IMPRESS Low Level Documentation

HOHENSTEIN

Bönnigheim, Germany

Illia Rohalskyi
Dr. Igor Kogut
Elias Brohammer

Version 1: 11.12.2023

Contents

Chapter 1

Introduction

1.1 Purpose of Documentation

This document serves as a comprehensive guide for developers, offering clear insights into working with the project's codebase and associated tools. It aims to facilitate seamless installation, configuration, usage, and testing of the software. While not delving into the intricate details of each module, developers can rely on the docstrings provided for methods, functions, classes, and module-level explanations to grasp the functionality of specific components. For a general overview of the software, please refer to High Level Documentation

1.2 Audience

Primarily designed for developers actively engaged in the project, this documentation caters to those joining the team, collaborating, or maintaining the software. It equips developers with the necessary guidance to navigate the codebase, comprehend the tools, and contribute effectively.

Chapter 2

Installation

To get started with this project, please follow these installation steps. Ensure that you have Git and Python's pip package manager installed on your system. Additionally, it's crucial to use Python version 3.8.10, as this was the specific version used for developing the application. Using any other Python version may lead to inconsistencies and library conflicts. While created on a Windows system, the code is designed to run seamlessly on any other operating system

2.1 Cloning the Repository

Begin by cloning the project's Git repository to your local machine using the following command:

```
git clone https://github.com/IlliaRohalskyi/IMPRESS.git
```

This will create a local copy of the project on your system.

2.2 Installing Dependencies

Navigate to the project directory using the command line, and then execute the following command to install the required Python dependencies from the 'requirements.txt' file:

```
pip install -r requirements.txt
```

This will ensure that you have all the necessary libraries and packages to run the project.

Please note, for Linux users it is necessary to install PostgreSQL development files. In Ubuntu, you can use following command to terminal:

```
sudo apt-get install libpq-dev
```

2.3 Setting Environment Variables

Afterward, you'll need to configure environment variables tailored to the project's specific needs. For guidance on this process, please consult the 'Configuration' chapter in this documentation. Keep in mind that configuring your infrastructure to pass the necessary environmental variables is expected. For additional insights into infrastructure setup, refer to the 'Cloud Infrastructure' and 'MLflow and DVC Integration' chapters.

2.4 Testing the Installation

To verify that the installation was successful, run any available unit tests. The 'Troubleshooting' section of this documentation provides further information on running tests and ensuring the system is set up correctly.

Warning: If there are no models in the MLflow registry, some tests may not run because the software lacks models to load. To address this, you need to train the model. Execute the following command in your terminal:

```
python src/pipelines/train_pipeline.py
```

This command will train the necessary models. Additionally, make sure to label them as "Production" in the MLflow model registry. Once you've marked the models as "Production," all tests should work smoothly.

With these installation steps completed, you are ready to start working with the project.

Chapter 3

Configuration

3.1 File Structure

The project follows a structured file hierarchy to maintain organization and accessibility. Below, we provide an overview of the main directories along with some of their key nested directories:

- `/src`: This directory contains the primary source code for the application.
  - `/components`: Within the `src` directory, the `components` directory houses modular components or submodules that make up the core functionality of the application.
  - `/pipelines`: Within the `src` directory, the `pipelines` directory stores pipelines of our software. Those pipelines are composed of components and individual functions that are specific to each pipeline. If there is a need to extend the software, those functions could be re-implemented in components folder as classes.
  - `/notebooks`: Within the `src` directory, the `notebooks` directory stores `.ipynb` file, which is a jupyter notebook used to explore and experiment with the data.
- `/tests`: The `tests` directory serves as a central location for a testing suite.
  - `/test_data`: Synthetic data files are stored here, providing the initial source data for testing purposes.
- `/Documentation`: This directory contains project-related documents, including this documentation.
- `/Presentation`: This directory contains a presentation with regards to this project.

- `/logs`: Log files generated by the application, such as error logs and access logs, are stored in this directory.
- `/ml_downloads`: Trained machine learning models used by the application are located in this directory.
- `/artifacts`: This folder stores artifacts generated during training process.
- `/.github/workflows`: This folder stores `.yaml` file, which is essential for CI/CD workflow, and is triggered by Github Actions.

This hierarchical file structure ensures that the project's components and resources are well-organized and easy to locate.

3.2 Environment Variables

To run the application successfully, several environment variables need to be set, and some of them should be treated as secrets, especially when used in GitHub Actions workflows. These environment variables are essential for various aspects of the application. Below is a list of the required environment variables along with their purposes:

- `DVC_USERNAME`: This variable is used for authentication with DVC (Data Version Control) and should be set to your DVC username.
- `DVC_TOKEN`: The DVC token provides secure access to your DVC repositories. It should be set to your DVC token.
- `MLFLOW_TRACKING_URI`: MLflow requires this URI to connect to the MLflow tracking server. Set it to the appropriate MLflow tracking server URI.
- `MLFLOW_TRACKING_USERNAME`: This is the username for authentication with the MLflow tracking server.
- `MLFLOW_TRACKING_PASSWORD`: The password for authentication with the MLflow tracking server.
- `DB_HOSTNAME`: The hostname or IP address of the database server. This is crucial for the application to connect to the database.
- `DB_NAME`: Set this variable to the name of the database used by the application.
- `DB_PASSWORD`: The password for accessing the specified database.
- `DB_USERNAME`: The username used to authenticate and access the database.
- `EMAIL_RECIPIENT`: This environment variable represents the email address to which notifications and reports will be sent.

- `EMAIL_SENDER`: The email address used as the sender when sending notifications.
- `EMAIL_PASS`: The password associated with the email sender's account for email notifications.
- `PREFECT_CLOUD_API_TOKEN`: This variable stores the API token for authentication.

Ensure that these environment variables are correctly set and securely managed, especially when dealing with sensitive information. It's essential to store sensitive variables as secrets in GitHub Actions workflows to protect your application and data.

Furthermore, there is a set of secrets specific to GitHub Actions that should be configured. These secrets are considered as environment variables, and while they might not be essential for running the application locally, they must be defined as secrets within the GitHub Actions workflow.

- `AWS_ACCESS_KEY_ID`: The AWS access key ID is used for authenticating with AWS services. Ensure that this key is properly configured with the necessary permissions to interact with Amazon Elastic Container Registry (ECR).
- `AWS_SECRET_ACCESS_KEY`: This AWS secret access key complements the access key ID and is essential for secure authentication with AWS services, including ECR.
- `ECR_ACCOUNT_ID`: The ECR account ID should be set to your AWS account ID. This ID is required for authentication when interacting with ECR, including pushing and pulling Docker images.
- `ECR_REGION`: Specify the AWS region in which your Amazon Elastic Container Registry is located. This region parameter ensures that the application communicates with the correct ECR registry.

  Ensure that you incorporate **all** the environmental variables mentioned within GitHub Actions secrets.

3.3 Database Setup

To support the functionality of the application, a PostgreSQL database is required with four main tables: 'online_data', 'archived_data', 'test_online_data', and 'test_archived_data'. Each table serves a specific purpose and plays a crucial role in the data management and prediction pipeline. Below is an overview of these tables:

- `online_data`: This table is designed to store data received from sensors. It serves as the primary data source for the prediction pipeline, where

time series data is processed and predictions are generated based on this incoming information. For a sample dataframe, please refer to the file, located in the following path:

`/tests/test_data/synthetic_online.csv`

- `archived_data`: The `archived_data` table is responsible for storing data received from the prediction pipeline. It includes experiment numbers, along with the associated washing/rinsing class and predictions. This historical data is valuable for analysis and monitoring. For a sample dataframe, please refer to the file, located in the following path:

  `/tests/test_data/synthetic_archived_data.csv`

- `test_online_data`: Similar to the `online_data` table, the `test_online_data` table stores sensor data specifically for testing purposes. It allows for testing and validation of the prediction pipeline in a controlled environment.

- `test_archived_data`: As with the `archived_data` table, the `test_archived_data` table is used to store data received from the testing phase, including experiment numbers, washing/rinsing class, and test predictions. This historical test data helps evaluate the performance of the prediction pipeline in testing scenarios.

These tables collectively facilitate the collection, storage, and analysis of sensor data, as well as the monitoring and testing of the prediction pipeline. Properly configuring and maintaining these tables is essential for the successful operation of the application.

Chapter 4

Usage

The primary use of this software involves automated deployment and execution of containerized prediction and monitoring pipelines on AWS servers. This is achieved through a CI/CD pipeline, which streamlines the deployment process. Here's a step-by-step guide to utilizing the software for this purpose:

4.1 Automated Deployment with CI/CD

1. **GitHub Actions Workflow**: The process begins with a GitHub Actions workflow that triggers when changes are pushed to the repository. This workflow is configured to perform testing and linting of the application.

2. **Testing and Linting**: GitHub Actions automatically conducts unit tests and checks code for compliance with coding standards. These tests ensure the software's health and quality.

3. **Containerization**: Once the tests pass successfully, GitHub Actions prepares Docker containers for the prediction and monitoring pipelines.

4. **Push to ECR**: The generated Docker containers are pushed to the Amazon Elastic Container Registry (ECR), which serves as a repository for storing Docker images.

5. **Scheduling with AWS Event Bridge**: Using AWS Event Bridge, scheduled tasks are created to run the prediction and monitoring pipelines as ECS tasks. This automated scheduling ensures that the pipelines are executed as per the defined schedule.

6. **Execution**: The prediction and monitoring pipelines are executed as ECS tasks, processing data and generating predictions. These pipelines are responsible for handling data from the 'online_data' and 'archived_data' tables.

7. **Monitoring and Reporting**: The monitoring pipeline captures potential data drift and generates reports. These reports are sent via email to the specified recipient using the configured email credentials.

This usage scenario outlines the automated deployment of containerized prediction and monitoring pipelines, ensuring the timely and efficient processing of data. The CI/CD pipeline, GitHub Actions, AWS ECR, and AWS Event Bridge are key components in this process.

For developers looking to make changes or further customize this deployment process, it's important to have a good understanding of the CI/CD pipeline, GitHub Actions configuration, AWS services, and the structure of the application. The provided instructions ensure that the software operates seamlessly in a containerized environment on AWS servers.

Chapter 5

Cloud Infrastructure

Our chosen cloud infrastructure is AWS, selected for its reliability and robust features, providing a solid foundation for our project. However, it's important to note that the software can be reconfigured to run on a different cloud provider with some adjustments.

To adapt the software to a different cloud environment, refinements in environmental variables and the GitHub Actions YAML file would be necessary. This flexibility allows for seamless integration into various cloud platforms, ensuring the software's adaptability to different infrastructures.

5.1 Database Hosting

We utilized AWS RDS for our database hosting, specifically hosting PostgreSQL. Our database includes four tables:

- `archived_data`
- `test_archived_data` (used for testing during the CI/CD pipeline)
- `online_data`
- `test_online_data` (employed for testing purposes in the CI/CD pipeline)

The differentiation between regular and test tables allows for effective software testing within our CI/CD pipeline.

5.2 Container Management

AWS Elastic Container Registry (ECR) serves as our repository for storing prediction and monitoring containers. Containers are versioned with corresponding Git hashes to facilitate rollback if needed. Upon each container push, the ECS task definition is automatically updated to use the latest containers.

5.3 ECS with Fargate

To streamline resource allocation concerns, we leveraged AWS ECS with Fargate. This choice allows us to focus on our applications without managing the underlying infrastructure intricacies.

5.4 Scheduled Tasks with EventBridge

AWS EventBridge orchestrates our scheduled tasks. The monitoring container runs once a month, while the prediction container is executed weekly. This ensures periodic monitoring and prediction updates as per our project requirements.

5.5 CI/CD and IAM Roles

Executing actions within our CI/CD pipeline necessitates specific IAM roles to ensure secure and controlled operations. These roles are carefully configured to grant the required permissions for CI/CD workflows.

5.6 Secrets Management

Sensitive information, such as secrets, is securely handled through AWS Parameter Store. This ensures that secrets are appropriately passed to task definitions, allowing our containers to run securely and seamlessly.

Our AWS-based infrastructure provides a reliable and scalable environment, enabling efficient deployment, monitoring, and maintenance of our machine learning applications.

Chapter 6

MLflow and DVC Integration

In our project, we've seamlessly integrated MLflow and DVC to enhance our machine learning workflow. This integration is specifically configured for use on Dagshub, and it's crucial to pass the correct environmental variables. Here's how we utilize these tools:

6.1 MLflow: Experiment Tracking and Model Registry

- `Experiment Tracking`: MLflow serves as a powerful tool for tracking machine learning experiments, allowing us to log and query experiments efficiently. This capability aids in comparing models, reproducing results, and fostering collaboration among team members.
- `Model and Artifacts Registry`: MLflow provides a centralized registry for models and artifacts. This ensures a systematic approach to packaging and organizing models, making them easily reproducible and deployable across different environments.

By incorporating MLflow, we elevate our experiment tracking and model management, promoting transparency and reproducibility.

6.2 DVC: Data Versioning Excellence

- `Data Versioning`: DVC excels as a dedicated tool for versioning and managing large datasets. In our case, we leverage DVC primarily for data versioning, ensuring that changes to datasets are systematically tracked, reproducible, and shareable across our machine learning projects.

The integration of MLflow for experiment tracking and model registry, combined with DVC's proficiency in data versioning, creates a cohesive and efficient framework for our machine learning development. Remember to pass the correct environmental variables for this setup to function optimally on Dagshub.

Chapter 7

Troubleshooting

In the development of our software, we have prioritized robust troubleshooting mechanisms to identify and resolve issues efficiently. Two primary resources aid in this process: extensive logging and a comprehensive suite of test cases.

7.1 Logging

Extensive logging has been integrated into our software to capture crucial information about its execution. Log files can be found in the `/logs` folder within the root directory of the project. These logs are instrumental in identifying potential bugs, understanding the flow of execution, and diagnosing issues during various stages of the software lifecycle.

When troubleshooting, examining the logs in the `/logs` folder provides valuable insights into the system's behavior, aiding developers in pinpointing the root cause of unexpected behaviors or errors.

7.2 Test Cases

To ensure the health and reliability of our software, we have developed a set of test cases located in the `/test` folder. This folder contains both integration and unit tests, collectively providing extensive coverage of our software's functionalities.

7.2.1 Integration Tests

Integration tests focus on verifying that various components of our software work together as expected. These tests simulate real-world scenarios, ensuring that the system behaves correctly in a holistic environment.

7.2.2 Unit Tests

Unit tests, on the other hand, target individual units of code to confirm their correctness in isolation. These tests help identify issues at the smallest level, enabling swift identification and resolution of bugs within specific functions or modules.

7.3 Utilizing Test Cases for Software Health

When troubleshooting, developers can leverage the test cases in the `/test` folder to validate the overall health of the software. Running both integration and unit tests aids in identifying potential issues early in the development process, fostering a proactive approach to software quality assurance.

By combining thorough logging practices, a robust set of test cases, our troubleshooting mechanisms empower developers to swiftly diagnose and resolve issues, contributing to the overall stability and reliability of our software.

Chapter 8

Contributing

Contributions to our software are highly welcomed! To streamline the process and maintain a cohesive codebase, follow these guidelines:

8.1 Isolating Components

When working on new features or improvements, isolate your changes by placing small, self-contained components in the `/components` folder under the `/src` directory. This modular approach ensures that each component can be developed, tested, and integrated independently.

8.2 Combining Components in Pipelines

Once you have successfully developed and tested your components, combine them in the `/pipelines` folder, also under the `/src` directory. This folder houses orchestrated sequences of components, forming the backbone of our machine learning workflows.

8.3 Code Style Guidelines

Maintaining a consistent code style is essential for a collaborative and readable codebase. We adhere to the PEP8 code style guidelines with specific exceptions: R0903, R0914, R0801.

8.4 Code Formatting Tools

To ensure code consistency and readability, we utilize the following tools:

- `isort:` A tool for sorting imports alphabetically, creating a clean and organized import section in your code.

- `black`: A code formatter that automatically reformats your code to comply with PEP8 standards, reducing manual formatting efforts.

Integrating these tools into your workflow helps maintain a unified code style and enhances the overall quality of the codebase.

8.5 Submitting Changes

When ready to contribute, follow these steps:

1. Fork the repository.
2. Create a new branch for your changes.
3. Make your modifications, adhering to the guidelines mentioned above.
4. Ensure that your changes pass all existing tests.
5. Submit a pull request with a clear description of your changes and their purpose.
6. Collaborate with other contributors and address feedback as needed.

By following these guidelines, you contribute to the success and maintainability of our software. Thank you for your contributions!