Module Code: CSMBD

Assignment report Title: Big Data & Cloud Computing Coursework Report

Student: 32824514

Task A: https://csgitlab.reading.ac.uk/fk824514/lightmapreduce
Task B: https://csgitlab.reading.ac.uk/fk824514/bigdatacoursework

Date (when the work is completed): 19.05.2025

Actual hours spent on the assignment: 30 hours

Task A: Cloud Computing

1 Abstract

MapReduce is an excellent distributed computing framework that combines many cheap and low-power servers to work together. Through the highly customizable capabilities provided by Mapper and Reducer, it can handle most big data mining tasks. This project aims to mimic the MapReduce framework to build a lightweight parallel computing framework and complete the task of identifying passengers with the highest number of flights.

2 Introduction

As various industries have access to data services, a huge amount of data is generated and collected every day. As the amount of data sours, distributed computing architecture emerges.

MapReduce is an excellent distributed computing framework that makes it possible for many cheap and low-power servers to work together. This job introduces a lightweight parallel computing framework Light MapReduce. Compared with MapReduce, it only focuses on data concurrency and integration in one server, which makes it have the same high-speed computing power and friendly interface as Hadoop, but with a more compact and lightweight architecture. It encapsulates the logic of parallel computing and only requires a customized Mapper and Reducer to complete a job.

At the end of this job, Light MapReduce will be validated on a specific task to identify passengers with the highest number of flights.

3 Development Overview

Light MapReduce is a parallel computing framework that runs on demand. Each job needs to be started once as Hadoop. It was written by Java and only supports Mappers and Reducers written in Java. In contrast, MapReduce supports Python and Java.

3.1 Software Architecture

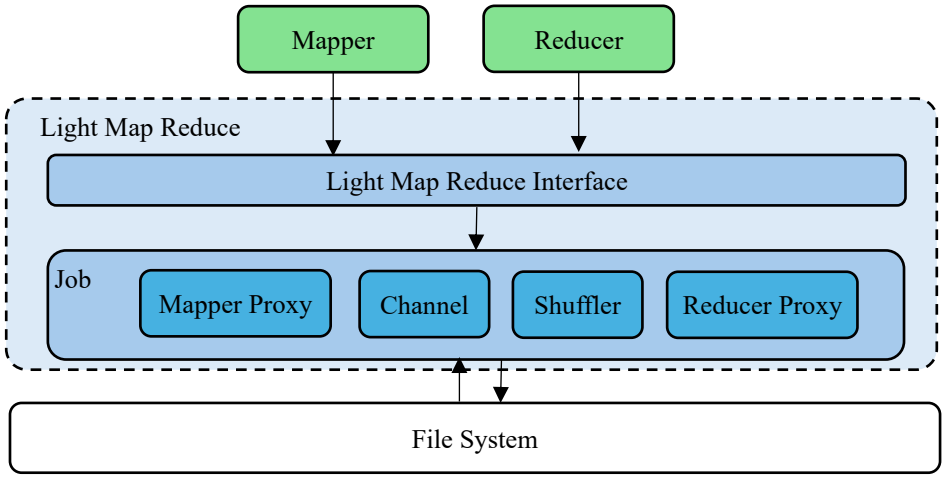


Image 1: Solution Architecture of Light MapReduce

Light MapReduce contains an interface module and a job module. The interface module is the user-facing entrance. It receives and validates parameters such as the names of a Mapper and a Reducer, input document paths and a path for output file. The parameters then be passed to the Job module to execute the job.

The job module consists of 4 components: Mapper Proxy, Channel, Shuffler and Reducer Proxy. Mapper Proxy is responsible for reading the list of file names and calling the provided Mapper to run tasks parallelly. The results of Mapper Proxy are storied on Channel. Shuffler then shuffles the results to a unified HashMap. Next, the Reducer Proxy calls the reducers to process the data concurrently. Finally, the results are collected and written on an output file.

3.2 Technology Architecture

Developed by Java, Light MapReduce is built using only Java built-in components, which minimizes dependencies. The components include New IO (NIO) [1] for file reading and writing, Thread Pool [2] for parallel computing, Message Queue (MQ) for sharing data, Collections [3] for processing of structured data, and Generics [4] and Template for reducing dependencies.

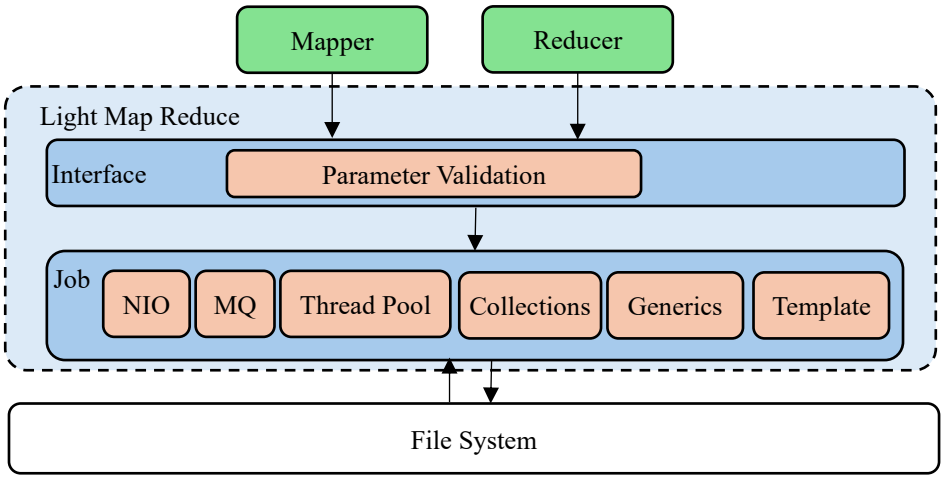


Image 2: Technology Architecture of Light MapReduce

4 Version Control Process

This project is managed by Gitlab, provided by the school of Computer Science, University of Reading: https://csgitlab.reading.ac.uk/fk824514/lightmapreduce. Considering that the project is relatively simple, all codes and sample data are currently maintained on the main branch. To try the code, please follow the steps below:

```
git clone https://csgitlab.reading.ac.uk/fk824514/lightmapreduce -b main
cd lightmapreduce
```

Code 1: Steps to Try the Code

To learn more about the code, please check out the README.md file. In addition, key codes and public interfaces and methods are documented. Fell relax to explore the code.

5 Flowchart

The following image shows the main workflow for a job running in Light MapReduce. After submitting a job, a configuration is wrapped and used to create a job pipeline. Several Mapper threads are created and run a mapper for each to read and process data. The results of a mapper are stored within the mapper at first, and then the result package is passed to a channel. A shuffler gets the data from the channel and aggregates to a map architecture. Then the key-value pairs are sent to multiple thread reducers for further processing. Finally, the results are collected, sorted and written on an output file.

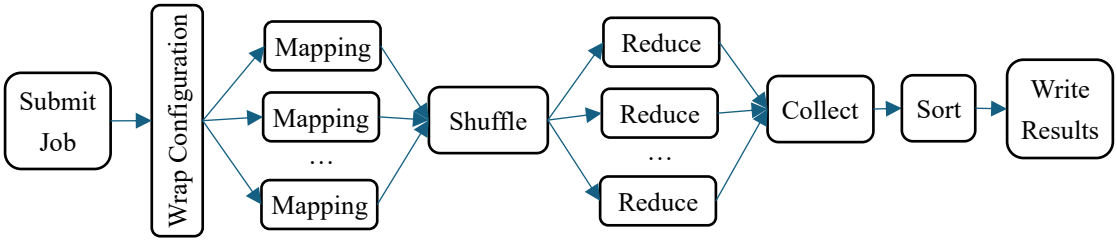


Image 3: Main Flow of a Job in Light MapReduce

6 Implementation of Light MapReduce

Image 4 shows the core of Light MapReduce (See Appendix A for further details). Light MapReduce is the user interface, accepting an input folder, output file, mapper name and output name as parameters. Concurrent Job contains the main process of a job and is called by the run function. Mapper is the super class of customized mappers, managing the operations of multi-threaded mappers. Subclass mappers are expected to implement a map method and to send the results by calling the emit intermediate function. Channel directly stores the results of mappers in a concurrent queue. Once the mappers are done, the results are collected by Shuffler and aggregated to a key-value pair map by calling a run function. The logic of Reducer is like that of Mapper. Please check the code for further details.

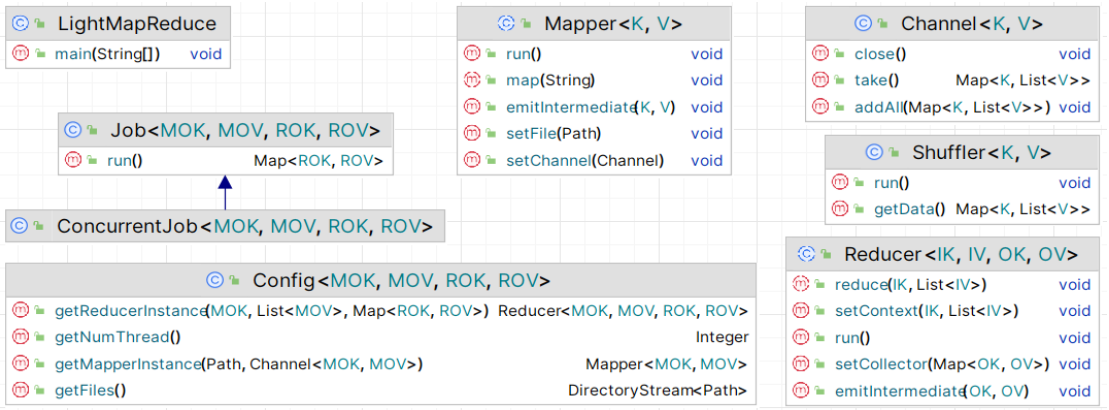


Image 4: The core diagrams of Light MapReduce

7 Implementation of the Task

The goal of the task is determining the passengers having had the highest number of flights. The input data is a CSV file, where each row of data represents a trip with a

passenger id. As the process shown below, the raw data (IDs) is collected by Mapper A and then passed to a Shuffler. Reducer A aggregates the data and stores it as a TXT file. Next, Mapper B reads the file and exchanges the Key-Value pairs to Num-IDs. Then they are shuffled to a unified map. Finally, the map data is distributed to Reducer B and converted into a string for output. By adjusting the threshold in Reducer B, the passengers having had the highest number of trips can be highlighted.

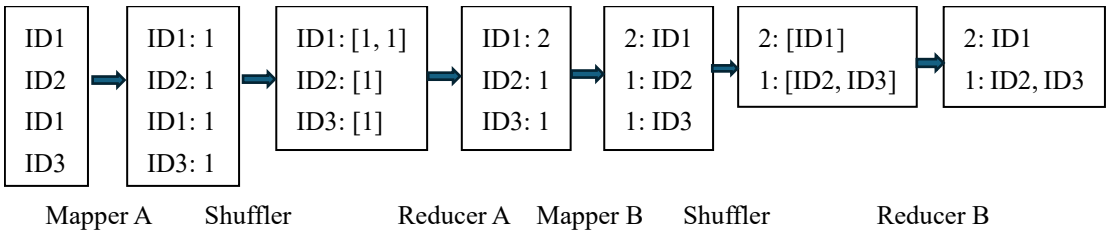


Image 5: A sample of the process of Job A and B

Mappers and reducers are constructed according to the requirements of the super classes and implement the corresponding methods to meet the above business.

8 Conclusion

This work builds a lightweight parallel computing framework Light MapReduce. It has minimal dependencies and is decoupled from the business. By customizing mappers and reducers, it can meet highly customized business scenarios. We found that multiple jobs were needed to achieve the goal of complex businesses. Potential optimizations could be considered supporting the submission of multiple jobs at the same time.

Task B: Big Data

1 Introduction

1.1 Background

Margie's Travel (MT) aims to modernize its customer service by developing a web app that predicts flight delays ≥15 minutes, primarily due to weather. The goal is to empower agents with real-time insights to improve traveler experience.

The solution requires processing large-scale historical flight and weather data to build a predictive model, ensuring scalability, accuracy, and integration with modern cloud tools. Ideally, this would be a prediction that could be updated instantly based on real-time conditions.

1.2 Solution Architecture

Handing large-scale datasets requires distributed processing methodologies. There are numerous cloud computing platforms globally, including Microsoft Azure, Amazon Web Services (AWS), Google Cloud Platform (GCP), and so on. Among these platforms, Azure excels in Artificial Intelligence Advanced Analytics capabilities and Enterprise-Grade Security and Compliance and was used in this project.

The main modules used in this project were:

Azure Data Factory: Integrate multiple data sources, load raw flight and weather data,

save intermediate data and build a pipeline to update data regularly and call the model to update the prediction results.

Azure Runtime Server and External Server: The factory can flexibly integrate internal and external services to perform computing, which improves the efficiency of collaboration and makes full use of resources.

Azure Databricks: Use notebook scripts to preprocess and aggregate weather and flight delay data, and use mlflow, PySpark and machine learning (ML) algorithms to train models and deploy services for prediction.

Azure SQL Database: As an intermediate medium, the database can store raw data, intermediate data and scored airport data for reading and display in Power BI. As a serverless application, database storage and scaling issues are optimized. Thanks to Spark's cloud nativeness, SQL can still be executed very efficiently on large amounts of data.

Power BI: Dashboard visualized delay probabilities, weather correlations, and ranked flight options for agents.

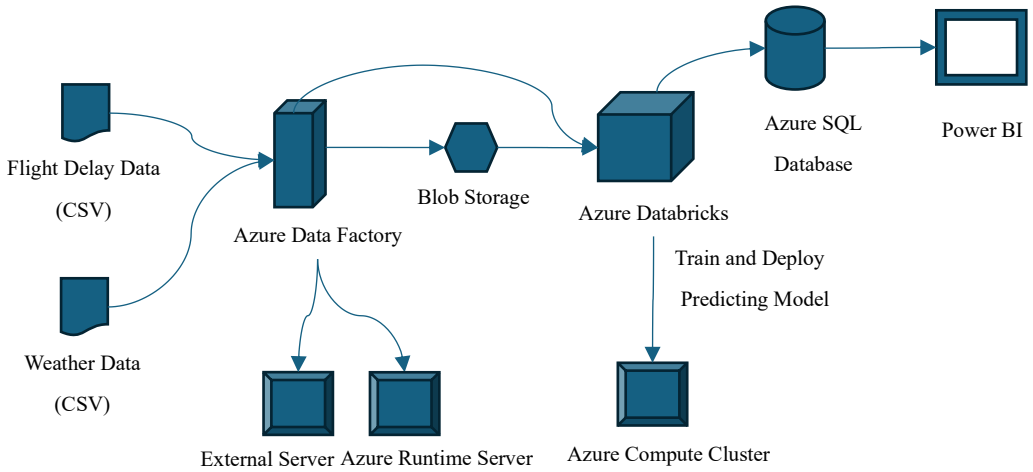


Image 6: Solution Architecture

2 Solution Implementation

2.1 Databricks Cluster

A computer cluster was created here to process data, train models, and deploy applications in Databricks. Its runtime version was 11.3 LTS ML, including Apache Spark 3.3.0, Scala 2.12 (See Appendix B for more details). Additionally, Mlflow and azureml-sdk[databricks] libraries were installed to support ML models.

2.2 Loading Sample Data

Sample data in three CSV files were uploaded to Azure platform through the catalog menu option (See Appendix C for more details).

2.3 Setup the Data Factory

In this project, Data Factory (See Appendix D for more details) is a data distribution center. It was used to integrate multiple data sources, load raw flight and weather data, save intermediate data and build a pipeline.

2.4 Data Factory Pipeline

In Data Factory, a pipeline was created to update data regularly and deploy an ML model to update the prediction results for visualization (See Appendix E for more details). It was scheduled to run once a month by a trigger.

2.5 Operation of ML

In this project, the problem can be defined as a binary classification problem, that is, whether a flight has a delay of more than 15 minutes. To solve this problem, two models, decision tree and random forest, are used for prediction. The decision tree model is used as the baseline model, and the random forest is used as the optimization model (See Appendix E for more details). The models were tuned by a Param Grid Builder and a 3-fold Cross Validator in a package of pyspark.ml.tuning. Through optimization, the decision tree and random tree finally achieved accuracies of 0.62 and 0.71 respectively.

2.6 Summarizing Data

Generated by the Azure Data Factory pipeline, the scored flight data was summarized and stored in a new table, named flight_delays_summary (See Appendix F for more details). The image below shows a sample of the summarizing table.

	OriginAirportCode	Month	DayofMonth	CRSDepHour	NumDelays	OriginLatLong
1	ATL	4	28	16	120	33.63666667,-84.427777...
2	MSP	4	12	19	88	44.88194444,-93.221666...
3	ATL	4	14	16	72	33.63666667,-84.427777...
4	MSP	4	11	19	72	44.88194444,-93.221666...
5	MSP	4	21	19	72	44.88194444,-93.221666...

Image 7: Summary of Flight Delay

2.7 Visualization of Data

Flight delay data was visualized in Power BI (See Appendix G for more details).
There was a Map, a Stacked Column Chart and a Tree Map to show delays at each airport.

3 Evaluation

3.1 Software Solution

This project tried two methods provided by PySpark: a decision tree and a random forest. Both algorithms are divide-and-conquer algorithms. They are insensitive to data types and do not require data type unification and normalization. When predicting models, the amount of calculation is also very small and fast. They are very suitable for model prediction of simple data. In this project, the decision tree model and the random forest model achieved 0.62 and 0.71 accuracy respectively.

The models were tuned by a Param Grid Builder and a 3-fold Cross Validator for better hyperparameters, which effectively improves the generalization ability and robustness

of the model. For further optimization, it is considering collecting more data and increasing the number of training times.

3.2 Cloud Computing Solution

Azure has a very complete tool chain, from storage resources, computing resources to result display. It is specifically reflected in the following aspects:

Scalability: Spark pools in Synapse/Databricks handled 10+ TB datasets efficiently. With the rapid growth of data, scalability is critical. Azure is fully capable of this.

Integration: Seamless toolchain (Data Factory → Databricks → ML → Power BI) minimized latency. Data can flow freely between various tools. If an automated pipeline is configured in the Data Factory, model training, deployment and prediction can be triggered regularly, ultimately achieving real-time or near real-time results.

Cost-Efficiency: The $100 credit provided to students is fully capable of completing this task, and the subsequent maintenance costs are also within an acceptable range.

3.3 Data Privacy and Security Risks

Data security is crucial in digital services, especially when data is uploaded and stored on public cloud services. During use, the data is also stored on a storage server in the UK. In terms of access rights, access is currently limited to the individual. Additionally, the data used in this case has been desensitized and does not involve sensitive data. The weather and flight information involved is public information on the Internet and has little impact.

4 Conclusion

This project proposes an executable, scalable and easy-to-maintain big data computing solution. Visualization services provide users with simple and clear information. When using the Azure platform, the technical chain is relatively long, and it is necessary to understand more components and sort out the relationship between them. However, in the long run, its maintenance is relatively simple. Future enhancements should focus on privacy-preserving ML and real-time data integration.

References

[1] Java NIO. (2022). Oracle Help Center; Core Libraries.
https://docs.oracle.com/en/java/javase/18/core/java-nio.html.

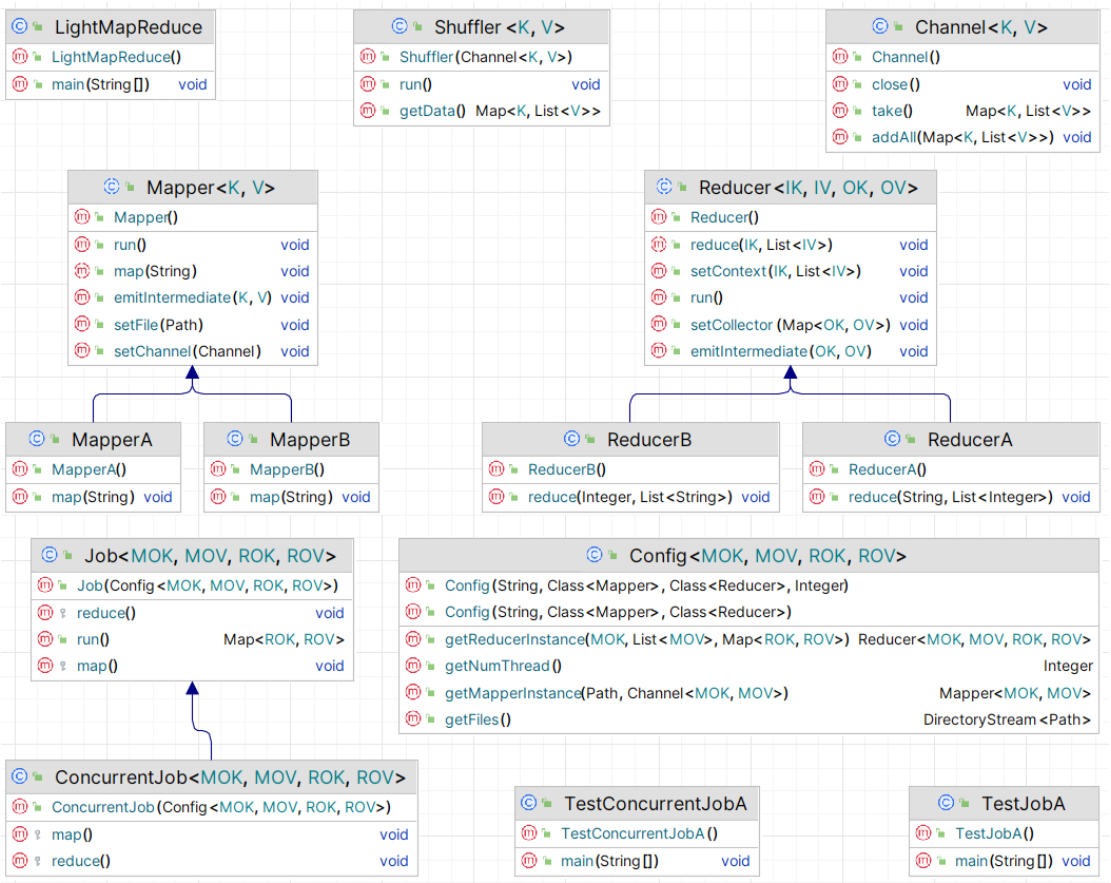
[2] Thread Pools (The JavaTM Tutorials > Essential Java Classes > Concurrency).
(n.d.). Docs.oracle.com.
https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html.

[3] Collections (Java Platform SE 8). (2025, May 17). Docs.oracle.com.
https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html.

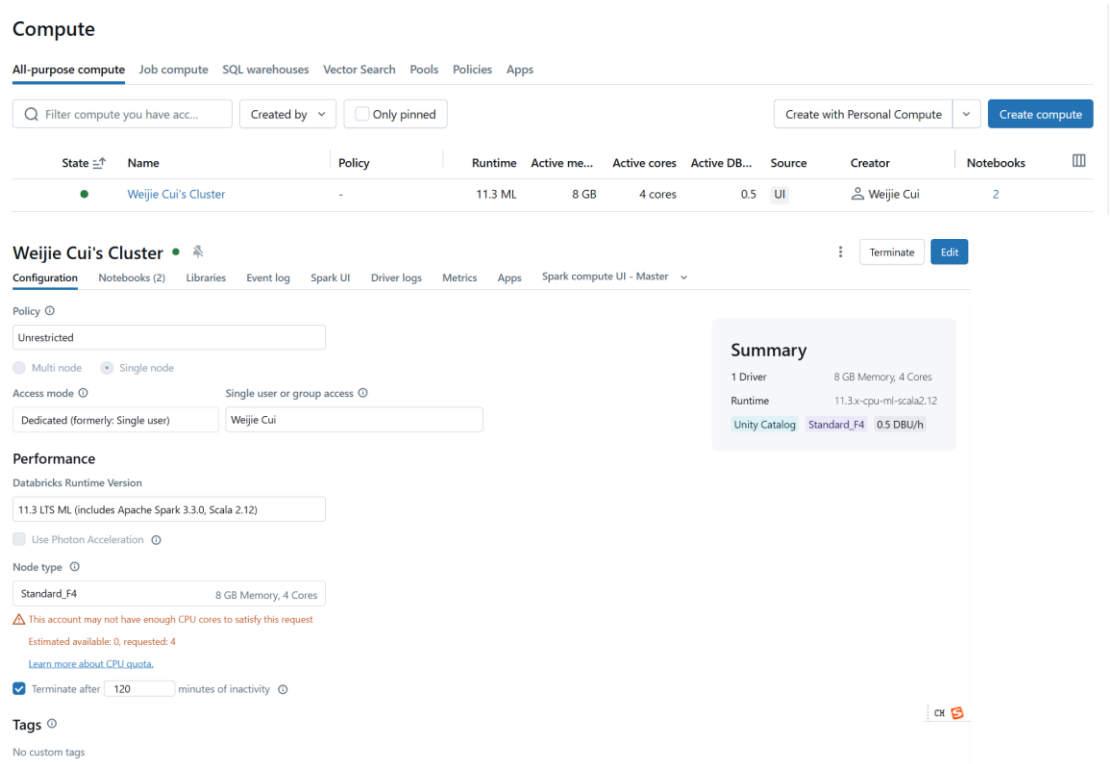
[4] Generic Types (The JavaTM Tutorials > Learning the Java Language > Generics
(Updated)). (2024). Oracle.com.
http://docs.oracle.com/javase/tutorial/java/generics/types.html

Appendices

A Diagrams of Light MapReduce



B Configuration of Databricks Compute Cluster



▼ Advanced options

Azure Data Lake Storage credential passthrough ⓘ

☐ Enable credential passthrough for user-level data access

Spark | Logging | Init Scripts | JDBC/ODBC

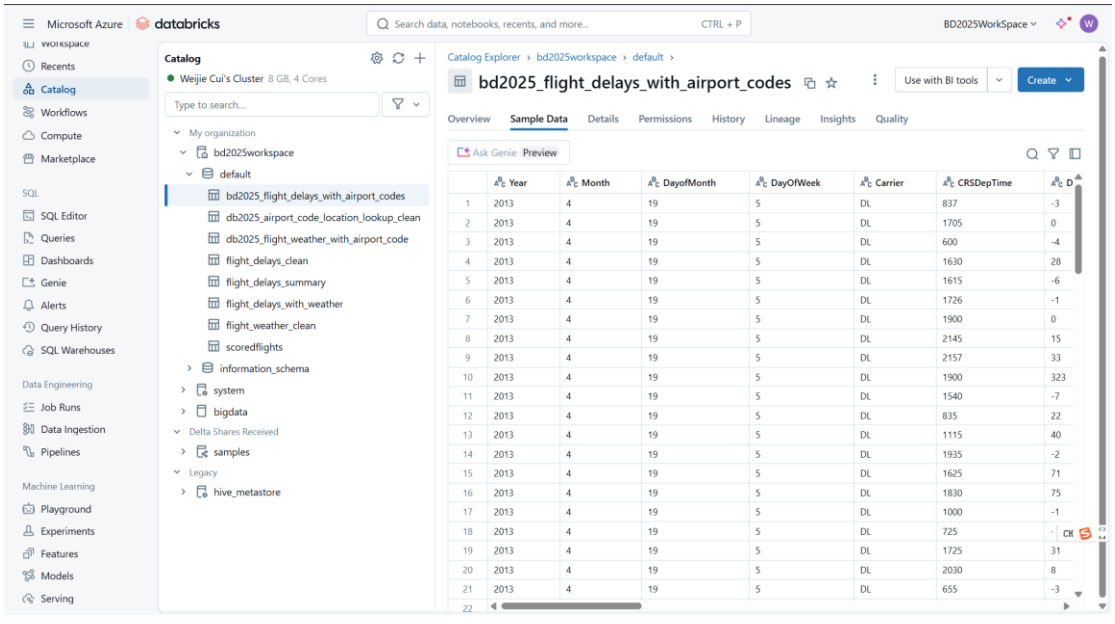
Spark config ⓘ

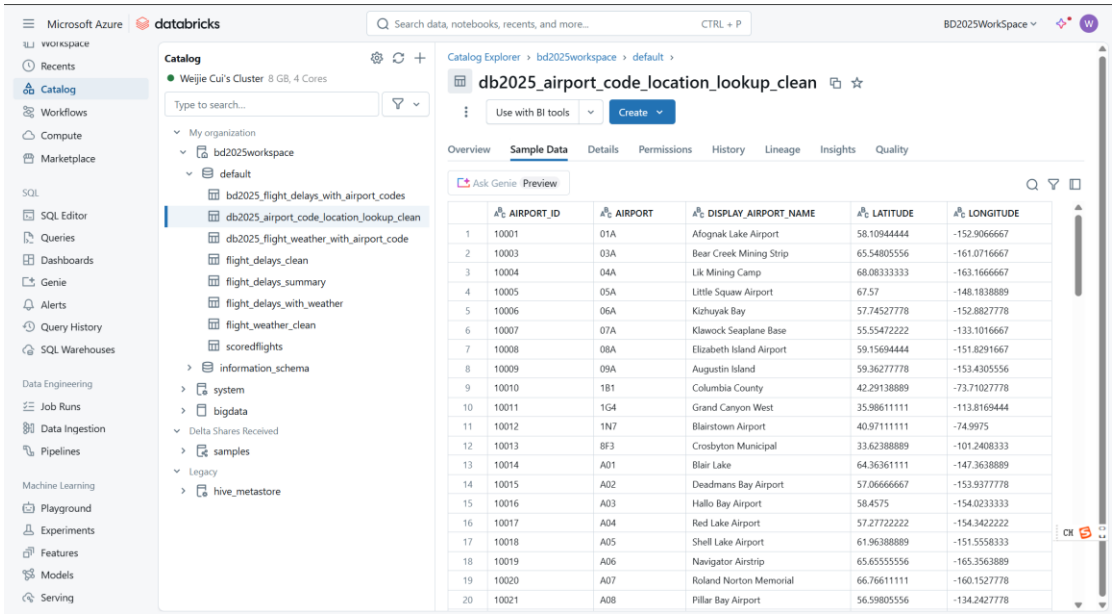
```
spark.master local[*, 4]
spark.databricks.cluster.profile singleNode
spark.hadoop.fs.azure.account.key.bd2025store.blob.core.windows
.net
hwy82mpx5fj3A7dTuTwm5odLTmP8I/UrEx5r/vEr73lgdHGT0/OD6C
IjvC2A1PlvGmFrfa+rZvPc+AStlTG4ow==
```

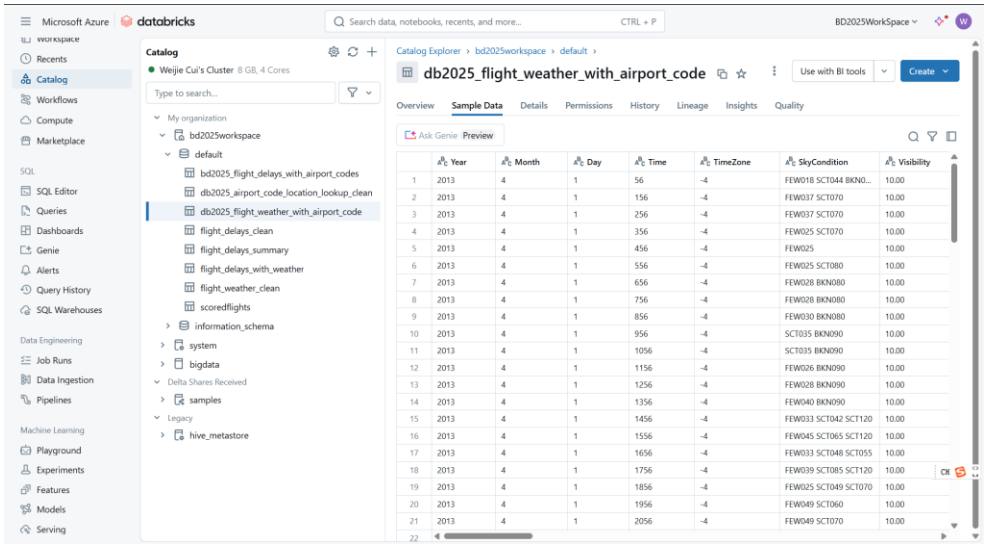
Environment variables ⓘ

No environment variables

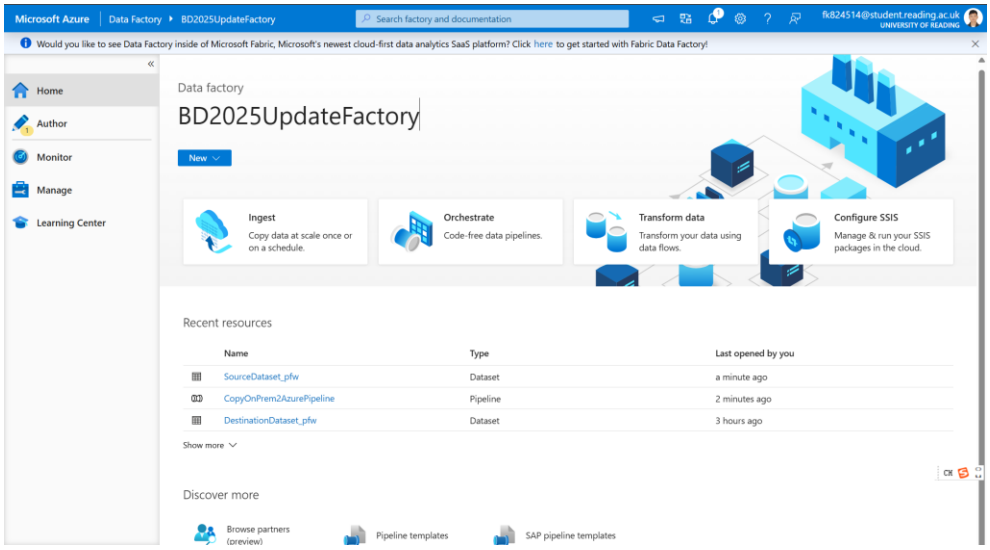
C Overview of Sample Data

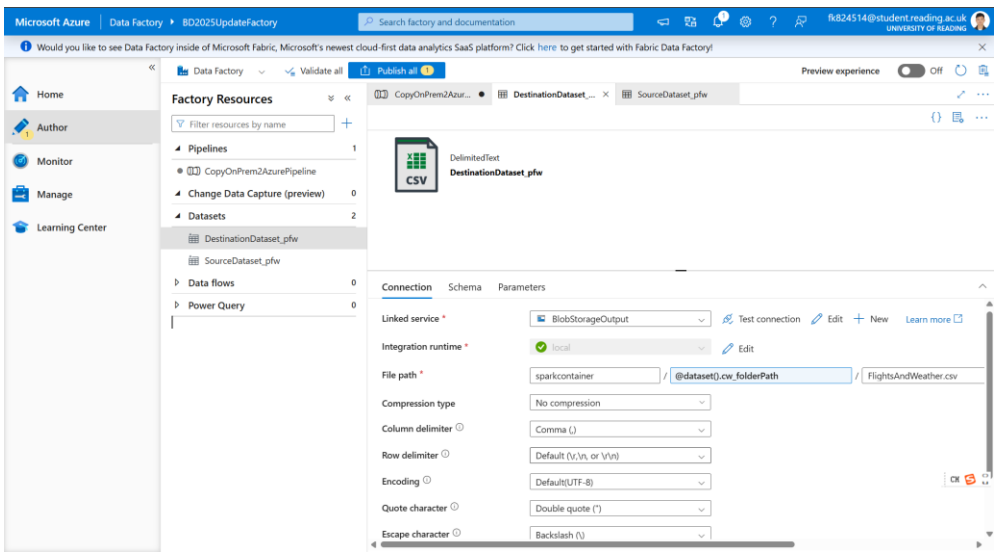


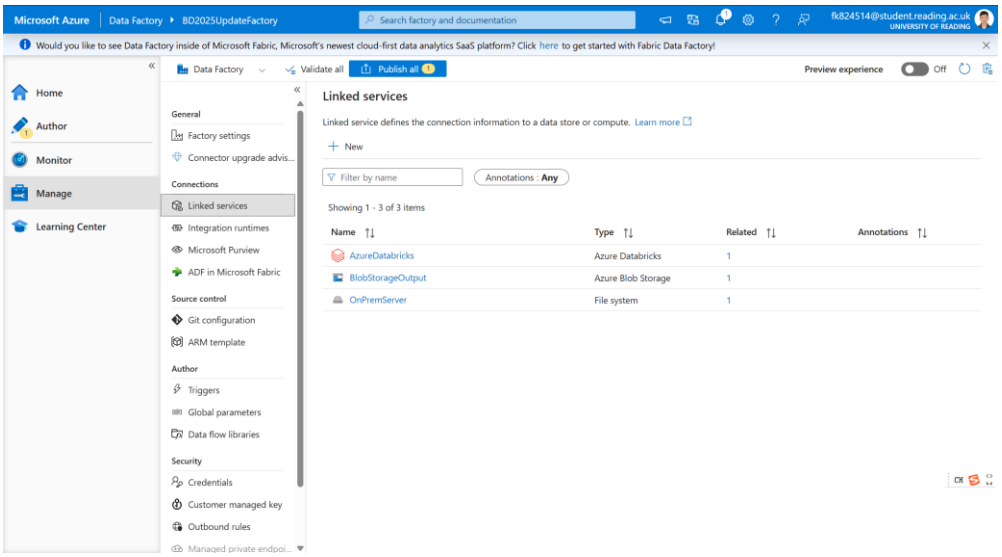


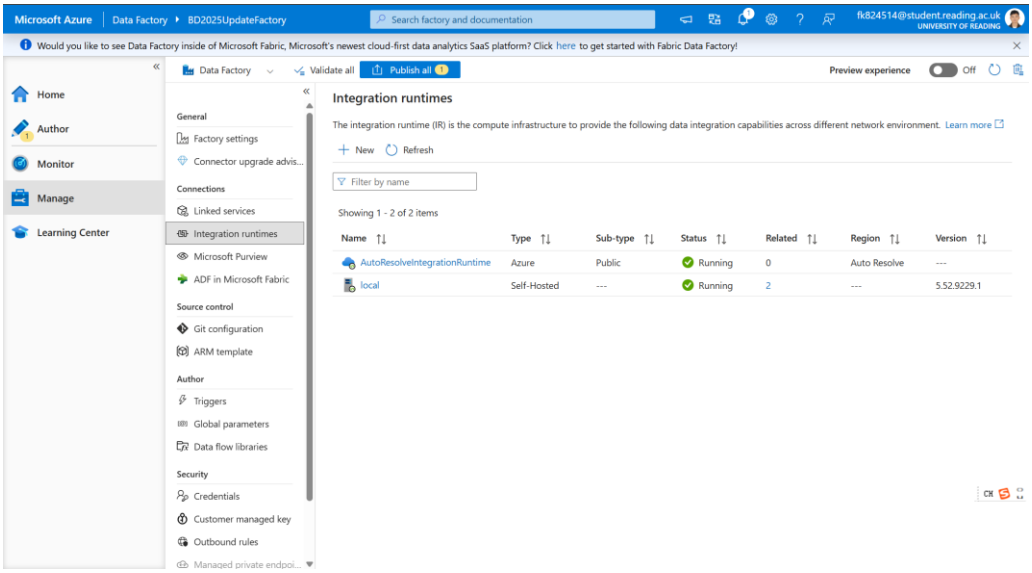


D Overview of Data Factory

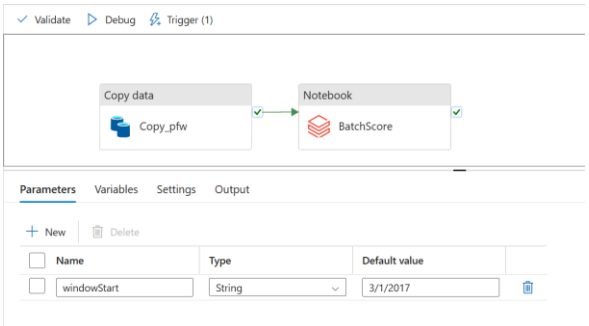


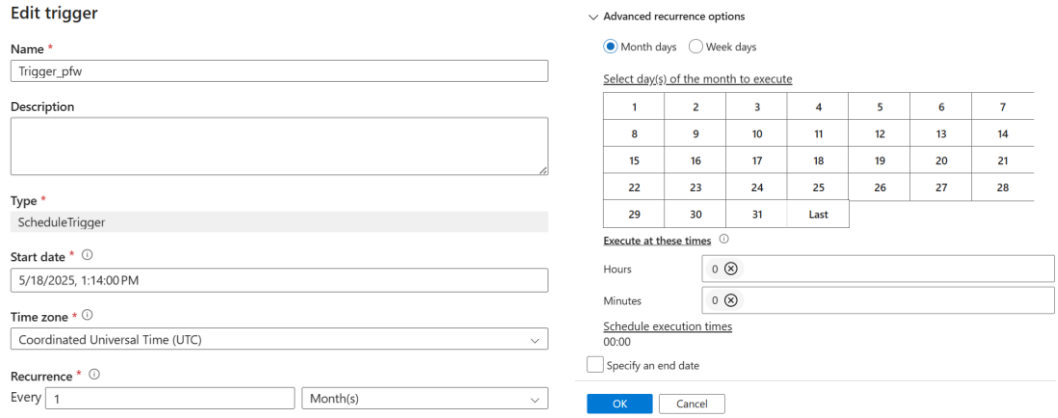






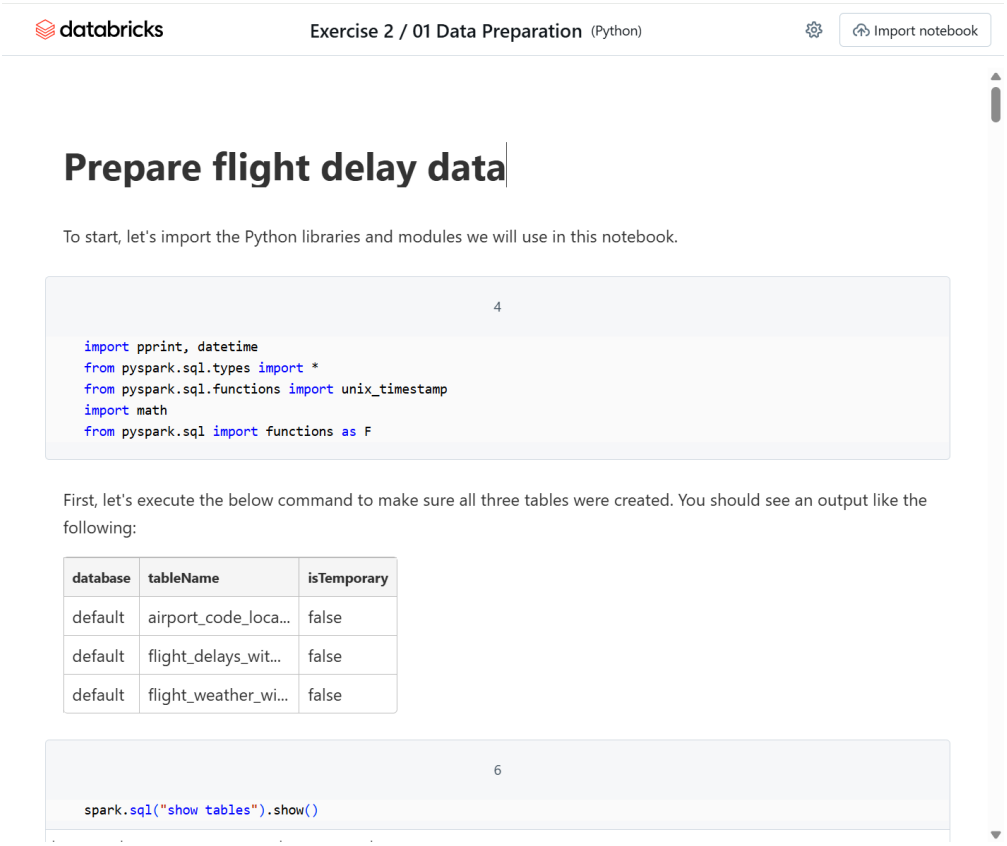
D Overview of Data Factory Pipeline and Trigger





E Files of Operating Machine Learning Models

E.1 Exercise 2 / 01 Data Preparation



```
+--------+--------------------+-----------+
|database|           tableName|isTemporary|
+--------+--------------------+-----------+
| default|bd2025_flight_del...|      false|
| default|db2025_airport_co...|      false|
| default|db2025_flight_wea...|      false|
+--------+--------------------+-----------+
```

Now execute a SQL query using the `%sql` magic to select all columns from flight_delays_with_airport_codes. By default, only the first 1,000 rows will be returned.

8

```
%sql
select * from bd2025_flight_delays_with_airport_codes
```

_sqldf: pyspark.sql.dataframe.DataFrame = [Year: string, Month: string ... 18 more fields]

Table

	Year	Month	DayofMonth	DayOfWeek	Carrier	CRSDepTime	DepD
1	2013	4	19	5	DL	837	-3
2	2013	4	19	5	DL	1705	0
3	2013	4	19	5	DL	600	-4
4	2013	4	19	5	DL	1630	28
5	2013	4	19	5	DL	1615	-6
6	2013	4	19	5	DL	1726	-1
7	2013	4	19	5	DL	1900	0
8	2013	4	19	5	DL	2145	15
9	2013	4	19	5	DL	2157	33
10	2013	4	19	5	DL	1900	323

Now let's see how many rows there are in the dataset.

10

```
%sql
select count(*) from bd2025_flight_delays_with_airport_codes
```

_sqldf: pyspark.sql.dataframe.DataFrame = [count(1): long]

Table

	count(1)
1	2719418

1 row

Based on the `count` result, you can see that the dataset has a total of 2,719,418 rows (also referred to as examples in Machine Learning literature). Looking at the table output from the previous query, you can see that the dataset contains 20 columns (also referred to as features).

Because all 20 columns are displayed, you can scroll the grid horizontally. Scroll until you see the **DepDel15** column. This column displays a 1 when the flight was delayed at least 15 minutes and 0 if there was no such delay. In the model you will construct, you will try to predict the value of this column for future data.

Let's execute another query that shows us how many rows do not have a value in the DepDel15 column.

```
%sql
select count(*) from bd2025_flight_delays_with_airport_codes where DepDel15 is null
```

_sqldf: pyspark.sql.dataframe.DataFrame = [count(1): long]

Table

	count(1)
1	27444

1 row

Notice that the `count` result is 27444. This means that 27,444 rows do not have a value in this column. Since this value is very important to our model, we will need to eliminate any rows that do not have a value for this column.

Next, scroll over to the **CRSDepTime** column within the table view above. Our model will approximate departure times to the nearest hour, but departure time is captured as an integer. For example, 8:37 am is captured as 837. Therefore, we will need to process the CRSDepTime column, and round it down to the nearest hour. To perform this rounding will require two steps, first you will need to divide the value by 100 (so that 837 becomes 8.37). Second, you will round this value down to the nearest hour (so that 8.37 becomes 8).

Finally, we do not need all 20 columns present in the flight_delays_with_airport_codes dataset, so we will pare down the columns, or features, in the dataset to the 12 we do need.

Using `%sql` magic allows us view and visualize the data, but for working with the data in our tables, we want to take advantage of the rich optimizations provided by DataFrames. Let's execute the same query using Spark SQL, this time saving the query to a DataFrame.

18

```
dfFlightDelays = spark.sql("select * from bd2025_flight_delays_with_airport_codes")
```

▸ dfFlightDelays: pyspark.sql.dataframe.DataFrame = [Year: string, Month: string ... 18 more fields]

Let's print the schema for the DataFrame.

20

```
pprint.pprint(dfFlightDelays.dtypes)
```

```
[('Year', 'string'),
 ('Month', 'string'),
 ('DayofMonth', 'string'),
 ('DayOfWeek', 'string'),
 ('Carrier', 'string'),
 ('CRSDepTime', 'string'),
 ('DepDelay', 'string'),
 ('DepDel15', 'string'),
 ('CRSArrTime', 'string'),
 ('ArrDelay', 'string'),
 ('ArrDel15', 'string'),
 ('Cancelled', 'string'),
 ('OriginAirportCode', 'string'),
 ('OriginAirportName', 'string'),
 ('OriginLatitude', 'string'),
 ('OriginLongitude', 'string'),
 ('DestAirportCode', 'string'),
 ('DestAirportName', 'string'),
 ('DestLatitude', 'string'),
 ('DestLongitude', 'string')]
```

Notice that the DepDel15 and CRSDepTime columns are both `string` data types. Both of these features need to be numeric, according to their descriptions above. We will cast these columns to their required data types next.

Perform data munging

To perform our data munging, we have multiple options, but in this case, we've chosen to take advantage of some useful features of R to perform the following tasks:

- `Remove rows with missing values`
  - `Generate a new column, named "CRSDepHour," which contains the rounded down value from CRSDepTime`
    - `Pare down columns to only those needed for our model`

SparkR is an R package that provides a light-weight frontend to use Apache Spark from R. To use SparkR we will call `library(SparkR)` within a cell that uses the `%r` magic, which denotes the language to use for the cell. The SparkR session is already configured, and all SparkR functions will talk to your attached cluster using the existing session.

25

```
%r
library(SparkR)

# Select only the columns we need, casting CRSDepTime as long and DepDel15 as int, into a new DataFrame
dfflights <- sql("SELECT OriginAirportCode, OriginLatitude, OriginLongitude, Month, DayofMonth, cast(CRSDepTime
as long) CRSDepTime, DayOfWeek, Carrier, DestAirportCode, DestLatitude, DestLongitude, cast(DepDel15 as int)
DepDel15 from bd2025_flight_delays_with_airport_codes")

# Delete rows containing missing values
dfflights <- na.omit(dfflights)
```

```
# Round departure times down to the nearest hour, and export the result as a new column named "CRSDepHour"
dfflights$CRSDepHour <- floor(dfflights$CRSDepTime / 100)

# Trim the columns to only those we will use for the predictive model
dfflightsClean = dfflights[, c("OriginAirportCode","OriginLatitude", "OriginLongitude", "Month", "DayofMonth",
"CRSDepHour", "DayOfWeek", "Carrier", "DestAirportCode", "DestLatitude", "DestLongitude", "DepDel15")]

createOrReplaceTempView(dfflightsClean, "flight_delays_view")
```

Now let's take a look at the resulting data. Take note of the **CRSDepHour** column that we created, as well as the number of columns we now have (12). Verify that the new CRSDepHour column contains the rounded hour values from our CRSDepTime column.

27

```
%sql
select * from flight_delays_view
```

▸ _sqldf: pyspark.sql.dataframe.DataFrame = [OriginAirportCode: string, OriginLatitude: string ... 10 more fields]

Table

	OriginAirportCode	OriginLatitude	OriginLongitude	Month	DayofMonth	CRSD
1	SAT	29.53388889	-98.46916667	5	1	
2	SAT	29.53388889	-98.46916667	5	2	
3	SAT	29.53388889	-98.46916667	5	3	
4	SAT	29.53388889	-98.46916667	5	4	
5	SAT	29.53388889	-98.46916667	5	5	
6	SAT	29.53388889	-98.46916667	5	6	
7	SAT	29.53388889	-98.46916667	5	7	

Now verify that the rows with missing data for the **DepDel15** column have been removed.

29

```
%sql
select count(*) from flight_delays_view
```

_sqldf: pyspark.sql.dataframe.DataFrame = [count(1): long]

Table

	count(1)
1	2691974

1 row

You should see a count of **2,691,974**. This is equal to the original 2,719,418 rows minus the 27,444 rows with missing data in the DepDel15 column.

Now save the contents of the temporary view into a new DataFrame.

31

```
dfFlightDelays_Clean = spark.sql("select * from flight_delays_view")
```

dfFlightDelays_Clean: pyspark.sql.dataframe.DataFrame = [OriginAirportCode: string, OriginLatitude: string ... 10 more fields]

Export the prepared data to persistent a global table

There are two types of tables in Databricks.

- Global tables, which are accessible across all clusters
- Local tables, which are available only within one cluster

To create a global table, you use the `saveAsTable()` method. To create a local table, you would use either the `createOrReplaceTempView()` or `registerTempTable()` method.

The `flight_delays_view` table was created as a local table using `createOrReplaceTempView`, and is therefore temporary. Local tables are tied to the Spark/SparkSQL Context that was used to create their associated DataFrame. When you shut down the SparkSession that is associated with the cluster (such as shutting down the cluster) then local, temporary tables will disappear. If we want our cleansed data to remain permanently, we should create a global table.

Run the following to copy the data from the source location into a global table named `flight_delays_clean`.

34

```
dfFlightDelays_Clean.write.mode("overwrite").saveAsTable("flight_delays_clean")
```

Prepare the weather data

To begin, take a look at the `flight_weather_with_airport_code` data that was imported to get a sense of the data we will be working with.

37

```
%sql
select * from db2025_flight_weather_with_airport_code
```

_sqldf: pyspark.sql.dataframe.DataFrame = [Year: string, Month: string ... 27 more fields]

Table

	Year	Month	Day	Time	TimeZone	SkyCondition	Visibility
1	2013	4	1	56	-4	FEW018 SCT044 BKN0...	10.00
2	2013	4	1	156	-4	FEW037 SCT070	10.00
3	2013	4	1	256	-4	FEW037 SCT070	10.00
4	2013	4	1	356	-4	FEW025 SCT070	10.00
5	2013	4	1	456	-4	FEW025	10.00
6	2013	4	1	556	-4	FEW025 SCT080	10.00
7	2013	4	1	656	-4	FEW028 BKN080	10.00
8	2013	4	1	756	-4	FEW028 BKN080	10.00
9	2013	4	1	856	-4	FEW030 BKN080	10.00
10	2013	4	1	956	-4	SCT035 BKN090	10.00
11	2013	4	1	1056	-4	SCT035 BKN090	10.00
12	2013	4	1	1156	-4	FEW026 BKN090	10.00
13	2013	4	1	1256	-4	FEW028 BKN090	10.00
14	2013	4	1	1356	-4	FEW040 BKN090	10.00
15							

9,328+ rows | Truncated data

Next, count the number of records so we know how many rows we are working with.

39

```
%sql
select count(*) from db2025_flight_weather_with_airport_code
```

_sqldf: pyspark.sql.dataframe.DataFrame = [count(1): long]

Table

	count(1)
1	406516

1 row

Observe that this data set has 406,516 rows and 29 columns. For this model, we are going to focus on predicting delays using WindSpeed (in MPH), SeaLevelPressure (in inches of Hg), and HourlyPrecip (in inches). We will focus on preparing the data for those features.

Let's start out by taking a look at the **WindSpeed** column. You may scroll through the values in the table above, but reviewing just the distinct values will be faster.

42

```
%sql
select distinct WindSpeed from db2025_flight_weather_with_airport_code
```

_sqldf: pyspark.sql.dataframe.DataFrame = [WindSpeed: string]

Table

	WindSpeed
1	7
2	15
3	11
4	29
5	3

Try clicking on the **WindSpeed** column header to sort the list by ascending and then by descending order. Observe that the values are all numbers, with the exception of some having `null` values and a string value of `M` for Missing. We will need to ensure that we remove any missing values and convert WindSpeed to its proper type as a numeric feature.

Next, let's take a look at the **SeaLevelPressure** column in the same way, by listing its distinct values.

45

```
%sql
select distinct SeaLevelPressure from db2025_flight_weather_with_airport_code
```

_sqldf: pyspark.sql.dataframe.DataFrame = [SeaLevelPressure: string]

Table

	SeaLevelPressure
1	29.68
2	29.45
3	30.43
4	29.58
5	30.59

Like you did before, click on the **SeaLevelPressure** column header to sort the values in ascending and then descending order. Observe that many of the features are of a numeric value (e.g., 29.96, 30.01, etc.), but some contain the string value of M for Missing. We will need to replace this value of "M" with a suitable numeric value so that we can convert this feature to be a numeric feature.

Finally, let's observe the **HourlyPrecip** feature by selecting its distinct values.

48

```
%sql
select distinct HourlyPrecip from db2025_flight_weather_with_airport_code
```

_sqldf: pyspark.sql.dataframe.DataFrame = [HourlyPrecip: string]

Table

	HourlyPrecip
1	0.55
2	0.07
3	0.75
4	0.59
5	1.53

Click on the column header to sort the list and ascending and then descending order. Observe that this column contains mostly numeric values, but also `null` values and values with `T` (for Trace amount of rain). We need to replace T with a suitable numeric value and convert this to a numeric feature.

Clean up weather data

To preform our data cleanup, we will execute a Python script, in which we will perform the following tasks:

- WindSpeed: Replace missing values with 0.0, and "M" values with 0.005
- HourlyPrecip: Replace missing values with 0.0, and "T" values with 0.005
- SeaLevelPressure: Replace "M" values with 29.92 (the average pressure)
- Convert WindSpeed, HourlyPrecip, and SeaLevelPressure to numeric columns
- Round "Time" column down to the nearest hour, and add value to a new column named "Hour"
- Eliminate unneeded columns from the dataset

Let's begin by creating a new DataFrame from the table. While we're at it, we'll pare down the number of columns to just the ones we need (AirportCode, Month, Day, Time, WindSpeed, SeaLevelPressure, HourlyPrecip).

53

```
dfWeather = spark.sql("select AirportCode, cast(Month as int) Month, cast(Day as int) Day, cast(Time as int)
Time, WindSpeed, SeaLevelPressure, HourlyPrecip from db2025_flight_weather_with_airport_code")
```

▸ dfWeather: pyspark.sql.dataframe.DataFrame = [AirportCode: string, Month: integer ... 5 more fields]

```
dfWeather.show()
```

```
|       SJU|    4|  1| 256|        9|           30.03|        null|
|       SJU|    4|  1| 356|        9|           30.03|        null|
|       SJU|    4|  1| 456|        7|           30.04|        null|
|       SJU|    4|  1| 556|        7|           30.05|        null|
|       SJU|    4|  1| 656|        9|           30.07|        null|
|       SJU|    4|  1| 756|       13|           30.10|        null|
|       SJU|    4|  1| 856|       14|           30.11|        null|
|       SJU|    4|  1| 956|       16|           30.12|        null|
|       SJU|    4|  1|1056|       17|           30.12|        null|
|       SJU|    4|  1|1156|       16|           30.10|        null|
|       SJU|    4|  1|1256|       16|           30.08|        null|
|       SJU|    4|  1|1356|       20|           30.05|        null|
|       SJU|    4|  1|1456|       18|           30.03|        null|
|       SJU|    4|  1|1556|       20|           30.02|        null|
|       SJU|    4|  1|1656|       15|           30.03|        null|
|       SJU|    4|  1|1756|       18|           30.03|        null|
|       SJU|    4|  1|1856|       11|           30.05|        null|
|       SJU|    4|  1|1956|       11|           30.06|        null|
+----------+-----+---+----+---------+----------------+------------+
only showing top 20 rows
```

Review the schema of the dfWeather DataFrame

56

```
pprint.pprint(dfWeather.dtypes)
```

```
[('AirportCode', 'string'),
 ('Month', 'int'),
 ('Day', 'int'),
 ('Time', 'int'),
 ('WindSpeed', 'string'),
 ('SeaLevelPressure', 'string'),
 ('HourlyPrecip', 'string')]
```

57

```
# Round Time down to the next hour, since that is the hour for which we want to use flight data. Then, add the
rounded Time to a new column named "Hour", and append that column to the dfWeather DataFrame.
df = dfWeather.withColumn('Hour', F.floor(dfWeather['Time']/100))

# Replace any missing HourlyPrecip and WindSpeed values with 0.0
df = df.fillna('0.0', subset=['HourlyPrecip', 'WindSpeed'])

# Replace any WindSpeed values of "M" with 0.005
df = df.replace('M', '0.005', 'WindSpeed')

# Replace any SeaLevelPressure values of "M" with 29.92 (the average pressure)
df = df.replace('M', '29.92', 'SeaLevelPressure')

# Replace any HourlyPrecip values of "T" (trace) with 0.005
df = df.replace('T', '0.005', 'HourlyPrecip')

# Be sure to convert WindSpeed, SeaLevelPressure, and HourlyPrecip columns to float
# Define a new DataFrame that includes just the columns being used by the model, including the new Hour feature
dfWeather_Clean = df.select('AirportCode', 'Month', 'Day', 'Hour', df['WindSpeed'].cast('float'), df
['SeaLevelPressure'].cast('float'), df['HourlyPrecip'].cast('float'))
```

▸ df: pyspark.sql.dataframe.DataFrame = [AirportCode: string, Month: integer ... 6 more fields]
▸ dfWeather_Clean: pyspark.sql.dataframe.DataFrame = [AirportCode: string, Month: integer ... 5 more fields]

Now let's take a look at the new `dfWeather_Clean` DataFrame.

```
display(dfWeather_Clean)
```

Table

	AirportCode	Month	Day	Hour	WindSpeed	SeaLevelPressure	Hour
1	SJU	4	1	0	13	30.059999465942383	0.004999
2	SJU	4	1	1	10	30.049999237060547	
3	SJU	4	1	2	9	30.030000686645508	
4	SJU	4	1	3	9	30.030000686645508	
5	SJU	4	1	4	7	30.040000915527344	
6	SJU	4	1	5	7	30.049999237060547	
7	SJU	4	1	6	9	30.06999969482422	
8	SJU	4	1	7	13	30.100000381469727	
9	SJU	4	1	8	14	30.110000610351562	
10	SJU	4	1	9	16	30.1200008392334	
11	SJU	4	1	10	17	30.1200008392334	
12	SJU	4	1	11	16	30.100000381469727	
13	SJU	4	1	12	16	30.079999923706055	
14	SJU	4	1	13	20	30.049999237060547	
15							

10,000+ rows | Truncated data

Observe that the new DataFrame only has 7 columns. Also, the WindSpeed, SeaLevelPressure, and HourlyPrecip fields are all numeric and contain no missing values. To ensure they are indeed numeric, we can take a look at the DataFrame's schema.

```
pprint.pprint(dfWeather_Clean.dtypes)
```

```
[('AirportCode', 'string'),
 ('Month', 'int'),
 ('Day', 'int'),
 ('Hour', 'bigint'),
 ('WindSpeed', 'float'),
 ('SeaLevelPressure', 'float'),
 ('HourlyPrecip', 'float')]
```

Now let's persist the cleaned weather data to a persistent global table.

```
63
dfWeather_Clean.write.mode("overwrite").saveAsTable("flight_weather_clean")
```

```
64
dfWeather_Clean.select("*").count()
```

```
Out[29]: 406516
```

Join the Flight and Weather datasets

With both datasets ready, we want to join them together so that we can associate historical flight delays with the weather data at departure time.

```
dfFlightDelaysWithWeather = spark.sql("SELECT d.OriginAirportCode, \
                d.Month, d.DayofMonth, d.CRSDepHour, d.DayOfWeek, \
                d.Carrier, d.DestAirportCode, d.DepDel15, w.WindSpeed, \
                w.SeaLevelPressure, w.HourlyPrecip \
                FROM flight_delays_clean d \
                INNER JOIN flight_weather_clean w ON \
                d.OriginAirportCode = w.AirportCode AND \
                d.Month = w.Month AND \
                d.DayofMonth = w.Day AND \
                d.CRSDepHour = w.Hour")
```

dfFlightDelaysWithWeather: pyspark.sql.dataframe.DataFrame = [OriginAirportCode: string, Month: string ... 9 more fields]

Now let's take a look at the combined data.

```
69
display(dfFlightDelaysWithWeather)
```

Write the combined dataset to a new persistent global table.

```
71
dfFlightDelaysWithWeather.write.mode("overwrite").saveAsTable("flight_delays_with_weather")
```

E.2 Exercise 2 / 02 Train and Evaluate Models

Train the model

Margie's Travel wants to build a model to predict if a departing flight will have a 15-minute or greater delay. In the historical data they have provided, the indicator for such a delay is found within the DepDel15 (where a value of 1 means delay, 0 means no delay). To create a model that predicts such a binary outcome, we can choose from the various Two-Class algorithms provided by Spark MLlib. For our purposes, we choose Decision Tree. This type of classification module needs to be first trained on sample data that includes the features important to making a prediction and must also include the actual historical outcome for those features.

The typical pattern is to split the historical data so a portion is shown to the model for training purposes, and another portion is reserved to test just how well the trained model performs against examples it has not seen before.

To start, let's import the Python libraries and modules we will use in this notebook.

5

```
from pyspark.ml import Pipeline, PipelineModel
from pyspark.ml.feature import OneHotEncoder, StringIndexer, VectorAssembler
from pyspark.sql.functions import array, col, lit
```

Load the cleaned flight and weather data

Load the data from the global table

8

```
dfDelays = spark.sql("select OriginAirportCode, cast(Month as int) Month, cast(DayofMonth as int) DayofMonth, CRSDepHour, cast
(DayOfWeek as int) DayOfWeek, Carrier, DestAirportCode, DepDel15, WindSpeed, SeaLevelPressure, HourlyPrecip from
flight_delays_with_weather")
cols = dfDelays.columns
```

dfDelays: pyspark.sql.dataframe.DataFrame = [OriginAirportCode: string, Month: integer ... 9 more fields]

9

```
display(dfDelays)
```

Table

	OriginAirportCode	Month	DayofMonth	CRSDepHour	DayOfWeek	Carrier	DestAirportCode
1	DTW	4	19	8	5	DL	MIA
2	DTW	4	19	8	5	DL	MIA
3	SLC	4	19	17	5	DL	JFK
4	PDX	4	19	6	5	DL	SLC
5	PDX	4	19	6	5	DL	SLC

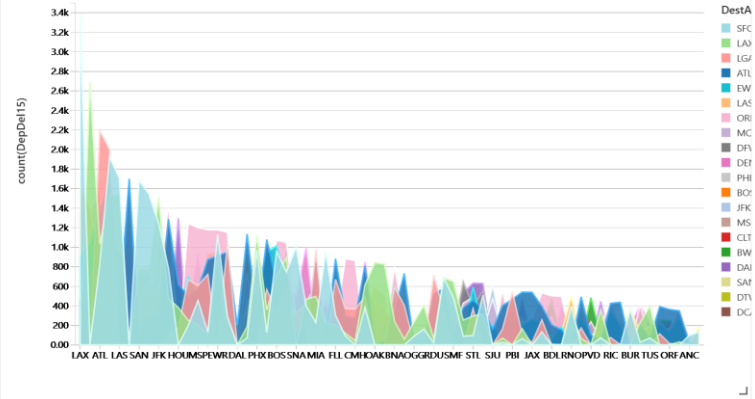
We can get a sense of which origin and destination airports suffer the most delays by querying against the table and displaying the output as an area chart. We've already configured the chart's settings so you should see a nice visual when you run the below command. If it displays in a table instead, just select the area chart option below the table.

11

```
%sql
select OriginAirportCode, DestAirportCode, count(DepDel15)
from flight_delays_with_weather where DepDel15 = 1
group by OriginAirportCode, DestAirportCode
ORDER BY count(DepDel15) desc
```

_sqldf: pyspark.sql.dataframe.DataFrame = [OriginAirportCode: string, DestAirportCode: string ... 1 more field]

Visualization



Truncated to 20 series.

This result is stored as `_sqldf` and can be used in other Python cells.

Sampling the data

To begin, let's evaluate the data to compare the flights that are delayed (DepDel15) to those that are not. What we're looking for is whether one group has a much higher count than the other.

14

```
dfDelays.groupBy("DepDel15").count().show()
```

```
+--------+-------+
|DepDel15|  count|
+--------+-------+
|       1| 591608|
|       0|2267686|
+--------+-------+
```

Judging by the delay counts, there are almost four times as many non-delayed records as there are delayed.

We want to ensure our model is sensitive to the delayed samples. To do this, we can use stratified sampling provided by the `sampleBy()` function. First we create fractions of each sample type to be returned. In our case, we want to keep all instances of delayed (value of 1) and downsample the not delayed instances to 30%.

16

```
fractions = {0: .30, 1: 1.0}
trainingSample = dfDelays.sampleBy("DepDel15", fractions, 36)
trainingSample.groupBy("DepDel15").count().show()
```

▸ trainingSample: pyspark.sql.dataframe.DataFrame = [OriginAirportCode: string, Month: integer ... 9 more fields]

```
+--------+------+
|DepDel15| count|
+--------+------+
|       1|591608|
|       0|681221|
+--------+------+
```

You can see that the number of delayed and not delayed instances are now much closer to each other. This should result in a better-trained model.

Select an algorithm and transform features

Because we are trying to predict a binary label (flight delayed or not delayed), we need to use binary classification. For this, we will be using the Decision Tree classifier algorithm provided by the Spark MLlib library. We will also be using the Pipelines API to put our data through all of the required feature transformations in a single call. The Pipelines API provides higher-level API built on top of DataFrames for constructing ML pipelines.

In the data cleaning phase, we identified the important features that most contribute to the classification. The `flight_delays_with_weather` is the result of the data preparation and feature identification process. The features are:

OriginAirportCode	Month	DayofMonth	CRSDepHour	DayOfWeek	Carrier	DestAirportCode	WindSpeed	SeaLevelPressure	HourlyPrecip
LGA	5	2	13	4	MQ	ORD	6	29.8	0.05

We also have a label named `DepDelay15` which equals 0 if no delay, and 1 if there was a delay.

As you can see, this dataset contains nominal variables like OriginAirportCode (LGA, MCO, ORD, ATL, etc). In order for the machine learning algorithm to use these nominal variables, they need to be transformed and put into Feature Vectors, or vectors of numbers representing the value for each feature.

22

```
categoricalColumns = ["OriginAirportCode", "Carrier", "DestAirportCode"]
stages = [] # stages in our Pipeline
for categoricalCol in categoricalColumns:
    # Category Indexing with StringIndexer
    stringIndexer = StringIndexer(inputCol=categoricalCol, outputCol=categoricalCol + "Index")
    # Use OneHotEncoderEstimator to convert categorical variables into binary SparseVectors
    encoder = OneHotEncoder(dropLast=False, inputCols=[stringIndexer.getOutputCol()], outputCols=[categoricalCol + "classVec"])
    # Add stages.  These are not run here, but will run all at once later on.
    stages += [stringIndexer, encoder]

# Convert label into label indices using the StringIndexer
label_stringIdx = StringIndexer(inputCol="DepDel15", outputCol="label")
stages += [label_stringIdx]
```

Now we need to use the `VectorAssembler` to combine all the feature columns into a single vector column. This includes our numeric columns as well as the one-hot encoded binary vector columns.

24

```
# Transform all features into a vector using VectorAssembler
numericCols = ["Month", "DayofMonth", "CRSDepHour", "DayOfWeek", "WindSpeed", "SeaLevelPressure", "HourlyPrecip"]
assemblerInputs = [c + "classVec" for c in categoricalColumns] + numericCols
assembler = VectorAssembler(inputCols=assemblerInputs, outputCol="features")
stages += [assembler]
```

Create and train the Decision Tree model

Before we can train our model, we need to randomly split our data into test and training sets. As is standard practice, we will allocate a larger portion (70%) for training. A seed is set for reproducibility, so the outcome is the same (barring any changes) each time this cell and subsequent cells are run.

Remember to use our stratified sample (`trainingSample`).

27

```
### Randomly split data into training and test sets. set seed for reproducibility
(trainingData, testData) = trainingSample.randomSplit([0.7, 0.3], seed=100)
# We want to have two copies of the training and testing data, since the pipeline runs transformations and we want to run a couple
different iterations
trainingData2 = trainingData
testData2 = testData
print(trainingData.count())
print(testData.count())
```

▸ testData: pyspark.sql.dataframe.DataFrame = [OriginAirportCode: string, Month: integer ... 9 more fields]
▸ testData2: pyspark.sql.dataframe.DataFrame = [OriginAirportCode: string, Month: integer ... 9 more fields]
▸ trainingData: pyspark.sql.dataframe.DataFrame = [OriginAirportCode: string, Month: integer ... 9 more fields]
▸ trainingData2: pyspark.sql.dataframe.DataFrame = [OriginAirportCode: string, Month: integer ... 9 more fields]

```
891009
381820
```

Our pipeline is ready to be built and run, now that we've created all the transformation stages. We just have one last stage to add, which is the Decision Tree. Let's run the pipeline to put the data through all the feature transformations within a single call.

Calling `pipeline.fit(trainingData)` will transform the test data and use it to train the Decision Tree model.

We will also use the MLflow library to track the details of this experiment, including testing results and the model we create.

```
from pyspark.ml.classification import DecisionTreeClassifier
import mlflow
import mlflow.spark

mlflow.start_run()

# Create initial Decision Tree Model
dt = DecisionTreeClassifier(labelCol="label", featuresCol="features", maxDepth=3)
stages += [dt]

# Create a Pipeline.
pipeline = Pipeline(stages=stages)
# Run the feature transformations.
#  - fit() computes feature statistics as needed.
#  - transform() actually transforms the features.
pipelineModel = pipeline.fit(trainingData)
trainingData = pipelineModel.transform(trainingData)
# Keep relevant columns
selectedcols = ["label", "features"] + cols
trainingData = trainingData.select(selectedcols)
display(trainingData)
```

▸ trainingData: pyspark.sql.dataframe.DataFrame

Table

	1.2 label	features	OriginAirpo
1	0	{"vectorType":"sparse","length":157,"indices":[45,70,86,150,151,152,153,154,155],"values":[1,1,1,5,1,6,3,7,29.799999237060547]}	ABQ
2	1	{"vectorType":"sparse","length":157,"indices":[45,70,84,150,151,152,153,154,155],"values":[1,1,1,5,1,9,3,5,29.799999237060547]}	ABQ
3	0	{"vectorType":"sparse","length":157,"indices":[45,70,86,150,151,152,153,154,155],"values":[1,1,1,5,1,12,3,13,29.75]}	ABQ
4	0	{"vectorType":"sparse","length":157,"indices":[45,70,82,150,151,152,153,154,155],"values":[1,1,1,5,1,16,3,17,29.719999313354492]}	ABQ
5	0	{"vectorType":"sparse","length":157,"indices":[45,70,104,150,151,152,153,154,155],"values":[1,1,1,5,1,17,3,25,29.75]}	ABQ

Let's make predictions on our test dataset using the `transform()` , which will only use the 'features' column. We'll display the prediction's schema afterward so you can see the three new prediction-related columns.

31

```
# Make predictions on test data using the Transformer.transform() method.
predictions = pipelineModel.transform(testData)
```

▸ predictions: pyspark.sql.dataframe.DataFrame

To properly train the model, we need to determine which parameter values of the decision tree produce the best model. A popular way to perform model selection is k-fold cross validation, where the data is randomly split into k partitions. Each partition is used once as the testing data set, while the rest are used for training. Models are then generated using the training sets and evaluated with the testing sets, resulting in k model performance measurements. The model parameters leading to the highest performance metric produce the best model.

We can use `BinaryClassificationEvaluator` to evaluate our model. We can set the required column names in `rawPredictionCol` and `labelCol` Param and the metric in `metricName` Param.

Let's evaluate the Decision Tree model with `BinaryClassificationEvaluator` .

34

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator
# Evaluate model
evaluator = BinaryClassificationEvaluator()
areaUnderRoc = evaluator.evaluate(predictions)
mlflow.log_metric("Area Under ROC", areaUnderRoc)
areaUnderRoc
```

```
Out[14]: 0.6242370457862387
```

Finally, we will save the model to disk and end the first MLflow run.

36

```
mlflow.spark.log_model(pipelineModel, "model")
modelpath = "/dbfs/mlflow/mt/model-dtree"
mlflow.spark.save_model(pipelineModel, modelpath)
mlflow.end_run()
```

```
2025/05/18 12:03:23 INFO mlflow.spark: Inferring pip requirements by reloading the logged model from the databricks artifact repository, whi
ch can be time-consuming. To speed up, explicitly specify the conda_env or pip_requirements when calling log_model().
```

Loading widget. This should take less than 30 seconds.

```
2025/05/18 12:04:26 WARNING mlflow.utils.environment: Encountered an unexpected error while inferring pip requirements (model URI: dbfs:/dat
abricks/mlflow-tracking/2478036556572309/82463ef339aa4c89931053d161fe75e0/artifacts/model/sparkml, flavor: spark). Fall back to return ['pys
park==3.3.0', 'pandas<2']. Set logging level to DEBUG to see the full traceback.
```

Loading widget. This should take less than 30 seconds.

Now we will try tuning the model with the `ParamGridBuilder` and the `CrossValidator`.

As we indicate 3 values for maxDepth and 3 values for maxBin, this grid will have 3 x 3 = 9 parameter settings for `CrossValidator` to choose from. We will create a 3-fold CrossValidator.

38

```
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

# Create ParamGrid for Cross Validation
```

```
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
paramGrid = (ParamGridBuilder()
             .addGrid(dt.maxDepth, [1, 2, 6, 10])
             .addGrid(dt.maxBins, [20, 40, 80])
             .build())
```

Run the cell below to create your 3-fold CrossValidator and use it to run cross validations. It can take **up to 5 minutes** to execute the cell.

Because we are training a new model, we will do this in another run of the same experiment. That way, we can compare the cross-validated version to the original decision tree and choose the better model for deployment.

40

```
mlflow.start_run()

# Create 3-fold CrossValidator
cv = CrossValidator(estimator=pipeline, estimatorParamMaps=paramGrid, evaluator=evaluator, numFolds=3)

# Run cross validations (this can take several minutes to execute)
cvModel = cv.fit(trainingData2)
```

Loading widget. This should take less than 30 seconds.

Loading widget. This should take less than 30 seconds.

Loading widget. This should take less than 30 seconds.

Now let's create new predictions with which to measure the accuracy of our model.

42

```
predictions = cvModel.transform(testData2)
```

▸ predictions: pyspark.sql.dataframe.DataFrame

We'll use the predictions to evaluate the best model. `cvModel` uses the best model found from the Cross Validation.

44

```
areaUnderRoc = evaluator.evaluate(predictions)
mlflow.log_metric("Area Under ROC", areaUnderRoc)
areaUnderRoc
```

```
Out[19]: 0.6206943472485974
```

Now let's view the best model's predictions and probabilities of each prediction class.

46

```
selected = predictions.select("label", "prediction", "probability", "OriginAirportCode", "DestAirportCode")
display(selected)
```

▸ selected: pyspark.sql.dataframe.DataFrame

Table

	1.2 label	1.2 prediction	probability	OriginAirportCode	DestAi
1	0	1	> {"vectorType":"dense","length":2,"values":[0.4393857931280043,0.56061420687199...	ABQ	LAX
2	0	1	> {"vectorType":"dense","length":2,"values":[0.4393857931280043,0.56061420687199...	ABQ	SLC

We need to take the best model from `cvModel` and generate predictions for the entire dataset (dfDelays), then evaluate the best model.

48

```
bestModel = cvModel.bestModel
finalPredictions = bestModel.transform(dfDelays)
areaUnderRoc = evaluator.evaluate(finalPredictions)
mlflow.log_metric("Final Area Under ROC", areaUnderRoc)
areaUnderRoc
```

▸ finalPredictions: pyspark.sql.dataframe.DataFrame

```
Out[21]: 0.6216832285847772
```

Finally, we will save this model to disk and end the run.

50

```
mlflow.spark.log_model(bestModel, "model")
modelpath = "/dbfs/mlflow/mt/model-dtree-cv"
mlflow.spark.save_model(bestModel, modelpath)
mlflow.end_run()
```

```
2025/05/18 14:44:21 INFO mlflow.spark: Inferring pip requirements by reloading the logged model from the databricks artifact repository, whi
ch can be time-consuming. To speed up, explicitly specify the conda_env or pip_requirements when calling log_model().
```

Loading widget. This should take less than 30 seconds.

```
2025/05/18 14:47:03 WARNING mlflow.utils.environment: Encountered an unexpected error while inferring pip requirements (model URI: dbfs:/dat
abricks/mlflow-tracking/2478036556572309/cc666e0ad98b4687a8774bc8255b7810/artifacts/model/sparkml, flavor: spark). Fall back to return ['pys
park==3.3.0', 'pandas<2']. Set logging level to DEBUG to see the full traceback.
```

Loading widget. This should take less than 30 seconds.

Save the model for batch scoring

There are two reasons for saving the model in this lab. The first is so you can access the trained model later if your cluster restarts for any reason, and also from within another notebook. Secondly, we will need to make the model available externally so we can perform batch scoring against it in Exercise 5. Save the model to DBFS so it can be accessed across any clusters in the Databricks Workspace.

NOTE: Save the model in the root of DBFS as this is where Spark Pipelines will look by default.

53

```
# Save the best model under /dbfs/flightDelayModel
bestModel.write().overwrite().save("/flightDelayModel")
```

E.3 Exercise 2 / 02 Train and Evaluate Models-RandomForest

Let's evaluate the Decision Tree model with `BinaryClassificationEvaluator`.

34

```
from pyspark.ml.evaluation import BinaryClassificationEvaluator
# Evaluate model
evaluator = BinaryClassificationEvaluator()
areaUnderRoc = evaluator.evaluate(predictions)
mlflow.log_metric("Area Under ROC", areaUnderRoc)
areaUnderRoc
```

```
Out[12]: 0.6806920953466743
```

Finally, we will save the model to disk and end the first MLflow run.

36

```
mlflow.spark.log_model(pipelineModel, "model-rftree")
modelpath = "/dbfs/mlflow/mt/model-rftree"
mlflow.spark.save_model(pipelineModel, modelpath)
mlflow.end_run()
```

```
2025/05/18 23:59:30 INFO mlflow.spark: Inferring pip requirements by reloading the logged model from the databricks artifact repository, whi
ch can be time-consuming. To speed up, explicitly specify the conda_env or pip_requirements when calling log_model().
```

As we indicate 3 values for maxDepth and 2 values for maxBin, this grid will have 3 x 2 = 6 parameter settings for `CrossValidator` to choose from. We will create a 3-fold CrossValidator.

38

```
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

# Create ParamGrid for Cross Validation
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
paramGrid = (ParamGridBuilder()
             .addGrid(dt.maxDepth, [1, 5, 10])
             .addGrid(dt.maxBins, [20, 40])
             .build())
```

Run the cell below to create your 3-fold CrossValidator and use it to run cross validations. It can take **up to 5 minutes** to execute the cell.

Because we are training a new model, we will do this in another run of the same experiment. That way, we can compare the cross-validated version to the original decision tree and choose the better model for deployment.

40

```
mlflow.start_run()

# Create 3-fold CrossValidator
cv = CrossValidator(estimator=pipeline, estimatorParamMaps=paramGrid, evaluator=evaluator, numFolds=3)

# Run cross validations (this can take several minutes to execute)
cvModel = cv.fit(trainingData2)
```

Loading widget. This should take less than 30 seconds.

Now let's create new predictions with which to measure the accuracy of our model.

```
42
predictions = cvModel.transform(testData2)
```

▸ predictions: pyspark.sql.dataframe.DataFrame

We'll use the predictions to evaluate the best model. `cvModel` uses the best model found from the Cross Validation.

```
44
areaUnderRoc = evaluator.evaluate(predictions)
mlflow.log_metric("Area Under ROC", areaUnderRoc)
areaUnderRoc
```

```
Out[21]: 0.710591154024279
```

Now let's view the best model's predictions and probabilities of each prediction class.

```
46
selected = predictions.select("label", "prediction", "probability", "OriginAirportCode", "DestAirportCode")
display(selected)
```

▸ selected: pyspark.sql.dataframe.DataFrame

Table

	1.2 label	1.2 prediction	probability	OriginAirportCode	DestAi
1	0	0	› {"vectorType":"dense","length":2,"values":[0.5235702193484856,0.47642978065151...	ABQ	LAX
2	0	0	› {"vectorType":"dense","length":2,"values":[0.5694228026576182,0.4305771973423 8...	ABQ	SLC
3	0	0	› {"vectorType":"dense","length":2,"values":[0.5321455808557082,0.46785441914429...	ABQ	LAX
4	1	0	› {"vectorType":"dense","length":2,"values":[0.5425796237890571,0.45742037621094...	ABQ	ORD
5	1	0	› {"vectorType":"dense","length":2,"values":[0.5336425100999682,0.46635748990003...	ABQ	IAH
6	0	0	› {"vectorType":"dense","length":2,"values":[0.5747763050156 05,0.4252236949843 95...	ABQ	SLC
7	0	0	› {"vectorType":"dense","length":2,"values":[0.5542514000082905,0.44574859999170...	ABQ	PHX
8	1	0	› {"vectorType":"dense","length":2,"values":[0.5062033458663 3,0.4937966541336 7]}	ABQ	LAX
9	0	0	› {"vectorType":"dense","length":2,"values":[0.5226701115748791,0.47732988842512...	ABQ	ORD
10	0	0	› {"vectorType":"dense","length":2,"values":[0.6529117064273123,0.34708829357268...	ABQ	SLC
11	0	0	› {"vectorType":"dense","length":2,"values":[0.5024105641050218,0.49758943589497...	ABQ	LAX
12	0	0	› {"vectorType":"dense","length":2,"values":[0.5317959734875106,0.46820402651248...	ABQ	PHX
13	0	0	› {"vectorType":"dense","length":2,"values":[0.5328407233974988,0.46715927660250...	ABQ	SLC
14	1	0	› {"vectorType":"dense","length":2,"values":[0.5063214579090064,0.49367854209099...	ABQ	DEN
15					

10,000+ rows | Truncated data

E.4 Exercise 2 / 03 Deploy as Web Service

Deploy model to Azure Databricks

In this notebook, you will deploy the best performing model you selected previously as a web service hosted in in Azure Databricks cluster.

```
4
import mlflow
import mlflow.spark
from mlflow.tracking import MlflowClient
import time
from mlflow.entities.model_registry.model_version_status import ModelVersionStatus

client = MlflowClient()
```

First, get the experiment from the prior notebook.

```
6
user_name = dbutils.notebook.entry_point.getDbutils().notebook().getContext().tags().apply('user')
experiment_name = "/Users/{user_name}/BigDataVis2/Exercise 2/02 Train and Evaluate Models".format(user_name=user_name)

experiment = client.get_experiment_by_name(experiment_name)
```

Get the latest run from the experiment--as a reminder, this is the cross-validation model. In order to retrieve the model itself, you will need to get the **parent** run ID, as that is where the model details are stored.

```
experiment_id = experiment.experiment_id
runs_df = client.search_runs(experiment_id, order_by=["attributes.start_time desc"], max_results=1)
# run_id = runs_df[0].data.tags["mlflow.parentRunId"]
run_id = runs_df[0].data.tags.get("mlflow.parentRunId", runs_df[0].info.run_id)
model_name = "Delay Estimator"

artifact_path = "model"
model_uri = "runs:/{run_id}/{artifact_path}".format(run_id=run_id, artifact_path=artifact_path)
model_uri
```

```
Out[11]: 'runs:/42664c06e20a4f2da45afeedd73f50d7/model'
```

The next step is to register this model with MLflow. This may take anywhere fdrom 30 seconds to 5 minutes to complete.

10

```
model_details = mlflow.register_model(model_uri=model_uri, name=model_name)

# Wait until the model is ready
def wait_until_ready(model_name, model_version):
  client = MlflowClient()
  for _ in range(10):
    model_version_details = client.get_model_version(
      name=model_name,
      version=model_version,
    )
    status = ModelVersionStatus.from_string(model_version_details.status)
    print("Model status: %s" % ModelVersionStatus.to_string(status))
    if status == ModelVersionStatus.READY:
      break
    time.sleep(1)

wait_until_ready(model_details.name, model_details.version)
```

```
Successfully registered model 'Delay Estimator'.
```

```
2025/05/18 19:06:22 INFO mlflow.store.model_registry.abstract_store: Waiting up to 300 seconds for model version to finish creation. Model n
ame: Delay Estimator, version 1
Model status: READY
Created version '1' of model 'Delay Estimator'.
```

Now that the model is registered, move it to Production. This will make the current model the production model, allowing you to serve this iteration of the model.

12

```
client.transition_model_version_stage(
  name=model_details.name,
  version=model_details.version,
  stage='Production',
)
model_version_details = client.get_model_version(
  name=model_details.name,
  version=model_details.version,
)
print("The current model stage is: '{stage}'".format(stage=model_version_details.current_stage))
```

```
<command-2478036556572374>:1: FutureWarning: ``mlflow.tracking.client.MlflowClient.transition_model_version_stage`` is deprecated since 2.9.
0. Model registry stages will be removed in a future major release. To learn more about the deprecation of model registry stages, see our mi
gration guide here: https://mlflow.org/docs/latest/model-registry.html#migrating-from-stages
  client.transition_model_version_stage(
The current model stage is: 'Production'
```

Serve the Model

You have already created a model, but the next step will be to serve the model. At present, the best way to do this is to select the **Models** menu option on the left-hand pane. Note that you will only see this menu in the **Machine Learning** view. If you are still in the **Data Science & Engineering** view, select the drop-down option from the menu and select **Machine Learning** first.

In the Models menu, select the **Delay Estimator** model. Navigate to the **Serving** menu and enable serving. This will build a Databricks cluster to allow you to perform inference.

Wait for both the Status indicator as well as the Production model indicator to read Ready and then copy the Model URL with "Production" in it. This will take 5-10 minutes to complete.

Test the scoring web service

In order to test the service, create two sample rows for testing and load them into a Pandas DataFrame.

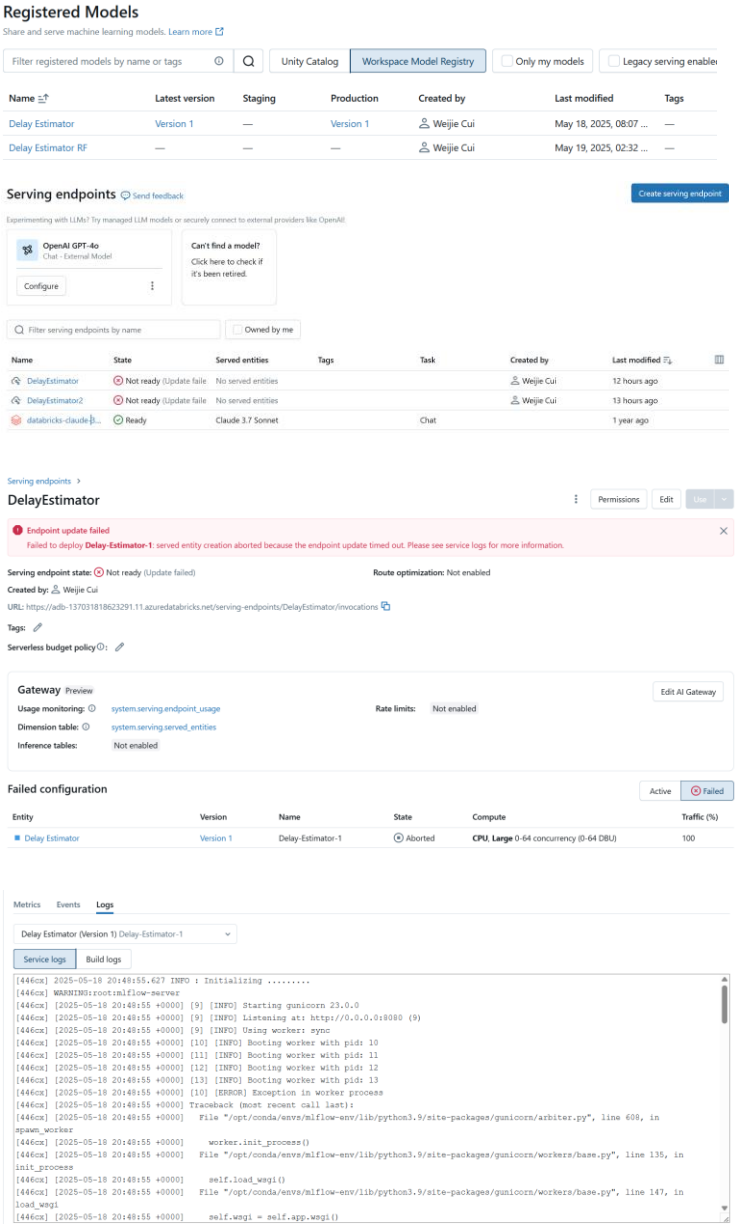
17

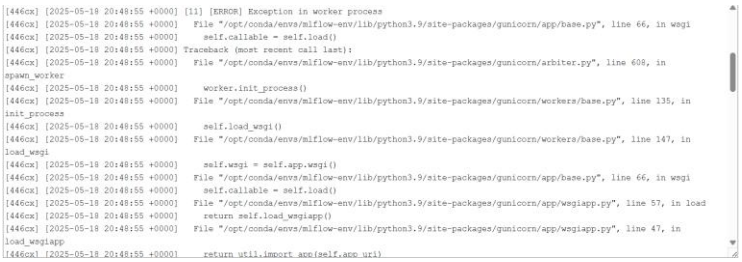
```
import json
import pandas as pd

# Create two records for testing the prediction
test_input1 = {"OriginAirportCode":"SAT","Month":5,"DayofMonth":5,"CRSDepHour":13,"DayOfWeek":7,"Carrier":"MQ",
"DestAirportCode":"ORD","WindSpeed":9,"SeaLevelPressure":30.03,"HourlyPrecip":0}

test_input2 = {"OriginAirportCode":"ATL","Month":2,"DayofMonth":5,"CRSDepHour":8,"DayOfWeek":4,"Carrier":"MQ","DestAirportCode":"MCO",
"WindSpeed":3,"SeaLevelPressure":31.03,"HourlyPrecip":0}

# package the inputs into a JSON string and test run() in local notebook
inputs = pd.DataFrame([test_input1, test_input2])
```

Fill in the values for `url` and `personal_access_token` in the function below and then run the following command to ensure that you get back results from the serving cluster. The `url` is the model serving URL you created in the **Serve the Model** task above, and `personal_access_token` is the PAT you created in the hands-on lab.

```
19
1  import os
2  import requests
3  import numpy as np
4
5  def create_tf_serving_json(data):
6    return {'inputs': {name: data[name].tolist() for name in data.keys()} if isinstance(data, dict) else data.tolist()}
7
8  def score_model(dataset):
9    # url = 'https://adb-1404434138497066.6.azuredatabricks.net/model/Delay%20Estimator/Production/invocations'
10   # personal_access_token = 'dapi20907ab7c44595a3e0c57984497889d7-3' # Enter your Personal Access Token here
11   url = 'https://adb-1370318186232911.11.azuredatabricks.net/serving-endpoints/DelayEstimator/invocations' # Enter your URL here
12   personal_access_token = 'dapi6bd599988c86f34baaaf32a125721b47' # Enter your Personal Access Token here
13   headers = {'Authorization': f'Bearer {personal_access_token}'}
14   data_json = dataset.to_dict(orient='split') if isinstance(dataset, pd.DataFrame) else create_tf_serving_json(dataset)
15   response = requests.request(method='POST', headers=headers, url=url, json=data_json)
16   if response.status_code != 200:
17     raise Exception(f'Request failed with status {response.status_code}, {response.text}')
18   return response.json()
19
20 score_model(inputs)

> Exception: Request failed with status 404, {"error_code":"RESOURCE_DOES_NOT_EXIST","message":"The given endpoint does not exist."}
```

According to the logs, the model deployment failed due to the conflict of Python packages in the base image.





E.5 Exercise 5 / 01 Deploy for Batch Scoring

Exercise 5 / 01 Deploy for Batch Scoring (Python)

For the batch scoring, we will persist the values in a new global persistent Databricks table. In production data workloads, you may save the scored data to Blob Storage, Azure Cosmos DB, or other serving layer. Another implementation detail we are skipping for the lab is processing only new files. This can be accomplished by creating a widget in the notebook that accepts a path parameter that is passed in from Azure Data Factory.

3

```
from pyspark.ml import Pipeline, PipelineModel
from pyspark.ml.feature import OneHotEncoder, StringIndexer, VectorAssembler, Bucketizer
from pyspark.sql.functions import array, col, lit
from pyspark.sql.types import *
```

Replace STORAGE-ACCOUNT-NAME with the name of your storage account. You can find this in the Azure portal by locating the storage account that you created in the lab setup, within your resource group. The container name is set to the default used for this lab. If yours is different, update the containerName variable accordingly.

5

```
accountName = "bd2025store" # STORAGE-ACCOUNT-NAME
containerName = "sparkcontainer"
```

Define the schema for the CSV files

```
data_schema = StructType([
        StructField('OriginAirportCode',StringType()),
        StructField('Month', IntegerType()),
        StructField('DayofMonth', IntegerType()),
        StructField('CRSDepHour', IntegerType()),
        StructField('DayOfWeek', IntegerType()),
        StructField('Carrier', StringType()),
        StructField('DestAirportCode', StringType()),
        StructField('DepDel15', IntegerType()),
        StructField('WindSpeed', DoubleType()),
        StructField('SeaLevelPressure', DoubleType()),
        StructField('HourlyPrecip', DoubleType())])
```

Create a new DataFrame from the CSV files, applying the schema

9

```
dfDelays = spark.read.csv("wasbs://" + containerName + "@" + accountName + ".blob.core.windows.net/FlightsAndWeather/*/*/
FlightsAndWeather.csv",
                    schema=data_schema,
                    sep=",",
                    header=True)
```

▸ dfDelays: pyspark.sql.dataframe.DataFrame = [OriginAirportCode: string, Month: integer ... 9 more fields]

Load the trained machine learning model you created earlier in the lab

11

```
# Load the saved pipeline model
model = PipelineModel.load("/flightDelayModel")
```

Make a prediction against the loaded data set

13

```
# Make a prediction against the dataset
prediction = model.transform(dfDelays)
```

▸ prediction: pyspark.sql.dataframe.DataFrame

Save the scored data into a new global table called **scoredflights**

15

```
prediction.write.mode("overwrite").saveAsTable("scoredflights")
```

F File of Exploring Data

Exercise 6 / 01 Explore Data (Python)

Summarize data using Azure Databricks

Select the scored data generated by the Azure Data Factory pipeline

```
%sql
select * from scoredflights
```

_sqldf: pyspark.sql.dataframe.DataFrame

Table

	OriginAirportCode	Month	DayofMonth	CRSDepHour	DayOfWeek	Carrier	DestAirportCode
1	SEA	4	19	21	5	WN	SMF
2	EWR	4	30	8	2	EV	MEM
3	CLT	4	28	16	7	YV	STL
4	BOS	4	7	6	7	DL	JFK
5	PVD	4	17	7	3	WN	MCO
6	MKE	4	22	17	1	DL	MSP
7	DAL	4	24	7	3	WN	MCI
8	FLL	4	15	9	1	VX	SFO
9	PHX	4	29	16	1	WN	SAN
10	MIA	4	14	14	7	AA	DCA
11	SMF	4	10	18	3	OO	SLC
12	CMH	4	30	11	2	9E	MSP
13	MSP	4	9	9	2	DL	SMF
14	ORD	4	25	8	4	UA	RSW
15							

3,809+ rows | Truncated data

Run the previous cell. You should see a table displayed with the scored data. Scroll all the way to the side. There you will find the prediction column containing the flight delay prediction provided by your machine learning model.

In the following cell, you will create a table that summarizes the flight delays data. Instead of containing one row per flight, this new summary table will contain one row per origin airport at a given hour, along with a count of the quantity of anticipated delays. We also join the **airport_code_location_lookup_clean** table you created at the beginning of the lab, so we can extract the airport coordinates.

```
%sql
SELECT  OriginAirportCode, Month, DayofMonth, CRSDepHour, Sum(prediction) NumDelays,
    CONCAT(Latitude, ',', Longitude) OriginLatLong
    FROM scoredflights s
    INNER JOIN db2025_airport_code_location_lookup_clean a
    ON s.OriginAirportCode = a.Airport
    WHERE Month = 4
    GROUP BY OriginAirportCode, OriginLatLong, Month, DayofMonth, CRSDepHour
    Having Sum(prediction) > 1
    ORDER BY NumDelays DESC
```

_sqldf: pyspark.sql.dataframe.DataFrame = [OriginAirportCode: string, Month: integer ... 4 more fields]

Table

	OriginAirportCode	Month	DayofMonth	CRSDepHour	NumDelays	OriginLatLong
1	ATL	4	28	16	120	33.63666667,-84.427777...
2	MSP	4	12	19	88	44.88194444,-93.221666...
3	ATL	4	14	16	72	33.63666667,-84.427777...
4	MSP	4	11	19	72	44.88194444,-93.221666...
5	MSP	4	21	19	72	44.88194444,-93.221666...

```
summary = spark.sql("SELECT  OriginAirportCode, Month, DayofMonth, CRSDepHour, Sum(prediction) NumDelays,     CONCAT(Latitude, ',',
Longitude) OriginLatLong FROM scoredflights s INNER JOIN db2025_airport_code_location_lookup_clean a ON s.OriginAirportCode = a.
Airport WHERE Month = 4 GROUP BY OriginAirportCode, OriginLatLong, Month, DayofMonth, CRSDepHour  Having Sum(prediction) > 1 ORDER BY
NumDelays DESC")
```

summary: pyspark.sql.dataframe.DataFrame = [OriginAirportCode: string, Month: integer ... 4 more fields]

```
summary.write.mode("overwrite").saveAsTable("flight_delays_summary")
```

Execute the following to verify the table has data

```
%sql
select * from flight_delays_summary
```

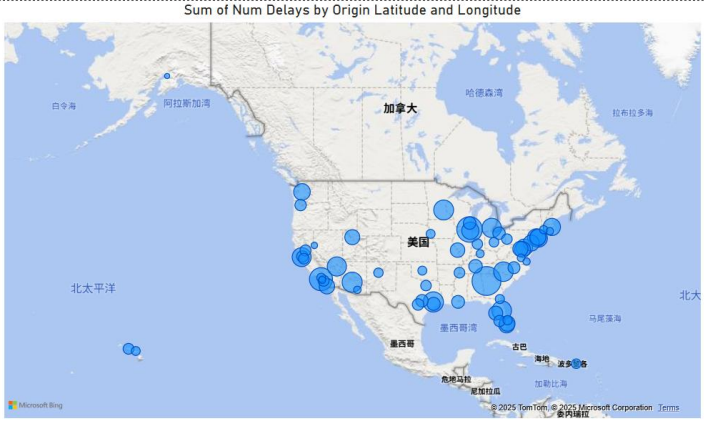
_sqldf: pyspark.sql.dataframe.DataFrame = [OriginAirportCode: string, Month: integer ... 4 more fields]

Table

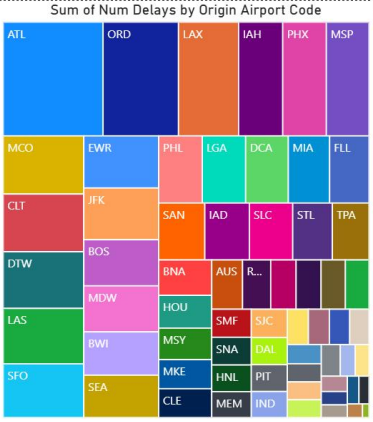
	OriginAirportCode	Month	DayofMonth	CRSDepHour	NumDelays	OriginLatLong
1	ATL	4	28	16	120	33.63666667,-84.427777...
2	MSP	4	12	19	88	44.88194444,-93.221666...
3	ATL	4	14	16	72	33.63666667,-84.427777...
4	MSP	4	11	19	72	44.88194444,-93.221666...
5	MSP	4	21	19	72	44.88194444,-93.221666...

G Visualization of Flight Delay Data

Sum of Num Delays by Origin Latitude and Longitude



Sum of Num Delays by Origin Airport Code



Sum of Num Delays by Day of Month