EBA5006: Graduate Certificate in Big Data Analytics

Real-time Bus Transit Analysis Based on Meteorology & Traffic

Zeng Fanyi

zfygni@hotmail.com

Contents

1. Introduction & Background

In urban life, public transportation serves as a lifeline for citizens, ensuring smooth and efficient mobility. However, the punctuality and reliability of bus services often face challenges due to the unpredictable nature of weather, varying passenger loads and stuff. These factors are critical factors influencing the overall efficacy of public transit systems. So, we aim to provide a more accurate method for predicting travel time between bus stations.

Our stakeholders are LTA (Land Transport Authority) and passengers. With this project, we hope we can Predict station-to-station intervals using passenger flow, rainfall, distance, and date. Then for LTA, we hope we can help them to optimize bus schedules and offer higher-quality services. Also, we can help them to monitor the operation of bus systems and provide data support for decision making. For passengers, we want to provide them with a platform to input departure and destination stops to estimate bus travel time so that we can improve their travel experience.

2. Data Source

In this project, we use two real-time datasets and three local datasets. Except the rainfall dataset which is got from 'Data.Gov.SG', the others are all got from 'DataMall'.

2.1 Dictionary for real-time dataset

2.1.1 Traffic speed bands

Description: Current traffic speeds on expressways and arterial roads, expressed in speed bands.

Attributes	Description	Sample
LinkID	Unique ID for this stretch of road	103046953
SpeedBand	Speed Bands Information. Total: 8 1 – indicates speed range from 0 < 9 ... 8 – speed range from 70 or more	2
StartLon	Longitude map coordinates for start point for this stretch of road.	103.86246461405193
StartLat	Latitude map coordinates for start point for this stretch of road.	1.3220591510051254
EndLon	Longitude map coordinates for end point for this stretch of road.	103.86315591911669
EndLat	Latitude map coordinates for end point for this stretch of road.	1.3215993547809128

2.1.2 Rainfall

Description: Precipitation readings at weather-station level, updated every five minutes.

Attributes	Description	Sample
id	Id of the monitoring area	s77
location	Location of the monitoring area	"latitude": 1.2937, "longitude": 103.8125
timestamp	timestamp	2024-01-27T14:35:00+08:00
value	0 - no rainfall ： > 0 - rainy	0

2.2 Dictionary for local dataset

2.2.1 Bus Routes

Description: Detailed route information for all services currently in operation.

Attributes	Description	Sample
ServiceNo	The bus service number	10
Direction	Distance travelled by bus from starting location to this bus stop (in kilometres)	1
StopSequence	The i-th bus stop for this route	28
BusStopCode	The unique 5-digit identifier for this physical bus stop	01219
Distance	Distance travelled by bus from starting location to this bus stop (in kilometres)	10.3

2.2.2 Bus Stop

Description: Detailed information for all bus stops currently being serviced by buses.

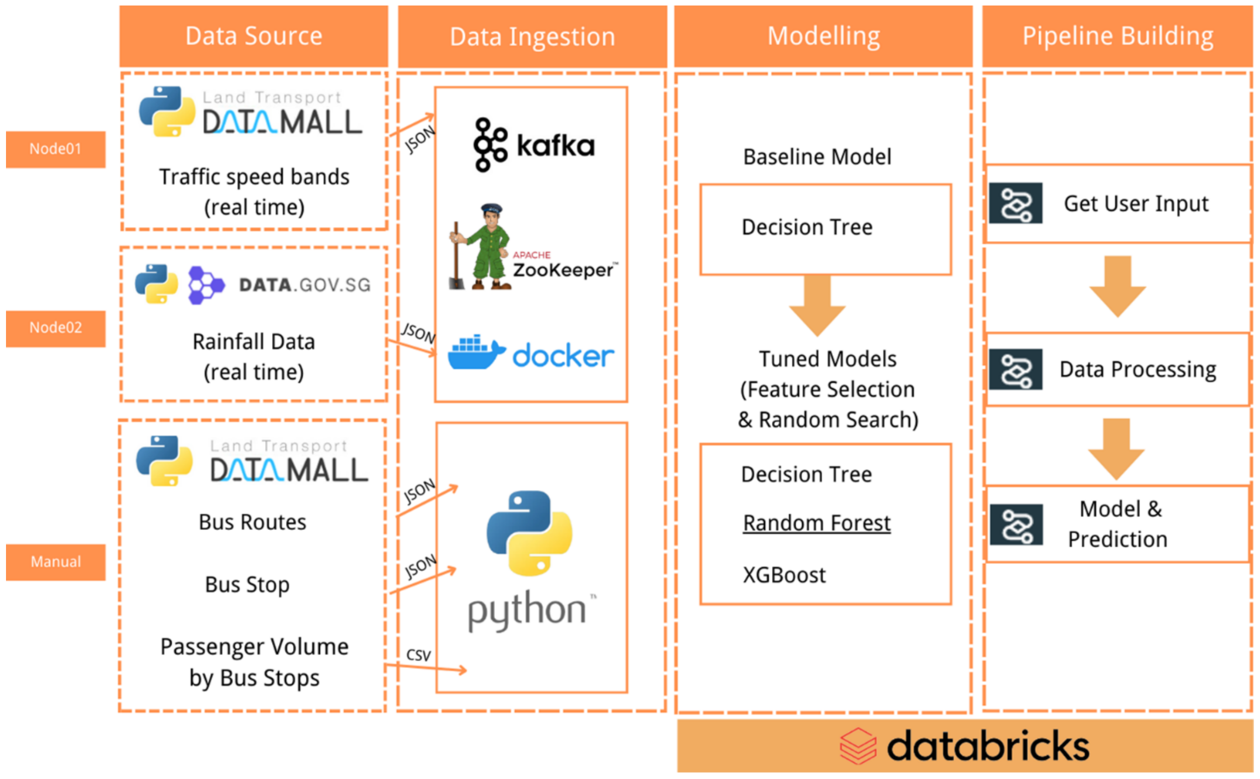
Attributes	Description	Sample
BusStopCode	The unique 5-digit identifier for this physical bus stop	01219
Latitude	Location coordinates for this bus stop	1,29
Longitude	Location coordinates for this bus stop	103.85

2.2.3 Passenger Volume by Bus Stops

Description: Tap in and tap out passenger volume by weekdays and weekends for individual bus stops.

Attributes	Description	Sample
DAY_TYPE PT_CODE	The unique 5-digit identifier for this physical bus stop	01219
DAY_TYPE	Two values: WEEKDAY; WEEKENDS/HOLIDAY	WEEKDAY
TIME_PER_HOUR	24h in a day from 0 to 23	20
TOTAL_TAP_IN_VOLUME	Passenger volume tap in bus	853
TOTAL_TAP_OUT_VOLUME	Passenger volume tap out bus	834

3. Pipeline Structure



3-1 Pipeline Structure

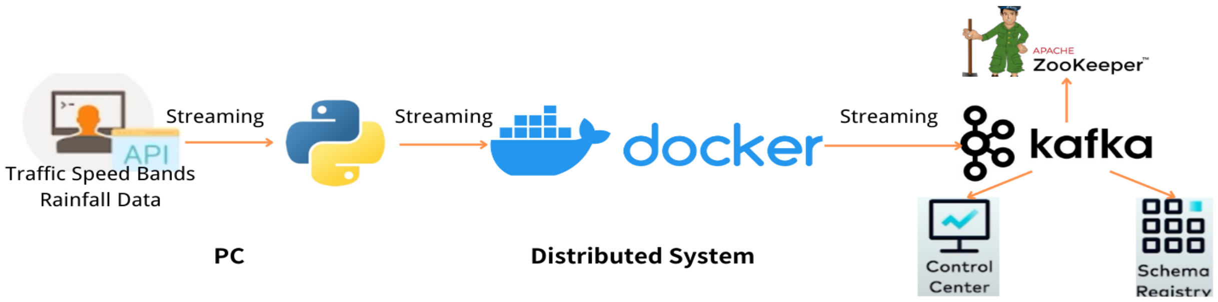
4. Data Ingestion

4.1 Real-time Data Processing Workflow

This section will initially introduce the structure of the Real-time Data Processing workflow. This provides a foundation for the subsequent stages of real-time data extraction.

4.1.1 API Connector

Our initial strategy involved the use of an API Connector for reading real-time data. This connector was configured to interact with Docker, facilitating data communication from a personal computer to Docker. Docker, a platform for building distributed systems, offered the capability to process streaming data real-time, enhancing the efficiency and scalability of our deployments.

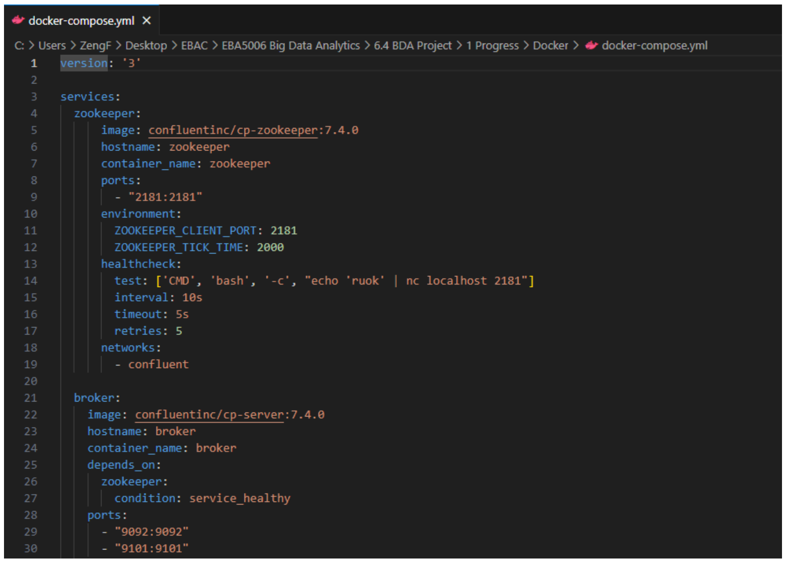


4-1 Real-time Data Processing Workflow

4.1.2 Docker Container Building and Setup

The next step was Docker Container Building, which encompassed four components: ZooKeeper, Schema-registry, Kafka Broker, and Control-center.

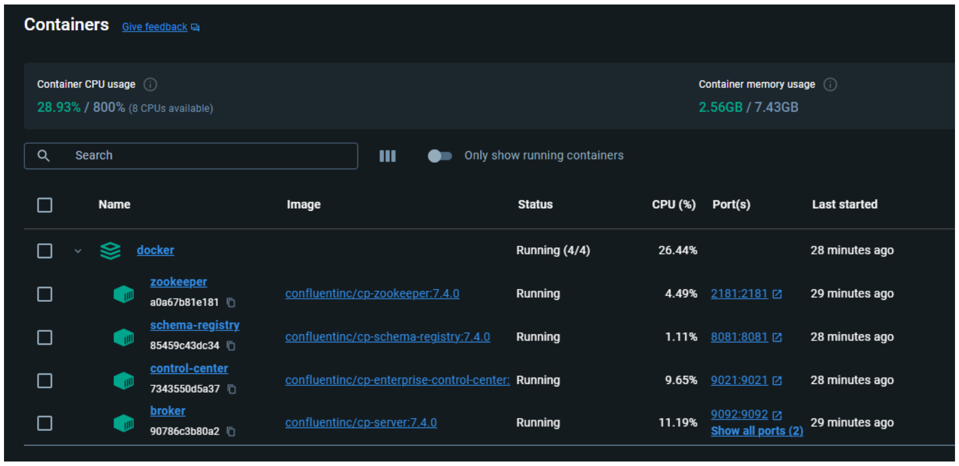
At this stage we need to use the 'docker-compose' to configure the specific components as well as interfaces of the Docker Container and complete the deployment of each container of Docker in the host machine.


docker-compose.yml ×
C: > Users > ZengF > Desktop > EBAC > EBA5006 Big Data Analytics > 6.4 BDA Project > 1 Progress > Docker > docker-compose.yml

```yaml
version: '3'

services:
  zookeeper:
      image: confluentinc/cp-zookeeper:7.4.0
      hostname: zookeeper
      container_name: zookeeper
      ports:
        - "2181:2181"
      environment:
        ZOOKEEPER_CLIENT_PORT: 2181
        ZOOKEEPER_TICK_TIME: 2000
      healthcheck:
        test: ['CMD', 'bash', '-c', "echo 'ruok' | nc localhost 2181"]
        interval: 10s
        timeout: 5s
        retries: 5
      networks:
        - confluent

  broker:
    image: confluentinc/cp-server:7.4.0
    hostname: broker
    container_name: broker
    depends_on:
      zookeeper:
        condition: service_healthy
    ports:
      - "9092:9092"
      - "9101:9101"
```

4-2 docker-compose (partial display)

These components collaborated to construct a robust environment for data processing. More specifically, Kafka's management and coordination were handled by ZooKeeper. The Schema-registry was responsible for storing Kafka's data schemas to simplify data evolution management. Kafka Broker manages the storage of real-time data, storing it in Kafka topics for subsequent consumption and processing by consumers. Control-center provided a user-friendly interface for Kafka ecosystem management.



Containers Give feedback

Container CPU usage 28.93% / 800% (8 CPUs available)

Container memory usage 2.56GB / 7.43GB

Search

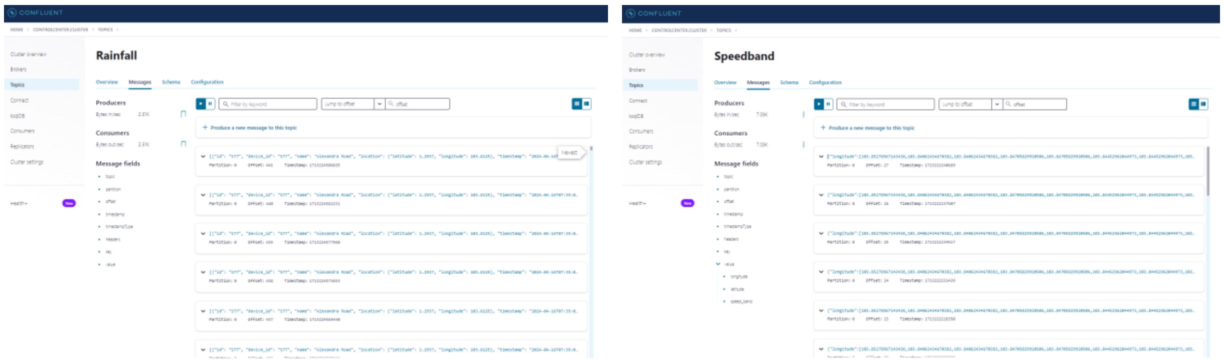
Only show running containers

Name	Image	Status	CPU (%)	Port(s)	Last started
docker		Running (4/4)	26.44%		28 minutes ago
zookeeper a0a67b81e181	confluentinc/cp-zookeeper:7.4.0	Running	4.49%	2181:2181	29 minutes ago
schema-registry 85459c43dc34	confluentinc/cp-schema-registry:7.4.0	Running	1.11%	8081:8081	28 minutes ago
control-center 7343550d5a37	confluentinc/cp-enterprise-control-center:	Running	9.65%	9021:9021	28 minutes ago
broker 90786c3b80a2	confluentinc/cp-server:7.4.0	Running	11.19%	9092:9092 Show all ports (2)	29 minutes ago

4-3 Docker Container

4.1.3 Kafka Streaming

The final step in this section was Kafka Streaming. A Kafka Producer was created in Python, which facilitated the creation of Kafka topics and downloading records in the Control-center. On the host machine, we install and continuously run the Kafka producer, fetching real-time rainfall and speed band data, and feeding them into Kafka topics. The records were then downloaded and prepared for local model training.



4-4 Download real-time data in Kafka Control-center

4.2 Data Extraction

During the extraction phase, we managed two types of data: real-time data and local data. This portion of data processing primarily focuses on processing data for local model training.

The real-time data consists of rainfall measurements and speed bands. We obtained rainfall data from 'Data.gov.sg' using an API processor to fetch hourly data from March 30th to April 5th. The speed bands data, on the other hand, was sourced from the 'Data Mall'. Since the data API interface for this type of information does not provide access to historical data, we could only retrieve the data once per hour. We achieved this by implementing the Kafka workflow mentioned earlier, which allowed us to download the hourly speed band data from March 30th to April 5th via the Kafka Control Centre.

As for the local data, we employed an API processor to gather information on passenger volume, bus stops, and bus routes from the 'Data Mall', covering the period from March 30th to April 5th. To streamline the analysis process, we limited our focus to Route 10, which operates from 6 a.m. to 12 a.m.

4.3 Data Transformation

4.3.1 First Stage – Raw Data Processing

i. bus_info

The bus_info table is a dataset containing information about bus route 10, used for training the model. It includes ten fields: Direction, StopSequence, BusStopCode_X, Latitude_X, Longitude_X, Distance, BusStopCode_Y, Latitude_Y, Longitude_Y, and Interval_Distance. bus_info is generated by processing and merging two tables, Bus Routes and Bus Stop, with Bus Routes serving as the primary table and using its BusStopCode as the primary key.

Due to the API's limitation of retrieving a maximum of 500 records per request, we extracted the necessary information from the *Bus Stop* API in batches and then consolidated it into one table.

Based on the *StopSequence* in the Bus Route, we determined adjacent stops and added a column *BusStopCode_Y* representing the next adjacent stop. Subsequently, we calculated the distance between these two adjacent stops using their information as *Interval_Distance*.

We then added longitude and latitude for each stop based on the *BusStopCode_X* in the Bus Stop table, resulting in the final *bus_info* dataset.

ii. speed_band_link

The speed_band_link table is a dataset containing the link IDs, latitude, and longitude information for all speed band measurement points, which was the common information of all the extracted traffic speed band data named “Traffic_2024MMDDHH00”. This information will be utilized in the subsequent process of matching speed band data with bus stop information.

4.3.2 Second Stage - Master Table

The master table named *merged_data* is a dataset containing all the information we needed to build the predictive model. We created a new table named *data* which includes various information for further analysis. First, we added a column named *date_hour* with values ranging from "2024-03-30 06" to "2024-04-06 00", resulting in 19 (19 operating hours a day) * 7 (7 days) * 146 (146 station-to-station intervals) = 19,418 rows. Subsequently, we derived the *day_type* column based on the *date_hour* column to distinguish between "WEEKDAY" and "WEEKENDS/HOLIDAY", and we extracted the *time_per_hour* (hour number) from the *date_hour* for subsequent matching.

Next, we extracted relevant information from the *bus_info* table. We added a column named *code_x*, where the 146 *BusStopCode_X* values from the *bus_info* table were sequentially appended to the *data* table, repeated 19*7 times. We also added *latitude_x* and *longitude_x* columns by matching the *code_x* values with the corresponding latitude and longitude values from the *bus_info* table. Additionally, we included a *distance* column by matching the *code_x* values with the respective distances from the *bus_info* table.

We then extracted information from the *transport_node_bus_202403* table which represents the average passenger volume of each bus stop on each day type and hour number, adding a *total_volume* column based on matching the *day_type, time_per_hour,* and *code_x.*

Following that, we extracted relevant information from the *rainfall_station* and *rainfall_hourly_value* tables. We found the nearest observation point of each rainfall station for each bus stop based on latitude and longitude information and matched the corresponding station ID to the nearest bus stop in the *data* table. We also added a *value* column which means rainfall value of each start bus stop by matching the station ID and *date_hour* in the *rainfall_hourly_value* table.

Subsequently, we extracted information from the *speed_band* table. Before proceeding with the matching process, we pre-processed the *speed_band* table by calculating the latitude and longitude of the centre point for each speed band link. This involved determining the average latitude and longitude of the starting and ending points of each link. Additionally, we converted the speed bands to speeds in kilometres per hour (km/h) by taking the average of the maximum and minimum speeds for each band. Finally, we saved a copy of the processed dataset named *speed_band_clean_original* and adjusted the hours in the *date_hour* column by subtracting 1 since we used the speed information in the next hour for modelling and saved it as another dataset named *speed_band_clean*. After preprocessing, we found the nearest observation point of each link ID for each bus stop based on latitude and longitude information, matched the corresponding link ID to the nearest bus stop in the *data* table, and added a *speed* column to *data* by matching the link ID and *date_hour*.

Finally, we added an *interval* column calculated as the ratio of distance to speed for the next hour, and encoded *day_type* and *time_per_hour*. Then we checked and deleted any missing values in the dataset, resulting in the final *merged_data* table.

4.4 Load

After extracting and transforming all the data, we can load it into the system in preparation for local model training.

The first step in this process involves loading all the necessary data, for which we used the Pandas library to import all required data into Python. Subsequently, we integrated these datasets. In the context of data analysis, data integration often involves combining data from different sources into a single, unified table. Accordingly, we merged all loaded data into a single table named "merged_data".

Following the data loading and integration, we proceeded with the division of the dataset, a crucial step in preparing data for machine learning applications. The dataset was split into two distinct subsets: 70% of the data was designated for training purposes, while the remaining 30% was allocated for testing.

5. Modelling

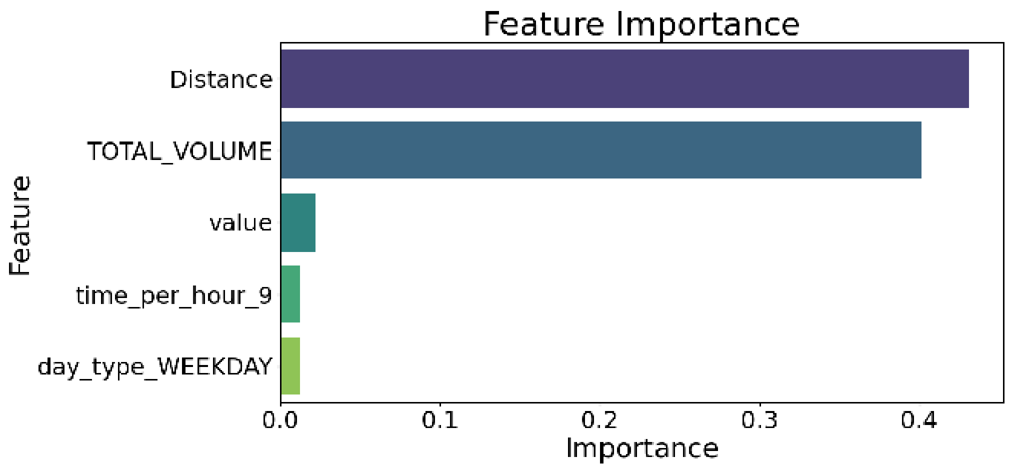
Our goal is to build a model to predict the station-to-station interval for the next hour based on the current hour, current date type, passenger flow, precipitation amount and distance between adjacent bus stops.

Since the operating hours of Route 10 are from 6 am to 12 am every day, we utilized historical real-time data including speed band and rainfall value from 6 am on March 30 to 12am on April 6 and local data extracted in March to build our predictive model. Due to a moderate number of observations which is 19,418 rows with a limited set of features, we chose tree-based models for our prediction.

We began by establishing a Baseline Model using a decision tree without feature selection and any parameter settings. However, we can see that the initial baseline model showed mediocre performance. Besides, there is a notable performance gap between the train and test sets, which indicates potential overfitting.

Tree Model Comparison		Baseline Model
		Decision Tree
R-squared	Train set	0.854
	Test set	0.558
MSE	Train set	0.062
	Test set	0.127

To address this, we explored three different tree-based models: decision tree, random forest, and XGBoost. The feature importance analysis revealed that distance and total volume were significant predictors of bus intervals for the next hour.



5-1 Feature Importance

As our objective also involved investigating the impact of precipitation on intervals, we also included rainfall value as selected features in our study. Therefore, we used distance between adjacent bus stops, passenger volume and rainfall value for predicting the station-to-station interval for the next hour.

To realize the adaptability of the model, allowing its parameters to dynamically adjust according to input variations, we employed grid search for preliminary parameter tuning to determine appropriate parameter ranges and subsequently utilized random search with MSE minimization as the criterion to identify the optimal hyperparameters for each model, and then exported the best-performed model to a joblib file after comparing the results.

This table shows the performance of all the models we built. Upon evaluation, random forests outperformed the other models on the test set, with a minimized mean squared error and a maximized R-squared value.

Tree Model Comparison		Baseline Model	Tuned Model		
		Decision Tree	Decision Tree	Random Forest	XGBoost
R-squared	Train set	0.854	0.787	0.750	0.551
	Test set	0.558	0.572	0.614	0.541
MSE	Train set	0.065	0.191	0.017	0.193
	Test set	0.127	0.123	0.111	0.132

Here are the best parameters for Random Forest: {'max_depth': 21, 'min_samples_leaf': 2, 'min_samples_split': 2, 'n_estimators': 67}

Given our dataset's characteristics - 19,418 rows with 3 independent variables and 1 dependent variable, random forest is an ideal choice for our dataset since it is inherently robust to noise and outliers.

Therefore, we selected random forests with the optimal parameters as follow as our predictive model and exported it as a joblib file for subsequent use in our workflow.

6. Workflow Building

6.1 Platform

After comparing several data pipeline construction environments, including AWS Data Pipeline, Google Cloud Dataflow, and Apache Airflow on Docker, we ultimately chose to use Azure Databricks primarily for the following reasons.

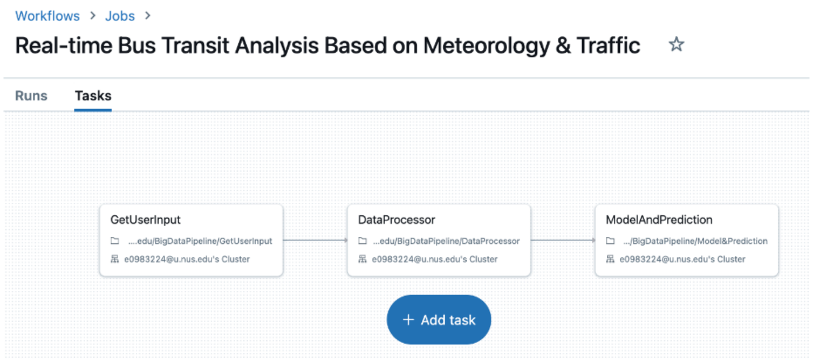
i. Azure Databricks offers a unified and collaborative environment for complex data engineering projects. This environment can integrate tasks across data engineering, data analytics, data management, and machine learning to construct a comprehensive and functional data pipeline. It also supports multiple development languages like Python, Scala, SQL, and R for the notebooks within its workspace. We can easily share files and compute resources in the developing process. These provide a perfect working environment for our team project.

ii. Azure Databricks has demonstrated superior cloud service capabilities, allowing us to complete our project within our budget constraint. It offers a $100 credit for student accounts, followed by a 14-day free tier provided by Databricks. This has given us ample time to learn and understand the tool, enabling us to complete our project on a very limited budget.

iii. Azure Databricks allows us to manage our workflows effectively. It features an exceptionally user-friendly interface for workflow development and monitoring. During the workflow development phase, we can smoothly create tasks, inspect and fine-tune their detailed parameters, sequences, and dependencies. In the application stage, it allows us to view the outcomes of each task and swiftly pinpoint any issues.

iv. Also, we considered past user experiences. A member of our team has previous experience with Databricks, which we believe can streamline our learning process and expedite our project completion due to the existing knowledge and familiarity.

6.2 Overview


Workflows > Jobs >
Real-time Bus Transit Analysis Based on Meteorology & Traffic
Runs
Tasks
GetUserInput
....edu/BigDataPipeline/GetUserInput
e0983224@u.nus.edu's Cluster
DataProcessor
...edu/BigDataPipeline/DataProcessor
e0983224@u.nus.edu's Cluster
ModelAndPrediction
.../BigDataPipeline/Model&Prediction
e0983224@u.nus.edu's Cluster
+ Add task

6-1 Workflow Construction

Upon selecting our tool, in the workflow design phase, we broke down the workflow into three sequential tasks to align with our project's objectives. The first task is configured to record the start time of the query and to collect user input. The second task engages in data preprocessing to derive the independent variables for our model, which encompasses the integration of real-time rainfall data with other static datasets. The final task is responsible for deploying the trained model to execute predictions and synthesize the outcomes. Critical parameters are transmitted between contiguous tasks as key-value pairs.

6.3 Workflow

Our workflow consists of three tasks, with a linear relationship between them. This means that the next task will only start running after the previous one has successfully completed, and the prediction results will only be generated after all three tasks have successfully run in sequence. Otherwise, an error will occur.

In the design of the entire workflow, there are two key elements: the *dbutils.jobs.taskValues.set()* and *dbutils.jobs.taskValues.get()* functions. These functions serve as bridges between each task in the workflow, helping us transfer variables from one task to the next. The advantage of this approach is that, compared to the traditional method of saving variables or data to files and then retrieving them from files, it reduces memory consumption and improves program efficiency.

6.3.1 Get User Input

User's inputs are designed to be stored in a txt file. Users need to input their trip's start bus stop code and end bus stop code at the corresponding location in a txt file. Then their inputs can be recognized and read into the variables of s*tart_bus_stop_code* and end_*bus_stop_code.*

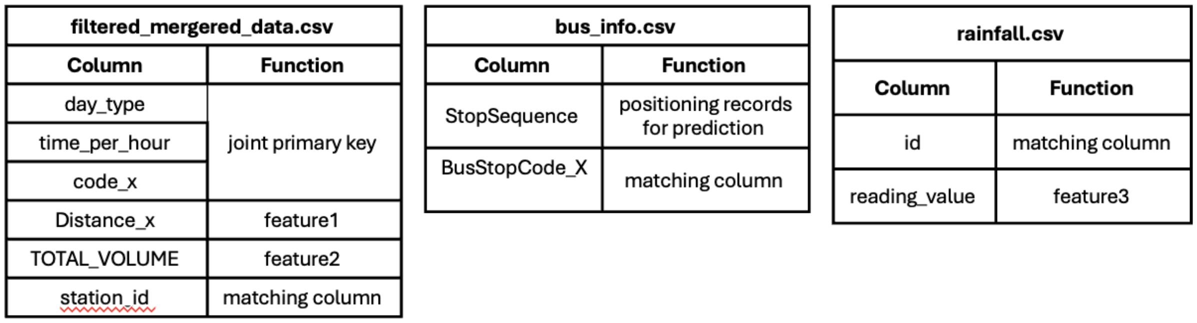
To match the data from Passenger Volume by Bus Stops dataset, we used the *datetime.now()* function to retrieve the current time in the Singapore time zone and assign this time to the variable *current_time*. Then based on the *current_time*, we calculated what day of the week it is today and used an if statement to determine whether the current time is a weekday or a weekend/holiday.

To transmit the calculated value to the next task, we used the built-in function *dbutils.jobs.taskValues.set()* in databricks. Since this function only supports passing data in key-value pair format, we converted the four variables, *start_bus_stop_code*, end_*bus_stop_code*, *current_time*, and *weekday*, into key-value pair format separately.

```
dbutils.jobs.taskValues.set(key='start_code', value=start_bus_code)
dbutils.jobs.taskValues.set(key='end_code', value=end_bus_code)
dbutils.jobs.taskValues.set(key='current_time', value=current_time)
dbutils.jobs.taskValues.set(key='weekday', value=weekday)
```

6.3.2 Data Processor

i. Data Preparation



filtered_mergered_data.csv	
Column	**Function**
day_type	joint primary key
time_per_hour	
code_x	
Distance_x	feature1
TOTAL_VOLUME	feature2
station_id	matching column

bus_info.csv	
Column	**Function**
StopSequence	positioning records for prediction
BusStopCode_X	matching column

rainfall.csv	
Column	**Function**
id	matching column
reading_value	feature3

6-2 Three Original Tables

In this task, we merged the above three tables to get one large table including all the information we need. Each of these three tables select and retain the necessary columns from *merged_data*, *bus_info* and *rainfall* respectively. Since we want to predict the time interval between each station and the next station then add these intervals together to get the result, we use *weekday*, *hour*, *start_code* or *end_code* of the query as the joint primary key, locate and determine the indexes of all records to be predicted, then extract all these records and its three independent variables. Each independent variable is passed to the third task (ModelAndPrediction) as a list which includes multiple values.

ii. Detail

We used the built-in function *dbutils.jobs.taskValues.get()* in databricks to get the task values (*start_bus_stop_code*, end*_bus_stop_code*, *current_time*, and *weekday*) we have defined from the first job task.

```
start_bus_code = dbutils.jobs.taskValues.get(taskKey = "GetUserInput", key = "start_code")
end_bus_code = dbutils.jobs.taskValues.get(taskKey = "GetUserInput", key = "end_code")
current_time = dbutils.jobs.taskValues.get(taskKey = "GetUserInput", key = "current_time")
weekday = dbutils.jobs.taskValues.get(taskKey = "GetUserInput", key = "weekday")
```

To get the real-time rainfall data, we used API to access the data from DATA.GOV.SG. Since the raw data is in the json type, we defined a function *format_data()* to transform the data into dataframe. Then we combine the rainfall data with the *filtered_merged_data* to organize all the fields we need for prediction. After that, we created three variables, *distance_ls, total_volume_ls* and *rainfall_value_ls*, to be used as predictive model inputs.

To pass variables to the third task, we used *dbutils.jobs.taskValues.set().*

6.3.3 Model & Prediction

We used function dbutils.jobs.taskValues.get() to get the task values (s *distance_ls, total_volume_ls* and *rainfall_value_ls*) we have defined from the second job task.

To introduce the pre-trained prediction model, we load a joblib file that contains the saved model and pass the parameters to the model to obtain predictive results.

Due to the possibility of multiple bus stops between the user-input *start_bus_stop_code* and *end_bus_stop_code*, and considering that the predictive model forecasts the interval only for intervals between adjacent stops, we segment the entire journey into multiple intervals between adjacent pairs of stops when processing the user's input. Therefore, to obtain the interval for the entire journey, we sum the intervals for each segment of the journey, resulting in the predicted interval for the entire journey.

7. Conclusion

We used different tools such as databricks and Kafka to organize and transform the data from various sources. Then we built and trained the prediction model, and finally created a workflow to deploy the prediction model.

7.1 Success Factors

7.1.1 Model's Comprehensiveness

Since rainfall and traffic volume greatly influence the efficiency of bus travel, the real time rainfall data and the patterns of traffic volume are considered in our prediction model to facilitate user trip planning.

7.1.2 Real-time Factors

The real-time rainfall data is updated every 5 minutes through API, so the workflow can ensure the timeliness of the predictive model results.

7.1.3 Integrity of the workflow

In the workflow, we integrate user input retrieval, data processing, and the predictive model into a single pipeline, which can enhance the efficiency of model deployment and the convenience of workflow maintenance.

7.2 Limitation

7.2.1 Time and Budget constraints

Since we used the Azure cloud service platform and our account version is a trial version, there are limitations on many computing resources. To control costs, the predictive model is only applicable for Bus Route 10. And we didn’t involve model training stage into the workflow, so the model parameter cannot be updated automatically.

7.2.2 Speed Band Data Retrieval Constraint

we cannot directly access the historical data. So, we assume that the traffic volume pattern remains similar on a weekly basis, and we fulfill the data based on a weekly cycle to train the model.

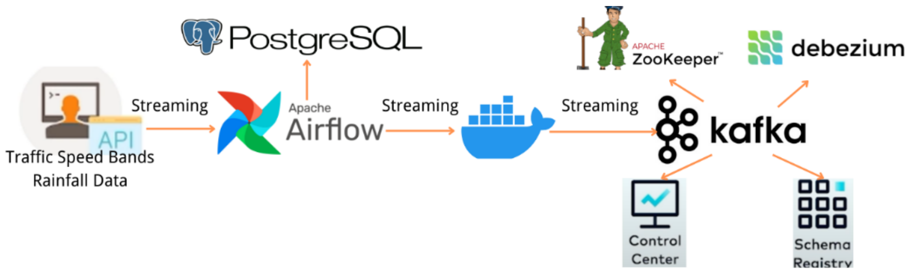
8. Additional Efforts

8.1 Attempts at Other Techniques for Real-time Data Processing

Despite the accomplishments we achieved with the methods described above, we explored other techniques for real-time data processing. We attempted to build a pipeline in both local and cloud environments but encountered challenges.

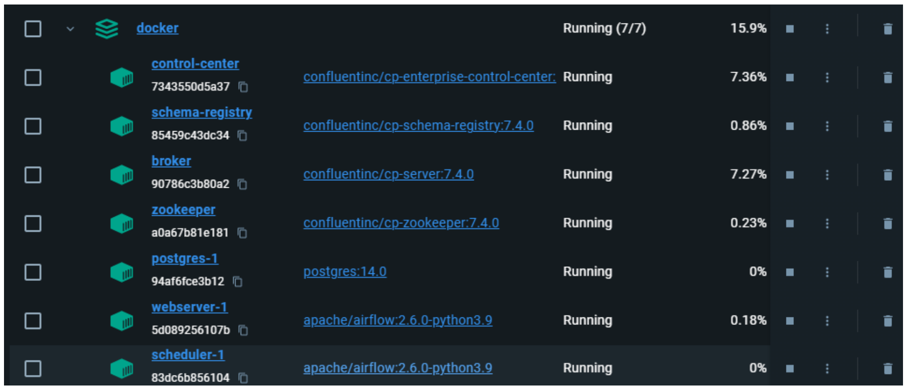
8.1.1 Local Environment

For the local environment, we attempted to create an automated ETL pipeline for real-time data using Airflow and added Debezium in Docker to support Kafka's connection with a Postgres database.



8-1 Kafka Workflow in Local Environment

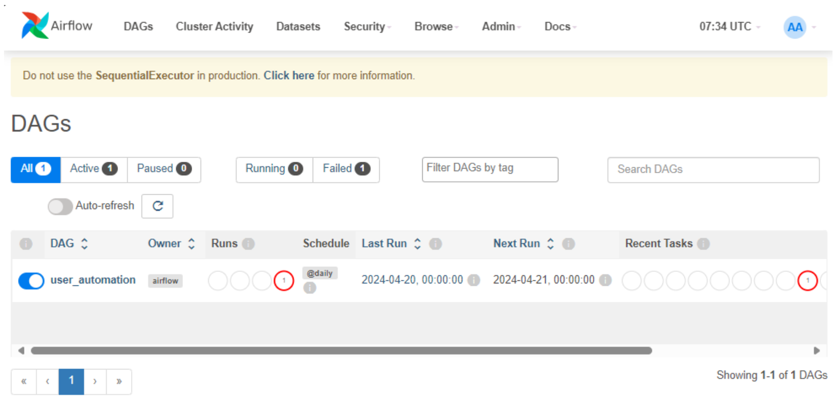
In addition to the previously mentioned containers, to manage the Airflow and Database component to Docker, we also need to refine 'docker-compose'.



docker		Running (7/7)	15.9%
control-center 7343550d5a37	confluentinc/cp-enterprise-control-center:	Running	7.36%
schema-registry 85459c43dc34	confluentinc/cp-schema-registry:7.4.0	Running	0.86%
broker 90786c3b80a2	confluentinc/cp-server:7.4.0	Running	7.27%
zookeeper a0a67b81e181	confluentinc/cp-zookeeper:7.4.0	Running	0.23%
postgres-1 94af6fce3b12	postgres:14.0	Running	0%
webserver-1 5d089256107b	apache/airflow:2.6.0-python3.9	Running	0.18%
scheduler-1 83dc6b856104	apache/airflow:2.6.0-python3.9	Running	0%

8-2 Docker Container with Airflow, Debezium and Postgres Database

We first build the Webserver and Scheduler for the Airflow. The Webserver is a user interface for viewing and managing data pipelines. Within this interface, users can view the status of current and historical tasks, set schedules for data pipelines, and manually trigger tasks. The Scheduler is a task scheduler, whose main responsibility is to execute data pipelines according to preset plans.



Do not use the SequentialExecutor in production. Click here for more information.

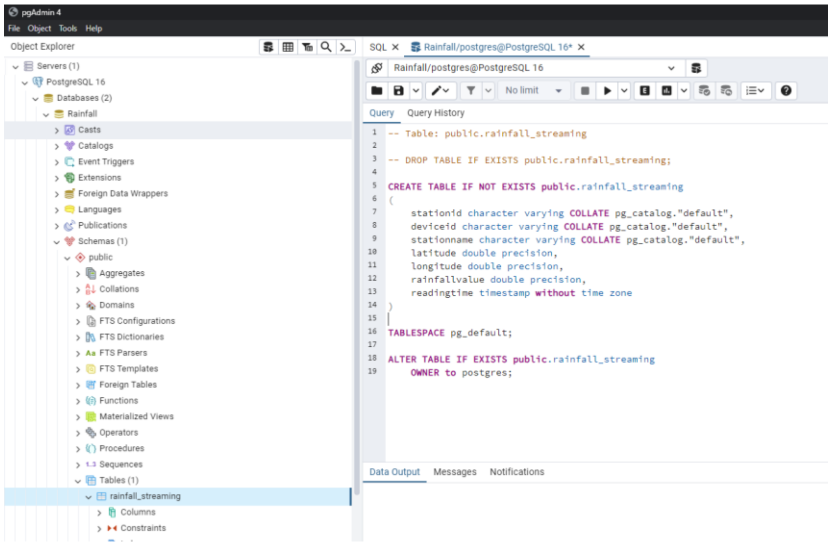
DAGs

DAG	Owner	Runs	Schedule	Last Run	Next Run	Recent Tasks
user_automation	airflow	1	@daily	2024-04-20, 00:00:00	2024-04-21, 00:00:00	1

Showing 1-1 of 1 DAGs

8-3 Airflow interface

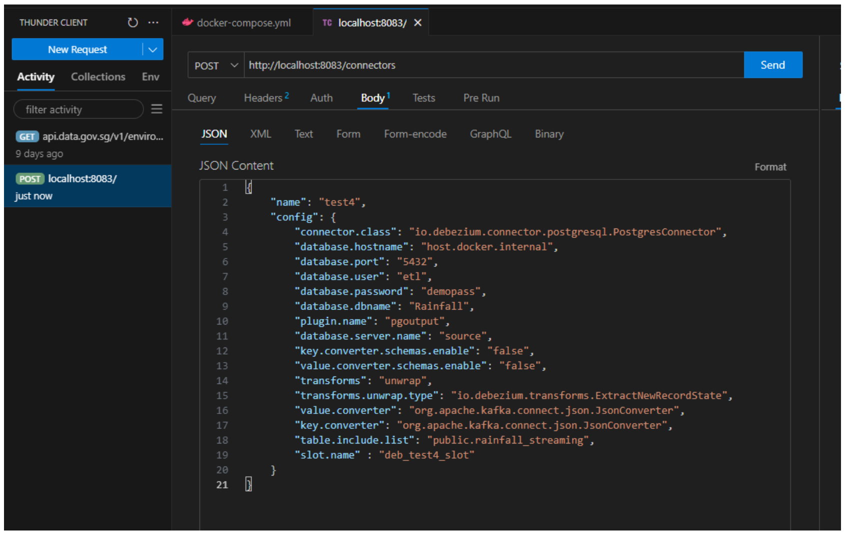
Second, on the database side, we have successfully built Postgres Database both locally and in Docker. In the local Postgres Database, we can use pdAdmin to manage our database and use Postgres SQL statements to add the appropriate tables ready to be written to.



```sql
-- Table: public.rainfall_streaming

-- DROP TABLE IF EXISTS public.rainfall_streaming;

CREATE TABLE IF NOT EXISTS public.rainfall_streaming
(
    stationid character varying COLLATE pg_catalog."default",
    deviceid character varying COLLATE pg_catalog."default",
    stationname character varying COLLATE pg_catalog."default",
    latitude double precision,
    longitude double precision,
    rainfallvalue double precision,
    readingtime timestamp without time zone
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS public.rainfall_streaming
    OWNER to postgres;
```

8-4 Postgres Database in pdAdmin 4

However, we are encountering some issues while connecting Kafka to the database. Initially, we tried to connect to our local Postgres Database using Kafka, and subsequently attempted to connect with a database built in Docker. Unfortunately, we encountered connection problems in both database environments. We were unable to set up the Debezium Connector for Kafka and the database, the main issue being that we could not find the correct host connector between the host machine and the virtual machine (Docker). Consequently, we were unable to write live data to the database for storage. Due to time constraints, we have decided to pause this part of the work.



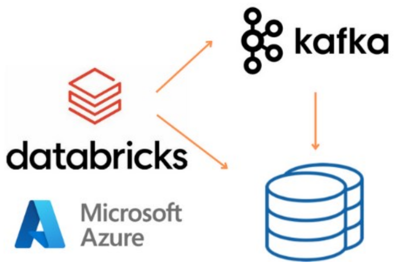
POST http://localhost:8083/connectors

```json
{
    "name": "test4",
    "config": {
        "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
        "database.hostname": "host.docker.internal",
        "database.port": "5432",
        "database.user": "etl",
        "database.password": "demopass",
        "database.dbname": "Rainfall",
        "plugin.name": "pgoutput",
        "database.server.name": "source",
        "key.converter.schemas.enable": "false",
        "value.converter.schemas.enable": "false",
        "transforms": "unwrap",
        "transforms.unwrap.type": "io.debezium.transforms.ExtractNewRecordState",
        "value.converter": "org.apache.kafka.connect.json.JsonConverter",
        "key.converter": "org.apache.kafka.connect.json.JsonConverter",
        "table.include.list": "public.rainfall_streaming",
        "slot.name" : "deb_test4_slot"
    }
}
```

8-5 Debezium Connector

8.1.2 Cloud Environment

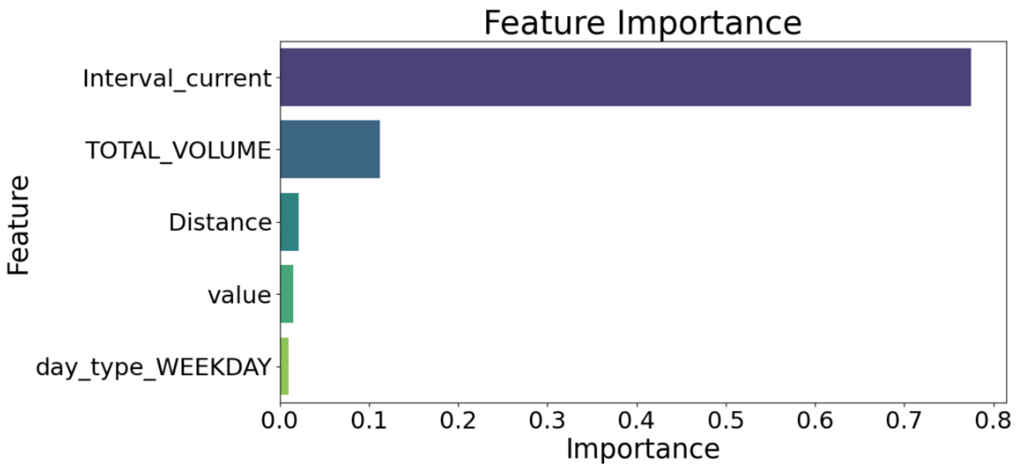
In the cloud environment, specifically Azure Databricks, we encountered difficulties due to insufficient cluster quota and budget constraints. These issues hindered us from creating Kafka and database sources.



8-6 Kafka Workflow in Cloud Environment

8.2 Model Enhancement

Since current bus interval may have some impacts on the bus interval for the next hour, we added another variable speed_current to our merged data table and calculated interval_current as the ratio of distance to current speed to build the predictive model. The feature importance analysis revealed that the current interval, distance and total volume were significant predictors of bus intervals for the next hour, so we chose these 4 features for modelling.



8-7 Feature Importance after adding *interval_current*

This table shows the results of all the updated models we built:

Tree Model Comparison		Baseline Model	Tuned Model		
		Decision Tree	Decision Tree	Random Forest	XGBoost
R-squared	Train set	0.918	0.804	0.794	0.757
	Test set	0.564	0.666	0.712	0.709
MSE	Train set	0.032	0.078	0.082	0.097
	Test set	0.158	0.122	0.104	0.106
MAE	Train set	0.037	0.094	0.101	0.110
	Test set	0.120	0.124	0.111	0.111

Here are the best parameters for updated Random Forest: {'max_depth': 11, 'min_samples_leaf': 4, 'min_samples_split': 2, 'n_estimators': 52}

The test R-squared value and test MSE are lower than the model without adjustment, which indicates better performance than before, so it will be used in our further study.