Uppsala University

Data Engineering II

1TD075

December 12, 2024

Building a Machine Learning CI/CD Pipeline

Author:
Alex Fooladi

Contents

1 Introduction

In the evolving landscape of machine learning, ensuring seamless integration and delivery of models to production environments is crucial. This project focuses on building an execution pipeline utilizing GitHooks to facilitate continuous integration (CI) and continuous delivery (CD) of machine learning models. The primary objective is to streamline the model training and deployment processes, ensuring that the models are reliably delivered to the production environment. In conjunction with building the pipeline, this project focuses on training a model that predicts the number of stars for a given GitHub repository. Stars on GitHub are used to indicate the popularity of a project and are also a measure of performance. The execution pipeline is however intended to be a generalized solution that can be implemented for any small machine learning project.

The available cloud resources for this project were five virtual machines on the Swedish Science Cloud and the pre-requisites were to have one VM as a development server and a second VM as a production server. The development server is where the model training and evaluation occurs while the production server serves a minimal application that utilizes the trained model.

2 Related Work

In the domain of machine learning CI/CD pipelines, several notable studies and projects have laid the groundwork for automating and streamlining the deployment of machine learning models. One such example is the work by Sato et al. (2019), which discusses the implementation of a continuous integration and continuous deployment (CI/CD) pipeline tailored for machine learning projects. The pipeline emphasizes the automation of model training, validation, and deployment processes, using tools such as Jenkins, Docker, and Kubernetes to achieve seamless integration and delivery. This approach ensures that new models are automatically tested and deployed to production environments whenever changes are committed to the repository, significantly reducing the time and effort required for manual intervention. [1]

In a report by Han et al. (2019), the authors address the identification of popular projects on GitHub, a widely-used open-source project platform.

Han et al. propose an approach to predict GitHub project popularity by first conducting surveys to determine a threshold for project popularity based on the number of stars. They then extract 35 features from both GitHub and Stack Overflow, categorized into project, evolutionary, and project owner dimensions. Using a random forest classifier, they analyze a dataset of 409,784 GitHub projects achieving an average AUC of 0.76. Key features influencing project popularity include the number of branches, open issues, and contributors, with these factors significantly impacting the distinction between popular and unpopular projects. [2]

3 System Architecture

The system architecture consists of a development environment, a production server, an integration platform, and a configuration environment as seen in Figure 1. The development environment is responsible for training, evaluating, and deploying the machine learning model. The production server hosts a web application where the user can interact with the trained model. The integration platform GitHub stores the code used throughout the environment and triggers the execution pipeline when the training code is updated. Lastly, the configuration environment is responsible for creating and contextualizing each part of the system from scratch.

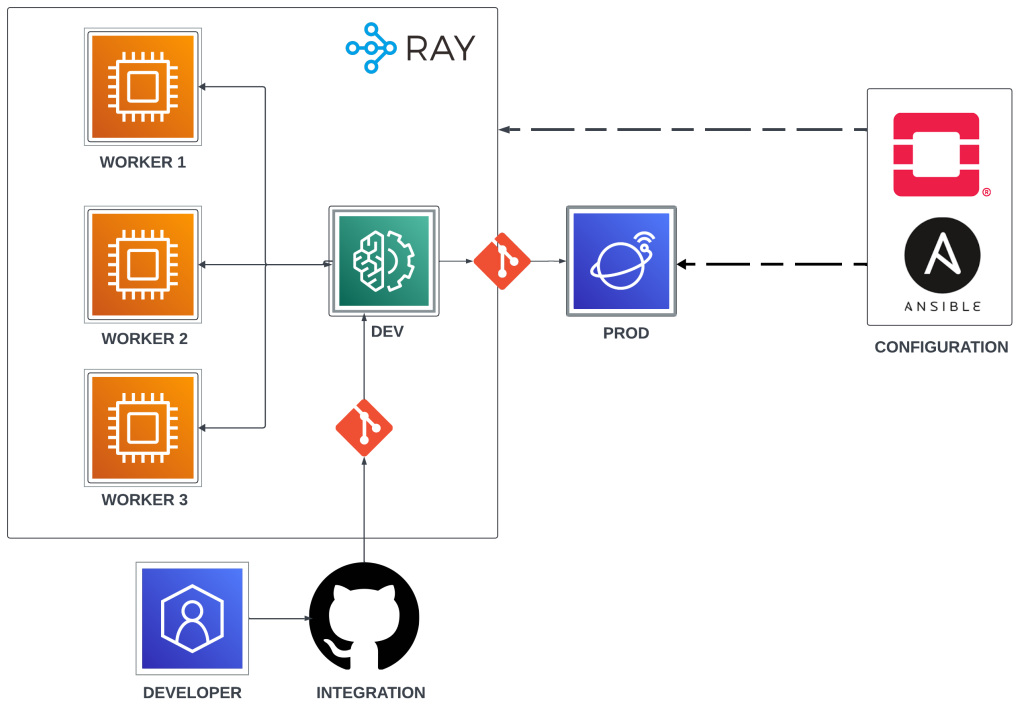

Figure 1: The System Architecture

3.1 Configuration

The configuration environment is fundamental as it is responsible for creating and contextualizing each part of the system from scratch. The configuration applications were set up in a client VM on the same network as the target virtual machines, ensuring seamless communication and management. OpenStack, an open-source cloud computing platform, was used to initialize the virtual machines on the Swedish Science Cloud. This involved creating VMs with the necessary specifications for different components of the system, configuring the networking to ensure secure and efficient communication, and allocating storage volumes to meet the data storage needs.

Once the VMs were initialized, Ansible, an open-source automation tool, was used to contextualize them by installing the necessary packages and applications. Ansible automated the installation of essential packages and dependencies on each VM and deployed the required applications and services. Both the initialization and contextualization scripts were run in docker to create an isolated and reproducible environment. This ensures that with the given configuration code, it is possible to recreate the system with minimal effort.

3.2 Development

To enable efficient scaling of the ML training, the development environment was configured as a Ray Cluster. Ray, an open-source framework built in Python, provides support for distributed machine learning, making this configuration an ideal solution to utilize computational resources [1]. The cluster consists of a head VM and three worker VMs where each VM is running Ray in a Docker container. This not only ensures that the cluster is running in isolation but also increases scalability. In the execution pipeline, the cluster is utilized for model selection and hyperparameter tuning, which are two jobs that greatly benefit from distributed computing. More on this in Section **.

3.3 Production

The production environment is where the system's functionality is showcased to end-users. It involves hosting a minimal application capable of taking a Git repository link as input and predicting the star count associated with the repository. For this purpose, the production application was built using Flask, a web application framework in Python. Flask provides a simple yet powerful platform for developing web applications, enabling efficient han-

[1]`https://docs.ray.io/en/latest/ray-overview/index.html`

dling of HTTP requests and responses. To handle tasks asynchronously and ensure scalability, Celery, a distributed task queue, was employed in conjunction with RabbitMQ, a message broker. Celery facilitates the execution of tasks asynchronously and in parallel, enhancing the responsiveness and efficiency of the production application. At each star prediction request, a job is submitted to Celery that employs a worker to load the machine learning model and predict the star count. Flask, RabbitMQ, and Celery all run as docker containers within the same virtual machine and are orchestrated with docker-compose.

3.4 Continuous Integration and Deployment

GitHub was utilized for version control, serving as the central repository for the code related to the machine learning training application, the production application, and the configuration scripts. A comprehensive Continuous Integration and Continuous Deployment (CI/CD) pipeline was established, originating from GitHub. In this setup, any change to the training application code triggers the deployment of a new model to the production server. This approach ensures that the latest model is deployed to the production environment efficiently and with minimal delay. To implement this pipeline, a post-receive hook was configured using GitHub Actions. This hook automatically pushes the latest code changes to the development server whenever modifications are made to the training code. Upon receiving these changes in a bare Git repository, the development server executes a post-receive hook that updates the working directory and initiates a new training job on the Ray cluster. Once the training process is completed, the best-performing model is saved to an empty repository. The changes are then pushed to a bare Git repository on the production server. This bare Git repository is equipped with a post-receive hook that commits the latest model to the working directory, thereby concluding the execution pipeline.

4 Machine Learning

The goal was to train a machine learning model that could predict the number of stars for a GitHub repository. A simple dataset was curated and a set of regression models were chosen to be compared against each other.

4.1 Dataset

The dataset consisted of the top 1000 GitHub repositories based on star count that were fetched with the GitHub API. The feature set is minimal and can be seen in Table 1. These are all features that were easily extracted from the initial GitHub API call.

Table 1: Feature Set

Feature	**Datatype**	**Description**
fork_count	Numerical	Number of forks
open_issues	Numerical	Number of open
disk_usage	Numerical	Disk usage in Bytes
topics_count	Numerical	Number of topics (categories)
topics	Categorical	A binary feature for each topic in the dataset

4.2 Training

The training was implemented in Python with the sci-kit learn library together with ray for distributed training on the ray cluster. Four regression models with potential were chosen from sci-kit learn and these can be seen in Table 1.

The training was divided into two phases, model selection and hyperparameter tuning. For both phases, the dataset was split into 80% training data and 20% test data. In the model selection phase, each model was trained on the training data with a 5-split cross-validation. The best performing model was then trained on the full training data and evaluated on the test data. In the next phase, hyperparameter tuning was performed on the best model. Again, a 5-split cross-validation was used for comparison for each combination of parameters, and the best performing model was evaluated on the test set. The model selection and hyperparameter tuning are both executed in the ray cluster each time a new deployment is triggered, utilizing all available resources.

Figure 2: Model Set

Model
RandomForestRegressor
LinearRegression
GradientBoost
XGBoost

5 Results

This section showcases the results and findings of training the machine learning model and a scalability analysis of the complete system architecture.

5.1 Machine Learning

To determine the performance of a model, R-squared was used during evaluation. This metric determines how well a model fits the actual data and is a real value up to 1. The highest score of 1 indicates that the model perfectly fits the data, while 0 and lower indicates a failure to capture any trend in the data. The training results show that RandomForest performed the best out of the four which can be seen in Table 2. Furthermore, the hyperparameter-tuned version of RandomForest performs slightly better than the baseline.

Table 2: Training Results

Model	**CV5 Score**	**Test Score**
RandomForestRegressor*	0.46	0.580
RandomForestRegressor	0.43	0.566
XGBoost	0.40	-
GradientBoost	0.34	-
LinearRegression	0	-

During the hyperparameter tuning, 100 trials were conducted with the Ray settings seen in Listing 1 and the best performing hyperparameters can be seen in Figure **??**.

Listing 1: Random Forest Grid Search

```
rf_search = {
    'n_estimators': tune.randint(100, 300),
    'max_depth': tune.randint(10, 30),
    'min_samples_split': tune.randint(2, 5),
    'min_samples_leaf': tune.randint(1, 5),
    'random_state': 42
}
```

Listing 2: Best Hyperparameters

```
best_hyperparams = {
    'n_estimators': 286,
    'max_depth': 23,
    'min_samples_split': 4,
    'min_samples_leaf': 2,
    'random_state': 42
}
```

5.2 Scalability Analysis

The system architecture has been designed to enable scaling when needed... - in this project performance in the machine learning is most crucial - deploy new vms that connect to ray cluster (easily distributed docker image) - on production side increase celery workers

The system architecture of this project has been designed to support scalability, particularly focusing on the performance of the development environment. The development environment where the machine learning model training takes place is configured as a Ray cluster. The cluster can scale up by deploying new worker nodes to the existing Ray Cluster, which allows for horizontal scaling. Since all worker nodes run ray in docker, the image can be distributed to a new virtual machine to seamlessly create a new worker. This flexibility ensures that as the demand for computational power grows, the system can scale accordingly without significant reconfiguration.

The production application was also designed with scalability in mind. The Flask front-end and the Celery worker containers could be replicated within the production server to scale vertically. Horizontal scaling is possible by utilizing Docker Swarm to orchestrate the containers on separate nodes. Then it would be possible to scale both horizontally and vertically by deploying containers on separate nodes and replicating within the given nodes. However, this requires a decent amount of reconfiguration but should be considered if necessary.

References

[1] D. Sato, "Continuous delivery for machine learning," 2019, accessed: 2024-05-31. [Online]. Available: https://martinfowler.com/articles/cd4ml.html

[2] W. Han, X. Xia, D. Lo, and S. Li, "Characterization and prediction of popular projects on github," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, 2019, pp. 397–406. [Online]. Available: https://xin-xia.github.io/publication/compsac19.pdf