Automated Cloud-Based ML Pipeline for GitHub Star Prediction with Scalable Deployment

Albert Bargalló i Sales **Michel Messo** **Sparsh Paliya**
Ville Friberg Elings **Adona Shinkur**

I. Abstract

This project presents an end-to-end automated pipeline for predicting GitHub repository star counts using machine learning. The system collects data via the GitHub API and trains multiple regression models with scalable distributed processing using Ray Tune across multiple workers. Model performance is evaluated using R^2 scores, with Gradient Boosting achieving the highest score of 0.6966 and being selected for deployment. The system architecture is provisioned through Cloud-init to create virtual machines on OpenStack cloud infrastructure. Ansible is used to automatically configure these VMs by installing dependencies, setting up Docker, and preparing the development and production environments. A robust CI/CD pipeline (implemented with Git hooks and GitHub Actions) fully automates the workflow from data collection and model training to deployment across Client, Development and Production servers. End users access a Flask web interface to input data for up to five repositories and receive star count predictions generated by the deployed model, demonstrating a scalable and reliable cloud-ready solution that integrates machine learning workflows with contextualization and orchestration practices.

II. Introduction

This project involves creating a distributed machine learning system with automated deployment of models from a development environment to a production environment. The machine learning model's task is to predict the amount of stars/stargazers[1] that a GitHub repository has given its other features.

A dataset of 1000 GitHub repositories with at least 50 stars is to be gathered using the GitHub API. Then, this data is to be used to train regression machine learning models which are to predict the target of stargazer amount by using features such as e.g. watchers and forks. After the models are trained, they are to be evaluated using the coefficient of determination R^2. The best model according to this metric (highest R^2 value) is then deployed from the development server (where it was trained) to a production server. There, users can use a web interface to enter the data of five GitHub repositories and get the amount of stars which this best model predicts.

III. Related Work

In [2], predicting stars on GitHub project using machine learning has been attempted. The work used the data of 2000 repositories unlike the project which this paper concerns. The work opted to use an unofficial archive site due to limitations in the GitHub API which can make it impossible to determine features like the amount of contributors (because of constraints on amount of "events" and timespan of events). A few machine learning algorithms was tested on different versions of datasets with different feature transformations and the method which yielded the highest R^2 value of $R^2 = 0.88$ was random forest with the features being log-scaled, i.e. generating transformed feature x' from original feature x through $x' = \log(1+x)$ (where the term of 1 is added to avoid the case of $\log(0)$ which is undefined).

In [1], predicting the popularity of GitHub projects was also attempted but instead by using amount of forks and watches as a measure of popularity. The work opts to looks at cross-repository patterns through "events" (e.g. a specific commit or pull request) and the order they occurred. Therefore, a long short-term memory recurrent neural network is used which is naturally more apt for sequential learning problems where sequence elements aren't in-

[1]The terms are practically synonymous and will be used interchangeably through this work, although GitHub uses the term stargazers.

dependent.

There exists public online projects on sites like GitHub where people have developed star predictors outside of a research context [3][4].

IV. System Architecture

The dataset was acquired through a script which is executed on the development machine which makes HTTP requests to the GitHub API where the content of the response is cached as a file. Figure 1 shows an overview of the system architecture for this project. In its most basic version, meaning the version without distributed hyperparameter tuning, there are three virtual machines: client, development, and production. The client machine is the main orchestrator, from which all other machines are started and managed. It does so by running *start_instance.py*, a Python script which connects to the OpenStack API and runs cloud-init files on the newly created machines.

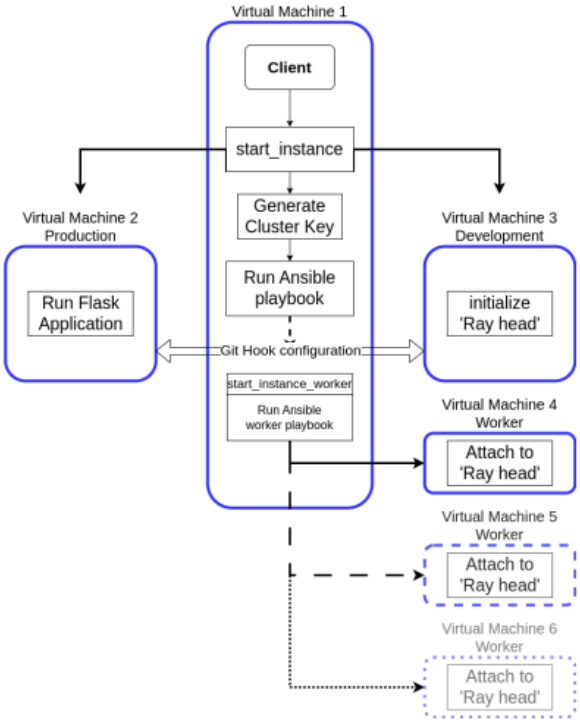

Figure I: System architecture figure

Once the machines are provisioned, the Ansible playbook needs to be run to execute the orchestrated deployment process. It automates the installation of packages, updating of apt, cloning of the GitHub repo, running of Docker, and setting up of git hooks. Docker is used to contextualize and start RabbitMQ and Celery, which are part of the back-end of the production server. The front-end Flask web-app is also hosted via Docker on the production server. On the development server, the playbook installs Python with essential machine learning packages, creating an environment for the development and testing of machine learning models. The setup of Git hooks for CI/CD functionality is also implemented in this process, where SSH key pairs are generated and distributed between development and production servers, enabling secure automated deployments.

A. Scalability

The system architecture is designed with horizontal scalability as a core principle. For distributed hyperparameter tuning and model training, support for creating worker VMs and orchestrating these with an Ansible playbook has been implemented. The idea is for the developer to run *start_instance_worker.py* to start new machines, then to run the *rayworker_configuration.yml* playbook, which installs all the needed packages and connects to the ray head. This distributed computing approach using Ray enables parallel execution of training tasks across multiple machines, significantly reducing the time taken for model training and hyperparameter tuning. This approach allows developers to scale the cluster up or down based on computational needs, with the worker nodes being managed on the Ray cluster. Additionally, the production environment uses Docker containers with RabbitMQ and Celery to automatically scale the web-app, by handling increases in prediction requests.

B. CI/CD Pipeline

The system implements a robust CI/CD pipeline that enables seamless deployment of improved machine learning models from development to production environments. Upon completion of the Ansible playbook configuration, Git hooks are established to automate the deployment workflow. When developers train new models on the development server and commit changes to the repository, a GitHub Action workflow is automatically triggered to validate model performance by comparing R^2 scores against the current production model. If the new model demonstrates superior performance, it is automatically saved in the Production directory. The deployment process is then completed through a bare Git repository setup on the development server, where developers can push the validated model to the production server using Git hooks. A post-receive hook script orchestrates the final deployment step by up-

dating the production environment with the new model, ensuring zero-downtime deployment. This automated pipeline eliminates manual intervention in the model deployment process while maintaining quality control through performance validation, allowing the Flask web application to immediately serve predictions using the latest and best-performing model.

V. Results

In the stage of early analysis of the dataset, it was discovered that the amount of stargazers was equal to the amount of watchers for all the repositories. This is most likely because of some fault in the github API since the amount of watchers weren't equal to the amount of stars for some repositories which were manually inspected. Therefore, the features derived from the API response (both the fields of "watchers" and "watchers_count") are harmful for the machine learning process since they seem to be completely invalid. Therefore, a feature was not created from this data.

A. Scalability Analysis

The GitHub Star Predictor application is designed to support horizontal scalability. As the number of users grows or as more repositories are submitted, the system can maintain performance by distributing the workload across multiple service instances.

Celery is used to handle background tasks, and RabbitMQ serves as the message broker that manages and routes these tasks to available workers. This design allows more Celery workers to be added when needed, enabling the system to scale horizontally without changes to its internal logic. This becomes especially important if long-running tasks such as data collection through API calls are introduced or if user interaction increases in future versions of the application.

For model optimization, Ray Tune is used to run hyperparameter tuning experiments in parallel. This approach allows a larger configuration space to be explored more quickly and enables efficient use of available compute resources. As datasets grow or models become more complex, the tuning process can be scaled out across multiple cores or nodes.

To evaluate this capability, Ray Tune was executed on different configurations with increasing CPU resources. The first experiment used a single virtual machine with 1 CPU. A second virtual machine with 2 additional CPUs was then added, followed by a third machine with 2 more CPUs, bringing the total to 6 CPUs. The resulting execution times, shown in Figure II, demonstrate a consistent reduction in tuning duration as CPU resources increased. This result confirms that Ray Tune benefits from horizontal scaling and is well-suited for distributed optimization workloads.

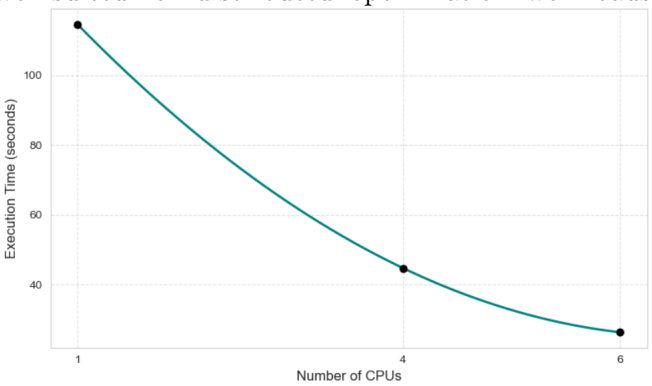

Figure II: Execution time vs CPU count

The system is containerized using Docker and managed through Docker Compose. Each component of the application, including Flask, Celery, and RabbitMQ, can be scaled independently. This containerized structure makes the application easy to deploy across different machines or in cloud environments and ensures that services can grow smoothly as demand increases.

B. Model Accuracy Comparison

Several regression algorithms were used to train and test a common dataset with the same preprocessing pipeline, evaluating the efficacy of the different models in predicting the number of stargazers on GitHub. The evaluation of each model's performance was based on three standard regression metrics:

- **R^2 Score (Coefficient of Determination):** Represents the proportion of total variance in the target variable accounted for by the model.
- **MSE (Mean Squared Error):** It is the average of the squared difference errors between the prediction and the real value.
- **MAE (Mean Absolute Error):** It is the average of absolute differences between predicted and actual values.

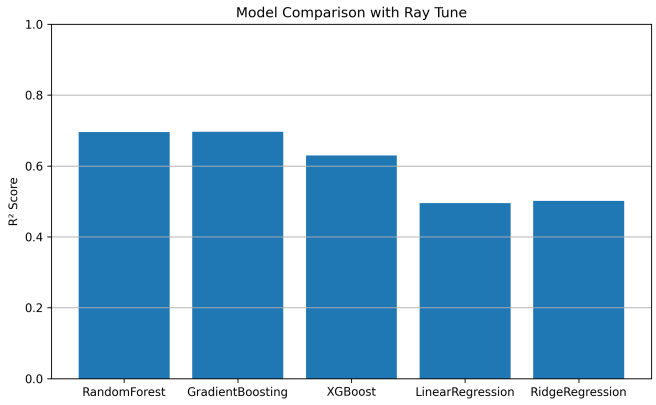

FIGURE III: MODEL COMPARISON BASED ON R^2 SCORE.

From the comparison, it appears that the tree ensemble models stood tall against the linear models. Gradient Boosting scored the best R^2 score (0.6966) and the least MSE, making it the most suitable model for this task. Random Forest and XGBoost performed closely behind, whereas linear approaches, i.e. Linear and Ridge Regression, lagged behind when it came to prediction capabilities, obtaining lower R^2 scores with substantially high error measures.

This suggests that the relationship between the input features and the number of stargazers is likely nonlinear, which the ensemble methods model better.

REFERENCES

[1] Neda Hajiakhoond Bidoki, Gita Sukthankar, Heather Keathley, and Ivan Garibay. A cross-repository model for predicting popularity in github. In *2018 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1248–1253, 2018. doi: 10.1109/CSCI46756.2018.00241.

[2] Mohammed Abdul Moid, Abdullah Siraj, Mohd Farhaan Ali, and Ahmed Osman Amoodi. Predicting stars on open-source github projects. In *2021 Smart Technologies, Communication and Robotics (STCR)*, pages 1–9, 2021. doi: 10.1109/STCR51658.2021.9588891.

[3] GitHub user "Doodies". Github - doodies/github-stars-predictor, 2018. URL `https://github.com/Doodies/Github-Stars-Predictor`.

[4] GitHub user "pcsingh". Github - pcsingh/predicting-repo-popularity, 2020. URL `https://github.com/pcsingh/Predicting-Repo-popularity`.