Spark: Assignment-6

Create a producer with a python connector in confluent kafka and stream your data.

```
from confluent_kafka import Producer
bootstrap_servers = 'your_kafka_broker_ip:port'       # Set Kafka broker details

producer_config = {                                   # Create producer
configuration
   'bootstrap.servers': bootstrap_servers
}

producer = Producer(producer_config)                  # Create a Kafka producer

topic = 'covid_infection_cases'                       # Define the topic to which you
want to send the data

data_file = 'path/to/your/data_file.csv'              # Specify the path to your local
data file
with open(data_file, 'r') as file:                    # Read data from local file
   lines = file.readlines()

for line in lines:                                    # Process each line and send it to
Kafka topic

   values = line.strip().split(',')                   # Assuming your data is
comma-separated and follows the specified column order
   record = {
      'case_id': int(values[0]),
      'province': values[1],
      'city': values[2],
      'group': bool(values[3]),
      'infection_case': values[4],
      'confirmed': values[5],
      'latitude': float(values[6]),
      'longitude': float(values[7])
   }
   producer.produce(topic=topic, value=record)
   producer.flush()

producer.close()                                      # Close the producer
```

Consume your data through the python connector and dump it in mongodb atlas.

```
pip install confluent-kafka                                    # Install the
required dependencies
pip install pymongo

from confluent_kafka import Consumer                           # Import the necessary
libraries
from pymongo import MongoClient


kafka_bootstrap_servers = 'your_kafka_bootstrap_servers'       # Configure
the Kafka consumer:
kafka_topic = 'your_kafka_topic'
kafka_consumer_group = 'your_kafka_consumer_group'
consumer_config = {
    'bootstrap.servers': kafka_bootstrap_servers,
    'group.id': kafka_consumer_group
}
consumer = Consumer(consumer_config)
consumer.subscribe([kafka_topic])

mongo_connection_string = 'your_mongodb_connection_string'     #
Configure the MongoDB connection
mongo_database = 'your_mongodb_database'
mongo_collection = 'your_mongodb_collection'
client = MongoClient(mongo_connection_string)
db = client[mongo_database]
collection = db[mongo_collection]


while True:                                                    #
Consume data from Kafka and insert into MongoDB
    message = consumer.poll(1.0)  # Poll for Kafka messages

    if message is None:
        continue

    if message.error():
        print(f"Consumer error: {message.error()}")
        continue
```

```
    kafka_value = message.value().decode('utf-8')  # Decode Kafka message value
    data = kafka_value.split(',')  # Assuming the data is comma-separated

    # Assuming the data order: case_id, province, city, group, infection_case,
confirmed, latitude, longitude
    document = {
        'case_id': data[0],
        'province': data[1],
        'city': data[2],
        'group': data[3],
        'infection_case': data[4],
        'confirmed': data[5],
        'latitude': data[6],
        'longitude': data[7]
    }

    collection.insert_one(document)  # Insert the document into MongoDB


consumer.close()  # Close the Kafka consumer                    # Close the Kafka
and MongoDB connections
client.close()  # Close the MongoDB connection
```

4. Collect your data as a pyspark dataframe and perform different operations.

Note: Consider only three files for creating a dataframe among all case, region and TimeProvince

a. Read the data, show it and Count the number of records.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Data Analysis").getOrCreate()

case_path = "path_to_case_file.csv"                    # Assuming the data files are
in CSV format and located in the specified paths
region_path = "path_to_region_file.csv"
time_province_path = "path_to_time_province_file.csv"

case_df = spark.read.csv(case_path, header=True, inferSchema=True)
        # Read the data files and create the DataFrames
region_df = spark.read.csv(region_path, header=True, inferSchema=True)
time_province_df = spark.read.csv(time_province_path, header=True,
inferSchema=True)
```

```
case_df.show()
region_df.show()
time_province_df.show()


case_count = case_df.count()
region_count = region_df.count()
time_province_count = time_province_df.count()

print("Number of records in 'case' DataFrame:", case_count)
print("Number of records in 'region' DataFrame:", region_count)
print("Number of records in 'time_province' DataFrame:", time_province_count)
```

b. Describe the data with a describe function.

```
case_df.describe().show()
region_df.describe().show()
time_province_df.describe().show()
```

The describe() function generates the summary statistics only for numeric columns. If you have non-numeric columns in your DataFrame, they will be excluded from the output.

c. If there is any duplicate value drop it.

```
case_df = case_df.dropDuplicates()
region_df = region_df.dropDuplicates()
time_province_df = time_province_df.dropDuplicates()
```

d. Use limit function for showcasing a limited number of records.

```
case_df.limit(5).show()
```

e. If you find the column name is not suitable, change the column name. [optional]

```
case_df = case_df.withColumnRenamed("old_column_name",
"new_column_name")
```

f. Select the subset of the columns.

```
subset_df = case_df.select("column1", "column2", "column3")
```

g. If there is any null value, fill it with any random value or drop it.

```
filled_df = case_df.fillna(“0”)
```

h. Filter the data based on different columns or variables and do the best analysis.

```
filtered_df = case_df.filter((case_df.confirmed > 100) & (case_df.province == 'Seoul'))
```

i. Sort the number of confirmed cases. Confirmed column is there in the dataset. Check with descending sort also.

```
sorted_df = case_df.orderBy('confirmed')
sorted_df_desc = case_df.orderBy(case_df.confirmed.desc())
```

j. In case of any wrong data type, cast that data type from integer to string or string to integer.

```
from pyspark.sql.functions import col

df = df.withColumn("column_1”, col("column_1").cast("integer"))               # Cast "column_name" from string to integer

df = df.withColumn("column_1", col("column_1").cast("string"))               # Cast "column_name" from integer to string
```

k. Use group by on top of province and city column and agg itwith sum of confirmed cases. For example : df.groupBy(["province","city"]).agg(function.sum("confirmed")

```
from pyspark.sql import functions as F

result = df.groupBy(["province", "city"]).agg(F.sum("confirmed").alias("total_confirmed"))          # Group by province and city, and aggregate with sum of confirmed cases

result.show()
                                # Show the result
```

I. For joins we will need one more file.you can use region file. User different different join methods.for example :
cases.join(regions, ['province','city'],how='left')

```
# Import necessary libraries
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder.getOrCreate()

# Read the data from files and create DataFrames
cases = spark.read.csv("cases.csv", header=True, inferSchema=True)
regions = spark.read.csv("regions.csv", header=True, inferSchema=True)

# Perform join using different join methods
inner_join = cases.join(regions, ['province', 'city'], how='inner')
left_join = cases.join(regions, ['province', 'city'], how='left')
right_join = cases.join(regions, ['province', 'city'], how='right')
full_outer_join = cases.join(regions, ['province', 'city'], how='full_outer')

# Show the results
inner_join.show()
left_join.show()
right_join.show()
full_outer_join.show()
```

Create Spark UDFs

Create function casehighlow()

If case is less than 50 return low else return high convert into a UDF (User Defined Function) function and mention the return type of function.
Note: You can create as many as udf based on analysis.

```
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

def casehighlow(case):                    # Define the UDF function
    if case < 50:
        return "low"
```

```
    else:
        return "high"

casehighlow_udf = udf(casehighlow, StringType())                    # Create the UDF

df = df.withColumn("case_category", casehighlow_udf("case"))                # Apply
the UDF to a DataFrame column

df.show()
```