
    predictions_lr_precip, {evaluator.labelCol: "PRCP"}))
print("RMSE for RF Max Temp:", evaluator.evaluate(
    predictions_rf_temp_max, {evaluator.labelCol: "TMAX"}))

print("RMSE for RF Precipitation:", evaluator.evaluate(
    predictions_rf_precip, {evaluator.labelCol: "PRCP"}))
print("RMSE for GBT Max Temp:", evaluator.evaluate(
    predictions_gbt_temp_max, {evaluator.labelCol: "TMAX"}))

print("RMSE for GBT Precipitation:", evaluator.evaluate(
    predictions_gbt_precip, {evaluator.labelCol: "PRCP"}))
```

```python
#mean absolute error
evaluator = RegressionEvaluator(
    metricName="mae")
print("MAE for LR Max Temp:", evaluator.evaluate(
    predictions_lr_temp_max, {evaluator.labelCol: "TMAX"}))
print("MAE for LR Precipitation:", evaluator.evaluate(
    predictions_lr_precip, {evaluator.labelCol: "PRCP"}))
print("MAE for RF Max Temp:", evaluator.evaluate(
    predictions_rf_temp_max, {evaluator.labelCol: "TMAX"}))

print("MAE for RF Precipitation:", evaluator.evaluate(
    predictions_rf_precip, {evaluator.labelCol: "PRCP"}))
print("MAE for GBT Max Temp:", evaluator.evaluate(
    predictions_gbt_temp_max, {evaluator.labelCol: "TMAX"}))

print("MAE for GBT Precipitation:", evaluator.evaluate(
    predictions_gbt_precip, {evaluator.labelCol: "PRCP"}))
```

```python
#accuracy
evaluator = RegressionEvaluator(
    metricName="r2")
print("R2 for LR Max Temp:", evaluator.evaluate(
    predictions_lr_temp_max, {evaluator.labelCol: "TMAX"}))
print("R2 for LR Precipitation:", evaluator.evaluate(
    predictions_lr_precip, {evaluator.labelCol: "PRCP"}))
print("R2 for RF Max Temp:", evaluator.evaluate(
    predictions_rf_temp_max, {evaluator.labelCol: "TMAX"}))

print("R2 for RF Precipitation:", evaluator.evaluate(
    predictions_rf_precip, {evaluator.labelCol: "PRCP"}))
print("R2 for GBT Max Temp:", evaluator.evaluate(
    predictions_gbt_temp_max, {evaluator.labelCol: "TMAX"}))

print("R2 for GBT Precipitation:", evaluator.evaluate(
    predictions_gbt_precip, {evaluator.labelCol: "PRCP"}))
```

Result

```
RMSE for LR Max Temp: 75.59659370083418
RMSE for LR Precipitation: 0.10065856402388097
RMSE for RF Max Temp: 61.07782728677386
RMSE for RF Precipitation: 42.33118449165573
RMSE for GBT Max Temp: 53.00993599031006
RMSE for GBT Precipitation: 31.927349064677763
MAE for LR Max Temp: 62.01657950017026
MAE for LR Precipitation: 0.06291730016504915
MAE for RF Max Temp: 47.622588575058955
MAE for RF Precipitation: 20.62738021530612
MAE for GBT Max Temp: 36.81086029848734
MAE for GBT Precipitation: 5.690242006291626
R2 for LR Max Temp: 0.6655213891156586
R2 for LR Precipitation: 0.9999988775670628
R2 for RF Max Temp: 0.7816611312690769
R2 for RF Precipitation: 0.8014911913428372
R2 for GBT Max Temp: 0.8355331190430807
R2 for GBT Precipitation: 0.8870764137377414
```

Cross-validation

- CrossValidator is used to fine-tune the Random Forest models through a grid search over specified parameter ranges. This process helps in identifying the best model settings to minimize prediction error.

```python
#cross validation
paramGrid = ParamGridBuilder() \
    .addGrid(rf_temp_max.numTrees, [10, 20]) \
    .addGrid(rf_temp_max.maxDepth, [5, 10]) \
    .build()

crossval = CrossValidator(estimator=pipeline_temp_max,
                          estimatorParamMaps=paramGrid,
                          evaluator=RegressionEvaluator(
                              labelCol="TMAX", predictionCol="prediction", metricName="rmse"),
                          numFolds=5)

cvModel = crossval.fit(train_data)
bestModel = cvModel.bestModel
best_predictions = bestModel.transform(test_data)
best_rmse = RegressionEvaluator(
    labelCol="TMAX", predictionCol="prediction", metricName="rmse").evaluate(best_predictions)

print("TMAX BEST RSME")
print(f"Best RMSE after CV: {best_rmse}")

#write to a text file
with open("output1.txt", "w", encoding="utf-8") as f:
    f.write("Best TMAX RMSE after CV: " + str(best_rmse) + "\n")
    f.close()

```

```python
paramGrid = ParamGridBuilder() \
    .addGrid(rf_precip.numTrees, [10, 20]) \
    .addGrid(rf_precip.maxDepth, [5, 10]) \
    .build()

crossval = CrossValidator(estimator=pipeline_precip,
                          estimatorParamMaps=paramGrid,
                          evaluator=RegressionEvaluator(
                              labelCol="PRCP", predictionCol="prediction", metricName="rmse"),
                          numFolds=5)

cvModel = crossval.fit(train_data)
bestModel = cvModel.bestModel
best_predictions = bestModel.transform(test_data)
best_rmse = RegressionEvaluator(
    labelCol="PRCP", predictionCol="prediction", metricName="rmse").evaluate(best_predictions)
print("PRCP BEST RSME")
print(f"Best RMSE after CV: {best_rmse}")


#write to a text file
with open("output2.txt", "w", encoding="utf-8") as f:
    f.write("Best PRCP RMSE after CV: " + str(best_rmse) + "\n")
    f.close()

```

- This is particularly important as it systematically explores a range of configurations to improve model robustness and accuracy.

Result :

```
TMAX BEST RSME
Best RMSE after CV: 53.67472704958553
```

```
PRCP BEST RSME
Best RMSE after CV: 32.3069921518924 94
```

Error Analysis and Output

- Errors between predictions and actual observations are calculated and stored, allowing an examination of where the model predictions deviate from real values.
- Results are saved in CSV files, making them accessible for further analysis or reporting.

```
Date,Station ID,TMAX,prediction,error
1935-05-19,USW00025325,122.0,196.21,-74.21
1935-05-20,USW00025325,100.0,191.086,-91.086
1935-05-24,USW00025325,156.0,177.652,-21.652
1935-05-26,USW00025325,172.0,215.426,-43.426
1935-06-02,USW00025325,89.0,234.899,-145.899
1935-06-09,USW00025325,122.0,248.143,-126.143
1935-06-10,USW00025325,122.0,240.537,-118.537
1935-06-11,USW00025325,167.0,240.17,-73.17
1935-06-13,USW00025325,178.0,252.868,-74.868
```

```
Date,Station ID,PRCP,prediction,error
1935-05-19,USW00025325,183.0,188.011,-5.011
1935-05-20,USW00025325,107.0,68.629,38.371
1935-05-24,USW00025325,3.0,19.979,-16.979
1935-05-26,USW00025325,0.0,21.473,-21.473
1935-06-02,USW00025325,91.0,55.023,35.977
1935-06-09,USW00025325,246.0,198.084,47.916
1935-06-10,USW00025325,274.0,198.084,75.916
1935-06-11,USW00025325,53.0,54.715,-1.715
1935-06-13,USW00025325,69.0,54.681,14.319
```

Challenges Faced

- Main challenges we faced were during Q4 in deciding what models to use.
- We had to be really efficient with our process of selecting the features and the way we eliminate the data which is not important in our analysis.
- The model training really tested our patience as it took too much time and resources to train. We had to increase our heap space and other memory utilisations.
- Even then it took many hours to train the models and get the error predictions.