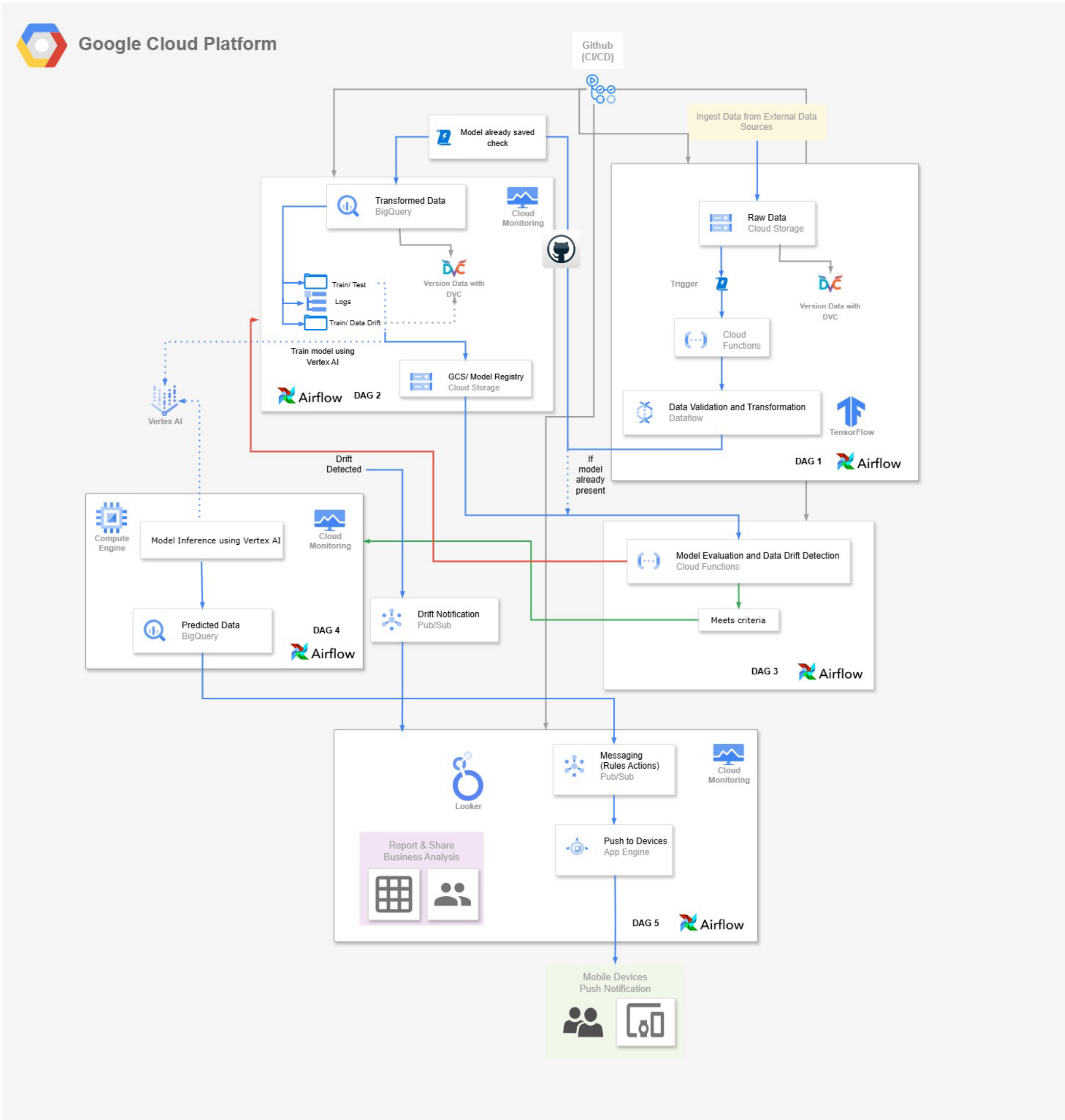
Comprehensive Data Pipeline Documentation for Telecom Churn Prediction and Device Upgrade Analysis

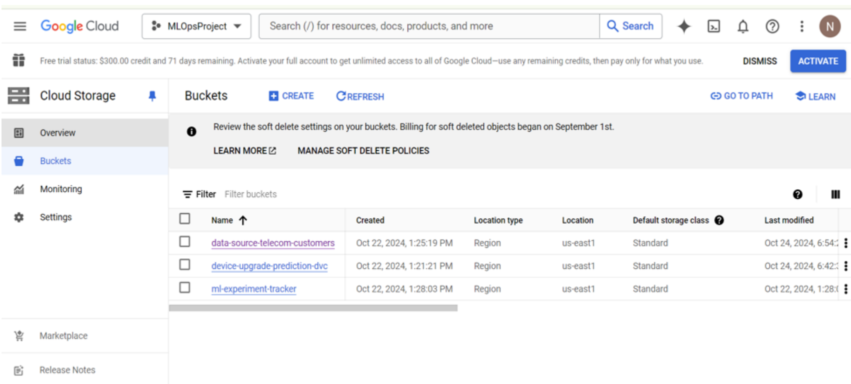


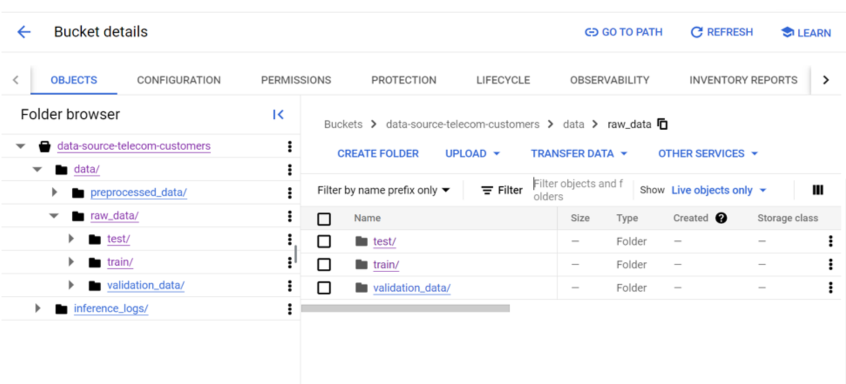
1. Ingesting data to GCS

Data Preparation: Split the raw data locally into train.csv and test.csv for training and validation, respectively. holdout.csv is divided into smaller batches (e.g., holdout_batch_1.csv) to enable batch processing and drift detection.

Upload to GCS: Upload the prepared datasets (train, test, and holdout batches) to designated GCS buckets, centralizing storage and ensuring accessibility for further processing and model training.

```bash
gs://data-source-telecom-customers/
    ├── raw_data/
    │   ├── train/
    │   │   └── train.csv                        # Raw training data
    │   ├── test/
    │   │   └── test.csv                         # Raw test data
    │   └── validation_data/
    │       ├── holdout_batch_1.csv              # Holdout batch 1
    │       ├── holdout_batch_2.csv              # Holdout batch 2
    │       ├── holdout_batch_3.csv              # Holdout batch 3
    │       └── holdout_batch_4.csv              # Holdout batch 4
    └── preprocessed_data/
        ├── train/
        │   └── train_preprocessed.csv           # Preprocessed training data
        ├── test/
        │   └── test_preprocessed.csv            # Preprocessed test data
        └── validation_data/
            ├── holdout_batch_1_preprocessed.csv # Preprocessed holdout batch 1
            ├── holdout_batch_2_preprocessed.csv # Preprocessed holdout batch 2
            ├── holdout_batch_3_preprocessed.csv # Preprocessed holdout batch 3
            └── holdout_batch_4_preprocessed.csv # Preprocessed holdout batch 4
```





2. Download Data from GCS to Local Machine

After storing data in the GCS bucket, it’s downloaded to the local environment for preprocessing. This step uses a script that fetches the raw train.csv, test.csv, and holdout data files directly from the GCS bucket to your local machine, readying the data for further cleaning and transformation.

download_data.py

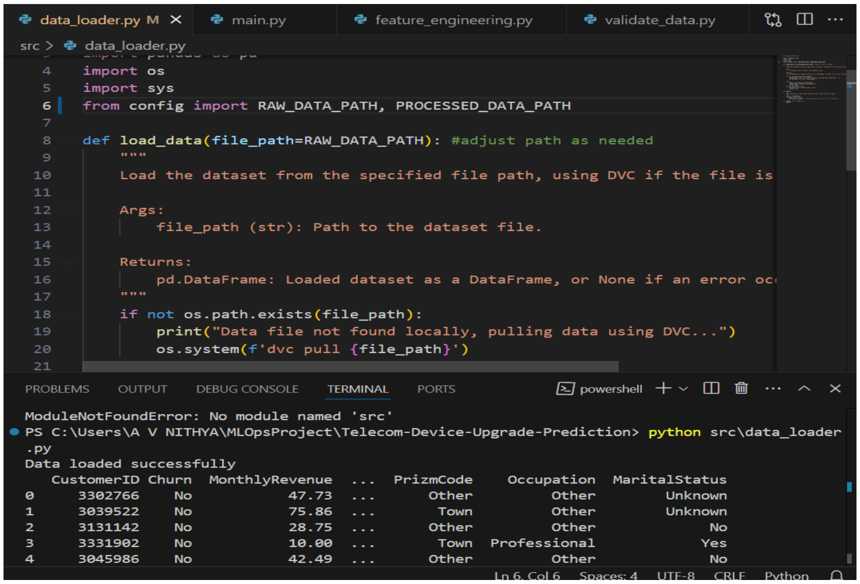
src > download_data.py > ...

```python
# %%
from google.cloud import storage
import os

# Define your GCS bucket and file paths
BUCKET_NAME = "data-source-telecom-customers"
TRAIN_FILE = "data/raw_data/train/train.csv"
TEST_FILE = "data/raw_data/test/test.csv"
HOLDOUT_DATASETS_PATH= "data/raw_data/validation_data"

# Define local folder paths
# Update these paths to match your local environment
LOCAL_TRAIN_PATH = "/Users/svs/Desktop/Projects/Telecom-Device-Upgrade-Prediction/data/raw/train.csv"
LOCAL_TEST_PATH = "/Users/svs/Desktop/Projects/Telecom-Device-Upgrade-Prediction/data/raw/test.csv"
LOCAL_HOLDOUT_PATH = "/Users/svs/Desktop/Projects/Telecom-Device-Upgrade-Prediction/data/raw"

HOLDOUT_DATASETS = ["holdout_batch_1.csv", "holdout_batch_2.csv", "holdout_batch_3.csv", "holdout_batch_4.csv"]

# Initialize the GCS client
client = storage.Client()
bucket = client.bucket(BUCKET_NAME)

def download_file_from_gcs(blob_name, local_path):
    """Downloads a file from GCS to a local directory."""
    blob = bucket.blob(blob_name)
    blob.download_to_filename(local_path)
    print(f"Downloaded {blob_name} to {local_path}")

# Download files from GCS to local environment
# Ensure the local directory exists
os.makedirs("/Users/svs/Desktop/Projects/Telecom-Device-Upgrade-Prediction/data/raw", exist_ok=True)
download_file_from_gcs(TRAIN_FILE, LOCAL_TRAIN_PATH)
download_file_from_gcs(TEST_FILE, LOCAL_TEST_PATH)

for dataset in HOLDOUT_DATASETS:
    HOLDOUT_GCS_PATH = HOLDOUT_DATASETS_PATH + "/" + dataset
    LOCAL_PATH = LOCAL_HOLDOUT_PATH + "/" + dataset
    download_file_from_gcs(HOLDOUT_GCS_PATH, LOCAL_PATH)
```

Note: This step is optional if you’re working with our repository, as it already includes the raw train.csv, test.csv, and holdout files from the GCS bucket. You can proceed directly to preprocessing if the files are already present locally.

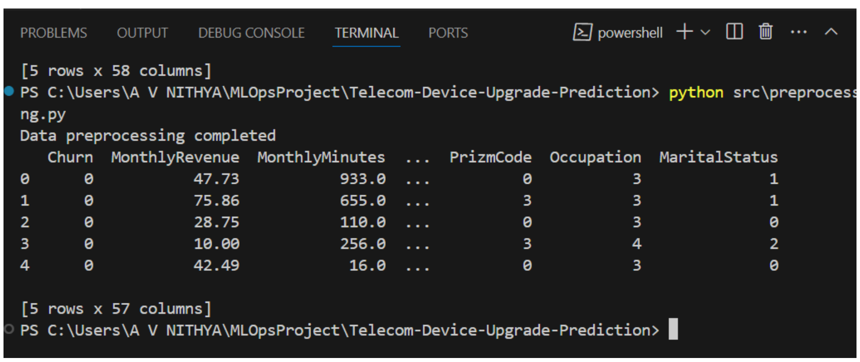
3. Data Loader

The data loader script (src/data_loader.py) is designed to retrieve datasets for processing. The script first checks if the raw data files (e.g., train.csv, test.csv) are available locally; if they are not found, they are automatically pulled from version control using DVC. Once available, the data is loaded into a panda DataFrame, preparing it for preprocessing in subsequent steps.



This approach ensures that the latest version of the dataset is always accessible, even if it is not stored locally, enabling a seamless transition to further stages in the pipeline.

4. Pre-processing Steps



In this step, preprocessing steps were performed to prepare the raw dataset for model training. The process was managed through a dedicated script, preprocessing.py, which includes several functions to handle data cleaning, encoding, and transformation:

- **Removal of Identifiers**: Columns with unique identifiers, such as customerID, were removed to protect sensitive information and retain only meaningful features for model training.
- **Handling Missing Values**: Missing values were imputed according to the category of the record (Churn or Non-Churn). Numerical columns were filled with median values, while categorical columns used the mode, ensuring that no essential information was lost.
- **Encoding Categorical Variables:**
  - Binary Variables: Used one-hot encoding and encoded as integers (e.g., "Yes" mapped to 1, "No" to 0).
  - Ordinal Variables: Mapped using predefined dictionaries (e.g., CreditRating, IncomeGroup) to retain their natural order.
  - Multi-class Variables: Label encoding was applied to make these variables suitable for model input.
- **Scaling of Numerical Columns**: Scaling was assessed but found unnecessary due to the use of tree-based models, which are not sensitive to feature scaling.

- **Testing Script: test_preprocessing.py:** To ensure each function in preprocessing.py worked as intended, a separate testing script, test_preprocessing.py, was developed using the unittest framework. Tests were created for each function, and specific issues, such as IntCastingNaNError, were addressed by adding .fillna() handling. This approach helped ensure compatibility and error-free execution across various data scenarios, reinforcing the robustness of the preprocessing pipeline.

5. Feature engineering

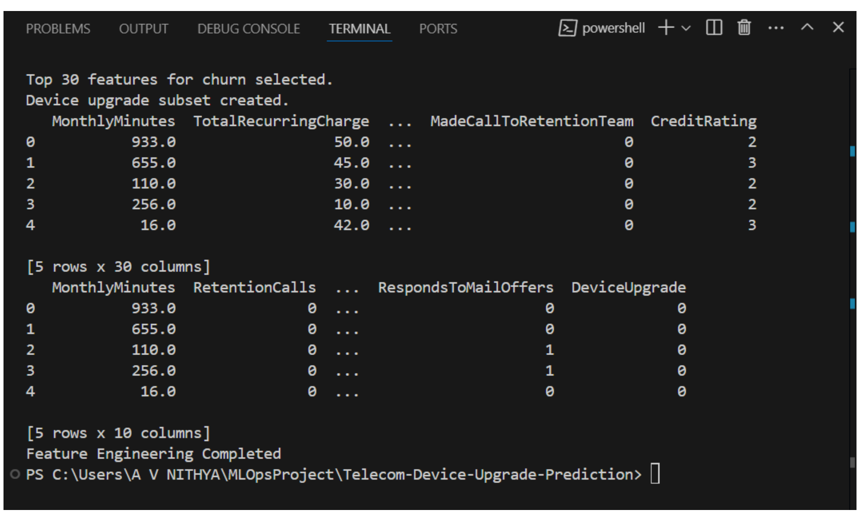
Extensive feature engineering was conducted to enhance model performance for predicting customer churn and identifying potential device upgrade candidates.

Feature Selection for Churn Prediction: Through experimentation using the SelectKBest method, it was determined that the optimal number of features (k) lies between 25 and 30 for customer churn prediction. Custom functions were developed to dynamically determine the best k within this range, selecting the most relevant features for the churn prediction model.

Creating a Device Upgrade Feature: In addition to churn prediction, an objective was to identify customers likely to upgrade their devices. Since the dataset lacked a direct indicator for device upgrades, a custom DeviceUpgrade feature was created using the following criteria:

- MonthlyMinutes > 3000
- RetentionCalls > 2
- RetentionOffersAccepted > 0
- HandsetWebCapable == 0
- HandsetRefurbished == 1
- CurrentEquipmentDays > 340
- CreditRating > 5
- MadeCallToRetentionTeam == 1
- RespondsToMailOffers == 1

These thresholds were defined based on domain-specific research, expert knowledge, and literature review, ensuring that the criteria were grounded in relevant insights.



6. Testing and Debugging

Unit Testing:

- Comprehensive tests were conducted using unittest and pytest to validate the functionality of the preprocessing.py and feature_engineering.py scripts.
- Errors encountered during testing, such as IntCastingNaNError, were resolved by implementing robust handling of missing values before typecasting.

Debugging:

- Issues with .astype() errors, particularly when converting columns with NaN values to integers, were addressed by adding appropriate checks and handling methods.
- Warnings from libraries were managed to ensure that all functions remain compatible with future library updates, reinforcing the reliability and stability of the codebase.

This rigorous testing and debugging process ensured that both preprocessing and feature engineering workflows were robust, accurate, and compatible with current and future data scenarios.

7. Airflow DAG

1. Configuring Airflow

1.1 Initialize the Airflow Database

```
airflow db init
```

1.2 Start the Airflow Web Server and Scheduler

- Start the web server to access the Airflow UI:
  ```
  airflow webserver --port 8080
  ```
- In a separate terminal, start the scheduler:
  ```
  airflow scheduler
  ```

1.3 Accessing the Airflow UI

- Open a browser and go to http://localhost:8080.
- Log in with default credentials or create an admin user if needed.

2. DAG Creation: telecom_pipeline

The telecom_pipeline DAG automates data loading, preprocessing, and feature engineering for telecom data to predict device upgrades. The DAG includes three primary tasks:

2.1 **Setting Up the DAG**

- Place the DAG script (telecom_pipeline.py) in the $AIRFLOW_HOME/dags directory.
- The DAG consists of three tasks:

  Task 1: Load Data - Loads raw data using load_data from data_loader.py and saves it temporarily.

  Task 2: Preprocess Data - Preprocesses the loaded data using preprocess_data, saving the output for further analysis.

  Task 3: Feature Engineering - Performs feature engineering to select key features for churn and device upgrade prediction.

2.2 **Task Overview**

- **load_data_task**: Loads raw data and saves it to /tmp/loaded_data.csv.

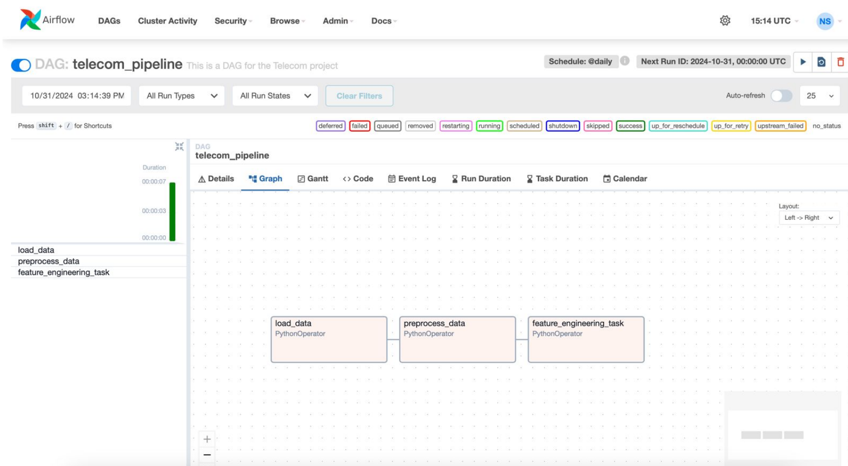
- **preprocess_data_task**: Reads, preprocesses, and saves data to /tmp/preprocessed_data.csv.
- **feature_engineering_task**: Reads preprocessed data, applies feature selection for churn prediction and device upgrade analysis, and saves results for further steps.

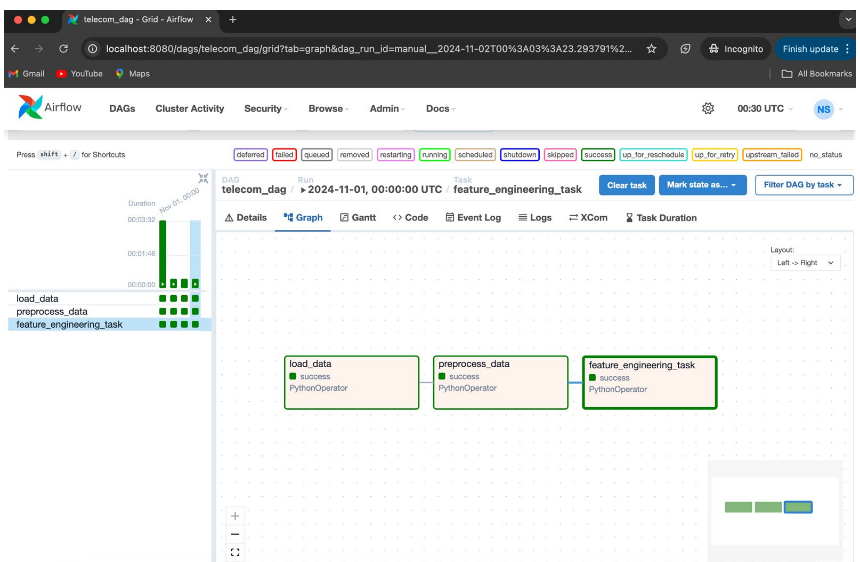
2.3 **Running the DAG**

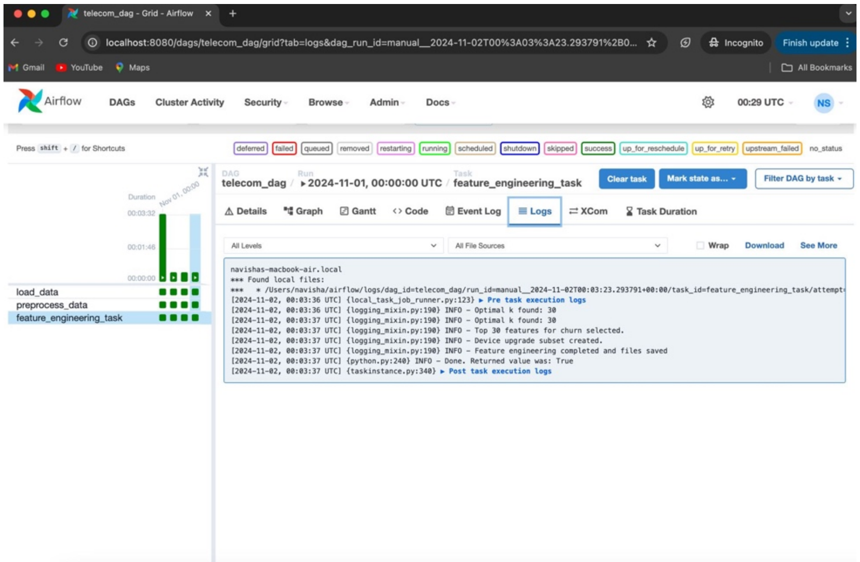
- After defining the DAG, access the Airflow UI (http://localhost:8080) and manually trigger the DAG to confirm functionality.

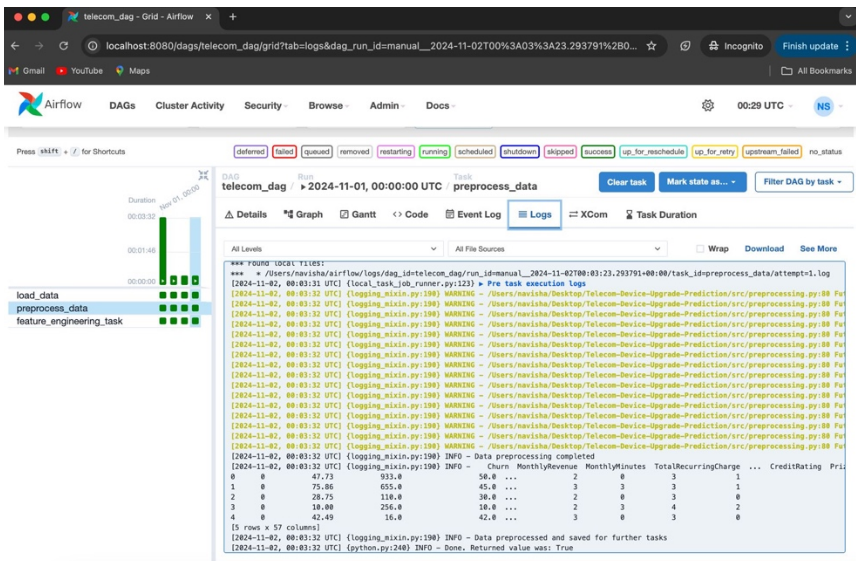
3. **Summary**

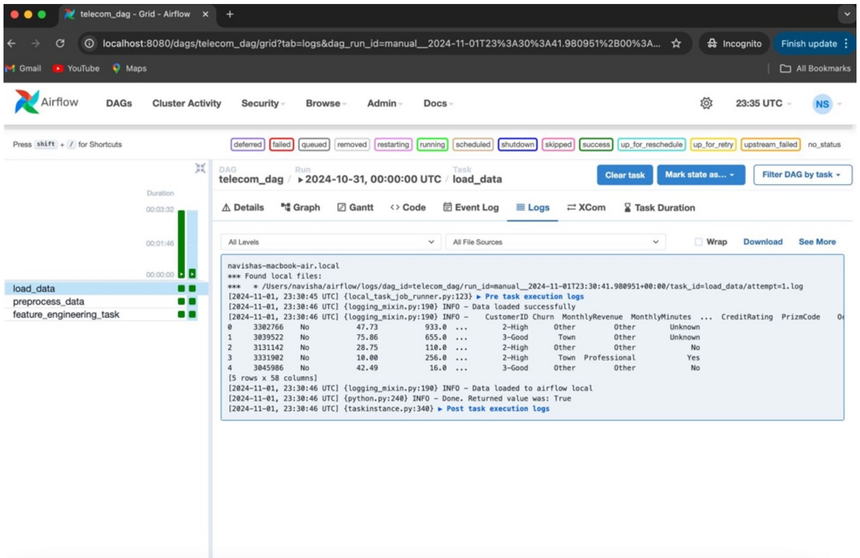
The Airflow DAG (telecom_pipeline) automates the workflow for loading, preprocessing, and engineering features, forming a foundation for more complex telecom data processing tasks. This setup facilitates reliable, repeatable data processing in Airflow.

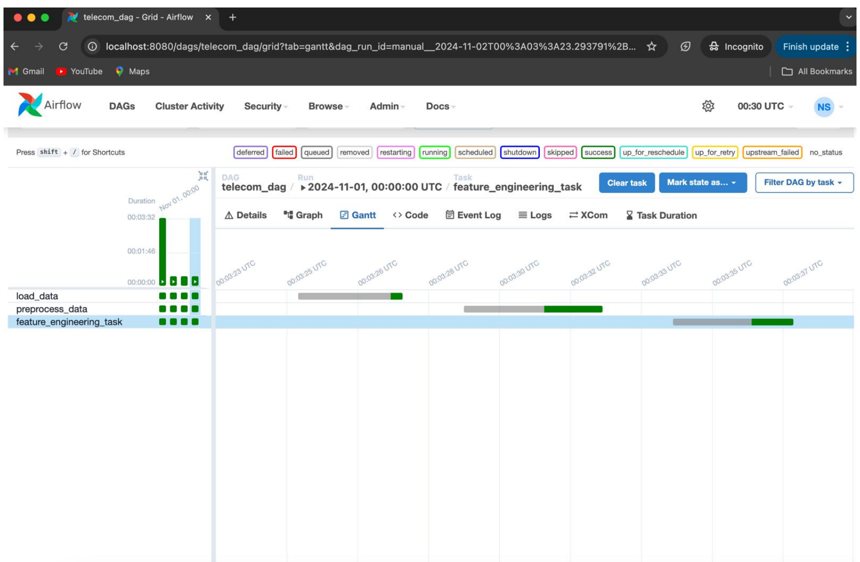





telecom_dag - Grid - Airflow
Airflow
DAGs
Cluster Activity
Security
Browse
Admin
Docs
00:29 UTC
telecom_dag
2024-11-01, 00:00:00 UTC
feature_engineering_task
Clear task
Mark state as...
Filter DAG by task
Details
Graph
Gantt
Code
Event Log
Logs
XCom
Task Duration
load_data
preprocess_data
feature_engineering_task
navishas-macbook-air.local
*** Found local files:
[2024-11-02, 00:03:36 UTC] {logging_mixin.py:190} INFO - Optimal k found: 30
[2024-11-02, 00:03:37 UTC] {logging_mixin.py:190} INFO - Optimal k found: 30
[2024-11-02, 00:03:37 UTC] {logging_mixin.py:190} INFO - Top 30 features for churn selected.
[2024-11-02, 00:03:37 UTC] {logging_mixin.py:190} INFO - Device upgrade subset created.
[2024-11-02, 00:03:37 UTC] {logging_mixin.py:190} INFO - Feature engineering completed and files saved
[2024-11-02, 00:03:37 UTC] {python.py:240} INFO - Done. Returned value was: True


telecom_dag - Grid - Airflow
Airflow
DAGs
Cluster Activity
Security
Browse
Admin
Docs
00:29 UTC
telecom_dag
2024-11-01, 00:00:00 UTC
preprocess_data
Clear task
Mark state as...
Filter DAG by task
Details
Graph
Gantt
Code
Event Log
Logs
XCom
Task Duration
load_data
preprocess_data
feature_engineering_task
[2024-11-02, 00:03:32 UTC] {logging_mixin.py:190} INFO - Data preprocessing completed
[5 rows x 57 columns]
[2024-11-02, 00:03:32 UTC] {logging_mixin.py:190} INFO - Data preprocessed and saved for further tasks
[2024-11-02, 00:03:32 UTC] {python.py:240} INFO - Done. Returned value was: True





8. Schema Generation

Schema generation and validation are set up using TensorFlow Data Validation (TFDV), ML Metadata (MLMD), and Airflow for automation.

- **Dependencies**: Install required libraries (tensorflow, tensorflow-data-validation, apache-airflow, ml-metadata) listed in requirements.txt.
- **Schema Generation**: generate_schema.py uses TFDV to create a schema from the dataset, saved as schema.pbtxt.
- **Data Validation**: validate_data.py checks new data against the schema, identifying anomalies (e.g., missing values or data type issues).
- **Logging with MLMD**: Schema versions and validation runs are logged in MLMD for tracking changes and ensuring consistency.
- **Automation with Airflow**: An Airflow DAG schedules daily schema generation and validation, automating data quality checks.
- **Alerts:** Notifications are added to alert the team to validation failures.

```
feature {
  name: "CustomerID"
  type: INT
  presence {
    min_fraction: 1.0
    min_count: 1
  }
  shape {
    dim {
      size: 1
    }
  }
}
feature {
  name: "Churn"
  type: BYTES
  domain: "Churn"
  presence {
    min_fraction: 1.0
    min_count: 1
  }
  shape {
    dim {
      size: 1
    }
  }
}
feature {
  name: "MonthlyRevenue"
  value_count {
    min: 1
    max: 1
```

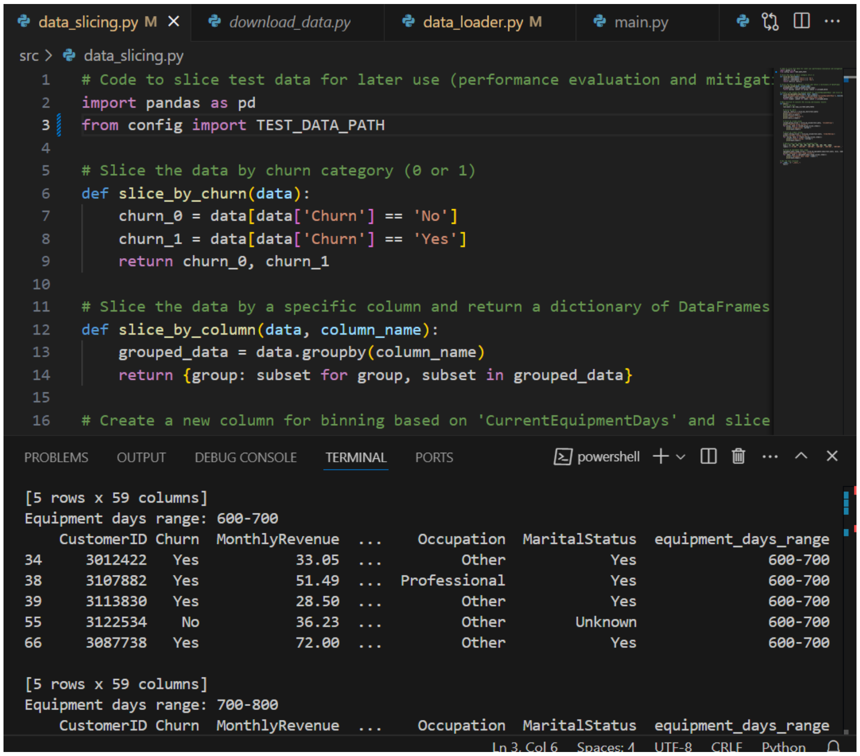
data/schema/schema.pbtxt

9. Data slicing

Data slicing is performed on the test dataset to create targeted subsets, enabling detailed evaluation of model performance and potential bias mitigation. This process involves segmenting the data based on specific variables that are relevant to our analysis, specifically Churn, IncomeGroup, and CreditRating.

- **Churn-Based Slicing**: The dataset is first divided into two subsets based on the Churn column, creating separate DataFrames for customers who have churned (Churn = Yes) and those who have not (Churn = No). This allows for targeted analysis of model performance for each group, ensuring fair representation.
- **Income Group and Credit Rating Slicing**: Further slicing is performed on IncomeGroup and CreditRating. DataFrames are generated for each unique value in these columns, allowing for group-based analysis. This helps assess how the model performs across different income levels and credit ratings, identifying potential biases in predictions related to these variables.
- **Equipment Days Range**: A new column, equipment_days_range, is created by binning CurrentEquipmentDays into intervals (e.g., 0-100, 100-200, etc.). These bins provide an additional segmentation based on how long customers have had their current equipment, giving insight into performance variations for customers at different stages of their device lifecycle.

These slices create distinct, manageable views of the dataset, facilitating a detailed evaluation of model performance and helping identify any biases across different customer segments.


data_slicing.py M
download_data.py
data_loader.py M
main.py
src > data_slicing.py
# Code to slice test data for later use (performance evaluation and mitigat
import pandas as pd
from config import TEST_DATA_PATH
# Slice the data by churn category (0 or 1)
def slice_by_churn(data):
churn_0 = data[data['Churn'] == 'No']
churn_1 = data[data['Churn'] == 'Yes']
return churn_0, churn_1
# Slice the data by a specific column and return a dictionary of DataFrames
def slice_by_column(data, column_name):
grouped_data = data.groupby(column_name)
return {group: subset for group, subset in grouped_data}
Create a new column for binning based on 'CurrentEquipmentDays' and slice
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
powershell
[5 rows x 59 columns]
Equipment days range: 600-700
CustomerID Churn MonthlyRevenue ... Occupation MaritalStatus equipment_days_range
34 3012422 Yes 33.05 ... Other Yes 600-700
38 3107882 Yes 51.49 ... Professional Yes 600-700
39 3113830 Yes 28.50 ... Other Yes 600-700
55 3122534 No 36.23 ... Other Unknown 600-700
66 3087738 Yes 72.00 ... Other Yes 600-700
[5 rows x 59 columns]
Equipment days range: 700-800
CustomerID Churn MonthlyRevenue ... Occupation MaritalStatus equipment_days_range
Ln 3, Col 6 Spaces: 4 UTF-8 CRLF Python