Data Science Project - Phase2

Amirhossein Arefzadeh 810101604

Amin Aghakasiri 810101381

Aria Azem 810101608

Database Implementation and Data Querying

We have three csv files for our dataset. The main one that we work on is "flights.csv" which has different information about the flights in the USA. Also we have "airports.csv" which has the information of the airports that appeared in the flights dataset. At the end, we have "airlines.csv" which has the information about the airlines that appeared in the flights dataset. Now we want to make our database for these datasets. Because the project is entirely local, does not require concurrent users, and must be portable, SQLite is selected. It lives in one file, needs no server process, yet still supports foreign-key constraintsFirst. We will create the tables showing proper relations between them by setting primary keys and foreign keys correctly. First, we read the csv files into pandas dataframes like below:

```python
airports_df  = pd.read_csv(airports_csv)
airlines_df  = pd.read_csv(airlines_csv)

airports_df = airports_df[["IATA_CODE", "AIRPORT", "CITY", "A_STATE",
                           "COUNTRY", "LATITUDE", "LONGITUDE"]].drop_duplicates()
airlines_df = airlines_df[["IATA_CODE", "AIRLINE"]].drop_duplicates()
```

✓ 0.0s Python

```python
flights_df = pd.read_csv(flights_csv)

flights_df = flights_df[["A_YEAR", "A_MONTH","A_DAY","DAY_OF_WEEK","AIRLINE","FLIGHT_NUMBER",
                         "TAIL_NUMBER", "ORIGIN_AIRPORT",
                         "DESTINATION_AIRPORT","SCHEDULED_DEPARTURE", "DEPARTURE_TIME", "DEPARTURE_DELAY", "TAXI_OUT",
                           "WHEELS_OFF", "SCHEDULED_TIME", "ELAPSED_TIME", "AIR_TIME", "DISTANCE", "WHEELS_ON", "TAXI_IN",
                         "SCHEDULED_ARRIVAL", "ARRIVAL_TIME", "ARRIVAL_DELAY", "DIVERTED", "CANCELLED",
                         "CANCELLATION_REASON", "AIR_SYSTEM_DELAY", "SECURITY_DELAY", "AIRLINE_DELAY",
                         "LATE_AIRCRAFT_DELAY", "WEATHER_DELAY"]].drop_duplicates()
flights_df.insert(0, "flight_id",  range(1, len(flights_df)+1))
```

✓ 0.7s Python

After that, it is time to connect to the database and build the tables.

```python
con = sqlite3.connect(db_path)
con.execute("PRAGMA foreign_keys = ON;")
cur = con.cursor()

for tbl in ("flights", "airports", "airlines"):
    cur.executescript(f"DROP TABLE IF EXISTS {tbl};")
```

The airlines table has the information about different airlines that are present in the flights table. The iata code is unique in this table and thus is the key. Also the airports table has the information about airports and again each iata code for each airport is unique. The last table is the flights table which has different information about flights. The iata codes for origin airport and destination airport are present in the airports table so we can join them based on this attribute later in queries. Also the iata codes for airlines are present in the airline table. Also the table has foreign keys that one of them references to iata code for airlines and the other references to iata code for airports. Below is the code for creating tables:

```
cur.executescript("""
CREATE TABLE flights(
    flight_id INT PRIMARY KEY,
    A_YEAR INT,
    A_MONTH INT,
    A_DAY INT,
    DAY_OF_WEEK INT,
    AIRLINE TEXT,
    FLIGHT_NUMBER INT,
    TAIL_NUMBER TEXT,
    ORIGIN_AIRPORT TEXT,
    DESTINATION_AIRPORT TEXT,
    SCHEDULED_DEPARTURE INT,
    DEPARTURE_TIME INT,
    DEPARTURE_DELAY INT,
    TAXI_OUT INT,
    WHEELS_OFF INT,
    SCHEDULED_TIME INT,
    ELAPSED_TIME INT,
    AIR_TIME INT,
    DISTANCE INT,
    WHEELS_ON INT,
    TAXI_IN INT,
    SCHEDULED_ARRIVAL INT,
    ARRIVAL_TIME INT,
    ARRIVAL_DELAY INT,
    DIVERTED INT,
    CANCELLED INT,
    CANCELLATION_REASON TEXT,
    AIR_SYSTEM_DELAY INT,
    SECURITY_DELAY INT,
    AIRLINE_DELAY INT,
    LATE_AIRCRAFT_DELAY INT,
    WEATHER_DELAY INT,

    FOREIGN KEY (ORIGIN_AIRPORT) REFERENCES airports(IATA_CODE),
    FOREIGN KEY (DESTINATION_AIRPORT) REFERENCES airports(IATA_CODE),
    FOREIGN KEY (AIRLINE) REFERENCES airlines(IATA_CODE)
);
""")
con.commit()
```

```
cur.executescript("""
CREATE TABLE airports(
    IATA_CODE TEXT PRIMARY KEY,
    AIRPORT TEXT,
    CITY TEXT,
    A_STATE TEXT,
    COUNTRY TEXT,
    LATITUDE REAL,
    LONGITUDE REAL
);
""")

cur.executescript("""
CREATE TABLE airlines(
    IATA_CODE    TEXT PRIMARY KEY,
    AIRLINE      TEXT
);
""")
```

At last, we will convert the df's to sql and make the tables:

```python
airports_df.to_sql("airports", con, if_exists="append", index=False)
airlines_df.to_sql("airlines", con, if_exists="append", index=False)
flights_df.to_sql("flights",  con, if_exists="append", index=False)

con.commit()
con.close()
print("SQLite database built at", db_path)
```

✓ 1.9s Python

```
SQLite database built at Database/flight_data.db
```

Now that the database and the tables and the relations between them are set correctly, it is time to execute some queries. First, we will execute "Select *" query for each of the tables (run_sql() is a function that we wrote for printing the queries):

```python
run_sql("""
SELECT *
FROM flights
""")
```

✓ 2.9s Python

	flight_id	A_YEAR	A_MONTH	A_DAY	DAY_OF_WEEK	AIRLINE	FLIGHT_NUMBER	TAIL_NUMBER	ORIGIN_AIRPORT	DESTINATION_AIRPORT	...	ARRIVAL_TIME	ARRIVAL_[
0	1	2015	1	1	4	AS	98	N407AS	ANC	SEA	...	408.0	
1	2	2015	1	1	4	AA	2336	N3KUAA	LAX	PBI	...	741.0	
2	3	2015	1	1	4	US	840	N171US	SFO	CLT	...	811.0	
3	4	2015	1	1	4	AA	258	N3HYAA	LAX	MIA	...	756.0	
4	5	2015	1	1	4	AS	135	N527AS	SEA	ANC	...	259.0	
5	6	2015	1	1	4	DL	806	N3730B	SFO	MSP	...	610.0	
...	...	...	...	...	...	...	...	...	...	...	...	...	
499997	499998	2015	2	3	2	WN	371	N438WN	MDW	PIT	...	NaN	
499998	499999	2015	2	3	2	WN	1267	N211WN	MDW	TPA	...	927.0	
499999	500000	2015	2	3	2	WN	136	N352SW	MSP	MKE	...	702.0	

10 rows × 32 columns

500,000 rows total

```python
run_sql("""
SELECT *
FROM airports
""")
```

✓ 0.0s Python

	IATA_CODE	AIRPORT	CITY	A_STATE	COUNTRY	LATITUDE	LONGITUDE
0	ABE	Lehigh Valley International Airport	Allentown	PA	USA	40.65236	-75.4404
1	ABI	Abilene Regional Airport	Abilene	TX	USA	32.41132	-99.6819
2	ABQ	Albuquerque International Sunport	Albuquerque	NM	USA	35.04022	-106.60919
3	ABR	Aberdeen Regional Airport	Aberdeen	SD	USA	45.44906	-98.42183
4	ABY	Southwest Georgia Regional Airport	Albany	GA	USA	31.53552	-84.19447
5	ACK	Nantucket Memorial Airport	Nantucket	MA	USA	41.25305	-70.06018
...	...	...	...	...	...	...	...
319	XNA	Northwest Arkansas Regional Airport	Fayetteville/Springdale/Rogers	AR	USA	36.28187	-94.30681
320	YAK	Yakutat Airport	Yakutat	AK	USA	59.50336	-139.66023
321	YUM	Yuma International Airport	Yuma	AZ	USA	32.65658	-114.60597

322 rows total

```python
run_sql("""
SELECT *
FROM airlines
""")
```

✓ 0.0s Python

	IATA_CODE	AIRLINE
0	UA	United Air Lines Inc.
1	AA	American Airlines Inc.
2	US	US Airways Inc.
3	F9	Frontier Airlines Inc.
4	B6	JetBlue Airways
5	OO	Skywest Airlines Inc.
6	AS	Alaska Airlines Inc.
7	NK	Spirit Air Lines
8	WN	Southwest Airlines Co.
9	DL	Delta Air Lines Inc.
10	EV	Atlantic Southeast Airlines
11	HA	Hawaiian Airlines Inc.
12	MQ	American Eagle Airlines Inc.
13	VX	Virgin America

14 rows total

Now it's time to write some more informative queries. Below query shows the average departure delay for each airline:

```python
run_sql("""
SELECT F.AIRLINE, AVG(F.DEPARTURE_DELAY) AS Delay
FROM flights F
GROUP BY F.AIRLINE;
""")
```

✓ 0.1s Python

	AIRLINE	Delay
0	AA	10.643729
1	AS	3.430815
2	B6	10.774558
3	DL	6.775430
4	EV	10.043026
5	F9	19.664576
6	HA	1.109583
7	MQ	16.141276
8	NK	14.016792
9	OO	12.458124
10	UA	14.169355
11	US	6.108520
12	VX	6.815032
13	WN	9.631456

14 rows total

Below query shows the number of flights from each airline:

```
run_sql("""
SELECT F.AIRLINE, COUNT(*) AS NumOfFlights
FROM flights F
GROUP BY F.AIRLINE;
""")
```

✓ 0.1s Python

	AIRLINE	NumOfFlights
0	AA	46950
1	AS	14149
2	B6	23062
3	DL	68555
4	EV	52965
5	F9	7291
6	HA	6858
7	MQ	31896
8	NK	9324
9	OO	51184
10	UA	40873
11	US	35591
12	VX	5049
13	WN	106253

14 rows total

Below query shows the number of cancelled flights per airline:

```
run_sql("""
SELECT F.AIRLINE, SUM(F.CANCELLED) AS Cancelled
FROM flights F
GROUP BY F.AIRLINE;
""")
```

✓ 0.1s Python

	AIRLINE	Cancelled
0	AA	1324
1	AS	83
2	B6	1479
3	DL	938
4	EV	2523
5	F9	122
6	HA	27
7	MQ	3136
8	NK	158
9	OO	1623
10	UA	1424
11	US	1268
12	VX	115
13	WN	2604

14 rows total

Below query shows the average fly time between two specific airports:

```
run_sql("""
SELECT F.ORIGIN_AIRPORT, F.DESTINATION_AIRPORT, AVG(F.AIR_TIME) AS AVG_FLY_TIME
FROM flights F
GROUP BY F.ORIGIN_AIRPORT, F.DESTINATION_AIRPORT;
""")
```

✓ 0.2s Python

	ORIGIN_AIRPORT	DESTINATION_AIRPORT	AVG_FLY_TIME
0	ABE	ATL	111.540541
1	ABE	DTW	83.267606
2	ABE	ORD	112.263158
3	ABI	DFW	32.394191
4	ABQ	ATL	150.263158
5	ABQ	BWI	192.121212
...	...	...	...
4177	YAK	CDV	36.84375
4178	YAK	JNU	35.133333
4179	YUM	PHX	32.188571

4,180 rows total

Below query shows the different information about the origin and the destination airport by joining the flights table and airports table:

```
run_sql("""
SELECT f1.flight_id AS FLIGHT_ID, f1.ORIGIN_AIRPORT, a1.A_STATE AS ORIGIN_AIRPORT_STATE, a1.AIRPORT AS ORIGIN_AIRPORT_FULL_NAME,
 f1.DESTINATION_AIRPORT, a2.A_STATE as DESTINATION_AIRPORT_STATE, a2.AIRPORT AS DESTINATION_AIRPORT_FULL_NAME
FROM flights f1, airports a1, airports a2
WHERE f1.ORIGIN_AIRPORT = a1.IATA_CODE AND f1.DESTINATION_AIRPORT = a2.IATA_CODE;
""")
```

✓ 0.9s Python

	FLIGHT_ID	ORIGIN_AIRPORT	ORIGIN_AIRPORT_STATE	ORIGIN_AIRPORT_FULL_NAME	DESTINATION_AIRPORT	DESTINATION_AIRPORT_STATE	DESTINATION_AIRPORT_FULL_N
0	1	ANC	AK	Ted Stevens Anchorage International Airport	SEA	WA	Seattle-Tacoma International Ai
1	2	LAX	CA	Los Angeles International Airport	PBI	FL	Palm Beach International Ai
2	3	SFO	CA	San Francisco International Airport	CLT	NC	Charlotte Douglas International Ai
3	4	LAX	CA	Los Angeles International Airport	MIA	FL	Miami International Ai
4	5	SEA	WA	Seattle-Tacoma International Airport	ANC	AK	Ted Stevens Anchorage Internat Ai
5	6	SFO	CA	San Francisco International Airport	MSP	MN	Minneapolis-Saint Paul Internat Ai
...	...	...	...	...	...	...	
499997	499998	MDW	IL	Chicago Midway International Airport	PIT	PA	Pittsburgh International Ai
499998	499999	MDW	IL	Chicago Midway International Airport	TPA	FL	Tampa International Ai
499999	500000	MSP	MN	Minneapolis-Saint Paul International Airport	MKE	WI	General Mitchell International Ai

500,000 rows total

For the final query, it shows the complete airline name of each flight by joining the flights table and airlines table:

```python
run_sql("""
SELECT f.flight_id AS FLIGHT_ID, f.AIRLINE AS IATA_CODE, a.AIRLINE AS AIRLINE_NAME
FROM flights f, airlines a
WHERE f.AIRLINE = a.IATA_CODE;
""")
```

✓ 0.4s Python

	FLIGHT_ID	IATA_CODE	AIRLINE_NAME
0	1	AS	Alaska Airlines Inc.
1	2	AA	American Airlines Inc.
2	3	US	US Airways Inc.
3	4	AA	American Airlines Inc.
4	5	AS	Alaska Airlines Inc.
5	6	DL	Delta Air Lines Inc.
...	...	...	...
499997	499998	WN	Southwest Airlines Co.
499998	499999	WN	Southwest Airlines Co.
499999	500000	WN	Southwest Airlines Co.

500,000 rows total

Feature Engineering, Data Preprocessing, and Preparation for Modeling

At first, we load the three data frames into flights, airports and airlines:

```
def load_data():
    con = connect_to_db()
    flights = pd.read_sql_query("SELECT * FROM flights", con)
    airports = pd.read_sql_query("SELECT * FROM airports", con)
    airlines = pd.read_sql_query("SELECT * FROM airlines", con)
    con.commit()
    con.close()
    return flights, airports, airlines
```

Then we do feature engineering. We will add some features(columns) in this part. One important parameter that can be useful in analysis is **distance** between origin and destination. We have the latitude and longitude of all airports in the airports data frame. So, we add these two features for origin and destination airports then we calculate the distance in kilometers.

```
def haversine(lat1, lon1, lat2, lon2):
    lat1, lon1, lat2, lon2 = map(np.radians, (lat1, lon1, lat2, lon2))
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = np.sin(dlat/2)**2 + np.cos(lat1)*np.cos(lat2)*np.sin(dlon/2)**2
    return 2 * 6371 * np.arcsin(np.sqrt(a))

flights['DISTANCE_KM'] = haversine(
    flights['ORIG_LATITUDE'], flights['ORIG_LONGITUDE'],
    flights['DEST_LATITUDE'], flights['DEST_LONGITUDE']
)
```

Another change we make is to format the date into dd-mm-yy (pandas format) instead of separate columns for day, month and year. A new feature for analysis is **IS_WEEKEND** column to analyze the effect of holidays and weekends on flight delays. We also add some time relevant columns like day of year, week of year and … to analyze the effect of these on flight delays. Another feature that may be useful is Time_bucket. This feature bins the day into 4 parts and tells the bin which flight time is:

```python
# Add date column:
flights['FLIGHT_DATE'] = pd.to_datetime(
    flights[['A_YEAR','A_MONTH','A_DAY']].rename(
        columns={'A_YEAR':'year','A_MONTH':'month','A_DAY':'day'}),
    format="%Y-%m-%d"
)

flights['DAY_OF_YEAR'] = flights['FLIGHT_DATE'].dt.dayofyear
flights['WEEK_OF_YEAR'] = flights['FLIGHT_DATE'].dt.isocalendar().week
flights['QUARTER'] = flights['FLIGHT_DATE'].dt.quarter
flights['IS_WEEKEND'] = flights['FLIGHT_DATE'].dt.weekday >= 5

flights['DEP_HOUR'] = (flights['SCHEDULED_DEPARTURE'] // 100).astype(int)
bins = [0, 6, 12, 18, 24]
labels = ['early_morning','morning','afternoon','evening']
flights['DEP_TIME_BUCKET'] = pd.cut(flights['DEP_HOUR'], bins=bins, labels=labels, right=False)
```

We use a time rolling feature. In this feature we calculate the mean delay of seven previous flights of that airline. This may be useful in delay analysis as long as recent flights of the airline may affect the delay of current flight:

```python
flights['DISTANCE_KM'] = haversine(
    flights['ORIG_LATITUDE'], flights['ORIG_LONGITUDE'],
    flights['DEST_LATITUDE'], flights['DEST_LONGITUDE']
)

# Calculating 7 previous flights avg delay:
flights = flights.sort_values(["AIRLINE","FLIGHT_DATE"])
flights["ARR_DELAY_FILLED"] = flights["ARRIVAL_DELAY"].fillna(0)

flights["AIRLINE_7D_MEAN"] = (
    flights
    .groupby("AIRLINE")["ARR_DELAY_FILLED"]
    .rolling(window=7, min_periods=1)
    .mean()
    .reset_index(level=0,drop=True)
)
```

For further analysis of distance effect on flight delay, we add a new feature which shows the delay of flight per kilometer. (delay / distance) At the end, we delete all unnecessary columns like IDs and tail numbers etc.

In the processing part, we handle the missing data and then we standardize the numeric value by sklearn library. We use one-hot encoding for some categorical columns, but we leave some other columns unchanged.

- These are the numeric values which we standardize:

```python
num_feats = ['DISTANCE_KM', 'AIRLINE_7D_MEAN','DEP_HOUR','DELAY_PER_KM','DEPARTURE_DELAY', 'SCHEDULED_DEPARTURE',
            'ARRIVAL_DELAY', 'TAXI_OUT', 'TAXI_IN', 'AIR_TIME', 'ELAPSED_TIME']
```

- These are the columns(features) which we use ordinal encode for:

```
cat_feats = ['DAY_OF_WEEK', 'WEEK_OF_YEAR', 'IS_WEEKEND']
```

- We also leave some features unchanged because encoding them doest help us. For instance, origin and destination airports and date remain unchanged:

```
unchanged_columns = ['AIRLINE', 'ORIGIN_AIRPORT', 'DESTINATION_AIRPORT', 'FLIGHT_DATE']
```

Now it's time to manage missing data. Some time-related numeric values are missing due to cancellation of the flight. So, we treat all of these cells as zeros. If these missing values have some other reasons, we put -1 in that cell to ignore them in our calculations. Then for other missing data we put median of that column in the cells.

Pipeline and Separate Files

As said in the project description, we separate codes for each part in the related python file in the Scripts folder. Then we add these files into the pipeline.py. For executing the program it is enough to run the pipeline.py. In the README.me is explained how to install requirements. By running the given command, all the libraries used in the project will be installed on the user's computer. Also all python files are briefly explained in the README.me. Below is the README.me in github:

README.md

Data Pipeline

Overview

This project processes and transforms data in a multi-step pipeline.

Steps

1. `import_to_db.py` : Imports the data into SQLite.
2. `database_connection.py` : Connect to database.
3. `load_data.py` : Loads and inspects the raw data.
4. `preprocess.py` : Cleans and standardizes the data.
5. `feature_eng.py` : Adds derived features.

How to Run

1. Install dependencies:

```
pip install -r requirements.txt
```

2. Execute the entire pipeline:

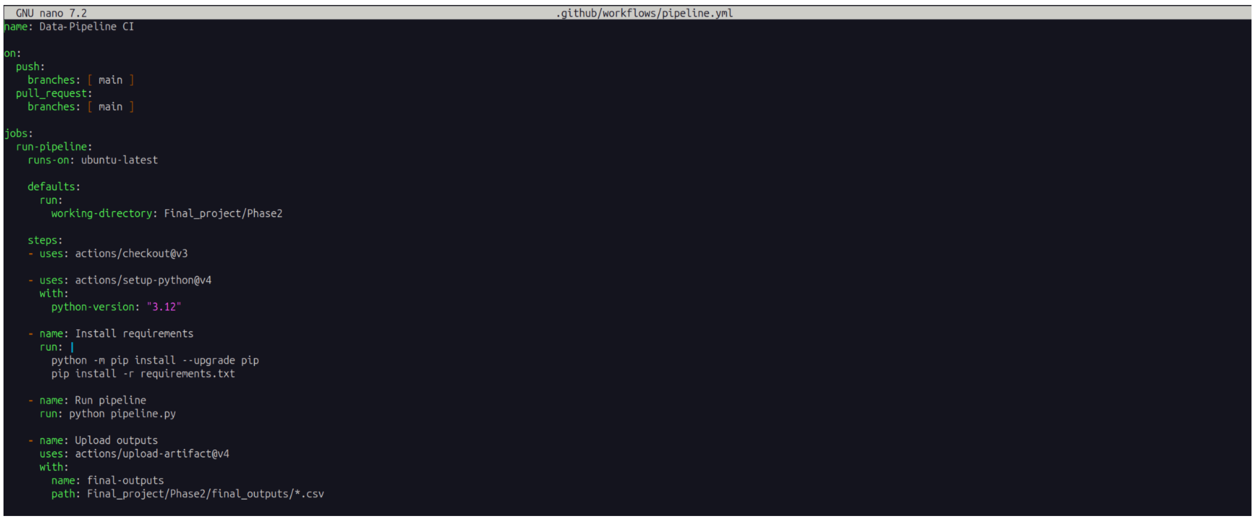
```
python pipeline.py
```

Output Files

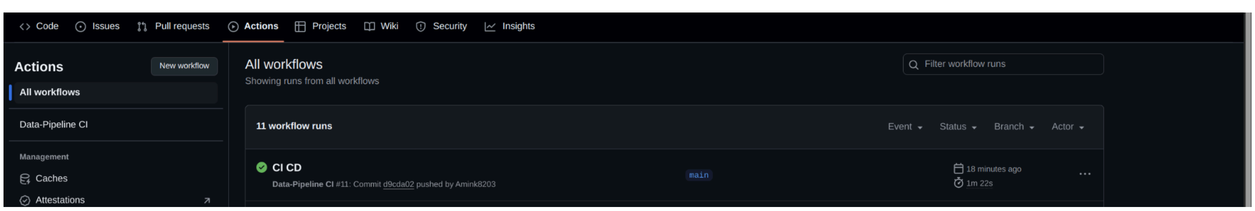
- `feature_eng.csv` : values haven't been normalized but new features are added
- `processed.csv` : Dataset with new columns and normalized values

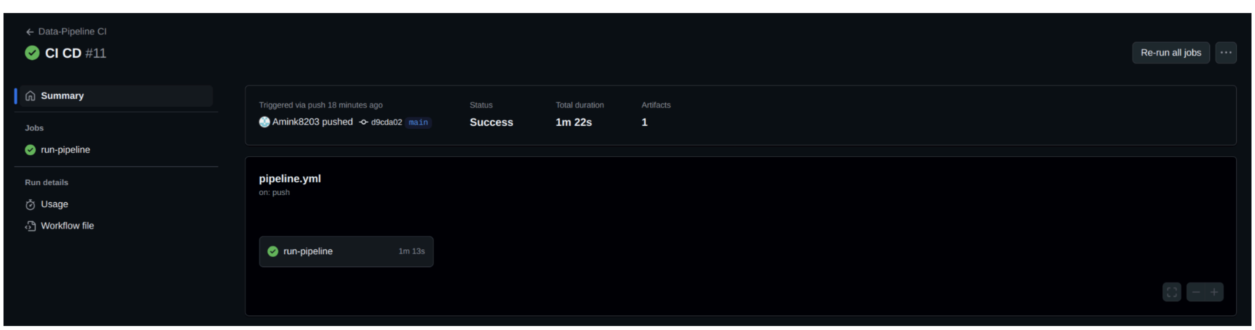
CI/CD Implementation

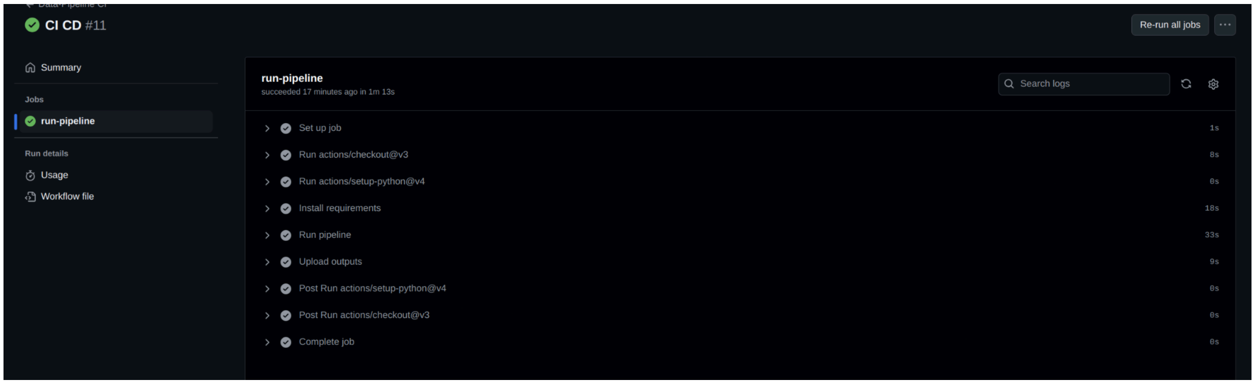
For this part, we must create CI/CD for automating the workflow of our project. The final csv outputs. Below is the pipeline.yml script:


```yaml
  GNU nano 7.2                         .github/workflows/pipeline.yml
name: Data-Pipeline CI

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  run-pipeline:
    runs-on: ubuntu-latest

    defaults:
      run:
        working-directory: Final_project/Phase2

    steps:
    - uses: actions/checkout@v3

    - uses: actions/setup-python@v4
      with:
        python-version: "3.12"

    - name: Install requirements
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt

    - name: Run pipeline
      run: python pipeline.py

    - name: Upload outputs
      uses: actions/upload-artifact@v4
      with:
        name: final-outputs
        path: Final_project/Phase2/final_outputs/*.csv
```

Also below is the images of the successful pipeline execution from GitHub Actions:







Bonus Part - MLOps

To make the entire pipeline portable and reproducible, I containerised it with Docker. I created a Dockerfile in the project root that starts from the lightweight python:3.12-slim base image, copies the whole repository into /app, installs the exact libraries listed in Final_project/Phase2/requirements.txt, and sets the default command to python pipeline.py inside the Phase2 folder. Building the image with:

```
docker build -t flight-pipeline:latest .
```

freezes both the code and its environment into a single artefact. Running the container via below command:

```
mkdir outputs
docker run --rm \
  -v $PWD/outputs:/app/Final_project/Phase2/final_outputs \
  flight-pipeline:latest
```

executes the full workflow and writes the two final CSV files `processed.csv` and `feature_eng.csv` into the `outputs/` directory. Because everything (Python, packages, SQLite database, scripts) is baked into the image, the pipeline now runs identically on any machine that has Docker installed. Below is the Dockerfile written for this part:

Also below is the image of successfully building the image and running the container and making final 2 csv output files and saving them into new “outputs” folder in repo root: