SCIENCE CHINA
Information Sciences

A Clustering-based Approach for Mining Dockerfile Evolutionary Trajectories

zhang yang, Wang HuaiMin and Filkov Vladimir

Citation: SCIENCE CHINA Information Sciences ; doi: 10.1007/s11432-017-9415-3

View online: http://engine.scichina.com/doi/10.1007/s11432-017-9415-3

Published by the Science China Press

SCIENCE CHINA
Information Sciences

• LETTER •

A Clustering-based Approach for Mining Dockerfile Evolutionary Trajectories

Yang ZHANG[1,2*], Huaimin WANG[1,2] & Vladimir FILKOV[3,4]

[1]*Key Laboratory of Parallel and Distributed Computing;*
[2]*College of Computer,National University of Defense Technology, Changsha, 410073, China;*
[3]*DECAL, University of California, Davis, CA, 95616, USA;*
[4]*Computer Science Department, University of California, Davis, CA, 95616, USA*

Citation Zhang Y, Wang H M, Filkov V, et al. A Clustering-based Approach for Mining Dockerfile Evolutionary Trajectories. Sci China Inf Sci, for review

Dear editor,

Docker[1)], as a de-facto industry standard [1], enables the packaging of an application with all its dependencies and execution environment in a light-weight, self-contained unit, *i.e.*, containers. By launching the container from Docker image, developers can easily share the same operating system, libraries, and binaries [2]. As the configuration file, the *dockerfile* plays an important role, because it defines the specific Docker image architecture and building orders [3]. As a project progresses through its development stages, the content of the dockerfile may be revised many times. This dockerfile evolution is indicative of how the project infrastructure varies over time, and different projects can exhibit different evolutionary trajectories. However, projects with similar goals and needs may converge to more similar trajectories than more disparate projects. Identifying software projects that have undergone similar changes can be very important for the discovery and implementation of the best practices when adopting new tools and pipelines, especially in the *DevOps* software development paradigm. The potential to implement the best practices through the analysis of the dockerfile evolutionary trajectories motivated this work.

This research studied dockerfile longitudinal changes at large scale and presented a clustering-based approach for mining convergent evolutionary trajectories. An empirical study of 2,840 projects was conducted, and six distinct clusters of dockerfile evolutionary trajectories were found. Furthermore, each cluster was summarized, accompanied by case studies, and the differences between different clusters were discussed. The proposed approach quantifies distinct dockerfile evolution modes and reflects the learning curves of project maintainers, and this benefits future project maintenance. Also, the proposed approach is generic and can be used for the study of general infrastructure configuration file evolution.

Dockerfile overview. A dockerfile[2)] is a text document that contains all the commands a user could call on the command line to assemble a Docker image [3]. Users can create an automated build that executes several command-line instructions in succession by using a `docker build`. Docker has provided multiple types of instructions for use in dockerfiles (see Appendix A in the supporting information for details). Docker runs the instructions in a dockerfile in order and treats lines that begin with "`#`" as a comment. A dockerfile must start with a `FROM` instruction, specifying the base build. Other parts are then added on top of the base one [4], and each instruction represents one layer in a Docker

* Corresponding author (email: yangzhang15@nudt.edu.cn)
1) https://www.docker.com/
2) https://docs.docker.com/engine/reference/builder/

image. Thus, the size of a dockerfile (number of lines) can reveal the size and complexity of the corresponding Docker image.

To meet the requirements of project development, the content of a dockerfile may be modified at different stages by project maintainers. These changes over time are called the *Dockerfile evolution*. *E.g.*, during a previous stage, the owner *inutano* of project *inutano/wpgsa-docker* added a `USER` instruction and new python scripts to the initial dockerfile. However, during the later stage, he just updated the plugin's *wPGSA* version, *e.g.*, "*0.2.0*"→"*0.3.0*". Thus, these changes reflect the dockerfile evolution, and the changes depend on the practices in individual projects. During the preliminary data gathering of more than 57,000 projects from Docker Hub, nearly 70% of them had changed their dockerfile at least once. In addition, over 5,000 projects changed their dockerfile more than 10 times.

Our approach. This research proposed a clustering-based approach that defined the dockerfile scale as the number of valid command lines of the dockerfile without blank lines and comments. The proposed approach consists of four steps.

• Step-1: Preprocessing. Initially, A number of projects with different dockerfile versions were imported. For each dockerfile, its scale was computed after removing blank lines and comments. Then, for project p with n dockerfile versions, the scale of the i-th version was defined as s_i, and the evolutionary trajectory of project p's dockerfile was represented by the vector $S_p = \{s_1, s_2, ..., s_n\}$.

• Step-2: Smoothing. The Kernel regression smoothing [5] was used to reduce the noise in the original dockerfile evolutionary vectors. Traditional parametric estimation methods may not capture all evolutionary trajectories with a limited mathematical formula, but Kernel regression is a non-parametric linear smoother that can reduce noise by estimating the conditional expectation of a random variable. For each dockerfile scale s_i in project p, a tuple vector was constructed, $(x_i, s_i), i = 1, .., n; x_i = i$. For the smoothing, our goal was to estimate r with some function $\hat{r}$, so that $s_i = r(x_i) + \epsilon_i, i = 1, ..., n$. The Gaussian kernel was used to define the kernel function as $K(x) = \frac{1}{\sqrt{2\pi}} exp(\frac{-x^2}{2})$.
Then, the function $\hat{r}$ of kernel regression [6] was defined as follows:

$$\hat{r}(x) = \frac{\sum_{i=1}^{n} K(\frac{x_i - x}{h})}{\sum_{i=1}^{n} K(\frac{x_i - x}{h})} * s_i \qquad (1)$$

Where h was the bandwidth, computed by the cross-validation least-squares function. After smoothing, a new evolutionary vector was obtained, $S_p^{'} = \{s_1^{'}, s_2^{'}, ..., s_n^{'}\}$. On average, the R-Squared coefficient for each Kernel regression model is 0.85 (median is 0.95) in our study.

• Step-3: Resampling. Next, one-dimensional linear interpolation [7] was used to resample our data, which returns one-dimensional piecewise linear interpolants to a function with given values at discrete data-points. For a given x_r, the value of $\hat{s_r}$ is computed by the following:

$$\hat{s_r} = \frac{s_i^{'} * (x_j - x_r) + s_j^{'} * (x_r - x_i)}{x_j - x_i} \qquad (2)$$

Where $(x_i, s_i^{'})$ and $(x_j, s_j^{'})$ represent two known points in our smoothed data, and the x_r is represented in the interval (x_i, x_j). For each project p, its resampling interval is set as $\frac{n}{20}$. Thus, after resampling, each project has a 20-dimensional vector $(\{\hat{s_1}, \hat{s_2}, ..., \hat{s_{20}}\})$ that characterizes its evolutionary trajectory. Since different projects may have different dockerfile scales, each project's vector value was normalized for comparison.

• Step-4: Clustering. Finally, the well known K-means algorithm [8] was used to cluster our dockerfile scale evolutionary trajectory vectors. K-means stores k centroids used to define clusters. A point is in a cluster if it is closer to that cluster's centroid than any other centroid. Based on the initial set $\hat{S} = \{\hat{S_1}, ..., \hat{S_i}, ..., \hat{S_N}\}, \hat{S_i} \in \mathbb{R}^{20}$ (N is the total number of projects), and a fixed positive integer, k, K-means can converge to a set of k cluster centroids, $C = \{C_1, C_2, .., C_k\}$ in $\mathbb{R}^{20}$, by minimizing the "K-means cost":

$$\Phi_S(C) = \sum_{S_i \in S} \min_{C_j \in C} ||S_i - C_j||^2. \qquad (3)$$

Afterwards, each S_i ($i \in N$) was marked with a specific cluster label, *i.e.*, Cluster-j ($j \in k$).

Our results. Data from two communities Docker Hub[3] and GitHub[4] were obtained (see Appendix B in the supporting information for details). Before performing our approach, the heatmap of the Dockerfile scale evolutionary trajectories of 2,840 projects was drawn (see Figure B1). By observing the color change trends, six clusters with significant differences were found. Thus, the number of clusters was set as k=6 in our K-means clustering process. After clustering all the projects' evolutionary vectors, we marked each project as belonging to one of the six categories (C1∼C6).

3) https://store.docker.com
4) https://github.com

For each cluster, the dockerfile scale variation curves of samples were drawn and case studies on randomly selected projects were conducted to validate the clusters. After our manual analysis, the specific paradigms describing the evolution trajectories of the six clusters were summarized, as follows (see App. C in the supporting information for details).

[C1] **Increasing and holding**. This cluster comprises 21.8% of the projects. In this paradigm, developers added new instructions or new settings to the dockerfile in early periods. However, after reaching to a certain size, they just updated the basic environment variables or changed the location of instructions. Thus, the dockerfile scale stayed stable.

[C2] **Constantly growing**. This cluster comprises 31.6% of the projects. In this paradigm, developers kept adding services, support, or plugins to the dockerfile, causing its scale to continually increase. Compared with other paradigms, the evolution path of this paradigm is easiest to understand.

[C3] **Holding and increasing**. This cluster comprises 19.2% of the projects. In this paradigm, developers just updated the basic environment variables in early periods, keeping the dockerfile scale stable for a while. Then, new instructions, plugins, or support were added, and the dockerfile scale increased.

[C4] **Increasing and decreasing**. This cluster comprises 10.2% of the projects. In this paradigm, developers kept adding new instructions in early periods, increasing the dockerfile scale. However, after reaching a large size, maintainers tried to reconstruct the dockerfile by moving useless plugins/services or moving settings to additional script files, reducing the dockerfile scale in later periods.

[C5] **Holding and decreasing**. This cluster comprises 9.5% of the projects. In this paradigm, developers change little in early periods and the dockerfile scale is stable. In later periods, developers tried to change the dockerfile structure by moving some instructions to additional script files or using a more light-weight base image, so the dockerfile scale dropped.

[C6] **Gradually reducing**. This cluster comprises 7.7% of the projects. In this paradigm, developers kept removing useless instructions or changed the base image to reduce image layers, continually decreasing the dockerfile scale. However, the decrease slows in later periods.

The differences between projects in the six clusters were further contemplated, and we found that they differ in project age and average dockerfile scale, indicating they have different project development stages and goals (see Appendix D in the supporting information for details).

Conclusion. This research studied the dockerfile evolution in Docker projects and proposed a clustering-based approach for mining dockerfile evolutionary trajectories that reveal projects of similar stages and goals. After performing an empirical study on 2,840 projects, six notable paradigms of the dockerfile scale evolutionary trajectories were obtained. Through case studies, each paradigm was summarized in detail. Our approach benefits future and current Docker project maintenance by providing a discrete characterization of dockerfile scale evolution over many existing projects, thus allowing co-localization of similar projects. Also, the proposed approach is generic and can be used to study general infrastructure configuration file evolution.

Acknowledgements This work was supported by the National Natural Science Foundation of China (Grant No. 61502512, 61432020), China Scholarship Council, and the USA NSF (Grant No. 1717370). Part of this study was performed during the visit in 2017 by the first author at the DECAL lab, UC Davis.

Supporting information Appendix A-D, and Figure B1. The supporting information is available online at info.scichina.com and link.springer.com. The supporting materials are published as submitted, without typesetting or editing. The responsibility for scientific accuracy and content remains entirely with the authors.

References

1 C. Anderson. Docker [software engineering]. IEEE Software, 2015, 32(3): pp. 102-c3.
2 D. Bernstein. Containers and cloud: From lxc to docker to kubernetes. IEEE Cloud Computing, 2014, 1(3): pp. 81-84.
3 D. Merkel. Docker: lightweight linux containers for consistent development and deployment. Linux Journal, 2014, 2014(239): p. 2.
4 D. Jaramillo, D. V. Nguyen, and R. Smart. Leveraging microservices architecture by using docker technology. In: Proceedings of SoutheastCon. IEEE, 2016. pp. 1-5.
5 N. S. Altman. An introduction to kernel and nearest-neighbor nonparametric regression. The American Statistician, 1992, 46(3): pp. 175-185.
6 G. C. Cawley and N. L. Talbot. Fast exact leave-one-out cross-validation of sparse least-squares support vector machines. Neural networks, 2004, 17(10): pp. 1467-1475.
7 M. J. Powell. A direct search optimization method that models the objective and constraint functions by linear interpolation. Advances in optimization and numerical analysis. Springer, 1994, pp. 51-67.
8 J. A. Hartigan and M. A. Wong. Algorithm as 136: A k-means clustering algorithm, Journal of the Royal Statistical Society. Series C (Applied Statistics), 1979, 28(1): pp. 100-108.