

A Two-Phase Bug Localization Approach Based on Multi-layer Perceptrons and Distributional Features

Damiano Distante and Stefano Faralli(✉)

University of Rome Unitelma Sapienza, Rome, Italy
{damiano.distante,stefano.faralli}@unitelmasapienza.it

Abstract. Bug localization is a challenging and time-consuming task of the process of bug fixing and, more in general, of software maintenance. Several approaches have been proposed in the literature which support developers in this task by identifying source code files in which the bug is likely to be located. However, the research on this topic never stopped, looking for new methods providing better accuracy and/or better efficiency. In this paper, we propose a two-phase bug localization approach which leverages multi-layer neural networks and distributional features. First phase locations are obtained thanks to a neural network trained on word embeddings representations of fixed bug reports. The second phase refines bug locations taking into account the number of times source code files co-occur in fixed bug locations. To evaluate the approach, we conducted a large-scale experiment on five open source projects, namely *Mozilla*, *Eclipse*, *Dolphin*, *httpd*, and *gcc*. Results show that, thanks to pre-trained word embeddings, we were able to implement a scalable approach with a training running time of few hours on large datasets. Performances are comparable to other existing deep learning approaches.

Keywords: Software maintenance · Bug localization · Machine learning · Word embeddings · Multilayer perceptron networks

1 Introduction

A software bug is a defect or mistake in the code of a computer program or system that may cause it to produce an unintended and unexpected behavior.

To inform developers of the presence of bugs, users describe the observed erroneous behaviors in the form of bug reports collected by issue trackers such as Bugzilla[1].

After a bug is reported, its life-cycle [7] towards resolution is started[2]. Such life-cycle usually includes a first phase of bug triage in which the bug is verified

[1] https://www.bugzilla.org.
[2] https://www.bugzilla.org/docs/4.2/en/html/lifecycle.html.

© Springer Nature Switzerland AG 2019
S. Misra et al. (Eds.): ICCSA 2019, LNCS 11619, pp. 518–532, 2019.
https://doi.org/10.1007/978-3-030-24289-3_38

to be actually present, its severity is assessed, and a level of priority is assigned. Next, the bug is assigned to a developer that will first try to localize it in the source code and then find a way to solve it. A patch fixing the bug is then provided and, once it is verified to actually work, it is later introduced in the system so to remove the bug from next releases of the software.

Bug localization is the task aimed at identifying the portion of the source code of a software system causing its to expose an erroneous behavior. This code portion may actually be spread over several source code files and, depending on the size and complexity of the software system, the task of bug localization may be very challenging and time-consuming.

In order to support developers in accomplishing this task, different approaches have been proposed in the literature (see Sect. 5) which adopt a variety of source code analysis and data mining techniques. However, the research on this topic never stopped, looking for both more scalable and more effective approaches [15].

In this work, we propose a two-phase bug localization approach based on multi-layer perceptrons and distributional features that has proved to effectively address the scalability issue.

In particular, in the first phase we leverage pre-trained compact vectors (also known as word embeddings) to represent bug reports. In the second phase we exploit observed distributional features to possibly improve the localization results provided by the neural network.

To evaluate our approach, we first tested it on a dataset including more than 300,000 bugs of 5 open source projects (namely, *Mozilla*, *Eclipse*, *Dolphin*, *httpd*, and *gcc*) and then answered the following research questions:

- **RQ1:** *To what extent are word embeddings based bug reports representations informative enough to train multi-layer perceptron networks to perform bug localization?*
- **RQ2:** *To what extent is it possible to improve the performance of the above classifiers by introducing observed distributional features.*

The paper is organized as follows. In Sect. 2 we formally introduce the task of bug localization. In Sect. 3 we describe the proposed two-phase approach in detail. In Sect. 4 we estimate the performances of the system and discuss our findings. The current state of the is briefly described in Sect. 5. Finally in Sect. 6 we report conclusive remarks and announce future work.

2 Problem Statement

The task of automatic bug localization consists in predicting the source code files in which the fault in the code causing the erroneous behavior of the system is located.

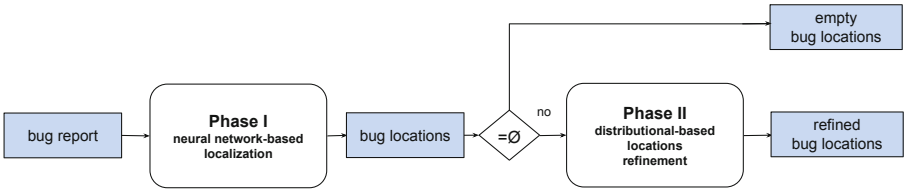

Fig. 1. Workflow of the two-phase approach.

More formally, let:

- ID_{new} be the ID of a new reported bug for a given system;
- $file_r$ be one of the source code files of the system;
- $Loc(ID_x)$ be the function that maps each bug ID_x into a vector such that $Loc(ID_x)_r$ is 1 if $file_r$ is involved into the fix of the bug ID_x;

then, the task of bug localization is defined as the prediction of the values of the function $Loc(ID_{new})$ for any arbitrary new reported bug.

We explain our approach for automatic bug localization in detail in next Sect. 3.

3 Approach

Our approach for bug localization is composed of two phases (see Fig. 1). In Phase 1, we predict the location of a newly reported bug ID_{new} using a localization function $Loc(ID_x)$ learned on top of *summaries* of past resolved bugs via Multi-Layer Perceptron (MLP) networks. The *summary* (or short description) of a bug is a short sentence provided by the bug reporter at reporting time which succinctly describes what the bug is about (e.g., *"Problem in user interface refresh"*). In Phase 2, we complement locations obtained in Phase 1 (particularly for bugs for which $Loc(ID_x)$ produced an empty result) by using distributional features.

3.1 Phase 1: Multi-layer Perceptron Network

To predict locations of a newly reported bug ID_{new} reported at time t_{new}, we propose a supervised approach (see Fig. 2) where:

1. we collect the set of resolved bugs RB, with a resolution time in between the two timestamps $(t_{new} - \Delta, t_{new})$, for which we could find one or more commits in the git repository of the project applied to fix the bug. In order to identify such commits, we search for a reference to the bug ID_x in the commit notes using a specific search pattern (regular expression) for each project.
2. for each collected bug $ID_x \in RB$ we compute a word embeddings vector as follows:

$$\boldsymbol{W}(S(ID_x)) = \frac{\sum_{i=0}^{|S(ID_x)|} \boldsymbol{we}(word_i(S(ID_x)))}{|S(ID_x)|}$$

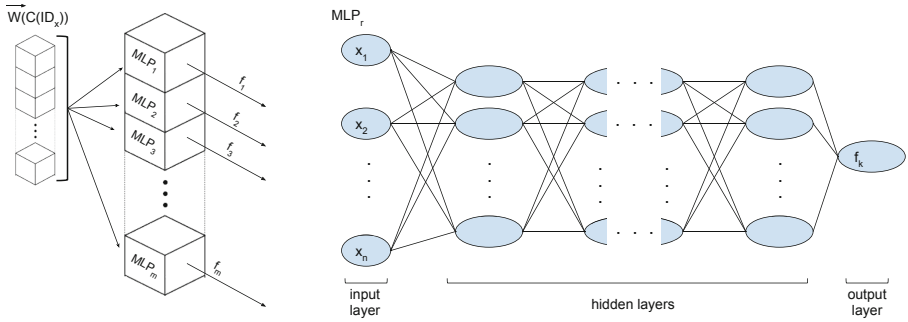

Fig. 2. Representation of the stack of Multi-Layer Perceptron networks used in Phase 1 of our approach to predict the location of fresh new reported bugs.

where: (i) $|S(ID_x)|$ is the number of words forming the *summary* of the reported bug (e.g., for a provided *summary* of the kind "Problem in user interface refresh" we count 5 words); (ii) $word_i$ is the function that returns the i-th word of a provided text (e.g., $word_4$("Problem in user interface refresh") returns "interface"); (iii) *we* is a function that retrieves a pre-trained word embedding vector for a given word from a given repository. Hence, we construct a vector representation of the text of the *summary* as the average of the vector representations of each word forming the text. In Fig. 3 we show an example (in two dimensions) where the word embedding vector for the *summary* "problem in user interface refresh" is obtained as the average vector of the component words "problem", "in", "user", "interface" and "refresh".

3. we define a localization function $Loc(ID_x)_r$, which for a given bug $ID_x \in RB$ and for a given file $file_r$ of the project returns:

$$Loc(ID_x)_r = \begin{cases} 1, & \text{if } file_r \text{ was modified to fix bug } ID_x \\ 0, & \text{otherwise} \end{cases}$$

in other words the function $Loc(ID_x)_r$ determines if a $file_r$ has been involved in a commit for the resolution of the bug ID_x;

4. as represented in Fig. 2, for each file f_r we train a multi-layer perceptron network MLP_r to predict if a file f_r is involved in the fix of a bug:

$$Loc(ID_{new})_r = MLP_r(\boldsymbol{W}(ID_{new})).$$

Each MLP_r is trained on the feature vectors $\boldsymbol{W}(ID_x)$ and on the target binary values of $Loc(ID_x)_r$ for the bugs $ID_x \in RB$ using a standard back propagation algorithm to determine networks weights and biases. The predicted localization of a bug ID_{new} will be the set $L(ID_{new}) = \{f_r : MLP_r(\boldsymbol{W}(ID_{new})) = 1\}$.

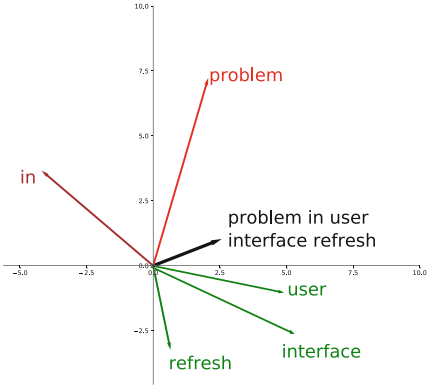

Fig. 3. Artificial example of creation of the word embeddings for a *summary* "problem in user interface refresh". The resulting vector is the average vector of the bi-dimensional word embedding vectors for the words "problem", "in", "user", "interface" and "refresh".

3.2 Phase 2: Distributional-Based Localization

To address the presence of empty predictions for a new reported bug ID_{new} (which affects the recall performances of the Phase 1 approach) we propose the following strategy:

Step. 1 we compute $\boldsymbol{W}(S(ID_{new}))$ as described in Sect. 3.1 step 2;
Step. 2 for each candidate file f_i involved in a commit which fixes a bug $ID_t \in RB$, we define $BL_{f_i} = \{ID_t : Loc(ID_t)_i = 1\}$ and create the following average vector representation:

$$\boldsymbol{W}(f_i) = \frac{\sum_{ID_t \in BL_{f_i}} \boldsymbol{W}(C(ID_t))}{|BL_{f_i}|};$$

Step. 3 we remove from the set of candidates those files f_i such that $|BL_{f_i}|$ is less then a parameter threshold $FreqTH$;
Step. 4 we then assign to the remaining candidate locations f_i the cosine similarity with the vector $\boldsymbol{W}(S(ID_{new}))$ as follows:

$$score(ID_{new})_{f_i} = \frac{\boldsymbol{W}(S(ID_{new})) \cdot \boldsymbol{W}(f_i)}{||\boldsymbol{W}(S(ID_{new}))|| \cdot ||\boldsymbol{W}(f_i)||};$$

Step. 5 we remove from the set of candidate locations those files f_i such that $score(ID_{new})_{f_i}$ is less then a parameter $SimTH$ threshold;
Step. 6 we finally create $L(ID_{new})$ as the set of top K scored candidate files.

4 Experiments

To evaluate the effectiveness of the proposed approach we designed two experiments. The experiments (see Sects. 4.3 and 4.4) were devoted to estimate the performances of each phase, namely the MLP based approach (described in Sect. 3.1) and the distributional based one (described in Sect. 3.2). The following two paragraphs describe the dataset and the measures involved in the two experiments.

Table 1. Selected timestamps

T	Training set starts	Training set ends and test set starts	Test set ends
1	28 October 2001	30 May 2003	27 July 2003
2	17 June 2003	15 January 2005	14 March 2005
3	2 February 2005	4 September 2006	1 November 2006
4	21 September 2006	22 April 2008	19 June 2008
5	10 May 2008	9 December 2009	5 February 2010
6	27 December 2009	29 July 2011	25 September 2011
7	16 August 2011	16 March 2013	13 May 2013
8	3 April 2013	3 November 2014	30 December 2014

4.1 Dataset

Our dataset includes more than 300,000 fixed bugs across 5 open source software systems (namely, *Mozilla*, *Eclipse*, *Dolphin*, *httpd*, and *gcc*) covering a period of 13 years between 28 October 2001 and 30 December 2014.

To recreate a realistic scenario, we split the 13 years into eight time windows from which we collected the fixed bug reports for both training and test (see Table 1). We empirically experimented better results with windows of about 19 and 2 months for training and test, respectively.

We extracted bug reports of fixed bugs from the *Bugzilla issue tracker* of each of the open source projects. For each fixed bug we harvested from the corresponding project version control system (a git repository) the list of source code files involved in the fix.

In Table 2, for each of the systems and time windows, we report the number of bugs and fixed files observed, with averages and standard deviations.

For the implementation of the step 2 of Phase 1 approach (see Sect. 3.1) we adopt *Glove* as a set of pre-trained word embedding vector collection [9]. *Glove* is a general model for distributed word representation. Training is performed on aggregated global word-word co-occurrence statistics from a corpus.

We used pre-trained vectors of length 300 from the collection "glove.6B" trained on a corpus made with the combination of "Wikipedia 2014" and "Gigaword 5" and covering 400K uncased words[3,4].

[3] https://nlp.stanford.edu/projects/glove/.
[4] https://catalog.ldc.upenn.edu/LDC2011T07.

Table 2. Dataset descriptive statistics

		#bugs			files					#bugs			files		
	T	total	train	test	total	avg	std		T	total	train	test	total	avg	std
mozilla.core	1	3,659	3,218	441	6,459	6.12	16.98	mozilla.firefox	1	96	31	65	48	3.55	2.39
	2	2,258	1,816	442	6,130	7.57	40.41		2	149	107	42	245	2.94	6.06
	3	3,600	3,144	456	7,699	5.55	28.34		3	1,120	899	221	1538	3.22	10.94
	4	4,517	4,168	349	16,756	7.45	101.8		4	1,148	997	151	1,458	3.06	5.89
	5	7,109	6,481	628	16,716	6.47	23.89		5	1,650	1,587	63	1,805	3.53	5.85
	6	11,623	9,923	1,700	27,054	7.11	40.76		6	2,019	1,761	258	2,994	4.26	11.23
	7	17,493	15,074	2,419	43,207	8.55	57.73		7	2,750	2,431	319	6,093	5.59	33.87
	8	21,725	19,680	2,045	54,293	8.95	45.63		8	5,687	5,247	440	8,816	4.98	10.57
eclipse.platform	1	483	29	454	1,620	146	357.92	gcc	1	—	—	—	—	—	—
	2	1,265	432	833	102	1.16	1.10		2	2,283	1,896	387	9,380	9.13	108.37
	3	1,793	1,224	569	4,294	8.84	134.71		3	2,818	2,505	313	4,347	7.10	17.85
	4	1,279	453	826	3,619	17.26	214.35		4	2,420	2,136	284	7,344	12.47	171.78
	5	993	669	324	4,588	13.14	162.27		5	2,316	1,958	358	11,907	833.81	1650.43
	6	681	526	155	3,989	47.33	369.53		6	2,987	2,657	330	11,770	34.63	344.22
	7	253	111	142	310	4.16	11.69		7	2,459	2,078	381	10,184	18.69	254.24
	8	220	72	148	370	7.75	24.39		8	2,228	1,765	463	7,294	11.46	154.37
dolphin	1	—	—	—	—	—	—	httpd	1	—	—	—	—	—	—
	2	—	—	—	—	—	—		2	357	324	33	296	2.01	2.26
	3	—	—	—	—	—	—		3	155	145	10	157	1.76	1.40
	4	130	69	61	67	2.32	1.99		4	213	172	41	413	3.64	16.74
	5	246	155	91	109	2.22	2.03		5	132	110	22	147	2.23	2.76
	6	193	135	58	127	2.51	2.32		6	200	161	39	237	2.49	2.84
	7	249	228	21	159	2.97	3.16		7	190	175	15	214	2.50	3.08
	8	170	163	7	127	2.14	1.66		8	122	112	10	134	2.25	2.60

We release the datasets under a Creative Commons Attribution 4.0 International (CC BY 4.0) license at the following link: https://sites.google.com/unitelmasapienza.it/buglocalization.

4.2 Measures

To evaluate our two phase approach for each time window T, we identify the training set RB (see Sect. 3) as the set of bugs resolved during the observation time window and $Test(T)$ as the test set. For each $ID_n \in Test(T)$, let $A(ID_n)$ be the set of actual locations for the bug ID_n and $L(ID_n)$ the set of predicted locations; we compute true positives (TP), false positives (FP), false negatives (FN) and true negatives (TN) as follows:

$$TP_n = |L(ID_n) \cap A(ID_n)|;$$
$$FP_n = |L(ID_n)/A(ID_n)|;$$
$$FN_n = |A(ID_n)/L(ID_n)|;$$

$$TN(ID_n) = |\bigcup_{ID_i \in RB} A(ID_i)| - FN_n - TP_n - FP_n.$$

Then, we aggregate the above values as $TP(T)$, $TN(T)$, $FP(T)$ and $FN(T)$ summing the $TP(ID_n)$, $TN(ID_n)$, $FP(ID_n)$ and $FN(ID_n)$ respectively of each bug with $ID_n \in Test(T)$.
We then compute standard precision, recall and F_1 as follows:

$$P = \frac{TP(T)}{TP(T)+FP(T)};$$

$$R = \frac{TP(T)}{TP(T)+FN(T)};$$

$$F_1 = 2 * \frac{(P(T)*R(T))}{P(T)+R(T)}.$$

To better estimate the performances we do report the following measures:

- FullyCovered (FC), the ratio of bugs for which the system provides a full correct location:

$$FC = \frac{|ID_n \in Test(T) : A(ID_n) = L(ID_n)|}{|Test(t)|}$$

- FullyUncovered (FU), the ratio of bugs for which the system provide an empty set of files:

$$FU = \frac{|ID_n \in Test(T) : L(ID_n) = \emptyset|}{|Test(T)|}$$

- PartiallyCovered (PC), the ratio of bugs for which the system provide at least a file (wrong or correct):

$$PC = \frac{|ID_n \in Test(T) : |L(ID_n)| > 0|}{|Test(T)|}$$

Finally, we additionally defined $R'(T)$ as the recall estimated by counting true positives and false negatives only when at least an answer (correct or wrong) is provided by our system.

4.3 Evaluation of Phase 1

To evaluate the Phase 1 approach described in Sect. 3.1 we empirically defined the layout of MLP networks by setting different values for the number and sizes of the hidden layers. Best performing networks consists of four hidden layers with 250, 160, 80 and 40 perceptrons respectively. We fixed an upper bound to the number of iterations equals to 500, and applied standard scaling to input vectors.[5]

In Table 3 we report for each dataset portion and for each time window T the resulting performance of bug localization. We noticed that:

- the average F_1 range from .51(±.04) to .78(±.11) for *gcc* and *eclipse.platform* respectively;

[5] For the implementation of the MLP network we used scikit-learn 0.19.2.

Table 3. Phase 1 Localization performances. Accuracies are around 0.99; Random performances are all under .001; Conf is the 0.95 confidence interval.

	mozilla.core							mozilla.firefox							eclipse.platform						
T	F_1	P	R	FC	PC	FU	R'	F_1	P	R	FC	PC	FU	R'	F_1	P	R	FC	PC	FU	R'
1	.63	.98	.46	.35	.68	.32	.58	.59	.52	.69	.60	1.0	.00	.68	.66	.66	.66	.66	1.0	.00	.66
2	.58	.98	.40	.37	.71	.28	.64	.87	1.0	.80	.75	.75	.25	1.0	.94	.98	.91	.91	.96	.03	.93
3	.49	.97	.33	.29	.58	.41	.55	.77	.98	.63	.50	.76	.23	.72	.70	.85	.59	.57	.80	.19	.74
4	.67	.95	.52	.41	.79	.20	.61	.60	.99	.43	.46	.76	.23	.47	.57	.89	.42	.69	.79	.20	.84
5	.62	.94	.46	.28	.73	.26	.53	.58	.93	.42	.35	.66	.33	.62	.63	.72	.56	.47	.80	.20	.62
6	.61	.92	.45	.32	.77	.22	.49	.58	.98	.41	.29	.74	.25	.48	.91	.95	.87	.78	.95	.04	.88
7	.50	.89	.34	.31	.79	.20	.38	.52	.87	.37	.28	.70	.29	.42	.86	1.0	.75	.57	.57	.42	1.0
8	.64	.89	.50	.29	.74	.25	.55	.49	.93	.33	.40	.76	.23	.42	1.0	1.0	1.0	1.0	1.0	.00	1.0
Avg	.59	.94	.43	.33	.72	.27	.54	.62	.9	.51	.45	.77	.23	.60	.78	.88	.72	.70	.86	.13	.83
Conf	.04	.02	.05	.03	.05	.05	.05	.09	.11	.12	.11	.07	.07	.14	.11	.09	.14	.12	.10	.10	.10
	dolphin							httpd							gcc						
T	F_1	P	R	FC	PC	FU	R'	F_1	P	R	FC	PC	FU	R'	F_1	P	R	FC	PC	FU	R'
1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
2	—	—	—	—	—	—	—	.67	1.0	.50	.37	.62	.37	.72	.71	.91	.58	.24	.99	.04	.58
3	—	—	—	—	—	—	—	1.0	1.0	1.0	1.0	1.0	.00	1.0	.75	.95	.61	.24	.98	.01	.61
4	.84	.88	.79	.72	.86	.13	.88	.87	1.0	.77	.70	.77	.22	.95	.11	.99	.05	.19	.95	.04	.06
5	.87	.90	.83	.79	.96	.03	.86	.69	1.0	.53	.70	.70	.30	1.0	.56	.48	.65	.26	.99	.04	.65
6	.32	1.0	.19	.33	.50	.50	.66	.16	.67	.09	.23	.35	.64	.16	.26	.70	.16	.27	.98	.01	.15
7	.76	.93	.64	.53	.69	.31	.77	.25	1.0	.14	.20	.20	.80	1.0	.93	.99	.88	.13	.99	.04	.88
8	.80	1.0	.66	.60	.60	.40	1.0	1.0	1.0	1.0	1.0	1.0	.00	1.0	.25	.95	.14	.25	.99	.04	.14
Avg	.72	.94	.62	.59	.72	.27	.83	.66	.95	.57	.60	.66	.33	.83	.51	.85	.43	.22	.98	.03	.44
Conf	.19	.05	.22	.16	.16	.17	.11	.25	.09	.27	.25	.22	.22	.23	.23	.14	.23	.03	.01	.01	.23

- the average ratio of FullyCoverd (FC) (see Sect. 4.2) is in the range .22(±.03) to .77(±.07) for *gcc* and *mozilla.firefox* respectively;
- the average ratio of PartiallyCoverd (PC) (see Sect. 4.2) is in the range .66(±.22) to .98(±.01) for *httpd* and *gcc* respectively.

In general the reported estimations enable us to deduce that the proposed system performs better when location consists in average of a fewer number of files per bug, and when there is a number of observed files in the training set which is large enough to cover the localization of unobserved bug (recall) but small enough to simplify the classification task (precision).

Finally, the R' measure (see Sect. 4.2) and the average ratio of bugs PC (for which at least a correct or wrong answer is provided) suggest the investigation of a second phase approach to complement the predicted partial localization.

4.4 Evaluation of Phase 2

In this section we estimate the performances of the proposed second phase approach (see Sect. 3.2) to refine the localization results provided by the MLP-based approach (see Sect. 3.1). Since the proposed methodology includes three parameters namely $FreqTH$, $SimTH$ and K (steps 3, 5 and 6 Sect. 3.1) we performed the parameter estimation on the *mozilla.core* dataset. First we tuned the $SimTH$ threshold and found the highest improvements of F_1,P,R for

Table 4. Phase 2 parameter tuning performances. We report the results of the comparison of best performance against MLP for some meaningful values of $SimTH$, when varying the two parameters $FreqTH$ and K in the set $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. Conf is the 0.95 confidence interval.

mozilla.core															
T	SimTH = 0.8			SimTH = 0.9			SimTH = 0.925			SimTH = 0.950			SimTH = 0.975		
	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R
1	−.008	−.055	.020	.001	−.003	.019	.006	.000	.017	.008	.000	.017	.008	.000	.016
2	−.004	−.043	.022	.003	−.011	.020	.006	−.001	.014	.003	−.002	.010	.002	−.002	.010
3	−.007	−.117	.035	.011	−.024	.034	.016	−.003	.035	.025	−.001	.029	.026	−.001	.028
4	−.014	−.062	.024	.004	−.017	.030	.012	−.004	.030	.015	−.002	.026	.015	−.002	.026
5	−.005	−.034	.013	−.001	−.005	.008	−.000	−.002	.007	.000	−.000	.005	.000	.000	.004
6	−.009	−.043	.007	−.002	−.011	.007	−.001	−.004	.006	−.000	−.001	.005	.000	−.000	.005
7	−.005	−.033	.005	−.001	−.014	.005	−.001	−.007	.003	−.000	−.002	.002	.000	−.001	.003
8	−.006	−.037	.008	−.001	−.017	.010	.001	−.008	.010	.002	−.003	.009	.003	−.001	.009
Avg	−.007	−.053	.017	.002	−.013	.017	.005	−.004	.015	.007	−.001	.013	.007	−.001	.013
Conf	.003	.023	.009	.004	.006	.009	.005	.002	.010	.008	.001	.008	.008	.001	.008

Table 5. Phase 2 localization performances $K = 10$, $FreqTH = 2$, $SimTH = 0.95$.

	mozilla						eclipse			dolphin			httpd			gcc		
	core			firefox			platform											
T	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R
1	.64	.98	.47	.59	.53	.69	.66	.66	.66	—	—	—	—	—	—	—	—	—
2	.58	.98	.41	.81	.85	.77	.94	.98	.91	—	—	—	.67	1.0	.50	.71	.91	.58
3	.51	.97	.35	.77	.98	.63	.70	.85	.59	—	—	—	1.0	1.0	1.0	.75	.95	.62
4	.67	.93	.54	.61	.99	.44	.57	.89	.42	.84	.89	.80	.87	1.0	.77	.13	.99	.07
5	.62	.94	.46	.64	.92	.49	.63	.72	.56	.87	.90	.84	.69	1.0	.54	.56	.49	.65
6	.61	.92	.45	.60	.97	.43	.91	.95	.88	.32	1.0	.19	.20	.71	.11	.26	.70	.16
7	.50	.89	.34	.53	.84	.38	.89	1.0	.80	.76	.93	.64	.25	1.0	.14	.93	.99	.88
8	.65	.89	.51	.49	.89	.34	1.0	1.0	1.0	.80	1.00	.66	1.0	1.0	1.0	.25	.95	.14
Avg	.60	.94	.44	.63	.87	.52	.79	.88	.73	.72	.94	.63	.67	.96	.58	.51	.85	.44
Conf	.05	.03	.06	.09	.12	.13	.14	.11	.17	.28	.07	.32	.31	.10	.34	.28	.17	.29

$SimTH \in \{0.950, 0.975\}$ see Table 4. Second we tuned $FreqTH$ and K and found the highest performances for $FreqTH = 2$ and $K = 10$ across the majority of time windows T. In Fig. 4 we report the diagrams of F_1,P and R for $T = 8$.

In Table 5 we report for each dataset portion and for each time window T the resulting performances after the second phase classification rearrangement. We observed that, across the datasets, the F_1 improvement is limited to a $+1\%$ and it is due to the increment of the recall performances.

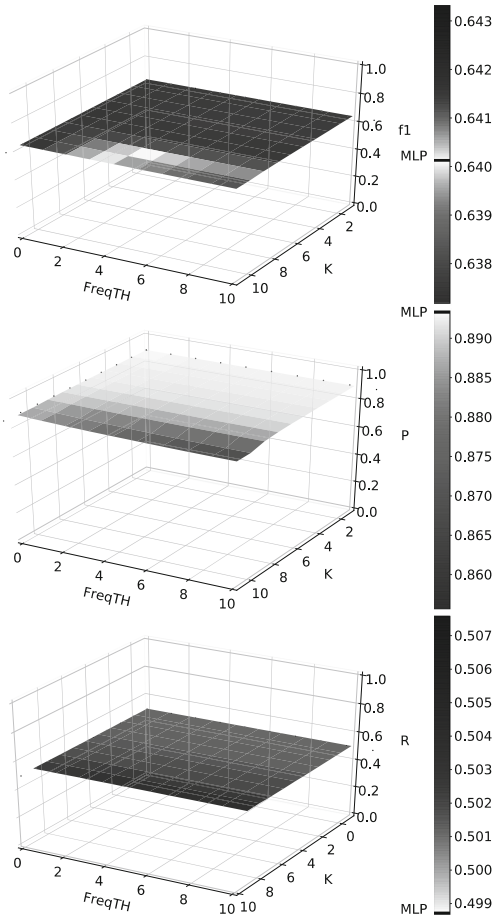

Fig. 4. From top to bottom F_1, P and R of Phase 2 approach with "mozilla.core", T = 8 $SimTH = 0.975$, valuing the two parameters $FreqTH$ and K in the set $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

4.5 Discussion

Our initial research questions (see Sect. 1) aimed at investigating the ability of bug reports, combined with observed distributional features, to be enough informative to learn models to predict the localization of fresh newly reported bugs.

To answer both questions, we proposed a two-phase approach (see Sect. 3) where, in contrast with existing deep-learning techniques and to cope with the scalability issue, bug reports are encoded with pre-trained vectors (i.e., Glove word embeddings).

From the results of our experiments (see Sect. 4) we can deduce the following:

RQ1: Bug report *summaries* are written in natural language, with different styles and can be from truly informative to ambiguous and of scarce textual content. In other words, in our experiments, due to the nature of bug report *summaries* and their arbitrary, we observed a wide variety of cases where our approach, is able to correctly predict the expected bug location (see the values of the FullyCoverd (FC) measure in Table 3) but also a wide range of cases where the system is not able to provide any answer (FullyUncovered (FU)). Moreover, the adoption of pre-trained word embeddings vectors enables the design of standard MLP networks which, in contrast to deep learning techniques, are able to benefit from both the availability of a compact input representation and the ability of applying standard back-propagation optimization algorithms. Overall, thanks to pre-trained word embeddings, we were able to implement scalable neural network classifiers with a training running time of few hours for our largest dataset (i.e., *mozilla.core*) on a single quad-core machine with 16 GB of RAM.

We also experimented to train our MLP networks with a combination of input vectors built on bug report *summaries* and bug report *descriptions*.[6] and observed a not significant improvement in recall but a substantial drop in precision.

RQ2: The simple distributional-based technique adopted for the second phase localization refinement is not able to significantly improve the results for the different datasets.

5 Related Work

Due to the high effort that localizing bugs in the source code of a software system usually requires, many approaches have been devised to support developers in this task and to automatize it.

Recently, as surveyed in [15], the state of the art approaches were based on a variety of information retrieval techniques devoted to localize a bug considering a bug report as a query and by "retrieving" the most relevant source files.

[6] also known as the comment at bug reporting time.

Latent Dirichlet Allocation-based document model [13] are used in [6,8,10] among others, to leverage topic models and to rank relevant source files.

In [1,2,11,16] among others, the most relevant source files are identified comparing the similarity between bug reports and between source files. Instead in [12] among others, the authors also combine information extracted from version history services.

To reduce the number of false positives in [3] relevant source file are provided only if "enough" evidence is provided by bug reports and version histories.

In [14] the authors localize bugs by leveraging domain knowledge through functional decomposition of source code files into methods, API descriptions of library components used in the code, the bug-fixing history, and the code change history.

Deep learning approaches have been recently investigated.

In [5], the authors propose a framework called Defect Prediction via Convolutional Neural Network (DP-CNN). Deep learning is applied to programs' Abstract Syntax Trees (ASTs) to encode source code tokens as numerical vectors trough word embedding. Then, the vectors are used to train a Convolutional Neural Network to automatically learn semantic and structural features of programs.

In [4] the authors presented a similar combination of deep learning and informational retrieval techniques. Revised Vector Space Model (rVSM) [3] are used to collect features for the textual similarity between bug reports and source files; instead, deep neural networks (DNN) are used to learn how to relate words in bug reports to code tokens and words in source files.

6 Conclusion and Future Work

We presented a two-phase bug localization approach based on multi-layer perceptrons networks and distributional features.

Phase 1 of our approach (see Sect. 3.1) uses MLP networks to learn a model from pre-trained word embedding representations of bug report *summaries* and predict bug locations. We observed in our experiment (see Sect. 4.3) that Phase 1 is a scalable method to compute partial, but precise, bug locations.

Phase 2 of our approach (see Sect. 3.2) uses distributional features to complement partial bug locations provided by Phase 1. Unfortunately, in our experiment (see Sect. 4.4) we did not observe a significant performance improvement.

Overall, we believe that Phase 1 methodology partial solutions can help the execution of more comprehensive approaches (such as those mentioned in Sect. 5), hence, coping with their scalability issues.

We are working on improving Phase 2 of our approach by investigating scalable solutions, such as those proposed in [5].

References

1. Davies, S., Roper, M., Wood, M.: Using bug report similarity to enhance bug localisation. In: 2012 19th Working Conference on Reverse Engineering, pp. 125–134, October 2012. https://doi.org/10.1109/WCRE.2012.22
2. Dilshener, T., Wermelinger, M., Yu, Y.: Locating bugs without looking back. In: Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, pp. 286–290. ACM, New York (2016)
3. Kim, D., Tao, Y., Kim, S., Zeller, A.: Where should we fix this bug? a two-phase recommendation model. IEEE Trans. Software Eng. **39**(11), 1597–1610 (2013). https://doi.org/10.1109/TSE.2013.24
4. Lam, A.N., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N.: Bug localization with combination of deep learning and information retrieval. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), pp. 218–229, May 2017. https://doi.org/10.1109/ICPC.2017.24
5. Li, J., He, P., Zhu, J., Lyu, M.R.: Software defect prediction via convolutional neural network. In: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 318–328, July 2017. https://doi.org/10.1109/QRS.2017.42
6. Lukins, S.K., Kraft, N.A., Etzkorn, L.H.: Bug localization using latent dirichlet allocation. Inf. Softw. Technol. **52**(9), 972–990 (2010). https://doi.org/10.1016/j.infsof.2010.04.002, http://www.sciencedirect.com/science/article/pii/S0950584910000650
7. Murphy-Hill, E., Zimmermann, T., Bird, C., Nagappan, N.: The design of bug fixes. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 332–341, May 2013. https://doi.org/10.1109/ICSE.2013.6606579
8. Nguyen, A.T., Nguyen, T.T., Al-Kofahi, J., Nguyen, H.V., Nguyen, T.N.: A topic-based approach for narrowing the search space of buggy files from a bug report. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pp. 263–272, November 2011. https://doi.org/10.1109/ASE.2011.6100062
9. Pennington, J., Socher, R., Manning, C.D.: Glove: global vectors for word representation. In: EMNLP, vol. 14, pp. 1532–1543 (2014)
10. Rao, S., Kak, A.: Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In: Proceedings of the 8th Working Conference on Mining Software Repositories, MSR 2011, pp. 43–52. ACM, New York (2011). https://doi.org/10.1145/1985441.1985451, https://doi.org/10.1145/1985441.1985451
11. Wang, S., Lo, D., Lawall, J.: Compositional vector space models for improved bug localization. In: 2014 IEEE International Conference on Software Maintenance and Evolution, pp. 171–180, September 2014. https://doi.org/10.1109/ICSME.2014.39
12. Wang, S., Lo, D.: Version history, similar report, and structure: putting them together for improved bug localization. In: Proceedings of the 22nd International Conference on Program Comprehension, ICPC 2014, pp. 53–63. ACM, New York (2014). https://doi.org/10.1145/2597008.2597148
13. Wei, X., Croft, W.B.: Lda-based document models for ad-hoc retrieval. In: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2006, pp. 178–185. ACM, New York (2006). https://doi.org/10.1145/1148170.1148204

14. Ye, X., Bunescu, R., Liu, C.: Learning to rank relevant files for bug reports using domain knowledge. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pp. 689–699. ACM, New York (2014). https://doi.org/10.1145/2635868.2635874
15. Zhang, T., Jiang, H., Luo, X., Chan, A.T.: A literature review of research in bug resolution: tasks, challenges and future directions. Comput. J. **59**(5), 741–773 (2016). https://doi.org/10.1093/comjnl/bxv114
16. Zhou, J., Zhang, H., Lo, D.: Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 14–24, June 2012. https://doi.org/10.1109/ICSE.2012.6227210