Code Repo - mlops-9 (main branch - check README.md for more information)

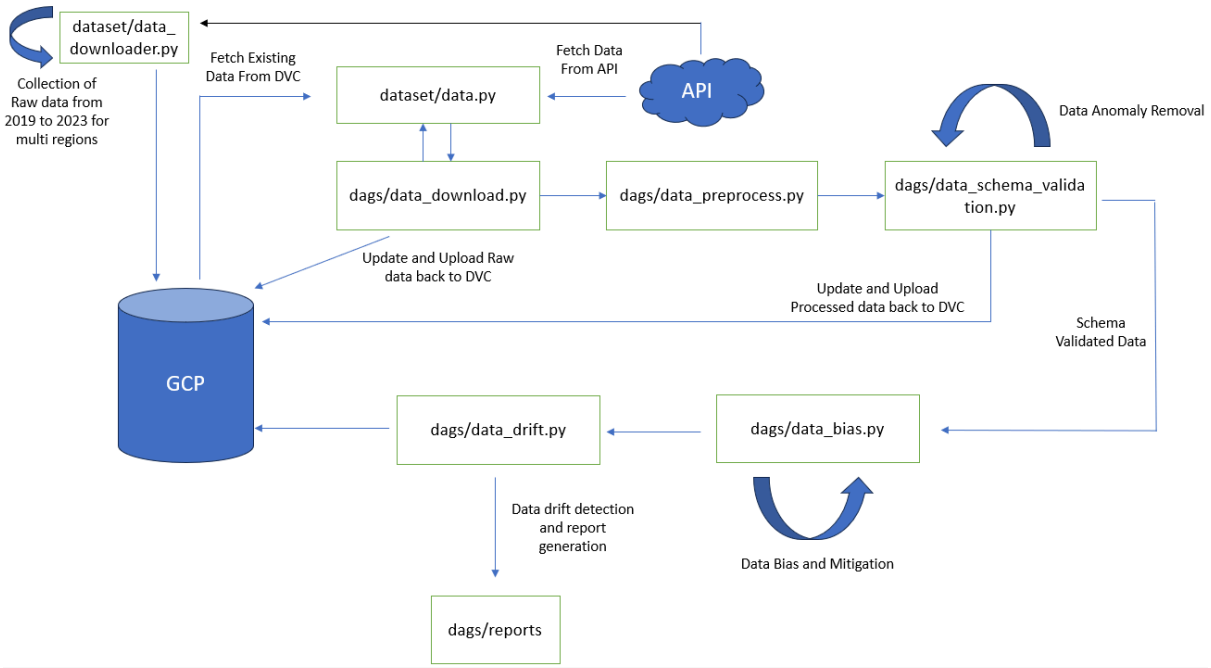
Introduction

In this project, we're building a practical electricity demand forecasting model that can provide short-term consumption predictions to help with grid stability and energy management. The model takes in dynamic inputs like real-time weather data (temperature, wind speed, humidity), population density, and time-based features to predict electricity demand for specific locations across the U.S. By accounting for the unique weather patterns and power consumption habits of different cities, our goal is to offer utility companies and grid operators a tool to anticipate demand fluctuations. This will enable them to better manage resources and keep the energy supply steady.

The data pipeline for this project is designed to be as automated and reliable as possible, with a focus on keeping the data clean, consistent, and versioned for easy tracking. We're using APIs to pull in real-time weather and population data, which Airflow then orchestrates, handling both the data collection and model training tasks.

At the core of everything is our data pipeline, which is responsible for moving data smoothly from ingestion to model prediction. We're pulling in data from multiple sources, including real-time weather and population density via APIs, and each dataset goes through steps like cleaning, transforming, and validating to keep things accurate and consistent. To capture patterns over time, we also create features like rolling averages and lagged weather variables. Airflow handles scheduling and organizing these tasks to keep things running automatically. The pipeline is designed with versioning, so we can easily trace predictions back to their data sources and retrain the model as needed. This setup forms a solid foundation, ensuring that high-quality data flows through the system and into the model for accurate predictions.

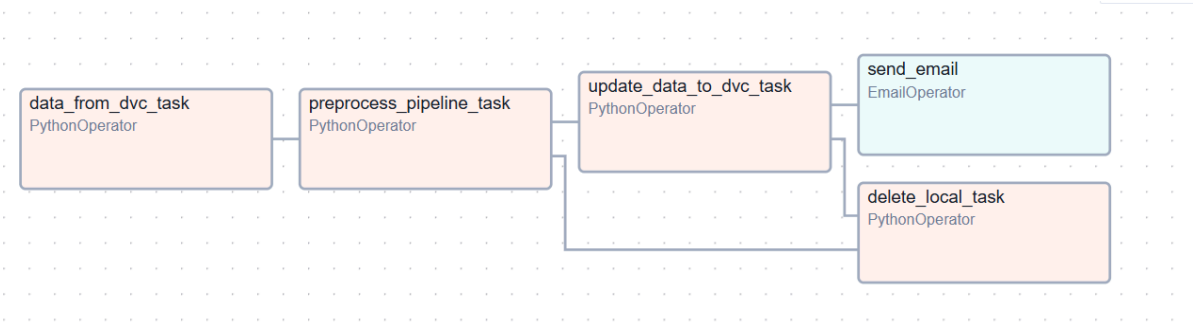
Data Pipeline



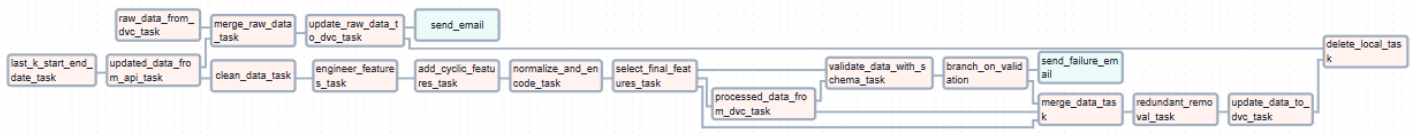
Overview

There are 4 DAG flows which we have implemented here as part of the data pipeline. Each of them is discussed in more detail in the other subheading -

1) **Raw data to preprocess data DAG**
   This DAG takes the raw data and applies the preprocessing, then saves it to DVC

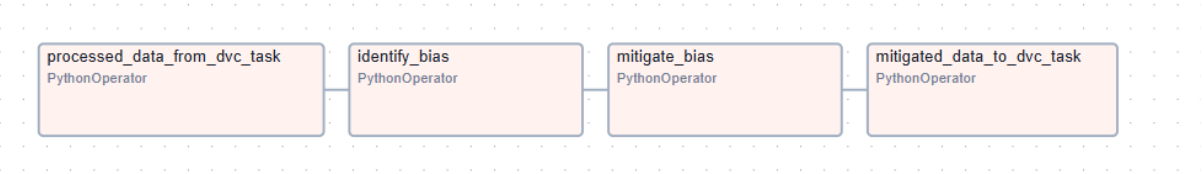


2) **Extracting New data from API flow DAG**
   This DAG flow gets the new data from API, applies preprocessing on it, identifies anomalies and fixes anomalies, merges new data with existing data and upload to DVC.

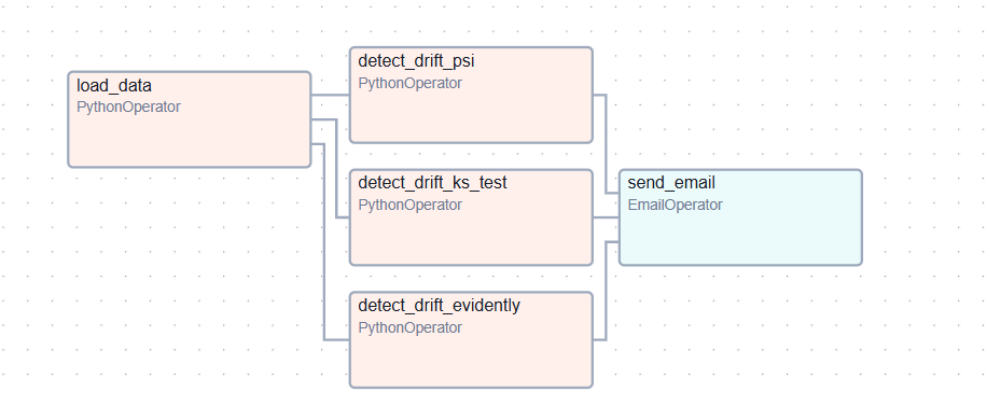


3) **Bias detection and mitigation DAG**

This DAG detects and mitigates bias on preprocessed data



4) **Data drift detection DAG**
This DAG detects any drift on the preprocessed data



1. Dataset

- The dataset comprises hourly weather data, population density, and electricity demand from multiple U.S. regions over the past five years. The primary goal is to predict short-term electricity consumption based on real-time and historical weather conditions, population, and other external factors. By building a comprehensive dataset, this project aims to model the correlation between these factors and electricity demand, helping to optimize grid management.
- Size: Estimated ~40,000 data points per region, with hourly data for 5 years across multiple cities.
- Format: CSV (processed from API responses)

- <u>Features</u>
    - Weather: Hourly data including temperature, humidity, wind speed, pressure, etc.
    - Population Density: High-resolution data (~1 km grid) for population density.
    - Location: latitude, longitude.
    - Time: Hourly timestamps over a 5-year period.

    - Label: Electricity demand (measured in MWh).

- Data Types
    - Numerical: Temperature, wind speed, population density, electricity demand.
    - Categorical: Location Zones.

- Data Sources
    - **Weather Data**: World Weather Online API offers detailed weather data for different cities, including temperature, wind speed, humidity, etc., on an hourly basis (World Weather Online).
    - **Electricity Demand Data**: EIA API provides access to real-time and historical electricity consumption data for different regions across the U.S. We are focusing on Texas - ERCO (ERCOT - Electric Reliability Council of Texas) - which has 8 regions within it covering the entire Texas state. This can be expanded to other regions and states.
    - **Population Density Data**: WorldPop API delivers population density data at high resolution (1 km grid), adjusted to match UN estimates, allowing us to account for population factors affecting electricity consumption (WorldPop).

2. Dataset Download Process

- The dataset download process is designed to collect essential data from external sources, including APIs for demand and weather information. This process operates on a configurable daily schedule, with each cycle fetching hourly data across multiple regions, such as Texas, New York, and New England.

- The primary data acquisition functionality is implemented in **dataset/scripts/data.py**, while the script **dataset/scripts/data_download.py** manages the download process. Additionally, historical data for the period from June 1, 2019, to December 31, 2023, for all three regions is downloaded using the Jupyter Notebook **experiments/data_downloader.ipynb**.

- Data Structure - The raw data includes columns such as datetime, tempF, windspeedMiles, weatherCode, and humidity, along with region and zone identifiers to enable efficient segmentation and analysis.

- The collected data is saved as **data/dataset_raw.csv** temporarily and pushed to DVC as the raw dataset. This dataset is then further processed using raw data to preprocess data DAG flow, while a separate DAG flow ensures the raw dataset is continuously updated with new daily API data.

3. Data Preprocessing

Here's a more detailed breakdown of each preprocessing step:

1. Data Cleaning

- Missing Value Imputation: Identifies and handles missing values in a flexible way. For example, missing numerical values can be filled with the mean, median, or a custom value.

- Duplicate Removal: Detects and removes duplicate entries to ensure data integrity and reduce redundancy. Duplication issues can often arise from merged datasets or multiple records for the same entity. This process is especially crucial in time series data, where multiple entries for the same time step can disrupt modeling efforts.

2. Feature Engineering

- Rolling Features: Adds rolling (moving average or sum) features to capture recent trends within a specified window size (e.g., 7 days, 30 days). Rolling statistics can smooth noise and highlight seasonal effects.

- Lagged Features: Introduces lagged versions of variables to capture delayed effects. For example, a 1-day lag feature captures values from the previous day, helping to model the dependency of today's values on past values. Lagged features are critical for capturing autocorrelations in time series.

3. Cyclic Features

- Sine and Cosine Encoding for Cyclic Data: Transforms cyclic temporal features (such as months, days, or hours) into two dimensions using sine and cosine functions. This transformation retains the cyclic nature of the data (e.g., December and January are close in time). Cyclic encoding is especially valuable for periodic data like seasons, times of day, and day-of-week effects.

- Handling Multiple Cyclic Patterns: If the dataset includes multiple cyclic features (e.g., daily and yearly cycles), each cyclic feature can be transformed separately. For

example, both day-of-year and time-of-day can be encoded, capturing seasonality as well as hourly patterns.

4. Normalization and Encoding

- Normalization of Numerical Features: Applies normalization techniques (e.g., Min-Max Scaling, Z-score Standardization) to ensure numerical features are on a similar scale, preventing certain features from dominating due to their scale. The choice of normalization method may depend on the model type; for instance, neural networks often benefit from Min-Max scaling.

- Categorical Encoding: Encodes categorical variables using techniques like One-Hot Encoding or Target Encoding. One-Hot Encoding is useful for nominal categories, while Target Encoding or Frequency Encoding can help with high-cardinality categorical data, where traditional one-hot encoding would lead to many sparse columns.

5. Feature Selection

- Correlation Analysis: Uses correlation analysis (e.g., Pearson, Spearman) to identify and drop features that are highly correlated and could introduce redundancy, thus simplifying the model without losing information.

- Feature Importance Analysis: Applies model-based techniques, such as using feature importances from tree-based models (Random Forests, Gradient Boosting), to rank features. Features with little contribution to the model's performance can be removed.

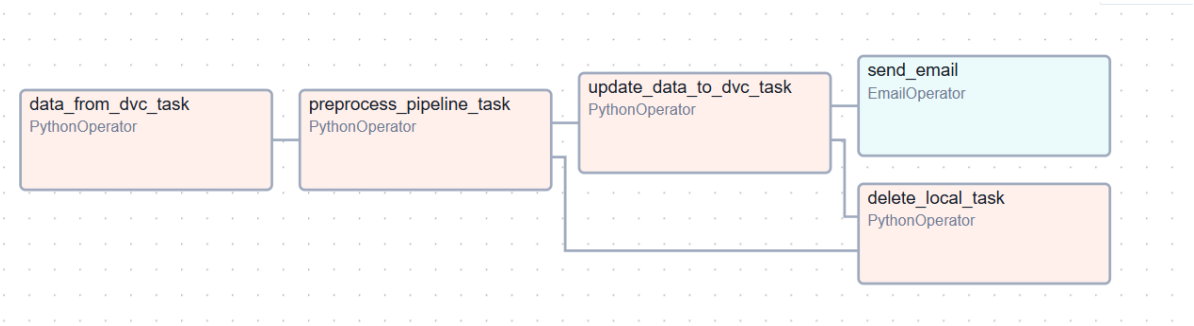
- Dimensionality Reduction: Optional dimensionality reduction techniques like Principal Component Analysis (PCA) or Factor Analysis can be applied for high-dimensional datasets, reducing noise and capturing the essential data structure in fewer features.

6. Modular and Reusable Code Structure

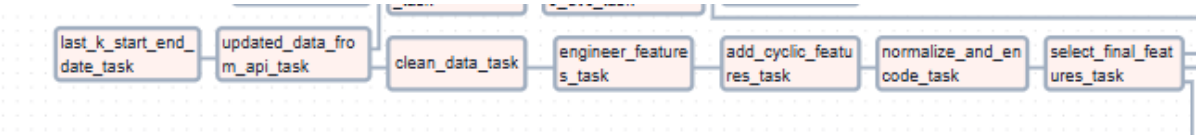
- Configurable Parameters: Each preprocessing step is parameterized (e.g., window size for rolling features, normalization methods) so that users can easily adjust settings based on dataset characteristics.

- Pipeline Structure: Code is organized into a pipeline or modular functions, enabling users to add or remove specific steps without breaking the flow. For example, a `Pipeline` in scikit-learn can chain steps sequentially, allowing seamless transformation, encoding, and normalization.

- Scalability: The preprocessing code is designed to handle various data sizes and types.



Update Raw data preprocess DAG Flow



New Data Preprocess DAG Flow

4. Data Schema Validation

- To ensure the consistency and reliability of the dataset, this DAG integrates a data validation step using the **DataSchemaAndStatistics** class. The schema defines the expected structure of the data, including column names, data types, and constraints. This schema is either inferred from an initial dataset or loaded from a predefined schema file, ensuring that new data adheres to the established format (**Pandera** library is used)

- The DataSchemaAndStatistics class **(dataset/scripts/data_schema.py)** uses pandera for schema management and validation. The infer_schema() method automatically infers the schema from an initial dataset, capturing essential characteristics like data types and column structure.

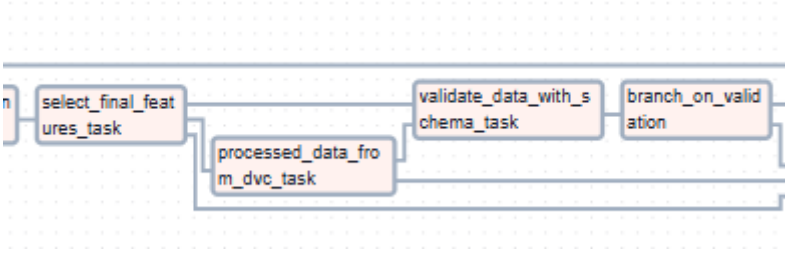
- Once inferred, this schema is stored in JSON format, making it easy to reload and validate against future datasets. The validate_data() method checks new data against the inferred schema, logging any discrepancies. A successful validation returns a 1, while failures return a 0 and log the validation errors. This validation process ensures that only clean and consistent data is processed and saved, enhancing the reliability of the data pipeline.

- If the new data does not meet the schema validation and anomalies are found, anomaly resolution takes place (as seen in **dataset/scripts/data_schema.py)**.

- The fix_anomalies function is designed to identify and correct common data anomalies within a DataFrame. Here's a breakdown of the anomalies it resolves:

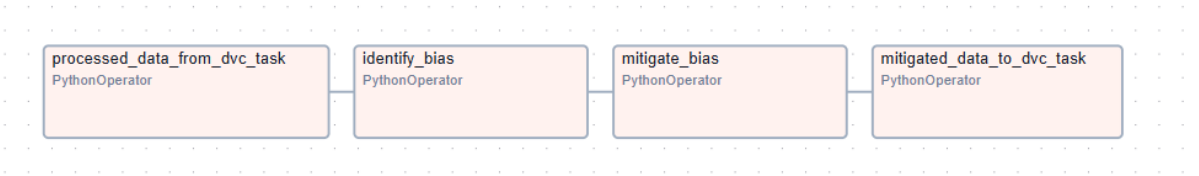
- **Missing Values**: Any missing values (NaNs) within the DataFrame are replaced with 0 to ensure there are no gaps in the dataset.
- **Duplicate Rows**: Duplicate entries in the DataFrame are removed to maintain data integrity and prevent redundancy.
- **Negative Values**: For certain columns, such as precipMM, visibility, HeatIndexF, windspeedMiles, humidity_rolling_mean, and others, negative values are not valid. The function checks each specified column and replaces any negative values with 0 to ensure logical consistency (e.g., temperature, wind speed, and pressure cannot be negative).

- When new data is ingested, it undergoes schema validation. If the data fails this validation due to any detected anomalies, such as missing values, duplicates, or negative values, the fix_anomalies function steps in to automatically resolve these issues. The process is logged for traceability.



Data schema validation Dag Flow

5. Data Bias and Mitigation



Bias Identification and Mitigation by Data Slicing

- The electricity data being collected is for multiple sub-regions in multiple locations. Due to this, data bias might be introduced into the dataset concerning the **subba-names feature.** Allowing this bias to be introduced into the model might cause the model to produce incorrect and biased predictions, reducing the application's overall performance.

- To make sure the model is fed unbiased data, a data bias identification and mitigation with Data Slicing dag is introduced. This process incorporates 4 steps/tasks:-

- **Processed_data_from_dvc_task:** this task is to fetch the latest preprocessed data set from the DVC.

- **Identify_bias:** This task is to identify data bias in the dataset by applying a data-slicing approach. The data is sliced on the **subba-name** feature, as it is the only categorical feature in the dataset.

  For the bias identification task, we implemented the **Metric Frame** approach to assess the completeness of the data itself. This helps to understand how well different subgroups are represented in the dataset (in our case, it is **subba-name**).

  Metric Frame is a class from the **Fairlearn** library that allows the calculation of metrics across different subgroups by sensitive feature(**subba-name**). It calculates the metrics(selection rate, accuracy, etc) for each subgroup and provides a breakdown analysis of the metrics.

  The **selection rate metric** is the proportion of positive outcomes for a given group/subgroup. It helps us identify if different subgroups are equally represented in the dataset. If a particular subgroup has a lesser selection rate, then it indicates a possible bias in the dataset.

  The **Accuracy** metric provides an overview of how well the model performs on each subgroup. It can help us identify if the model's performance for each subgroup is consistent. If a particular subgroup has a lesser accuracy than others, there might be a possible bias in the dataset.

  The Metric Frame returns 3 types of outputs:

  1. Metric_frame.by_group: calculated metrics for each subgroup.

```
By Group Metric Analysis
                                Selection Rate  Accuracy
subba-name
Capital - NYIS                             1.0       1.0
Central - NYIS                             1.0       1.0
Dunwoodie - NYIS                           1.0       1.0
ERCO - Coast                               1.0       1.0
ERCO - East                                1.0       1.0
ERCO - Far West                            1.0       1.0
ERCO - North                               1.0       1.0
ERCO - North Central                       1.0       1.0
ERCO - South                               1.0       1.0
ERCO - South Central                       1.0       1.0
ERCO - West                                1.0       1.0
Genesee - NYIS                             1.0       1.0
Hudson Valley - NYIS                       1.0       1.0
ISNE - Connecticut                         1.0       1.0
ISNE - Maine                               1.0       1.0
ISNE - New Hampshire                       1.0       1.0
ISNE - Northeast Mass.                     1.0       1.0
ISNE - Rhode Island                        1.0       1.0
ISNE - Southeast Mass.                     1.0       1.0
ISNE - Vermont                             1.0       1.0
ISNE - Western/Central Mass.               1.0       1.0
Long Island - NYIS                         1.0       1.0
Millwood - NYIS                            1.0       1.0
Mohawk Valley - NYIS                       1.0       1.0
New York City - NYIS                       1.0       1.0
North - NYIS                               1.0       1.0
West - NYIS                                1.0       1.0
```

2. Overall Metrics: overall metrics across the whole dataset without considering the subgroups providing a holistic view of the data as a whole

```
Overall Metric Analysis

 Selection Rate    1.0
Accuracy          1.0
dtype: float64
```

3. Differences Between Groups / Demographic Parity Difference: maximum difference between the best and worst performing subgroups for each metric, indicating if a potential bias exists or not.

```
Demographic Parity Difference between groups

 Selection Rate    0.0
Accuracy          0.0
dtype: float64
```

To see which subgroup/subba-name has potential bias, we check the absolute difference between the selection rate of the subgroup with the overall selection rate of the dataset. If the difference is more than a threshold of 0.05 (set based on our problem statement and requirement), we handle it with a mitigation strategy to remove the bias.

- **Mitigate_bias:** In this task, we aim to mitigate bias by performing mitigation strategies. Primarily, the bias we noticed in the dataset is caused by the unequal distribution of entries for each subgroup. To handle it, we used resampling strategies to tackle it. As of now, we employed a basic resampling strategy but further down the process, more efficient resampling strategies and model-based mitigation approaches will be used.

- **Mitigated_data_to_dvc_task:** This is the final task, where, the resampled data with minimum to no bias, is written to DVC for versioning before it is used to train the model.

Bias Detection using trained model

The primary goal is to evaluate if a machine learning model provides consistent performance across various segments, such as geographical regions or environmental conditions. It achieves this by comparing model accuracy and error rates for each segment, highlighting any disparities. For instance, if the model is predicting something like energy usage or demand, it might systematically perform better in certain areas ("zones") or under specific weather conditions ("cloud cover"). Detecting these patterns helps identify biases, enabling more responsible and transparent use of the model.

This tool works by analyzing model performance across specific "sensitive features" in the data, such as geographical zones or environmental conditions. Here we have used three categorical columns as sensitive features: **zone**, **subba-name** and **cloudcover**

- The process begins by preparing and splitting the data into training and testing sets. After making predictions on the test data, the tool uses these sensitive features to define distinct groups within the dataset, enabling analysis across various segments, such as "high" vs. "low" cloud cover or different geographical regions.

- **Fairlearn**, a fairness-focused library, is crucial in this process. It provides data slicing capabilities that make it easy to evaluate performance for each sensitive group, ensuring an accurate and structured comparison.

- Through Fairlearn's **'MetricFrame'**, the 'ModelBiasDetector' calculates specific performance metrics (like **Mean Squared Error** and **Mean Absolute Error**) for each group independently. This breakdown reveals any significant disparities in accuracy or error rates, highlighting areas where the model may underperform for certain groups, which could indicate bias.

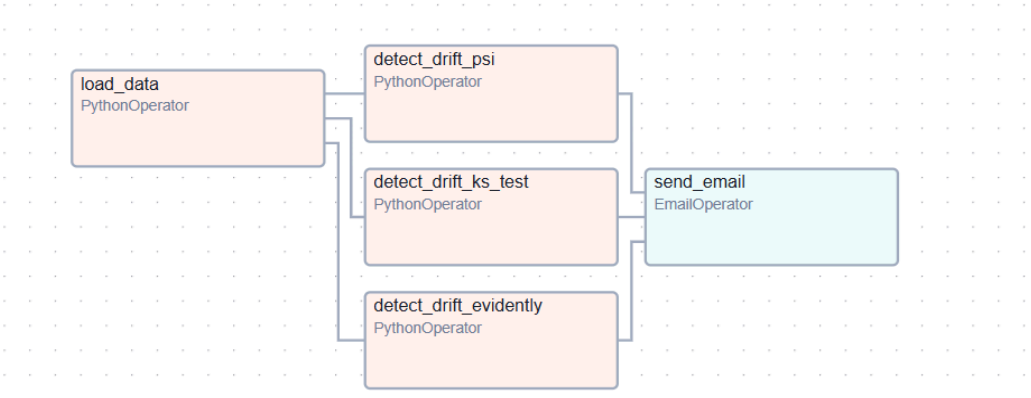
```
Sensitive Features Data (First 5 rows):
subba-name: 155810      ISNE - Connecticut
143027     ISNE - Rhode Island
724741    Mohawk Valley - NYIS
813756            ERCO - South
136242    ISNE - New Hampshire
Name: subba-name, dtype: object
cloudcover_high_low: ['low' 'high' 'high' 'high' 'low']
```

subba-name	cloudcover_high_low	Mean Squared Error	Mean Absolute Error
Capital - NYIS	**high**	0.000325	0.009357
	low	0.000288	0.008707
Central - NYIS	**high**	0.000310	0.008924
	low	0.000345	0.009713
Dunwoodie - NYIS	**high**	0.000308	0.009116
	low	0.000292	0.009000
ERCO - Coast	**high**	0.000960	0.020303
	low	0.000753	0.017704
ERCO - East	**high**	0.000792	0.018074
	low	0.000705	0.016700

By leveraging Fairlearn, the ‘ModelBiasDetector’ offers a transparent and systematic approach to bias detection, allowing organizations to make informed adjustments to improve model fairness across diverse populations or conditions.

Outcome
The results give a breakdown of the model's performance by sensitive group, highlighting any discrepancies in error rates or prediction accuracy. This information can then be used to refine the model, making it fairer across different segments and enhancing its reliability in diverse contexts.

6. Data Drift Detection



Data Drift Detection DAG Flow

To ensure our model maintains accuracy and relevance over time, we implement a Data Drift Detection module that identifies changes in the underlying data distribution between a baseline dataset and newly collected data. Data drift, if left unchecked, can degrade model performance as it indicates that the patterns in the new data deviate from those in the baseline data used to train the model. By monitoring data drift, we proactively address this issue and make necessary adjustments to the model.

Our data drift detection process consists of multiple statistical tests, leveraging both Evidently AI and statistical metrics like the Kolmogorov-Smirnov (KS) test and Population Stability Index (PSI). Here's an overview of each method and its role in detecting drift:

1. **Evidently AI Data Drift Detection:**

Evidently AI's DataDriftPreset metric provides a comprehensive report on data drift across multiple features, both numerical and categorical. We use this library to generate a detailed HTML report and JSON output summarizing drift results for each feature.

Implementation:

- Define a ColumnMapping to specify which features are numerical and which are categorical.
- Run Evidently's DataDriftPreset report on the baseline and new datasets.
- Generate an HTML report for visualization and save it to a specified path for detailed analysis.
- Extract results as a JSON object to integrate drift insights into our pipeline.

Output: Drift status and statistics for each feature, allow us to pinpoint which variables exhibit significant changes.

2. **Kolmogorov-Smirnov (KS) Test:**

The KS test is used to detect drift specifically in numerical features by comparing the cumulative distribution functions (CDFs) of the baseline and new data. It assesses whether the two samples come from the same distribution.

Implementation:

- For each numerical feature in the dataset, apply the ks_2samp function from scipy.stats, which returns a p-value indicating the likelihood that the baseline and new distributions are similar.
- If the p-value is below a threshold (typically 0.05), we conclude that drift has occurred in that feature.

Output: For each feature, the p-value and a drift flag (True if drift is detected, False otherwise).

3. **Population Stability Index (PSI):**

The PSI metric quantifies the stability of a feature's distribution by comparing how data is distributed across bins in the baseline and new datasets. This index is widely used in business and risk management settings.

Implementation:

- For each feature, bin the baseline and new data into a specified number of intervals.
- Calculate the PSI value by comparing the proportion of values in each bin between the two datasets.
- If the PSI value exceeds a certain threshold (commonly 0.1 or 0.2), it indicates drift for that feature.

Output: For each feature, the PSI value and a drift flag (True if PSI exceeds the threshold).

Steps in the Data Drift Detection Pipeline:

- **Load Data:** The pipeline begins by loading the latest dataset from DVC, which provides version-controlled access to the most recent raw data.
- **Run Evidently AI Drift Report:** Using the DataDriftDetector class, generate a comprehensive drift report with Evidently. This report is saved as an HTML file and summarized as a JSON object for analysis and further actions within the pipeline.
- **Kolmogorov-Smirnov (KS) Test for Numerical Drift:** Perform a KS test on each numerical feature, storing the p-value and drift flag results for evaluation.
- **PSI Test for Numerical Drift:** Calculate the PSI value for each numerical feature to assess distribution stability. The PSI results are saved for analysis and flagging of drifted features.
- **Email Notifications:** Upon completion, Airflow sends an email with the drift detection summary, including the HTML report if applicable, to notify stakeholders of the results.

7. Testing

The testing strategy for this MLOps project is comprehensive and focuses on several critical components of the data pipeline. The tests are designed to ensure the reliability and accuracy of various processes, from data retrieval to preprocessing and bias detection.

The tests are run automatically whenever a pull request is created to main branch. The tests can be run directly by following command as well - **pytest dags/tests/tests.py**

- For data download, the tests verify the accuracy of date calculations and proper API data retrieval. This is crucial because the project relies on timely and correct data acquisition. The tests check if the functions correctly calculate date ranges and if the API returns data for the expected dates.
- Data preprocessing tests are extensive, covering data cleaning, feature engineering, and normalization. These tests ensure that missing values are handled correctly, duplicates are removed, and new features (such as rolling statistics and lag features) are created accurately. The tests also verify the proper normalization of numerical

features and encoding of categorical variables. This is vital for maintaining data quality and preparing features correctly for model training.

- The data schema validation tests confirm that the API data adheres to the expected schema and that anomalies like negative values and NaN entries are correctly handled. This helps maintain data consistency and prevents errors due to unexpected data formats.
- The data drift detection tests focus on verifying the accuracy and robustness of methods used to detect changes in data over time. The tests for the DataDriftDetector class evaluate three key drift detection techniques: the Kolmogorov-Smirnov (KS) test, the Population Stability Index (PSI), and the Evidently-based data drift report. Each test checks if drift detection results are returned in the expected format and if drift is flagged correctly based on specified thresholds.
- Lastly, the bias detection and mitigation tests are particularly important for ensuring fairness in the machine learning pipeline. These tests validate the structure and metrics of the bias detection output and ensure that the mitigation process maintains all expected subgroups in the data. This is crucial for addressing potential biases in the data that could lead to unfair or discriminatory model predictions.

These unit tests are configured to run automatically via GitHub Actions whenever there's a pull request to the main branch. This automated testing process helps catch potential issues early, ensures new changes don't break existing functionality, and maintains the overall reliability of the MLOps pipeline. By integrating these tests into the CI/CD pipeline, the project can consistently maintain high standards of code quality and data processing throughout its development lifecycle.

8. Data Versioning

- In this project, we use Data Version Control (DVC) as our primary tool for managing and versioning datasets. DVC enables us to track changes to our data files and ensures that we maintain a version history of the data used in different stages of the project.

- The purpose of this setup is to ensure that datasets are version-controlled to maintain consistency and reproducibility throughout the project. DVC (Data Version Control) is employed to track and manage different versions of the dataset, with relevant .dvc files included in the repository. The raw data files are stored as CSVs in DVC, containing historical data to allow easy integration with newly fetched data from APIs.

- Three primary dataset files are stored and tracked within DVC: data_raw.csv.dvc, data_preprocess.csv.dvc, and bias_mitigated_data.csv.dvc. The data_raw.csv file holds the unprocessed API data, including all columns and without any preprocessing applied. The data_preprocess.csv file contains the dataset after five stages of preprocessing, with selected features prepared for further analysis. Finally, bias_mitigated_data.csv includes data that has

undergone bias mitigation across features and zones, making it ready to be fed directly into the machine learning model. This structure ensures that all stages of data processing are versioned and traceable for model training and evaluation.

9. Performance

Dataset preprocessing performance

- The purpose is to optimize the data preprocessing pipeline, addressing efficiency and performance concerns when handling large datasets. The dataset, consisting of approximately 120,000 rows, initially faced significant processing delays during each step of the preprocessing workflow. By analyzing a Gantt chart of the process, it was evident that sequential processing was a bottleneck, prompting the implementation of a chunking strategy to accelerate data transformation and generation.

- Two chunking strategies were employed to optimize the pipeline: (1) Chunking by fixed number of chunks divides the dataset into manageable segments based on a predefined chunk count, allowing multiple chunks to be processed in parallel. This method ensures uniform workload distribution across processing units, improving speed while maintaining resource efficiency. (2) Chunking by weather and demand in geographical regions involves segmenting data based on geographical zones, with each chunk representing specific weather and demand patterns for regions such as Texas, New York, and New England. This region-based segmentation enables targeted preprocessing, particularly for regional feature engineering, and supports parallel execution, significantly reducing overall runtime.

- By implementing these chunking strategies, the data preprocessing tasks are effectively parallelized, allowing the DAG (Directed Acyclic Graph) in the workflow orchestration tool (such as Apache Airflow) to execute steps concurrently. This parallelization has led to noticeable improvements in speed and scalability, facilitating faster data preparation without compromising accuracy or resource usage.