Amazon Reviews Sentiment Analysis

Team

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

1. Introduction

Project Overview

This document details a model pipeline designed for sentiment analysis, leveraging both BERT and RoBERTa models. The purpose of this pipeline is to automate data ingestion, model training, hyperparameter tuning, bias detection, and deployment, ensuring a robust and efficient machine learning workflow.

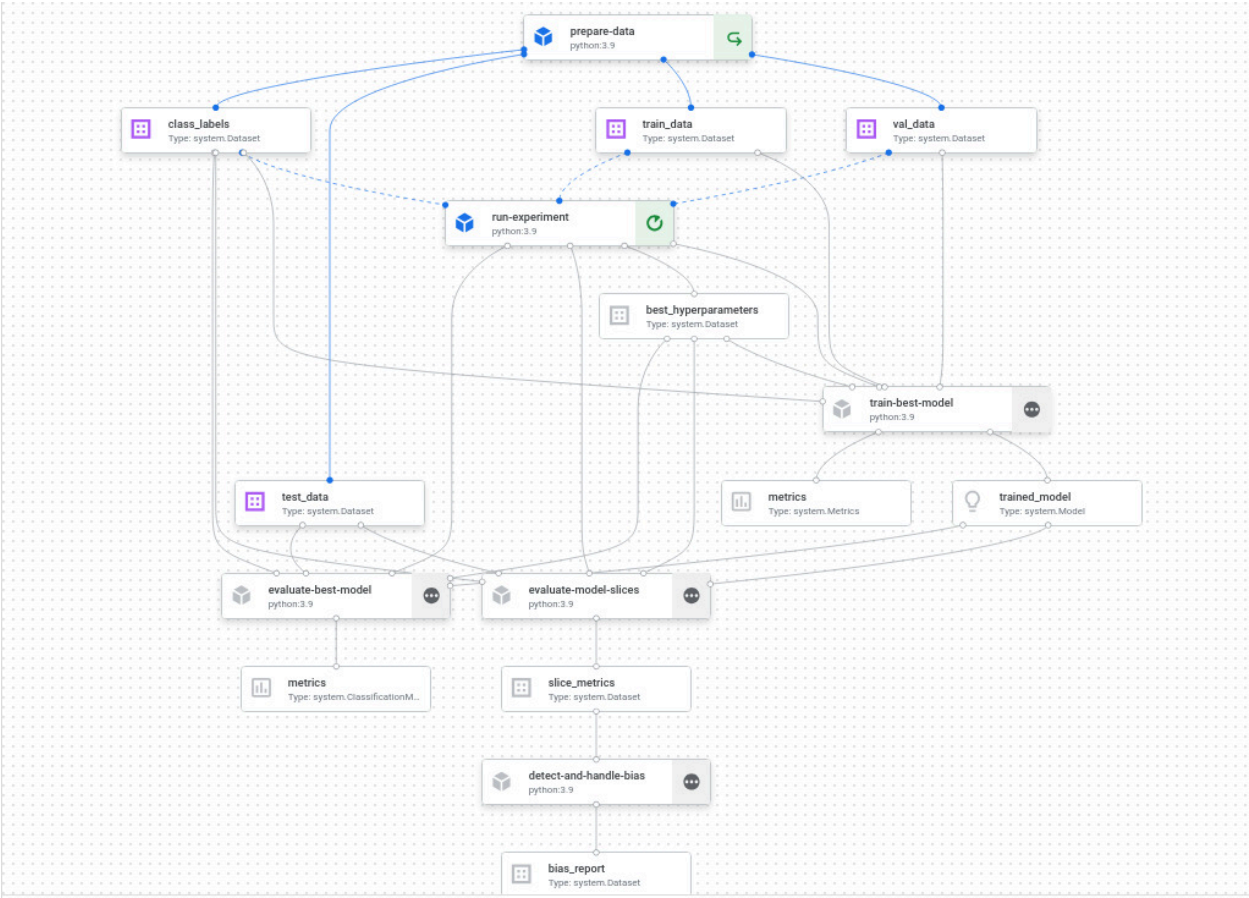
Objectives of the Model Pipeline

The key objectives of the pipeline include streamlining the data processing and model training process, optimizing model performance through hyperparameter tuning, ensuring model fairness, and automating model deployment.

2. Pipeline Overview

Pipeline Architecture

The pipeline is a sequential workflow consisting of multiple scripts, each dedicated to specific tasks like data preparation, model training, evaluation, bias detection, and deployment. The primary orchestrator script is *'pipeline.py'*, which triggers various scripts in sequence and monitors the progress.



VertexAI pipeline to use GPU compute on cloud

Tools & Frameworks Used

The pipeline utilizes several tools including **PyTorch**, **Transformers** library (BERT and RoBERTa), **Optuna** for hyperparameter tuning, **MLflow** for experiment tracking, **Google Cloud Storage** (GCS) for saving models, and **Flask** for serving predictions.

3. Data Ingestion

Data Loader (`data_loader.py`)

Loads the raw data from CSV files and applies preprocessing steps such as filling missing values, label encoding, and data transformation. The *SentimentDataset* class in this module formats the data for use by BERT/RoBERTa models.

Important Functions:

- *load_and_process_data(data_path):* Loads the data from the CSV file, fills missing values, converts price columns to numeric, and label-encodes the sentiment labels.
- *split_data_by_timestamp(df):* Splits the dataset into training, validation, and test sets based on the timestamp of reviews.
- *SentimentDataset Class:* This custom PyTorch dataset class processes text and additional features to create inputs suitable for BERT and RoBERTa models. It uses tokenization and handles features like price, helpful_votes, etc.

Detailed Steps:

- *Loading Data:* The CSV file containing raw labeled data is loaded using Pandas.
- *Handling Missing Values:* Missing values in text and title columns are replaced with empty strings, while price values are coerced to numeric values, with "unknown" replaced by None.
- *Feature Engineering:* Additional features such as price_missing are created to indicate if the price is missing. Other features like helpful_vote and verified_purchase are converted into numerical formats.
- *Label Encoding:* Sentiment labels are encoded into numerical values using LabelEncoder for model training.

Data Preprocessing (`prepare_data.py`)

Further processes data by balancing classes (using oversampling) and splits the dataset into training, validation, and test sets, which are saved as pickled objects.

Important Functions:

- *balance_data(df, column, min_samples):* Balances the data by oversampling underrepresented classes to ensure each slice has at least the minimum number of samples.
- *split_and_save_data(data_path, output_dir):* Splits the data and saves training, validation, and test sets as pickled objects.

Detailed Steps:

- *Balancing Data:* The data is balanced to ensure that each class has a minimum number of samples. Underrepresented classes are oversampled using the resample function from sklearn.
- *Splitting Data:* The dataset is split into training, validation, and test sets based on timestamps, ensuring a chronological order for generalizability. The split data is then saved as pickle files for future use.

4. Hyperparameter Tuning ('experiment_runner_optuna.py')

Hyperparameters like learning rate, batch size, weight decay, and dropout are tuned using **Optuna** to maximize the model's F1 score.

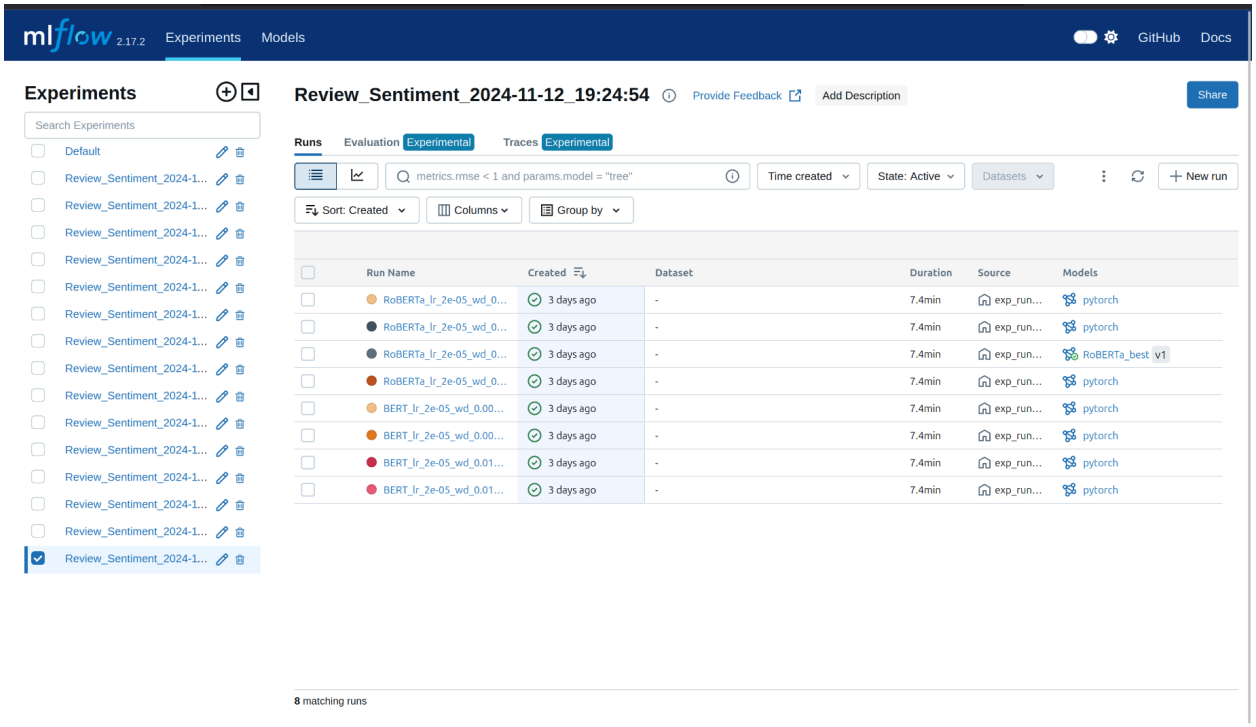
Important Functions:

- *objective(trial):* Defines the objective function for Optuna, which includes sampling hyperparameters, training the model, and evaluating its performance.
- *find_best_hyperparameters():* Runs the hyperparameter tuning process using Optuna and saves the best hyperparameters.

Detailed Steps:

- *Defining Search Space:* Hyperparameters are defined within a certain range to be explored by Optuna.
- *Optimization Objective:* The objective function evaluates model performance (F1 score) on the validation set for each set of hyperparameters.
- *Logging with MLflow:* Each trial's hyperparameters and corresponding metrics are logged using MLflow for future reference.

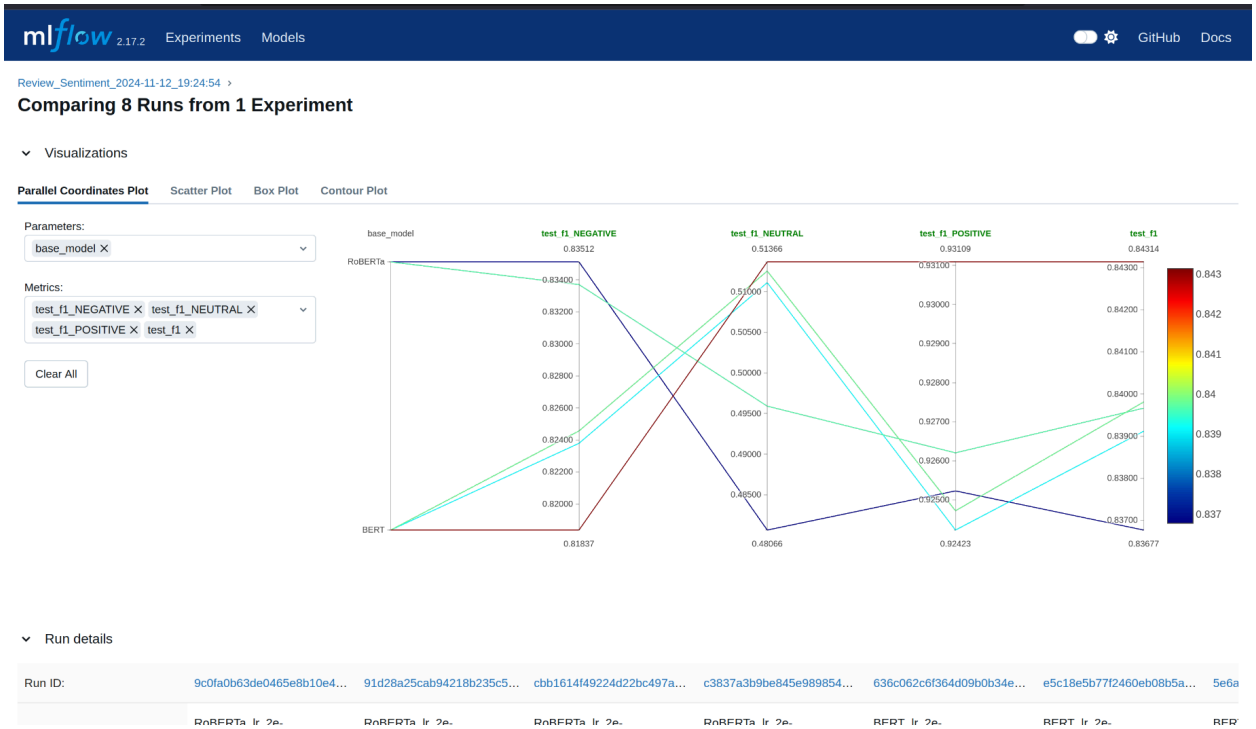
MLflow Logging in the Workflow



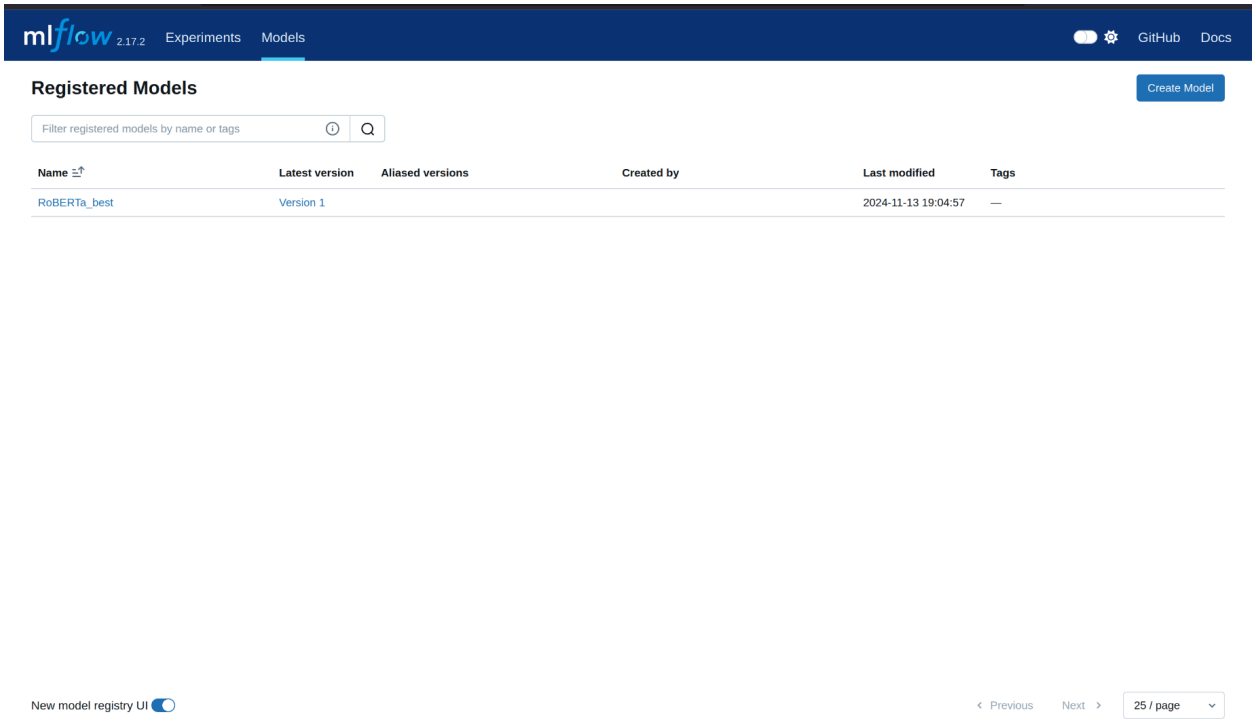
MLflow is integrated into this workflow to enable comprehensive tracking and management of model experiments. Here's how MLflow is used:

1. **Experiment Setup**:
    - A unique experiment is created using `mlflow.set_experiment()` with a timestamped name, ensuring clear organization of runs for this specific task.
2. **Parameter Logging**:
    - During each training run, key hyperparameters such as `model_name`, `learning_rate`, `batch_size`, `num_epochs`, `weight_decay`, and `dropout_rate` are logged using `mlflow.log_params()`. This makes it easy to trace how different configurations affect model performance.
3. **Metric Logging**:
    - After evaluating the model on the test dataset, metrics like `test_f1` are logged using `mlflow.log_metric()`. These metrics provide a quantitative measure of the model's performance.
4. **Run Management**:
    - Each training and evaluation run is tracked as a separate entity within the experiment. The runs are uniquely identified by descriptive names, such as `BERT_lr_1e-5_wd_1e-4_do_0.1`.
5. **Best Model Tracking**:
    - The best-performing model and its associated hyperparameters are logged as the "best run." This allows easy identification and reuse of the optimal configuration.
6. **Integration with Optuna**:
    - As Optuna performs hyperparameter optimization, each trial's results are logged to MLflow, providing a unified view of all experiments and their outcomes.
7. **Artifact Logging**:
    - Model artifacts, like trained weights or configuration files, are logged to MLflow, enabling seamless retrieval and deployment of models.

Comparing the models:



Registered best model artifact:



5. Model Selection & Training

Once the experiments are run through ***mlflow*** for various model architecture and model parameters, the best model hyperparameters with the highest F1 score on the test dataset are saved as an json file. This file is referenced to train the final model.
Example run hyperparam:


```
{
   "model_name": "BERT",
   "learning_rate": 0.00028630244431857374,
   "batch_size": 64,
   "num_epochs": 4,
   "weight_decay": 0.00016445644670785872,
   "dropout_rate": 0.40669162857535246
}
```


Model Architectures Used

BERT (`bert_model.py`)

A custom BERT-based model designed for sentiment analysis, enhanced with additional features like price, helpful votes, and verified purchase status to improve prediction accuracy.

Custom Model Details

- **Base Model**:
  The model is built upon `BertForSequenceClassification` from Hugging Face Transformers.
- **Additional Layers**:
  - A **linear layer** processes the numerical features (`price`, `helpful votes`, `verified purchase`), transforming them into a feature vector of the same size as BERT's pooled output.
  - The transformed feature vector is then **concatenated** with BERT's pooled output.
  - A **final dense layer** (classifier) processes the concatenated output to produce class probabilities.
- **Training Configuration**:
  The model is trained using a cross-entropy loss function. Hyperparameters like learning rate, batch size, and number of epochs are optimized via hyperparameter tuning.

Important Functions

1. **CustomBERTModel Class**:
   Defines the custom BERT model architecture with additional layers to handle numerical features.
2. **initialize_bert_model(num_labels)**:
   Initializes the BERT model, loads pre-trained weights, and adds the custom layers.
3. **train_bert_model(...)**:
   Trains the model using Hugging Face's Trainer API, enabling flexible training configurations and argument customization.

RoBERTa (`roberta_model.py`)

A custom RoBERTa-based model designed for the same sentiment classification task, with support for additional numerical features similar to the BERT model.

Custom Model Details

- **Base Model**:
  The model uses `RobertaForSequenceClassification` as the base architecture.
- **Additional Layers**:
  - A **linear layer** processes numerical features (e.g., `price`, `helpful votes`, `verified purchase`).
  - The processed features are **concatenated** with RoBERTa's pooled output.
  - A **final dense layer** outputs class probabilities based on the concatenated features.
- **Training Configuration**:
  Similar to the BERT model, the training uses cross-entropy loss with configurable hyperparameters like learning rate, batch size, and epochs.

Important Functions

1. **CustomRoBERTaModel Class**:
   Defines the custom RoBERTa model architecture, including layers for additional feature processing.
2. **initialize_roberta_model(num_labels)**:
   Initializes the RoBERTa model, loads pre-trained weights, and adds the feature integration layers.
3. **train_roberta_model(...)**:
   Trains the RoBERTa model using the Trainer API with customizable training arguments.

Summary of Custom Additions

Both models (BERT and RoBERTa) are augmented with:

- A **linear transformation layer** to handle additional numerical features.
- **Concatenation of features** with the pooled output from the pre-trained base models.
- A **dense classifier layer** for final predictions.

These modifications ensure the models leverage both textual and numerical data, improving their performance on domain-specific sentiment analysis tasks.

Why BERT and RoBERTa are Good for This Use Case:

BERT and RoBERTa are great choices for this sentiment analysis task because they are designed to understand the context of words in a sentence, which is really important when analyzing customer reviews. For example, in a review, the same word can have a different meaning depending on how it's used, and these models excel at capturing those nuances.

Both BERT and RoBERTa come pre-trained on massive amounts of text data, so they already have a strong understanding of language. This means we don't need to collect and train on a huge dataset ourselves, which saves a lot of time and resources. We can just fine-tune them specifically for our sentiment analysis task.

What makes them even better for our use case is that we can customize them. For instance, in addition to processing text, we can add features like the price of the product, whether the review was marked as helpful, or if the purchase was verified. These extra features help the model make even more accurate predictions.

Finally, both models are highly reliable and have been proven to perform really well on similar tasks, so we can trust them to deliver strong results. In short, BERT and RoBERTa give us the best of both worlds: a deep understanding of language and the flexibility to tailor the models to our specific needs.

Training Process (`train.py`, `train_save.py`, `roberta_train.py`, `train_save_model.py`)

Important Functions:

- *train_model(...):* Trains the BERT or RoBERTa model using specified hyperparameters.
- *evaluate_on_test_set(...):* Evaluates the model on the test set and calculates performance metrics like accuracy, precision, recall, and F1 score.
- *train_and_save_final_model(...):* Loads the best hyperparameters, trains the final model, and saves it locally and optionally uploads it to Google Cloud Storage.

Detailed Steps:

- *Data Preparation:* The data is loaded, preprocessed, and tokenized using either BERT or RoBERTa tokenizer.
- *Model Initialization:* Depending on the configuration, either a BERT or RoBERTa model is initialized with additional feature layers.
- *Training Setup:* Training arguments such as learning rate, batch size, number of epochs, and weight decay are defined using TrainingArguments.
- *Training Execution:* The model is trained using the Trainer class, which handles optimization, evaluation, and logging.
- *Saving Models:* The trained model is saved both locally and optionally uploaded to Google Cloud Storage for further use.

6. Model Evaluation

Evaluation Metrics (`evaluate_model.py`)

Evaluation is performed using accuracy, precision, recall, and F1 score.

Important Functions:

- *evaluate_model(model, test_dataset, label_encoder):* Evaluates the model on the test dataset and computes various metrics.
- *compute_metrics(eval_pred):* Computes accuracy, precision, recall, and F1 score for the given predictions.

Detailed Steps:

- *Loading Test Data:* The test data is loaded from the pickle files.
- *Evaluation Execution:* The trained model is evaluated on the test data, generating predictions.
- *Metric Computation:* The predictions are compared with the true labels, and metrics are computed using sklearn's metrics functions

7. Model Bias Detection and Mitigation

Evaluation Across Slices (`evaluate_model_slices.py`)

The model is evaluated across different slices (e.g., by year or category) to identify if it performs differently across specific subgroups.

Important Functions:

- *evaluate_slices(data, model, tokenizer, data_path):* Evaluates the model performance across different slices of the dataset.
- *evaluate_dataset(data, model, tokenizer):* Evaluates a single dataset slice and computes metrics.

Detailed Steps:

- *Slice Definition:* The data is split into slices based on columns such as year and main_category.
- *Slice Evaluation:* Each slice is evaluated separately to compute metrics, which are compared to the full dataset's metrics to detect discrepancies.

Bias Detection (‘bias_detect.py’)

Bias is detected by evaluating the model across different slices of the dataset.

Important Functions:

- *detect_bias(slice_metrics_path):* Loads the slice metrics and identifies potential bias by comparing slice metrics to overall metrics.

Detailed Steps:

- *Loading Metrics:* Metrics for different slices are loaded from a CSV file.
- *Defining Bias Criteria:* Bias is flagged if a slice's F1 score is significantly lower than the overall F1 score, and if the slice has enough samples to be statistically significant.
- *Logging Biased Slices:* Any biased slices are logged with details about the slice and its corresponding metrics.

The sentiment analysis pipeline, consisting of six key steps (data preparation, hyperparameter tuning, model training, model evaluation, slice-based evaluation, and bias detection), has been successfully deployed on Google Cloud's Vertex AI platform using Kubeflow Pipelines. The pipeline is compiled into a JSON file and submitted as a PipelineJob to Vertex AI, leveraging Google Cloud services and MLflow for experiment tracking. This deployment enables seamless execution of the entire workflow in a cloud environment, allowing for efficient scaling and integration with other cloud-based components as they are migrated to the cloud infrastructure.

8. CI/CD Pipeline Automation

Pipeline Integration (‘pipeline.py’)

This script orchestrates the entire model training pipeline.

Logging Setup

Logging is configured using Python's built-in logging module. The pipeline.log file records all significant events, including successes and errors, throughout the pipeline run. This setup is crucial for debugging and tracking pipeline progress, ensuring that each step is transparent and traceable.

Flag File Paths and Retrigger Logic

Flag files (FLAG_PATH_F1 and FLAG_PATH_BIAS) are used to prevent infinite retriggering of specific steps in the pipeline. These flags ensure that the retriggering process only happens when necessary, preserving computational resources and preventing redundant retraining.

Important Functions:

- *run_script(script_name):* Executes a specific script as part of the pipeline and captures its output.
- *restart_pipeline_from_experiment_runner():* Retriggers the pipeline from the experiment tuning step if performance is below the threshold.
- *extract_f1_score(log_output):* Extracts the F1 score from the logs to determine whether the pipeline should be retriggered.

Detailed Steps:

- *Step-by-Step Execution:* The script sequentially executes prepare_data.py, experiment_runner_optuna.py, train_save.py, evaluate_model.py, evaluate_model_slices.py, and bias_detect.py.
- *Failure Handling:* If any step fails, appropriate logs are generated, and the pipeline stops to prevent further errors.
- *Retrigger Logic:* If the F1 score falls below a defined threshold, the pipeline retriggers the hyperparameter tuning step to improve performance.

Continuous Integration for Model Training (`pipeline.py`)

The pipeline can be rerun whenever new data is available or modifications are made, ensuring consistency.

Running Scripts as Subprocesses (run_script())

The run_script() function is used to run each part of the pipeline as a subprocess using subprocess.Popen(). This function captures the output, both stdout and stderr, and logs it for reference. If a script completes successfully, it logs the success message, whereas any failure will be logged as an error, helping in tracking and debugging. This function supports the automation and seamless execution of various pipeline components.

Retriggering the Experiment Runner (restart_pipeline_from_experiment_runner())

If the extracted F1 score falls below a defined threshold, the restart_pipeline_from_experiment_runner() function retriggers the experiment runner (experiment_runner_optuna.py). This function is crucial for continuous optimization of the model, ensuring that only well-performing models proceed through the pipeline. After retriggering, the function continues the pipeline, resuming subsequent steps like model saving and evaluation.

Continuing the Pipeline (process_pipeline())

The process_pipeline() function ensures that the pipeline continues from a specific point if necessary. The steps include retraining (train_save.py), evaluation (evaluate_model.py and evaluate_model_slices.py), and bias detection (bias_detect.py). If any of these steps fail, the pipeline halts, allowing for troubleshooting. This function provides flexibility to pick up the pipeline from a specific stage, contributing to efficient and automated model development.

Automated Model Validation & Bias Detection

The pipeline automatically validates model performance and detects bias, with retrigger mechanisms in place to address issues.

9. Model Deployment

Pushing to Model Registry (`train_save_model.py`)

The final trained model is saved locally and can be uploaded to Google Cloud Storage.

Important Functions:

- *train_and_save_final_model(hyperparameters):* Trains the model with the best hyperparameters and saves the state dictionary.

- *upload_to_gcs(bucket_name, source_file_name, destination_blob_name):* Uploads the trained model to Google Cloud Storage for easy access.

Detailed Steps:

- *Saving Model Locally:* The model's state dictionary is saved in .pth format.
- *Uploading to GCS:* The saved model is uploaded to Google Cloud Storage for deployment, enabling easy access for inference.

Deployment Steps (‘app.py’)

The app.py script is a Flask-based service that serves the model through a REST API.

Important Functions:

- *load_model():* Loads the trained model from the local filesystem and prepares it for inference.
- *predict():* Defines the /predict endpoint, accepting POST requests and returning sentiment predictions.
- *health():* Provides a /health endpoint to check the server status.

Detailed Steps:

- *Loading Model:* The trained model is loaded from a local file.
- *Prediction Endpoint:* A /predict endpoint is defined, which accepts POST requests containing text data and returns sentiment predictions.

Health Check Endpoint: A /health endpoint is provided to ensure the server is running properly.

Docker Containerization for Registry (`Dockerfile`)

For reproducibility and scalability, the pipeline is containerized using Docker. This ensures consistent environments across different deployments.

Scripts Overview

Below is an overview of key scripts and their roles:

- ***pipeline.py:*** Orchestrates the entire pipeline, manages retriggering in case of failures or low metrics.
- ***train.py, train_save.py, train_save_model.py:*** Handle different stages of training, saving, and deployment preparation for the models.
- ***experiment_runner.py, experiment_runner_optuna.py:*** Run hyperparameter tuning experiments, track results, and identify the best model.

- ***evaluate_model.py, evaluate_model_slices.py:*** Perform model evaluation and assess performance across different dataset slices.
- ***bias_detect.py:*** Detects bias by analyzing the performance discrepancies among different slices.
- ***app.py:*** Implements the REST API for serving model predictions.

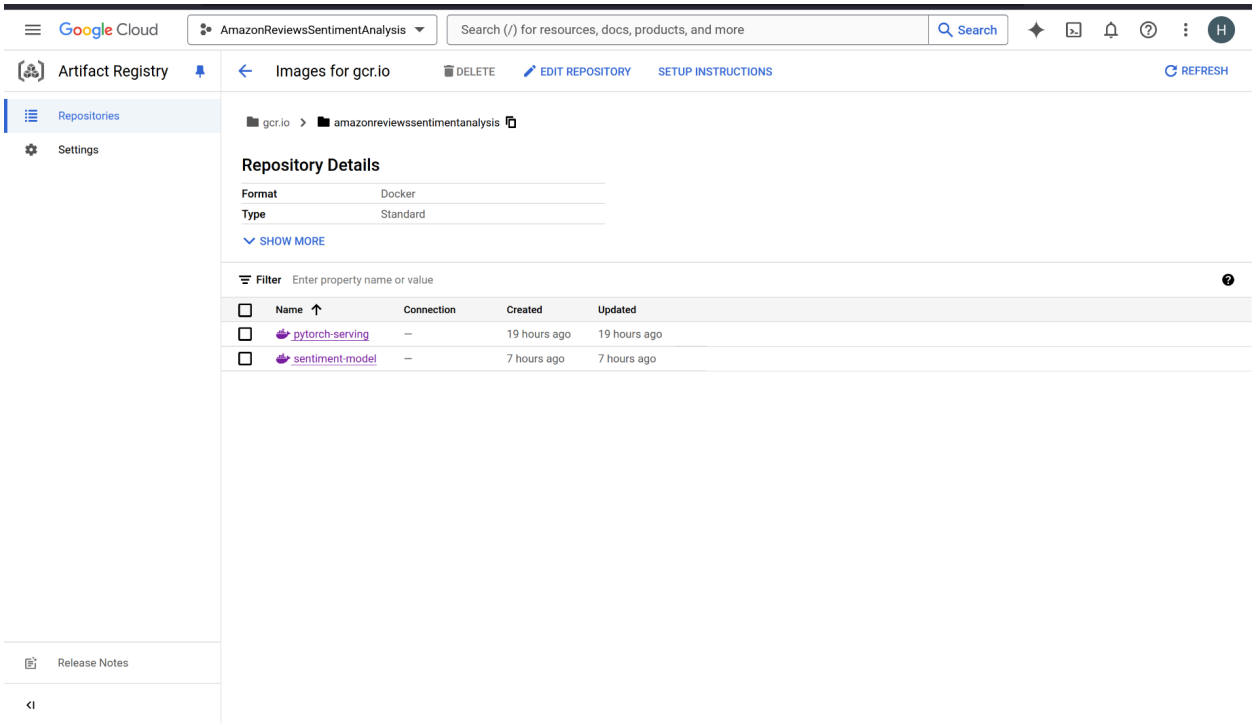
Containerization and Deployment to GCR

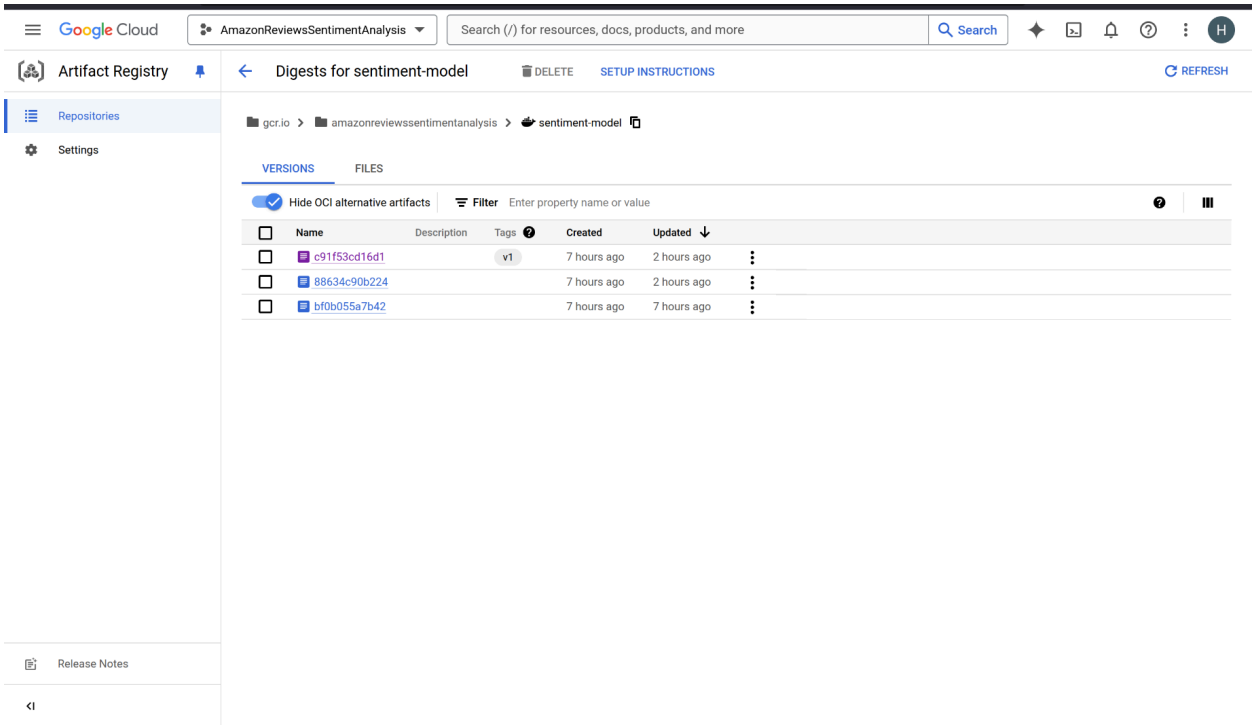
To make the application portable and scalable, it is containerized using **Docker** and pushed to **Google Container Registry (GCR).** The container includes the trained model, the Flask application, and all required dependencies.

Key Steps

1. **Containerization**:
    - The application is packaged into a Docker container, which includes the Flask API and the model.
2. **Pushing to GCR**:
    - The container is uploaded to GCR, making it accessible for deployment on Google Cloud services like Kubernetes or Compute Engine (Model Deployment Phase with integrated KubeFlow cloud pipelines)

For details on the containerization process, refer to the Dockerfile provided in the project.


Google Cloud
AmazonReviewsSentimentAnalysis
Search (/) for resources, docs, products, and more
Search
Artifact Registry
Images for gcr.io
DELETE
EDIT REPOSITORY
SETUP INSTRUCTIONS
REFRESH
Repositories
Settings
gcr.io
amazonreviewssentimentanalysis
Repository Details
Format
Docker
Type
Standard
SHOW MORE
Filter
Enter property name or value
Name
Connection
Created
Updated
pytorch-serving
–
19 hours ago
19 hours ago
sentiment-model
–
7 hours ago
7 hours ago
Release Notes


Google Cloud
AmazonReviewsSentimentAnalysis
Search (/) for resources, docs, products, and more
Search
Artifact Registry
Digests for sentiment-model
DELETE
SETUP INSTRUCTIONS
REFRESH
Repositories
Settings
gcr.io
amazonreviewssentimentanalysis
sentiment-model
VERSIONS
FILES
Hide OCI alternative artifacts
Filter
Enter property name or value
Name
Description
Tags
Created
Updated
c91f53cd16d1
v1
7 hours ago
2 hours ago
88634c90b224
7 hours ago
2 hours ago
bf0b055a7b42
7 hours ago
7 hours ago
Release Notes

12. RAG Pipeline

Overview

The RAG (Retrieval-Augmented Generation) pipeline is designed to support the summarization of review data, running simultaneously with the model pipeline. This section provides detailed documentation on the RAG pipeline, including the file structure, key functions, data flow, and how it integrates with the overall system. The primary objective of the RAG pipeline is to retrieve relevant aspects of the data, generate summaries, and store embeddings, which can later be used for question-answering tasks or further summarization. The RAG pipeline is implemented using Apache Airflow for orchestration.

This pipeline for summary generation for each category runs only once a month, generating summaries and moving to the next step. This helps stakeholders get a monthly analysis of the activities in the previous month, making it easy to track different sub-categories' performance for each month. Moreover, these summaries are created to provide detailed insights into various aspects of each category, beyond just the final sentiment of all the summaries for that sub-category.

File Structure

- **dags/**
    - **utils/:** This folder contains the core utility files for RAG processing, including document processing, embedding generation, and summarization scripts.
        - **config.py:** Contains configuration constants, such as file paths for processed data and document storage.
        - **document_processor.py:** Handles the extraction of text and summarization of review documents.
        - **embeddings.py:** Generates and stores document embeddings using Sentence Transformers and Pinecone.
        - **extracting_unique_aspects.py:** Extracts unique aspects or categories from labeled review data.
        - **load_utils.py:** Utility functions for loading prompt templates and splitting text into manageable chunks.
        - **main.py:** Entry point script for the RAG pipeline, invoking various data processing and embedding generation steps.
        - **review_data_processing.py:** Processes review data, performs aggregation, and prepares documents for further summarization and embedding.

        - **summarization.py:** Summarizes the aggregated data using a hierarchical summarization approach with a pre-trained BART model.
        - **summary_generator_main.py:** Processes documents using the summarizer and saves the results to a refined JSON file.
    - **rag_data_preprocessing_dag.py:** Defines an Apache Airflow DAG to orchestrate the RAG data preprocessing pipeline.
- **Dockerfile** and **docker-compose.yaml:** Define the Docker environment and Airflow services for running the RAG pipeline.

Configuration (config.py)

The config.py file provides paths for accessing and storing processed data throughout the pipeline, including paths for raw document storage, refined output, and prompt templates used during summarization. It ensures that the rest of the pipeline components consistently reference the same data locations.

Document Processing (document_processor.py)

- ***Prompt Template Loading:*** The load_prompt_template function reads the prompt template used to generate coherent prompts for OpenAI's GPT models. The prompts are key for ensuring the generation model receives consistent context.
- ***Text Splitting:*** The split_text function divides the input text into smaller chunks, manageable for the OpenAI API, ensuring each chunk is within the token limit (12,000 tokens).
- ***Text Analysis:*** analyze_chunk_with_summary takes text chunks and interacts with GPT to provide summaries for each. This function forms the core summarization logic for each document.
- ***Document Processing:*** The process_document_with_summary function coordinates summarizing large documents by splitting them into chunks, summarizing each chunk, and combining them into a final comprehensive summary.

Embeddings (embeddings.py)

- ***Document Loading:*** The load_documents_from_json function reads the previously processed documents from a JSON file.
- ***Embedding Generation:*** The generate_embeddings function leverages a pre-trained model (all-MiniLM-L6-v2) to create vector embeddings for each document, which are essential for efficient similarity-based retrieval.

- ***Pinecone Indexing:*** The initialize_pinecone function sets up Pinecone for storing embeddings, while upsert_embeddings_to_pinecone uploads these embeddings to Pinecone. This allows for scalable and fast similarity searches during the RAG process.

Data Preprocessing (review_data_processing.py)

- ***Data Loading:*** load_and_process_data loads review data from CSV files and performs basic preprocessing, such as extracting months and years from review timestamps and dropping unnecessary columns.
- ***Data Aggregation:*** The aggregate_data function aggregates reviews by key attributes like year, month, and extracted category, and computes metrics such as average rating, helpful votes, and sentiment.
- ***Document Preparation:*** prepare_documents converts aggregated data into a list of dictionary-style documents, each representing a summary-ready unit of review data.
- ***Document Saving:*** save_documents saves these documents to JSON format, providing an input for further summarization and embedding generation.

Summary Generation (summary_generator_main.py)

- ***Document Processing:*** The process_document_with_summary function is utilized in this script to generate summaries for each document. The summary includes detailed chunk-level and overall summaries for each document. The script uses the GPT-4o-mini model from OpenAI to generate these chunks, which was chosen as the most cost-effective solution compared to other models like Gemini.
- ***Cleaning and Validation:*** The clean_and_validate_json function handles formatting issues with the JSON content of each document summary, ensuring compatibility with downstream components.
- ***Saving Results:*** Summarized documents are saved into intermediate (processed_output_path) and refined JSON files (refined_output_path).

RAG Data Preprocessing DAG (rag_data_preprocessing_dag.py)

The DAG script orchestrates the various stages of the RAG pipeline using Apache Airflow:

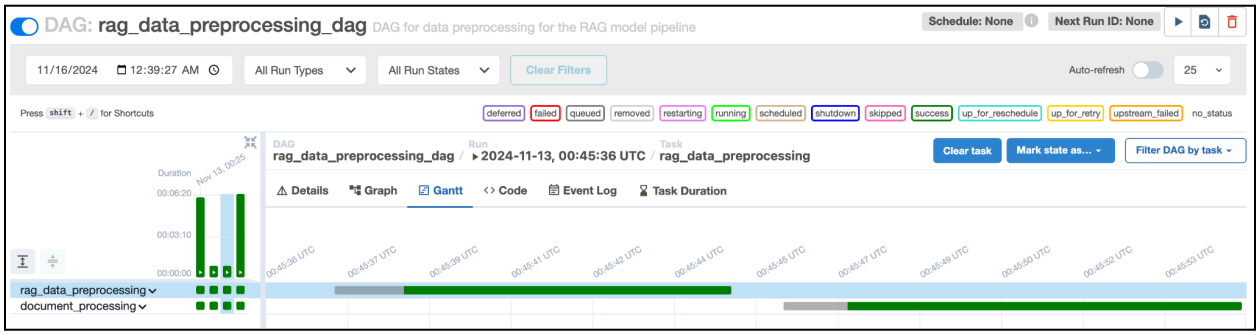
- ***RAG Data Preprocessing Task Group:*** Handles data loading, processing, aggregation, and document preparation tasks, ensuring the raw reviews are structured into aggregated JSON documents.
    - load_and_process_data_task, aggregate_data_task, prepare_documents_task, and save_documents_task work sequentially to transform raw data into summarized, structured JSON files.

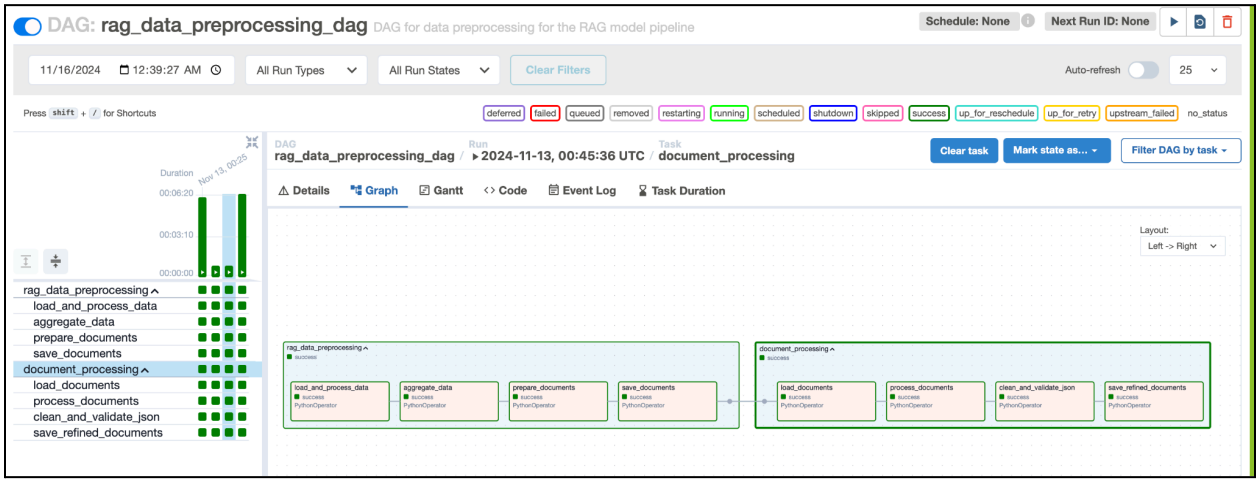
- ***Document Processing Task Group:*** Summarizes, cleans, and validates the document data.
    - process_documents_task generates summaries for each document, while clean_and_validate_json_task ensures each summary is correctly formatted and valid JSON.

Docker Integration

- ***Dockerfile:*** The Dockerfile is used to create a custom Docker image, extending the base Apache Airflow image. It sets up the required directories, installs necessary dependencies from requirements.txt, and copies Airflow DAGs, configuration, and test scripts.
- ***docker-compose.yaml:*** Defines the services required to run the Airflow environment for the RAG pipeline. It includes containers for the Airflow webserver, scheduler, worker, PostgreSQL database, and Redis for managing the message broker. It also exposes the necessary ports for the Airflow UI.

The Docker setup ensures that the entire RAG pipeline is containerized, making it easy to deploy, replicate, and manage the pipeline in different environments.





13. Conclusion

Summary of Pipeline Capabilities

This model pipeline provides a comprehensive framework for training, tuning, evaluating, and deploying machine learning models. It integrates state-of-the-art tools like BERT/RoBERTa, Optuna, MLflow, and Flask, ensuring the model's efficiency, fairness, and usability.

Summary of RAG Pipeline

The RAG pipeline complements the model pipeline by processing and summarizing reviews, generating embeddings for efficient retrieval, and organizing data for enhanced usability. It incorporates modern NLP models like BART for summarization and uses Pinecone for efficient embedding storage and retrieval, all orchestrated with Apache Airflow. This integration provides a powerful system for extracting insights from large volumes of review data, summarizing them, and making them available for downstream analysis or model consumption.