MLOps Pipeline using Apache Airflow

Mauzum shamil

Machine Learning Operations (MLOps) integrates machine learning model development and deployment into a robust, automated pipeline.

In this Project, I'll explain how to build an MLOps pipeline using Apache Airflow to automate preprocessing, model training, and deployment tasks.

about the Dataset

The given dataset contains app usage behaviour with five key columns:

-Date (usage day)

-App (e.g., Instagram, WhatsApp)

-Usage (minutes spent)

-Notifications (alerts received)

-=and Times Opened (app launches).

Introduction of the project

The goal of this pipeline is to streamline the process of analyzing screentime data by automating its preprocessing and utilizing machine learning to predict app usage. To ensure seamless execution, we will design an Airflow DAG to schedule and automate daily data preprocessing tasks to support a robust and scalable workflow.

data preprocessing

In [1]:
```
import pandas as pd
```

In [2]:
```
from sklearn.preprocessing import MinMaxScaler
```

In [3]:
```
data = pd.read_csv(r"C:\Users\mauzu\Downloads\screentime_analysis.csv")
```

In [7]:
```
data
```

Out[7]:

	Date	App	Usage (minutes)	Notifications	Times Opened
0	2024-08-07	Instagram	81	24	57
1	2024-08-08	Instagram	90	30	53
2	2024-08-26	Instagram	112	33	17
3	2024-08-22	Instagram	82	11	38
4	2024-08-12	Instagram	59	47	16
...	...	...	...	...	...
195	2024-08-10	LinkedIn	22	12	5
196	2024-08-23	LinkedIn	5	7	1
197	2024-08-18	LinkedIn	19	2	5
198	2024-08-26	LinkedIn	21	14	1
199	2024-08-02	LinkedIn	13	4	1

200 rows × 5 columns

In [6]:
```
# checking for missing values and duplicates
print(data.isna().sum())
```

```
Date              0
App               0
Usage (minutes)   0
Notifications     0
Times Opened      0
dtype: int64
```

In [5]:
```
print(data.duplicated().sum())
```

```
0
```

In [8]:
```
# convert DAte column to datetime and extract features
data['Date'] = pd.to_datetime(data['Date'])
data['DayOfWeek'] = data['Date'].dt.dayofweek
data['Month'] = data['Date'].dt.month
```

In [31]:
```
# encode the categorical 'App' column using one hot encoding
data = pd.get_dummies(data,columns=['App'], drop_first=True)
```

In [10]:
```
data
```

Out[10]:

	Date	App	Usage (minutes)	Notifications	Times Opened	DayOfWeek	Month
0	2024-08-07	Instagram	81	24	57	2	8
1	2024-08-08	Instagram	90	30	53	3	8
2	2024-08-26	Instagram	112	33	17	0	8
3	2024-08-22	Instagram	82	11	38	3	8
4	2024-08-12	Instagram	59	47	16	0	8
...	...	...	...	...	...	...	...
195	2024-08-10	LinkedIn	22	12	5	5	8
196	2024-08-23	LinkedIn	5	7	1	4	8
197	2024-08-18	LinkedIn	19	2	5	6	8
198	2024-08-26	LinkedIn	21	14	1	0	8
199	2024-08-02	LinkedIn	13	4	1	4	8

200 rows × 7 columns

In [12]:
```
# Scale numerical features using MinMaxScaler
Scaler = MinMaxScaler()
data[['Notifications','Times Opened']] = Scaler.fit_transform(data[['Notificatio
```

In [13]:
```
data
```

Out[13]:

	Date	App	Usage (minutes)	Notifications	Times Opened	DayOfWeek	Month
0	2024-08-07	Instagram	81	0.163265	0.571429	2	8
1	2024-08-08	Instagram	90	0.204082	0.530612	3	8
2	2024-08-26	Instagram	112	0.224490	0.163265	0	8
3	2024-08-22	Instagram	82	0.074830	0.377551	3	8
4	2024-08-12	Instagram	59	0.319728	0.153061	0	8
...	...	...	...	...	...	...	...
195	2024-08-10	LinkedIn	22	0.081633	0.040816	5	8
196	2024-08-23	LinkedIn	5	0.047619	0.000000	4	8
197	2024-08-18	LinkedIn	19	0.013605	0.040816	6	8
198	2024-08-26	LinkedIn	21	0.095238	0.000000	0	8
199	2024-08-02	LinkedIn	13	0.027211	0.000000	4	8

200 rows × 7 columns

In [15]:
```
# feature engineering
data['Previous_Day_Usage'] = data['Usage (minutes)'].shift(1)
data['Notifications_x_Timesopened'] = data['Notifications'] * data['Times Opened
```

In [32]:
```
data
```

Out[32]:

	Date	Usage (minutes)	Notifications	Times Opened	DayOfWeek	Month	Previous_Day_Usage
1	2024-08-08	90	0.204082	0.530612	3	8	81.0
2	2024-08-26	112	0.224490	0.163265	0	8	90.0
3	2024-08-22	82	0.074830	0.377551	3	8	112.0
4	2024-08-12	59	0.319728	0.153061	0	8	82.0
5	2024-08-28	50	0.285714	0.255102	2	8	59.0
...	...	...	...	...	...	...	...
195	2024-08-10	22	0.081633	0.040816	5	8	19.0
196	2024-08-23	5	0.047619	0.000000	4	8	22.0
197	2024-08-18	19	0.013605	0.040816	6	8	5.0
198	2024-08-26	21	0.095238	0.000000	0	8	19.0
199	2024-08-02	13	0.027211	0.000000	4	8	21.0

199 rows × 15 columns

In [33]:
```
data.isna().sum()
```

Out[33]:
```
Date                          0
Usage (minutes)               0
Notifications                 0
Times Opened                  0
DayOfWeek                     0
Month                         0
Previous_Day_Usage            0
Notifications_x_Timesopened   0
App_Facebook                  0
App_Instagram                 0
App_LinkedIn                  0
App_Netflix                   0
App_Safari                    0
App_WhatsApp                  0
App_X                         0
dtype: int64
```

In [26]:
```
# Drop rows with missing values
data = data.dropna(subset=['Previous_Day_Usage'])
```

In [46]:
```
# save the preprocessed data to a file
data.to_csv('preprocessed_screentime_analysis.csv', index=False)
```

The above code performs data preprocessing to prepare the screentime dataset for machine learning. It begins by loading the dataset and ensuring data quality through checks for missing values and duplicates. It then processes the Date column to extract useful temporal features like DayOfWeek and Month. The App column is transformed using one-hot encoding to convert it into a numeric format.

The process scales numerical columns, such as Notifications and Times Opened, using MinMaxScaler to ensure uniformity. Feature engineering creates lagged (Previous_Day_Usage) and interaction (Notifications_x_TimesOpened) features to enhance predictive power.

Train the model

After preprocessing we will do random forest on the model to predict app usage

In [18]:
```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error
```

In [34]:
```
# split the data into features and target variable
x = data.drop(columns=['Usage (minutes)','Date'])
y = data['Usage (minutes)']
```

In [35]:
```
# train test split

x_train, x_test, y_train, y_test = train_test_split(x,y, test_size=0.2, random_s
```

In [36]:
```
# train the model
model = RandomForestRegressor(random_state=42)
model.fit(x_train,y_train)
```

Out[36]:
```
RandomForestRegressor
RandomForestRegressor(random_state=42)
```

In [37]:
```
# evaluate the model
predictions = model.predict(x_test)
mae = mean_absolute_error(y_test,predictions)
print(f'mean_absolute_error: {mae}')
```

```
mean_absolute_error: 17.89475
```

In the above code, we are splitting the preprocessed data into training and testing sets, training a Random Forest Regressor model, and evaluating its performance.

First, the process separates the target variable (Usage (minutes)) from the features and performs an 80-20 train-test split. The training data is used to train the RandomForestRegressor model. After completing the training, the model generates

predictions on the test set, and the Mean Absolute Error (MAE) metric quantifies the average difference between the predicted and actual values to assess performance.

The Mean Absolute Error (MAE) of 17.89475 indicates that, on average, the model's predicted screentime differs from the actual screentime by approximately 15.4 minutes. This gives a measure of the model's predictive accuracy, showing that while the model performs reasonably well, there is still room for improvement in reducing this error to make predictions more precise.

automating preprocessing with a Pipeline using Apache Airflow

Apache Airflow enables the automation of tasks using Directed Acyclic Graphs (DAGs). Here, we will use a DAG to build a pipeline to preprocess data daily. First, install Apache Airflow:

What is an Airflow DAG? Think of an Airflow DAG (Directed Acyclic Graph) as a map for a journey that consists of several stops or tasks. Here's how the journey works:

DAG Definition:

Imagine writing a story. The DAG is like the title of your story, where you define the start date, how often the story should be told (schedule), and other settings.

Operators:

These are the action scenes in your story. Each operator represents a task that needs to be completed, such as running a script or calling a function.

Task Dependencies:

This is the sequence of events. Just like in a story where one event leads to another, task dependencies determine the order in which tasks are executed.

Schedule:

This is the timing of your story. You decide how often the story is retold (e.g., daily, weekly) using CRON expressions or preset intervals.

Task Instances:

When you tell your story (run your DAG), Airflow creates instances of each task, like actors performing scenes, to complete the journey.

In [ ]:
```
pip install apache-airflow
```

define DAG and task to build the pipeline

In [49]:
```
# define the data preprocessing function
```

In [55]:
```
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime,timedelta

def preprocess_data():
    file_path = r"C:\Users\mauzu\Downloads\screentime_analysis.csv"
    data = pd.read_csv(file_path)

    data['Date'] = pd.to_datetime(data['Date'])
    data['DayOfWeek'] = data['Date'].dt.dayofweek
    data['Month'] = data['Date'].dt.month

    data = data.drop(columns=['Date'])

    data = pd.get_dummies(data, columns=['App'], drop_first=True)

    data = data.dropna(subset=['Previous_Day_Usage'])

    scaler = MinMaxScaler()
    data[['Notifications', 'Times Opened']] = scaler.fit_transform(data[['Notifi

    preprocessed_path = 'preprocessed_screentime_analysis.csv'
    data.to_csv(preprocessed_path, index=False)
    print(f"Preprocessed data saved to {preprocessed_path}")

# define the DAG
dag = DAG(
    dag_id = 'data_preprocessing',
    schedule_interval ='@daily',
    start_date=datetime.now() + timedelta(days=1),
    catchup = False,
)

# define the task
preprocess_task = PythonOperator(
    task_id = 'preprocess',
    python_callable = preprocess_data,
    dag=dag,
)
```

```
C:\Users\mauzu\AppData\Local\Temp\ipykernel_32320\4175664560.py:27 Removed
```

The above code defines a Directed Acyclic Graph with a single task to preprocess screentime data. The preprocess_data function loads the dataset, extracts temporal features (DayOfWeek and Month) from the Date column, encodes the App column using one-hot encoding, and scales numerical features (Notifications and Times Opened) using MinMaxScaler.

Next, the system saves the processed data to a new CSV file. The Airflow DAG schedules this task daily, which ensures automation and reproducibility in the data preparation process.

Setting Up Apache Airflow for MLOps Pipelines

1. Initializing the Database

```
airflow db init
```

When you run this command, it sets up the backend database that Airflow uses to store its metadata. This includes information about your tasks, DAGs (Directed Acyclic Graphs), task execution history, and schedules. Think of it as the brain of your Airflow setup—it keeps track of everything that's happening so that your workflows can run smoothly.

2. Starting the Webserver

```
airflow webserver --port 8080
```

The webserver is like the control panel for Airflow. When you start it, it opens up a web-based interface that you can access through your browser at http://localhost:8080.

Here, you can:

See a dashboard that shows all your DAGs and their current status. Dive into details for each task, view logs, and manage your DAGs (like turning them on or off or manually triggering them). While 8080 is the default port, you can specify a different one if it's already in use or if you prefer another.

3. Starting the Scheduler

```
airflow scheduler
```

The scheduler is responsible for executing tasks according to the schedule defined in your DAGs. It continuously monitors the state of each task and determines when they should run.

For example:

-If a task is scheduled to run daily at midnight, the scheduler will ensure it gets executed at that time.

-It takes care of task dependencies, ensuring tasks run in the correct order.

Accessing the Airflow UI

Once you have the webserver running, you can access the Airflow UI by navigating to http://localhost:8080 in your web browser. This user interface is a powerful tool that allows you to:

Monitor DAGs: See which DAGs are running, paused, or completed. Trigger DAGs: Manually start a DAG run if needed. View Task Details: Check the status, logs, and

execution history of each task. Manage DAGs: Enable or disable DAGs, and manage their schedules.

5. Validating the Output

After you've set up your DAGs and tasks, it's important to verify that they're working correctly. Once a DAG has run, you should check the output to ensure it meets your expectations.

For example:

If you have a data_preprocessing DAG that transforms raw data, you should examine the preprocessed file to make sure it contains the updated and cleaned data as intended.

Summary

Building an **MLOps pipeline** using **Apache Airflow** involves several key steps:

- **Initializing the Database:** Sets up the infrastructure for managing tasks and DAGs.
- **Starting the Webserver:** Provides a user-friendly interface to manage and monitor workflows.
- **Starting the Scheduler:** Automates task execution based on defined schedules.
- **Accessing the Airflow UI:** Enables interaction and control of workflows through a web interface.
- **Checking the Output:** Ensures workflow outputs are correct and meet expectations.

By automating tasks through **DAGs**, Airflow enhances **efficiency**, **scalability**, and **reproducibility**, streamlining data preprocessing, model training, and deployment to maintain and scale your MLOps pipeline.

About the Author

I'm Mauzum Shamil, a **Data Scientist** passionate about building scalable solutions for data-driven challenges. I enjoy exploring technologies like Apache Airflow, Python, SQL, and Machine Learning, focusing on enhancing efficiency and automation in workflows.

Feel free to connect with me or explore my work through the links below:

- **GitHub:** github.com/mauzumshamil
- **LinkedIn:** linkedin.com/in/mauzum-shamil
- **Portfolio:** linktr.ee/mauzum_shamil

Let's create, automate, and innovate together!