Final Project Report

CS5830 : Big Data Laboratary
Image Classification

By
R S Harish Krishnan - ME20B081
R Prithiviraj - ME20B136
B Srivathsan - ME20B176

Mar 2024

Contents

1 Problem Statement

To integrate various MLOps tools, including Data Version Control (DVC), MLflow, FastAPI, Prometheus, Grafana, Docker, and Git, to facilitate an end-to-end machine learning model development pipeline.

The dataset for this task involves a 5-class image classification, trained on 2000 images and validated on 500 images. The test data consists of 500 images. The FastAPI application allows users to upload an image and receive the model's predicted class via the Swagger UI.

In the upcoming sections, a detailed description of the various steps involved in the development phase is provided, including Data Processing, Model Development, Model Tracking, Hosting the Model with FastAPI, Health Monitoring using Prometheus, Visualizations through Grafana, Containerization with Docker, and Version Control using Git.

2 Data proessing Pipeline

Data Version Control (DVC) was used to manage the data pipeline for the image classification task. The process involves several stages: unzipping the dataset, preprocessing the data, and evaluating the preprocessed data against the original for content structure. Each stage reads parameters from a YAML file, ensuring that the stages execute only when there are changes in their dependencies. This approach guarantees reproducible results and efficient workflow management. We also incorporate Git for version controlling the codebase, while DVC handles data tracking. This ensures that both code and data are versioned and tracked together, making sure they are both properly documented

2.1 Stages of the Pipeline

- **Unzipping the Dataset**
  - **Objective**: Extracts the dataset from a compressed file.
  - **Dependencies**: The compressed dataset file.
  - **Description**: The dataset(initially in a zip file) is unzipped to a specified directory. This stage will re-run only if the zip file changes.
- **Preprocessing the Data**
  - **Objective**: Prepares the images for the classification task. Used multiprocessing library available in python to run multiple processes simultaneously. This is an alternative to Apache Spark for the small dataset used in the task. Usage of Spark becomes justifiable for larger datasets only.
  - **Dependencies**: The unzipped dataset and preprocessing parameters (from a YAML file).
  - **Description**: This stage involves operations like resizing images, converting to tensor and mean-variance normalizing. Parameters controlling these operations are specified in a YAML file. If either the input data or the YAML parameters change, this stage will re-execute.
- **Evaluating Preprocessed Data**

- **Objective**: Compare the preprocessed data against the original data to ensure content structure integrity.
- **Dependencies**: The original dataset, preprocessed dataset, and evaluation parameters (from a YAML file).
- **Description**: This involves checking for consistency and structural integrity between the original and preprocessed datasets. Metrics and validation criteria are defined in the YAML file. This stage re-runs if there are changes in any of the dependent files or parameters.

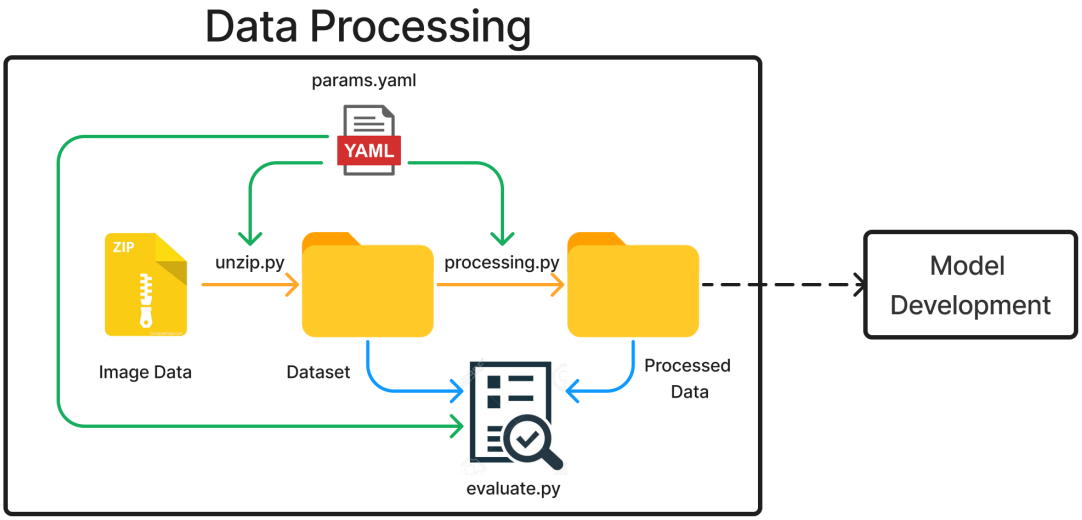

Figure 1: Data Processing Pipeline

2.2 YAML Configuration

A YAML file is used to specify parameters for each stage, such as preprocessing techniques, evaluation metrics, and other relevant settings. This file ensures that the stages are parameterized and flexible.

2.3 Dependency Tracking

- DVC monitors dependencies for each stage. If any dependencies (input data or parameters) change, DVC triggers the re-execution of the dependent stage.
- Git is used alongside DVC to track changes in the codebase, while DVC specifically tracks changes in data files and model artifacts if any.

2.4 Reproducibility and Version Control

- **Reproducibility**: By using DVC, each stage in the pipeline can be reproduced exactly. DVC ensures that the same dataset version and processing parameters are used, making the results consistent and reproducible.
- **Version Control**:

– **Git**: Used for tracking changes in the codebase, including scripts, notebooks, and configuration files.

– **DVC**: Manages data files, model weights, and other large files, ensuring they are versioned and tracked. This allows us to reproduce the exact environment and data state used in previous experiments.

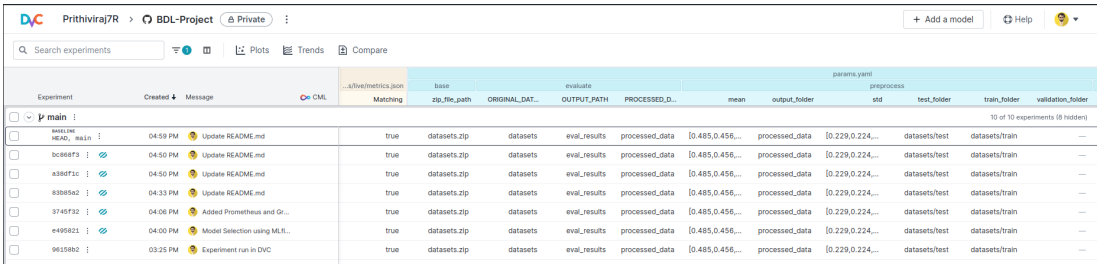

Figure 2: DVC Live Studio for tracking Experiments

3 Model Development and Tracking

3.1 Model Development

For this image classification task, a Convolutional Neural Network (CNNs) was employed. The model's architecture includes two convolutional layers with max pooling, with 2 fully connected layers for classification. Five model variants, differing in filter size, number of filters, padding, and batch normalization, were compared based on performance metrics.

3.2 mlflow tracking

All model parameters and metrics were recorded using MLflow, and the MLflow server was employed to visualize the results and compare the models through plots.

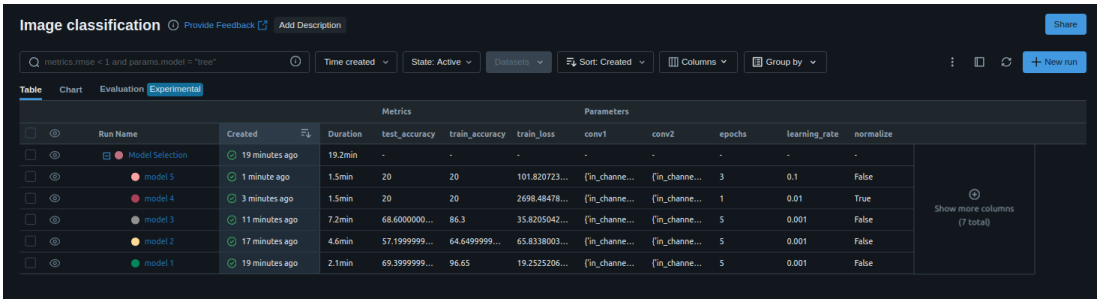

Figure 3: mlflow UI for model tracking

4 Model Deployment and Monitoring

4.1 Model Deployment using FastAPI

- **Model Selection**: The best model based on performance on the test data was chosen to be deployed.

- **FastAPI Application Design**:
  - A FastAPI application is designed to host the model's predictive capabilities.
  - The API endpoint accepts image uploads, processes the image, passes it through a pretrained CNN model, and returns the predicted class label.
- **Deployment**:
  - The application is hosted using Uvicorn.
  - The application is accessible via SwaggerUI for API documentation and testing.
  - Scalability: Designed to scale with additional instances of the FastAPI application using containerization tools like Docker.
  - Monitoring: Integrated logging and monitoring to track usage and performance metrics of the API.

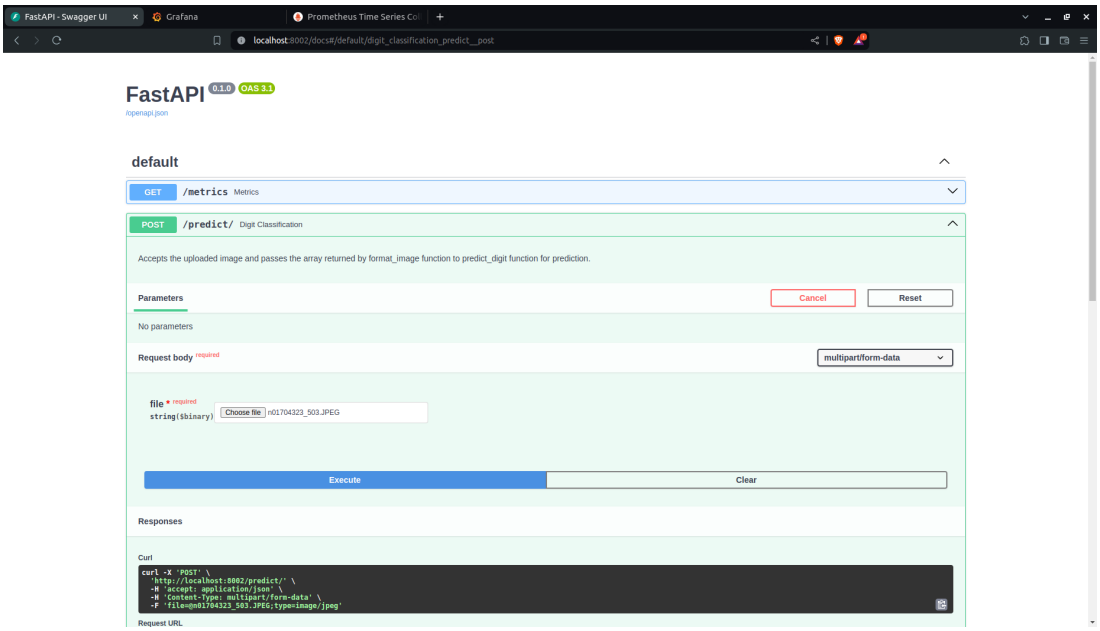

Figure 4: Swagger UI for uploading Images for prediction

4.2 Health Monitoring using Prometheus and Grafana

- **Instrumentation with Instrumentator**:
  - The application uses the Instrumentator to instrument and expose metrics.
  - Various Prometheus metrics are set up to monitor:
    - API usage
    - Runtime
    - Memory and CPU usage
    - Network I/O
- **Dashboard Visualization with Grafana**:

- Customized dashboards have been created using Grafana to visualize and analyze Prometheus metrics for the Rest API.
- The dashboards empower proactive monitoring and help troubleshoot issues.
- Scalability: Designed the monitoring setup to scale with the application, allowing additional metrics and dashboards to be added as needed.

Integrated Prometheus and Grafana setup with existing DevOps tools for streamlined operations and maintenance.

Figure 5: Dashboard for visualization of various metrics in Grafana

5 Docker: Containerization

- **Containerization of FastAPI Application**:
  - Docker was used to containerize the FastAPI application, ensuring a consistent and reproducible environment across different stages of development and deployment, with port mapping for supporting multiple ports.
- **Integration with Prometheus and Grafana**:
  - Alongside FastAPI, Prometheus and Grafana were integrated within the Docker container to collect and store real-time metrics, visualize these metrics, providing a comprehensive dashboard for real-time analytics and insights into the application's performance.
- **Streamlined Deployment Process**:
  - This containerized setup streamlined the deployment process, enhanced scalability, and ensured that monitoring and visualization tools were readily available and consistently configured across all environments.
- **Additional Benefits**:

- **Isolation**: Docker containers provide isolation, allowing the FastAPI application, Prometheus, and Grafana to run consistently across different stages of development and on different platforms.
- **Resource Efficiency**: Docker containers are lightweight and share the host system's kernel, leading to efficient resource utilization and enabling the deployment of multiple instances of the application on the same host.
- **Portability**: Docker containers encapsulate the application and its dependencies, making it easy to deploy and run the application on any platform that supports Docker.

6 Git: Version Control

- **Implementation with Git**:
  - Implemented version control using Git with a well-structured repository to include:
    * Data processing scripts with .dvc files
    * Model training code
    * Code to run the FastAPI application
    * Monitoring setups with Docker configurations
- **Documentation and Usage Steps**:
  - Proper documentation detailing methods and usage steps has been included in the repository.
  - Ensured each part of the project, from setup to execution, has been documented using GitHub.
- **Benefits of Version Control**:
  - **Collaboration**: Git enables collaboration among team members by allowing them to work on different parts of the project simultaneously and merge their changes seamlessly.
  - **History and Traceability**: Git maintains a detailed history of changes, making it easy to track modifications, revert to previous versions, and understand the evolution of the project over time.
  - **Backup and Recovery**: With Git, the entire project repository is stored on a remote server, providing a reliable backup and facilitating recovery in case of data loss or system failure.

The link for the repository is found below: GitHub