一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
SS2020-8,DC2020-25(2020-10)

言語モデルに基づくDockerfileコード補完システムの提案

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 〒 565–0871 大阪府吹田市山田丘 1–5
E-mail: †{k-hanaym,shinsuke,kusumoto}@ist.osaka-u.ac.jp

あらまし　コスト削減や計算資源の有効活用を目的として，1 台の物理サーバ上に複数の仮想サーバ（コンテナ）を構築するコンテナ型仮想化が広く利用されている．本研究の対象は，コンテナ型仮想化技術でデファクトスタンダードとなっている，Docker と呼ばれるプラットフォームにある．Docker では，スクリプトを記述し Dockerfile と呼ばれるファイルを作成することで，コンテナの構築を行う．インフラ構成を計算機が解読可能なファイルにより管理することで，ヒューマンエラーの防止や管理の自動化を行えるほか，従来のソフトウェア開発で得られた知見をインフラ構成に応用できる．しかし，このような比較的新しい技術においては，開発支援や静的解析など研究が未熟な領域が存在する．本研究ではコード補完に着目し，Dockerfile 作成を支援するシステムの構築を目指す．提案を実現するため事前に収集したデータセットを用いて機械学習を行い，言語モデルを作成し，コード補完システム Humpback を作成した．さらに，推薦精度向上を目指し，Docker 固有の問題を解決するための手法を導入した．評価実験の結果，Humpback は平均で 93.0%という高い推薦精度を持つことが示された．

キーワード　Docker，コード補完，言語モデル，機械学習，LSTM

1. はじめに

コスト削減や計算資源の有効活用を目的として，サーバ仮想化技術が広く利用されている．仮想化には様々な方法があるが，近年はコンテナ型仮想化が主流となっている [1]．コンテナ型仮想化では，ホスト OS 上に論理的な区画（コンテナ）を作成し，独立した環境を提供する．Docker [2] はコンテナ型仮想化でデファクトスタンダードとなっている技術であり，IT 企業の 87%以上が Docker を利用しているほか，様々な OSS コミュニティでも採用されている [3] [4].

Docker におけるコンテナは，*Dockerfile* と呼ばれるファイルに手続き的な命令を書き連ねて構成される．このように，インフラ構成を計算機が解読可能なファイルを通して行うプロセスを *Infrastructure-as-Code* (IaC) と呼ぶ [5]．IaC の活用によりインフラ構成をアプリケーションコードと同等に管理できるため，スケーリングと監視の自動化を行えるだけでなく，ソフトウェア開発で培われてきた知見をインフラ構成に導入できる [6]．しかし，IaC は比較的新しい技術分野であり，開発支援や静的解析など研究がまだ未熟な領域も存在する [7].

本研究ではそのような技術領域の中でも，コード補完に着目した．コード補完とは，文字列の入力中に次の字句を推測し補完候補のリストを表示する機能であり，ソフトウェア開発で頻繁に用いられる [8] [9]. Docker のような発展途上の技術に特化したコード補完システムを提供することで，ヒューマンエラーの低減だけでなく，事前収集した Dockerfile の統計処理による過去の知見の再利用にも繋がると考えた．

本研究の貢献は以下の通りである．

（1） **コード補完システムの実現に向けた Docker の課題点を整理した**．Docker のベースイメージでは，コンテナ作成の基となる Linux ディストリビューションを指定するため，Dockerfile の内容はベースイメージによって大きく異なる．その違いを考慮してコード補完を行わなければ，良い精度は望めない（2.3 節）．

（2） **上記の課題点に対する解決策を示した**．Docker 固有の課題点を解決するため，予測を行うための言語モデルを Dockerfile のベースイメージによって変更する，モデルスイッチングという手法を導入した．本研究では，Dockerfile の内容を時系列データとして扱い，コード補完システムを実現している．なお，言語モデルの生成には Long Short-Term Memory (LSTM) [10] を使用した (3.2 節).

（3） **Dockerfile に特化した新たなコード補完システム Humpback を実装した**．図 1 は Humpback のスクリーンショットである．Humpback はオンラインエディタとして実装されており，ブラウザからすぐに使用できる．[1] 予測単語は瞬時に提示されるため，開発者は開発速度を落とすことなく快適に Humpback を活用できる．評価実験の結果，Humpback の推薦精度は平均で 93.0%と非常に高く，Dockerfile の開発支援において有用であると示された．また，我々の導入した手法が精度向上に有効であることも確認した (4.4 節).

（注1）：https://sdl.ist.osaka-u.ac.jp/~k-hanaym/humpback/

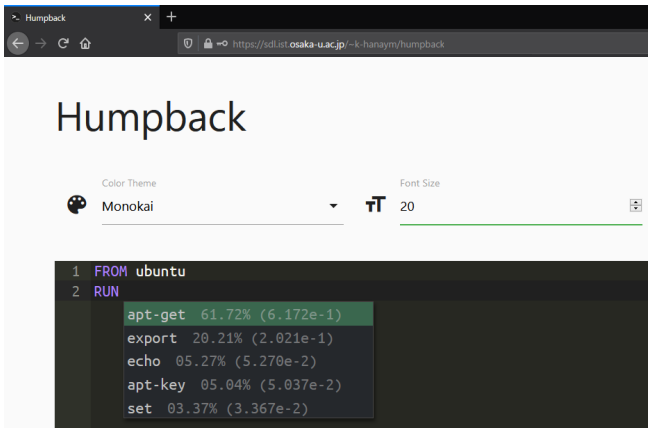

図 1: Humpback のスクリーンショット

2. 背　　景

2.1 コード補完

コード補完とは，文字列の入力中に次の字句を推測し，ポップアップを使用して補完候補のリストを表示する機能である．開発者はそのリストから入力値を選択することで，誤字脱字などの一般的なエラーを削減できる．コード補完を利用するその他の利点として，より長く記述的な名前を変数や関数などに付与できるようになることが挙げられる．長い名前を付与するとプログラム理解やバグの削減に効果が期待できる [11] 一方，それらを手動で入力するのは煩わしく間違いも起きやすい．しかし，コード補完を使用した入力処理の自動化により，それらの問題が解消される．ソフトウェア開発において，開発者はコード補完を頻繁に利用しており [8]，その回数は 1 分間に数回にも及ぶことが知られている [9].

しかし，伝統的なコード補完では全ての補完候補を表示するため，開発者が非常に長いリストの中から入力値を適切に選ばなくてはならない．この課題を解決するために，多くのインテリジェントコード補完が提案されている [12] [13] [14] [15]．N-gram のような統計的言語モデル，Best Matching Neighbor (BMN) や Recurrent Neural Network (RNN) などは，インテリジェントコード補完システムを構築するのに高い性能を発揮してきた．言語モデルは長さ m の単語列 w が与えられたとき，その生起確率 $P(w_1,...,w_m)$ を与える．異なる単語に対してこの生起確率を算出することで，相対的な尤度を得られる．そしてインテリジェントコード補完は文脈を考慮し，言語モデルを用いて予測単語から可能性のある単語を絞り込むことで，従来のコード補完よりも効果的に開発者の生産性を向上できる．

2.2 Docker，Infrastructure-as-Code とその課題

Docker は，アプリケーションを開発・出荷・実行するためのオープンプラットフォームである [2]．コンテナを用いた OS レベルの仮想化によりアプリケーションを開発・実行環境から隔離し，アプリケーションの素早い提供や，可搬性の向上を可能とする．その性質から Docker の人気は急速に高まっており，現在 Docker はコンテナ型仮想化でデファクトスタンダードとなっている．IT 企業の 87%以上が Docker を利用しているほか，様々な OSS コミュニティでも採用されている [3] [4].

Docker におけるコンテナはコマンドを毎回実行するだけでなく，*Dockerfile* と呼ばれるファイルを作成することでも構築できる．Dockerfile を作成しコンテナの構成情報を手続き的な命令を通して設定することで，再現性のあるビルドを可能にしている [16]．このように，ソフトウェアシステムのテストやデプロイを行う環境をスクリプトで設定し，自動化するプロセスを *Infrastructure-as-Code* (IaC) と呼ぶ [5]．インフラ構成をアプリケーションコードと同じように管理できるため，ヒューマンエラーの防止やスケーリングと監視の自動化だけでなく，ソフトウェア開発で培ったノウハウの応用が可能となる [6]．その特徴から，近年は実務者と研究者ともに IaC への興味関心が高まっている [17] [18].

しかし，注目の高まりとは反対に，IaC に関する研究には未熟な領域が存在する [7]．既存研究も少なく，さらにそれらの大部分は IaC 自体の手法を実装したり拡張したりするツールの提案をしているに過ぎない．すなわち開発支援や静的解析などの未開拓な技術領域においては，ソフトウェア工学で培った知識を IaC に活かす余地があると言える．

2.3 IaC におけるコード補完に向けた課題

前節で述べた技術領域の中で，本研究ではコード補完に着目した．Docker 固有のコード補完システムを構築する上で考慮すべき課題として，ベースイメージの違いがある．ベースイメージとは，コンテナを作成する基となるイメージファイルであり，Dockerfile の中で `FROM` 命令によって指定する．Linux ディストリビューションがここで決定されるため，どのベースイメージを指定するかによって Dockerfile の内容が大きく異なる．また，Dockerfile は入れ子構造を持っており，上位の構文に入れ子になった状態で埋め込み言語（主に bash）が記述される [1]．例えばベースイメージに Ubuntu を指定した場合は，コンテナ内でコマンドを実行する `RUN` 命令の中で `apt-get` コマンドを使用し，CentOS を指定した場合は `yum` コマンドを使用するといった具合である．そのためベースイメージの違いを考慮しなければ，高い精度でのコード補完は難しいと言える．

3. コード補完システム：Humpback

3.1 システムの概要

本研究では，Dockerfile に特化したコード補完システムを提案する．この提案を実現するため，コード補完システム Humpback の実装を行った．既存研究におけるコード補完システムの実装には様々な手法が用いられているが，本研究では機械学習による手法を採用している．事前に収集した Dockerfile を統計処理し，文脈を考慮した予測を行うことで，過去の知見を再利用できる．また，2.3 節で述べたベースイメージの違いによる問題を解決するため，モデルスイッチングという手法を考案した．

3.2 方　法　論

Humpback の方法論について，学習段階と予測段階に分けて説明する．

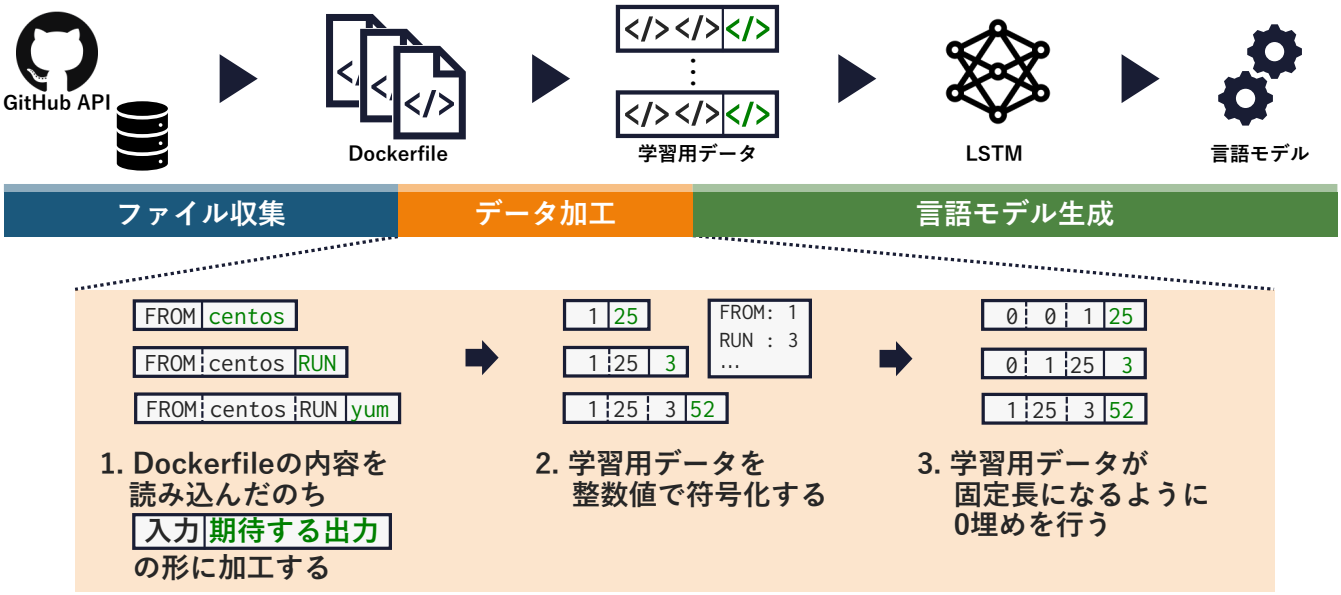

図 2: 学習段階の全体像

3.2.1 学習段階

図 2 は学習段階の全体像を示している．学習段階について，ファイル収集，データ加工，言語モデル生成の順に説明する．

ファイル収集

GitHub API[2]は，特定のプログラミング言語で記述されたファイルを含む GitHub 上のリポジトリを検索できる．GitHub API を用いて Dockerfile を含むリポジトリを検索し，人気度を示す Star 数の多い順にそれらをプルしたのち Dockerfile だけを抽出した．

データ加工

収集した Dockerfile を読み込み，その内容を空白や改行で区切ってトークン列とする．その際，入力と期待する出力が対になるように加工を行う．例えば `FROM centos RUN yum` という記述があった場合，`FROM` という入力に対して `centos` が対応し，`FROM centos` という入力に対しては `RUN` が対応する．次に，学習器が効率的に学習用データを解釈できるように整数値で符号化する．この状態では各データの要素数が異なるため，最後に全データの要素数が固定長になるように 0 埋めを行う．

言語モデル生成

Humpback では，推薦単語の予測に言語モデルを使用している．言語モデルには N-gram や BMN，RNN といった種類があるが，本研究では Dockerfile の内容を時系列データとみなし，言語モデルの生成に Long Short-Term Memory (LSTM) [10] を用いた．LSTM とは自然言語処理の分野においてよく用いられる RNN アーキテクチャである．古典的な RNN でも時系列データを処理できるが，長期の記憶保存ができないという欠点があった．LSTM は RNN の中間層を LSTM ブロックと呼ばれるユニットに置き換えることで，長期にわたる依存を持つ学習を可能としている．

3.2.2 予測段階

図 3 は予測段階の全体像を示している．Humpback による予測では，ベースイメージの違いによる問題に対する解決策として，モデルスイッチングという手法を活用している．ベースイメージを考慮した補完を行うため，各ベースイメージに対して事前学習済みのモデルを用意しておく．そして Humpback は入力 Dockerfile のベースイメージによって，予測を行うモデルを切り替える．例えば入力 Dockerfile のベースイメージが Ubuntu であれば，ベースイメージが Ubuntu である Dockerfile のみで学習したモデルを用いて予測を行う．

しかし，ベースイメージ名から Linux ディストリビューションを判別できない場合もある．例えば `openjdk:11-jdk` というベースイメージ名に Java の開発環境が含まれていると推測できるが，どのディストリビューションが使われているかは確認できない．そこで，各ベースイメージにおいて使用されているディストリビューションを解析するために，ベースイメージ検出器を作成した．ベースイメージ検出器は入力 Dockerfile からコンテナを作成したのち，`/etc/os-release` に記述されているディストリビューションの識別子を確認する．この解析結果によって，ディストリビューションが明示されていない場合でも，Humpback がモデルスイッチを実行できるようになる．なお，`openjdk:11-jdk` の Linux ディストリビューションは Debian GNU/Linux である．

3.3 データクレンジング

より高い精度を持つコード補完システムを構築するために，データクレンジングを実施した．データクレンジングとは，データの品質を向上させる処理である [19]．データセットに含まれる重複やエラー，表記ゆれなど検知し，削除や修正により正規化を行う．本研究では Dockerfile 特有のデータクレンジングとして，抽象化，具体化，そしてノイズ除去を行った．図 4 は各データクレンジングの例を示している．

（注2）：https://docs.github.com/en/graphql

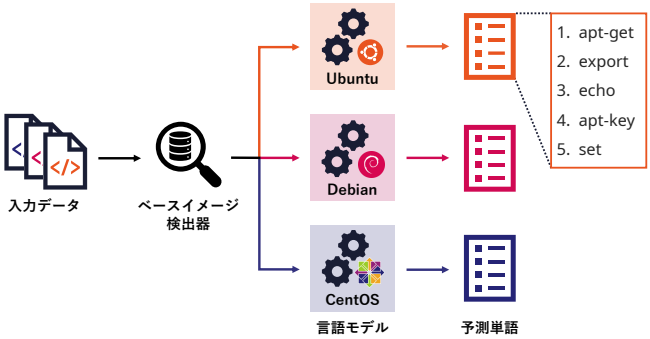

図 3: 予測段階の全体像

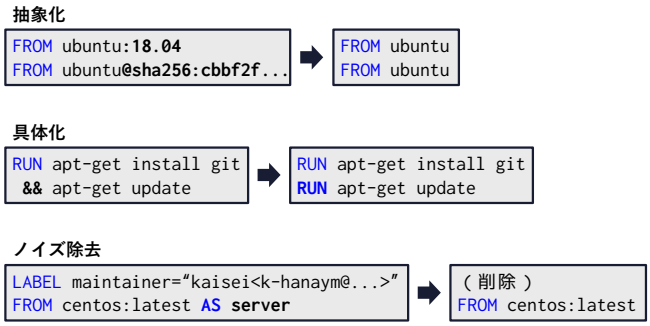

図 4: データクレンジングの例

抽　象　化

抽象化の例としては，ベースイメージのタグとダイジェストの消去が挙げられる．FROM 命令では，ベースイメージ名の他にバージョンの指定を行うタグとダイジェストを記述できる．しかし，ベースイメージのバージョンの違いによって使用するコマンド（apt-get など）に大きな差はないため，その違いを吸収して学習効率を向上できると考えた．

具　体　化

具体化の例としては，&&の直前の命令への置換が挙げられる．RUN 命令内で&&を使用すると，複数のコマンドを一つの RUN 命令にまとめて記述できる．RUN 命令は実行される度にイメージファイルのレイヤが増えるため，可能な限り&&を使用しサイズ増加を抑えることが推奨されている [20]．しかし，学習用データの作成という観点では，&&の示している命令を明示することによって，より正確な学習が可能になると考えた．

ノ イ ズ 除 去

ノイズ除去の例としては，学習に不必要な作者名やバージョンなどのメタ情報，およびビルドステージの別名指定などの消去が挙げられる．Dockerfile では LABEL 命令によってメタ情報を設定したり，AS 命令によって中間イメージに名前を付けられる．しかしこれらの情報は，言語モデルが Dockerfile の内容を学習する上では余分であると考え消去した．

3.4 実　　装

Humpback の実装に当たっては，言語モデルの獲得とオンラインエディタの構築を行った．言語モデルの獲得には 3 つのライブラリおよびフレームワークを活用した．TensorFlow[3]は機械学習のソフトウェアライブラリ，Keras[4]は高水準のニューラルネットワークライブラリ，そして Optuna[5]はハイパーパラメータ自動最適化フレームワークである．

オンラインエディタの構築には，獲得した言語モデルを展開するための JavaScript ライブラリである TensorFlow.js[6]，およびブラウザで動作するエディタの Ace[7]を使用した．図 1 に示す通り，Humpback はブラウザからすぐに使用できる．カラーテーマの変更やフォントサイズの変更など，通常のエディタと同様の使用感で開発を行えるが，文字列を入力すると Humpback に組み込み済みの言語モデルを利用した予測を行う．その予測単語は瞬時に提示されるため，開発者は開発速度を落とすことなく快適に Humpback を活用できる．

表 1: 評価メトリクスの例

正解	予測単語	ランク	ランクの逆数	Top-1 精度
RUN	**RUN**, FROM, CMD	1	1	✓
apt	yum, apk, **apt**	3	1/3	
install	update, **install**, delete	2	1/2	

4. 評 価 実 験

4.1 評価メトリクス

我々の導入した手法が Humpback の推薦精度向上に寄与しているか確かめるために，評価実験を行った．$Acc(k)$（Top-k 精度）と Mean Reciprocal Rank (MRR) [21] を精度評価のメトリクスとして使用した：

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

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

ここで，$|Q|$ はテストデータのサンプル数の合計，N_{top-k} は $|Q|$ 回の予測における上位 k 位の予測単語のうち正答単語数，そして $rank_i$ は i 番目のクエリに対する最初の正答単語の位置を表す．$Acc(k)$ と MRR のいずれにおいても，その値が 1 に近いほど性能が良いことを示す．表 1 に評価メトリクスの例を示す．この場合，$Acc(1)$ は $1/3 \approx 0.33$ であり，MRR は $(1 + 1/3 + 1/2)/3 = 11/18 \approx 0.61$ である．

4.2 データセット

GitHub API を用いて 21,190 個の Dockerfile を収集した．各 Linux ディストリビューションにおける Dockerfile 数とバージョン数を表 2 の左側に，学習エポック数と学習時間を表 2 の右側に示す．データセットの中で主要なディストリビューションは Alpine Linux，Debian GNU/Linux，Ubuntu である．Ubuntu のデータセットは最も多様なバージョンから構成されており，

（注3）：https://www.tensorflow.org/

（注4）：https://keras.io/

（注5）：https://preferred.jp/en/projects/optuna/

（注6）：https://www.tensorflow.org/js

（注7）：https://ace.c9.io/

表 2: データセットと学習の詳細

Linux	Dockerfile 数	バージョン数	エポック数	学習時間
Alpine	1,105 (5.2%)	9	94	3h00m
Debian	17,011 (80.2%)	6	81	3d15h12m
Ubuntu	1,497 (7.0%)	19	55	2h30m
その他	1,577 (7.4%)	-	-	-

1,497 ファイルに 19 個のバージョンが含まれている．その他のディストリビューションには Amazon Linux，CentOS，Fedora，Oracle Linux Server，VMware Photon OS/Linux が含まれている．学習における活性化関数，最適化関数や各層のユニット数といったハイパーパラメータは，それぞれ Optuna によって最適になるよう調整されている．

4.3 実験設計

3 つの Linux ディストリビューション Alpine，Debian，そして Ubuntu を対象として，我々の導入した手法の有無で推薦精度に変化が生じるかを確かめた．比較対象として，データクレンジングとモデルスイッチングのいずれも行わないモデル（Nothing モデル）と，データクレンジングは行うがモデルスイッチングは行わないモデル（Normal モデル）を準備した．Humpback はモデルスイッチングにより複数の言語モデルを使い分けるが，Nothing モデルと Normal モデルはともに単一の言語モデルである．両モデルの学習用データには，ディストリビューションによって区別していない 21,190 個全ての Dockerfile を使用した．

また，`RUN` 命令内の記述を Shell 構文，それ以外の記述を Docker 構文と定義し，それぞれ推薦精度の確認を行った．なお，Docker 構文にはコンテナ内にファイルをコピーする `COPY` 命令や，外部に公開するポートを指定する `EXPOSE` 命令などが含まれる．すなわち本実験における評価軸は，データクレンジングとモデルスイッチングの有無，Linux ディストリビューション，そして構文の違いである．

テスト用データの作成において，まずは収集済みデータセットから 100 個の Dockerfile を抽出し，各 Dockerfile においてランダムな位置で正解単語を 1 つ選ぶ．次に，ファイルの先頭から正解単語の手前までの内容を言語モデルに与え，予測単語を生成する．そして予測単語を正解単語と照らし合わせて，$Acc(k)$ と MRR を計算する．ここまでの操作を 1 ラウンドとし，各評価軸ごとに実験を 10 ラウンド行った．

4.4 実験結果

4.4.1 概　　要

表 3 は Top-1 精度 $Acc(1)$ と Top-5 精度 $Acc(5)$，そして MRR の平均値を示している．ここで，NTG は Nothing モデル，NML は Normal モデル，そして HB は Humpback による予測を表している．太字になっている数値は同カテゴリにおいて最良の値である．表 3 が示す通り，ほぼ全ての評価軸において Humpback，Normal モデル，Nothing モデルの順に高い精度を持つという結果になった．Humpback の Top-1 精度は平均で 93.0%，最高で 98.3%（Debian，Docker 構文）という非常に高い数値を達成している．また，3.4 節でも述べた通り，予測単語は瞬時に提示される．正確性と瞬時性を有する Humpback は，開発者の生産性向上に大きく寄与できると考える．

4.4.2 導入手法による精度向上

実験結果をさらに詳細に分析するため，導入した手法ごとにその効果に着目する．まず，データクレンジングを行っていない Nothing モデルに焦点を当てると，他のモデルと比較して精度が非常に低い．Top-1 精度の最低値は 8.7%（Alpine，Docker 構文）であり，これは約 10 回の予測で 1 回しか正解単語を予測できていないことになる．この結果から，データクレンジングは精度向上に大きく寄与することが確認できる．Dockerfile の内容は `RUN` 命令が大半を占めており，データクレンジングを行う場合は 3.3 節の具体化で述べた`&&`の置換処理が頻繁に実行される．その結果，データクレンジングを行った学習用データはより具体性のあるものとなり，効果的な学習が出来たものと考えられる．

次にモデルスイッチングの効果を確認するため，Normal モデルと Humpback を比較すると，Humpback では Top-1 精度が最大で 5.9%改善されている（Ubuntu，Docker 構文）．また，Shell 構文では全体的に精度が引き上げられており，平均で 1.9%改善されている．モデルスイッチングを活用している Humpback では，Linux ディストリビューションの違いからなるコマンドの差異を予測に反映できる．従って Shell 構文の予測においては，どのディストリビューションに対してもモデルスイッチングの効果が発揮され，精度向上に繋がったと考えられる．以上の結果から，データクレンジングとモデルスイッチングのいずれも，Dockerfile に特化したコード補完システムを構築する上で有用であると示された．

4.4.3 ディストリビューションによる違い

Humpback の精度に注目すると，いずれの構文においても Debian における精度が最も高い．表 2 に示した通り，Debian の Dockerfile 数は 17,011 個で全体の約 80%を占める．学習用データ数が多ければそれだけ効果的な学習が行えるため，Debian における精度が最も高くなったのは学習用データ数が影響しているものと考えられる．

Alpine と Ubuntu を比較すると，Alpine は構文によって精度にそれほど差がない一方，Ubuntu は Docker 構文の精度が 95.8%と非常に高い．しかし両ディストリビューションにおけるデータ数はほぼ同じである．この点に関しては，Ubuntu では Docker 構文における記述に似通ったものが多く，学習がより効率的に行われた結果，データ数がほぼ同じ Alpine よりも高い精度を獲得したと考えられる．現段階では実際にどのような記述が多いのかは確認できていないため，今後の課題としたい．

5. おわりに

本研究では，言語モデルに基づく Dockerfile に特化したコード補完システムを提案し，新たなコード補完システム Humpback を実装した．Humpback はオンラインエディタとして実装されており，ブラウザからすぐに使用できる．また，予測単語は瞬時に提示されるため，開発者は開発速度を落とさず快適に

表 3: 実験結果の平均値

Linux	Docker 構文									Shell 構文								
	Top-1 精度			Top-5 精度			MRR			Top-1 精度			Top-5 精度			MRR		
	NTG	NML	HB	NTG	NML	HB	NTG	NML	HB	NTG	NML	HB	NTG	NML	HB	NTG	NML	HB
Alpine	8.7%	90.1%	**90.7%**	13.8%	93.0%	**93.1%**	0.113	0.914	**0.918**	12.4%	89.2%	**91.6%**	23.5%	**94.2%**	94.1%	0.176	0.914	**0.928**
Debian	54.9%	96.1%	**98.3%**	60.1%	96.7%	**98.9%**	0.576	0.965	**0.986**	46.6%	89.6%	**91.7%**	64.7%	92.7%	**93.9%**	0.548	0.911	**0.927**
Ubuntu	15.8%	89.9%	**95.8%**	20.4%	98.9%	**99.5%**	0.185	0.943	**0.976**	19.9%	88.9%	**90.1%**	36.8%	91.3%	**91.8%**	0.261	0.900	**0.909**

Humpback を活用できる．Dockerfile に特化した問題を解決し Humpback の精度を向上させるため，データクレンジングとモデルスイッチングという手法を導入した．評価実験によって，Humpback は平均 93.0%という高い Top-1 精度を持ち，そして我々の導入した手法が精度向上に貢献していることを確認した．

今後の課題として以下が挙げられる．

改良手法の更なる考案

現状では Humpback の実装にデータクレンジングとモデルスイッチングを導入しており，これらの手法が Humpback の精度向上に寄与することは 4.4 節で述べた．しかし Dockerfile に固有の課題をさらに考察し，本研究で導入した手法以外にどのような手法を導入すれば精度向上に繋がるか調査する．

データセットの改良

本研究における収集済みデータセットには，21,190 個の Dockerfile が含まれている．4.4.3 項に示した通り，現状では Dockerfile 数の最も多い Debian における予測が最も高い推薦精度を持つ．そこで Dockerfile を追加で収集し，より豊富なデータを用いた学習によって，推薦精度が向上するかを確認したいと考える．また，現状では学習用データとテスト用データを区別せずに使用している．しかし，推薦精度をより正確に算出するため，これらを区別したデータセットで学習および評価を行うことは重要な課題である．

他のコード補完システムとの比較

Humpback は Dockerfile に特化したコード補完システムとして実装された．しかし 4.3 節で述べた通り，評価実験では我々の用意したモデルを対象に比較を行っており，他のコード補完システムとの比較はなされていない．Humpback と既存研究による他のコード補完システムを比較し，推薦精度や推薦速度の点から相対的な評価を行うことは，コード補完システムの性能を確かめる上で重要であると考える．

謝辞 本研究の一部は，日本学術振興会科学研究費補助金基盤研究（B）（課題番号：18H03222）の助成を得て行われた．

文　献

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

[2] Docker, “Docker overview”. https://docs.docker.com/get-started/overview/

[3] Portworx, “Annual container adoption report,” 2019. https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf

[4] J. Cito, G. Schermann, J.E. Wittern, P. Leitner, S. Zumberi, and H.C. Gall, “An empirical analysis of the docker container ecosystem on github,” International Working Conference on Mining Software Repositories, pp.323–333, 2017.

[5] Y. Jiang and B. Adams, “Co-evolution of infrastructure and source code - an empirical study,” International Working Conference on Mining Software Repositories, pp.45–55, 2015.

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

[7] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, “A systematic mapping study of infrastructure as code research,” Information and Software Technology, vol.108, pp.65–77, 2019.

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

[9] G.C. Murphy, M. Kersten, and L. Findlater, “How are java software developers using the eclipse ide?,” IEEE Software, vol.23, no.4, pp.76–83, 2006.

[10] F.A. Gers, J. Schmidhuber, and F. Cummins, “Learning to forget: Continual prediction with lstm,” IEEE Conference Publication, vol.2, pp.850–855, 1999.

[11] A. Schankin, A. Berger, D.V. Holt, J.C. Hofmeister, T. Riedel, and M. Beigl, “Descriptive compound identifier names improve source code comprehension,” International Conference on Software Engineering, pp.31–40, 2018.

[12] I. Kuraj and R. Piskac, “Complete completion using types and weights,” ACM SIGPLAN Notices, vol.48, pp.27–38, 2013.

[13] V. Raychev, M. Vechev, and E. Yahav, “Code completion with statistical language models,” ACM SIGPLAN Notices, vol.49, no.6, pp.419–428, 2014.

[14] S. Proksch, J. Lerch, and M. Mezini, “Intelligent code completion with bayesian networks,” ACM Transactions on Software Engineering and Methodology, vol.25, no.1, pp.1–31, 2015.

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

[16] Y. Zhang, B. Vasilescu, H. Wang, and V. Filkov, “One size does not fit all: An empirical study of containerized continuous deployment workflows,” ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp.295–306, 2018.

[17] T. Sharma, M. Fragkoulis, and D. Spinellis, “Does your configuration code smell?,” International Working Conference on Mining Software Repositories, pp.189–200, 2016.

[18] C. Parnin, E. Helms, C. Atlee, H. Boughton, M. Ghattas, A. Glover, J. Holman, J. Micco, B. Murphy, T. Savor, M. Stumm, S. Whitaker, and L. Williams, “The top 10 adages in continuous deployment,” IEEE Software, vol.34, no.3, pp.86–95, 2017.

[19] S. Wu, “A review on coarse warranty data and analysis,” Reliability Engineering and System Safety, vol.114, no.1, pp.1–11, jun 2013.

[20] Docker, “Best practices for writing dockerfiles”. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

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