Data Science Project – Phase3

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:

```
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):

```
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_D
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

```
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

```
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:

```
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:

```
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:

```
# 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:

```
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:

```
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 the median of that column in the cells.

Model Development and Training

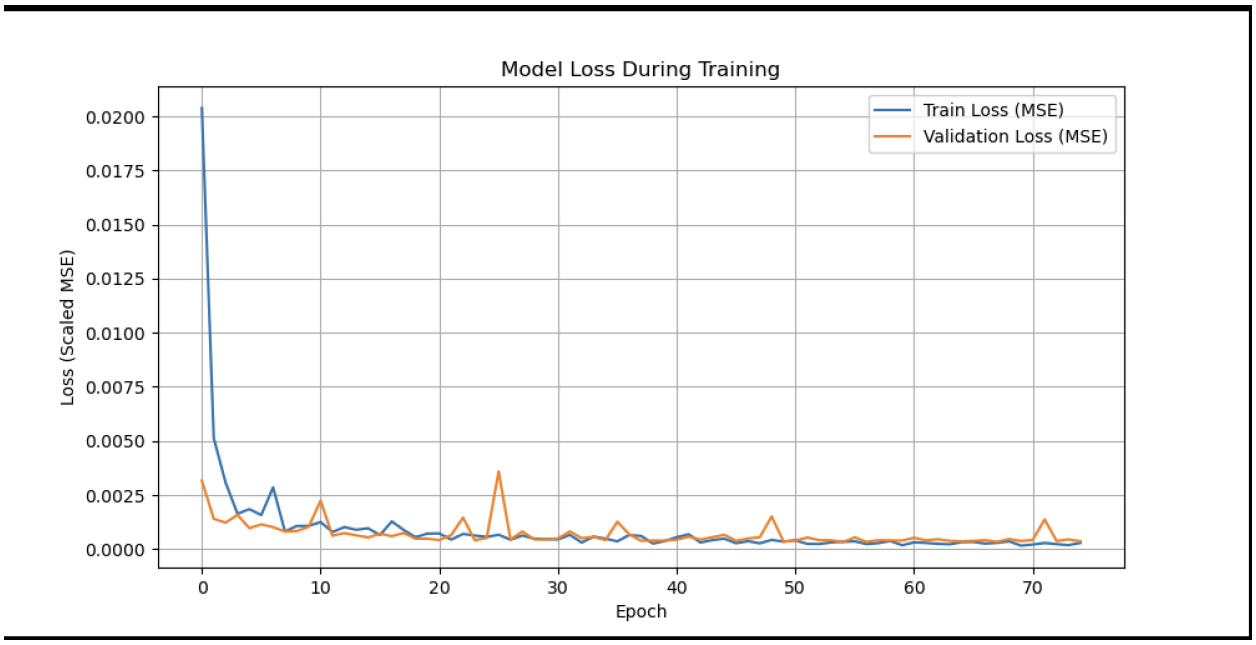
For this part, we have used a neural network for our task because our data is huge enough (500000) that a neural network would perform best in comparison to the other models for classification. Our target feature is the arrival delay of the flights to their destination. Below is the architecture of our model:

```
cat_inputs_train = {}
cat_inputs_valid = {}
cat_input_layers = []
cat_emb_layers = []
temp_full_cat_data_for_embedding_dim = X_processed_full.copy()

for col in cat_cols_model:
    temp_full_cat_data_for_embedding_dim[col] = temp_full_cat_data_for_embedding_dim[col].fillna(-1).astype("int32")
    max_val_for_col = temp_full_cat_data_for_embedding_dim[col].max()
    n_cat = max_val_for_col + 1
    embed_dim = min(50, (n_cat // 2) + 1)

    inp = layers.Input(shape=(1,), dtype="int32", name=f"{col}_inp")
    cat_input_layers.append(inp)
    emb = layers.Embedding(input_dim=n_cat, output_dim=embed_dim, name=f"{col}_emb")(inp)
    emb = layers.Flatten()(emb)
    cat_emb_layers.append(emb)
    cat_inputs_train[f"{col}_inp"] = X_train_proc[col].fillna(-1).astype("int32").values
    cat_inputs_valid[f"{col}_inp"] = X_valid_proc[col].fillna(-1).astype("int32").values

num_inp  = layers.Input(shape=(X_train_num_scaled.shape[1],), name="num_inp")
if cat_emb_layers:
    x = layers.concatenate([num_inp] + cat_emb_layers)
else:
    x = num_inp

x = layers.Dense(256, activation="relu")(x)
x = layers.Dense(128, activation="relu")(x)
x = layers.Dense(64, activation="relu")(x)
x = layers.Dense(32, activation="relu")(x)
out = layers.Dense(1, name="regression_output")(x)
model_inputs = [num_inp] + cat_input_layers if cat_input_layers else [num_inp]
model = models.Model(inputs=model_inputs, outputs=out)
model.compile(optimizer="adam", loss="mse", metrics=["mae"])
model.summary()
```

We have used embedding layers for categorical variables that are converted to numerical variables because if we want to use one hot encoding on them, it would be very inefficient (for example 4000 columns (features) for Tail_number which is not desirable). After that, we fitted our model on the train set. Also we have used early stopping to decrease overfitting.

```python
early_stop = callbacks.EarlyStopping(monitor='val_loss', patience=20, restore_best_weights=True)
train_model_inputs = {"num_inp": X_train_num_scaled, **cat_inputs_train}
valid_model_inputs = {"num_inp": X_valid_num_scaled, **cat_inputs_valid}

gpu_devices = tf.config.list_physical_devices('GPU')
device_to_use = '/GPU:0' if gpu_devices else '/CPU:0'
with tf.device(device_to_use):
    history = model.fit(
        x=train_model_inputs,
        y=y_train_scaled,
        validation_data=(valid_model_inputs, y_valid_scaled),
        batch_size=256,
        epochs=100,
        callbacks=[early_stop],
        verbose=1
    )
```

Next, we have drawn the plot of loss of the model:



For the next part, we will evaluate the model using different metrics:

```
----- Prediction Metrics on Saved Test Data (Original Scale) -----
MSE   : 0.396
RMSE  : 0.630
MAE   : 0.278
R²    : 0.9997
SMAPE : 14.05%
```

The Mean Squared Error (MSE) for our flight arrival delay predictions is 0.396 minutes². This metric represents the average squared difference between predicted and actual delays. While its units (minutes squared) make direct interpretation less intuitive, the

very low value signifies that the model generally makes predictions with small squared errors, heavily penalizing any large deviations.

The Mean Absolute Error (MAE) achieved by the model is 0.278 minutes. This metric signifies that, on average, our predictions deviate from the actual flight arrival times by only about 17 seconds. MAE provides a straightforward interpretation of the average error magnitude and is less influenced by outliers than RMSE. This very low MAE further underscores the model's high precision.

R^2 indicates the proportion of the variance in the dependent variable (actual flight arrival delays) that is predictable from the independent variables (the features your model uses). It essentially measures how well the model's predictions approximate the real data points. An R^2 of 1 indicates that the regression predictions perfectly fit the data. The R-squared (R^2) value for our model is an outstanding 0.9997. This indicates that 99.97% of the variance in flight arrival delays within our test dataset is accounted for by our model's predictions. Such a high R^2 value, very close to the maximum of 1, suggests an exceptionally strong goodness-of-fit and that the features selected and the model architecture are highly effective in explaining delay patterns. While such high values are excellent, they also prompt a review to ensure no data leakage or issues with test set independence.

SMAPE is a measure of prediction accuracy expressed as a percentage.

$$SMAPE = \frac{100\%}{n} \sum_{i=1}^{n} \frac{|PredictedDelay_i - ActualDelay_i|}{(|ActualDelay_i| + |PredictedDelay_i|)/2}$$

The Symmetric Mean Absolute Percentage Error (SMAPE) is 14.05%. This metric provides a relative measure of error, indicating that the average prediction error is approximately 14.05% of the average of the absolute actual and predicted delay values. While our absolute error metrics (MAE and RMSE) are exceptionally low (sub-minute), the SMAPE value suggests that when errors are viewed in percentage terms,

particularly for flights with shorter delays where small absolute errors can represent a larger percentage, there is this level of relative deviation. It would be beneficial to analyze the error distribution, especially for flights with low actual delay values, to gain further insight into this percentage error.

Pipeline

We have a run_pipeline.py file that with the argument “train”, it will read the data from the database, preprocess it and train the model and save X_test and y_test to the database and save the model too. Then with the “prediction” argument, it will read the test data from the dataset and predict it on the saved model and print the metrics. Now we will explain the other scripts.

create_database.py will create the database from our data and import it into the database. Aslo load_data.py will read from the database. feature_engineering.py will do the feature_engineering on the data. preprocess.py will do the preprocessing on the data. train_model.py will train the model on the preprocessed data and save the model for later prediction. Also it will save the image of the loss plot in the “Models” directory and also it will save the preprocessed X_test and y_test into the database for later predictions. At last, the predict_on_saved_test.py script will read the X_test and y_test from the database and predict on the saved model and will print the final metrics and also save the y_pred into the database. We can read the real y_test and y_pred from the dataset like below:

```python
import pandas as pd

with sqlite3.connect(DB_PATH) as conn:
    df_true_values = pd.read_sql_query("SELECT * FROM saved_y_test", conn)
    display(df_true_values.head(10))

    # q = """
    #     SELECT yt.actual_arrival_delay
    #     FROM saved_y_test yt
    # """
    # query = pd.read_sql_query(q, conn)
    # display(query)
```

✓ 0.2s Python

	# flight_id	# actual_arrival_delay
0	11	-30.0
1	27	-3.0
2	30	2.0
3	68	-4.0
4	80	9.0
5	89	-18.0
6	134	-6.0
7	169	27.0
8	194	11.0
9	213	3.0

10 rows x 2 cols 10 per page Page 1 of 1

```python
import pandas as pd

with sqlite3.connect(DB_PATH) as conn:
    df_predicts = pd.read_sql_query("SELECT * FROM pipeline_test_predictions", conn)
    display(df_predicts.head(10))

    # q = """
    #     SELECT yp.predicted_arrival_delay
    #     FROM pipeline_test_predictions yp
    # """
    # query = pd.read_sql_query(q, conn)
    # display(query)
```

✓ 0.0s Python

	# flight_id	# predicted_arrival_delay
0	11	-30.336095809936523
1	27	-3.2222816944122314
2	30	1.9420818090438843
3	68	-4.065832138061523
4	80	9.057666778564453
5	89	-18.058673858642578
6	134	-5.812082290649414
7	169	26.77565574645996
8	194	10.948182106018066
9	213	3.1814582347869873

10 rows x 2 cols 10 per page Page 1 of 1

MLFlow

1- Introduction and Objectives

To systematically manage the lifecycle of our flight delay prediction model, we incorporated MLflow, an open-source platform for MLOps. The primary objectives of this integration were to:

- Track and compare different experimental runs.
- Log and version control key parameters, performance metrics, and model artifacts.
- Enhance the reproducibility of our modeling process.
- Facilitate model versioning and management through the MLflow Model Registry.

This section details the setup, workflow, and outcomes of using MLflow in our project.

2- MLflow Implementation Strategy

1. Installation and Setup: MLflow was installed into our Python environment via pip. All experiments were configured to log data to a local `mlruns` directory, which serves as the backend store for the MLflow Tracking Server accessed via the `mlflow ui` command.

2. Experiment and Run Management: A dedicated MLflow experiment, named "Flight_Delay_Prediction" (or as configured by the user), was established using `mlflow.set_experiment()`. Each execution of our main pipeline script (`run_pipeline.py`), whether for training or prediction, was initiated as a distinct MLflow run using the `with mlflow.start_run() as run:` context manager. This ensured that all associated metadata, parameters, metrics, and artifacts were logged to

the correct run within the designated experiment. Unique run names incorporating timestamps were used for better identification.

3. Comprehensive Logging: Our strategy involved logging various aspects of the pipeline to ensure a detailed record of each run:

- **Parameters (`mlflow.log_param()`):**
  - Pipeline operation mode (e.g., `pipeline_action: "train"` or `"prediction"`).
  - Key data characteristics (e.g., `full_dataset_rows`, `train_set_size`, `validation_set_size`, `db_test_set_size`).
  - Critical model training hyperparameters (e.g., `batch_size`, `epochs`, `optimizer`, `early_stopping_patience`).
  - Details of the model architecture (e.g., `dense_layers` configuration, `embedding_details` serialized as JSON).
  - Status indicators for various pipeline stages (e.g., `training_status`, `db_test_set_save_status`).
- **Metrics (`mlflow.log_metrics()`):**
  - **Validation Metrics:** During model training in `train_model.py`, the `mlflow.keras.MLflowCallback` was employed to automatically log training and validation metrics (loss, MAE, RMSE) at the end of each epoch. Additionally, the best validation scores achieved (based on early stopping) were explicitly logged (e.g., `best_val_loss_mse`, `best_val_mae`, `best_val_rmse`).
  - **Test Metrics:** During the prediction workflow in `predict_on_saved_test.py`, evaluation metrics (MSE, RMSE, MAE, R^2, SMAPE) calculated on the dedicated, database-stored test set were logged.
- **Artifacts (`mlflow.log_artifact()` and `mlflow.keras.log_model()`):**

    - **Preprocessing Objects:** The fitted scikit-learn `ColumnTransformer` (`preprocessor/preprocessor.pkl`) and `StandardScaler` instances (`scalers/scaler_x.pkl`, `scalers/scaler_y.pkl`) were saved as artifacts.
    - **Trained Model:** The Keras model was logged using `mlflow.keras.log_model()`. This process stores the model in MLflow's format (which includes the Keras `model.h5` or `saved_model` directory, `conda.yaml`, `python_env.yaml`, and `MLmodel` files) under the `keras_model` artifact path. An `input_example` derived from the validation data was provided to `mlflow.keras.log_model()` to facilitate automatic model signature inference.
    - **Visualizations:** The plot of training and validation loss over epochs (`plots/training_loss_plot.png`) was logged.
    - **Data Snapshots:**
        - The entire SQLite database (`flight_data.db`) was logged as an artifact (`database_snapshot/flight_data.db`) at the end of a successful training run. This snapshot includes the raw data, engineered features, and the specific test set (`saved_processed_X_test`, `saved_y_test`) created during that run, ensuring data reproducibility.
        - The predictions made on the test set during the prediction workflow were logged as a CSV artifact (`predictions_output/test_set_predictions.csv`).
- **Tags (`mlflow.set_tag()`):**
    - Descriptive tags were added to runs to provide additional context, such as `model_source` (pointing to the local path of a saved model copy) or `pipeline_action`.

4. Model Registry and Versioning: When logging the Keras model, the `registered_model_name="FlightDelayKerasModel"` argument was used. This automatically:

- Creates the "FlightDelayKerasModel" in the MLflow Model Registry if it doesn't exist.
- Registers each successfully logged model as a new, sequential version under this registered model name (e.g., Version 1, Version 2, etc.).

This allows for a clear lineage of model versions, each linked back to the specific run that produced it, along with all its parameters, metrics, and artifacts.

3- Navigating and Utilizing MLflow UI

The MLflow User Interface, accessed via `mlflow ui`, provided a web-based dashboard to visualize and manage our experiments. Key functionalities utilized include:

- **Experiment View:** Listing all runs under the "Flight_Delay_Prediction" experiment.
- **Run Detail View:** For each selected run, the UI displays:
    - **Parameters:** All logged parameters.
    - **Metrics:** Logged metrics, including interactive plots for metrics logged over time (like epoch-wise training/validation loss).
    - **Artifacts:** A file browser to access and download all logged artifacts, including the Keras model, scalers, plots, and the database snapshot.
    - **Tags:** All set tags.
- **Model Registry View:** Listing registered models (e.g., "FlightDelayKerasModel") and their versions. Selecting a specific version shows its details, source run, and (if successfully inferred) its input/output schema.

4- Model Signature and Schema

A crucial aspect of logging models with MLflow is the **model signature**, which defines the expected schema (data types and shapes) for the model's inputs and outputs. This is typically inferred by MLflow when an `input_example` is provided during `mlflow.keras.log_model()`.

As observed in the MLflow UI for our registered "FlightDelayKerasModel" (e.g., Version 3 in the provided screenshot), the "Schema" tab currently displays "Inputs (0)" and "Outputs (0)". This indicates that the `input_example` provided during the model logging process was not successfully processed by MLflow to infer and save the signature. This was traced back to internal `FileNotFoundError` exceptions when MLflow attempted to write the `input_example.json` to a temporary directory.

While fallback mechanisms were implemented to ensure the model artifact itself was still logged to MLflow and registered, the absence of an auto-inferred schema is a limitation for the current versions. Successfully resolving the underlying issue preventing `input_example` processing (which could be related to data type serialization within the example, temporary file system interactions, or specific MLflow/dependency version behaviors) would enable the schema to be populated. A populated schema would list expected inputs like `num_inp`, `cat_DAY_OF_WEEK_inp`, etc., with their tensor specifications, and similarly for the model's outputs, greatly enhancing model understanding, validation, and deployment readiness.

5- Conclusion and Benefits

Integrating MLflow into our flight delay prediction pipeline has significantly improved our MLOps practices. The key benefits include:

- **Structured Experiment Tracking:** All relevant information from each run is systematically captured and organized.

- **Enhanced Reproducibility:** The ability to retrieve exact parameters, code context (via source run), model versions, and even data snapshots (the logged database) is invaluable for reproducing results.
- **Systematic Model Versioning:** The Model Registry provides a clear audit trail of model development and allows for easy access to any specific version.
- **Centralized Artifact Management:** Models, preprocessing objects, plots, and data files are co-located with their respective run metadata.
- **Improved Analysis and Comparison:** The MLflow UI facilitates the comparison of different runs and model versions based on their logged metrics and parameters.

Despite the challenge with automatic schema inference via `input_example` in the current setup, the core benefits of experiment tracking, artifact logging, and model versioning have been successfully achieved, laying a strong foundation for further model development and potential deployment. Future work will focus on resolving the schema inference issue to further enrich the metadata stored for each model version.