

ASSIGNMENT 3 - EEET2574
GROUP PROJECT
DATA PIPELINES TO PREDICT
TOMORROW RAIN IN AUSTRALIA

Lecturer: Dr Arthur Tang

Group 4:

Nguyen Bao Tri - s3749560

Nguyen Danh Duc - s3777226

Nguyen Duc Dung - s3803749

Ho Duy Hoang - s3672214

TABLE OF CONTENT

I. OVERVIEW	2
1. Statement of Problem	2
2. Scope	3
3. Deliverables:	3
4. Members and Contribution	4
II. SOLUTION DESIGN	4
1. Solution	4
1.1. Datasets	5
2. Data Pipelines For Kaggle dataset and Machine Learning Model	5
2.1. Machine Learning Model	5
2.2. Data Pipeline Architecture	10
2.3. Pricing	17
3. Data Pipelines For OWM	18
3.1. Handling Open Weather Map Real Time Data	18
3.2. Required libraries/packages:	22
3.3. Data pipeline architecture:	22
3.4. Pricing	25
III. CONCLUSION	26
IV. REFERENCES	26

I. OVERVIEW

1. Statement of Problem

Weather forecast has always been one of the greatest achievements of humankind as the information of upcoming weather conditions greatly enhance and support our daily life. For instance, many of our activities, especially those outdoors, are significantly affected by the rain, and by being aware of a rainy day in advance, we can rearrange our schedule and daily tasks or prepare for the inconvenience upon executing plans, thus, maintain our productivity. In reality, there are a myriad of existent weather forecast systems and they have been developing for years with the support of magnificent facilities, therefore, the problem we are trying to tackle is not so significant. Our objective is purely educational as we aim to efficiently apply the knowledge of big data that we learnt to build a rain prediction system, specifically, build data pipelines that get real time weather data and forecasts the occurrence of rain in the next day.

Big data as it is defined, is a collection of (structured and unstructured) data that is so enormous, diversified, rapidly changing, and complicated that standard technologies or software are incapable of processing it in a given amount of time. In this case the collectible and recorded weather data can be considered as big data. The huge amount of historical weather statistics can be used to train a predictive model, as there is plenty, it helps to reinforce the performance of the model significantly. On the other hand, the streaming raw data that we get in real time is the crucial factor to answer the enunciated question. The data is obtained continually and updated nearly every hour, ensuring its veracity, and the model is developed for accurate comparison, with a limited margin of error, allowing for the prediction of `predict_rain_tomorrow` is conceivable, but requires a substantial quantity of data processing.

2. Scope

This project's objective is to compare the data from the.csv file we gather with the data acquired constantly from OWM's streaming data and provide a comparison and diagnostic of whether or not it will rain tomorrow. Data collected via OWM and Kaggle will be processed by two individuals. All of the process is done automatically through two different data pipelines, one is for the training of the model and one is for handling the real time data before being fed into the model to make future predictions. The

project focuses more on the design of the data pipelines using AWS services and external technologies such as MongoDB to store data.

The project is considered a success when the data pipelines run as required and handle every task in terms of processing the data correctly. To be more precise, the model needs to achieve high accuracy, expected to be more than 0.8 regarding evaluation metrics (R2 score, F1 score, and accuracy score) and successfully deployed on AWS Sagemaker. Data processing jobs, especially the real time streaming data from OWM, have to precisely make the current day data ready to fit the model at the last node with no difficulties.

3. Deliverables:

The final result will have a pipeline to process and handle the historical data then use it to train a machine learning model that predict the rain occurrence the next day, this model is deployed to AWS Sagemaker for usage and a pipeline that transform raw real-time weather data collected hourly from OWM API to weather record of a day with fields that fit the features of the trained model. The weather record for the day is updated hourly as the addition of a record in a certain hour and is used to predict tomorrow's rain.

4. Members and Contribution

- Nguyen Bao Tri, s3749560: design solution, design data pipelines, ETL streaming data.
- Nguyen Danh Duc, s3777226: build streaming data pipeline, data engineer .
- Nguyen Duc Dung, s3803749: train and deploy predicting model.
- Ho Duy Hoang, s3672214: design and build historical data pipeline, data engineer.

II. SOLUTION DESIGN

1. Solution

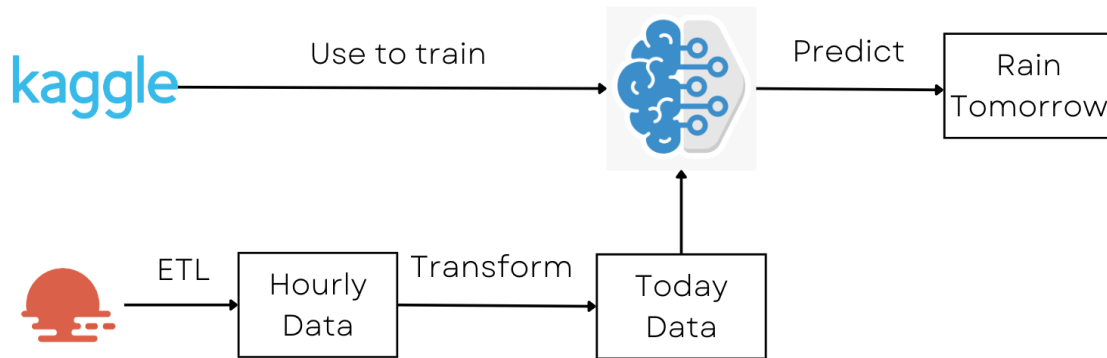


Figure 1. Simple Solution Design

In order to tackle the whole big global problem, our initial goal is to solve the issue in a certain area first. The reasons for this decision are not only applying the “divide and conquer” strategy but also due to the lack of data resources that can enhance the training procedure and provide better results. Moreover, the primary product of this project is not the predicting model. Specifically, this project scope is to introduce a system, a data pipeline that ingest and process data, including real time data to forecast the next day rain occurrence, and the location is Sydney regarding the data sources that we happened to get our hands on.

1.1. Datasets

The datasets that are used are the dataset of daily recorded weather data of various locations in Australia in 10 years, from 2007 to 2017, from Kaggle [1] and the streaming data of weather conditions in Sydney that is collected from Open Weather Map Api.

The Kaggle dataset is historical data that is observed from numerous weather stations across Australia. The data has various fields that are beneficial to train a machine learning model that can predict the rain occurrence tomorrow such as: the minimum and maximum temperature in a day; wind direction, wind speed, humidity, pressure, cloud coverage, temperature in 2 different times of a day, etc.

The Open Weather Map Api provides real time records of weather statistics in certain locations, for this project, it is Sydney. The data is collected hourly and then is processed and aggregated to get data for the current day. Each response from the api includes fields that are similar to the data from the Kaggle dataset, certainly: wind direction, wind speed, humidity, pressure, cloud coverage, temperature, and current weather conditions such as raining, cloudy, etc.

2. Data Pipelines For Kaggle dataset and Machine Learning Model

2.1. Machine Learning Model

This stage is crucial since it plays a significant part in our project; it is the basis for comparing OWM streaming data, therefore it must be processed and trained correctly. We obtain dataset information from Kaggle; nevertheless, the quantity of data is quite vast, therefore we must pick and train the model using appropriate procedures. In this project, we use Databrick as a platform to train and predict models to save budget and then later deploy using Sagemaker. Firstly, we have to import all the necessary libraries for the code to run smoothly.

```
#Import libraries
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import plotly.express as px
from math import sqrt
import warnings
warnings.filterwarnings("ignore")

from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
import sklearn.metrics as metrics
from sklearn.metrics import r2_score
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import classification_report
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score, accuracy_score, confusion_matrix
from sklearn.model_selection import StratifiedKFold
from datetime import datetime
from sklearn.utils import resample
from sklearn.impute import SimpleImputer
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_classif
```

Figure 2. All library for the pipeline

Next step is to load the csv file into Databrick and run. We also have some steps just to apply for the csv file but not the other files' type and then display the table to check if the code runs or not.

```
# File location and type
file_location = "/FileStore/tables/weatherAUS.csv"
file_type = "csv"

# CSV options
infer_schema = "false"
first_row_is_header = "true"
delimiter = ","

# The applied options are for CSV files. For other file types, these will be ignored.
df = spark.read.format(file_type) \
  .option("inferSchema", infer_schema) \
  .option("header", first_row_is_header) \
  .option("sep", delimiter) \
  .load(file_location)

display(df)
```

Table													
	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpeed	WindDir9am	WindDir3pm	WindSp	
1	2008-12-01	Albury	13.4	22.9	0.6	NA	NA	W	44	W	WNW	20	
2	2008-12-02	Albury	7.4	25.1	0	NA	NA	WNW	44	NNW	WSW	4	
3	2008-12-03	Albury	12.9	25.7	0	NA	NA	WSW	46	W	WSW	19	
4	2008-12-04	Albury	9.2	28	0	NA	NA	NE	24	SE	E	11	
5	2008-12-05	Albury	17.5	32.3	1	NA	NA	W	41	ENE	NW	7	
6	2008-12-06	Albury	14.6	29.7	0.2	NA	NA	WNW	56	W	W	19	
7	2008-12-07	Albury	14.3	25	0	NA	NA	W	50	SW	W	20	

Truncated results, showing first 1,000 rows.

Figure 3. Load csv file and run demo

This step is important because the column objects must be converted to the numerical column format. We will need to give numbers to today's and tomorrow's precipitation to help the algorithm. A bar graph is used to illustrate the information. A chart that displays categorical data using rectangular bars whose height or length is proportionate to the representative values. After that, we will alter the date format to "year," "month," and "day" to facilitate comparisons. Obviously, modify the capitalization to lowercase. In addition, we verify that there are no nulls in the training. Calculating null values is essential for minimizing mistakes and simplifying model analysis.

```
# Convert object column into numerical column
train['RainToday'].replace({'No': 0, 'Yes': 1},inplace = True)
train['RainTomorrow'].replace({'No': 0, 'Yes': 1},inplace = True)
# Handling class imbalance
no = train[train.RainTomorrow == 0]
yes = train[train.RainTomorrow == 1]
yes_oversampled = resample(yes, replace=True, n_samples=len(no), random_state=123)
train = pd.concat([no, yes_oversampled])

fig = plt.figure(figsize = (8,5))
train.RainTomorrow.value_counts(normalize = True).plot(kind='bar', color= ['skyblue','navy'], alpha = 0.9, rot=0)
plt.title('RainTomorrow Indicator No(0) and Yes(1) after Oversampling (Balanced Dataset)')
plt.show()

# Changing the formate of column "Date"
train['Date'] = pd.to_datetime(train['Date'])

# Split the column into "year", "month", "date" and dropping the original
train['year'] = train.Date.apply(lambda x: x.year)
train['month'] = train.Date.apply(lambda x: x.month)
train['day'] = train.Date.apply(lambda x: x.day)
train.drop(columns=['Date'],axis=1, inplace = True)

train_cols=train.select_dtypes(['object']).columns
train[train_cols] = train[train_cols].astype(str).apply(lambda x: x.strip())

# Change all text to lower-case
train = train.applymap(lambda s: s.lower() if type(s) == str else s)
train.columns=train.columns.str.strip().str.lower()

# Checking for null values in training dataset
missing = pd.DataFrame(train.isnull().sum(),columns = ['no.of missing values'])

missing['% missing_values']= (missing/len(train)).round(2)*100

total = train.isnull().sum().sort_values(ascending=False)
```

Figure 4. Handle “Date” column and check null

After doing all the necessary steps, we will have to extract numerical features and here is the bar plot when we transform it into numerical.

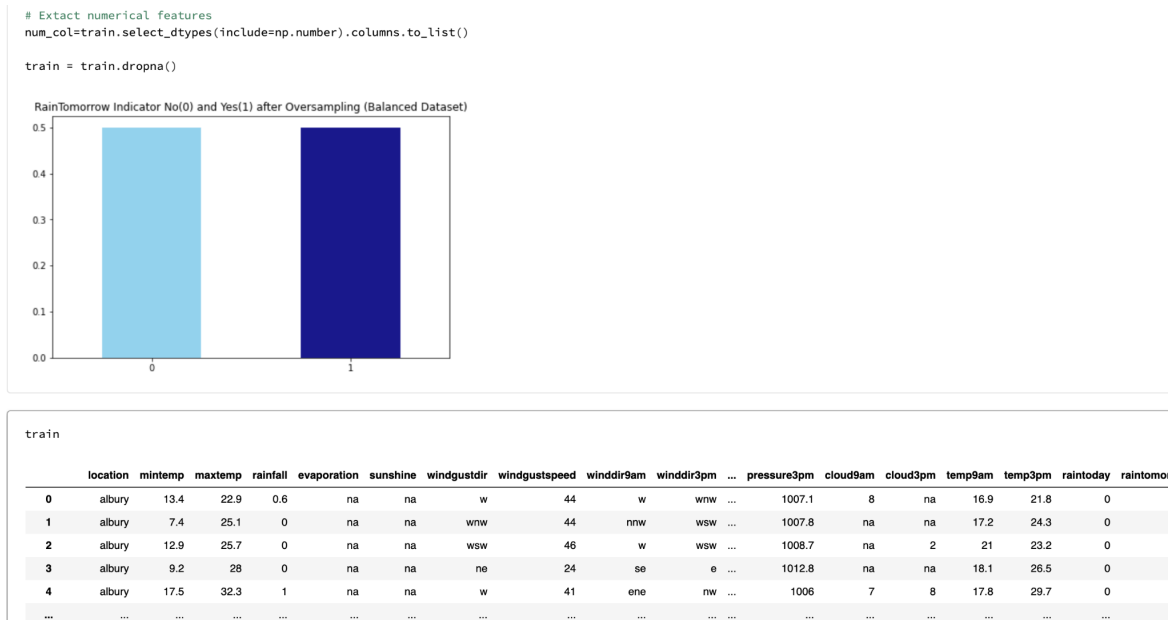


Figure 5. Bar plot with the train table

Model that we use in this case conflict with the data in OWM streaming data so we have to drop some information in our model, the OWM model lacks sufficient ["maxtemp", "rainfall", "evaporation", "sunshine", "windgustdir", "windgustspeed"] information. We also discovered that optimizing the cost of shipping data from Sagemaker would be prohibitively expensive, therefore it was important to restrict the returned data, and our team decided to retain only three cities: Melbourne, Sydney, and Albury. The remaining cities were sadly to abandon.

```
# modeling

# drop feature 3 4 5
train = train.drop(["maxtemp", "rainfall", "evaporation", "sunshine", "windgustdir", "windgustspeed"], axis=1)

drops = []
for i in range(len(train['location'])):
    if train['location'].iloc[i] in ['albury', 'melbourne', 'sydney']:
        drops.append(i)

train = train.iloc[drops]
```

Figure 6. Drop conflicting information and other cities

The crucial step is to apply the algorithm to the model. Here, we apply the Random Forest algorithm. RMSE (Root mean square deviation) and MAE (Mean absolute error) are utilized as assessment measures. This stage involves knowledge of the method and its appropriate implementation in order to obtain the exact and closest rounding parameters and limit large mistakes.

```
# # Define models and parameters
# from sklearn.ensemble import RandomForestClassifier

# rfm = RandomForestClassifier()
# n_estimators = [10, 100, 1000]
# max_features = ['sqrt', 'log2']

# # Define grid search
# grid = dict(n_estimators=n_estimators,max_features=max_features)
# cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
# rf_grid_search = GridSearchCV(estimator=rfm, param_grid=grid, n_jobs=-1, cv=cv, scoring='accuracy',error_score=0)
# rf_grid_result = rf_grid_search.fit(X_train, trainY)

# # Results
# print("Best parameters) ",rf_grid_result.best_params_)
# print("Model accuracy :",rf_grid_result.best_score_)



---


from sklearn.ensemble import RandomForestClassifier

scikit_model = RandomForestClassifier()
scikit_model.fit(trainX, trainY)

spark = SparkSession \
    .builder \
    .appName("Spark") \
    .config("spark.driver.bindAddress", "127.0.0.1") \
    .config("spark.python.worker.reuse", "true") \
    .getOrCreate()

spark_model = SklearnModel(scikit_model,spark)

# Train the Spark model using the PySpark DataFrame
model = spark_model.fit(trainX, trainY)

# Make predictions using the trained model
predictions = model.transform(testX)
predictions.show()
```

Figure 7. Apply Random Forest Algorithm

```
rfm_pred = scikit_model.predict(X_test)

rfm_r2=r2_score(rfm_pred, testY)
rfm_rmse = sqrt(mean_absolute_error(rfm_pred, testY))
print('Mean squared error: %.4f'
      % metrics.mean_squared_error(rfm_pred, testY))
print('Root mean absolute error: ', rfm_rmse)
print('Mean absolute error: ', mean_absolute_error(rfm_pred, testY))
print('R2 score is: ', rfm_r2)
print("F1 score:",f1_score(rfm_pred, testY))
print("Testing accuracy score: ",accuracy_score(rfm_pred, testY))
print(classification_report(testY, rfm_pred))
```

```
Mean squared error: 0.5373
Root mean absolute error: 0.732976740395481
Mean absolute error: 0.5372549019607843
R2 score is: 0.0
F1 score: 0.0
Testing accuracy score: 0.4627450980392157
```

	precision	recall	f1-score	support
0	0.46	1.00	0.63	1298
1	0.00	0.00	0.00	1507
accuracy			0.46	2805
macro avg	0.23	0.50	0.32	2805
weighted avg	0.21	0.46	0.29	2805

Figure 8. Appropriate evaluation metrics

2.2. Data Pipeline Architecture

With this dataset, we would like to apply a data pipeline that combines many services and platforms. This pipeline aims to address the cost and experience issues. The data will initially be loaded into MongoDB. Then, we will push the raw data through databricks and ETL it. The modified data will simultaneously be mounted with AWS S3 and stored on S3. After transforming and storing the data, it will be transmitted to SageMaker for model training and deployment. Each component of this data pipeline is detailed in depth below, along with its advantages and disadvantages.

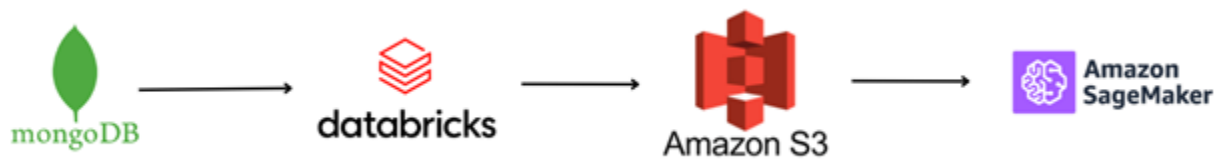


Figure 9. Data pipeline with Kaggle dataset

- **MongoDB:** When evaluating a cloud service to store data, MongoDB was the first service that came to mind. MongoDB is a document database utilized to develop highly accessible and scalable web applications [2]. As a document database, MongoDB simplifies the storage of organized and unstructured data for developers. The primary advantage of using MongoDB is its Scalability and Transactionality [2]. MongoDB's engineering advancements enable a vast number of reads and writes. It is typical to embed items within one another when modeling data in MongoDB. Consequently, this will be useful if we wish to update or add new data in the future.

At the beginning, to load data from local to MongoDB, we need to create a cluster. After create a new cluster, we need to create a connection between locally and MongoDB cluster on the cloud by using these following command (load_data_mongodb.ipynb):

1. Connect to MongoDB cluster on the cloud

```
import pymongo

# Replace this with your MongoDB cluster

client = pymongo.MongoClient("mongodb+srv://admin:admin123@assignment3.asq1nv.mongodb.net/?retryWrites=true&w=majority")
db = client.kaggle_dataset

# Issue the serverStatus command and print the results
serverStatusResult=db.command("serverStatus")
print(serverStatusResult)
```

Figure 10. Connect to MongoDB cluster

Then, we need to create a collection to store raw data on the MongoDB cluster.

2. Create a collection

In the first step, we create a new collection.

Add weather data

```
weatherAUS = pd.read_csv("weatherAUS.csv")

weather = weatherAUS.to_dict(orient='records')

db.weatherAUS.insert_many(weather)
```

Figure 11: Create a collection on MongoDB database

- **Databricks:** In the next step, we will push the raw data from MongoDB to Databricks for ETL. Databricks is a single, cloud-based platform that can manage all of the data needs, which means that our complete data team can cooperate on it. Databricks not only unifies and simplifies our data systems, but it is also quick, cost-effective, and naturally scalable to very huge data sets [3]. Therefore, to perform an ETL step on Databricks, we need to create a connection between these platforms and transfer data between them.

First, we need to create a connection between Databricks and MongoDB, then load the raw data from MongoDB to store on cluster of Databricks (connect_and_ETL.ipynb)

```
1 # Create a connection between Databricks and MongoDB, then load data from MongoDB to Databricks
2 df = spark.read.format("com.mongodb.spark.sql.DefaultSource").option("uri", "mongodb+srv://admin:admin123@assignment3.asq1nv.mongodb.net/?retryWrites=true&w=majority").option("database",
3 "kaggle_dataset").option("collection", "weatherAUS").load()
df.show()
```

Figure 12. Create and Load connection from MongoDB to Databricks

► (2) Spark jobs
 ► of: pyspark.sql.DataFrame + [CloudIp: double, CloudIam: double ... 22 more fields]

[CloudIp]	[CloudIam]	Date	[Evaporation]	[Humidity3pm]	[Humidity9am]	Location	[MaxTemp]	[MinTemp]	[Pressure3pm]	[Pressure9am]	RainToday	RainTomorrow	Rainfall	Sunshine	Temp3pm	Temp9am	WindDir3pm	WindDir9am	WindGustDir	WindGustSpeed	WindSpeed3pm	WindSpeed9am
24.0	NaN	2000-12-01	NaN	22.0	71.0	Albany	22.9	13.4	1007.1	1007.7	No	No	0.0	NaN	23.0	16.9	WSW	W	W	44.0	24.0	24.0
22.0	NaN	2000-12-02	NaN	25.0	44.0	Albany	25.1	7.4	1007.0	1010.6	No	No	0.0	NaN	24.3	17.2	WSW	WSW	WSW	44.0	22.0	22.0
26.0	NaN	2000-12-03	NaN	30.0	38.0	Albany	25.7	12.9	1008.7	1007.6	No	No	0.0	NaN	23.2	21.0	WSW	W	WSW	46.0	26.0	26.0
9.0	NaN	2000-12-04	NaN	16.0	45.0	Albany	28.0	9.2	1012.0	1017.6	No	No	0.0	NaN	26.5	18.1	E	SE	NE	24.0	9.0	9.0
20.0	NaN	2000-12-05	NaN	33.0	82.0	Albany	32.3	17.5	1006.0	1010.8	No	No	1.0	NaN	29.7	17.0	WSW	ENE	W	41.0	20.0	20.0
24.0	NaN	2000-12-06	NaN	23.0	55.0	Albany	29.7	14.6	1005.4	1009.2	No	No	0.2	NaN	28.9	20.6	W	W	WSW	56.0	24.0	24.0
24.0	NaN	2000-12-07	NaN	19.0	49.0	Albany	25.0	14.3	1008.2	1009.6	No	No	0.0	NaN	24.6	18.1	W	SW	W	50.0	24.0	24.0
24.0	NaN	2000-12-08	NaN	19.0	48.0	Albany	26.7	7.7	1010.1	1013.4	No	No	0.0	NaN	25.5	16.3	W	SSE	W	35.0	24.0	24.0

Figure 13. Results when load to Databricks

Then we will perform an ETL for this raw dataset. After that, we need to create a connection to AWS S3 and store the cleaned data on it. Besides this, the connection requires the access key and secret key of AWS. So we need to read the csv file with AWS keys to Databricks. Then, we access these keys via the CSV file.

Cmd 11

```

1 # Read the csv file with AWS keys to databricks
2 # Define file type
3 file_type = "csv"
4 # Whether the file has a header
5 first_row_is_header = "true"
6 # Delimiter used in the file
7 delimiter = ","
8 # Read the CSV file to spark dataframe
9 aws_keys_df = spark.read.format(file_type)\
10 .option("header", first_row_is_header)\
11 .option("sep", delimiter)\
12 .load("/FileStore/tables/asm3user1_accessKeys.csv")

```

Figure 14. Get AWS keys

Cmd 12

```

1 ACCESS_KEY = aws_keys_df.select('Access key ID').take(1)[0]['Access key ID']
2 SECRET_KEY = aws_keys_df.select('Secret access key').take(1)[0]['Secret access key']
3
4 ENCODED_SECRET_KEY = urllib.parse.quote(string=SECRET_KEY, safe='')

```

Figure 15. Access AWS keys

After creating the connection, we load the cleaned data to AWS S3 using the Boto3 library and place it as a CSV file.

```

Cmd 13

1  from io import StringIO
2  import boto3
3
4  s3_resource = boto3.resource('s3',
5                               aws_access_key_id=ACCESS_KEY,
6                               aws_secret_access_key= SECRET_KEY)
7  AWS_S3_BUCKET = 'chrisho251'
8  csv_buffer = StringIO()
9  train.to_csv(csv_buffer)
10 s3_resource.Object(AWS_S3_BUCKET, 'kaggle_cleaned_data.csv').put(Body=csv_buffer.getvalue())
  
```

Figure 16. Save the cleaned data to AWS S3

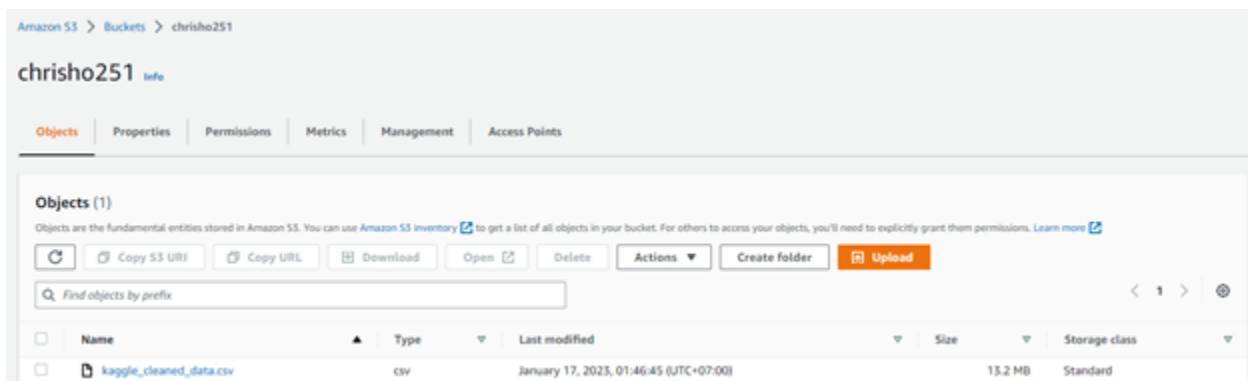


Figure 17. Loaded successful to AWS S3

- **AWS SageMaker:** The final step is to train the model with the cleaned data. In this step, we will use AWS SageMaker to perform it. Briefly describing Amazon SageMaker, it offers a single, web-based visual interface via which all Machine Learning development phases may be executed, hence boosting the efficiency of data science teams by up to 10 times. Amazon SageMaker Studio provides customers with total access, control, and insight into each step required to construct, train, and deploy models. The data can be uploaded, new notebooks can be created, training and tuning of models, traveling back and forth between phases to tweak trials, comparing findings, and finally deploying the models into production can all occur in a single location, enabling users to become productive. SageMaker Studio can typically be used to do all Machine Learning development tasks, including notebooks, experiment management, automatic model generation, debugging, and detection of model and data drift [4]. And in this stage, we'll build a notebook file and use it to train the model.

After logging in to the Amazon SageMaker portal, select Notebook instances from the left-hand navigation pane, and then select the option to Create a new notebook instance.

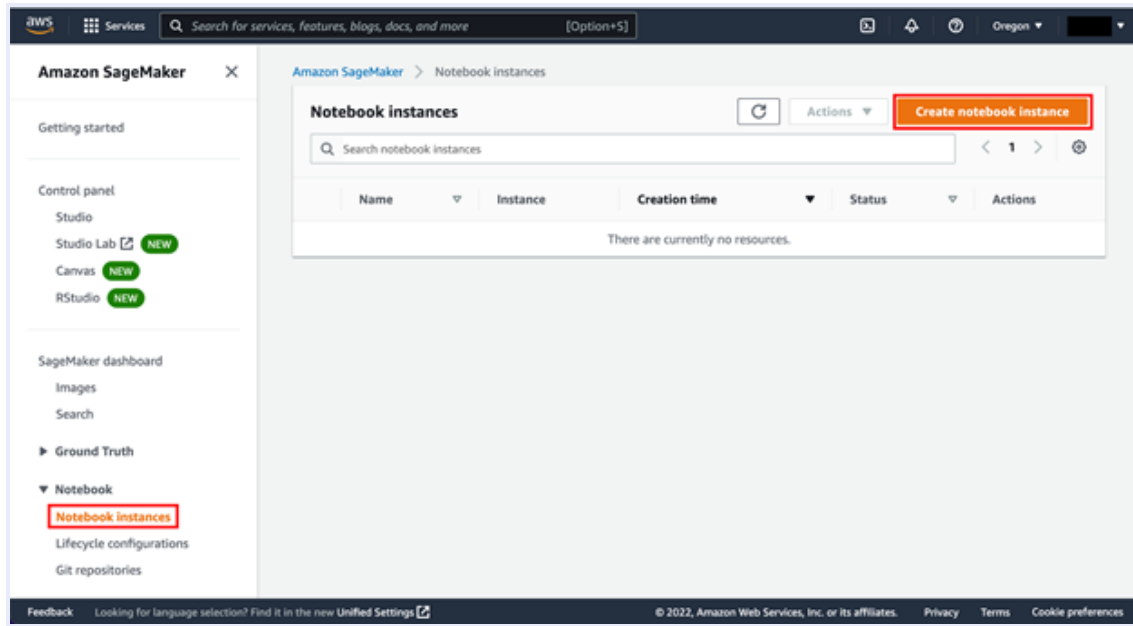


Figure 18. Create new Notebook instances

The newly created SageMaker-Tutorial notebook instance is presented with a Status of Pending after the new Notebook has been created successfully. When the Status changes to "InService," this indicates that the notebook is now ready for use. Choose Open Jupyter when the state of the SageMaker-Tutorial notebook instance you are using switches to "InService." After that, select New, and after that, select conda python3 from the drop-down menu.

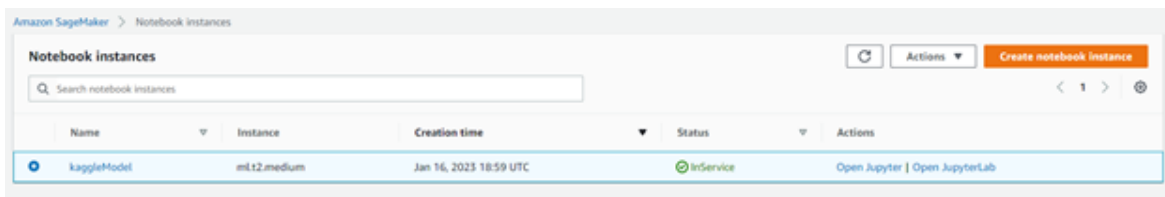


Figure 19. Successful create a new Notebook

In the Jupyter portal, create a new Notebook with "conda_python3" type (for exp: kaggle_model.ipynb)

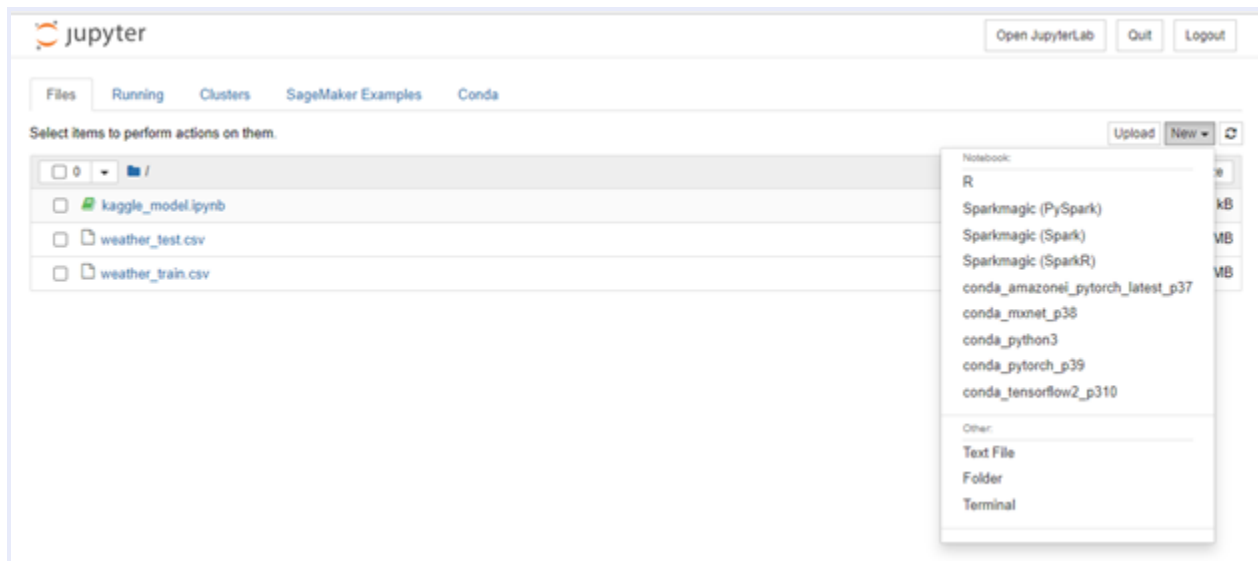


Figure 20. Create a new Notebook

After that, we connect with AWS S3 and pull data to place on Sagemaker. Since a new data frame, these data are ready for training.



Figure 21. Get data from AWS S3

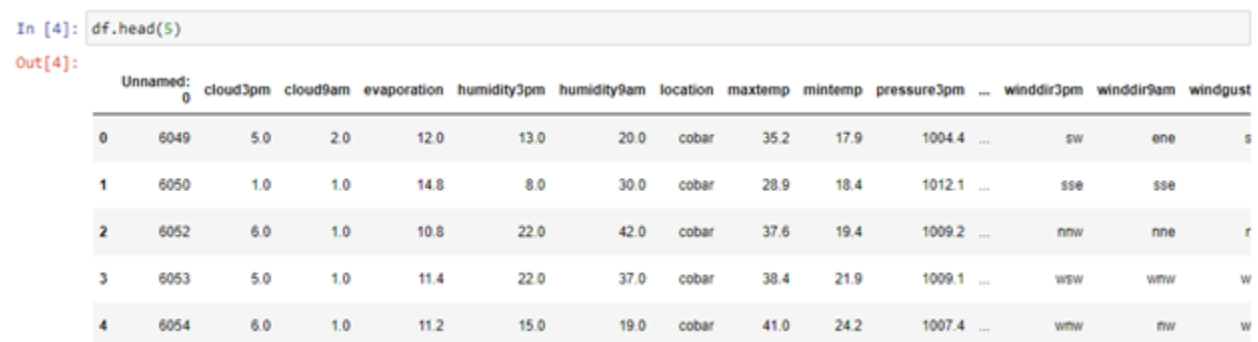


Figure 22. Create a new data frame with selected dataset


```
In [14]: rfm_pred = scikit_model.predict(X_test)

rfm_r2 = r2_score(rfm_pred, testY)
rfm_rmse = sqrt(mean_absolute_error(rfm_pred, testY))
print('Mean squared error: %.4f'
      % metrics.mean_squared_error(rfm_pred, testY))
print('Root mean absolute error: ', rfm_rmse)
print('Mean absolute error: ', mean_absolute_error(rfm_pred, testY))
print('R2 score is: ', rfm_r2)
print('F1 score:', f1_score(rfm_pred, testY))
print('Testing accuracy score: ', accuracy_score(rfm_pred, testY))
print(classification_report(testY, rfm_pred))

/home/ec2-user/anaconda3/envs/python3/lib/python3.10/site-packages/sklearn/base.py:450: UserWarning: X does not have valid feature names, but RandomForestClassifier was fitted with feature names
  warnings.warn(

Mean squared error: 0.4874
Root mean absolute error: 0.6981354301081508
Mean absolute error: 0.4873930787722927
R2 score is: 0.0
F1 score: 0.0
Testing accuracy score: 0.5126069212277072
      precision    recall  f1-score   support

     0       0.51      1.00      0.68       9169
     1       0.00      0.00      0.00        8718

 accuracy          0.26          0.50          0.34       17887
 macro avg          0.26          0.51          0.35       17887
 weighted avg          0.26          0.51          0.35       17887
```

Figure 23. Result when training model with AWS Sagemaker

2.3. Pricing

Service	Price	Amount	Total cost
MongoDB	- Free tier		Free
Databricks Community	- Free tier		Free
Amazon S3	- Free tier	12 months free	Free
Amazon SageMaker	- Free trial two months - Calculated after two month free trial	- 4 data scientist(s) x 1 Studio Notebook instance(s) = 4.00 Studio Notebook instance(s) - 4.00 Studio Notebook instance(s) x 4	Total cost for Studio Notebooks (monthly): 903.04 USD

		hours per day x 20 days per month = 320.00 SageMaker Studio Notebook hours per month - 320.00 hours per month x 2.822 USD per hour instance cost = 903.04 USD(monthly On-Demand cost)	
--	--	---	--

Table 1. Pricing of Historical data pipeline

3. Data Pipelines For OWM

Our data pipeline architecture is designed as below. We use AWS Lambda to extract data from Open Weather Map (OWM) API and deliver it to AWS Kinesis Firehose every hour being scheduled by AWS Cloudwatch. From Firehose, the data will be transferred and inserted into MongoDB Cloud using the MongoDB App Services.

3.1. Handling Open Weather Map Real Time Data

As declared above and knowing the features of the machine learning model, OWM data needs to be handled and transformed properly so that it can be used to put into the deployed model to produce prediction afterwards. This section focuses mainly on the data transformation, specific detail of data flow is discussed further in the Data Pipelines section.

Firstly, each hour, by calling the OWM API, we got the following result in JSON format:

```

"coord": {
  "lon": 151.2073,
  "lat": -33.8679
},
"weather": [
  {
    "id": 804,
    "main": "Clouds",
    "description": "overcast clouds",
    "icon": "04n"
  }
],
"base": "stations",
"main": {
  "temp": 292.28,
  "feels_like": 292.55,
  "temp_min": 289.84,
  "temp_max": 293.36,
  "pressure": 1020,
  "humidity": 88
},
"visibility": 10000,
"wind": {
  "speed": 1.01,
  "deg": 19,
  "gust": 1.91
},
"clouds": {
  "all": 97
},
"dt": 1673893923,
"sys": {
  "type": 2,
  "id": 2002865,
  "country": "AU",
  "sunrise": 1673895659,
  "sunset": 1673946513
},
"timezone": 39600,
"id": 2147714,
"name": "Sydney",
"cod": 200

```

Figure 24. Open Weather Map Api response

This is the current weather data in Sydney. Our goal is to extract the necessary value as defined. At this step, **Pandas**, a Python library that supports processing data frames is used to transform the API response to usable data. The response is converted to a pandas dataframe with selected fields prior to engineering the features.

```

# convert json response to a pandas df
# record_path = weather to get the nested data
# meta to get the necessary fields
columns = [['main', 'temp'], ['main', 'temp_min'], ['main', 'temp_max'], ['main', 'pressure'], [
  'main', 'humidity'], ['wind', 'speed'], ['wind', 'deg'], ['wind', 'gust'], ['clouds', 'all']]
df = pd.json_normalize(data, record_path='weather',
  meta=columns, errors='ignore')

```

Figure 25. Converting API response to pandas dataframe with selected fields

Feature Engineering steps are later implemented with certain awareness such as units that are used for the model features and the recorded timestamp for later queries. Most of the fields are in exact units with the special exception of WindDir value, which is originally in degree and needs to be converted into cardinal direction. Moreover, the value that determines the occurrence of rain on the current day is extracted through the 'main' attribute of the response.

```
def degrees_to_cardinal(degree):
    dirs = ['N', 'NNE', 'NE', 'ENE', 'E', 'ESE', 'SE', 'SSE',
            'S', 'SSW', 'SW', 'WSW', 'W', 'WNW', 'NW', 'NNW']
    ix = round(degree / (360 / len(dirs)))
    return dirs[ix % len(dirs)]
```

Figure 26. WindDirection convert function

```
df['main'].mask(df['main'] == 'Rain', 1, inplace=True)
df['main'].mask(df['main'] != 'Rain', 0, inplace=True)
```

Figure 27. Create a field to determine the rain occurrence

Proper data type is also a consideration, hence, redefine step is also included in this step. Here is the final data that is put into MongoDB after being extracted and transformed through AWS Kinesis and AWS Lambda functions at the first stages of the data pipeline.

Raining	Temp	MinTemp	MaxTemp	Pressure	Humidity	WindSpeed	WindDeg	WindGustSpeed	Cloud	Date	Time	WindDir
0	20.57	19.69	21.27	1013.0	75.0	4.63	170.0	NaN	75	2023-01-11	00:14:22	S

Figure 28. Transformed record at that moment

A MongoDB schema is used to store the record in different hours of the day. In order to proceed to the prediction stage, another transformation step is implemented to create data for the current day. This step is executed in another Lambda function in the pipeline. The objective of this step is coming up with data that has the same features with the trained model. Mostly it is related to aggregating data to get certain features at 9am and 3pm. The method is to get the recorded data at the moments that are closest to 3pm and 9am.

```
today_df['delta_9am'] = today_df['Hour'].apply(lambda x: delta(x, 9))
today_df['delta_3pm'] = today_df['Hour'].apply(lambda x: delta(x, 15))

drop_col = ['_id', 'Time', 'Raining', 'MaxTemp', 'MinTemp', 'WindDeg',
            'WindGustSpeed', 'Hour', 'delta_9am', 'delta_3pm']

dict_col_for9am = {'Cloud': 'cloud9am', 'Humidity': 'humidity9am', 'Pressure': 'pressure9am',
                  'Temp': 'temp9am', 'WindDir': 'winddir9am', 'WindSpeed': 'windspeed9am'}
dict_col_for3pm = {'Cloud': 'cloud3pm', 'Humidity': 'humidity3pm', 'pPressure': 'pressure3pm',
                  'Temp': 'temp3pm', 'WindDir': 'winddir3pm', 'WindSpeed': 'windspeed3pm'}

df_for9am = today_df[today_df.delta_9am ==
                    today_df.delta_9am.min()].drop(drop_col, axis=1).rename(columns=dict_col_for9am)
df_for3pm = today_df[today_df.delta_3pm ==
                    today_df.delta_3pm.min()].drop(drop_col, axis=1).rename(columns=dict_col_for3pm)

df_res = today_df.groupby('Date').agg(
    mintemp=('MinTemp', 'min'), maxtemp=('MaxTemp', 'max'))

df_res = pd.concat([df_res, df_for9am.set_index('Date'),
                  df_for3pm.set_index('Date')], axis=1)
```

Figure 29. Create fields at 9am and 3pm

For raintoday value, it is determined whether there are any recorded values that have a raining field = 1, which displays there was rain on that day.

```
# get RainToday value
df_res['raintoday'] = 1 if 1.0 in today_df['Raining'] else 0
```

Figure 30. Field raintoday

The final data has the exact features with the model and is written to a CSV file save s3 that is loaded to Sagemaker with the deployed model and proceeds to make predictions. This step constantly updates the data for the day and guarantees there will always be available data to predict for tomorrow.

Date	mintemp	maxtemp	cloud9am	humidity9am	pressure9am	temp9am	winddir9am	windspeed9am	cloud3pm	humidity3pm	Pressure	temp3pm	winddir3pm	windspeed3pm	year	month	day	raintoday	location
2023-01-17	24.1	28.2	40.0	64.0	1018.0	299.48	ENE	6.17	40.0	64.0	1018.0	299.48	ENE	6.17	2023	1	17	0	syd

Figure 31. Data for the current day

3.2. Required libraries/packages:

- We use python with the open-source framework AWS Serverless Application Model (SAM) to build the pipeline from lambda to Firehose and extract the data from MongoDB to S3. SAM uses the YAML to define applications and support converting SAM syntax to AWS CloudFormation syntax when deployed which makes the building process faster.
- There are some packages that are required for successfully deploying the data pipeline, including boto3, requests, pymongo, IPython, and pandas. Conda is used to establish a python3.8 environment in order to install all the necessary packages since we need to construct a virtual environment to run locally using Docker.

3.3. Data pipeline architecture:

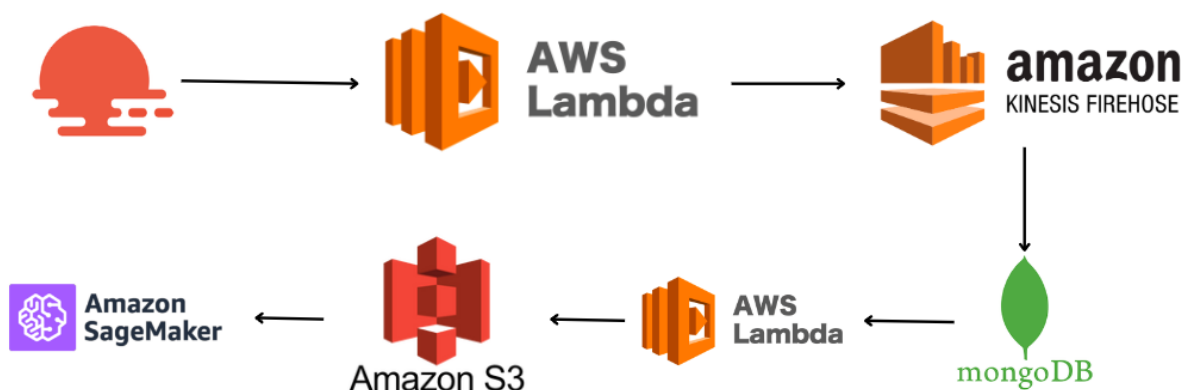


Figure 32. Example setting the image lambda function


- **AWS Lambda Service:** Due to the daily data call restriction, we must schedule the event to occur after 1 hour using the AWS CloudWatch service. We will utilize AWS Lambda to initiate the data ingestion processes, and then we process the data and transform it (we had mentioned details in the Solution) before uploading the records to the Kinesis Firehose. Besides the ingestion functions, we also have the other function called OWMHoursToDayFunction, every hour, this function is used to extract data from MongoDB and aggregate them to data for the current day and send it to CSV files stored in an S3 bucket requires integration with AWS CloudWatch to define the schedule for execution.

```

Globals:
  Function:
    Timeout: 900
    MemorySize: 128

Resources:
  IngestionOWMFunction:
    Type: AWS::Serverless::Function
    Properties:
      PackageType: Image
      MemorySize: 128
      Timeout: 900
      Architectures:
        - x86_64
      Events:
        CheckWebsiteScheduledEvent1:
          Type: Schedule
          Properties:
            Schedule: rate(1 hour)
      Metadata:
        Dockerfile: Dockerfile
        DockerContext: ./owm
        DockerTag: python3.8-v1
  
```

Figure 33. Example setting the image lambda function


Execution result: succeeded ([logs](#))

▼ Details

The area below shows the last 4 KB of the execution log.

```
{
  "records": [
    [
      {
        "Raining": 0,
        "Temp": 292.63,
        "MinTemp": 290.7,
        "MaxTemp": 293.76,
        "Pressure": 1020.0,
        "Humidity": 88.0,
        "WindSpeed": 1.54,
        "WindDeg": 140.0,
        "WindGustSpeed": NaN,
        "Cloud": 20,
        "Date": "2023-01-16",
        "Time": "20:24:20",
        "WindDir": "SE"
      }
    ]
  ]
}
```

Summary	
Code SHA-256	Request ID
055151d4546bbe518ee7ea5d85ce3350764e962ee10cd1a674d6ad334f48711e	ead472a9-e67f-40e9-b972-0f54527fd39c
Duration	Billed duration
2463.89 ms	2464 ms
Resources configured	Max memory used
128 MB	127 MB

Log output

The section below shows the logging calls in your code. [Click here](#) to view the corresponding CloudWatch log group.

```
START RequestId: ead472a9-e67f-40e9-b972-0f54527fd39c Version: $LATEST
END RequestId: ead472a9-e67f-40e9-b972-0f54527fd39c
REPORT RequestId: ead472a9-e67f-40e9-b972-0f54527fd39c  Duration: 2463.89 ms  Billed Duration: 2464 ms  Memory Size: 128 MB  Max Memory Used: 127 MB
```

Figure 34. Example result for Lambda

- **AWS Kinesis Firehose:** We select the AWS Kinesis Firehose to construct the data pipeline. The data is directly transferred into storage services like AWS S3 bucket or AWS DynamoDB via the almost real-time streaming pipeline known as AWS Kinesis Firehose. In this instance, MongoDB will be the location where the streaming data is ingested. AWS Kinesis Data Stream is a different streaming pipeline that is comparable. In comparison to Firehose, Kinesis Data Stream offers real-time data streaming that is extremely scalable and reliable. The cost is one issue that is important, though. For Firehose, you only need to pay for the data you transmit over the service, while for Kinesis Data Stream, you also need to pay for shards per hour in addition to the data you send.
- **MongoDB Cloud:** All the data in the Kinesis Firehose will go directly into MongoDB, the NoSQL database. MongoDB Cloud is the Cloud services of MongoDB, providing the App Services which help us with the HTTPS Endpoints to export the Url for the AWS Kinesis Firehose to connect. Because the data sent from Firehose has been encoded with base64, MongoDB App Services provides the Functions (work nearly the same as AWS Lambda Service) to help us decode the data and transform it before putting it into the database. We decide to use MongoDB instead of AWS DynamoDB or AWS Redshift because the free tier of MongoDB provides us 1 millions requests and 10GB free storage, enough for us in this project.

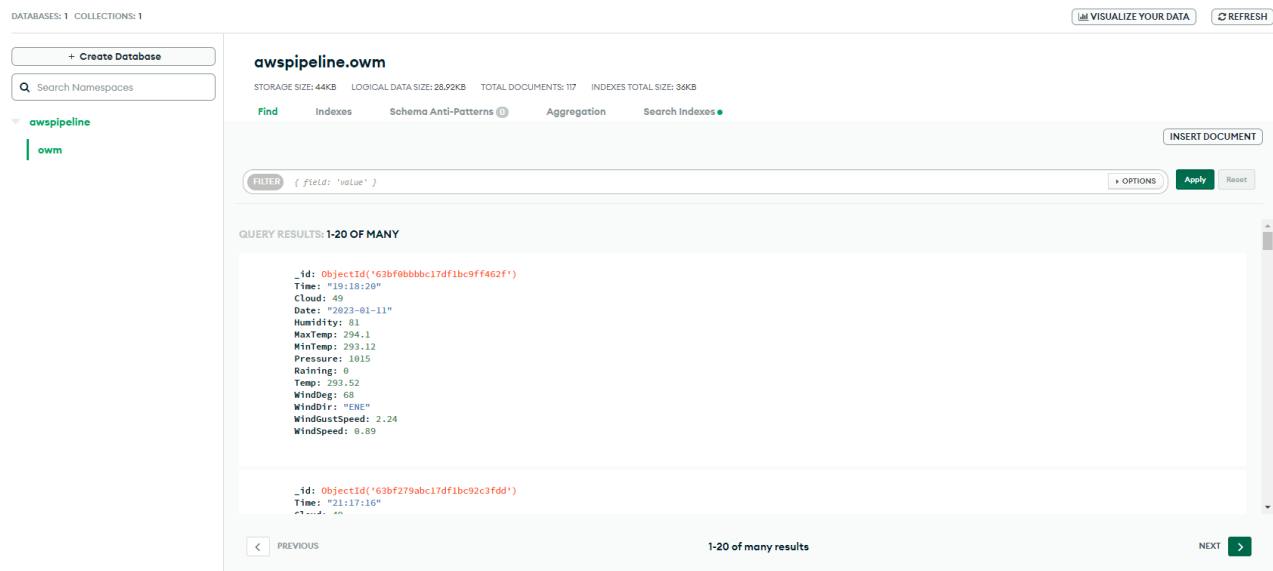


Figure 35. Data stored in MongoDB Cloud

3.4. Pricing

Because of the Lambda invoked every 1 hours, this estimated cost will calculate the pipeline per week

Service	Price	Amount	Total cost
Amazon Lambda	\$0.20 per 1M requests	- 48 requests / day - 336 requests / week	\$0.0000672
	\$0.0000166667 for every GB-second	- 128 MB / request - 2s / request - 64 MB / s	~\$0.00035
Amazon CloudWatch	Free tier		Free
Amazon Cloudformation	- Free tier - \$0.0009 per handler operation)	2 handlers	\$0.0126
Amazon Kinesis Firehose	- Free tier - \$0.029 per GB of data ingested	1 streams and ~168 records (~246B / record)	~\$0.000001
MongoDB	- Free tier		Free

Table 2. Pricing of Streaming data pipeline

III. CONCLUSION

Despite the fact that the big picture is complete and we can anticipate rain for tomorrow, this project still has several constraints. This can be a premise for the project to develop even more and from there develop a comprehensive system for weather forecasting. Although the project is for educational reasons solely, we can foresee that it will grow in the future. There are many ways to enhance and polish the whole project but are considered out of scope due to the limited time to work on and one of those is using newly produced data to reinforce the trained dataset so that the model is freshly updated continuously so that it gives better performances.

IV. REFERENCES

- [1] "Rain in Australia", Kaggle.com, 2021. [Online]. Available:
<https://www.kaggle.com/jsphyg/weather-dataset-rattle-package>. [Accessed: 11- Jan2022]
- [2] "What is mongodb?," *MongoDB*. [Online]. Available:
<https://www.mongodb.com/what-is-mongodb>. [Accessed: 15-Jan-2023].
- [3] "What is Databricks?," *What is Databricks? | Databricks on AWS*. [Online]. Available:
<https://docs.databricks.com/introduction/index.html>. [Accessed: 15-Jan-2023].
- [4] P. Forte, "PM," *Amazon*, 1976. [Online]. Available:
https://aws.amazon.com/pm/sagemaker/?trk=a5d4c613-ebfe-4261-8673-3abec6168005&sc_channel=ps&s_kwid=AL%214422%2110%2171399764321901%2171400284796187&ef_id=d2f0d206e0ea1b265d10b1c0ffbcbfa7%3AG%3As. [Accessed: 15-Jan-2023].

Github repo for code:

Historical Data: <https://github.com/tringuyenbao/historical-data>

AWS Weather Pipeline: <https://github.com/ducnd58233/aws-weather-pipeline>

AWS OWM Hour To Day: <https://github.com/ducnd58233/aws-owm-hour-to-day>