```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
#Install Apache Spark
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!wget -q https://apache.osuosl.org/spark/spark-3.5.3/spark-3.5.3-bin-hadoop3.tgz
!tar xf spark-3.5.3-bin-hadoop3.tgz
!pip install -q findspark
!pip install -U airportsdata
```

```
Collecting airportsdata
  Downloading airportsdata-20241001-py3-none-any.whl.metadata (8.9 kB)
Downloading airportsdata-20241001-py3-none-any.whl (912 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 912.7/912.7 kB 14.9 MB/s eta 0:00:00
Installing collected packages: airportsdata
Successfully installed airportsdata-20241001
```

```
#Import Apache Spark
import os
import requests
import logging
```

```
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-3.5.3-bin-hadoop3"

import findspark
findspark.init()

from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Colab").getOrCreate()

# Try Spark Version
spark.range(5).show()
```

```
+---+
| id|
+---+
|  0|
|  1|
|  2|
|  3|
|  4|
+---+
```

Data Ingestion with PySpark on Google Colab for the ETL Pipeline

In this phase of our ETL pipeline, we focus on **data ingestion** using PySpark within the Google Colab environment. The goal of this step is to load the flight dataset, which will be used for further analysis and modeling, ensuring that the information is available in the appropriate format for processing and enrichment.

```
from pyspark.sql import SparkSession

# Start Spark session
spark = SparkSession.builder \
  .appName("FlightDelayAnalysis") \
  .config("spark.executor.memory", "4g") \
  .config("spark.driver.memory", "4g") \
  .config("spark.executor.cores", "2") \
  .getOrCreate()

def ingest_fligths_data(file_path):

  df_flights = spark.read.csv(file_path, header=True, inferSchema=True)

  df_flights.show(5)

  return df_flights
```

ENRICHMENT: Getting ICAO Code from Python Database airportsdata

In this part of the ETL process, we enrich our dataset by retrieving the **ICAO codes for airports** using the airportsdata Python library. This enrichment step is crucial for standardizing airport codes and ensuring consistency across our dataset.

```
from pyspark.sql.functions import udf
from pyspark.sql.types import FloatType, StringType
import airportsdata

# Call External API AirportDB
def get_airport_icao_code(airport_code):
  airports = airportsdata.load('IATA')  # key is the IATA location code
  return airports[airport_code]['icao']

# UDF Functions for Spark
```

```
get_airport_icao_code_udf = udf(get_airport_icao_code, StringType())

def enrich_with_airport_icao_data(df):
    df_enriched = df \
        .withColumn('icao_origin', get_airport_icao_code_udf(df['origin'])) \
        .withColumn("icao_dest", get_airport_icao_code_udf(df["dest"]))
    return df_enriched
```

ENRICHMENT: Getting day of the week by data

In this part of the ETL process, we enrich our dataset by retrieving the **day of the week** by using the datetime Python library. This enrichment step is crucial for using necessary APIs services and ensuring consistency across our dataset.

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

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Concatenate Date
def get_week_date(year, month, day):
    try:
        date = datetime.datetime(year, month, day)
        day_of_week_name = date.strftime("%A")  # Get the day of the week as a string
        logger.debug(f"Day of the week for {year}-{month}-{day}: {day_of_week_name}")

        return day_of_week_name
    except Exception as e:
        logger.error(f"Error concatenating date: {e}")
        return None

# UDF Functions for Spark
get_week_date_udf = udf(get_week_date, StringType())

def enrich_get_week_date(df):
    df_enriched = df.withColumn('week_date', get_week_date_udf(df['year'], df['month'], d
    return df_enriched
```

ENRICHMENT: Concatenating Date String

Process of enriching date information for integration with the Weatherbit API. The goal is to

concatenate date data into a format that is compatible with Weatherbit API.

1. **Date Formatting**

- **Input Format:** Separeted date format
- **Desired Output Format:** `YYYY-MM-DD` (e.g., `2024-10-04`)

2. **Concatenation Logic**

- Extract relevant date components:
  - Year: `YYYY`
  - Month: `MM`
  - Day: `DD`
- Concatenate components into a single string:

```
concatenated_date = f"{year}-{month}-{day}"
```

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

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Concatenate Date
def concatenate_flight_date(year, month, day):
    try:
        year = str(year).zfill(4)
        month = str(month).zfill(2)
        day = str(day).zfill(2)
        date = f"{year}-{month}-{day}"
        logger.debug(f"Concatenated date: {date}")
        return date
    except Exception as e:
        logger.error(f"Error concatenating date: {e}")
        return None

# UDF Functions for Spark
concatenate_flight_date_udf = udf(concatenate_flight_date, StringType())

def enrich_concatenate_flight_date(df):
    df_enriched = df.withColumn('flight_date', concatenate_flight_date_udf(df['year'], df
    return df_enriched
```

ENRICHMENT: Getting Airport Latitude and Longitude by using Airport DB API

This section details the process of enriching data by retrieving airport latitude and longitude from the Airport DB API, which will be utilized in Google Colab for further analysis or visualization.

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

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

api_token = "faeba3a4e3975301f56be55a2434e73eb9d3eb406108c58f7dd0e4915b70c3e446ca0889baf5

# Call External API AirportDB
def get_airport_coordinates(airport_code):

    url = f"https://airportdb.io/api/v1/airport/{airport_code}?apiToken={api_token}"
    try:
        response = requests.get(url)
        response.raise_for_status()
        data = response.json()
        latitude = data.get("latitude_deg")
        longitude = data.get("longitude_deg")
        logger.debug(f"Coordinates for {airport_code}: ({latitude}, {longitude})")
        return latitude, longitude
    except requests.RequestException as e:
        logger.error(f"Request failed for {airport_code}: {e}")
        return None, None

# UDF Functions for Spark
def get_latitude(airport_code):
    lat, lon = get_airport_coordinates(airport_code)
    logger.debug(f"Latitude for {airport_code}: {lat}")
    return lat

def get_longitude(airport_code):
    lat, lon = get_airport_coordinates(airport_code)
    logger.debug(f"Longitude for {airport_code}: {lon}")
    return lon

# Registering UDF Functions
get_latitude_udf = udf(get_latitude, FloatType())
get_longitude_udf = udf(get_longitude, FloatType())
```

```
def enrich_with_airport_coordinates_data(df):
    df_enriched = df \
        .withColumn('origin_latitude', get_latitude_udf(df['icao_origin'])) \
        .withColumn('origin_longitude', get_longitude_udf(df['icao_origin'])) \
        .withColumn("dest_latitude", get_latitude_udf(df["icao_dest"])) \
        .withColumn("dest_longitude", get_longitude_udf(df["icao_dest"]))
    return df_enriched
```

ENRICHMENT: Getting Airport Latitude and Longitude by using Python Database airportsdata

This section details the process of enriching data by retrieving airport latitude and longitude from the Python Database airportsdata, which will be utilized in Google Colab for further analysis or visualization.

```
from pyspark.sql.functions import udf
from pyspark.sql.types import FloatType
import airportsdata

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def get_airport_data(airport_code):
  airports = airportsdata.load('IATA')  # key is the IATA location code
  return airports[airport_code]

# UDF Functions for Spark
def get_latitude(airport_code):
    airport_data = get_airport_data(airport_code)
    logger.debug(f"Latitude for {airport_code}: {airport_data['lat']}")
    return airport_data['lat']

def get_longitude(airport_code):
    airport_data = get_airport_data(airport_code)
    logger.debug(f"Longitude for {airport_code}: {airport_data['lon']}")
    return airport_data['lon']

# Registering UDF Functions
get_latitude_udf = udf(get_latitude, FloatType())
get_longitude_udf = udf(get_longitude, FloatType())

def enrich_with_airport_coordinates_data_database(df):
    df_enriched = df \
        .withColumn('origin_latitude', get_latitude_udf(df['origin'])) \
```

```
        .withColumn('origin_longitude', get_longitude_udf(df['origin'])) \
        .withColumn("dest_latitude", get_latitude_udf(df["dest"])) \
        .withColumn("dest_longitude", get_longitude_udf(df["dest"]))
    return df_enriched
```

ENRICHMENT: Getting Wind Speed by using Weatherbit API

This section outlines the process of enriching data by retrieving wind speed from the Weatherbit API, to be utilized in Google Colab for further analysis or visualization.

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

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

api_key = "b22e5fc24c48474aa32a0edb1ea3fd60"

# Call External API Weatherbit
def get_wind_speed(lat, lon, date):

    #url = 'https://api.weatherbit.io/v2.0/history/agweather'
    #não foi utilizado dados históricos devido ao princing da ferramenta

    url = 'https://api.weatherbit.io/v2.0/current'

    params = {
        'lat': lat,
        'lon': lon,
        'key': api_key
    }
    try:
        response = requests.get(url, params=params)
        response.raise_for_status()
        data = response.json()
        wind_speed = data['data'][0]['wind_spd']
        logger.debug(f"Wind speed for ({lat}, {lon}) on {date}: {wind_speed}")
        return wind_speed
    except requests.RequestException as e:
        logger.error(f"Request failed for ({lat}, {lon}) on {date}: {e}")
        return None

# UDF Functions for Spark
get_wind_speed_udf = udf(get_wind_speed, FloatType())
```

```
get_wind_speed_udf = udf(get_wind_speed, FloatType())

def enrich_with_weather_data(df):
    df_enriched = df \
        .withColumn('wind_speed_origin', get_wind_speed_udf(df['origin_latitude'], df['or
        .withColumn('wind_speed_dest', get_wind_speed_udf(df['dest_latitude'], df['dest_l
    return df_enriched
```

ENRICHMENT: Getting random Wind Speed based on Beaufort Wind Scale

This section outlines the process of enriching data by retrieving wind speed randomly using Beaufort Wind Scale, to be utilized in Google Colab for further analysis or visualization.

- **0 (Calm):** < 1 km/h (0–0.3 m/s)
- **1 (Light air):** 1–5 km/h (0.3–1.5 m/s)
- **2 (Light breeze):** 6–11 km/h (1.6–3.3 m/s)
- **3 (Gentle breeze):** 12–19 km/h (3.4–5.4 m/s)
- **4 (Moderate breeze):** 20–28 km/h (5.5–7.9 m/s)
- **5 (Fresh breeze):** 29–38 km/h (8.0–10.7 m/s)
- **6 (Strong breeze):** 39–49 km/h (10.8–13.8 m/s)
- **7 (High wind, moderate gale, near gale):** 50–61 km/h (13.9–17.1 m/s)
- **8 (Gale, fresh gale):** 62–74 km/h (17.2–20.7 m/s)
- **9 (Strong gale):** 75–88 km/h (20.8–24.4 m/s)
- **10 (Storm, whole gale):** 89–102 km/h (24.5–28.4 m/s)
- **11 (Violent storm):** 103–117 km/h (28.5–32.6 m/s)
- **12 (Hurricane force):** > 118 km/h (>32.7 m/s)

```
from pyspark.sql.functions import udf
from pyspark.sql.types import FloatType
import random

def generate_random_beaufort():
    # Generate a random floating-point number between 0 and 12 with two decimal places
    beaufort_number = round(random.uniform(0, 12), 2)

    return beaufort_number

# UDF Functions for Spark
generate_random_beaufort_udf = udf(generate_random_beaufort, FloatType())

def enrich_with_weather_data_random_beaufort(df):
    df_enriched = df
```

```
    df_enriched = df \
        .withColumn('wind_speed_origin', generate_random_beaufort_udf()) \
        .withColumn('wind_speed_dest', generate_random_beaufort_udf())
    return df_enriched
```

ENRICHMENT: Transforming Arriving Delay Data in to Labels

This section outlines the process of enriching data by transforming arriving delay data in to labels, to be utilized in Google Colab for further analysis or visualization.

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

def get_arr_delay_label(arr_delay):
    # Handle None values (null in Spark DataFrame)
    if arr_delay is None:
        return 0
    elif arr_delay <= 0:
        return 1
    else:
        return 0

# UDF Functions for Spark
get_arr_delay_label_udf = udf(get_arr_delay_label, IntegerType())

def enrich_with_arr_delay_label(df):
    df_enriched = df \
        .withColumn('arr_delay_label', get_arr_delay_label_udf(df['arr_delay']))
    return df_enriched
```

ENRICHMENT: Transforming Departure Delay Data in to Labels

This section outlines the process of enriching data by transforming departure delay data in to labels, to be utilized in Google Colab for further analysis or visualization.

```
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType
import random

def get_dep_delay_label(dep_delay):
```

```
def get_dep_delay_label(dep_delay):
    # Handle None values (null in Spark DataFrame)
    if dep_delay is None:
        return 0
    elif dep_delay <= 0:
        return 1
    else:
        return 0

# UDF Functions for Spark
get_dep_delay_label_udf = udf(get_dep_delay_label, IntegerType())

def enrich_with_dep_delay_label(df):
    df_enriched = df \
        .withColumn('dep_delay_label', get_dep_delay_label_udf(df['dep_delay']))
    return df_enriched
```

Main Enrichment Job

Details the main enrichment job of the ETL process, focusing on enhancing the dataset by integrating external data sources and performing necessary transformations.

```
from pyspark.sql.functions import col, sum, concat_ws

#ingesting data
df_flights = ingest_fligths_data('drive/MyDrive/code/case-machine-learning-engineer-pleno

#enriching getting airport coordinates data on AirportDB API
#Not used due princing and API's limitation
#df_flights_enriched = enrich_with_airport_coordinates_data(df_flights_enriched)

#enriching getting weather/wind data on Weatherbit API
#Not used due princing and API's limitation
#df_flights_enriched = enrich_with_weather_data(df_flights_enriched)


#enriching getting ICAO airport code
df_flights_enriched = enrich_with_airport_icao_data(df_flights)

#enriching concatenating flight date
df_flights_enriched = enrich_concatenate_flight_date(df_flights_enriched)

#enriching getting week date
df_flights_enriched = enrich_get_week_date(df_flights_enriched)

#enriching concatenating flights as route
df_flights_enriched = df_flights_enriched.withColumn("route", concat_ws("-", df_flights_e
```

```
#enriching getting airport coordinates data on Python Database
df_flights_enriched = enrich_with_airport_coordinates_data_database(df_flights_enriched)

#enriching getting Departure Delay Label
df_flights_enriched = enrich_with_arr_delay_label(df_flights_enriched)

#enriching getting Arrive Delay Label
df_flights_enriched = enrich_with_dep_delay_label(df_flights_enriched)

#enriching getting Wind Speed based on random
df_flights_enriched = enrich_with_weather_data_random_beaufort(df_flights_enriched)

df_flights_enriched.show(5)
```

```
+---+----+-----+---+--------+--------------+---------+--------+--------------+-------
| id|year|month|day|dep_time|sched_dep_time|dep_delay|arr_time|sched_arr_time|arr_del
+---+----+-----+---+--------+--------------+---------+--------+--------------+-------
|  0|2013|    1|  1|   517.0|           515|      2.0|   830.0|           819|     11
|  1|2013|    1|  1|   533.0|           529|      4.0|   850.0|           830|     20
|  2|2013|    1|  1|   542.0|           540|      2.0|   923.0|           850|     33
|  3|2013|    1|  1|   544.0|           545|     -1.0|  1004.0|          1022|    -18
|  4|2013|    1|  1|   554.0|           600|     -6.0|   812.0|           837|    -25
+---+----+-----+---+--------+--------------+---------+--------+--------------+-------
only showing top 5 rows
```


```
+---+----+-----+---+--------+--------------+---------+--------+--------------+-------
| id|year|month|day|dep_time|sched_dep_time|dep_delay|arr_time|sched_arr_time|arr_del
+---+----+-----+---+--------+--------------+---------+--------+--------------+-------
|  0|2013|    1|  1|   517.0|           515|      2.0|   830.0|           819|     11
|  1|2013|    1|  1|   533.0|           529|      4.0|   850.0|           830|     20
|  2|2013|    1|  1|   542.0|           540|      2.0|   923.0|           850|     33
|  3|2013|    1|  1|   544.0|           545|     -1.0|  1004.0|          1022|    -18
|  4|2013|    1|  1|   554.0|           600|     -6.0|   812.0|           837|    -25
+---+----+-----+---+--------+--------------+---------+--------+--------------+-------
only showing top 5 rows
```


Processing Output Job

This section details the processing output job of the ETL pipeline, focusing on preparing and exporting the enriched data for further analysis or storage and answering all the questions.

1. Qual é o número total de voos no conjunto de dados?

```
total_flights = df_flights_enriched.count()
print(f"Total flights: {total_flights}")
```

```
Total flights: 336776
```

2. Quantos voos foram cancelados? (Considerando que voos cancelados têm dep_time e arr_time nulos)

```
cancelled_flights = df_flights_enriched.filter((df_flights_enriched.dep_time.isNull()) &
print(f"Cancelled Flights: {cancelled_flights}")
```

```
Cancelled Flights: 8255
```

3. Qual é o atraso médio na partida dos voos (dep_delay)?

```
from pyspark.sql.functions import avg

avg_dep_delay = df_flights_enriched.agg(avg("dep_delay")).first()[0]
print(f"Average Departure Delay: {avg_dep_delay} minutes")
```

```
Average Departure Delay: 12.639070257304708 minutes
```

4. Quais são os 5 aeroportos com maior número de pousos?

```
from pyspark.sql.functions import desc

top_5_airports = df_flights_enriched.groupBy("dest").count().orderBy(desc("count")).limit
top_5_airports.show()
```

```
+----+-----+
|dest|count|
+----+-----+
| ORD|17283|
| ATL|17215|
| LAX|16174|
| BOS|15508|
| MCO|14082|
+----+-----+
```

5. Qual é a rota mais frequente (par origin-dest)?

```
most_frequent_route = df_flights_enriched.groupBy("route").count().orderBy(desc("count"))
most_frequent_route.show()
```

```
+-------+-----+
|  route|count|
+-------+-----+
|JFK-LAX|11262|
+-------+-----+
```

6. Quais são as 5 companhias aéreas com maior tempo médio de atraso na chegada? (Exiba também o tempo)

```
top_5_carriers = df_flights_enriched.groupBy("carrier").agg(avg("arr_delay").alias("avg_a
top_5_carriers.show()
```

```
+-------+------------------+
|carrier|     avg_arr_delay|
+-------+------------------+
|     F9|21.920704845814978|
|     FL|20.115905511811025|
|     EV| 15.79643108710965|
|     YV|15.556985294117647|
|     OO|11.931034482758621|
+-------+------------------+
```

7. Qual é o dia da semana com maior número de voos?

```
top_day_of_week = df_flights_enriched.groupBy("week_date").count().orderBy(desc("count"))
top_day_of_week.show()
```

```
+---------+-----+
|week_date|count|
+---------+-----+
|   Monday|50690|
+---------+-----+
```

8. Qual o percentual mensal dos voos que tiveram atraso na partida superior a 30 minutos?

```
from pyspark.sql.functions import col

# Criar uma coluna "is_delayed" como 1 para atrasos > 30 min, e 0 caso contrário
df_flights_delays = df_flights_enriched.withColumn("is_delayed", col("dep_delay") > 30)

# Converter valores booleanos para inteiros
df_flights_delays = df_flights_delays.withColumn("is_delayed_int", col("is_delayed").cast

# Calcular o percentual de voos com atraso na partida maior que 30 minutos por mês
monthly_delays = df_flights_delays.groupBy("month").agg((avg("is_delayed_int").alias("per
monthly_delays.show()
```

```
+-----+-------------------+
|month|    percent_delayed|
+-----+-------------------+
|   12|0.17967539653264478|
|    1|0.12649624287278632|
|    3|0.15404139706145212|
|    4|0.16379871303593377|
|   10|0.09412626950057586|
|   11|0.08832994266691326|
|    2|0.13431827775432673|
|    6|0.20992142175222148|
|    5|0.15641270853256828|
|    9|0.08918958778851117|
|    8| 0.1469435872542561|
|    7| 0.2167105494119712|
+-----+-------------------+
```

9. Qual é a origem mais comum para voos que pousaram em Seattle (SEA)?

```
common_origin_for_sea = df_flights_enriched.filter(df_flights_enriched.dest == "SEA").gro
common_origin_for_sea.show()
```

```
+------+-----+
|origin|count|
+------+-----+
|   JFK| 2092|
+------+-----+
```

10. Qual é a média de atraso na partida dos voos (dep_delay) para cada dia da semana?

```
avg_delay_by_day_of_week = df_flights_enriched.groupBy("week_date").agg(avg("dep_delay").
avg_delay_by_day_of_week.show()
```

```
+---------+------------------+
|week_date|     avg_dep_delay|
+---------+------------------+
|Wednesday|11.803512219083876|
|  Tuesday|10.631682565455652|
|   Friday| 14.69605749486653|
| Thursday|16.148919990957108|
| Saturday| 7.650502333676133|
|   Monday|14.778936729330908|
|   Sunday|11.589531801152422|
+---------+------------------+
```

11. Qual é a rota que teve o maior tempo de voo médio (air_time)?

```
longest_avg_air_time_route = df_flights_enriched.groupBy("route").agg(avg("air_time").ali
longest_avg_air_time_route.show()
```

```
+-------+-----------------+
|  route|     avg_air_time|
+-------+-----------------+
|JFK-HNL|623.0877192982456|
+-------+-----------------+
```

12. Para cada aeroporto de origem, qual é o aeroporto de destino mais comum?

```
from pyspark.sql import Window
from pyspark.sql.functions import desc, row_number

# Create a window partitioned by origin and ordered by flight count in descending order
```

```
windowSpec = Window.partitionBy("origin").orderBy(desc("flight_count"))

# Add row number based on the partition
most_common_dest_per_origin = df_flights_enriched.groupBy("origin", "dest") \
    .count() \
    .withColumnRenamed("count", "flight_count") \
    .withColumn("rank", row_number().over(windowSpec)) \
    .filter("rank = 1")  # Keep only the top destination for each origin

# Show the result
most_common_dest_per_origin.select("origin", "dest", "flight_count").orderBy("origin").show
```

```
+------+----+------------+
|origin|dest|flight_count|
+------+----+------------+
|   EWR| ORD|        6100|
|   JFK| LAX|       11262|
|   LGA| ATL|       10263|
+------+----+------------+
```

13. Quais são as 3 rotas que tiveram a maior variação no tempo médio de voo (air_time)?

```
from pyspark.sql.functions import stddev

routes_with_highest_variation = df_flights_enriched.groupBy("route").agg(stddev("air_time
routes_with_highest_variation.show()
```

```
+-------+------------------+
|  route|      std_air_time|
+-------+------------------+
|LGA-MYR| 25.32455988429677|
|EWR-HNL| 21.26613546847427|
|JFK-HNL|20.688824842787056|
+-------+------------------+
```

14. Qual é a média de atraso na chegada para voos que tiveram atraso na partida superior a 1 hora?

```
avg_arr_delay_for_long_dep_delays = df_flights_enriched.filter(df_flights_enriched.dep_de
avg_arr_delay_for_long_dep_delays.show()
```

```
+------------------+
|     avg_arr_delay|
+------------------+
|119.04880549963919|
+------------------+
```

15. Qual é a média de voos diários para cada mês do ano?

```
daily_avg_flights_per_month = df_flights_enriched.groupBy("month", "day").count().groupBy
daily_avg_flights_per_month.show()
```

```
+-----+-----------------+
|month|avg_daily_flights|
+-----+-----------------+
|   12|907.5806451612904|
|    1|871.0967741935484|
|    6|941.4333333333333|
|    3|930.1290322580645|
|    5|928.9032258064516|
|    9|919.1333333333333|
|    4|944.3333333333334|
|    8|946.0322580645161|
|    7|949.1935483870968|
|   10|931.9032258064516|
|   11|908.9333333333333|
|    2|891.1071428571429|
+-----+-----------------+
```

16. Quais são as 3 rotas mais comuns que tiveram atrasos na chegada superiores a 30 minutos?

```
most_common_routes_with_long_arrival_delay = df_flights_enriched.filter(df_flights_enrich
most_common_routes_with_long_arrival_delay.show()
```

```
+-------+-----+
|  route|count|
+-------+-----+
|LGA-ATL| 1563|
|JFK-LAX| 1286|
|LGA-ORD| 1188|
+-------+-----+
```

17. Para cada origem, qual o principal destino?

```
from pyspark.sql import Window
from pyspark.sql.functions import col, count, row_number

# Repartition the DataFrame based on the "origin" column to avoid skew
df_flight_counts = df_flights_enriched.repartition("origin").groupBy("origin", "dest").ag

# Define a window partitioned by 'origin', ordering by flight count (most frequent destin
window = Window.partitionBy("origin").orderBy(col("flight_count").desc())

# Cache the intermediate DataFrame to avoid recomputation
df_flight_counts.cache()

# Use row_number() to rank destinations for each origin and select the most frequent dest
df_top_destinations = df_flight_counts.withColumn("row", row_number().over(window)) \
                                      .filter(col("row") == 1) \
                                      .select("origin", "dest")

# Show the most frequent destination for each origin
df_top_destinations.show(5)
df_flight_counts.unpersist()
```

```
+------+----+
|origin|dest|
+------+----+
|   LGA| ATL|
|   EWR| ORD|
|   JFK| LAX|
+------+----+

DataFrame[origin: string, dest: string, flight_count: bigint]
```

Pergunta final: Enriqueça a base de dados de voos com as condições meteorológicas (velocidade do vento) para os aeroportos de origem e destino. Mostre as informações enriquecidas para os 5 voos com maior atraso na chegada.

```
# Selecting the top 5 flights with the largest arrival delay
df_top_5_arr_delay = df_flights_enriched.orderBy(col("arr_delay").desc()).limit(5)

# Display the information for the top 5 flights with the largest arrival delays, including
```

```
df_top_5_arr_delay.select("origin", "dest", "arr_delay", "wind_speed_origin", "wind_speed_d
```

```
+------+----+---------+-----------------+---------------+
|origin|dest|arr_delay|wind_speed_origin|wind_speed_dest|
+------+----+---------+-----------------+---------------+
|   JFK| HNL|   1272.0|             3.36|           3.36|
|   JFK| CMH|   1127.0|            10.43|          10.43|
|   EWR| ORD|   1109.0|             2.78|           2.78|
|   JFK| SFO|   1007.0|              5.1|            5.1|
|   JFK| CVG|    989.0|             3.57|           3.57|
+------+----+---------+-----------------+---------------+
```

Machine Learning Modeling Phase for Flight Delay Prediction

In this phase, we will focus on the **machine learning modeling** process for predicting flight delays. The goal is to create a model that can predict the **arrival delay** based on various features from the flight dataset, including the flight's departure delay, weather conditions (such as wind speed), and other relevant factors.

Remove rows where any column has a null value

```
# Remove rows where any column has a null value
df_flights_enriched = df_flights_enriched.filter(df_flights_enriched['dep_delay'].isNotNu

df_flights_enriched.show(5)
```

```
+---+----+-----+---+--------+-------------+---------+--------+-------------+-------
| id|year|month|day|dep_time|sched_dep_time|dep_delay|arr_time|sched_arr_time|arr_del
+---+----+-----+---+--------+-------------+---------+--------+-------------+-------
|  0|2013|    1|  1|   517.0|          515|      2.0|   830.0|          819|     11
|  1|2013|    1|  1|   533.0|          529|      4.0|   850.0|          830|     20
|  2|2013|    1|  1|   542.0|          540|      2.0|   923.0|          850|     33
|  3|2013|    1|  1|   544.0|          545|     -1.0|  1004.0|         1022|    -18
|  4|2013|    1|  1|   554.0|          600|     -6.0|   812.0|          837|    -25
+---+----+-----+---+--------+-------------+---------+--------+-------------+-------
only showing top 5 rows
```

Feature Engineering: Standardizing the Features

Standardize the features before applying PCA. This is done using StandardScaler.

```
#df_sampled = df_flights_enriched.sample(fraction=0.0001, seed=42).repartition(200)

#row_count = df_sampled.count()

# Print the row count
#print(f"Total number of rows: {row_count}")


# Import necessary libraries
#from pyspark.ml.feature import VectorAssembler, StandardScaler

# Sample the data (adjust the fraction as needed)
#df_sampled = df_flights_enriched.sample(fraction=0.001, seed=42)

# Create the feature vector
#assembler = VectorAssembler(inputCols=["dep_delay_label", "wind_speed_origin", "wind_spe
                            # outputCol="features_vector")
#df_flights_vector = assembler.transform(df_sampled)

# Standardize the features
#scaler = StandardScaler(inputCol="features_vector", outputCol="scaled_features", withMea
#scaler_model = scaler.fit(df_flights_vector)
#df_scaled = scaler_model.transform(df_flights_vector)

# Cache the DataFrame if needed
#df_scaled.cache()

# Show the results
#df_scaled.select("features_vector", "scaled_features").show(5)

# Unpersist the DataFrame when no longer needed
#df_scaled.unpersist()
```

Feature Engineering: Applying PCA (Principal Component Analysis)

Once the data is standardized, PCA has been applied to reduce the number of features while keeping most of the variance.

```
#from pyspark.ml.feature import PCA

# Initialize PCA model to reduce to 2 principal components
#pca = PCA(k=2, inputCol="scaled_features", outputCol="pca_features")

# Fit the PCA model and transform the data (repartition to optimize PCA)
#df_pca = pca.fit(df_scaled.repartition(100)).transform(df_scaled)
```

```
# Cache the result of PCA transformation to reuse if needed
#df_pca.cache()

# Show the resulting principal components
#df_pca.select("pca_features").show(5, truncate=False)

#df_pca.unpersist()
```

Modelling: Spliting Data

In the **Modelling: Splitting Data** phase, we divide our dataset into two main parts: **training** and **testing** data. This is a critical step in machine learning because it helps evaluate how well a model generalizes to unseen data.

```
from pyspark.ml.feature import StringIndexer, VectorAssembler

# Combine features and target in one DataFrame
df_combined = df_flights_enriched.select("dep_delay_label", "wind_speed_origin", "wind_sp

# Split the combined DataFrame into training and testing sets (80% train, 20% test)
train_df, test_df = df_combined.randomSplit([0.8, 0.2])

# Separate the features and target for the training set
X_train = train_df.select("dep_delay_label", "wind_speed_origin", "wind_speed_dest", "dis
Y_train = train_df.select("arr_delay_label")

# Separate the features and target for the testing set
X_test = test_df.select("dep_delay_label", "wind_speed_origin", "wind_speed_dest", "dista
Y_test = test_df.select("arr_delay_label")

# Index the 'route' column (convert from string to numeric)
indexer = StringIndexer(inputCol="route", outputCol="route_index")

df_flights_indexed = indexer.fit(df_flights_enriched).transform(df_flights_enriched)

# Assemble the features into a single vector (including the indexed 'route')
assembler = VectorAssembler(
    inputCols=["dep_delay_label", "wind_speed_origin", "wind_speed_dest", "distance", "ro
    outputCol="features"
)

# Apply the assembler to create a new DataFrame with features vector
df_combined = assembler.transform(df_flights_indexed)

# Select the relevant columns (features and target)
df_combined = df_combined.select("features", "arr_delay_label")
```

```
df_combined = df_combined.select("features", "arr_delay_label")

# Split the data into training and testing sets (80% training, 20% testing)
train_df, test_df = df_combined.randomSplit([0.8, 0.2], seed=42)
```

Modelling: Model Selection

```
from pyspark.ml.regression import LinearRegression

# Initialize the Linear Regression model
lr = LinearRegression(featuresCol='features', labelCol='arr_delay_label')
```

Modelling: Model Training & Prediction

```
# Train the model on the training data
lr_model = lr.fit(train_df)

# Make predictions on the test data
predictions = lr_model.transform(test_df)
```

Modelling: Model Evaluation

```
from pyspark.ml.evaluation import RegressionEvaluator

# Evaluate the model using RMSE and R-squared
evaluator_rmse = RegressionEvaluator(labelCol="arr_delay_label", predictionCol="predictio
rmse = evaluator_rmse.evaluate(predictions)

evaluator_r2 = RegressionEvaluator(labelCol="arr_delay_label", predictionCol="prediction"
r2 = evaluator_r2.evaluate(predictions)

# Print the evaluation metrics
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"R-squared (R2): {r2}")

# Optionally, print the model coefficients and intercept
print(f"Coefficients: {lr_model.coefficients}")
print(f"Intercept: {lr_model.intercept}")
```

```
Root Mean Squared Error (RMSE): 0.42141717514824234
R-squared (R2): 0.26829220233148443
Coefficients: [0.5192592416093328,0.0001741561219753016,0.0001741561219753016,3.03279
Intercept: 0.24335139446500...
```

```
Intercept: 0.24325120446592291
```

Modelling: Exporting Pickle file

```
# Extract the coefficients and intercept of the Linear Regression model
model_data = {
    "coefficients": lr_model.coefficients.toArray(),  # Convert to array
    "intercept": lr_model.intercept
}

# Save the model data to a .pkl file
import pickle
with open('linear_regression_model_flight_delay_prediction.pkl', 'wb') as f:
    pickle.dump(model_data, f)

print("Model data has been saved to 'linear_regression_model_flight_delay_prediction.pkl'
```

```
Model data has been saved to 'linear_regression_model_flight_delay_prediction.pkl'
```

Modelling: Testing the model .pkl file

```
import numpy as np
import pickle

# Load the model data from the .pkl file
with open('linear_regression_model_flight_delay_prediction.pkl', 'rb') as f:
    model_data = pickle.load(f)

# Function to manually apply the linear regression formula: y = X * coefficients + interc
def manual_predict(features, coefficients, intercept):
    return np.dot(features, coefficients) + intercept

# Example: Making a prediction with a sample feature vector
sample_features = np.array([0.5, 1.2, 0.3, 100, 2])

prediction = manual_predict(sample_features, model_data["coefficients"], model_data["inte
print(f"Prediction: {prediction}")
```

```
Prediction: 0.5060890064144823
```