Uppsala University

Data Engineering II

V24 1TD075

Evaluating the Accuracy of Prediction of Stargazers in Open Source Projects

---

Author:
Erik Larsson
MD Masudul Islam
Yagna Karthik Vaka
Sahil Sandal
Anand Mathew M S

GitHub repository for this project can be found *here*.

May 31, 2024

1 Introduction

In the age of social media, the concept of popularity has become crucial in identifying the performance and reach of various entities. GitHub, a hub for developers measures the popularity and acceptance of a repository within the developer community using "stargazers" or "stars" on a repository. This project aims to evaluate the accuracy of different prediction models in forecasting the number of stars a GitHub repository will receive based on other observable information on the repository.

The objective of this project is to extract metadata using GitHub API, develop machine learning models, compare and choose the best algorithm to predict the star count based on historical data of GitHub repositories, build a scalable data engineering pipeline with CI/CD, and display the results in the form of a dashboard.

2 Related Work

Prior studies have focused on predicting repository popularity using various features available through the GitHub API. Main Features such as the number of forks, repository size, open issues, watchers, contributors, stars, and commits activity have been utilized to build predictive models. Machine learning techniques like linear regression, decision trees, and ensemble methods have shown promise in accurately predicting repository popularity metrics.

3 System Architecture

3.1 Data Extraction and Handling

The data was extracted from the GitHub API by querying the URL and with the PyGithub package for Python. Both of these methods queried repositories on GitHub having stars greater than or equal to 50 for a number of features. After the fetching of information from each repository, the script was made to sleep for a fixed period to respect the rate limit of the GitHub API. Data from 1947 repositories was collected this way. This was then divided into:

Training Dataset: The dataset is used by different models to generate models, tune hyperparameters and obtain model accuracy scores.
Test Data: Consists of 5 random data points for final testing on the Production VM.

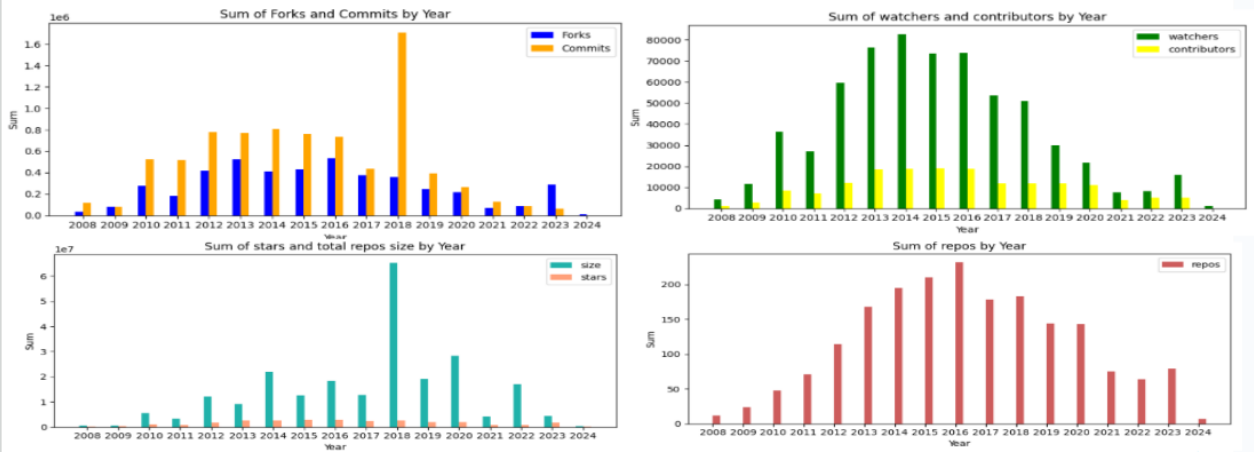

Figure 1: Data Plots

3.2 Machine Learning Implementation

3.2.1 Feature Extraction:

- Historical data of the top 1947 GitHub repositories with at least 50 stars was collected using the GitHub API.
- Main Extracted features included the number of forks, watchers, open issues, commits, contributors, creation date, update date, and push date. Other features such as full name, URL, has_issues, has_projects, has_downloads, license, allow_forking, topics, score, visibility, and so on around 33 total features are included.
- Included some boolean features that can help in regularization to reduce the risk of overfitting by simplifying the model.
- We have completed the data cleaning process such as dropping some features (language, license, topics as they have more missing values) and also checked whether we have duplicates in the dataset.
- As shown in Figure 1, the data plots demonstrate significant trends of GitHub repositories.

3.2.2 Model Training and Hyperparameter Tuning:

- Split the dataset into training (80%) and testing (20%) sets.

- Trained on these regression models: Linear Regression, Ridge Regression, Lasso Regression, Bagging Regressor, Random Forest Regressor, AdaBoost Regressor, and Gradient Boosting Regressor.
- Utilized Ray Tune for hyperparameter tuning with a random search algorithm, Grid Search Algorithm, and ASHAScheduler for early stopping.
- Defined custom training functions and configured search spaces for each model's hyperparameters:
  - Linear Regression: `fit_intercept`
  - Ridge/Lasso Regression: `alpha`
  - Random Forest: `n_estimators`, `max_depth`, `min_samples_split`
  - Bagging: `n_estimators`, `max_features`, `max_samples`
  - Adaptive & Gradient Boosting: `n_estimators`, `max_depth`, `learning_rate`
- Conducted 10 trials for each model to identify the best hyperparameters.
- Evaluated model performance using R-squared scores:
- Saved the trained model files as .pkl (pickle) files and recorded the accuracy scores in an accuracy.txt file.
- Compared the R-squared scores of all models to determine the best-performing model, with the Gradient Boosting model achieving the highest R-squared value but the Random Forest with the most consistency.

3.3 Data Engineering Pipeline

The system consists of an Orchestration client which spawns the Production and Development servers using OpenStack client. The Ansible orchestration tool is then used to perform contextualization on the servers, handle the configuration of ssh keys, set up git repositories on both the servers connected by git hooks, and execute docker-compose up to build and run containers on the servers. Containers on the development server form a scalable Ray cluster with a Ray head and workers for training the models. On the Production server, containers set up a RabbitMQ message queue with a Celery worker as the backend and a Flask application as frontend for the dashboard. With this set up the manual tasks are reduced and the manual labor necessary to push a new model to production is to go into the development

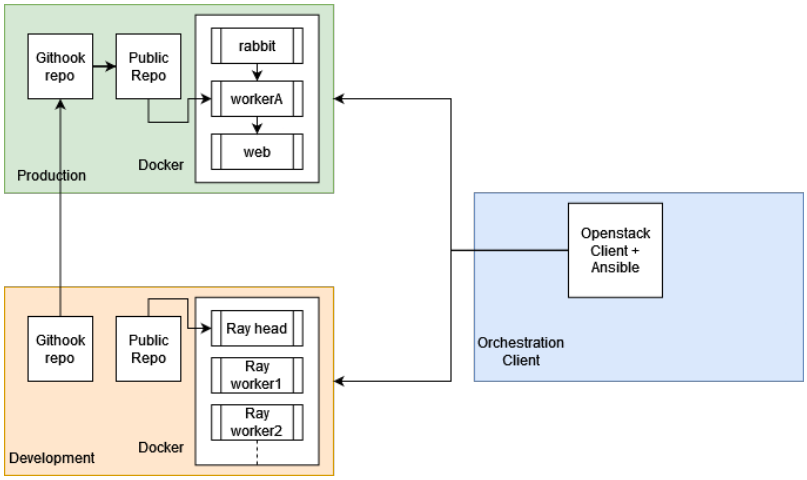

Figure 2: System Architecture

server, access ray head container and train the model. After training, executing a script will push the model to a git repository where it can be pushed to the production server and automatically updated.

3.3.1 Orchestration

Ansible Playhook

- Setup Playbook: Installs necessary dependencies and sets up Docker on both Development and Production VMs.
- Deployment Playbook: Manages the initialization of Git repositories and ensures the correct deployment of models and their results.

3.3.2 Docker

Development and Production servers are containerized using Docker and Docker Compose to ensure robustness and scalability.

- **Development:** The docker containers are used to constitute a scalable Ray cluster consisting of a single Ray head and workers connected to the Ray head which can be scaled on demand. The machine learning models are trained here.
- **Production:** Docker containers are used to constitute components of the Dashboard. The RabbitMQ container acts as the message queue which handles the communication between the Flask web app container which produces

tasks and the Celery worker container which consumes the tasks. The worker contains functions that execute the task which is pushed through the queue and picked up by the Flask app that renders it onto the dashboard.

4 Results

4.1 Comparison of Machine Learning Models

```
Best hyperparameters for Linear Regression:  {'fit_intercept': False}
Best hyperparameters for Ridge:  {'alpha': 57.64361514505448}
Best hyperparameters for Lasso:  {'alpha': 0.2244372405316268}
Best hyperparameters for Random Forest:  {'n_estimators': 84, 'max_depth': 8, 'min_samples_split': 6}
Best hyperparameters for Gradient Boosting:  {'n_estimators': 91, 'max_depth': 4, 'learning_rate': 0.02335196577754960 7}
Best Linear Regression R-squared: 0.7285163540382144
Best Ridge R-squared: 0.7289509149530737
Best Lasso R-squared: 0.7285263599682479
Best Random Forest R-squared: 0.8469022610340251
Best Gradient Boosting R-squared: 0.8318645948885932
              Model  R-squared
0  Linear Regression   0.728516
1              Ridge   0.728951
2              Lasso   0.728526
3      Random Forest   0.846902
4  Gradient Boosting   0.831865
R-squared values have been written to test_accuracy.txt
```

Figure 3: Performance results for different models

Based on the model performance evaluation, the Random Forest model achieved the highest R-squared value of 0.846, indicating the best predictive performance among the evaluated models. The Gradient Boosting model also performed well, with an R-squared value of 0.831. The Linear Regression, Ridge, and Lasso models showed similar performance with R-squared values of approximately 0.728. The hyperparameter tuning for each model revealed the optimal settings: Linear Regression with `fit_intercept` set to False, Ridge with `alpha` set to 57.643, Lasso with `alpha` set to 0.224, Random Forest with 84 estimators, max depth of 8, and min samples split of 6, and Gradient Boosting with 91 estimators, max depth of 4, and learning rate of 0.023. These results demonstrate that ensemble methods, specifically Random Forest and Gradient Boosting, are more effective in predicting the number of stars for GitHub repositories compared to linear models.

4.2 Scalability Analysis

To assess the scalability of our model training process, we conducted a strong scalability study by varying the number of CPUs. The scalability tests were made on

the container level by first training with the ray head node, and later connecting ray workers. To conduct the tests the containers were restricted to utilizing one CPU and the total CPUs available on the development instance was four. Therefore the limit for the system was one ray head and three connected workers.

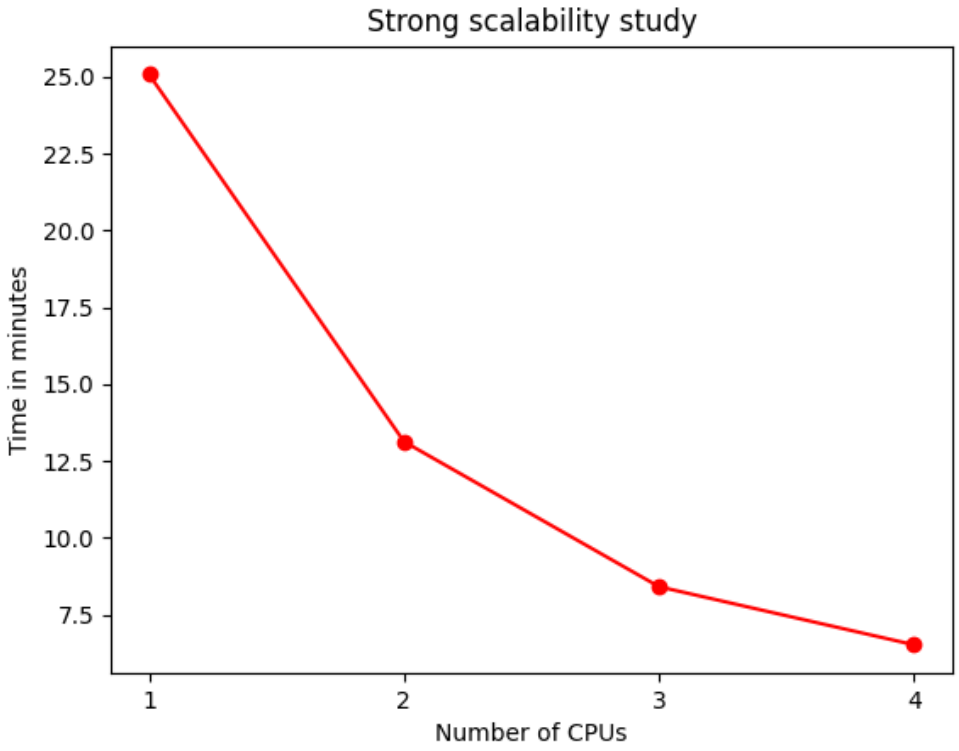

Figure 4: Time taken for training with varying number of CPUs

The graph demonstrates a significant decrease in training time as the number of CPUs increases, indicating good scalability of our training process. Specifically, the time taken for training was reduced from 25 minutes with 1 CPU to 13 minutes with 2 CPUs, around 8.5 minutes with 3 CPUs. This demonstrates that with the increase in computational resources, the training time decreases, highlighting the efficiency of our model training setup in handling larger datasets and more complex models. The reason for the training time not decreasing linearly is likely because the models are trained on 1941 git repositories and the models are trained sequentially which data parallel training is utilized. The same model is applied to every worker and the data is split. Since the data size is limited the overhead to split the data increases with

more workers its possible to assume it is the overhead that results in the time not decreasing linearly after 1 worker. With more data, however, it's possible to assume that the system is scalable.

One drawback with the current setup is that the system can only scale at the container level at the moment since in the Ansible playbook file a ray head is initialized with every development server. To utilize more development servers docker swarm would be a good option and rewriting the ansible playbook so the swarm manager is hosting the ray head and the ray workers are set up on the swarm workers.

5 Conclusion

Our findings indicate that ensemble methods, specifically the Random Forest and Gradient Boosting models, outperformed linear models in terms of predictive accuracy. The Random Forest model achieved the highest R-squared value of 0.846, followed by the Gradient Boosting model with an R-squared value of 0.831. This demonstrates the effectiveness of ensemble techniques in capturing complex patterns in the data.

Additionally, we conducted a strong scalability study to assess the efficiency of our training process. The results showed a significant reduction in training time with the increase in the number of CPUs, highlighting the scalability of our approach. The training time decreased from 25 minutes with 1 CPU to below 7.5 minutes with 4 CPUs, demonstrating that our system can efficiently handle larger datasets and more complex models.

Overall, this project provides valuable insight into the factors influencing the popularity of GitHub repositories and presents a robust methodology for predicting repository starts using advanced machine learning techniques. The implementation of Docker and Ansible for containerization and orchestration further ensures a scalable and reproducible workflow, making it a good solution for real-world applications.

Wish you have a wonderful summer :-)