Name- Puspak Chakraborty, Roll No. DA24S002

High Level Design: Doodle Classifier System

This document provides a comprehensive high-level design for the Doodle Classifier.

Overview

The Doodle Classifier is a web-based application that allows users to draw doodles, classify them using a machine learning model, and save the drawings with their associated categories.

Along with the front end, backend and inference api, it also features a retraining pipeline, which takes care of updating the underlying model as and when enough data is accumulated.

System Architecture

The system consists of several interconnected components:

1. **User Interface (UI)**: A web frontend for users to draw doodles and view classification results.
2. **FastAPI Backend**: REST API service that processes requests, handles image data, and communicates with the ML model hosted by MLFlow.
3. **MLflow Model Server**: Hosts the latest trained machine learning model for doodle classification with help of MLFlow.
4. **Monitoring Stack**: Prometheus and Grafana for system metrics collection and visualization

UI Component

- **Port**: 9080
- **Responsibilities**:
    - Provide drawing interface for users.
    - Send doodle data to backend for classification
    - Display classification results.
    - Send image data if required to backend for further training.

FastAPI Backend

- **Port**: 8000
- **Dependencies**: MLflow Model Server
- **Key Endpoints**:
    - /save-image/: Saves user-drawn images with category labels
    - /invocations/: Forwards prediction requests to the MLflow model
    - /metrics: Exposes metrics which are ingested by Prometheus
- **Responsibilities**:
    - Process base64-encoded image data
    - Save images to the appropriate category folder
    - Forward prediction requests to the MLflow model
    - Provide metrics for monitoring
    - Handle errors and logging

MLflow Model Server

- **Port**: 5002
- **Endpoint**: /invocations/
- **Responsibilities**:
    - Host the latest trained doodle classification model
    - Process prediction requests
    - Return classification results

Monitoring Stack

Prometheus

- **Port**: 9090
- **Responsibilities**:
    - Collect metrics from the FastAPI backend and windows exporter running in host machine and make them available to Grafana

Grafana

- **Port**: 3000
- **Dependencies**: Prometheus
- **Responsibilities**:
    - Visualize metrics from Prometheus
    - Provide dashboards for system monitoring

Data Flow

1. **Image Classification Flow**:
    - User draws a doodle on the UI
    - UI encodes the image as base64 and sends to backend
    - Backend forwards the encoded image to model hosted by MLflow
    - Model returns top classification results
    - Results are displayed to the user
2. **Image Saving Flow**:
    - If user is not satisfied with top prediction, user submits a doodle with a category
    - UI sends the image and category to backend
    - Backend decodes the image and saves it to the appropriate category folder
    - Confirmation is returned to the UI
3. **Metrics Flow**:
    - Backend tracks metrics (requests, latency)
    - Prometheus scrapes metrics from the backend and host machine
    - Grafana visualizes the metrics from Prometheus

Metrics and Monitoring

The system tracks several key metrics:

- **Prediction Requests**: Count of model prediction requests
- **Prediction Latency**: Time taken for predictions

- **Api Errors**: Errors occurring in the two apis
- **Save Image Requests**: Count of image save operations
- **Save Image Latency**: Time taken for image saving.
- **System Statistics**: CPU Usage, RAM Usage, Disk IO, Network IO

The system is containerized using Docker and orchestrated with Docker Compose:

- All components mentioned in System Architecture (other than MLFlow server) run in separate containers
- Entire architecture can be built in one go using Docker compose
- Inter-container communication is configured
- Volumes are used for persistent storage
- Environment variables configure component behaviour

Design of Pipeline used for retraining

1. Pipeline Architecture Overview

The retraining pipeline is orchestrated as a sequence of modular Python scripts, each representing a distinct stage. Execution and status tracking are managed by a pipeline orchestrator and a PostgreSQL database. The stages are:

2. Component Roles and Responsibilities

2.1 Orchestration and Tracking

- **pipeline_orchestrator.py**:
    - Sequentially executes each stage as a subprocess.
    - Tracks run and stage statuses in the PostgreSQL database using functions from db_util.py.
    - Handles failure, success, and conditional stage skipping based on data thresholds and errors.

2.2 Data Ingestion and Preparation

- **scan_file_count.py**:
    - Scans the user-uploaded image directory, counts new images, and logs category statistics.
    - Used to determine if enough new data exists to trigger retraining.
- **move_preexisting_new_data_to_old_data.py**:
    - Moves or merges previously uploaded data into old dataset, makes space for upcoming images to be converted to npy files as new data for testing.
    - Cleans up new data directory after merging.
- **convert_image_to_npy_delete_png.py**:
    - Converts newly uploaded PNG images to .npy arrays and saves them to npy_data/uploaded_data/new_data folder to ensure that these are the ones that get picked during retraining.
    - Deletes original PNGs after conversion.

2.3 Data Versioning

- **version_data.py**:
    - Uses DVC and Git to add, commit, and track changes made to the training dataset.
    - Ensures reproducibility and data provenance for each model version.

2.4 Model Retraining

- **retrain_model.py**:
    - Loads the updated dataset, splits into train/validation sets, and prepares data loaders.
    - Loads the previous best model, unfreezes the last layer, and retrains using new data.
    - Logs training and validation metrics, saves the best model checkpoint.

2.5 Model Registration

- **register_latest_model.py**:
    - Registers the newly trained model in the model registry, and tags it appropriately to ensure the MLFlow server picks up the latest tagged model.
    - Enables deployment and version management.

2.6 Data Hygiene

- **clean_empty_images_folders.py**:
    - Removes empty category folders to keep the data directory clean.

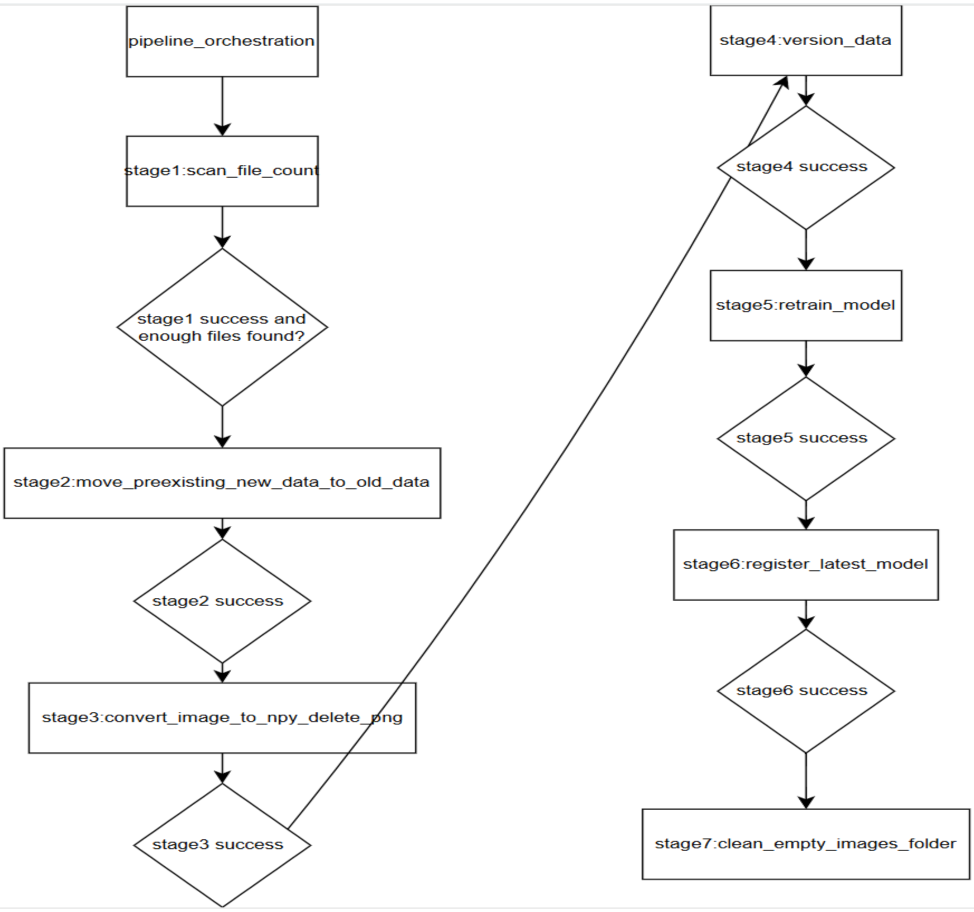

Fig: Retraining pipeline

3. Monitoring, Logging, and Error Handling

- **Logging**: Each script logs to a stage-specific log file (e.g., stage1.log, stage5.log) for debugging and auditability.
- **Database Tracking**: Pipeline run and stage statuses, start/end times, and failures are tracked in PostgreSQL for monitoring and dashboarding, and a separated UI screen is made available to track these pipeline runs.
- **Failure Handling**: On failure, the orchestrator marks the current and all subsequent stages as failed and halts the pipeline.
- **Conditional Execution**: If enough new data isn't present, the pipeline marks subsequent stages as 'not_triggered' and exits gracefully.

4. Versioning and Reproducibility

- **Data Versioning**: DVC and Git are used to track changes in the dataset after each retraining cycle, ensuring reproducibility.
- **Model Versioning**: Trained models are saved with unique identifiers and registered for deployment and rollback.

<u>Overall high level design diagram</u>