Development of Code Completion System for Dockerfiles

Kaisei Hanayama, Shinsuke Matsumoto, and Shinji Kusumoto

Containerization, in which multiple virtual servers (i.e., containers) are built on a single physical server, is widely employed for cost reduction and effective resource utilization. The object of this study is Docker, the de facto standard containerization platform. Containers in Docker are built by writing configuration scripts and creating files called Dockerfile. Managing the infrastructure as code makes it possible to apply knowledge gained from conventional software development to infrastructure configuration. However, infrastructure as code is a relatively new technology, some domains of which have not been fully researched. In this study, we focus on code completion and aim to construct a system that supports the development of Dockerfiles. The proposed system applies machine learning with long short-term memory to a pre-collected dataset to create language models and uses model switching to overcome a Docker-specific code completion problem. Evaluation experiments show that the implemented code completion system, Humpback, has a high average recommendation accuracy of 88.9%.

1 Introduction

Server virtualization is broadly used for cost reduction and efficient resource utilization. Containerization, a type of virtualization technology, has become mainstream [4]. Containerization creates logical compartments (i.e., containers) on the host operating system (OS). Each container provides an independent environment. Docker[†1] is the de facto standard containerization platform [2] [7].

Containers in Docker are configured by writing imperative instructions in files called Dockerfiles. The process of managing infrastructure configuration through machine-readable definition files is called infrastructure as code (IaC). IaC enables developers to manage infrastructure configuration in the same way as application code, which enables automated scaling, prevents human error, and allows the incorporation of know-how cultivated in software development [1]. However, IaC is a relatively new technology field and thus some areas are still in development [8], such as development support, static analysis, and best practices.

In this study, we focus on code completion, a widely used feature in software development [6]. A code completion system for an emerging technology such as Docker can considerably improve productivity by allowing the reuse of existing knowledge and reducing common errors.

One concern when building a Docker-specific code completion system is base image differences. A base image, which includes a Linux distribution, is an image file on which a container is created. A Dockerfile can have a nested language; embedded scripting languages (mainly bash) are described in a nested state in the top-level syntax [4]. The contents of Dockerfiles differ considerably depending on the base image. For example, for an Ubuntu base image, the `apt-get` command is used in the `RUN` instruction, whereas for a CentOS base image, the `dnf` command is used. For accurate code completion, base image differences must thus be taken into account.

Dockerfile に特化したコード補完システムの実装.

華山魁生, 柗本真佑, 楠本真二, 大阪大学大学院情報科学研究科, Graduate School of Information Science and Technology, Osaka University.

コンピュータソフトウェア, Vol.38, No.4 (2021), pp.53–59.
[研究論文 (レター)] 2021 年 1 月 14 日受付.

†1 https://www.docker.com/

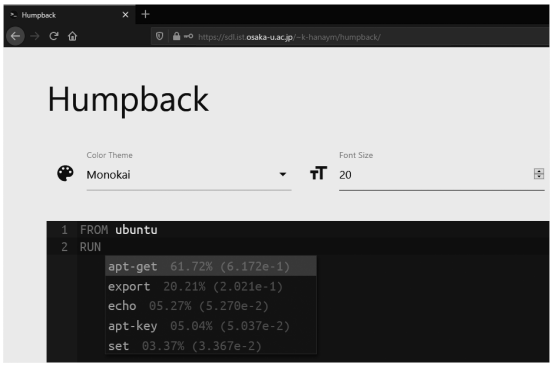

Fig. 1 Screenshot of Humpback

The contributions of this paper are as follows:

1. A solution to Docker-specific challenges is presented. We introduce model switching to overcome the problem caused by base image differences. With model switching, language models for prediction are selected based on the base image. In this study, the code completion system is realized by treating the contents of a Dockerfile as time-series data. Long short-term memory (LSTM) [3] is employed to generate language models (section 3. 2).

2. A Docker-specific code completion system, Humpback, is implemented. Figure 1 shows a screenshot of Humpback. Humpback is available online and can be used in a web browser.†2 Evaluation experiments show that Humpback has a high average accuracy of 88.9% and is useful for developing Dockerfiles. It is confirmed that model switching improves accuracy (section 4. 5).

2 Background

2. 1 Code completion

Code completion is extensively used in software development. Developers use code completion as frequently as several times a minute [6]. A pop-up dialog is used to display a list of candidate words after the user has typed some characters. Developers select the desired word from the list, which reduces typos and other common errors. Another benefit is the facilitation of the use of descriptive (i.e., long) names for variables, methods, and other entities. Manually entering long names is cumbersome and error-prone. These problems can be solved by automating the input process with code completion.

Traditional code completion systems display all candidate words, which means that developers must choose the appropriate one from an extremely long list. A large number of intelligent code completion systems have been proposed to overcome this problem [5][9]. Systems that use statistical language models such as N-gram, Transformer [10], and recurrent neural network (RNN)-based approaches have achieved high performance. Given token sequence w of length m, the language model applies the probability $P(w_1, ..., w_m)$ to the whole sequence. This probability indicates the relative likelihood of words, which allows the construction of code completion systems. Intelligent code completion systems consider the context and calculate probabilities based on language models to narrow the list of candidate words. Compared to a traditional code completion system, an intelligent one more effectively enhances developer productivity.

2. 2 Docker, infrastructure as code and challenges

Docker is an open containerization platform for developing, shipping, and running applications. Docker isolates applications from the development environment with containers, allowing quicker delivery of applications, improved portability, and efficient resource utilization. Due to its rapid rise in popularity, Docker has become the de facto standard container technology; over 87% of information technology companies use Docker [7]. Docker is also widely used in the open-source software community [2].

Containers in Docker can be built by interactively executing commands or by creating configuration files called Dockerfiles. A Dockerfile sets up containers through imperative instructions, enabling reproducible builds. Interest in IaC has thus grown among developers and researchers.

However, research on IaC is still in its infancy [8]. Most of the limited number of studies on IaC propose tools or frameworks for implementing or extending the practices of IaC itself. Knowledge in software engineering, such as that on development support, static analysis, and best practices, can be applied to IaC.

†2 https://sdl.ist.osaka-u.ac.jp/∼k-hanaym/humpback/

3 Humpback: Code completion system for Dockerfiles

3.1 System overview

We propose Humpback, a code completion system for Dockerfiles. Humpback helps developers to reduce errors and be more efficient when writing Dockerfiles. Various methods have been used to implement intelligent code completion systems. Here, we employ language models. Statistically processing pre-collected Dockerfiles and performing contextual predictions make it possible to reuse existing knowledge. We also introduce model switching to overcome the problem caused by base image differences.

3.2 Methodology

The methodology of Humpback is divided into the learning phase and the prediction phase.

3.2.1 Learning phase

The learning phase includes file collection, data processing, and language model generation. Figure 2 shows an overview of the learning phase.

File collection: The GitHub GraphQL API[†3] allows users to find repositories with specific programming language files. We search for repositories with Dockerfiles using the GitHub GraphQL API, pull these repositories in order of their star count (i.e., popularity), and extract the Dockerfiles.

Data processing: The contents of the collected Dockerfiles are divided into token sequences. The inputs are paired with the expected outputs. For example, for the statement `FROM centos RUN dnf`, `centos` is expected after `FROM` and `RUN` is expected after `FROM centos`. Next, these training data are encoded using integer values for the learner to interpret efficiently. The number of elements in the training data varies. Therefore, 0-padding is performed to obtain fixed-length data.

Language model generation: Humpback uses language models for word prediction. There are several types of language model, including N-gram, Transformer, and RNN. We assume that the contents of Dockerfiles are time-series data. LSTM [3], an RNN architecture used in the field of deep learning and natural language processing, is employed to generate language models. Although a standard (or *vanilla*) RNN can also process time-series data, it is incapable of storing long-term memory. LSTM is an improved version of RNN; it uses individual units in addition to standard units and can learn long-term dependencies.

To compare LSTM with conventional methods, we also created language models using N-gram. However, the average accuracies of them were inferior, and LSTM clearly showed better performance. Therefore, we propose an LSTM-based method in this paper.

3.2.2 Prediction phase

Humpback uses model switching to overcome the problem caused by base image differences. Figure 3 shows an overview of the prediction phase. Pretrained language models for each base image are prepared in advance. Humpback switches models for prediction considering the base image of input Dockerfiles. For instance, if the base image of input data is Ubuntu, a model trained with Dockerfiles, whose base images are Ubuntu, is used to make predictions.

However, in some cases, the Linux distribution cannot be identified from the base image name. For example, we can guess that "`openjdk:11-jdk`" includes the Java development environment, but cannot guess its Linux distribution. We created a base image detector to determine the Linux distribution for a given Dockerfile. First, the base image detector builds a container from the Dockerfile. Then, it identifies the distribution based on the file `/etc/os-release`, which contains OS information. Humpback can thus switch models for prediction even if the Linux distribution is not explicitly specified. For example, the base image detector identified the distribution of `openjdk:11-jdk` as Debian.

3.3 Implementation

Three libraries/frameworks are used to implement Humpback, namely TensorFlow[†4], a software library for machine learning, Keras[†5], a high-level neural network library, and Optuna[†6], a hyperparameter auto-optimization framework. Candidate words are presented immediately, and thus developers can use Humpback without slowing down their development process.

†3 https://docs.github.com/en/graphql

†4 https://www.tensorflow.org/

†5 https://keras.io/

†6 https://preferred.jp/en/projects/optuna/

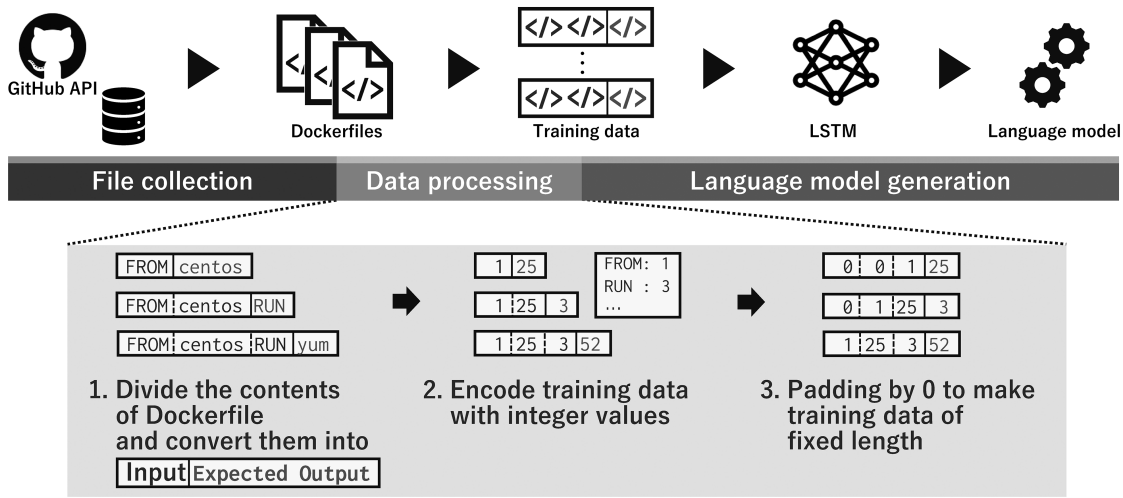

Fig. 2 Overview of learning phase

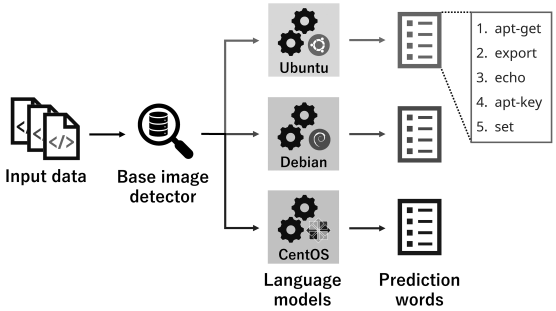

Fig. 3 Overview of prediction phase

4 Evaluation Experiment

4.1 Evaluation metrics

We conducted evaluation experiments to verify that model switching improves the accuracy of code completion. $Acc(k)$ (top-k accuracy) and the mean reciprocal rank (MRR) were used as metrics for evaluating recommendation accuracy:

$$Acc(k) = \frac{N_{top-k}}{|Q|} \tag{1}$$

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{2}$$

where N_{top-k} is the number of relevant recommendations in the top k suggestions, $|Q|$ is the total number of queries, and $rank_i$ is the rank position of the first relevant word for the i-th query. For both $Acc(k)$ and MRR, a value closer to 1 indicates better model performance. Table 1 shows an example of the evaluation metrics. In this case, $Acc(1)$ is $1/3 \approx 0.33$ and MRR is $(1+1/3+1/2)/3 = 11/18 \approx 0.61$.

Table 1 Example of evaluation metrics

Answer	Recommendations	Rank	Reciprocal rank	Top-1
RUN	**RUN**, FROM, CMD	1	1	✓
apt	dnf, apk, **apt**	3	1/3	
install	update, **install**, delete	2	1/2	

4.2 Dataset

We collected 21,190 Dockerfiles using the GitHub GraphQL API and applied the base image detector (section 3.2.2) to the whole dataset. The dataset contained 6,035 different base images. The Linux distribution of each base image was identified. The numbers of Dockerfiles and their versions for various Linux distributions are shown on the left side of Table 2. The major distributions in the dataset are Debian, Ubuntu, and Alpine Linux. The dataset for Ubuntu has the most variety, with 19 versions in 1,497 files. In the table, "Others" includes Amazon Linux, CentOS, Fedora, Oracle Linux Server, and VMware Photon OS.

4.3 Learning

We divided the dataset into training and testing sets (80/20 split on the file level). We also per-

Table 2 Details of dataset and learning parameters

Distribution	# of Dockerfiles	# of versions	# of epochs	Duration
Alpine	1,105 (5.2%)	9	56	4h39m
Debian	17,011 (80.2%)	6	71	1d7h33m
Ubuntu	1,497 (7.0%)	19	24	5h12m
Others	1,577 (7.4%)	-	-	-

formed 4-fold cross-validation during training. The number of epochs and the learning duration are shown on the right side of Table 2. Hyperparameters such as the activation function, optimization function, and number of units in each layer were optimized using Optuna.

4.4 Experiment design

We compared the recommendation accuracy for the three major distributions in the dataset, both with and without model switching. For the case without model switching, we created a generic model that was trained with all Dockerfiles. Humpback uses multiple language models with model switching, but this generic model used a single language model. All 21,190 Dockerfiles, without their distribution information, were used as training data for the generic model. Two syntaxes were defined; descriptions in the `RUN` instruction were defined as *Shell syntax* and other descriptions were defined as *Docker syntax*. There were three axes of comparison: presence or absence of data cleansing and model switching, the Linux distribution, and the syntax.

We first extracted 100 Dockerfiles from the dataset and set the correct answer to a random position in each Dockerfile. Next, the contents from the beginning of the file to just before the correct answer (i.e., seeds) were given to the language models. Seeds that were extremely short or whose base image could not be identified were excluded. Finally, $Acc(k)$ and MRR were computed by checking the candidate words against the correct answer. Ten rounds of the above process were performed for each comparison axis.

4.5 Experiment results

4.5.1 Overview

Table 3 shows the average scores of $Acc(1)$, $Acc(5)$, and MRR. "Gen." refers to the generic model (i.e., without model switching). "Hump." refers to Humpback (i.e., with model switching). The numbers in bold indicate the best scores in a given category.

Prediction with Humpback is more accurate for almost all evaluation axes. Humpback achieved a high average top-1 accuracy of 88.9%, (up to 97.8% for Debian with Docker syntax). As described in section 3.3, the candidate words are presented instantly. With its quickness and high accuracy, we believe Humpback can improve productivity.

4.5.2 Accuracy improvement through model switching

A comparison of the generic model with Humpback reveals that the top-1 accuracy is improved by up to 26.0% (for Ubuntu with Docker syntax). Prediction for Shell syntax was generally more accurate, with an average improvement of 6.8%. Model switching allows Humpback to consider the command differences between Linux distributions. Therefore, model switching is practical for all distributions in predicting Shell syntax, which leads to improved accuracy. These results show that model switching is useful for a code completion system for Dockerfiles.

4.5.3 Differences by distribution

Prediction for Debian is the most accurate for both types of syntax. As shown in Table 2, the number of Dockerfiles in Debian is 17,011, accounting for about 80% of the total. A large number of training data improve learning. The highest accuracy was achieved for Debian because a large number of training data were for Debian.

Prediction for Ubuntu was more accurate than that for Alpine Linux even though these distributions had very similar numbers of training data. It is assumed that many of the descriptions in Dockerfiles for Ubuntu were similar to each other and fewer similarities in the descriptions in Dockerfiles for Alpine. As a result of more efficient training, prediction for Ubuntu was more accurate than that for Alpine Linux.

5 Conclusion

In this study, we proposed Humpback, a code completion system for Dockerfiles. Humpback is available online and can be used in a web browser. We introduced model switching to overcome a Docker-specific problem and improve the prediction accuracy of Humpback. Evaluation experiments

Table 3 Average scores in experiment (Gen.: generic model, Hump.: Humpback)

Distribution	Docker syntax						Shell syntax					
	Top-1 accuracy		Top-5 accuracy		MRR		Top-1 accuracy		Top-5 accuracy		MRR	
	Gen.	Hump.	Gen.	Hump.	Gen.	Hump.	Gen.	Hump.	Gen.	Hump.	Gen.	Hump.
Alpine	79.9%	**82.1%**	86.1%	**86.6%**	0.827	**0.843**	75.8%	**83.7%**	84.9%	**87.9%**	0.798	**0.856**
Debian	97.4%	**97.8%**	98.4%	**98.7%**	0.979	**0.982**	95.7%	**95.8%**	98.8%	**98.9%**	0.971	**0.972**
Ubuntu	70.7%	**86.7%**	78.1%	**89.6%**	0.742	**0.878**	75.1%	**87.1%**	84.2%	**92.0%**	0.791	**0.893**

showed that Humpback has a high average top-1 accuracy of 88.9% and that model switching improves the accuracy of Humpback.

In future work, we will develop additional methods to improve the accuracy of Humpback further. One example of the improvement is to change the language model from LSTM to another language model, such as vanilla RNN and Transformer. As mentioned in section 3. 2, we have compared LSTM with N-gram but not checked for other language models. We can select an appropriate language model for Dockerfiles by implementing a code completion system for each language model and comparing their accuracy. We will also compare Humpback with other code completion systems. To the best of our knowledge, Humpback is the first Docker-specific code completion system. Existing research has targeted other programming languages, which utilize language features not found in Dockerfiles, such as type information and objects. Therefore, fair comparisons between Humpback and other systems will require ingenuity.

Acknowledgements This work was partially supported by MEXT/JSPS KAKENHI Grant No. 18H03222.

References

[1] Artac, M., Borovssak, T., Di Nitto, E., Guerriero, M., and Tamburri, D. A.: DevOps: Introducing infrastructure-as-code, in *International Conference on Software Engineering Companion*, 2017, pp. 497–498.

[2] Cito, J., Schermann, G., Wittern, J. E., Leitner, P., Zumberi, S., and Gall, H. C.: An Empirical Analysis of the Docker Container Ecosystem on GitHub, in *International Working Conference on Mining Software Repositories*, 2017, pp. 323–333.

[3] Gers, F. A., Schmidhuber, J., and Cummins, F.: Learning to forget: Continual prediction with LSTM, in *International Conference on Artificial Neural Networks*, Vol. 2, No. 470, 1999, pp. 850–855.

[4] Henkel, J., Bird, C., Lahiri, S. K., and Reps, T.: A Dataset of Dockerfiles, in *International Working Conference on Mining Software Repositories*, 2020, pp. 1–5.

[5] Kuraj, I. and Piskac, R.: Complete Completion using Types and Weights, *ACM SIGPLAN Notices*, Vol. 48, No. 6, jun 2013, pp. 27–38.

[6] Murphy, G. C., Kersten, M., and Findlater, L.: How are java software developers using the eclipse IDE?, *IEEE Software*, Vol. 23, No. 4 (2006), pp. 76–83.

[7] Portworx: Annual Container Adoption Report, 2019. https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf.

[8] Rahman, A., Mahdavi-Hezaveh, R., and Williams, L.: A Systematic Mapping Study of Infrastructure as Code Research, *Information and Software Technology*, Vol. 108 (2019), pp. 65–77.

[9] Raychev, V., Vechev, M., and Yahav, E.: Code Completion with Statistical Language Models, *ACM SIGPLAN Notices*, Vol. 49, No. 6 (2014), pp. 419–428.

[10] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I.: Attention is all you need, *Advances in Neural Information Processing Systems*, 2017, pp. 5999–6009.

Kaisei Hanayama

received the BI degree from Osaka University in 2019. He is currently a master's course student in the Graduate School of Information Science and Technology at Osaka University. His research interests include source code analysis and its applications.

Shinsuke Matsumoto

received the ME and DE degrees from Nara Institute of Science and Technology in 2008 and 2010, respectively. He is currently an assistant professor in the Graduate School of Information Science and Technology at Osaka University. His research interests include empirical software engineering.

Shinji Kusumoto

received the ME and DE degrees in Information and Computer Sciences from Osaka University in 1990, and 1993, respectively. He is currently a professor in the Graduate School of Information Science and Technology at Osaka University. His research interests include software metrics and software quality assurance technique. He is a member of the IPSJ, IEICE, JSSST, IEEE, JFPUG, SPM, and SEA.