Amazon Reviews Sentiment Analysis

Team

Valli Meenaa Vellaiyan, Niresh Subramanian, Venkata Subbarao Shirish Addaganti, Harshit Sampgaon, Prabhat Chanda, Praneeth Korukonda

---

Contributions

---

1. **Data Collection & Sampling:** Niresh Subramanian and Prabhat Chanda
2. **Data Validation:** Valli Meenaa Vellaiyan and Venkata Subbarao Shirish Addaganti
3. **Data Preprocessing:** Harshit Sampgaon and Praneeth Korukonda

---

1. Introduction

In the competitive world of e-commerce, understanding customer feedback is essential for improving product offerings and enhancing the overall customer experience. This project focuses on analyzing sentiment in Amazon reviews to gain valuable insights into customer attitudes. By categorizing reviews as positive, neutral, or negative, Amazon can extract actionable insights to inform product decisions, optimize customer service, and support strategic initiatives aimed at enhancing customer satisfaction.

The primary objective of this project is to automate the process of analyzing review data. This includes data ingestion, pre-processing, model training, deployment, and monitoring. By analyzing sentiment trends, the solution provides insights into key pain points, emerging trends, and areas of potential improvement across various product categories, directly contributing to data-driven decision-making and a more customer-focused business strategy.

---

2. Dataset Overview

Dataset: UCSD Amazon Reviews 2023

The UCSD Amazon Reviews 2023 dataset is a comprehensive collection of customer reviews across multiple Amazon product categories, containing approximately 338 million reviews. This dataset serves as the foundation for our sentiment analysis model, with its large volume and diversity providing a robust basis for building scalable machine learning models.

Key Features

- **Review Text**: Main content of customer feedback.
- **Star Rating**: Ratings ranging from 1 to 5 stars.
- **Product Category**: Category of the product reviewed.
- **Review Timestamp**: Date and time of the review.
- **Product Metadata**: Additional product-related details.
- **Verified Purchase**: Indicator for reviews from verified purchases.
- **Review Helpfulness**: Upvotes or downvotes on the review, if available.

Data Rights and Privacy

The dataset is available for non-commercial use and excludes user identifiers to protect privacy. This project adheres to data minimization principles and complies with relevant privacy regulations.

---

3. Folder Structure and Data Organization

The project's files are organized to support the full lifecycle of the data pipeline. The structure includes folders for configuration files, datasets at different stages, and modularized code to facilitate each phase of data processing, model development, and deployment.

Primary Folders

- **data_pipeline**: Contains data pipeline assets and configurations, including Airflow DAGs for data ingestion, pre-processing, and validation.
- **data**: Stores datasets through various stages of processing (e.g., raw, cleaned, labeled).
- **model_pipeline**: Placeholder for model-specific pipeline assets, if needed.
- **logs**: Logs generated during Airflow task and DAG executions.
- **tests**: Test suite covering different data pipeline stages to ensure functionality and data integrity.

---

4. Data Pipeline Design

Docker Setup

This setup guide walks you through building and running the data pipeline environment using Docker, including a custom Airflow image and Docker Compose for service management.

Prerequisites

- **Docker Desktop** installed on your system.

Setup Instructions

1. Build the Docker Image

To build the custom Docker image for the Airflow environment, run:

docker build -t custom-airflow:latest .

This will create an image based on the Dockerfile provided, which includes all necessary dependencies and directory structure for running the pipeline.

2. Start the Docker Compose Services

The docker-compose.yaml file configures the services, including Airflow, PostgreSQL, and Redis. Ensure environment variables for SMTP and Airflow user configurations are either set in your environment or in a .env file.

Setting up airflow service, run:

docker compose up airflow-init

This command will:

- Start Airflow services (web server, scheduler, worker, and triggerer).

To start all services, run:

docker compose up -d

This command will:

- Restart Airflow services (web server, scheduler, worker, and triggerer).
- Set up **Redis** as the message broker.
- Set up **PostgreSQL** as the Airflow database backend.

3. Access the Airflow Web Interface

Once all services are running, you can access the Airflow web interface at http://localhost:8080.

The default credentials for Airflow are:

- **Username**: airflow
- **Password**: airflow

4. Verify Setup

After logging in, you should see the DAGs located in the data_pipeline/dags folder displayed in the Airflow interface.

Airflow DAGs Overview

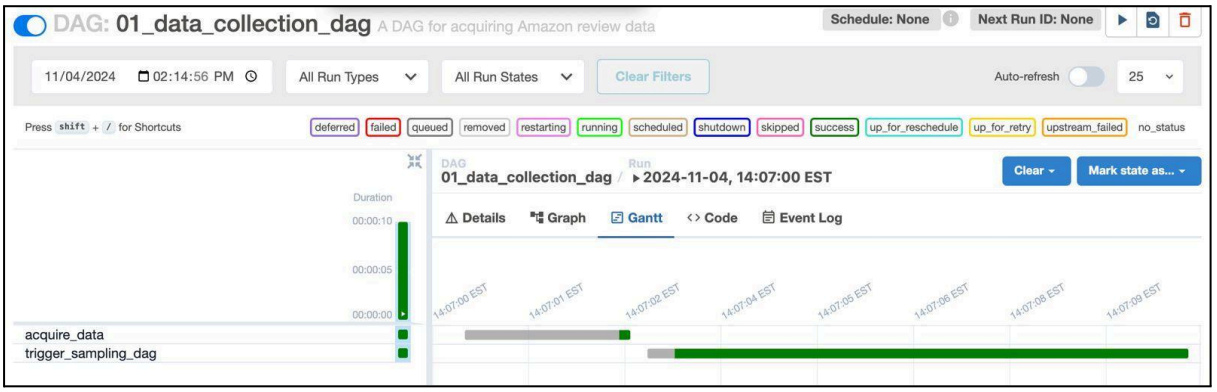
This project utilizes Airflow to manage and automate data pipeline stages. The DAGs support modular functionality across stages such as data acquisition, sampling, validation, and pre-processing, facilitating efficient and scalable data handling.

Pipeline Stages

- **Data Acquisition**: Extracts and ingests Amazon review data.
- **Data Sampling**: Samples review data across categories to balance representation.
- **Data Validation**: Validates data quality and structure, ensuring integrity for subsequent analysis.
- **Data Preprocessing**: Cleans, labels, and tags the reviews to prepare them for sentiment analysis.

Each stage is designed to run independently, ensuring a seamless workflow across data transformations while maintaining a sequential flow.

DAG-1: Data Acquisition

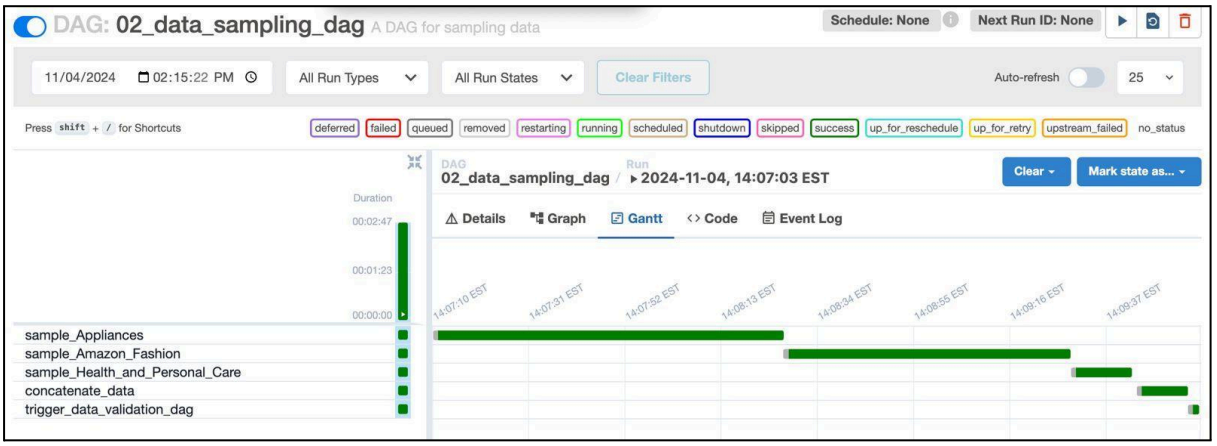


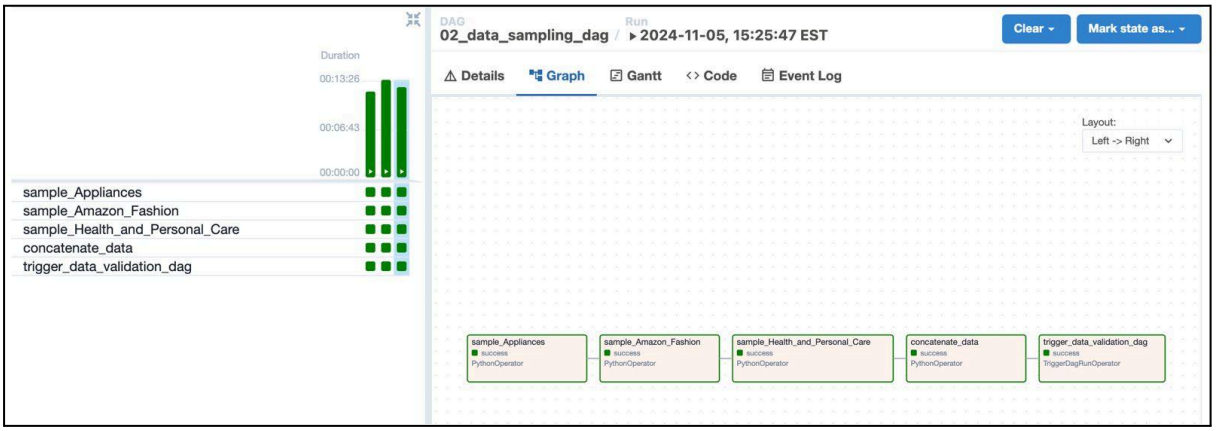
DAG ID: **01_data_collection_dag**
This DAG performs data acquisition for Amazon review data. The DAG consists of two main tasks:

Task 1: Acquire Data (acquire_data)

- Objective: Acquires and ingests Amazon review data.
- Process:
    - Calls the acquire_data function from the data_acquisition module to perform the data extraction.
    - Skips redownloading if file exists in data/raw directory.
    - Uses ThreadPoolExecutor to parallelize downloads, optimizing speed.
    - Allows tuning of concurrent downloads through MAX_WORKERS in config.py for efficient resource use.
    - Captures detailed logs for each download, including start/end times and errors.
    - This function handles the logic of retrieving data from the source and saving it locally.
- Output: Data is saved in a designated directory, ready for further processing.

DAG-2: Data Sampling





DAG ID: **02_data_sampling_dag**

This DAG is designed to sample and process Amazon review data by category using Python and Pandas. It orchestrates a sequence of tasks that handle data loading, filtering, joining with metadata, and sampling by specified review categories.

Task 1: Sample Data by Category (sample_data_{category})

- Objective: Sample data for each specified Amazon review category.
- Details:
  - For each category listed in the CATEGORIES configuration, a PythonOperator task is created to execute the sample_category function.
  - Each task loads, filters, joins, and samples review data using Pandas.
  - The sample_category function in sampling.py:
    - Reads JSONL.GZ files containing reviews and metadata.
    - Filters reviews by date and removes unnecessary fields.
    - Joins review data with product metadata for enriched sampling.
    - Performs stratified sampling to ensure diverse representation across months, main_category, and ratings.
  - The sampled data is then saved to CSV files, split by year (2018–2019 and 2020).

- Output: Saves sampled data as CSV files for each category in TARGET_DIRECTORY_SAMPLED, enabling further analysis by year.

Task 2: Concatenate Sampled Data (concatenate_data)

- Objective: Combine all sampled data CSV files across categories into unified datasets.
- Details:
    - This task reads the sampled data files from each category and consolidates them into two comprehensive CSVs: one for 2018–2019 and another for 2020.
    - The output CSV files are saved in the specified directory (TARGET_DIRECTORY_SAMPLED).
- Output: Produces sampled_data_2018_2019.csv and sampled_data_2020.csv containing concatenated review data for each time period.

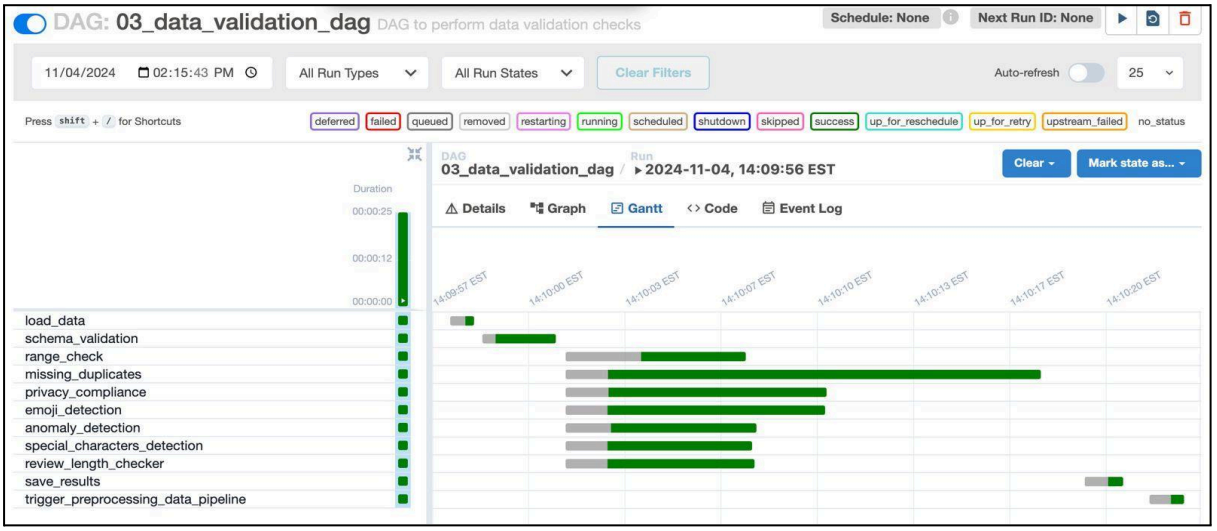
Task 3: Trigger Data Validation DAG (trigger_data_validation_dag)

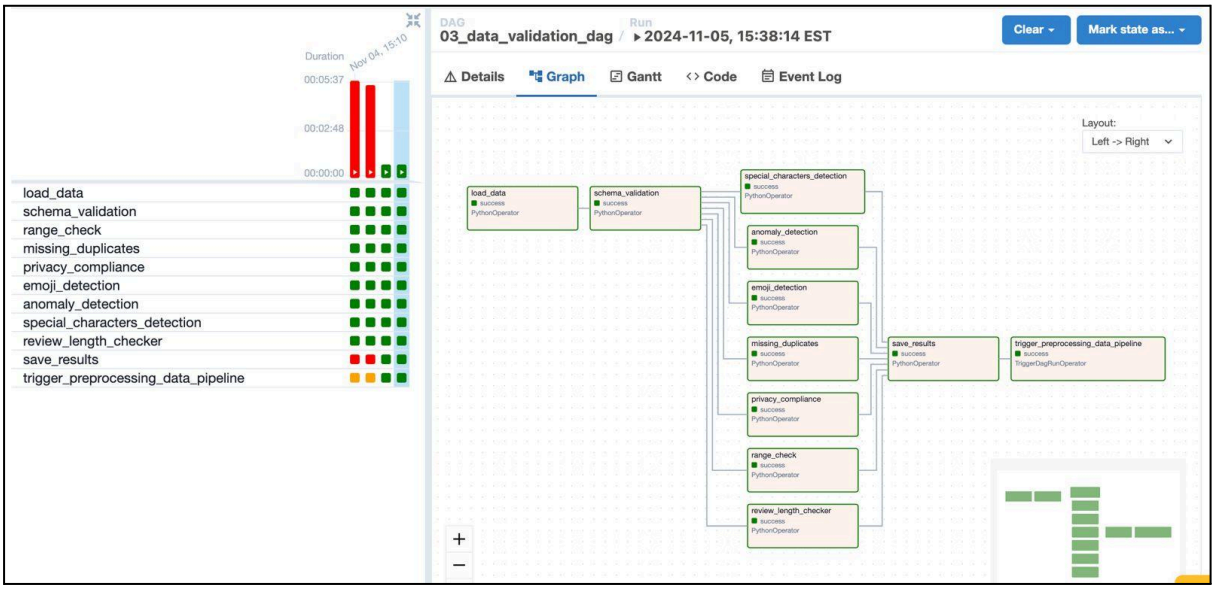
- Objective: Initiate the 03_data_validation_dag after data concatenation is completed.
- Details:
    - This task triggers a separate DAG to validate the concatenated data, ensuring data quality and completeness.
    - It acts as a starting point for further data validation or processing steps in a separate pipeline.
- Output: Marks the completion of data sampling and concatenation, ensuring readiness for validation.

Configuration and Logging

- Logging: Logging is set up to capture both console and file logs. Detailed logs are saved in /opt/airflow/logs, as specified by the LOG_DIRECTORY variable, and each task's operations are recorded, including any errors.
- Error Handling: In case of task failure, an email alert is sent to vallimeenaavellaiyan@gmail.com.

DAG-3: Data Validation





DAG ID: **03_data_validation_dag**
This DAG performs validation checks on the dataset to ensure data quality, integrity, and compliance with predefined standards. The DAG consists of several main tasks:

Task 1: Schema Validation (schema_validation)

- Objective: Validates the schema of the dataset against expected column types.
- Process:
    - Loads the dataset and checks if the schema matches the defined structure using the validate_schema function.
    - If validation fails, an error is raised, and the process stops.
- Output: Logs the status of schema validation.

Task 2: Range Check (range_check)

- Objective: Checks numerical columns for valid value ranges.
- Process:
    - Loads the dataset and applies the check_range function to identify any values that fall outside of acceptable limits.
- Output: Logs the rows that failed the range check and the overall status.

Task 3: Missing and Duplicates Check (missing_duplicates)

- Objective: Identifies any missing or duplicate entries in the dataset.
- Process:
    - Loads the dataset and uses the find_missing_and_duplicates function to determine if there are any missing values or duplicate rows.
- Output: Logs the indices of missing and duplicate rows along with the status.

Task 4: Privacy Compliance Check (privacy_compliance)

- Objective: Ensures that data complies with privacy regulations.
- Process:
    - Loads the dataset and applies the check_data_privacy function to identify any rows that may breach privacy guidelines.
- Output: Logs the rows that failed the privacy check and the status.

Task 5: Emoji Detection (emoji_detection)

- Objective: Detects and flags any emojis in the dataset.
- Process:
    - Loads the dataset and uses the detect_emoji function to identify any rows containing emojis.
- Output: Logs the indices of rows with emojis and the overall status.

Task 6: Anomaly Detection (anomaly_detection)

- Objective: Detects anomalies within the dataset.
- Process:
    - Loads the dataset and applies the detect_anomalies function to identify unusual patterns or outliers.
- Output: Logs the status of anomaly detection.

Task 7: Special Characters Detection (special_characters_detection)

- Objective: Checks for the presence of special characters in the dataset.
- Process:
    - Loads the dataset and applies the check_only_special_characters function to identify any rows with only special characters.
- Output: Logs the rows with special characters and the status.

Task 8: Review Length Check (review_length_checker)

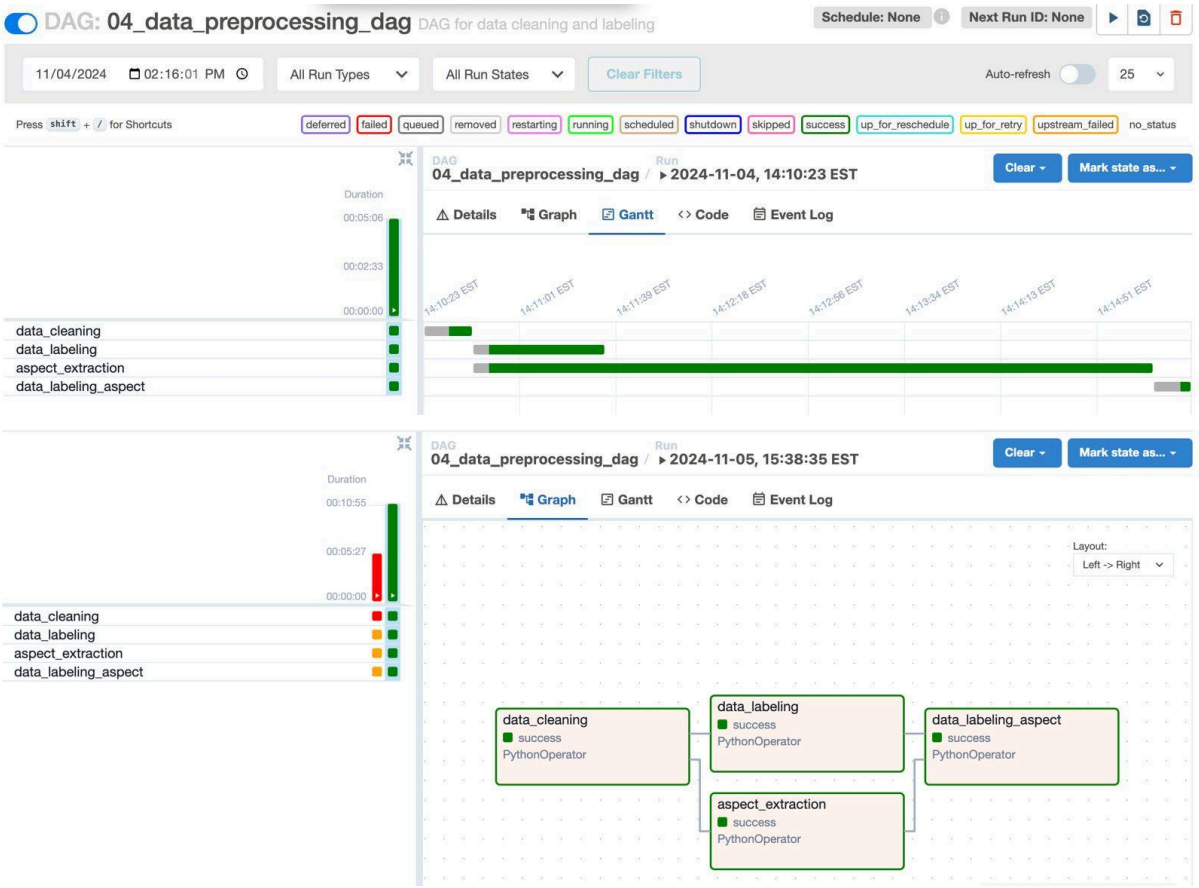
- Objective: Validates the length of reviews and titles in the dataset.
- Process:

- Loads the dataset and applies the check_review_title_length function to determine if any reviews or titles are too short or too long.
- Output: Logs the results of review length checks.

Task Dependencies

- Flow:
  - All validation tasks are executed in parallel.
  - The final task, save_results, runs after all validation tasks complete successfully to aggregate results and save them to a CSV file.

DAG-4: Data Preprocessing



DAG ID: 04_data_preprocessing_dag
This DAG performs data cleaning, labeling, and aspect-based sentiment analysis on Amazon review data. The DAG consists of four main tasks:

Sampled Data Data Card [Input Data]

Shape: (136,256 rows x 18 columns)
Columns:

- rating, title, text, asin, parent_asin, user_id, timestamp, helpful_vote, verified_purchase, review_date_timestamp, main_category, product_name, categories, price, average_rating, rating_number, review_month, year

Task 1: Data Cleaning (data_cleaning)

- Objective: Cleans the raw data, removes unwanted emojis based on a validation file, and saves the cleaned data.
- Process:
    - Loads raw data and a validation file with emoji indices to be removed.
    - Applies the clean_amazon_reviews function from the data_cleaning_pandas module, which removes unwanted content and emojis.
    - Saves the cleaned data to a specified file path.
- Output: cleaned_data.csv - the cleaned dataset without unwanted emojis.

Task 2.1: Data Labeling (data_labeling)

- **Objective**: Labels the cleaned data with overall sentiment.
- **Process**:
    - Loads the cleaned data file.
    - Applies the apply_labelling function from the data_labeling module, leveraging **Snorkel's weak supervision** and **VADER** sentiment analysis to generate sentiment labels based on the content.
    - Uses **Snorkel** labeling functions (LFs) like lf_positive_negative_ratio, lf_high_rating, lf_low_rating, and lf_vader_sentiment to classify reviews as "Positive," "Negative," or "Neutral."
    - Trains Snorkel's LabelModel on these weak labels to create a final sentiment classification for each review.
    - Saves the labeled data with the predicted sentiment labels added to the dataset.
- **Issues and Solutions**:
    - **Issue**: The initial approach using a custom rule-based function alongside ratings produced low-accuracy labels.
      Solution: Integrated **VADER sentiment analysis** with ratings to enhance label accuracy.
    - **Issue**: Training the **Snorkel LabelModel** was time-consuming.
      Solution: Parallelized the labeling task with the aspect extraction process to optimize compute resource utilization.

Output: labeled_data.csv - Dataset labeled with overall sentiment.

Labeled Data Data Card [Output Data - 1]

Shape: (136,055 rows x 19 columns)
Columns:

- rating, title, text, asin, parent_asin, user_id, timestamp, helpful_vote, verified_purchase, review_date_timestamp, main_category, product_name, categories, price, average_rating, rating_number, review_month, year, sentiment_label

Task 2.2: Aspect Extraction (aspect_extraction)

- **Objective**: Identifies and extracts specific aspects within the reviews, such as "delivery," "quality," and "cost."
- **Process**:
    - Loads the cleaned data file.
    - Uses NLTK to identify and expand aspects within review text. This includes:
        - Using **WordNet** to generate synonyms for predefined aspects (e.g., "delivery" includes "shipping," "arrive").
        - Applying the tag_and_expand_aspects function from the aspect_extraction module to identify sentences associated with each aspect.
        - Tokenizing each review sentence and tagging part of speech (POS) for accurate aspect identification.
    - For each detected aspect, sentences containing relevant keywords are tagged and saved in the dataset.
    - Saves the aspect-extracted data for further sentiment analysis.
- **Issues and Solutions**:
    - **Issue**: Encountered NLTK path issues, causing errors when loading required packages.
      Solution: Verified and set the correct NLTK data path in the code to ensure proper package loading.
    - **Issue**: Aspect extraction was time-consuming due to the processing involved in tokenizing sentences and tagging parts of speech (POS).
      Solution: Parallelized the aspect extraction process with other data labeling for the entire review task to improve efficiency and reduce overall processing time.
- **Output**: aspect_extracted_data.csv - Dataset with tagged aspects for further sentiment analysis.

Task 3: Aspect-Based Data Labeling (data_labeling_aspect)

- **Objective**: Applies sentiment labeling to specific aspects within the reviews.
- **Process**:
    - Loads the aspect-extracted data file containing identified aspects with relevant sentences.
    - Uses VADER sentiment analysis to assign sentiment labels to each aspect:

        - The apply_vader_labeling function uses VADER's compound score to label each aspect as **POSITIVE**, **NEGATIVE**, or **NEUTRAL**.
    - Maps numeric labels to descriptive sentiment labels for clarity.
    - Saves the aspect-labeled data, providing a granular sentiment analysis for each identified aspect.
- **Output**: labeled_aspect_data.csv - Dataset with aspect-specific sentiment labels.

Aspect Labeled Data Data Card [Output Data - 2]

Shape: (34,801 rows x 21 columns)
Columns:

- rating, title, text, asin, parent_asin, user_id, timestamp, helpful_vote, verified_purchase, review_date_timestamp, main_category, product_name, categories, price, average_rating, rating_number, review_month, year, aspect, relevant_sentances, sentiment_label

Task Dependencies

- Flow:
    - The data_cleaning task is executed first.
    - data_cleaning is followed by parallel tasks aspect_extraction and data_labeling.
    - Finally, both aspect_extraction and data_labeling tasks must complete before starting the data_labeling_aspect task.

Function Implementations for DAGs

Data Collection and Sampling

1. Data Acquisition

Function: setup_logging()

Function Description:
Configures logging with both file and stream handlers, setting a dynamic filename based on the current timestamp.

Key Steps:

- Sets up a logger instance.
- Configures both file and console handlers.
- Dynamically generates a log file name using the current timestamp.

Optimizations:

- **Reuses Logger Instance:** Uses a single logger instance with conditional handling to prevent adding duplicate handlers.

Function: **download_file(url, file_name, chunk_size=8192)**

Function Description:
Downloads a file in chunks, showing progress, and saves it locally. Checks for existing files to avoid redundant downloads.

Key Steps:

- Checks if the file already exists to avoid re-downloading.
- Streams the file in chunks, saving each chunk incrementally.
- Displays download progress using a progress bar.

Optimizations:

- **File Existence Check:** Skips download if the file already exists.
- **Streaming and Chunked Downloads:** Downloads the file in chunks to manage memory usage efficiently.
- **Progress Display:** Utilizes tqdm for visual feedback during the download process.

Function: **download_category(category, review_base_url, meta_base_url, target_directory)**

Function Description:
Initiates downloading for review and metadata for a given category, logging the start and end times.

Key Steps:

- Constructs the URLs for review and metadata based on the category.
- Initiates the download process and logs the start and completion times.

Optimizations:

- **URL Building and Directory Handling:** Implements URL construction and directory management within the function for seamless integration with multi-threaded downloads.

Function: **acquire_data()**

Function Description:
Starts the download process for all categories concurrently using ThreadPoolExecutor.

Key Steps:

- Utilizes ThreadPoolExecutor to manage concurrent downloads for each category.

- Coordinates the downloading tasks and monitors their completion.

Optimizations:

- **Multi-threaded Downloads:** Leverages ThreadPoolExecutor for parallel downloads to maximize network and CPU utilization.
- **Dynamic Worker Count:** Adjusts the number of concurrent workers through a defined MAX_WORKERS setting to optimize performance.

---

2. Data Concatenation and Storage

Function: concatenate_and_save_csv_files()

Function Description:
Loads CSV files, concatenates data by year, and saves combined files.

Key Steps:

- Reads CSV files matching specific naming patterns.
- Concatenates the loaded data based on the year and saves the resulting DataFrame.

Optimizations:

- **Selective Loading and Concatenation:** Only reads files that match a specific naming pattern, reducing unnecessary file operations.
- **Batch Concatenation:** Groups files by year for efficient processing, minimizing memory usage during concatenation.

---

3. Data Processing and Sampling

Function: read_jsonl_gz_in_chunks(file_path, chunksize=10000)

Function Description:
Reads a large compressed JSONL file in chunks, yielding DataFrames.

Key Steps:

- Opens the JSONL.GZ file for reading.
- Reads the file in defined chunks, yielding DataFrames for further processing.

Optimizations:

- **Chunked Reading:** Efficiently manages large datasets by reading them in manageable chunks to prevent memory overload.

Function: load_jsonl_gz(file_path, nrows=None)

Function Description:
Loads data from a JSONL.GZ file with an optional row limit.

Key Steps:

- Reads the JSONL.GZ file and loads the data into a DataFrame.
- Applies a row limit if specified, allowing for early termination.

Optimizations:

- **Chunked Loading:** Combines chunks for memory efficiency and enables early termination when a row limit is defined.
- **Progress Monitoring:** Uses tqdm for visual feedback on load progress.

Function: process_reviews_df(reviews_df)

Function Description:
Filters reviews within a specific date range and drops unnecessary columns.

Key Steps:

- Applies date filtering based on a defined range.
- Drops irrelevant columns to reduce DataFrame size.

Optimizations:

- **Combined Operations:** Applies date filtering and column dropping in a single step to optimize memory usage.

Function: join_dataframes(filtered_reviews_df, metadata_df)

Function Description:
Joins review data with metadata to enrich information based on a common key (parent_asin).

Key Steps:

- Renames and selects relevant columns from both DataFrames.
- Performs a left join on the specified key to combine data.

Optimizations:

- **Selective Column Inclusion:** Limits the joined DataFrame size by selecting only relevant columns, enhancing memory efficiency.
- **Left Join Usage:** Reduces the size of the resulting DataFrame by including only matching rows from the filtered reviews DataFrame.

Function: sample_data(joined_df)

Function Description:
Performs stratified sampling on the joined data, grouping by month and rating.

Key Steps:

- Groups the DataFrame by month and rating.
- Performs sampling to ensure representation across categories.

Optimizations:

- **Stratified Sampling:** Ensures representative sampling across different categories, preserving original data distributions.
- **In-place Operations:** Minimizes the creation of copies of DataFrames, applying operations directly to reduce overhead.

Function: process_sampled_data(sampled_df)

Function Description:
Further processes the sampled data by splitting it based on year, preparing separate files.

Key Steps:

- Splits the DataFrame into separate DataFrames based on the year.
- Prepares the data for efficient storage.

Optimizations:

- **Year-based Filtering:** Efficiently organizes data into year-specific DataFrames for easier handling and storage.

Function: sample_category(category_name)

Function Description:
Orchestrates the end-to-end sampling process for a category, from loading data to saving sampled files.

Key Steps:

- Loads the data for the specified category.
- Processes the data and applies sampling.
- Saves the final sampled files.

Optimizations:

- **Modularized Data Processing:** Reuses individual functions for downloading, processing, and sampling, enhancing code maintainability.
- **Error Handling:** Implements try-except blocks to manage category-specific errors, ensuring uninterrupted execution.

Data Validation

Function : detect_anomalies(data: pd.DataFrame) -> bool

Function Description:
Detects anomalies in a given DataFrame by checking for outliers in numeric columns, verifying date ranges, and ensuring categorical values are consistent with expected categories.

Key Steps:

- **Outlier Detection:** Identifies outliers in specified numeric columns using the z-score method.
- **Temporal Anomalies:** Checks if dates in the review_date_timestamp column exceed a specified cutoff date (December 31, 2023).
- **Category Consistency Checks:** Validates values in categorical columns (e.g., main_category, verified_purchase, categories) against predefined expected values.

Optimizations:

- **Logging Enhancements:** Detailed logging captures the progress and results of each anomaly check, aiding in debugging.
- **Dynamic Column Handling:** Checks for the presence of each column before processing to prevent errors from missing columns.
- **Reduced Redundancy:** Category consistency checks are encapsulated in a separate function, enhancing code reuse and readability.

Function : check_category_consistency(column, expected_values)

Function Description:
Verifies that all entries in a specified column match a set of expected values and logs any unexpected entries.

Key Steps:

- Logs the expected categories and counts unexpected values in the specified column.

Optimizations:

- **Separation of Concerns:** Isolates category checking logic, making the main function cleaner and more maintainable.

- **Efficient Value Checking:** Utilizes the isin method for efficient membership checking against expected categories.

Function : detect_emoji(data: pd.DataFrame, text_column: str = "text") -> (list, bool)

Function Description:
Identifies rows in a DataFrame that contain emojis in a specified text column.

Key Steps:

- Iterates through the specified text column and uses a regular expression pattern to detect emojis.
- Logs the indices of rows containing emojis and returns a list of these indices.

Optimizations:

- **Error Handling:** Checks if the specified column exists and logs an error if it does not, enhancing robustness.
- **Detailed Logging:** Provides clear logging for both successful detections and the absence of emojis.

Function : detect_non_english(review)

Function Description:
Checks if a review is non-English based on the presence of non-ASCII characters.

Key Steps:

- Uses a regex pattern to identify non-ASCII characters, assuming these indicate a non-English review.
- Logs detected non-English reviews along with their language classification.

Optimizations:

- **Simplified Logic:** Implements a straightforward regex-based approach for language detection.
- **Focused Logging:** Only logs details for non-English reviews to minimize log clutter.

Function : detect_emojis(review)

Function Description:
Extracts emojis from a given review text.

Key Steps:

- Applies a regular expression to find all emojis within the text.

Optimizations:

- **Pattern Compilation:** Compiles the regex pattern once at the beginning, improving performance during multiple calls.
- **Selective Logging:** Logs only when emojis are found, keeping the logs relevant.

Function : process_reviews(df)

Function Description:
Processes a DataFrame to detect both non-English reviews and emojis.

Key Steps:

- Iterates through the DataFrame to apply detect_non_english and detect_emojis on each review.

Optimizations:

- **Batch Processing:** Structured to handle entire DataFrames efficiently rather than individual rows, enhancing performance.

Function : find_missing_and_duplicates(data: pd.DataFrame) -> (list, list, bool)

Function Description:
Checks a DataFrame for missing values and duplicate rows, specifically targeting critical columns defined in CRITICAL_COLUMNS.

Key Steps:

- Iterates through critical columns to check for missing values.
- Identifies duplicate rows in the DataFrame.
- Collects and returns indices of missing values and duplicates.

Optimizations:

- **Iterate with Conditions:** Combines missing value checks and duplicate checks into a single process to streamline validation.
- **Use of isnull():** Employs isnull().any() to quickly assess the presence of any missing values in critical columns.
- **Efficient Index Handling:** Uses data[data.duplicated()].index.tolist() to gather duplicate indices efficiently.

Function : check_data_privacy(data: pd.DataFrame) -> (list, bool)

Function Description:
Examines specified columns for potential Personally Identifiable Information (PII), such as emails and phone numbers.

Key Steps:

- Defines regex patterns for various types of PII.
- Iterates over the DataFrame to check for potential PII based on the defined patterns.
- Collects indices of rows that contain PII and logs them.

Optimizations:

- **Regex Pattern Matching:** Utilizes regex patterns stored in a dictionary to check for multiple PII types efficiently in a single loop.
- **Valid Phone Check:** Implements a helper function is_valid_phone_number to validate phone numbers, enhancing robustness.
- **Set for Uniqueness:** Converts PII indices to a set before returning them as a list, eliminating duplicates effectively.

Function : check_range(data: pd.DataFrame) -> (dict, bool)

Function Description:
Validates whether numeric values in specified columns fall within acceptable predefined ranges.

Key Steps:

- Defines a dictionary for range checks (minimum and maximum values).
- Iterates through the DataFrame to check each specified column against its defined range.
- Collects invalid row indices for each column and compiles them into a dictionary.

Optimizations:

- **Dynamic Range Checks:** Utilizes a dictionary to define range checks, making the function adaptable for new columns and conditions.
- **Min/Max Handling:** Separates checks for values below the minimum and above the maximum, providing detailed logs for both scenarios.
- **Set Removal of Duplicates:** Uses a set to ensure unique invalid indices, improving the efficiency of the data integrity check.

Function : check_review_title_length(data: pd.DataFrame) -> (list, list, bool)

Function Description:
Checks if the lengths of review texts and titles meet specified character limits.

Key Steps:

- Initializes lists for short and long reviews/titles.
- Applies vectorized string operations to compute lengths of reviews and titles.
- Compares these lengths against specified limits and categorizes them.

Optimizations:

- **Combined Length Check:** Uses .str.len() for vectorized length checks, which is faster than iterating over rows.
- **Logging Efficiency:** Consolidates logging into a single call to report results, reducing overhead and improving clarity.
- **Default Empty String Handling:** Replaces NaN values with empty strings to avoid errors in length calculations.

Function : validate_schema(data: pd.DataFrame) -> bool

Function Description:
Validates the schema of a given DataFrame against a predefined expected schema, checking for missing columns and verifying data types.

Key Steps:

- Initiates the schema validation process and logs the start.
- Iterates over the EXPECTED_SCHEMA to check for the existence of columns and validate their data types.
- Logs errors for any discrepancies and collects information about missing or extra columns.

Optimizations:

- **Error Logging:** Logs detailed messages that include sample data, aiding in troubleshooting.
- **Use of Set Operations:** Employs set operations for checking unexpected columns, enhancing performance for membership testing.

Function : check_only_special_characters(data: pd.DataFrame) -> list

Function Description:
Identifies reviews that consist solely of special characters, logging their row indices.

Key Steps:

- Initializes a list to store indices of invalid reviews.
- Iterates over the review text, using regex to identify those that contain only special characters.

- Logs details about invalid reviews found, including handling cases where all reviews may be missing (null).

Optimizations:

- **Regular Expression Usage:** Efficiently matches patterns using regex, which is faster than manual string checks.
- **Conditional Checks:** Verifies that the review is a string before applying the regex to avoid unnecessary errors.
- **Detailed Logging:** Provides specific error messages for invalid reviews and effectively manages null review cases.

Data Preprocessing :

Function: vader_sentiment_label(row: pd.Series) -> str

Function Description:
Classifies the sentiment of a review text as positive, negative, or neutral based on the VADER sentiment analysis tool.

Key Steps:

- Extracts the relevant_sentences from the row of the DataFrame.
- Uses the VADER sentiment analyzer to calculate the compound sentiment score.
- Categorizes the sentiment based on the score:
    - **Positive**: score ≥ 0.05
    - **Negative**: score ≤ -0.05
    - **Neutral**: $-0.05 <$ score < 0.05

Optimizations:

- **Compound Score Classification**: Efficiently uses compound sentiment scores for classification, avoiding the need for complex rule-based sentiment evaluation.
- **Simple Sentiment Mapping**: Simplifies the sentiment categorization process using simple thresholds.

Function: apply_vader_labeling(data: pd.DataFrame) -> pd.DataFrame

Function Description:
Applies the vader_sentiment_label function across the entire DataFrame to label reviews with sentiment labels.

Key Steps:

- Logs the start and end times of the sentiment labeling process.
- Uses the apply() function to apply vader_sentiment_label to every row.

- Maps numeric sentiment labels (0, 1, 2) to descriptive labels ("NEGATIVE", "POSITIVE", "NEUTRAL").
- Returns the DataFrame with an added sentiment_label column.

Optimizations:

- **Batch Processing**: Uses apply() to process rows in batches, optimizing for performance over row-by-row iteration.
- **Efficient Mapping**: Maps sentiment labels using a dictionary, which is faster than multiple conditional checks.

Function: get_synonyms(word: str) -> set

Function Description:
Retrieves synonyms for a given word from the WordNet lexical database using NLTK.

Key Steps:

- Accepts a word as input and retrieves its synsets (sets of synonyms) from WordNet.
- Extracts the lemma names (synonyms) for each synset.
- Returns the synonyms as a set, with underscores replaced by spaces for multi-word phrases.

Optimizations:

- **Efficient Synset Lookup**: Uses WordNet's built-in synset functionality, which is optimized for finding related words.
- **Set Return Type**: Returns synonyms as a set, ensuring uniqueness and reducing memory overhead.

Function: tag_and_expand_aspects(data: pd.DataFrame, aspect_keywords: dict) -> pd.DataFrame

Function Description:
Tags and expands reviews by identifying and associating sentences with predefined aspects.

Key Steps:

- Iterates over each row (review) and splits it into sentences.
- Tokenizes sentences and performs part-of-speech tagging using pos_tag().
- Matches words in the sentence to predefined aspect keywords from the aspect_keywords dictionary.
- If an aspect is identified, creates new rows for the associated aspect and sentence.
- Returns the updated DataFrame with expanded rows for each aspect.

Optimizations:

- **Efficient Tokenization**: Uses nltk.pos_tag() to efficiently tag parts of speech, which helps in accurately detecting aspect-related sentences.
- **Aspect-Based Expansion**: Expands the dataset based on aspect detection, improving the granularity of the data for further analysis.

Function: clean_amazon_reviews(data: pd.DataFrame) -> pd.DataFrame

Function Description:
Cleans and preprocesses the Amazon reviews dataset, handling missing values, duplicates, and text anomalies.

Key Steps:

- Removes duplicate rows and drops rows with missing text or rating values.
- Adjusts data types to optimize memory usage, such as converting rating and price columns to appropriate types.
- Fills missing values in 'title' and 'price' columns.
- Cleans up extra spaces and HTML tags in the review text using regular expressions.
- Demojizes emojis in review text to standard text.

Optimizations:

- **Memory Optimization**: Converts data types to save memory, improving performance for large datasets.
- **Efficient Text Cleaning**: Uses regular expressions to efficiently clean review text and remove unwanted characters.
- **Null Handling**: Properly handles missing values by filling them with default values, ensuring no data loss.

Function: lf_positive_negative_ratio(row) -> str

Function Description:

This labeling function computes the ratio of positive to negative words in the review text to classify the sentiment as "Positive", "Negative", or "Neutral." If both positive and negative words are present, it calculates a ratio and classifies the sentiment based on thresholds. If only one type of word is found, it assigns the sentiment accordingly.

Key Steps:

1. **Text Processing**: Counts the occurrences of positive and negative words in the review text.
2. **Ratio Calculation**: If both positive and negative words are present, computes the ratio of positive to negative words.
3. **Classification**:
    - If the ratio is between 0.9 and 1.1, it assigns a "Neutral" label.

    - If the ratio is greater than 1.1, it assigns "Positive."
    - If the ratio is less than 0.9, it assigns "Negative."
    - If only positive words are found, labels the sentiment as "Positive," and if only negative words are found, it labels the sentiment as "Negative."
4. **Fallback**: If neither positive nor negative words are present, it returns "Neutral."

Optimizations:

- **Efficient Text Search**: Uses lower() to handle case-insensitivity and ensures that all words are checked regardless of casing.
- **Conditional Checks**: Early returns for conditions where no positive or negative words are found to reduce unnecessary calculations.

Function: lf_high_rating(row) -> str

Function Description:

This labeling function classifies reviews with high ratings (4 or more) as "Positive" and other reviews as "Neutral." It helps capture sentiment based on explicit user ratings.

Key Steps:

1. **Rating Check**: Checks if the review rating is greater than or equal to 4.
2. **Label Assignment**:
    - Returns "Positive" if the rating is 4 or more.
    - Returns "Neutral" if the rating is less than 4.

Optimizations:

- **Direct Comparison**: Uses a simple comparison (>= 4) for efficient checking.
- **Lightweight Logic**: The function applies minimal computation, ensuring quick execution.

Function: lf_low_rating(row) -> str

Function Description:

This labeling function classifies reviews with low ratings (2 or lower) as "Negative" and other reviews as "Neutral." It is effective in detecting negative sentiment based on the explicit rating.

Key Steps:

1. **Rating Check**: Checks if the review rating is less than or equal to 2.
2. **Label Assignment**:
    - Returns "Negative" if the rating is 2 or lower.
    - Returns "Neutral" if the rating is above 2.

Optimizations:

- **Efficient Rating Check**: Simple comparison (<= 2) for quick execution.
- **Minimal Computational Overhead**: Only applies logic for low ratings, improving speed.

Function: lf_vader_sentiment(row) -> str

Function Description:

This labeling function uses the VADER sentiment analyzer to classify the sentiment of the review text as "Positive," "Negative," or "Neutral." It leverages a sentiment score generated by VADER's polarity_scores function.

Key Steps:

1. **Sentiment Analysis**: Uses the VADER SentimentIntensityAnalyzer to obtain a sentiment score from the review text.
2. **Classification**:
    - Returns "Positive" if the compound sentiment score is greater than or equal to 0.05.
    - Returns "Negative" if the score is less than or equal to -0.05.
    - Returns "Neutral" if the score is between -0.05 and 0.05.
3. **Fallback**: If the text is empty, the function returns "Neutral."

Optimizations:

- **VADER Analyzer**: Utilizes a highly optimized and pre-built sentiment analysis tool, avoiding the need to manually build complex models.
- **Quick Sentiment Score Retrieval**: Efficient extraction of sentiment scores using the polarity_scores method.

Function: apply_labelling(df_cleaned: pd.DataFrame) -> pd.DataFrame

Function Description:

This function applies a set of labeling functions to a cleaned DataFrame of reviews and returns a DataFrame with added sentiment labels. It also trains a label model using weak supervision and generates predictions based on the applied labeling functions.

Key Steps:

1. **Labeling Function Setup**: The function initializes and applies a list of labeling functions (including lf_high_rating, lf_low_rating, and lf_vader_sentiment).
2. **Label Application**: The function uses LFApplier to apply the labeling functions to the DataFrame. It logs the time taken to apply the functions for performance tracking.
3. **Label Model Training**: The function trains a LabelModel using the weak labels generated by the labeling functions. The model is trained for up to 500 epochs to optimize label prediction.

4. **Label Prediction**: After training, the function predicts labels using the trained label model and adds the predicted labels as a new column to the DataFrame.
5. **Error Handling**: Includes comprehensive error handling with logging to capture issues at every step of the process.

Optimizations:

- **Parallel Processing of Labels**: Uses LFApplier to efficiently apply multiple labeling functions to the dataset in parallel.
- **Model Fitting Efficiency**: Tracks the time taken to fit the label model and optimize performance during the fitting process.
- **Logging and Monitoring**: Implements detailed logging to track the labeling and prediction processes, making it easier to identify and resolve issues.

GitHub Actions Workflows for Data Pipeline Testing

To ensure data pipeline robustness, three GitHub Actions workflows have been implemented: **Data Collection Function Test**, **Data Validation Function Test (Schema Validation)**, and **Data Preprocessing Tests**. These workflows automate testing of critical components, ensuring that any changes to the data pipeline modules trigger immediate, thorough validation. Below is a breakdown of each workflow and its contributions to data integrity and code quality.

---

1. Data Collection Function Test

This workflow is dedicated to testing the functions responsible for data collection within the pipeline. By ensuring comprehensive validation of changes to the data collection module, it helps maintain data integrity and reliability.

- **Trigger Events**: The workflow is triggered by push or pull_request events involving the data_pipeline/dags/utils/data_collection/ path. This ensures timely testing for any updates to data collection functions.
- **Workflow Steps**:
    - **Check Out Code**: The workflow uses actions/checkout to pull the latest code from the repository, with a shallow clone (fetch-depth: 1) to speed up the process.
    - **Set Up Python**: Python 3.10 is installed using actions/setup-python, matching the project's version requirements.
    - **Cache pip Packages**: To avoid redundant installations, dependencies are cached with actions/cache, using a unique key based on the operating system and a hash of data_collection_test_requirements.txt.
    - **Install Dependencies**: The required packages specified in data_collection_test_requirements.txt are installed to set up the test environment.
    - **Set PYTHONPATH**: The PYTHONPATH environment variable is configured to include the repository root, ensuring correct module imports.

- **Run Tests**: Tests are executed with pytest in the data_pipeline/tests/data_collection/ directory. Warnings are disabled (--disable-warnings) for a cleaner output, helping to confirm that data collection functions are working as expected.

---

2. Data Validation Function Test (Schema Validation)

This workflow focuses on schema validation, ensuring that incoming data adheres to predefined formats and structures before further processing. This prevents errors from propagating through the pipeline due to improperly structured data.

- **Trigger Events**: Activated on push or pull_request events that affect files in data_pipeline/dags/utils/data_validation/. This setup provides continuous testing for any modifications to data validation functions.
- **Workflow Steps**:
  - **Check Out Code**: The latest code is retrieved using actions/checkout@v3 with a shallow clone (fetch-depth: 1).
  - **Set Up Python**: The workflow sets up Python 3.10 with actions/setup-python@v4, as specified by the project.
  - **Cache pip Packages**: Dependencies are cached with actions/cache@v3 using a key based on a hash of test_requirements.txt. This reduces installation time by reusing cached packages unless dependencies change.
  - **Install Dependencies**: Packages from data_pipeline/tests/data_validation/test_requirements.txt are installed to ensure that all necessary libraries for schema validation are available.
  - **Set PYTHONPATH**: The PYTHONPATH variable is updated to include the repository root, enabling direct resolution of the data validation functions during testing.
  - **Run Tests**: The pytest tool runs schema validation tests located in the data_pipeline/tests/data_validation directory. The --disable-warnings flag ensures that test outputs are clean, highlighting any discrepancies in data format or structure.

---

3. Data Preprocessing Tests

This workflow automates testing for data preprocessing functions, verifying that transformations are correctly applied and consistently produce reliable results.

- **Trigger Events**: Initiates on push events affecting the data_pipeline/dags/utils/data_preprocessing/ directory or the related test directory (data_pipeline/tests/data_preprocessing/). This setup ensures that any updates to preprocessing functions or tests trigger validation.
- **Workflow Steps**:
  - **Check Out Code**: The latest code is retrieved with actions/checkout, ensuring that recent updates are included in the test process.
  - **Set Up Python**: Python 3.12.7 is installed using actions/setup-python to match the project's environment requirements.

- **Cache Python Packages**: Dependencies are cached with actions/cache, leveraging a key based on the OS and a hash of requirements.txt. This speeds up workflow execution by avoiding redundant installations.
- **Install Dependencies**: Necessary packages from data_pipeline/requirements.txt are installed, with pip upgraded beforehand for compatibility with the latest packages.
- **Run All Tests**: Each test file in data_pipeline/tests/data_preprocessing/ is identified and executed sequentially with unittest. If any test fails, the workflow halts, ensuring comprehensive coverage of all preprocessing test cases.

CodeQLSecurity and Vulnerability Scan

To ensure the robustness and security of our data pipeline, we incorporated CodeQL to scan the entire main branch for potential vulnerabilities, including exposed secret keys or other sensitive information. This tool allowed us to proactively identify and address any security concerns within our codebase.

Fortunately, the scan confirmed that no secret keys were exposed, and all identified vulnerabilities were promptly resolved. This final security check reinforces our commitment to maintaining a secure and reliable pipeline.
