Humpback: Code Completion System for Dockerfiles Based on Language Models

Kaisei Hanayama[a], Shinsuke Matsumoto[a] and Shinji Kusumoto[a]

[a]*Graduate School of Information Science and Technology, Osaka University, Suita, Osaka, 565–0871, Japan*

Abstract

The object of this study is Docker, the de facto standard containerization platform. Containers in Docker are built by creating files called Dockerfiles. Managing the infrastructure as code makes it possible to incorporate knowledge gained from conventional software development. 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 code completion system, Humpback, applies machine learning to a pre-collected dataset with long short-term memory to create language models and uses model switching to overcome a Docker-specific code completion problem. Evaluation experiments show that Humpback has a high average accuracy of 96.9%.

Keywords

Docker, code completion, machine learning, language model, long short-term memory

1. Introduction

Server virtualization is broadly used for cost reduction and efficient resource utilization. Among various methods of virtualization, containerization has become mainstream [1]. Containerization creates logical compartments (i.e., containers) on the host operating system. Each container provides an independent environment.

Docker[1] is the de facto standard containerization platform [2]. 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, allowing automated scaling and the prevention of human error [3]. However, IaC is a relatively new technology and thus some areas are still in progress [4], such as development support and static analysis.

In this study, we focus on code completion, a widely used feature in software development [5]. We believe that providing a code completion system for emerging technology such as Docker can considerably improve productivity by reusing existing knowledge and reduce common errors.

Joint Proceedings of SEED & NLPaSE co-located with APSEC 2020, 01-Dec, 2020, Singapore k-hanaym@ist.osaka-u.ac.jp (K. Hanayama); shinsuke@ist.osaka-u.ac.jp (S. Matsumoto); kusumoto@ist.osaka-u.ac.jp (S. Kusumoto)

© 2020 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

[1]https://www.docker.com/

CEUR-WS.org/Vol-2799/Paper8_NLPaSE.pdf

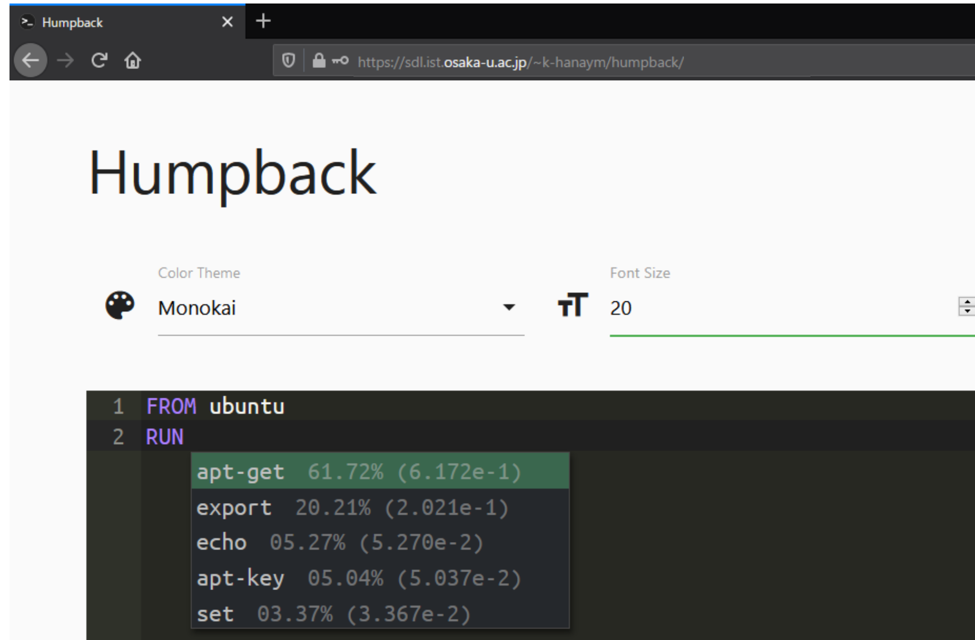

Figure 1: Screenshot of Humpback

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 containers are created. Dockerfiles can have a nested language; embedded scripting languages (mainly bash) are described in a nested state in the top-level syntax [1]. The contents of Dockerfiles differ considerably depending on the base image. For example, for an base image includes Ubuntu, 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.

The contributions of this paper are as follows:

1. A solution to the Docker-specific challenge is presented. We introduce model switching to overcome the problem caused by base image differences. With model switching, language models for predictions are switched depending on the base image. Long short-term memory (LSTM) [6] is employed to generate language models (section 3.2).

2. A novel 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 accuracy of 96.9% and is useful for developing Dockerfiles (section 4.4).

[2]https://sdl.ist.osaka-u.ac.jp/~k-hanaym/humpback/

2. Background

2.1. Code completion

Code completion is extensively used in software development [5]. A pop-up dialog is used to display candidate words after the user has typed some characters. Developers select the desired word from the list, reducing typos and other common errors. Another benefit is the faciliation of the use of descriptive (i.e., long) names for variables. Manually entering long variable names is cumbersome and error-prone.

Traditional code completion systems display all candidates, which can be extremely long list. A large number of intelligent code completion systems have been proposed to overcome this problem [7]. Systems that use statistical language models such as N-gram and recurrent neural network (RNN)-based approaches have achieved high performance. Given token sequence w of length m, the language model gives the probability $P(w_1, ..., w_m)$. This probability indicates the relative likelihood of words, which allows the construction of code completion systems. Intelligent code completion considers the context and calculates 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, an open containerization platform, isolates applications from the development environment with containers, allowing efficient resource utilization. Docker has become the de facto standard container technology; over 87% of information technology companies use Docker [2].

Containers in Docker can be built by interactively executing commands or by creating configuration files called Dockerfiles. Dockerfiles set up containers through imperative instructions, enabling reproducible builds. A process for specifing the environment in which software systems will be tested and/or deployed by configuration scripts is called IaC. Developers can manage infrastructure configuration in the same way as application code, allowing automated scaling and the prevention of human error [3]. Interest in IaC has thus grown [8].

Research on IaC is still in its infancy [4]. There are relatively few studies on IaC, and most of them propose tools for implementing the practices of IaC itself. Knowledge in software engineering, such as that on development support and static analysis, can be applied to IaC.

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 enhance efficiency when writing Dockerfiles. Various methods have been used to implement code completion systems. Here, we employ language models. Statistically processing pre-collected Dockerfiles and performing contextual predictions makes it possible to reuse existing knowledge. We also introduce model switching to overcome the problem, caused by base image differences.

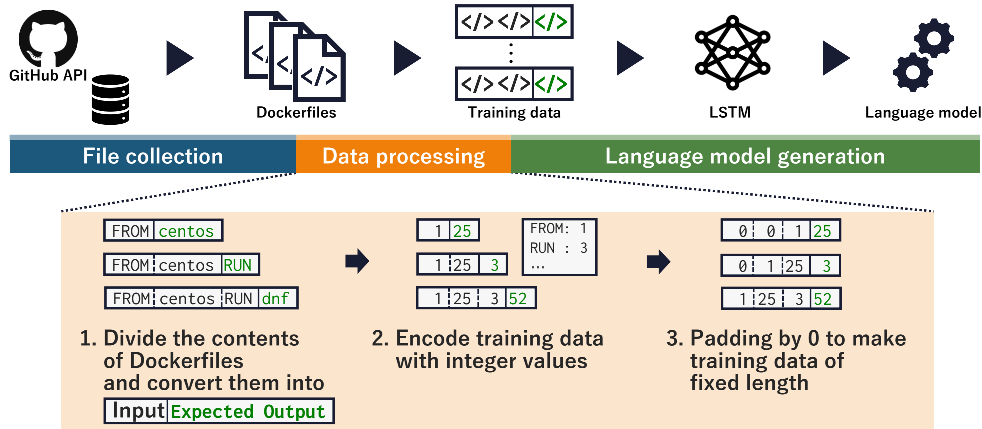

Figure 2: Overview of learning phase

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: We search for repositories with Dockerfiles using the GitHub API[3], 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, if there is a statement `FROM centos RUN dnf`, then `FROM` expects `centos` and `FROM centos` expects `RUN`. Next, these 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 prediction. We assume that the contents of Dockerfiles are time-series data. LSTM [6], an improved RNN architecture used in the field of deep learning, is employed to generate language models. The middle layer of the RNN is replaced with LSTM blocks, which allow for learning with long-term dependency.

3.2.2. Prediction phase

Humpback uses model switching to overcome the problem caused by base image differences. Pre-trained language models for each Linux distribution are prepared in advance. Humpback switches models for prediction depending on the input data. For instance, if the base image of input data is Ubuntu, a model trained with Dockerfiles whose base images are Ubuntu is used.

[3]https://docs.github.com/en/graphql

Table 1
Details of dataset and learning

Distribution	# of Dockerfiles	# of versions	# of epochs	Duration
Debian	17,011 (80.2%)	6	81	3d15h12m
Ubuntu	1,497 (7.0%)	19	55	2h30m
Alpine	1,105 (5.2%)	9	94	3h00m
Others	1,577 (7.4%)	-	-	-

However, it is impossible to identify the Linux distribution from the base image name in some cases. For example, we can guess that "`openjdk:11-jdk`" will include 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 `/etc/os-release` file. We analyzed the base images of the entire dataset (section 4.2). With these results, Humpback can switch models for prediction even if the Linux distribution is not explicitly specified. For example, the base image detecor 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, thus developers can use Humpback without slowing down their development process.

4. Evaluation Experiment

4.1. Evaluation metrics

We conducted evaluation experiments to verify that model switching improves the accuracy of code completion. Top-k accuracy ($Acc(k)$) and the mean reciprocal rank (MRR) [9] are used as metrics for evaluating accuracy:

$Acc(k) = \frac{N_{top-k}}{|Q|}, MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i}$ where N_{top-k} refers to the number of relevant recommendations in top k suggestions, $|Q|$ represents the total number of queries, and $rank_i$ denotes 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.

4.2. Dataset

We collected 21,190 Dockerfiles using the GitHub API. The numbers of Dockerfiles and their versions for various Linux distributions are shown on the left side of Table 1. The major

[4]https://www.tensorflow.org/
[5]https://keras.io/
[6]https://preferred.jp/en/projects/optuna/

Table 2
Average scores in experiment (Gen.: generic model, Hump.: Humpback)

Distribution	Docker syntax					
	Top-1 accuracy		**Top-5 accuracy**		**MRR**	
	Gen.	Hump.	Gen.	Hump.	Gen.	Hump.
Alpine	94.2%	**96.2%**	**98.3%**	98.0%	0.9611	**0.9706**
Debian	97.2%	**98.8%**	98.0%	**99.5%**	0.9762	**0.9913**
Ubuntu	91.9%	**96.9%**	98.9%	**99.4%**	0.9542	**0.9810**
	Shell syntax					
	Top-1 accuracy		**Top-5 accuracy**		**MRR**	
	Gen.	Hump.	Gen.	Hump.	Gen.	Hump.
Alpine	92.4%	**94.4%**	96.8%	**96.9%**	0.9433	**0.9565**
Debian	95.5%	**97.5%**	98.3%	**99.5%**	0.9689	**0.9843**
Ubuntu	96.6%	**97.7%**	98.9%	**99.5%**	0.9766	**0.9855**

distributions in the dataset are Alpine Linux, Debian GNU/Linux, and Ubuntu. 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/Linux.

The number of epochs and the learning duration are shown on the right side of Table 1. Hyperparameters such as the activation/optimization function, and number of units in each layer were optimized using Optuna.

4.3. Experiment design

There were three axes of comparison: the presence or absence of model switching, the Linux distribution, and the syntax. 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. Two syntaxes were defined; descriptions in the RUN instruction were defined as *Shell syntax* and other descriptions were defined as *Docker 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 were given to the language models and predictions were generated. Then $Acc(k)$ and MRR were computed by comparing the predictions against the correct answer. Ten rounds of the above process were performed for each comparison axis.

4.4. Experiment results

Table 2 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 axis. Model switching is thus beneficial for building Docker-specific code completion systems. Humpback achieved an outstanding average Top-1 accuracy of 96.9% (up to 98.8% for Debian, Docker syntax). Moreover, the accuracy improved by up to 5.0% (Ubuntu, Docker syntax) compared to that for the generic

model. As described in section 3.3, the candidate words are presented instantly. With its quickness and high accuracy, Humpback can significantly improve productivity.

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. Evaluation experiments showed that Humpback has a high average accuracy of 96.9%, and that model switching improves the accuracy of Humpback. In future work, we will further improve the accuracy of Humpback and compare Humpback with other code completion systems.

Acknowledgments

This work was supported in part by MEXT/JSPS KAKENHI Grant No. 18H03222.

References

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

[2] Portworx, Container adoption survey, 2019. URL: https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf.

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

[4] A. Rahman, R. Mahdavi-Hezaveh, L. Williams, A systematic mapping study of infrastructure as code research, Information and Software Technology 108 (2019) 65–77.

[5] M. Bruch, M. Monperrus, M. Mezini, Learning from examples to improve code completion systems, in: European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2009, pp. 213–222.

[6] F. A. Gers, J. Schmidhuber, F. Cummins, Learning to forget: continual prediction with lstm, in: International Conference on Artificial Neural Networks, volume 2, 1999, pp. 850–855.

[7] A. Svyatkovskiy, S. Fu, Y. Zhao, N. Sundaresan, Pythia: Ai-assisted code completion system, in: International Conference on Knowledge Discovery and Data Mining, 2019, pp. 2727–2735.

[8] C. Parnin, E. Helms, C. Atlee, H. Boughton, M. Ghattas, A. Glover, J. Holman, J. Micco, B. Murphy, T. Savor, M. Stumm, S. Whitaker, L. Williams, The top 10 adages in continuous deployment, IEEE Software 34 (2017) 86–95.

[9] D. R. Radev, H. Qi, H. Wu, W. Fan, Evaluating web-based question answering systems, in: International Conference on Language Resources and Evaluation, 2002, pp. 1153–1156.