Uppsala Universitet

Data Engineering II 7.5 hp

Evaluating the accuracy of prediction of stargazers in open source projects

Project group 16

Joakim Michalak, Adam Orucu,
Sophia Zhang Pettersson

May 31, 2021

Contents

1 Introduction

This study was carried out as a part of the course “Data Engineering II” at Uppsala University. The purpose of the study was for us to learn more about orchestration in the context of machine learning. The study topic is prediction of the popularity of GitHub repositories, based on repository properties.

GitHub is the world’s largest collection of open source software, with more than 31 million users and 96 million repositories [3]. It offers storage and versioning control and is suitable both for individual work and group projects. It also provides mechanisms for social interaction, such as the possibility to give star ratings to repositories, and to follow other users activities [6]. According to [2], the number of stars (or "stargazers") given to a repository can be seen as a proxy of its popularity. In a recent study from 2020, the relation between the number of stars, the number of forks and the number of subscribers as measurements of popularity was analyzed [3]. In [6], the authors instead analyze what kind of activities make individual developers popular, and postulate that there may be a connection between the popularity of developer and the popularity of the repositories the developer is contributing to.

Several studies have been conducted to analyze metadata from GitHub repositories in general. In [4], the authors made a comparison between using a feature-based random forest model and a simple star count to predict whether a repository was used in a “engineered software project” or not, and found that the random forest model had better overall results. In [2], the authors found that repository features such as programming language, application domain, type of owner and number of forks have significant impact of the repositories popularity. A weaker correlation was found with the number of contributors, whereas none was found with e.g. repository age or the number of commits.

Our research question in this study is “how can we create scalable ML models for predicting the number of stars for a given GitHub repository”.

Our architecture design scirpts and code as well as scaling tests are given in the group project repository in `Github:https://github.com/adamorucu/de2_project`. The contribution statement is included at the end of the report.

2 System Architecture

In this section the infrastructure needed to solve the task at hand will be described. The architecture was designed to be able to scale for bigger problem sizes and to be able to do that with minimum user effort. Therefore, scripts that utilise OpenStack API, Docker containers and Ansible were created. Although theoretically these scripts could be run from any device that has the right permissions, we choose to create a separate instance that we call the "client" to manage the cluster from.

There are two main parts the architecture consists of, namely development and production. This was done in order to separate the section where the models are trained and tested where many different trained models might exist at the same time, and the part where we present the users with a final model which they can use to ask for predictions. Both these parts have one server that is considered the "head". This, aside from giving easy access to desired parts provides the ability to complete separate the access given the privileges a person possesses. For example, the development team would only be able to access the head development server, therefore have access to the clusters computational power but not need to worry about what happens within them or the production server.

The heads of the two mentioned parts of the architecture, two servers can be launched using a script that uses OpenStack API to launch two instances and then configure with in the desired way using Ansible as a central manager [5]. The parts of the architecture will be described over the next two subsection. The diagram of the entire architecture can be viewed in figure 1.

2.1 Development Cluster

It should be possible to use the solution to train and test different models on large datasets, and the amount of computation can be expected to grow quickly for bigger datasets. This is especially true as tuning of hyperparameters is needed to find the best model for the given dataset. Based on our previous experiences we decided to create a Ray cluster to be able to distribute the work to different machines, something that would reduce the computation time significantly. This Ray cluster was then used to train and test, with results as described in section 4.

In order to not waste resources unnecessarily, it is good to scale the cluster size according to the data size, and useful to have an easy way to do so. For this reason we use a Python script that uses the OpenStack API to launch a new "slave" instance, then we use Ansible to set up the instance according to our configurations; updates, files and packages. Once the slaves are set up they can be connected to the head of the Ray cluster - that is the main development server.

After the models are tuned using Ray, they can be tested to pick the best one and push into the production server.

2.2 Production Server

As previously mentioned, the production server is used to provide the users the ability to query the trained model. Since the query is not computationally demanding, we decided that it is not paramount that we have a cluster for computing and responding to the queries. Instead we put the operations into Docker containers, which enable to scale the workers on the device, if the need would arise. Docker containers also provide a good encapsulation of software and required packages [1].

The production server contains three different types of containers, with different aims. One holds the Flask application which hosts the website we provide the users to make queries to the chosen trained model. Second container contains a Celery worker which performs the actual request i.e. loads the model and makes a prediction, then returns the output. The last container holds a RabbitMQ message broker which stores the queue of tasks the worker is supposed to accomplish. Having these parts in separate containers makes the system easier to scale. For example, if the prediction requests become too many that the worker cannot keep up, one might just use Docker's scale feature to launch more worker containers that will take the extra tasks waiting the queue.

2.3 Deployment

Aforementioned, after the tuning, training and testing processes the best model is sent to the production server. This architecture can be viewed in figure 1. Every time there is a small modification in the model (e.g. parameter changes, new/more data to train on) the model in the production needs to be updated. This can be mundane and time consuming action, so we automated it by using git hooks. A bare git repository was initialised on the production server which contains the best model. The workers on this device use this model to make the predictions when a user requests it. When a change occurs in the development server, the new model is packed using `pickle` and push to the remote repository on production where it can be loaded before a prediction is made.

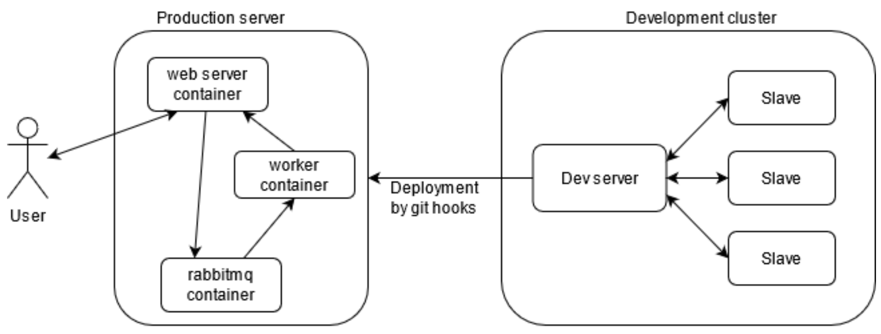


Figure 1: Diagram of the architecture

3 Data Collection

We created a Python script (based on the library PyGithub) to download repository metadata, using the Github API. The script is executed from the command line shell and stores the resulting data as a file in the Ubuntu file system. Since the data processing in this task is batch-oriented, this approach was deemed as a robust and simple solution.

By querying the API, the script gets a list of the top 1000 repositories according to the number of stargazers. For each item in the list, the available metadata for the corresponding repository is downloaded (again using the API). Finally, the relevant features are extracted and stored as a CSV file where each row represents a specific repository and each column contains the values for a specific feature. The script has logic for handling the Github API rate limitations (pause – continue) and accepts command line parameters for execution configuration such as whether urls are to be included in the relevant features or not.

4 Model Training

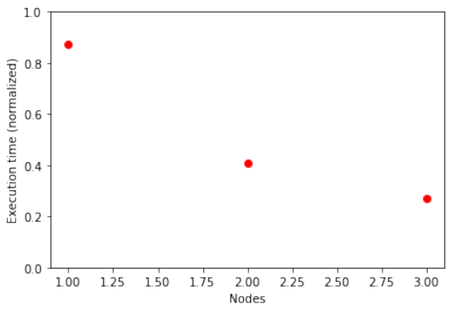
The training of the models is done on the development server. It is done in an interactive Jupyter Notebook, where multiple models can be trained and evaluated, and further pushed to the production if deemed to be ready for production. The preprocessing is done by selecting relevant features. The following models are trained: Random Forest Regressor, XGBoost and Linear Regression. Further, hyperparameter tuning is performed. The evaluation of all models is done by calculating the R-squared measure on a separate test dataset.

5 Results

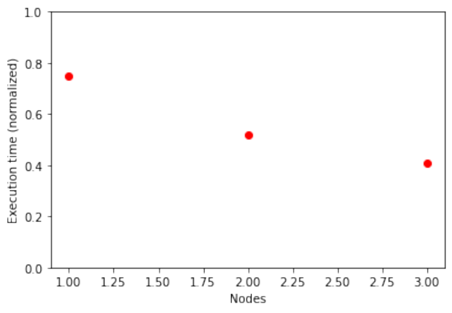
5.1 Scalability Analysis

To measure the scalability of our infrastructure, we mainly focus on the development side, where hyperparameter tuning is performed. The infrastructure allows for horizontal scalability for the hypertuning, since hyperparameter tuning is a time consuming task which can be parallelized. To test the scalability, both weak and strong scaling experiments are performed. In weak scaling experiments, the problem size is increased proportionally with the number of nodes, whereas in strong scaling we add more nodes while keeping the problem size constant. Thus, for weak scaling, the execution time should ideally be constant. For strong scaling, the execution time should decrease proportionally as the number of nodes is increased.

The strong scaling experiment is done by performing hyperparameter tuning on a training set of 800 data points on 1, 2 and 3 nodes. The hyperparameter tuning is performed on a Random Forest model with 324 combinations of different parameters. Figure 2 a) shows the results of the experiment. We can see that the decrease of execution time is almost proportional to the nodes added, which indicates that the architecture has good horizontal scaling properties.



(a) Strong scaling experiment with 3 nodes.



(b) Weak scaling experiment with 3 nodes.

Figure 2: Strong and weak scaling experiments.

The weak scaling experiment is done by performing hyperparameter tuning on a training set of 300 data points for 1 node, 600 for 2 nodes and 900 for 3 nodes. Similar to the strong scaling experiment, the training is performed on a Random Forest model with 324 combinations of different parameters. Figure 2 b) shows the results of the experiment. The expected result with 100% efficiency would be to have a constant execution time, however the results actually show an overperforming efficiency. This may be explained by the small amount of data points being used. For better accuracy of the experiment, larger training sets could have been chosen. However, based on these experiments, the results are promising and indicate that the hyperparameter tuning is able to scale horizontally with high efficiency.

5.2 Model Accuracies

The models are evaluated based on the R-squared measure after tuning the hyperparameters for each model. Each model is trained on a training set, after which the R-squared measure is evaluated using a separate test set. The results show values of 0.75 for Random Forest Regressor, 0.65 for XGBoost and 0.71 for Linear Regression. Hence, the Random Forest model is chosen and pushed to production. R-squared measures in percentage how well the data fits a regression model. For the best model, 75% of the variation in the output stars can be explained by the chosen features which indicates that the features correlates with the amount of stars.

6 Discussion and Conclusion

Using industry-standard platforms and tools, we managed to create a scalable infrastructure in a machine learning context, including management of deployment from development to production. The solution is well documented in the form of Ansible scripts and Docker container specifications, which makes it straightforward to set up a full-fledged infrastructure from scratch. By incorporating the cluster functionality of Ray in the setup, the model training environment can easily be adapted to handle different problem sizes. The deployment process is realized using a combination of Git hooks and `pickle`, providing a solution that is both stable and very easy to use.

We found that the random forest reached the best performance in terms of accuracy, reaching an R-squared measure of 0.75. This could likely be improved by further preprocessing, for example optimized feature selection and feature engineering. The accuracy would probably also be higher if we used a larger training set. Furthermore, if the model was to be used on randomly selected repositories, we would need to extend the training data to a more representative selection of repositories. As instructed, the models were only trained on the top 1000 repositories with the highest number of stars.

Our scalability experiments for the hyperparameter tuning indicate that the setup is horizontally scalable, and exhibits both strong and weak scaling properties. However, the experiments would benefit from utilizing larger training sets, for better accuracy. The execution times were fairly small, and the results in the weak scaling experiment indicate that other effects than the ones we actually intend to measure might have a significant impact on the execution time, such as fluctuations in available computational power or latency when initiating a computation.

References

[1] Ben Blamey, Andreas Hellander and Salman Toor (2019), Apache Spark Streaming, Kafka and HarmonicIO: A Performance Benchmark and Architecture Comparison for Enterprise and Scientific Computing, https://arxiv.org/abs/1807.07724

[2] Hudson Borges, Andre Hora, et.al. (2016), Understanding the Factors that Impact the Popularity of GitHub Repositories , https://arxiv.org/pdf/1606.04984.pdf

[3] Abduljaleel Al-Rubaye, Gita Sukthankar (2020), Scoring Popularity in GitHub, https://arxiv.org/pdf/2011.04865.pdf

[4] Meiyappan Nagappan Munaiah N, Kroh S, Cabrey C, Nagappan M. (2016), Curating GitHub for engineered software projects. https://doi.org/10.7287/peerj.preprints.2617v1

[5] Denis Weerasiri, Moshe Chai Barukh, Boualem Benatallah, Quan Z. Sheng, and Rajiv Ranjan. (2017), A taxonomy and survey of cloud resource orchestration techniques. ACM Comput. Surveys 50, 2 (2017), 26.

[6] Joicy Xavier, Autran Macedo, Marcelo de A. Maia, (2014), Understanding the popularity of reporters and assignees in the Github https://ksiresearch.org/seke/seke14paper/seke14paper_171.pdf

7 Statement of contribution

There are only three members in our group for this project. To enhance the learning experience, all our group members, frequently participated in zoom meetings for formulation of the problem and discussions about architectural design and experimental setup, and continuously shared updates about the task progress and problems through Slack and Github. On the other hand, to improve the study efficiency, we distribute the sub-tasks as following:

Joakim Michalak: I trained the different models with hyperparameter tuning and made the evaluations, worked on the code for the Flask webserver which loads the model from Githook and predicts the amount of stars for the user's input and contributed to the report.

Adam Orucu: I worked on the architecture of the solution. Wrote Dockerfiles, Ansible and Python scripts. Also, wrote the system architecture section of the report.

Sophia Zhang Pettersson: I created the solution for data collection and contributed to the report (overall structure, writing of Introduction, Data Collection and Conclusion sections, proofreading).