Expert Systems With Applications 202 (2022) 117267

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

DeepLib: Machine translation techniques to recommend upgrades for third-party libraries

Phuong T. Nguyen, Juri Di Rocco, Riccardo Rubei, Claudio Di Sipio, Davide Di Ruscio *

Department of Information Engineering, Computer Science and Mathematics, Università degli studi dell'Aquila, 67100 L'Aquila, Italy

ARTICLE INFO

Keywords:
Mining software repositories
Deep learning
Encoder–decoder neural network
Third-party libraries upgrade

ABSTRACT

To keep their code up-to-date with the newest functionalities as well as bug fixes offered by third-party libraries, developers often need to replace an old version of third-party libraries (TPLs) with a newer one. However, choosing a suitable version for a library to be upgraded is complex and susceptible to error. So far, Dependabot is the only tool that supports library upgrades; however, it targets only security fixes and singularly analyzes libraries without considering the whole set of related libraries. In this work, we propose DeepLib as a practical approach to learn upgrades for third-party libraries that have been performed by similar clients. Such upgrades are considered safe, i.e., they do not trigger any conflict, since, in the training clients, the libraries already co-exist without causing any compatibility or dependency issues. In this way, the upgrades provided by DeepLib allow developers to maintain a harmonious relationship with other libraries. By mining the development history of projects, we build migration matrices to train deep neural networks. Once being trained, the networks are then used to forecast the subsequent versions of the related libraries, exploiting the well-founded background related to the machine translation domain. As input, DeepLib accepts a set of library versions and returns a set of future versions to which developers should upgrade the libraries. The framework has been evaluated on two real-world datasets curated from the Maven Central Repository. The results show promising outcomes: DeepLib can recommend the next version for a library as well as a set of libraries under investigation. At its best performance, DeepLib gains a perfect match for several libraries, earning an accuracy of 1.0.

1. Introduction

While working with a software project, developers often use third-party libraries (TPLs) that offer tailored functionalities (He, He et al., 2021; Nguyen, Di Rocco, Di Ruscio, Di Penta, 2019; Raemaekers et al., 2017; Visser et al., 2012). Existing TPLs enable developers to exploit ready-to-use programming utilities without needing to reinvent everything from scratch. However, TPLs are subject to change (He, He et al., 2021), and to keep up with the new functionalities, in a software project, developers need to replace an old library with a more updated one. In fact, adopting the wrong version of a library might cause unavoidable disruptions (Derr et al., 2017) due to conflicts or breaking changes in the hosting code (Raemaekers et al., 2017). In this respect, the migration of TPLs is complex and susceptible to errors that need to be thoroughly managed to avoid any negative impacts on the entire project (He, He et al., 2021).

Being afraid of incompatibility and breaking changes (Huang et al., 2019), developers tend to procrastinate the upgrade of TPLs (Derr et al., 2017; Kula et al., 2018). They prefer to stay in the *comfort zone*, keeping the most stable versions and continuously ignoring the accumulative *maintenance debt* (Visser et al., 2012). A recent empirical study (Wang et al., 2020) shows that more than 50% of the considered projects never update more than half of their libraries. However, delaying the updates of used libraries, due to such difficulties and the related ripple effects, can harm software systems from different points of view, including security. Moreover, it can increase the accumulated technical debt, which is a measure reflecting *"the implied cost of additional rework, caused by deciding for an easy solution now instead of deciding for a better choice that would take longer to be implemented"* (Li et al., 2015).

In recent years, various bots have been conceived to support software development (DevBots) (Erlenhov et al., 2019). Dependabot is among the most recent approaches, and it aims to address the problem

The code (and data) in this article has been certified as Reproducible by Code Ocean: (https://codeocean.com/). More information on the Reproducibility Badge Initiative is available at https://www.elsevier.com/physical-sciences-and-engineering/computer-science/journals.

* Corresponding author.
E-mail addresses: phuong.nguyen@univaq.it (P.T. Nguyen), juri.dirocco@univaq.it (J. Di Rocco), riccardo.rubei@graduate.univaq.it (R. Rubei), claudio.disipio@graduate.univaq.it (C. Di Sipio), davide.diruscio@univaq.it (D. Di Ruscio).

https://doi.org/10.1016/j.eswa.2022.117267
Received 22 October 2021; Received in revised form 15 April 2022; Accepted 15 April 2022
Available online 20 April 2022

of upgrading TPLs. In particular, Dependabot is integrated into GitHub, and it continuously scans a project's dependencies. When vulnerable TPLs are discovered, upgrades are recommended. The main limitation of approaches like Dependabot is that they target specific kinds of upgrades (e.g., security fixes) and libraries are singularly analyzed and not considered as a whole by limiting the accuracy of the resulting recommendations.

In the context of open-source software, developing new systems by reusing existing components raises relevant challenges in *(i)* searching for relevant modules; and *(ii)* adapting the selected components to meet some pre-defined requirements. To this end, recommender systems in software engineering have been developed to support developers in their daily tasks (Di Rocco et al., 0000; Robillard et al., 2014). Such systems have gained traction in recent years as they can provide developers with a wide range of valuable items, including code snippets (Nguyen, Di Rocco, Di Ruscio, Ochoa et al., 2019; Nguyen, Di Rocco, Di Sipio et al., 2021), tags/topics (Di Rocco et al., 2020; Di Sipio et al., 2020), third-party libraries (Nguyen, Di Rocco, Di Ruscio, Di Penta, 2019), documentation (Rubei et al., 2020), to mention but a few. Through the CROSSMINER project (Di Rocco et al., 0000), we conceptualized various techniques and tools for extracting knowledge from existing open source components to provide tailored recommendations to developers, helping them complete their current development tasks.

The proliferation of disruptive deep neural networks in recent years has culminated in Deep Learning (Goodfellow et al., 2016), which proves to achieve profound achievement across several application domains (LeCun et al., 2015). In this work, we propose DeepLib, a novel approach to the recommendation of library upgrades, exploiting cutting-edge deep learning techniques. In particular, DeepLib has been built on top of long short-term memory and encoder–decoder neural networks to predict future versions of libraries. Such techniques have been successfully applied in various domains, including machine translation (Cho et al., 2014). By analyzing the migration history of mined projects, we build matrices containing libraries and their versions in chronological order, which are fed to the recommendation engine. As output, DeepLib delivers *(i)* the next version for a single library *lib* that the developer would like to upgrade for the project at hand; and *(ii)* the next version for a set of libraries that should also be upgraded due to the recommended upgrade for library *lib*. For the former, we built a long short-term memory recurrent network (Hochreiter & Schmidhuber, 1997) (LSTM), while for the latter, we developed an Encoder–Decoder LSTM (Sutskever et al., 2014) to predict a set of versions.

To the best of our knowledge, there exist no comparable tools recommending upgrades related to library versions. Thus, we cannot compare our system with any reusable baselines but evaluate it through extensive experiments on two considerably large datasets from the Maven Central Repository. This aims at studying the system's capability in real-world scenarios. The experimental results show that once being fed with decent training data, DeepLib can effectively suggest the next version for a single library and a set of libraries, demonstrating its feasibility in the field.

This is an extended version of our preliminary work (Nguyen, Di Rocco, Rube et al., 2021), where we presented a tool for predicting the next version of a library in a software project. In this work, we improve the original approach by adding an Encoder–Decoder neural network to recommend upgrades for a set of libraries instead of only one library as in the workshop paper. Furthermore, we conducted an empirical study on an additional dataset to investigate the generalizability of DeepLib. Last but not least, we extended the related work section to cover additional related techniques that the workshop paper did not consider.

In this respect, our work has the following contributions:

- A novel approach named DeepLib to the recommendation of updates for TPLs.
- An empirical study on the proposed approach exploiting real migration history data collected from Maven.
- The DeepLib tool,[1] together with the curated datasets, has been made available online to facilitate future research (Di Rocco et al., 2022).

We structure our paper into the following sections. Section 2 presents a motivating example to the research problem as well as the background to recurrent and Encoder–Decoder neural networks. Next, we introduce the proposed approach in Section 3, and present the evaluation materials and methods in Section 4. Afterward, Section 5 reports and analyzes the experimental results obtained through the evaluation. Some discussions and the probable threats to the validity of the findings are provided in Section 6. The related work is reviewed in Section 7, and the paper is finally concluded in Section 8.

2. Motivations and background

To facilitate the presentation, hereafter, we consider the following terms:

- *library* or *dependency*: A software module which is developed by a third party, and provides tailored functionalities. A library evolves over the course of time by offering new functionalities or bug fixes (Bauer et al., 2012; Derr et al., 2017);
- *repository* or *client*: A software project that is hosted in OSS platforms, e.g., GitHub, Maven and that makes use of some third-party libraries;

To replace the constituent third-party libraries, the developer can either *(i)* migrate an existing library to another library with similar functionalities; or *(ii)* upgrade a library from an old version to a newer one. While the former has been intensively investigated (He, He et al., 2021), the latter remains largely unexplored. Thus, in the scope of this work, we focus on upgrading libraries that are used by the software project at hand.

We first describe a motivating example in Section 2.1, and then make an overview of Dependabot, which is highly related to the problem addressed in this paper (Section 2.2). Afterwards, we briefly recall background related to long short-term memory recurrent neural networks (Section 2.3) and sequence-to-sequence learning (Section 2.4).

2.1. Motivating example

During the development cycle, with respect to library usage, a repository is normally updated by adopting a new version of libraries, even adding new (or removing deprecated) libraries. To better motivate our work, we consider in Table 1 a running example with maintainers working on the repository named *org.apache.hadoop:hadoop-auth*[2] which depends on a set of the following four libraries:

- lib_1: *log4j:log4j*;
- lib_2: *org.slf4j:slf4j-log4j12*;
- lib_3: *org.apache.httpcomponents:httpclient*;
- lib_4: *commons-codec:commons-codec*.

In Table 1, the latest version of the *hadoop-auth* repository is *3.0.0-alpha3* (the green row), and let us assume that the maintainers would like to upgrade the used libraries. However, they do not know for sure which version should be used for the constituent libraries, i.e., all the cells are filled with a question mark (**?**). One may think of a simple heuristic that migrates a library to the next version, or the latest one. However, by carefully investigating the table, we can see that such a heuristic does not work in every case. In particular, there are two additional possible changes that developers can perform on library

[1] https://github.com/MDEGroup/DeepLib/.
[2] https://bit.ly/2WP3ysS.

Table 1
Migration path of the *org.apache.hadoop:hadoop-auth* repository.

Client version	lib_1	lib_2	lib_3	lib_4	Timestamp
2.0.2-alpha	1.2.17	1.6.1	0	1.4	2012-10-02T00:44:04
2.3.0	1.2.17	1.7.5	4.2.5	1.4	2014-02-11T13:55:58
2.4.1	0	1.7.5	4.2.5	1.4	2014-06-21T06:08:34
2.5.1	0	0	4.2.5	0	2014-09-05T23:05:15
2.6.0	1.2.17	0	4.3.1	0	2014-11-13T22:35:37
2.7.2	1.2.17	1.7.10	4.2.5	1.4	2016-01-14T21:32:14
3.0.0-alpha3	1.2.17	1.7.10	4.5.2	1.4	2017-05-26T20:39:35
*	?	?	?	?	

Table 2
Libraries with clients performing backward upgradings.

Library	C_1	C_2	C_3
org.slf4j:slf4j-api	5717	1.7.25	2034
com.fasterxml.jackson.core:jackson-databind	2939	2.9.5	521
com.google.guava	2802	21	541
org.apache.commons:commons-lang3	2485	3:3.6	861
org.scala-lang:scala-library	2153	2.11.12	321
org.slf4j:slf4j-log4j12	1728	1.7.12	220
commons-io:commons-io	1599	2.6	877
org.apache.httpcomponents:httpclient	1151	4.5.5	426
commons-codec:commons-codec	946	1.11	640
ch.qos.logback:logback-classic	945	1.2.3	185
log4j:log4j	910	1.2.17	667
joda-time:joda-time	873	2.9.9	469
junit:junit	580	4.12	305
commons-logging:commons-logging	499	1.2	310
org.clojure:clojure	349	1.3.0	244
commons-lang:commons-lang	322	2.6	230
com.google.code.gson	303	2.8.2	51
com.google.code.findbugs:jsr305	290	3.0.2	125
org.springframework:spring-test	289	3.2.17	25
org.projectlombok:lombok	261	1.16.20	86
org.mockito:mockito-core	99	2.15.0	19
javax.servlet:javax.servlet-api	94	3.1.0	63
org.assertj:assertj-core	80	3.9.1	23
org.testng:testng	53	6.9.10	9
org.scalatest:scalatest	42	3.0.4	7
javax.servlet:servlet-api	30	2.5	17

dependencies: *(i)* removal of a library; and *(ii)* downgrade migration, as we explain as follows.

▷ **Removal of a library.** In the table, the versions of the repository are listed in chronological order, i.e., using their timestamp, that means when moving down the table, from the top to the bottom, we encounter newer versions of the repository. A cell with 0 implies that the library in the column is not included by the repository version represented in the row. It is worth noting that the presence of a library is subject to change from version to version. For instance, lib_1 has been used by version *2.0.0-alpha*, *2.0.2-alpha*, and *2.3.0*. When the repository is upgraded from *2.3.0* to *2.4.1*, lib_1 is removed. However, the library is then re-introduced when moving from *2.5.1* to *2.6.0*. In this respect, we see that the ability to recommend a 0 is also useful.

▷ **Downgrade upgradings.** We can see that the upgrading is not always done upward, i.e., moving the library to a higher version, since there are also backward migrations. For instance, when the client moves from *2.6.0* to *2.7.2*, lib_3 is downgraded from version *4.3.1* to *4.2.5*. However, the library is then updated to version *4.5.2* by client *3.0.0-alpha3*. This motivates us to perform an investigation on more libraries to see if downgrade migrations are just a special case, or they are commonplace. We crawled the migration history of 26 libraries from the Maven Central Dependency Graph (Benelallam et al., 2018). For each library, the migration history of all of its clients was analyzed. Afterwards, we counted the number of clients that, at certain point in their history updates, migrate to an older version of the library under investigation. Table 2 shows the result of our study where we report: $\mathbf{C}_1$: The number of clients that migrate downward (sorted in descending order); $\mathbf{C}_2$: The most downgrading version; and $\mathbf{C}_3$: The number of clients that migrated downward with the version in Column $\mathbf{C}_2$.

Among the mined libraries, *org.slf4j:slf4j-api* is the library with the largest number of clients with downward migrations. In particular, 5717 clients contain libraries being upgraded back to an older version. For the *org.slf4j:slf4j-api* library, *1.7.25* is the version with most downgrading upgrades, i.e., 2034 clients. Considering the other libraries in Table 2 as a whole, it is evident that backward migration is considerably popular. In other words, simply migrating a library to the next version, or the latest one is not always a solution. In practice, we need to perform both backward and downward upgrading.

In fact, to select the right version of each library, developers need to read the documentations very carefully and be informed of the internal changes. They also seek help in Q&A forums like Stack Overflow for probable solutions. Take as an example, with respect to the *org.apache.httpcomponents:httpclient* library (see Table 1), a developer creates a post[3] in Stack Overflow to ask for support concerning the upgrading of the current version *4.2.5*. Unfortunately, there is no proper solution to the raised issue.

According to an empirical work (Kula et al., 2018), systems are less likely to upgrade their library dependencies, with 81.5% of systems remaining with a popular older versions. In fact, developers cite upgrading as a practice that requires extra effort and added responsibility. Therefore, an automatic mechanism to recommend the future version of a library, or even for the whole set of libraries, is highly desirable, aiming to reduce the burden related to library upgrade.

Motivation 1. The upgrade of a library version is a complex task and it cannot be done just by moving to the next, or the latest version of the library.

2.2. GitHub *Dependabot*

This section gives an overview of the GitHub Dependabot approach (Dependabot, 0000), which aims at addressing the problem of upgrading TPLs used by existing GitHub repositories. In particular, Dependabot focuses on automating security updates of vulnerable dependencies. On a regular basis, Dependabot checks if new dependency versions are available. If yes, it informs developers and directly sends pull requests to the repository under investigation to update the corresponding dependency manifest with the new versions.

For instance, Fig. 1 shows Dependabot in action. It suggests possible upgrades to solve some vulnerabilities of the used *lucene-core*, *log4j*, and *junit* libraries. However, it is important to remark that GitHub Dependabot tends to recommend the closest non-vulnerable version for each dependency used in the project. Thus, libraries are singularly analyzed and recommendations are given without considering the set of used dependencies as a whole. Therefore, the cases represented with the 0 values in Table 1 cannot be explicitly managed. It is worth noting that the upgrades suggested by Dependabot are mainly based on vulnerability databases, e.g., the WhiteSource Vulnerability Database,[4] without taking into account ripple effects that might occur due to the co-existence of some additional dependencies in the given project.

Motivation 2. GitHub Dependabot is an initial attempt to recommend library upgrades. Nevertheless, it mainly analyzes libraries singularly and provides recommendations without considering the set of used dependencies as a whole. Thus, it deals with limited upgrading scenarios.

Altogether, we conclude that though there exists a tool to recommend upgrading of libraries, there is still a need for an automatic mechanism to support developers with migration, i.e., proper machinery to automatically choose a suitable version for their libraries

[3] https://bit.ly/3A8RN2s.
[4] https://www.whitesourcesoftware.com/vulnerability-database/.

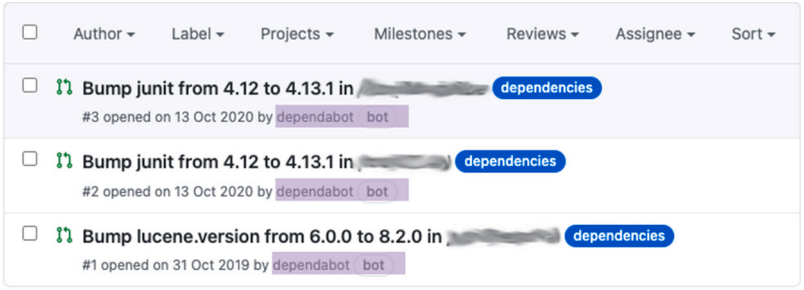

Fig. 1. GitHub Dependabot pull requests (names are blurred due to privacy).

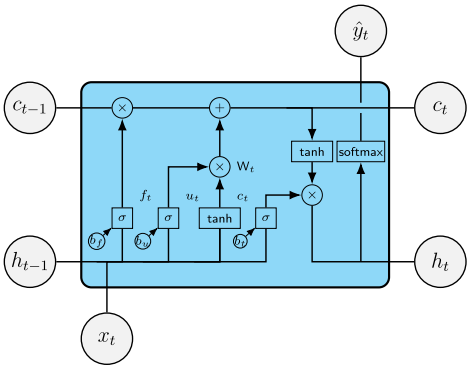

Fig. 2. An LSTM cell.
Source: Reproduced Olah (2020).

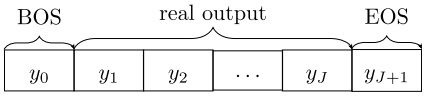

Fig. 3. Output sequence.

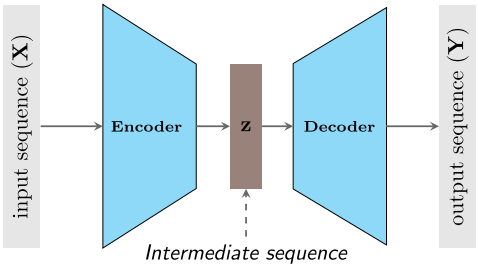

Fig. 4. The Encoder–Decoder architecture.
Source: Reproduced Cho et al. (2014).

considered as a whole. In this paper, we present an approach being able to leverage the knowledge of already updated projects to assist the developer in choosing a suitable version for the used libraries. In the next subsections, we review two neural networks dealing with time series data as a base for further presentations.

2.3. Long short-term memory neural networks

Long short-term memory recurrent neural networks (LSTMs) have been developed to work with time-series and sequence data (Hochreiter & Schmidhuber, 1997). LSTMs can remove or add information thanks to their internal design, thereby retaining worthy/valuable information and forgetting useless information. Fig. 2 depicts an LSTM cell, whose main modules are explained as follows.

In the figure, c_t and h_t are cell state and hidden state, respectively, which are propagated to the next cell. Given a cell, the output of the previous unit and the current input are fed as the input data. Considering $i_t = [h_{t-1}, x_t]$ as the concatenation of h_{t-1} (the hidden state vector from the previous time step) and x_t (the current input vector) then the following formulas are derived:

$$f_t = \sigma(W_f \cdot i_t + b_f) \tag{1}$$

$$u_t = \sigma(W_u \cdot i_t + b_u) \tag{2}$$

$$c_t = tanh(W_c \cdot i_t + b_c) \tag{3}$$

$$W_t = c_{t-1} \cdot f_t + N_t \cdot u_t \tag{4}$$

Where the *sigmoid* and *tanh* are defined as: $\sigma(x) = (1 + exp(-x))^{-1}$ and $\tanh(x) = 2 \cdot \sigma(2x) - 1$; $W_\times$ and $b_\times$ are the weight and bias matrices for different network entry, hidden state matrix. The sigmoid function is used to discard useless and retain useful information; Eqs. (1) and (2) are used to compute the forget and the update values, respectively. The same output h_t is ported as the hidden state to the next cell, and output of the current step (Shi et al., 2019). The size of c_t is the number of hidden units in the LSTM cell.

The Softmax function is used as the activation function, rendering a set of real numbers to probabilities which sum to 1.0 (Rawat & Wang, 2017). Given C classes, and y_k is the output of the kth neuron, the final prediction is the class that gets the maximum probability, i.e., $\hat{y} = argmax\ p_k$, $k \in \{1, 2, \ldots, C\}$, where p_k is computed as: $p_k = \frac{exp(y_k)}{\sum_{k=1}^{C} exp(y_k)}$.

In this work, we propose a practical solution to the prediction of the next version for a TPL by training an LSTM with data collected from several OSS projects.

2.4. Sequence-to-sequence learning

Encoder–Decoder LSTMs (Sutskever et al., 2014) are used to deal with sequence-to-sequence (seq2seq) prediction problems such as text summarization. An Encoder–Decoder LSTM transforms an input sequence $X = (x_1, x_2, \ldots, x_I)$ into an output sequence $Y = (y_1, y_2, \ldots, y_J)$.

Fig. 3 depicts an output sequence, where y_0 and y_{J+1} represent the beginning (BOS) and the end of sequence (EOS), respectively. In this respect, generating Y when X is given as input boils down to computing the conditional probability $P_\theta(Y|X)$ below:

$$P_\theta(Y|X) = \prod_{j=1}^{J+1} P_\theta(y_j|Y_{<j}, X) \tag{5}$$

where $P_\theta(y_j|Y_{<j}, X)$ is the probability of generating the jth of the output y_j, given $Y_{<j}$ and X. A seq2seq process involves two phases: *(i)* generating a fixed size vector z from X, i.e., $z = f(X)$; and *(ii)* generating Y from z.

Correspondingly, as shown in Fig. 4 an Encoder–Decoder LSTM consists of the following components:

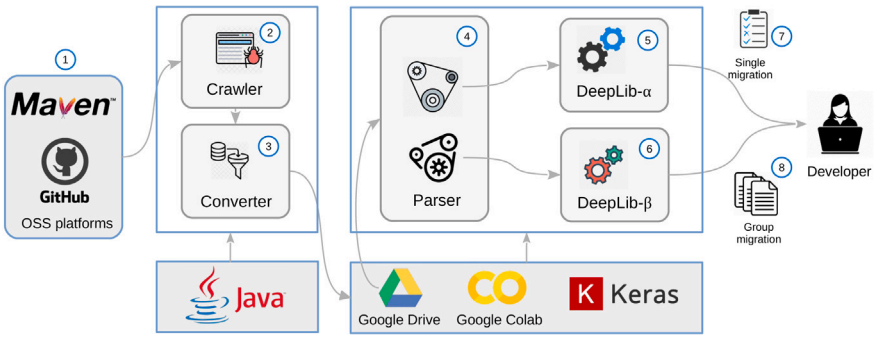

Fig. 5. System architecture.

- *Encoder*: This is a stack of LSTM cells and it accepts X as input to generate a fixed size vector z.
- *Intermediate sequence*: z is the resulting vector obtained by encoding information contained in X.
- *Decoder*: The component consists of LSTM cells to produce the output sequence using the information fed by the encoder, i.e., z.

Encoder–Decoder LSTMs have gained big succeed in various domains (Sutskever et al., 2014) as they are highly suitable to generate sequences from sequences. In this way, it is reasonable to use them to solve our issue, i.e., predicting a chain of library versions from past migrations. In the next section, we present in detail the conceived approach to recommendation of library updates.

3. Proposed approach

This section brings in our proposed approach to recommendation of library updates, exploiting migration history of mined OSS projects. The architecture conceived to realize the DeepLib framework is presented in Section 3.1. We provide two types of recommendations as follows. For suggesting the next version of a single library we build the first module named DeepLib-α (Nguyen, Di Rocco, Rube et al., 2021), an LSTM to accept as input a set of versions and returns the future version for each library (Section 3.2). Afterwards, in Section 3.3 we present the second module, i.e., DeepLib-β built on top of an Encoder–Decoder LSTM to recommend the next version for a set of libraries.

3.1. Architecture

In practical use, there are two levels of upgrade: *(i)* the *library level*; and *(ii)* the *source code level*. By the former, developers need to replace a library with a suitable version. In contrast, by the latter, they have to change the affected source code to make it conform with the new library versions and the related APIs. In the scope of this work, we deal with the first type of upgrade, i.e., recommending a suitable version for libraries. The upgrade at the source code level is left as our future work.

The conceived architecture is depicted in Fig. 5. DeepLib has been implemented on top of the Keras framework[5] and trained using Google Colab.[6] Data is fetched from OSS platforms ①, e.g., GitHub and Maven with Crawler ②. The collected data is then aligned, sorted, and transformed into a suitable format to store in CSV files by Converter ③. It is necessary to upload the data to Google Drive for further processing. The Parser component ④ builds migration matrices for DeepLib-α ⑤ and DeepLib-β ⑥ to provide updates for a single library ⑦ and multiple libraries ⑧. The succeeding subsections explain the two modules in detail.

3.2. DeepLib-α: Recommending the next version for a single library

DeepLib-α has been developed using an LSTM to recommend a version for a third-party library. We mine development history of software projects to build a migration matrix, whose rows represent clients and columns represents their versions. To populate such a matrix, we analyze each software client and fill the correct version for all libraries, one by one. Starting from a set of OSS projects, we parse each client to build a migration matrix for DeepLib-α as shown in Fig. 7(a). From the resulting matrix, we insert one more column on the right side. For each client, the last cell is filled with the version of the library by the next client.

To illustrate the transformation process, Fig. 7(a) depicts the migration matrix for lib_1 for the *org.apache.hadoop:hadoop-auth* repository in Table 1. On the left, there is the original matrix, whose the corresponding migration matrix is depicted on the right. For instance, the first row contains the versions of the four libraries, i.e., *(1.2.15, 1.6.1, 0, 1.4)* while the last column is the future version of lib_1, i.e., *1.2.17*, which is actually the version of lib_1 by the next client (Version *2.0.2*). This can be interpreted as follows:

> "*Given that in the current client we use version 1.2.15 for* lib_1*, 1.6.1 for* lib_2*, no* lib_3*, and version 1.4 for* lib_4*, then in the next version of the client we should adopt 1.2.17 for* lib_1*"*.

By repeating the same process, we can populate the migration matrices for other libraries. For the sake of clarity, only the matrix for lib_1 is shown in this section.

Since LSTMs only work with numbers, it is necessary to encode each library version using a unique number. Moreover, the σ and tanh functions (see Section 2.3) accept values in the [0..1] range, we also need to normalize all the numbers to meet this requirement. The right most part of Fig. 7(a) depicts the migration matrix after the encoding and normalizing phases.[7]

Fig. 6 explains how DeepLib-α works, with respect to the example in Fig. 7(a). The data to feed the system is a tuple of the form $x_t = \langle lib_1^{v1}, lib_2^{v1}, lib_3^{v1}, lib_4^{v1} \rangle$ and $y_t = \langle lib_1^{v2} \rangle$, which captures the migration path of a client. DeepLib-α uses input features from recent events, i.e., $X = \{x_t\}$, $t \in T^p$ to forecast the future version of each libraries, $Y = \{y_t\}$, $t \in T^f$, where T^p and T^f are time in the past and the future, respectively. By each time step t, only one vector x_t is fed to the LSTM cell. For illustration purposes, we consider only four time steps, i.e., t_0, t_1, t_2, and t_3 and the input data is given below.

- At t_0: $x_0 = (0.500, 0.333, 0.000, 1.000)$, $y_0 = (1.000)$.
- At t_1: $x_1 = (1.000, 0.333, 0.000, 1.000)$, $y_1 = (1.000)$.
- At t_2: $x_2 = (1.000, 1.000, 0.333, 1.000)$, $y_2 = (0.000)$.

[5] https://keras.io/.
[6] https://colab.research.google.com/.

[7] Matrix encoding and normalizing is conveniently done with the *LabelEncoder()* and *MinMaxScaler()* utilities embedded in Python.

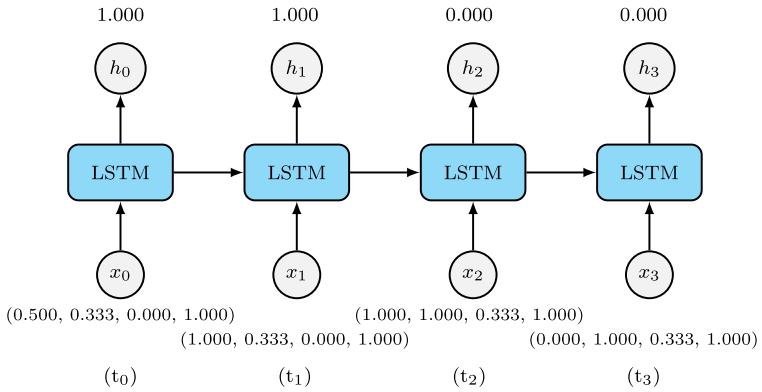

Fig. 6. Architecture of DeepLib-α.

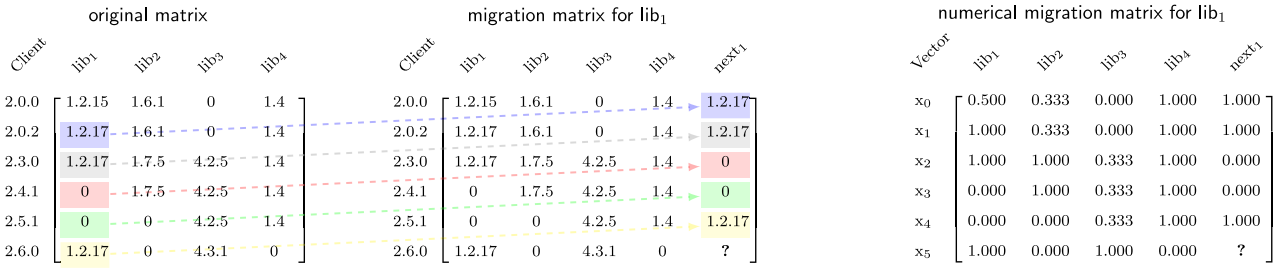

Fig. 7. Migration matrices and input data for DeepLib-α.

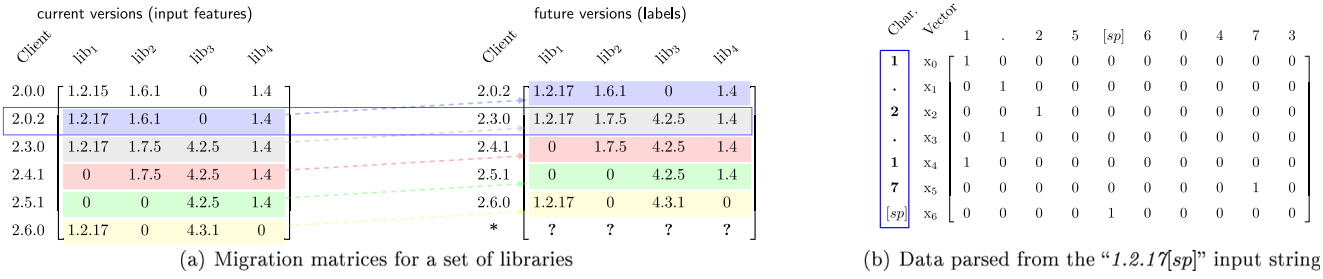

Fig. 8. Migration matrices and input data for DeepLib-β.

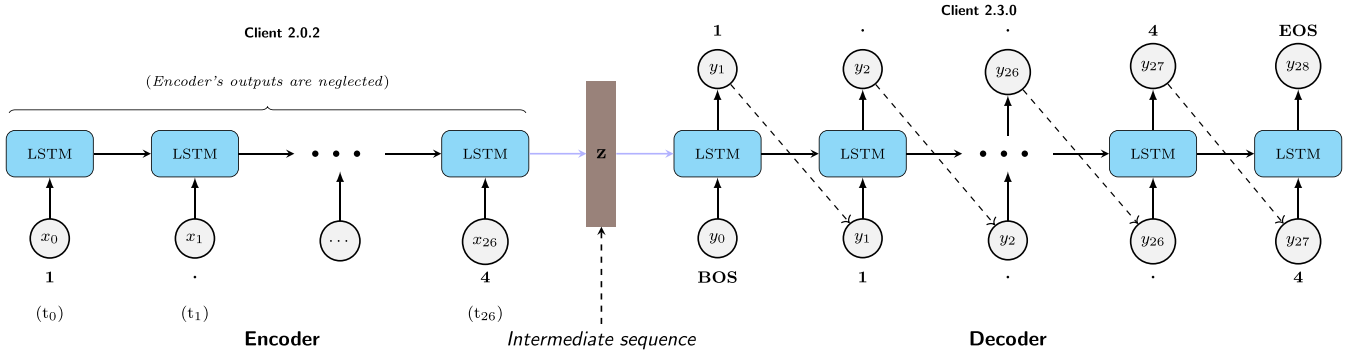

Fig. 9. DeepLib-β: Training with input sequence $(1.2.17, 1.6.1, 0, 1.4)$ and output sequence $(1.2.17, 1.7.5, 4.2.5, 1.4)$.

– At t_3: $x_3 = (0.000, 1.000, 0.333, 1.000)$, $y_3 = (0.000)$.

The same procedure can be done for other input entries to train DeepLib-α. Being based on the technique presented in Section 2.3, the tool uses the trained weights and biases to perform predictions for unknown input data.

3.3. DeepLib-β: Recommending the next version for the whole set of libraries

Though we can exploit DeepLib-α to recommend the next version for a set of libraries by computing the next version for each of them, one by one, in this section we introduce DeepLib-β to compute the next migration path for a set of libraries as a whole. This is helpful for developers who want to upgrade all libraries at once.

In Fig. 8(a), we illustrate how the migration matrix for the example in Table 1 is populated. On the left side, we depict the original matrix, each row corresponds to a version of the considered client (referred as client hereafter for simplicity). On the right-hand side, there is the resulting migration matrix, and each row is filled with all the library versions of the next client of the row in the left matrix. For instance, the future version of client *2.0.2* is *2.3.0*, i.e., marked with the blue frame, and this is expressed as:

> "*Given that in the current client we use 1.2.17 for lib_1, 1.6.1 for lib_2, no lib_3, and 1.4 for lib_4, then in the next version of the client we should adopt 1.2.17 for lib_1, 1.7.5 for lib_2, 4.2.5 for lib_3, and 1.4 for lib_4*".

The input and output sequences are represented as follows: X = "*1.2.17[sp]1.6.1[sp]0[sp]1.4*" and Y = "*[BOS]1.2.17[sp]1.7.5[sp]4.2.5 [sp]1.4[EOS]*", where $[sp]$ is a space, [BOS] and [EOS] are special characters to signal the beginning and end of an output sequence (see Fig. 3).

DeepLib-β is built based on the seq2seq learning model (Sutskever et al., 2014), converting a sequence into an output sequence of versions. To feed DeepLib-β, we transform the input data into vectors with entries in the [0..1] range. First, a corpus of all the characters used to form the library versions is curated. For instance, "*1.2.17*$[sp]$" contains the following characters: "*1*", ".", "*2*", "*7*", and "$[sp]$". Then each character is encoded using a one-hot vector whose length corresponds to the corpus's number of characters (Ω). For Fig. 8(b), we get a corpus consisting of $\Omega = 10$ characters, and the maximum sequence length $\Theta = 27$. In this way, a sequence is represented as a 2D matrix of size $(\Theta \times \Omega)$, each row corresponds to the one-hot vector of a character. A sequence with length smaller than Θ is padded with $[sp]$ to fill the gap. The same process is done in `Decoder` to form the output sequence. For the sake of presentation, we illustrate how the parsing is done for the "*1.2.17*$[sp]$" input sequence in Fig. 8(b). The first and second column represent the sequence and its vectors, respectively, each row corresponds to a one-hot vector. The remaining columns represent the characters.

Fig. 9 shows how DeepLib-β works with respect to the example in Fig. 8(a). `Encoder` consists of a stack of LSTM units to encode input sequences, and `Decoder` is also made of LSTM units and it decodes the input sequence. We feed a vector at every time step to `Encoder`. Similarly, by `Decoder`, we introduce the label vectors singularly, moreover, the output of a time step is fed as input to the next one. It has been shown that, training in the reverse order of the input sentence brings a better prediction performance (Sutskever et al., 2014). Therefore, in the scope of this work, every time there is an input sequence, we reverse it and feed as input to DeepLib-β. Following the paradigm in Fig. 4, DeepLib-β learns from training data to predict the future versions for an input sequence of versions.

4. Evaluation

We evaluate DeepLib to study its capability to provide a developer with accurate recommendations featuring suitable migration steps. After formulating the research questions in Section 4.1, we introduce the datasets, the experimental settings and the metrics in Sections 4.2 and 4.3, respectively.

4.1. Research questions

The evaluation was conducted to answer the following research questions:

- **RQ**$_1$: *How well can DeepLib-α recommend the next version for a single library?* We perform experiments to investigate to which extent DeepLib-α is able to recommend the next version for each single library. Such type of recommendation is desired when developers prefer to upgrade libraries one by one.

Table 3
Summary of the datasets.

	Library	Alias	η_V	η_C	η_M
Dataset $\mathbf{D}_1$	*junit:junit*	L_{01}	29	101,541	2073
	org.slf4j:slf4j-api	L_{02}	74	44,233	16,187
	org.scala-lang:scala-library	L_{03}	228	25,417	19,508
	com.google.guava:guava	L_{04}	90	24,532	8921
	org.mockito:mockito-core	L_{05}	259	20,762	855
	com.android.support:appcompat-v7	L_{06}	59	19,772	1194
	commons-io:commons-io	L_{07}	25	19,198	3332
	ch.qos.logback:logback-classic	L_{08}	75	18,655	3100
	org.commons:commons-lang3	L_{09}	18	17,224	3915
	org.clojure:clojure	L_{10}	67	15,954	234
Dataset $\mathbf{D}_2$	*com.fasterxml.jackson.core*	L_{11}	120	15,891	9022
	log4j:log4j	L_{12}	20	15,618	1865
	org.slf4j:slf4j-log4j12	L_{13}	74	13,890	3607
	org.scalatest:scalatest	L_{14}	18	13,216	10
	javax.servlet:javax.servlet-api	L_{15}	17	13,271	355
	com.google.code.gson:gson	L_{16}	35	13,300	1562
	javax.servlet:servlet-api	L_{17}	17	13,271	427
	commons-lang:commons-lang	L_{18}	15	10,845	1848
	org.apache.httpcomponents:httpclient	L_{19}	54	10,136	3236
	org.slf4j:slf4j-simple	L_{20}	72	9689	568
	org.springframework:spring-context	L_{21}	155	9591	8563
	org.assertj:assertj-core	L_{22}	44	9628	495
	commons-logging:commons-logging	L_{23}	18	9423	1639
	org.projectlombok:lombok	L_{24}	40	9742	1021
	commons-codec:commons-codec	L_{25}	15	9.064	1728
	org.testng:testng	L_{26}	78	8941	440
	com.google.code.findbugs:jsr305	L_{27}	12	8355	1804
	org.springframework:spring-test	L_{28}	107	7736	1236
	joda-time:joda-time	L_{29}	38	7565	3187

- **RQ**$_2$: *How well can DeepLib-β recommend the next version for a set of libraries?* Similar to **RQ**$_1$, we extend the evaluation and study if DeepLib-β can recommend the next version for a set of libraries. This is useful in practice, especially for developers who want to upgrade all libraries at once, instead of only one.
- **RQ**$_3$: *What contributes to an improvement in DeepLib's performance?* Our proposed tool is a data-driven approach, and its performance is heavily dependent on the input data. In this research question, we investigate *when* the system cannot obtain a high prediction accuracy, i.e., it fails, and especially *why*. This aims to find a practical way to avoid the common pitfalls that adversely affect its recommendation capability in the field.

4.2. Data extraction

The Maven Central Repository[8] consists of a huge number of artifacts,[9] and includes additional features such as statistical reports, the list of most popular libraries, the list of dependencies for each artifact. To evaluate DeepLib, we rely on two datasets, named $\mathbf{D}_1$ and $\mathbf{D}_2$, collected from more than 1000 public Maven repositories.

$\mathbf{D}_1$ and $\mathbf{D}_2$ consist of migration history for the *top ten* and the *next top 19* popular libraries, respectively. Given a set of libraries, we crawled all of their versions together with the list of clients and their corresponding release date. Moreover, we mined dependency links from a client to the used libraries with Maven Dependency Graph, a graph-based representation of the collected artifacts in Maven and their relationships. To generate a dependency graph from a set of libraries, we made use of an existing dataset (Benelallam et al., 2018) using MavenMiner (Benelallam et al., 2019).

Afterwards, we performed additional steps to remove unuseful clients by filtering the datasets with the following constraints: A client

[8] https://mvnrepository.com/.

[9] At the time of writing, there are more than 17 millions of artifacts in the Maven Central Repository.

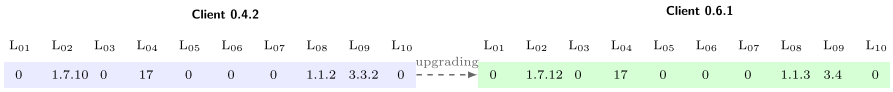

Fig. 10. Upgrading *com.hubspot:SingularityService* from 0.4.2 to 0.6.1.

should *(i)* have more than one version; *(ii)* migrate at least one library among the considered libraries; and *(iii)* use at least four of the given libraries. This allows us to keep the resulting matrices not too sparse.

Compared to the clients in $\mathbf{D}_2$, those in $\mathbf{D}_1$ contain more upgrades from one library version to another. The use of $\mathbf{D}_1$ and $\mathbf{D}_2$ allows us to investigate if DeepLib can work well under different situations of the input data, i.e., if it still provides suitable upgrades even when the data is sparse. Table 3 reports the main characteristics of the datasets: each row features an input library with its name, the number of versions (η_V), the number of clients that use at least one version of the library (η_C), the number of clients that migrate from one library version to another (η_M). We obtained 35,300 rows and 56,230 rows for $\mathbf{D}_1$ and $\mathbf{D}_2$, respectively.

4.3. Settings and metrics

▷ **Experimental settings.** The evaluation is done to study if our approach can recommend a future version for a library and a set of libraries. We opted for the ten-fold cross-validation technique (Kohavi, 1995), widely chosen to evaluate machine learning models. A dataset is split into $k = 10$ equal parts, so-called *folds*. One fold is used as testing data for each validation round, and the remaining $k-1$ folds are merged to create the training data. The testing fold represents projects that need upgrading recommendations, while the training folds correspond to the existing upgrades collected from real clients. In particular, by the D_1 dataset, there are 35,300 upgrades and thus each testing fold consists of $35,300/10 = 3530$ rows, while the training data is composed of $(35,300/10) \times 9 = 31{,}770$ rows (upgrades). By D_2, we have 56,230 upgrades, and each testing fold has $56,230/10 = 5,623$ rows, while the training data is composed of $(56,230/10) \times 9 = 50{,}607$ rows (upgrades).

The evaluation simulates a real development scheme where *the system needs to provide the active projects with recommendations using the data from a set of available training projects.* Within the training data, we also used 80% and 20% of the data for training and validation, respectively. While the training phase is used to teach the models, the validation part is done to calibrate their hyperparameters. For each repository, the libraries and their versions corresponding to the older client are fed as input data, while the libraries and their versions corresponding to the newer client are used as label.

We take as an example of a project used for training as follows. Fig. 10 shows the upgrading of the *com.hubspot:SingularityService*[10] repository from Version 0.4.2 to Version 0.6.1. The repository invokes four libraries, i.e., L_{02}, L_{04}, L_{08}, and L_{09}. On the left, we show the list of versions for the libraries of the older client numbered *0.4.2*. The remaining cells are filled with 0, indicating that the corresponding libraries are not present. On the right, there is the list of versions for the libraries of the newer client numbered *0.6.1*. Following the paradigm presented in Figs. 6 and 9, to train DeepLib, the left part is used as input data (query), and the right part is used as label. In practice, this means DeepLib is expected to provide recommendations consisting of versions as shown in the right part, given that it has been fed with the versions on the left part.

▷ **Ground-truth data.** For a testing client $TC_{current}$, all the library versions of its next client TC_{next} are saved as ground truth data. We use $TC_{current}$ to feed DeepLib, which returns a future version for each library of $TC_{current}$. We evaluate if the recommended versions match with the ground-truth data.

[10] https://bit.ly/2MnvnXn.

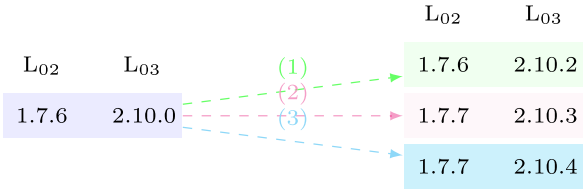

Fig. 11. Clients with different migration patterns.

By checking the datasets, we see that clients sharing the same set of library versions can be updated with various paths. For example, the following three projects: *(1): org.skinny-framework:skinny-common_2.10*,[11] *(2): org.scalikejdbc:scalikejdbc-interpolation_2.10*,[12] and *(3): org.scalikejdbc:scalikejdbc-config_2.10*[13] depends on L_{02} and L_{03} (see Table 3), and their migration is shown in Fig. 11.

All of the starting clients have the same library versions, i.e., L_{02}: *1.7.6* and L_{03}: *2.10.0*. However, by their next client, each of the projects has a different migration pattern. For instance, while L_{02} is kept as *1.7.6* for *(1)*, it is upgraded to *1.7.7* in *(2)* and *(3)*. Similarly, the starting version of L_{03} is *2.10.0* and it is updated to three different versions, i.e., *2.10.2*, *2.10.3*, and *2.10.4* by *(1)*, *(2)* and *(3)*, respectively.

A recent empirical study (Derr et al., 2017) showed that a large number of libraries can be upgraded by at least one closest version, without causing any code changes. This happens since though TPLs provide several APIs, developers normally make use a small fraction of them. That means for L_{02}, replacing *1.7.6* with *1.7.7*, or for L_{03} substituting *2.10.2* with *2.10.3* or *2.10.4*, does not trigger any incompatibilities. Altogether, in our evaluation, we consider multiple ground-truth paths for a client. With respect to the example in Fig. 11, all the migrations on the right are ground-truth data for the client on the left.

To validate the performance, we must cover all possible cases concerning the ground-truth data. In fact, by carefully checking the dataset, we see that projects with different updates account for a small fraction. In particular, only 2% and 4% of projects in $\mathbf{D}_1$ and $\mathbf{D}_2$, respectively, have multiple migrations, and the others have single migration. In this respect, the prediction of the next version for a library must be unique, so as to avoid overwhelming developers.

▷ **Metrics.** We evaluate how well DeepLib recommends versions that eventually match with those stored in the ground-truth data. With DeepLib-α, we compute accuracy according to each library (Acc_{lib}): the metric measures the ratio of clients with correct predictions (δ) to the total number of clients (n). While for DeepLib-β, we compute accuracy by each project (Acc_{pro}), i.e., the ratio of the number of correctly predicted versions (Δ) to the total number of libraries ($\mathbb{L}$) as follows.

$$Acc_{lib} = \frac{\delta}{n} \tag{6}$$

$$Acc_{pro} = \frac{\Delta}{\mathbb{L}} \tag{7}$$

Besides accuracy, we compute correlation efficients using the Spearman ρ and the Kendall τ, and measure the effect size with Cliff's delta (Grissom & Kim, 2005): the larger the effect size is, the stronger is the relationship between the samples.

[11] https://bit.ly/3b5RhZn.
[12] https://bit.ly/38fr3ln.
[13] https://bit.ly/38Xw9lk.

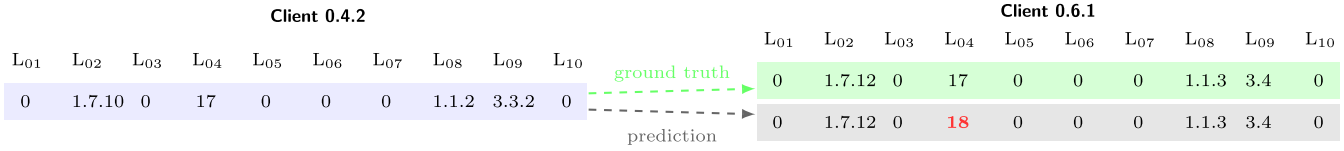

Fig. 12. Recommendation for the *com.hubspot:SingularityService* repository.

5. Results

We report a recommendation example provided by DeepLib in Section 5.1. Afterwards, the research questions are answered in Sections 5.2, 5.3, and 5.4.

5.1. Explanatory example

To illustrate how DeepLib recommends upgrades to third-party libraries, we show in Fig. 12 the recommendation results for the *com.hubspot:SingularityService*[14] that has been introduced in Fig. 10. The repository invokes four libraries, i.e., L_{02}, L_{04}, L_{08}, and L_{09}. The left side depicts the list of versions for the libraries of the older client *0.4.2*, and on the right, there is the list of versions for the libraries of the newer client *0.6.1*. The top row on the right depicts the real versions of all libraries for the next client *0.6.1* (the ground-truth data), and the bottom row represents the recommendations provided by DeepLib. Among the four libraries, three of them are upgraded to a new version. In particular, L_{02}: *1.7.10* → *1.7.12*, L_{08}: *1.1.2* → *1.1.3*, L_{09}: *3.3.2* → *3.4*.

The scenario is challenging as it requires a big upgrading step, i.e., changing almost all the constituent libraries at once. We select the example since it is a typical one in the evaluation. In particular, by carefully checking the given datasets, we realized that the majority of the clients perform big migrations. In this sense, we expect DeepLib to provide proper recommendations to assist developers in migrating their software clients, given that big migrations may make the prediction more challenging.

The second row of Fig. 12 presents the versions suggested by DeepLib for Client *0.6.1*. As it can be seen, the tool recommends correct upgrading for three libraries, namely L_{02}, L_{08}, and L_{09}. It only mispredicts for L_{04}, by providing *18* instead of *17*, the correct one. Moreover, DeepLib accurately predicts all the zeros, i.e., the libraries that are not invoked. This seems to be trivial at fight sight, however as we pointed out before (see Section 2.1), recommending a zero makes sense, also considering the fact that wrongly suggesting a version rather than 0, when a 0 is actually needed, may make developers confused.

Summary. DeepLib provides relevant recommendations to the *explanatory repository*, given that a big step is required, i.e., upgrading at the same time by almost all the constituent libraries.

5.2. RQ_1: How well can DeepLib-α recommend the next version for a single library?

We performed experiments on both datasets and the prediction results for $\mathbf{D}_1$ and $\mathbf{D}_2$ are shown in Table 4. For each library, besides the accuracy for each fold from **F01** to **F10**, we also average out the scores to get the final accuracy, which is shown in the last column of the tables. Moreover, the cells with an accuracy smaller than 0.700 are marked using the light red color, signaling an inferior performance.

For $\mathbf{D}_1$, there are ten libraries in total, and the results obtained by DeepLib-α for the dataset are shown in the upper part of Table 4. Overall, the table demonstrates that DeepLib-α can provide accurate predictions for almost all the libraries. For instance, with L_{01}, by all the testing rounds DeepLib always gets an accuracy larger than 0.90, and the average accuracy is 0.970. This also applies to other libraries, such as L_{05} or L_{07}. Especially, by L_{06} we see a maximum accuracy for most of the folds. It is our assumption that the quality of training data is the main contributing factor to the performance gain. We are going to validate this hypothesis in Section 5.4.

We analyze the results obtained by DeepLib-α on $\mathbf{D}_2$ in the lower part of Table 4. The table shows a similar outcome to that when running DeepLib-α on $\mathbf{D}_1$. By most of the libraries, DeepLib-α yields a good prediction performance, i.e., the accuracy is generally larger than 0.90. By L_{14}, DeepLib-α gets the maximum performance by nine among the ten folds.

However, besides the good predictions for most of the libraries of both datasets, it is evident that DeepLib-α suffers a setback by some of them. For instance, by L_{02}, DeepLib-α obtains a low accuracy for most of the folds: by only three among the ten folds, the tool gets an accuracy larger than 0.700, while by the remaining ones, it gains a lower accuracy. A mediocre performance is also seen by L_{11} and L_{21}, compared to the other libraries. This implies that DeepLib-α fails to retrieve accurate predictions for some libraries. We ascertain the cause in Section 5.4.

Answer to RQ_1. Being fed with proper training data, DeepLib is able to recommend the next version for a single library, obtaining a high prediction accuracy for the majority of the libraries.

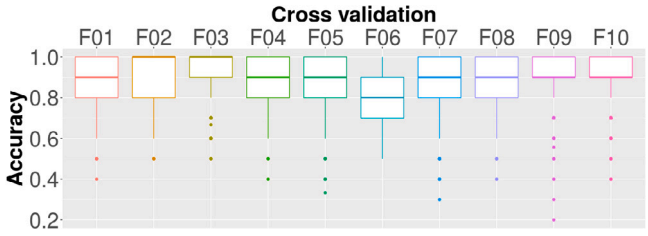

Fig. 13. RQ_2: $\mathbf{Acc}_{pro}$ obtained by DeepLib-β on $\mathbf{D}_1$.

5.3. RQ_2: How well can DeepLib-β recommend the next version for a set of libraries?

We report the final results for all the projects for each fold among **F01–F10** using boxplots. The accuracies obtained by DeepLib-β for both $\mathbf{D}_1$ and $\mathbf{D}_2$ are shown in Figs. 13 and 14, respectively. By examining the results, we encounter several cases similar to the one in Section 5.1, i.e., DeepLib-β recommends decent upgradings, also when big migration steps are required.

In Fig. 13, apart from some outliers, by most of the folds for $\mathbf{D}_1$, we get an accuracy larger than 0.80. By many projects, DeepLib-β earns an accuracy of 1.00, suggesting that the tool correctly predicts all the library versions for these projects.

Fig. 14 shows a similar outcome for $\mathbf{D}_2$, compared to $\mathbf{D}_1$. It is worth noting that by $\mathbf{D}_2$ there are 19 libraries, resulting in longer input and output sequences, and this should make the prediction more challenging. It is evident that for almost all the folds, DeepLib-β gets an accuracy close to 1.0, and this is demonstrated by the narrow boxplots converging to the upper bound of the diagram.

Nevertheless, similar to the results in $\mathbf{RQ}_1$, by both datasets we still witness projects with which DeepLib-β gets a low performance. As seen

[14] https://bit.ly/2MnvnXn.

Table 4
RQ$_1$: **Acc**$_{lib}$ obtained by DeepLib-α on $\mathbf{D}_1$ and $\mathbf{D}_2$.

	Library	Cross validation										
		F01	F02	F03	F04	F05	F06	F07	F08	F09	F10	Average
Dataset $\mathbf{D}_1$	L_{01}	0.975	0.956	0.988	0.991	0.954	0.954	0.962	0.965	0.986	0.970	0.970
	L_{02}	0.558	0.728	0.792	0.667	0.347	0.635	0.656	0.742	0.611	0.515	0.625
	L_{03}	0.748	0.824	0.741	0.856	0.908	0.944	0.902	0.670	0.552	0.669	0.781
	L_{04}	0.767	0.735	0.805	0.776	0.678	0.608	0.766	0.776	0.954	0.857	0.772
	L_{05}	0.961	0.989	0.988	0.972	0.976	0.623	0.952	0.994	0.966	0.974	0.939
	L_{06}	0.997	0.994	1.000	0.999	1.000	1.000	1.000	1.000	1.000	1.000	0.999
	L_{07}	0.952	0.981	0.970	0.969	0.945	0.947	0.952	0.960	0.990	0.961	0.963
	L_{08}	0.889	0.950	0.958	0.922	0.967	0.971	0.921	0.924	0.992	0.967	0.946
	L_{09}	0.937	0.952	0.967	0.966	0.960	0.852	0.966	0.900	0.984	0.952	0.944
	L_{10}	0.994	1.000	0.998	1.000	1.000	0.993	0.987	1.000	1.000	1.000	0.997
Dataset $\mathbf{D}_2$	L_{11}	0.340	0.227	0.524	0.683	0.801	0.109	0.648	0.701	0.641	0.766	0.544
	L_{12}	0.976	0.999	0.970	0.962	0.942	1.000	0.958	0.968	0.979	0.979	0.973
	L_{13}	0.964	0.987	0.845	0.828	0.815	1.000	0.825	0.798	0.989	0.818	0.887
	L_{14}	1.000	1.000	1.000	1.000	1.000	1.000	1.000	0.996	1.000	1.000	0.999
	L_{15}	0.996	0.996	0.989	0.993	0.990	1.000	0.976	0.989	0.976	0.991	0.990
	L_{16}	0.966	0.979	0.953	0.956	0.980	1.000	0.977	0.909	0.942	0.977	0.964
	L_{17}	0.998	0.998	0.993	0.995	0.979	1.000	0.986	0.996	0.989	0.995	0.993
	L_{18}	0.976	0.999	0.964	0.953	0.895	0.959	0.936	0.963	0.995	0.984	0.963
	L_{19}	0.878	0.986	0.810	0.848	0.793	0.306	0.804	0.778	0.809	0.757	0.777
	L_{20}	0.994	0.999	0.994	0.971	0.990	1.000	0.989	0.973	1.000	1.000	0.991
	L_{21}	0.822	0.930	0.706	0.822	0.577	0.214	0.580	0.483	0.243	0.365	0.574
	L_{22}	0.898	0.985	0.989	0.985	1.000	1.000	0.992	0.998	0.950	0.992	0.979
	L_{23}	0.976	1.000	0.983	0.963	0.906	0.959	0.969	0.955	0.964	0.951	0.962
	L_{24}	0.992	0.997	0.929	0.944	1.000	1.000	0.998	0.986	0.963	0.954	0.976
	L_{25}	0.959	0.990	0.952	0.942	0.902	0.978	0.936	0.956	0.997	0.978	0.959
	L_{26}	0.975	0.991	0.966	0.983	0.993	1.000	0.983	0.944	0.968	0.997	0.980
	L_{27}	0.866	0.995	0.969	0.959	0.988	1.000	0.974	0.911	1.000	0.999	0.966
	L_{28}	0.962	0.996	0.981	0.904	0.939	0.999	0.863	0.934	0.921	0.921	0.942
	L_{29}	0.902	0.966	0.854	0.917	0.863	0.924	0.848	0.863	0.949	0.947	0.903

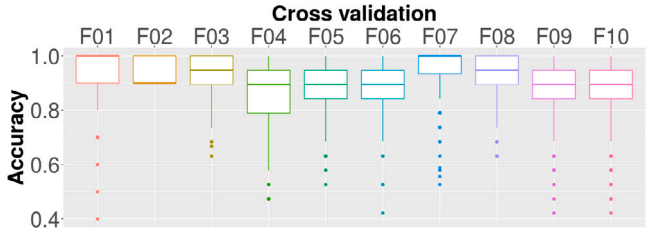

Fig. 14. RQ$_2$: **Acc**$_{pro}$ obtained by DeepLib-β on $\mathbf{D}_2$.

in the figures, some points lie further down the boxplots, corresponding to a mediocre performance. We attribute such a setback to the quality of the training data, and we find out the reason in the next research question.

Answer to RQ$_2$. On the given datasets, DeepLib-β successfully predicts the next version for libraries by most of the projects.

5.4. *RQ$_3$: What contributes to an improvement in DeepLib's performance?*

Through **RQ**$_1$ and **RQ**$_2$, we see that DeepLib obtains an encouraging result for most of the libraries as well as clients. Nevertheless, it *fails* in some certain cases. In Table 4, we encounter cells marked in red, corresponding to an accuracy lower than 0.700. Similarly, by Figs. 13 and 14, there are outliers residing near the minimum whiskers, also suggesting a low accuracy. It is necessary to find out the rationale behind such a setback, as this helps reveal the pitfalls that one can avoid when deploying DeepLib.

According to Table 3, there are three variables: number of versions (η_V), number of clients (η_C), and number of migrations (η_M). We perform quantitative analyses to study the relationships between these variables and the average accuracy (the last column of Table 4). We compute correlation efficients using the Spearman ρ and the Kendall τ, and measure the effect size with Cliff's delta (Grissom & Kim, 2005).

Table 5
RQ$_3$: Correlation coefficients and effect size, DeepLib-α.

Metric	Score	Acc vs. η_V	Acc vs. η_C	Acc vs. η_M
Spearman	ρ	-4.84×10^{-1}	-1.29×10^{-1}	-8.48×10^{-1}
	p-value	7.78×10^{-3}	5.04×10^{-1}	5.98×10^{-9}
Kendall	τ	-3.03×10^{-1}	-7.61×10^{-2}	-6.71×10^{-1}
	p-value	2.19×10^{-2}	5.61×10^{-1}	3.32×10^{-7}
Cliff's delta	–	1.0 (large)	1.0 (large)	1.0 (large)

Table 5 reports the results obtained for the outcomes produced by DeepLib-α.

As can be seen, there is a low correlation between accuracy and η_V, and this is enforced by both coefficients, i.e., $\rho = -4.84 \times 10^{-1}$ and $\tau = -3.03 \times 10^{-1}$. Moreover, the difference is statistically significant, i.e., p-value $= 7.78 \times 10^{-3}$ and 5.98×10^{-9}. The table also shows that the effect is large by the considered relationships, i.e., Cliff's delta is 1.0. This essentially means that the more versions a library has, the lower accuracy DeepLib-α obtains. In other words, having a large number of library versions negatively impacts on the prediction performance.

A similar trend is seen with the relationship between accuracy and η_M. In particular, $\rho = -8.48 \times 10^{-1}$ and $\tau = -6.71 \times 10^{-1}$, which means accuracy is disproportionate to the number of migrations. The difference is statistically significant and the effect size is large. Altogether, this suggests that it is more difficult for DeepLib to provide good recommendations for a library associated with a large number of migrations.

We cannot draw any concrete conclusions about the relationship between accuracy and η_C, as the p-value is larger than 0.05, although both ρ and τ are very small. This means by some libraries, having more clients is beneficial to predictions, while by some others, it is not. This is understandable since in principle having more training data is helpful (Yamashita et al., 2018), however, the prediction performance depends also on η_V and η_M, and as shown above, accuracy is greatly affected by these parameters.

We suppose that this happens due to the structure of the networks, i.e., the current weights are sufficient to memorize a certain amount of versions/migrations. However, if there are more versions or migrations, the network fails to absorb all the patterns. Such a limitation can be overcome with deeper networks (Nguyen et al., 2021b), i.e., by padding additional hidden units to DeepLib. To validate the hypothesis, we increased the number of units from 40 to 100 and reran the experiments on the libraries with which DeepLib-α gets a low accuracy by most of the folds, i.e., L_{02}, L_{11}, and L_{21}. As expected, we see a gain in accuracy by these libraries. For the sake of clarity, we report the change in accuracy with respect to Table 4 as follows: $Acc_{lib}(L_{02})$: 0.625 → 0.632, $Acc_{lib}(L_{11})$: 0.544 → 0.559, $Acc_{lib}(L_{21})$: 0.574 → 0.613. While by L_{02} and L_{11} there is a marginal increase, we can see a substantial gain by L_{21}. Similarly, by running DeepLib-β with more hidden units, compared to the results in Figs. 13 and 14, we got a minimum and maximum increase in Acc_{pro} of 5% and 18%, respectively.

The improvement suggests that one can enhance DeepLib's prediction performance for those libraries having a large number of versions/migrations by means of deeper networks, i.e., with more hidden units. In this way, we suppose that it is possible to further boost up the predictions for any libraries/clients in practical use, by choosing a suitable network configuration according to the input data.

Answer to RQ_3. DeepLib suffers a deficiency in performance on libraries with a large number of versions and/or migrations. However, depending on the input data, the system's performance can be enhanced with deeper networks.

6. Discussion

We discuss the practicality as well as possible extension of DeepLib in Section 6.1. The threats that might hamper the validity of our findings are presented in Section 6.2.

6.1. Applicability and future developments of DeepLib

A question that might arise at any time is: "*How can DeepLib be deployed in practice?*" As we see from Section 3, it is necessary to collect TPLs together with a set of clients associated with them. Afterwards, we build migration matrices to feed the recommendation engine. Once the collected data has been used to train the system, it can be removed to give place to new data, i.e., projects coming from OSS platforms. In other words, the knowledge learned from data is embedded in the internal weights and biases of the networks. This makes DeepLib a lightweight framework that can be easily deployed. We plan to integrate DeepLib into the Eclipse IDE, providing instant suggestions to developers while they are coding.

The performance of DeepLib is driven by the availability and quality of the training data. Thus we anticipate that it will fail if some library versions, e.g., newly released versions, never occurred in the migration history. In this respect, DeepLib is supposed to learn better only when enough movement history is available, given a specific library version.

Recommendation of library upgrade is a complex problem, and this has been confirmed by various studies (Derr et al., 2017; Raemaekers et al., 2017; Visser et al., 2012). In fact, there are two levels of migration: *(i)* the *library level*; and *(ii)* the *source code level*. By the former, a developer needs to be provided with a suitable version of a library, whereas by the latter, she has to adapt the affected source code to make it work with the new library versions and related APIs. In the scope of this paper, we tackle the first issue, i.e., recommending library migration. We expect that similar techniques employed to build DeepLib can be exploited to support source code migration. In particular, by collecting related projects, we can build matrices with old APIs as features and new APIs as the label. This enables us to deploy the same techniques used in DeepLib-α and DeepLib-β to predict suitable migration steps. This, however, requires us to parse source code to extract the API functions. Moreover, in the scope of this work, we consider only release time for the recommendation of updates. In practice, there are many artifacts having multiple major versions maintained at the same time. Thus, it would make sense to consider also semantic versioning, especially when upgrading on major versions and minor versions. We are going to tackle these issues in our future work.

According to the empirical evaluation, we see that the performance of DeepLib-α and DeepLib-β depends very much on the availability of the training data, i.e., among the two tools, there is no absolute winner in all cases. In particular, DeepLib-α is good at recommending the next version for a single library which consists of enough training data, i.e., when there are a large number of upgrades for the considered library. Meanwhile DeepLib-β outperforms DeepLib-α when there is a balance between the number of upgrades for all the related libraries. The rationale behind such a difference is as follows. In machine translation, an Encoder–Decoder LSTM can better predict the next sequence of words when there exists a frequent combination of words in the input sequence (Cho et al., 2014). Since DeepLib-β is built on top of an Encoder–Decoder LSTM, it inherits the essential qualities of the original machine translation technique. Altogether, this suggests that in practice, the selection of a suitable recommendation strategy for libraries should be made according to the quality of the input data.

6.2. Threats to validity

Threats to *internal validity* are related to the confounding factors in our approach and evaluation that could have influenced the final results. A possible threat is that the datasets might not fully reflect real-world development scenarios as we were are able to consider only popular libraries. In practice, developers tend to work on a variety of libraries. To mitigate the threats, we crawled a wide range of clients across several repositories. Still, we believe that considering data from other sources, such as GitHub, can help fully eliminate the threat.

The main threat to *external validity* concerns the generalizability of our findings. DeepLib has been evaluated on projects collected from Maven, since we have suitable software to fetch the data. We anticipate that our tool is also applicable to other platforms, as long as they support versioning. We plan to generalize DeepLib to data from GitHub in our future work. In fact, contributions in Maven come by strictly following a well-defined process, which is not the case for GitHub repositories, where projects are uploaded in an ad hoc manner. In this respect, it is necessary to carefully investigate the performance of DeepLib on projects curated from GitHub.

7. Related work

Recently, the issue of recommending development of third-party libraries and API usage has been intensively studied (He, Xu et al., 2021). We review notable recommender systems by focusing on those related to the adoption of TPLs and API migrations.

7.1. Recommendation of TPLs

LibRec (Thung et al., 2013) employs a combination of rule mining and collaborative filtering strategies to retrieve libraries considering similar projects to the one given as input. Ouni et al. (2017) develops LibFinder that uses a multi-objective algorithm to detect semantic similarity in source code. CrossRec (Nguyen, Di Rocco, Di Ruscio, Di Penta, 2019) assists developers in selecting suitable TPLs. The system exploits a collaborative filtering technique to recommend libraries by relying on the set of dependencies, which have been included in the project being developed.

The usage of domain-specific category (DSC) concept has been investigated to foster TLP maintenance (Katsuragawa et al., 2018). By relying on a rule mining technique, the approach is capable of

categorizing GitHub projects exploiting labels available on the Maven Central repository. LibSeek (He et al., 2020) employs the matrix factorization (MF) technique to predict relevant TLPs for mobile apps. It adopts an adaptive weighting scheme to reduce the skewness caused by popular libraries. Furthermore, the MF-based algorithm is used to integrate neighborhood information by computing the similarity of libraries contained in the matrix.

Req2Lib (Sun et al., 2020) has been recently proposed to recommend TPLs given textual description of project requirements. The tool employs a seq2seq LSTM which is trained with description and libraries belonging to configuration file. Additionally, a domain-specific embedding model obtained from Stack Overflow is used to encode words in high-dimensional vectors.

All the previously mentioned systems can recommend libraries that can be added in the project being developed. However, they do not provide suggestions that can help upgrade already included dependencies as done by DeepLib.

7.2. Recommendation of migration

Xu et al. proposed Meditor (Xu et al., 2019) to analyze GitHub commits to extract migration-related (MR) changes by mining *pom.xml* files. Once MR updates have been found, the tool employs the WALA framework to check their consistency by analyzing the developer's context and apply them directly. Apiwave (Hora & Valente, 2015) infers and retrieves relevant information related to TPLs, i.e., popularity and migration data. The tool uses two different modules to discover the popularity by analyzing import statements of projects, i.e., the removal of certain API decreases its popularity. Additionally, the system can infer migration data from each API replacement.

Teyton et al. developed a tool that relies on a graph-based structure to address the migration of TPLs (Teyton et al., 2012). The tool makes use of a token-based filter applied on pom.xml to discover libraries' information, i.e., names and versions. Afterwards, each TPL is encoded as a node on the graph and edges express four different visual patterns to suggest the most suitable migration changes given the input library.

SimilarAPI (Chen, 2020) suggests alternative TPLs by exploiting an unsupervised ML approach. Given a set of projects crawled from GitHub, the tool is able to extract API call sequences by using a well-founded partial program analysis Java tool. Then, the obtained information is encoded by using skip thoughts, an RNN capable of embedding the word semantics in a vector space. Given input TPLs, SimilarAPI retrieves the most similar ones with a ranked list of APIs

Fazzini et al. proposed APIMigrator (Fazzini et al., 2020) to support the migration of APIs in Android apps. Starting from API usage (AU) information extracted from the documentation, the tool can obtain generic migration examples that represent historical migration data. Then, generic migration patches are extracted from these explanatory AU to migrate the target app. APIMigrator performs the actual migration by mapping the list of obtained patches one at a time to avoid possible issues due to the overlapping.

LibBandAid (Duan et al., 2019) is an approach to automatic generation of updates for TPLs in Android apps. Given an app with an outdated library and a newer version of the library, the tool recommends how to update the old library in a way that guarantees backward compatibility.

A recent work (Kula et al., 2018) attracts the community attention over the *migration awareness* problem as well as the efforts required to apply actual changes. As the first step, the authors proposed a model to detect TPLs evolution by excerpting migration data from GitHub projects. The proposed model is based on the assumption that systems with more dependencies tend to have more frequent updates. Additionally, the authors conducted a user study to measure the developer's behavior considering the two main migration awareness mechanisms, i.e., security advisories and new releases announcement. The findings of the work are: *(i)* the majority of the software systems rarely update the older but reliable libraries; *(ii)* security advisories provide incomplete solutions; and *(iii)* developers consider the migration task as a non crucial task for the development.

Differently from the aforementioned approaches, DeepLib is able to learn from what other projects have done to recommend the next upgrades that maintainers should operate on one or more libraries already included in their project. DeepLib is even able to recommend removals of dependencies according to migrations that have been performed. Migrating the source code that might get affected by the recommended upgrades is not in the scope of this paper, and we plan it as future work.

8. Conclusion and future work

Software systems heavily rely on third-party libraries (TPLs), which provide a wide range of functionalities that can be reused without the need to re-implement them from scratch. Even though TPLs evolve, e.g., to fix security holes or to increase the provided capabilities, most systems rarely update their dependencies. Developers consider TPL migration as a practice that can introduce extra efforts and responsibility. Things can get even more complex when developers have to identify which libraries need to be migrated and what are the target versions that have to be considered.

To reduce the burden related to the identification of the upgrades that need to be operated on the current system we proposed DeepLib, a novel approach to recommendation of the next version for the used TPLs by considering migration histories of several OSS projects. Our proposed tool is able to extract relevant migration data and encode it in matrices. Then, deep learning techniques are employed to provide recommendations that are relevant for the current configuration. Once being deployed, DeepLib allows developers to quickly select a suitable migration, by relying on the experience of other projects with similar sets of TPLs and thus, helping to eliminate any possible complexity concerning the technical details which were already mentioned in Section 2.

As future work, we plan to evaluate DeepLib on specific ecosystems including that of Android apps. Afterward, we also intend to investigate the possibility of applying the proposed techniques to support the migration of source code, which can be affected by the proposed upgrade plans. Moreover, an open research issue is investigating the feasibility of DeepLib in predicting potential conflict on the dependent libraries when there are multiple target libraries. To this end, it is necessary to have adequate training data collected from Maven and carefully validated by humans. Therefore, we consider this as a part of our future research agenda.

CRediT authorship contribution statement

Phuong T. Nguyen: Conceptualization, Methodology, Writing – original draft, Writing – review & editing. **Juri Di Rocco:** Data curation, Visualization, Writing – review & editing. **Riccardo Rubei:** Data curation, Validation, Writing – review & editing. **Claudio Di Sipio:** Software, Validation, Writing – original draft. **Davide Di Ruscio:** Methodology, Writing – original draft, Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The research described in this paper has been partially supported by the AIDOaRT Project, which has received funding from the European Union's H2020-ECSEL-2020, Federal Ministry of Education, Science and Research, Grant Agreement n 101007350. We thank the anonymous reviewers for their valuable comments and suggestions that helped us improve the paper.

References

Bauer, V., Heinemann, L., & Deissenboeck, F. (2012). A structured approach to assess third-party library usage. In *2012 28th IEEE international conference on software maintenance (ICSM)* (pp. 483–492). http://dx.doi.org/10.1109/ICSM.2012.6405311.

Benelallam, A., Harrand, N., Soto-Valero, C., Baudry, B., & Barais, O. (2019). The maven dependency graph: a temporal graph-based representation of maven central. In *2019 IEEE/ACM 16th international conference on mining software repositories (MSR)* (pp. 344–348). IEEE.

Benelallam, A., Harrand, N., Valero, C. S., Baudry, B., & Barais, O. (2018). Maven central dependency graph. http://dx.doi.org/10.5281/zenodo.1489120.

Chen, C. (2020). Similarapi: Mining analogical APIs for library migration. In *2020 IEEE/ACM 42nd international conference on software engineering: companion proceedings (ICSE-companion)* (pp. 37–40). ISSN: 2574-1926.

Cho, K., van Merriënboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: Encoder–decoder approaches. In *Proceedings of SSST-8, eighth workshop on syntax, semantics and structure in statistical translation* (pp. 103–111). Doha, Qatar: Association for Computational Linguistics, http://dx.doi.org/10.3115/v1/W14-4012, URL https://www.aclweb.org/anthology/W14-4012.

Dependabot, (0000). Keep all your packages up to date with dependabot - The GitHub Blog. URL https://github.blog/2020-06-01-keep-all-your-packages-up-to-date-with-dependabot/.

Derr, E., Bugiel, S., Fahl, S., Acar, Y., & Backes, M. (2017). Keep me updated: An empirical study of third-party library updatability on android.. In B. M. Thuraisingham, D. Evans, T. Malkin, & D. Xu (Eds.), *ACM conference on computer and communications security* (pp. 2187–2200). ACM, ISBN: 978-1-4503-4946-8, URL http://dblp.uni-trier.de/db/conf/ccs/ccs2017.html#DerrBFA017.

Di Rocco, J., Di Ruscio, D., Di Sipio, C., Nguyen, P. T., & Rubei, R. (0000). Development of recommendation systems for software engineering: the CROSSMINER experience, 26 (4) 69. http://dx.doi.org/10.1007/s10664-021-09963-7 ISSN 1573-7616.

Di Rocco, J., Di Ruscio, D., Di Sipio, C., Nguyen, P., & Rubei, R. (2020). TopFilter: AN approach to recommend relevant GitHub topics. In *ESEM '20*, *Proceedings of the 14th ACM / IEEE international symposium on empirical software engineering and measurement (ESEM)*. New York, NY, USA: Association for Computing Machinery, ISBN: 9781450375801, http://dx.doi.org/10.1145/3382494.3410690.

Di Rocco, J., Di Sipio, C., Nguyen, P. T., Di Ruscio, D., & Rubei, R. (2022). DeepLib: Machine translation techniques to recommend upgrades for third-party libraries. URL https://codeocean.com/capsule/8397858/tree/v1.

Di Sipio, C., Rubei, R., Di Ruscio, D., & Nguyen, P. T. (2020). Using a multinomial naïve Bayesian (MNB) network to automatically recommend topics for GitHub repositories. In *EASE'20*, *Proceedings of the 24th international conference on evaluation and assessment in software engineering, EASE2020, Trondheim, Norway, April 15-17, 2020* (pp. 24–34). ACM, http://dx.doi.org/10.1145/3383219.3383227.

Duan, Y., Gao, L., Hu, J., & Yin, H. (2019). Automatic generation of non-intrusive updates for third-party libraries in android applications. In *22nd international symposium on research in attacks, intrusions and defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019* (pp. 277–292). USENIX Association, URL https://www.usenix.org/conference/raid2019/presentation/duan.

Erlenhov, L., Gomes de Oliveira Neto, F., Scandariato, R., & Leitner, P. (2019). Current and future bots in software development. In *2019 IEEE/ACM 1st international workshop on bots in software engineering (BotSE)* (pp. 7–11). Montreal, QC, Canada: IEEE, ISBN: 978-1-72812-262-5, http://dx.doi.org/10.1109/BotSE.2019.00009.

Fazzini, M., Xin, Q., & Orso, A. (2020). Apimigrator: an API-usage migration tool for android apps. In *Proceedings of the IEEE/ACM 7th international conference on mobile software engineering and systems* (pp. 77–80). Seoul Republic of Korea: ACM, ISBN: 978-1-4503-7959-5, http://dx.doi.org/10.1145/3387905.3388608, URL https://dl.acm.org/doi/10.1145/3387905.3388608.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press, http://www.deeplearningbook.org.

Grissom, R. J., & Kim, J. J. (2005). *Effect sizes for research: a broad practical approach* (2nd ed.). Lawrence Earlbaum Associates.

He, H., He, R., Gu, H., & Zhou, M. (2021). A large-scale empirical study on java library migrations: Prevalence, trends, and rationales. In *ESEC/FSE 2021*, *Proceedings of the 29th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering* (pp. 478–490). New York, NY, USA: Association for Computing Machinery, ISBN: 9781450385626, http://dx.doi.org/10.1145/3468264.3468571.

He, Q., Li, B., Chen, F., Grundy, J., Xia, X., & Yang, Y. (2020). Diversified third-party library prediction for mobile app development. *IEEE Transactions on Software Engineering*, 1.

He, H., Xu, Y., Ma, Y., Xu, Y., Liang, G., & Zhou, M. (2021). A multi-metric ranking approach for library migration recommendations. In *2021 IEEE international conference on software analysis, evolution and reengineering (SANER)* (pp. 72–83). http://dx.doi.org/10.1109/SANER50967.2021.00016.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, [ISSN: 0899-7667] *9*(8), 1735–1780. http://dx.doi.org/10.1162/neco.1997.9.8.1735.

Hora, A., & Valente, M. T. (2015). Apiwave: Keeping track of API popularity and migration. In *2015 IEEE int. conf. on software maintenance and evolution (ICSME)* (pp. 321–323). http://dx.doi.org/10.1109/ICSM.2015.7332478.

Huang, J., Borges, N., Bugiel, S., & Backes, M. (2019). Up-to-crash: Evaluating third-party library updatability on android. In *2019 IEEE European symposium on security and privacy (EuroS P)* (pp. 15–30). http://dx.doi.org/10.1109/EuroSP.2019.00012.

Katsuragawa, D., Ihara, A., Kula, R. G., & Matsumoto, K. (2018). Maintaining third-party libraries through domain-specific category recommendations. In *2018 IEEE/ACM 1st international workshop on software health (SoHeal)* (pp. 2–9).

Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *14th international joint conference on artificial intelligence* (pp. 1137–1143). San Francisco: Morgan Kaufmann Publishers Inc., ISBN: 1-55860-363-8.

Kula, R. G., German, D. M., Ouni, A., Ishio, T., & Inoue, K. (2018). Do developers update their library dependencies?: an empirical study on the impact of security advisories on library migration. *Empirical Software Engineering*, *23*(1), 384–417. http://dx.doi.org/10.1007/s10664-017-9521-5, ISSN 1382-3256, 1573-7616.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444. http://dx.doi.org/10.1038/nature14539.

Li, Z., Avgeriou, P., & Liang, P. (2015). A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, [ISSN: 01641212] *101*, 193–220. http://dx.doi.org/10.1016/j.jss.2014.12.027.

Nguyen, P. T., Di Rocco, J., Di Ruscio, D., & Di Penta, M. (2019). CrossRec: SUpporting software developers by recommending third-party libraries. *Journal of Systems and Software*, [ISSN: 0164-1212] Article 110460, URL http://www.sciencedirect.com/science/article/pii/S0164121219302341.

Nguyen, P. T., Di Rocco, J., Di Ruscio, D., Ochoa, L., Degueule, T., & Di Penta, M. (2019). FOCUS: A Recommender system for mining API function calls and usage patterns. In *ICSE '19*, *Proceedings of the 41st international conference on software engineering* (pp. 1050–1060). Piscataway, NJ, USA: IEEE Press, http://dx.doi.org/10.1109/ICSE.2019.00109.

Nguyen, P. T., Di Rocco, J., Di Sipio, C., Di Ruscio, D., & Di Penta, M. (2021). Recommending API function calls and code snippets to support software development. *IEEE Transactions on Software Engineering*, 1. http://dx.doi.org/10.1109/TSE.2021.3059907.

Nguyen, P. T., Di Rocco, J., Rubei, R., Di Sipio, C., & Di Ruscio, D. (2021). Recommending third-party library updates with LSTM neural networks. In *Procs. 11th Italian information retrieval workshop 2021*. URL http://52.178.216.184/paper7.pdf.

Nguyen, P. T., Di Ruscio, D., Pierantonio, A., Di Rocco, J., & Iovino, L. (2021). Convolutional neural networks for enhanced classification mechanisms of metamodels. *Journal of Systems and Software*, [ISSN: 0164-1212] *172*, Article 110860. http://dx.doi.org/10.1016/j.jss.2020.110860, URL https://www.sciencedirect.com/science/article/pii/S0164121220302508.

Olah, C. (2020). Understanding LSTM networks. URL https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Ouni, A., Kula, R. G., Kessentini, M., Ishio, T., German, D. M., & Inoue, K. (2017). Search-based software library recommendation using multi-objective optimization. *Information and Software Technology*, [ISSN: 0950-5849] *83*(C), 55–75. http://dx.doi.org/10.1016/j.infsof.2016.11.007.

Raemaekers, S., van Deursen, A., & Visser, J. (2017). Semantic versioning and impact of breaking changes in the maven repository. *Journal of Systems and Software*, [ISSN: 0164-1212] *129*, 140–158, URL http://www.sciencedirect.com/science/article/pii/S0164121216300243.

Rawat, W., & Wang, Z. (2017). Deep convolutional neural networks for image classification: A comprehensive review. *Neural Computation*, [ISSN: 0899-7667] *29*(9), 2352–2449. http://dx.doi.org/10.1162/neco_a_00990.

Robillard, M. P., Maalej, W., Walker, R. J., & Zimmermann, T. (Eds.), (2014). Recommendation systems in software engineering. Berlin, Heidelberg: http://dx.doi.org/10.1007/978-3-642-45135-5, ISBN 978-3-642-45134-8 978-3-642-45135-5.

Rubei, R., Di Sipio, C., Nguyen, P. T., Di Rocco, J., & Di Ruscio, D. (2020). PostFinder: MIning stack overflow posts to support software developers. *Information and Software Technology*, [ISSN: 0950-5849] *127*, Article 106367. http://dx.doi.org/10.1016/j.infsof.2020.106367, URL http://www.sciencedirect.com/science/article/pii/S0950584920301361.

Shi, X., Shao, X., Guo, Z., Wu, G., Zhang, H., & Shibasaki, R. (2019). Pedestrian trajectory prediction in extremely crowded scenarios. *Sensors*, *19*, 1223. http://dx.doi.org/10.3390/s19051223.

Sun, Z., Liu, Y., Cheng, Z., Yang, C., & Che, P. (2020). Req2Lib: A semantic neural model for software library recommendation. In *2020 IEEE 27th international conference on software analysis, evolution and reengineering (SANER)* (pp. 542–546). http://dx.doi.org/10.1109/SANER48275.2020.9054865, ISSN: 1534-5351.

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *NIPS'14*, *Proceedings of the 27th international conference on neural information processing systems - Volume 2* (pp. 3104–3112). Cambridge, MA, USA: MIT Press.

Teyton, C., Falleri, J.-R., & Blanc, X. (2012). Mining library migration graphs. In *2012 19th Working Conf. on Reverse Engineering* (pp. 289–298). http://dx.doi.org/10.1109/WCRE.2012.38.

Thung, F., Lo, D., & Lawall, J. (2013). Automated library recommendation. In *2013 20th working conference on reverse engineering (WCRE)* (pp. 182–191). http://dx.doi.org/10.1109/WCRE.2013.6671293.

Visser, J., van Deursen, A., & Raemaekers, S. (2012). Measuring software library stability through historical version analysis. In *ICSM '12, Proceedings of the 2012 IEEE international conference on software maintenance (ICSM)* (pp. 378–387). USA: IEEE Computer Society, ISBN: 9781467323130, http://dx.doi.org/10.1109/ICSM.2012.6405296.

Wang, Y., Chen, B., Huang, K., Shi, B., Xu, C., Peng, X., Wu, Y., & Liu, Y. (2020). An empirical study of usages, updates and risks of third-party libraries in java projects. In *2020 IEEE international conference on software maintenance and evolution (ICSME)* (pp. 35–45). http://dx.doi.org/10.1109/ICSME46990.2020.00014.

Xu, S., Dong, Z., & Meng, N. (2019). Meditor: Inference and application of API migration edits. In *2019 IEEE/ACM 27th int. conf. on program comprehension (ICPC)* (pp. 335–346). http://dx.doi.org/10.1109/ICPC.2019.00052.

Yamashita, R., Nishio, M., Do, R. K. G., & Togashi, K. (2018). Convolutional neural networks: an overview and application in radiology. *Insights Into Imaging*, [ISSN: 1869-4101] *9*(4), 611–629. http://dx.doi.org/10.1007/s13244-018-0639-9.