Contents lists available at ScienceDirect

The Journal of Systems & Software

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

In practice

Automated defect prioritization based on defects resolved at various project periods

Mustafa Gökçeoğlu [a], Hasan Sözer [b,*]

[a] Vestel Electronics, Manisa, Turkey
[b] Ozyegin University, İstanbul, Turkey

ARTICLE INFO

Article history:
Received 8 December 2020
Received in revised form 20 March 2021
Accepted 4 May 2021
Available online 12 May 2021

Keywords:
Software maintenance
Defect prioritization
Process automation
Machine learning
Industrial case study

ABSTRACT

Defect prioritization is mainly a manual and error-prone task in the current state-of-the-practice. We evaluated the effectiveness of an automated approach that employs supervised machine learning. We used two alternative techniques, namely a Naive Bayes classifier and a Long Short-Term Memory model. We performed an industrial case study with a real project from the consumer electronics domain. We compiled more than 15,000 issues collected over 3 years. We could reach an accuracy level up to 79.36% and we had 3 observations. First, Long Short-Term Memory model has a better accuracy when compared with a Naive Bayes classifier. Second, structured features lead to better accuracy compared to textual descriptions. Third, accuracy is not improved by considering increasingly earlier defects as part of the training data. Increasing the size of the training data even decreases the accuracy compared to the results, when we use data only regarding the recently resolved defects.

1. Introduction

Thousands of defects are reported for large-scale systems every year. For instance, the number of defects reported for Eclipse, Mozilla and Gnome projects over a period of 3 months were recorded as 2764, 6976 and 3263, respectively (Lamkanfi et al., 2010). It is necessary to systematically manage and trace these defects to reduce costs and improve system reliability (Strate and Laplante, 2013). There exist tools like JIRA[1] to support this process. Despite the tool support, many tasks still have to be performed manually in state-of-the-practice. One of these tasks is the prioritization of defects. Usually, a priority is assigned to each defect in ordinal scale, determining how soon it needs to be fixed. Manual prioritization of defects is not only effort consuming, but also error-prone. A study based on five open-source projects revealed that 33.8% of all bug reports were misclassified (Herzig et al., 2013). This is a critical issue especially in the consumer electronics domain, where resources are limited and severities of defects are subject to a high variation (Dirim and Sozer, 2020). Some of the defects are highly critical and they have to be fixed immediately. Some other defects might not be even bothering. More often than not, consumer electronics products are shipped with known software faults (Zoeteweij et al., 2007). In fact, some defects might not be even pointing at actual bugs (Herzig et al., 2013) and as such they might not require any fix at all. Hence, it is very important to prioritize the reported defects quickly and correctly.

There have been several approaches proposed for automating the defect prioritization task (Menzies and Marcus, 2008; Lamkanfi et al., 2010; Zhou et al., 2016). These approaches are mainly based on supervised machine learning techniques, where a model is trained based on features of a set of defects reported in the past and their actual priorities. Then, the priority of a defect can be predicted based on this model. So far, researchers have experimented with alternative machine learning techniques for automated bug prioritization. However, they have not investigated the impact of the time frame of the training data on the accuracy of the prioritization. The development and maintenance of software systems take place in a long time frame, spanning at least several years. Types and priorities of defects that are reported at the beginning of a project can be very different than those reported later on. A longer time frame can be considered as an advantage since more data might be available for training a machine learning model. However, one needs to determine how far back in the project history should be considered for training. It might be disruptive to involve features of all the earlier defects as part of the training data. We focus on this perspective of the problem, which has been neglected in the literature. Furthermore, we present an industrial case study unlike previous studies, which all employ open source projects for evaluation.

We employed two machine learning algorithms. First, we used the Naive Bayes algorithm, which is pointed out as the

* Corresponding author.
E-mail address: hasan.sozer@ozyegin.edu.tr (H. Sözer).

[1] https://www.atlassian.com/software/jira

https://doi.org/10.1016/j.jss.2021.110993
0164-1212/© 2021 Elsevier Inc. All rights reserved.

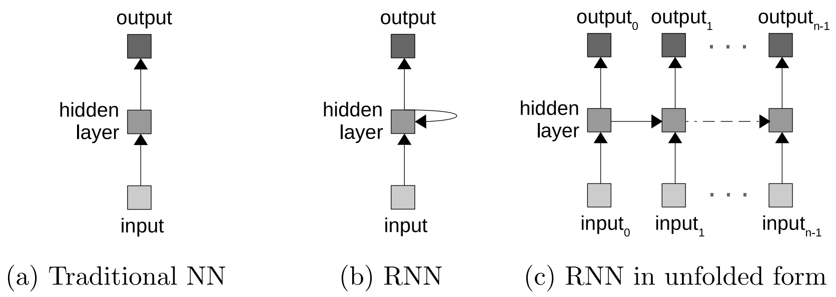

Fig. 1. Traditional Neural Network (NN) vs. Recurrent NN (RNN).

mostly used and the most successful one for our classification purpose (Lamkanfi et al., 2010; Zhou et al., 2016). Second, we employed a *Long Short-Term Memory (LSTM)* (Hochreiter and Schmidhuber, 1997) model. LSTM is a type of neural network model that is capable of learning relations among a sequence of inputs. Hence, not only it learns from the training data, but also it learns what to forget, what to remember, and for how long. Therefore, an LSTM-based approach is expected to be effective for handling defect characteristics that vary over a long period of software development life-cycle. LSTM has been recently used in a study (Ramay et al., 2019) for bug classification. However, that study employs only textual descriptions of bugs collected from open source projects. In our study, we used more than 15,000 issues collected over 3 years for a real project from the consumer electronics domain. We trained our classifiers with both unstructured, textual descriptions of defects and structured features collected from defect reports. We could reach an accuracy level up to 79.36%. We observed that LSTM has a better accuracy when compared with a Naive Bayes classifier. We also observed that structured features lead to better accuracy compared to textual descriptions. Accuracy mostly decreases when structural features are combined with an additional feature derived from textual descriptions. Finally, we discovered that accuracy is not improved by considering increasingly earlier defects as part of the training data. Increasing the size of the training data even decreases the accuracy. We obtained better results when we use data only regarding the recent defects reported in the last 6 months only.

The remainder of this paper is organized as follows. In the following section, we provide background information on LSTM. In Section 3, we explain our approach for automated bug prioritization. In Section 4, we introduce the experimental setup for our industrial case study and we present the obtained results. We discuss these results and lessons learned in Section 5. We summarize related studies in Section 6. Finally, we conclude the paper in Section 7.

2. Background

In this section, we provide background information about *Long Short-Term Memory (LSTM)* networks. LSTM (Hochreiter and Schmidhuber, 1997) is a type of *Recurrent Neural Network (RNN)*. Fig. 1(a) depict a traditional feed-forward neural network (Hagan et al., 1995), whereas Figs. 1(b) illustrate RNN. RNN is distinguished by a feedback loop from the hidden layer to itself. This loop enables RNN to keep an internal state, while consuming a sequence of inputs. Fig. 1(c) depicts RNN in unfolded form. RNN neurons are able to transfer activation states triggered by inputs that are consumed earlier in the sequence. Although RNNs are effective for learning short-term relations among inputs, they fall short for learning relations among inputs in a sequence that are far apart from each other. In other words, there is no guarantee for an RNN to learn correlations of fed data that are relatively distant in time.

LSTM is a special type of RNN that is capable of learning both *short term* and *long term* relations among a sequence of inputs. This capability is achieved by employing a sophisticated transfer function that propagates state information (Greff et al., 2017). This function is composed of multiple sub-components, called gates. Each of these gates involves an independent neural network of its own, and it controls the state by filtering, adding, or updating information during the state transfer. Altogether, based on the input training data, an LSTM network learns what to forget, what to remember, and for how long. In other words, the default behavior of an LSTM network is to remember a context until it is automatically decided that it is no longer of value.

The difference between a traditional RNN module and a typical transfer module in LSTM is depicted in Fig. 2. An RNN module combines the previous state information vector (i.e., recurrent) with the input vector, which is fed to an activation function. This function can be typically a step function, a non-linear function such as sigmoid (Hagan et al., 1995), or rectified linear units (ReLU) (Nair and Hinton, 2010). The output of the function is used both as the output and the state information to be propagated. On the other hand, an LSTM module employs 4 network layers as activation functions. These network layers are also trained along with the whole network for learning which part of the input should be passed through or filtered out (forget gate), which part should be added (input gate) to which part (block gate) of the current state, and which part of this state should be reflected to the output (output gate).

There exist several practical applications of LSTM mainly concentrated on language modeling (Gers and Schmidhuber, 2001) such as Google Translate (Wu et al., 2016). It can also be considered as a good candidate for prediction tasks concerning software defects since successful software systems are under maintenance for many years. Defect characteristics can be subject to high variation over these years. An LSTM-based approach can handle this variation by automatically learning the context of various project periods. In fact, LSTM has been recently used for predicting severities of bug reports based on their textual descriptions (Ramay et al., 2019). We employed structural features such as *assignee* and *bug category* for the prediction task. These features do not constitute sequential input like textual descriptions written with natural languages. Nevertheless, we employed LSTM due to our observation that attributes of bugs and their priorities change in time. In fact, they do not only change in time but also depend on the attributes and priorities of previous bugs in time. Therefore, we treated the dataset as sequential. The data collected per bug instance is not sequential; however, we consider data regarding a number of bug instances in time as sequential data. The LSTM network is supposed to remember some of the previous bugs and forget some others. In the meantime, it is also supposed to learn automatically what to forget, what to remember, and for how long. We also used the Naive Bayes algorithm, which has been pointed out to be successful for bug prioritization (Lamkanfi et al., 2010; Zhou et al., 2016). We trained our models with

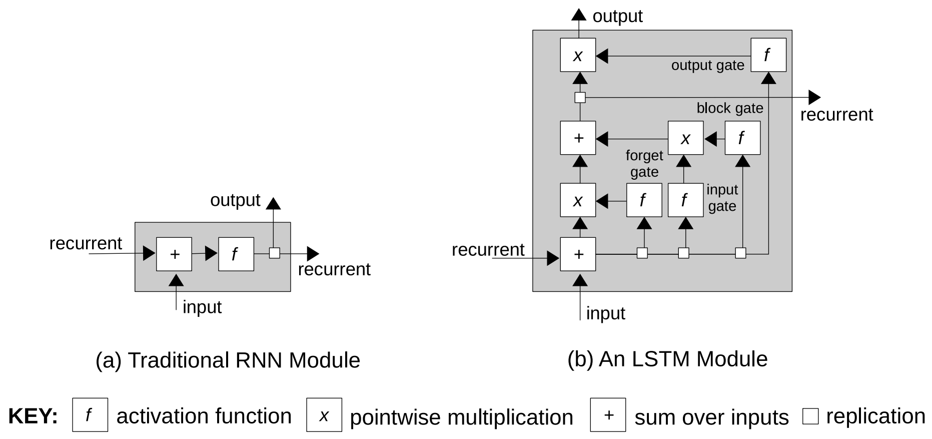

Fig. 2. Traditional RNN vs. LSTM network.

both unstructured, textual descriptions of defects and structured features collected from defect reports. In the following section, we describe our approach and design decisions.

3. The overall approach

Fig. 3 depicts our overall approach. We experimented with 4 models for predicting the actual priority of reported defects. We extracted both structural attributes (e.g., *bug type*, *category*, *assignee*) and textual descriptions from defect reports. A Naive Bayes classifier is trained to perform the prediction task only with structured attributes. A second Naive Bayes classifier is trained to perform the prediction task only with textual descriptions. A third model employs two cascading Naive Bayes classifiers. Hereby, prediction results obtained from the first classifier based on textual descriptions are used as a feature together with structured attributes for the second classifier. Finally, an LSTM model is used for performing the prediction task. This model is based only on structural attributes. Our results and analysis showed that textual descriptions were not effective in achieving accurate predictions as explained in the following section. Therefore, they were not involved in our study with the LSTM model.

Fig. 4 depicts the approach we employed for combining structural features with an additional feature derived from predictions based on textual descriptions of previous periods. The overall project timeline is divided into a set of periods as described in Section 4.2. The training data is also divided according to the period in which the data is created. The steps taken for a prediction task regarding *Period n* are enumerated from 1 to 5 and they are listed below:

1. The first Naive Bayes classifier is trained with textual descriptions of defects in *Period n-2*;
2. Priorities of defects in *Period n-1* are predicted with the first classifier based on their textual descriptions and prediction results are combined with structural attributes of the corresponding defects from *Period n-1*;
3. The dataset obtained in *Step 2* is used for training the second classifier;
4. Priorities of defects in *Period n* are predicted with the first classifier based on their textual descriptions and prediction results are combined with structural attributes of the corresponding defects from *Period n*;
5. The dataset obtained in *Step 4* is used for predicting the priorities with the second classifier.

Figs. 3 and 4 depict only a single instance of the process. We repeated this process for each period multiple times. We used a different training set in each repetition. In the first test, the training set included data only from the previous period. The second training was based on two previous periods. The last test involved all the data from the beginning of the project until the corresponding period. The process depicted in Fig. 4 employs two cascading Naive Bayes classifiers. Therefore, it requires data from at least two previous periods for the prediction task concerning a particular period.

Table 1
LSTM and Network Configuration Parameters.

Parameter name	Value
Number of epochs	5
Attribute normalization	None
Batch size	100
Activation function	Soft sign
Gate activation functions	Sigmoid
Network configuration updater and bias updater	Adam, SGD

LSTM models are subject to several parameters to be tuned. Table 1 lists the parameter values that are used in our models. These values were determined to achieve high accuracy by performing a grid search over possible combinations of a set of candidate values. The number of epochs was set as 2 and increased one by one until the accuracy level converged to a level without further significant changes. We did not use any normalization. We considered 20, 50, 100, 120, 150 and 200 as candidate values for the batch size. We tested ReLU, Soft Sign, Softmax and Sigmoid as activation functions. We tested Adam, Adadelta, SGD, Adagrad as optimization algorithms. We omitted additional layers like dropout layer since they did not have a significant impact on accuracy.

In the following, we explain details regarding our experimental evaluation performed in the context of an industrial case study.

4. Industrial case study

In this section, we introduce our case study on Smart TV projects being developed and maintained at Vestel,[2] which is one of the largest consumer electronics companies in Europe. In addition to the products of its own, Vestel is producing Smart TV systems for 157 different brands from 145 different countries (Gebizli and Sozer, 2016).

In the following subsection, we first present and motivate our research questions. Then, we explain the experimental setup and our dataset.

2 http://www.vestel.com.tr

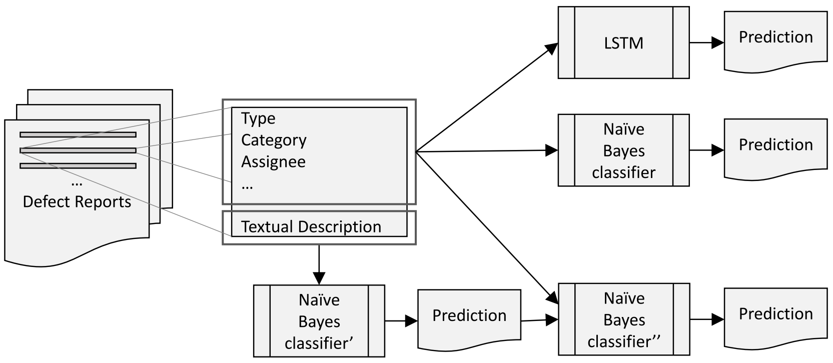

Fig. 3. The overall approach.

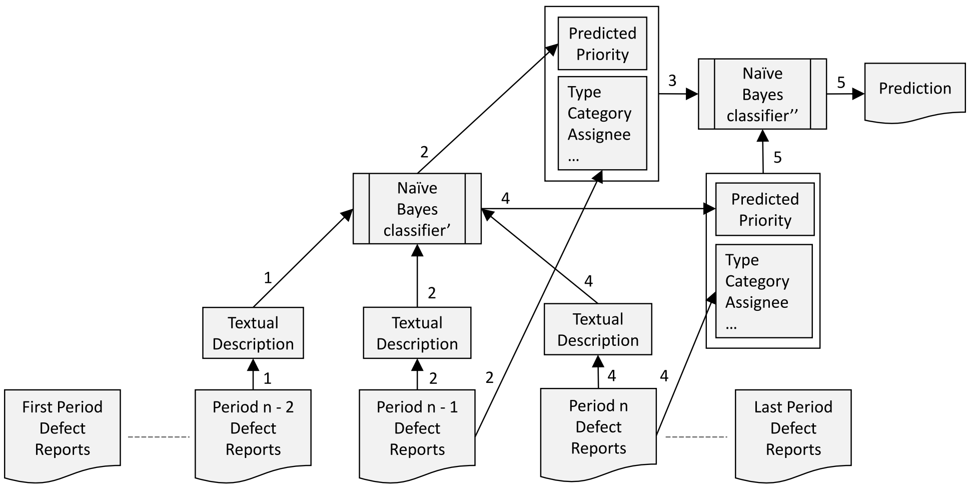

Fig. 4. Combining structural features with an additional feature derived from predictions based on textual descriptions of previous periods.

4.1. Research questions

In our case study, we aim at answering the following research questions:

- **RQ1**: How does the accuracy of prioritization affected by using defect attributes, textual descriptions, or both as training data?
- **RQ2**: How does the accuracy of prioritization affected by considering increasingly earlier defects as part of the training data?
- **RQ3**: How does the accuracy of prioritization affected by employing a deep recurrent neural network instead of a Naive Bayes classifier?

Our first question is regarding the accuracy of prioritization, when it relies only on structured defect attributes, only on textual descriptions and both of these at the same time. We can train our model with data in various sizes depending on how far back in the project history is considered for training. The second question is defined for investigating the impact of this variation. In particular, we are interested in the accuracy of alternatives listed in *RQ1* at various project periods. Projects in the consumer electronics domain are typically carried in a sequence of periods. For instance, the first period is usually dedicated to establish a solid hardware–software interface. This proceeds with the development and maintenance of several software layers, on top of which new features are to be integrated. Types of bugs that appear in each period are naturally related to the type of maintenance and development tasks within that period. For instance, one might expect more hardware-related issues in the first period. Types of defects in this period might be very different than those reported at later periods. Hence, our hypothesis is that the accuracy can be decreased if we employ increasingly earlier defect reports for training. Given this hypothesis is true, we expect that a deep recurrent neural network, i.e., LSTM would be effective for defect prioritization. The last research question is defined for evaluating the accuracy achieved with this approach.

4.2. Data collection and experimental setup

We obtained all the defect reports collected over 3 years regarding Smart TV software. These reports are recorded as JIRA bug issues by test engineers. We filtered them based on their creation date. Duplicate bugs were already removed from the system before our study. We further removed incomplete bug reports, which do not comprise information regarding attributes (e.g., *assignee*, *bug category*) that are used in our approach. As a result, we obtained a dataset that comprises 15,318 defect reports[3]. There are more than 250 attributes that can be specified as part of each bug report in addition to a textual description of the bug. However, only some of these are used and considered to be important by test engineers. On the other hand, some attributes like *creation date* are automatically set by the tool. In our case study, we used attributes that are considered as important, as such specified (almost) for every bug. We used only those attributes that would be available at the time of creating

[3] Due to confidentiality issues, we cannot share the dataset publicly.

Table 2
The collected set of bug attributes that are used as features.

#	Name	Description
1	Priority	Severity of the failure
2	**Approval type**	Severity of bug for production
3	Assignee	One of the software engineering teams
4	Bug category	Type of bug
5	Component	Related software component
6	Labels	Related project/activity
7	Test type	Automated/Manual
8	Summary	Textual description

Table 3
Approval types that determine the priority of a bug.

Approval type	Description
*S*1	Very urgent; the bug can stop the production
*S*2	Urgent
*S*3	Not urgent but needs to be fixed
*S*4	No need to be fixed anytime soon

the issue. There are for instance attributes like *resolution type* and *date of resolution*, which become available only after the bug is fixed. These attributes can directly endorse a particular priority and they are not known at the time of creation of a bug. Therefore, we excluded such attributes from our dataset.

We used bug attributes that are enumerated with their names and descriptions in Table 2. Hereby, the attribute that we want to predict is the second attribute named *Approval Type*. This attribute shows the severity of a bug and determines its priority. There are 4 severity levels defined as listed in Table 3.

The first attribute listed in Table 2, *Priority* is assigned by test engineers. Both *Approval Type* and *Priority* indicate the importance of a bug from different perspectives and they are commonly not consistent in practice. *Priority* refers to the severity of the bug in terms of the failure that is causes. For instance, a bug in a newly introduced feature would be considered to be highly severe if it leads to a system crash. Therefore, the priority of such a bug would be high although the corresponding feature is not going to be integrated to the product anytime soon. Prioritization of bug fixes determines the order/schedule in which fixes has to be applied as long as the resources are available. *Priority* (failure severity) is obviously a factor that is relevant for this decision. However, it is not the only factor. There are other factors steered by business decisions such as where and when the corresponding feature is integrated, the importance of the feature (for end users) and the production schedule (any bug that can suspend the production usually has the upmost priority). *Approval Type* represents the real priority, which takes bug priority as well as all the other factors into account.

Each detected bug is assigned to a software engineering team for investigation and resolution. The *Assignee* attribute specifies this team. *Bug category* specifies the type of bug. Categories include *functional*, *performance* and *usability* among others. They identify whether the bug is affecting the functionality or one of the quality attributes of the system. *Component* specifies the software component (e.g., *GUI*) related with the bug. Fig. 5 shows the values of this attribute for the bugs in our dataset over time. The x-axis enumerates the set of 17 major components and the y-axis represents time. The depicted time series data clearly shows the variance of a bug attribute throughout the software development life-cycle. For instance, bugs associated with the 17th component are reported only in the late periods of the project. *Labels* provide a categorization that is defined and used within the company. Hereby, categories mainly represent various projects and activities (e.g., *Certification*) undertaken for Smart TV software development. They are associated with tasks

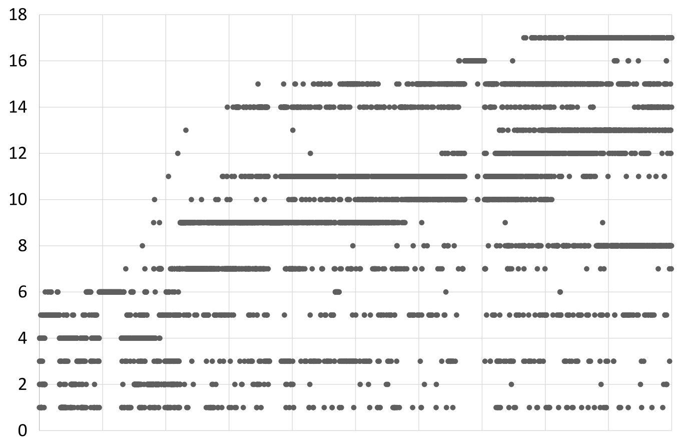

Fig. 5. *Component* attributes (See Table Table 2) of reported bugs over time.

Table 4
Bugs that have to be re-prioritized manually.

Period	# of re-prioritized bugs	# of bugs	%
P1	191	665	28.72
P2	1995	2207	90.39
P3	1710	2672	63.99
P4	3714	4424	83.95
P5	2893	3305	87.53
P6	1461	2045	71.44

as the scope of detected bugs. *Test type* specifies the type of test that revealed the bug. This attribute can be set as either one of *automated* and *manual*. Finally, *Summary* includes a textual description of the bug. Unlike all the previous attributes, *Summary* is not structured. It is provided as a free-form text by a test engineer to describe how the failure appears.

In current state-of-the-practice, the *Approval Type* attribute is set manually by test engineers. However, even the manually assigned severity category is usually not accurate. The initially assigned approval type has to be changed multiple times throughout the software development lifecycle. Table 4 lists for each period, the number and ratio of bugs, for which the initially assigned approval type had to be modified later at least once. We can observe that the ratios are very high for all the periods. These results confirm our argument that test engineers are not able to estimate the actual priority of a bug accurately. The actual priority (i.e., approval type) depends on many factors including business constraints and the impact of the bug on the production process.

In our study, our goal is to predict *Approval Type* that is stabilized towards the end of the project. We would like to predict this at the beginning so that changes (re-prioritizations) can be minimized. We considered the stabilized version of the attribute as the ground truth because this version reflects the actual priority given for bugs in bug-fixing schedule. *Priority* is assigned by the tester who observes the failure and the attribute identifies the severity of this failure. On the other hand, *Approval Type* is assigned by upper management by considering the priority of the bug as well as its impact on the production schedule. This process can be delayed based on the amount of bugs to be analyzed and the workload of those who are making these assignments for multiple projects at the same time. As a result, some important bugs can be missed and cause a delay in production. Therefore, it is valuable to automatically predict *Approval Type* early and accurately. We use (supervised) machine learning for this purpose. In particular, we use a Naive Bayes classifier as many previous studies did (Lamkanfi et al., 2010; Zhou et al., 2016) as well as an

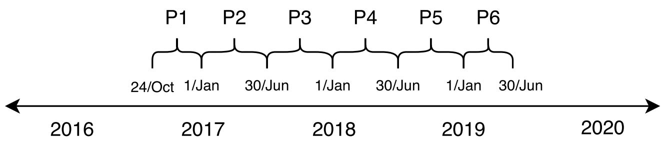

Fig. 6. Periods over which the data collection is performed.

Table 5
The duration and the number of bugs associated with each period.

Period	Duration	# of bugs per *Approval type*				
name	(months)	S1	S2	S3	S4	Total
P1	3	71	63	7	524	665
P2	6	1444	311	100	352	2207
P3	6	1351	66	149	1106	2672
P4	6	2537	130	356	1401	4424
P5	6	2444	25	191	645	3305
P6	6	1139	197	170	539	2045

LSTM model. We experiment with various datasets for training to be able to answer our research questions.

To answer *RQ1*, we trained our model and performed prediction only with structured attributes first. Then, we performed classification only by using the *Summary* attribute. Finally, we employed two cascading classifiers to combine the two types of attributes as explained in Section 3. We employed textblob[4] for text classification. We used the Naive Bayes classifier and LSTM model implemeted by the Weka[5] toolset, while working with structured attributes.

To answer *RQ2*, we repeated our tests described above by varying the data used for training. We splitted our dataset into 6 successive periods. Fig. 6 depicts these enumerated periods from *P1* to *P6*.

The way that the dataset is divided into periods was not based on our decision. Each period is associated with a certain set of tasks. There are half-year targets and annual targets for completing these tasks. That is why these periods are 6 months long in general as depicted in Fig. 6. The overall timeline including the periods and milestones are determined by the project management. Table 5 lists the duration and the number of bugs associated with each period.

We performed the prediction of approval type for bugs in each period separately. We first performed this prediction task after training the model with data from just the previous period. Then we used data collected within the two previous periods. We repeated this process until the last test, where we use all the data from the beginning of the project until the corresponding period for training. Our approach that employs two cascading classifiers cannot be applied for the first test since it requires data from at least two past periods for the prediction task concerning the current period.

To answer *RQ3*, we repeated our tests described above with an LSTM model. However, we did not use the *Summary* attribute for this approach. Our results and post-mortem analysis regarding previous experiments with Naive Bayes classifiers showed that textual descriptions do not constitute high-quality data for training.

We evaluated the performance of our prioritization approach based on the accuracy of the prediction of the *Approval Type* setting for bugs. Recall that the initially assigned approval type usually changes multiple times throughout the software development lifecycle. However, the setting stabilizes towards the end of a period. Therefore, we considered the final setting as the ground truth, i.e., the correct approval type.

In addition to the evaluation of the overall prediction performance, we performed an additional evaluation by paying special attention for bugs that have *S1* as the approval type. These are the most critical bugs. Therefore, we measured the classification accuracy for these bugs separately. Overall, we used the following metrics (Powers, 2011), where *TN*, *TP*, *FN* and *FP* represent the number of *true negatives*, *true positives*, *false negatives* and *false positives*, respectively:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

$$F = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{4}$$

Hereby, *TN*, *TP*, *FN* and *FP* cases are identified as follows:

- *TN*: Approval type is not predicted as *S1*, and the defect has indeed a different approval type.
- *TP*: Approval type is predicted as *S1*, and approval type of the defect is indeed *S1*.
- *FN*: Approval type is not predicted as *S1*, but approval type of the defect is *S1*.
- *FP*: Approval type is predicted as *S1*, but the defect has a different approval type.

Accuracy (Eq. (1)) is a metric that measures the ratio of correct predictions among all the predictions. It basically evaluates the correctness of predictions. However, it can be a misleading metric for imbalanced datasets. For instance, the approval type for 2,444 out of 3,305 bug reports in *P5* is set as *S1* (See Table 5). As a result, a very simple prediction model that classifies all the bugs as *S1* (i.e., null accuracy) can already reach 0.74 accuracy score for that period. Therefore, we employed additional metrics. *Precision* (Eq. (2)) evaluates to what extent bugs classified as *S1* are indeed among the most critical ones. Low precision leads to waste of resources. *Recall* (Eq. (3)) evaluates to what extent the most critical bugs are captured by classifying them as *S1*. Low recall means that we are overlooking critical bugs. One can achieve a very high precision by being very conservative in classifying bugs as *S1*. However, this would lead to a low recall. On the other hand, one can achieve 100% recall by classifying all the bugs as *S1*. None of the critical bugs would be missed by this way. However, it would lead to low precision. *F-Measure* (Eq. (4)) balances these two measures. We present the results in the following section.

4.3. Results

Table 6 lists the overall results regarding predictions that are performed only with structured attributes of bugs. Hereby, the first column enumerates the performed tests. The second and third columns list the period(s) of bugs that are used for training and testing the prediction model, respectively. The last two columns list prediction accuracies. Results obtained with Naive Bayes and LSTM models are compared and higher accuracy values are highlighted with bold fonts. The overall highest accuracy obtained regarding each period is underlined.

The accuracy is very low for *P2*. This can be expected due to the limited training data in *P1* that comprise 665 bugs in total. Note that there are 2,207 bugs in *P2*. Therefore, we consider the first test as an outlier. Nevertheless, we present the results for the sake of completeness. We can observe from the overall results

[4] https://textblob.readthedocs.io/en/dev/
[5] https://www.cs.waikato.ac.nz/ml/weka/

Table 6
Accuracy of predictions that are performed only with structured attributes of bugs.

Test #	Training data	Test data	Naive Bayes accuracy	LSTM accuracy
1	P1	P2	**0.333**	0.159
2	P2	P3	0.540	**0.601**
3	P1,P2	P3	0.552	**0.558**
4	P3	P4	0.5267	**0.581**
5	P2,P3	P4	0.516	**0.578**
6	P1,P2,P3	P4	0.514	**0.529**
7	P4	P5	0.756	**0.794**
8	P3,P4	P5	0.731	**0.775**
9	P2,P3,P4	P5	0.687	**0.754**
10	P1,P2,P3,P4	P5	0.679	**0.713**
11	P5	P6	**0.664**	0.660
12	P4,P5	P6	0.655	**0.672**
13	P3,P4,P5	P6	0.632	**0.667**
14	P2,P3,P4,P5	P6	0.637	**0.665**
15	P1,P2,P3,P4,P5	P6	0.633	**0.662**

that LSTM has a better accuracy when compared with a Naive Bayes classifier. The highest accuracy is obtained in test 7 with LSTM as 79.36%. In this test, approval types of bugs in *P5* are predicted based on a model trained with data regarding bugs in the previous period, *P4*. In tests 8, 9 and 10, the same test is performed by extending the training data from increasingly earlier periods. We observe that the accuracy decreases as we incorporate data from earlier periods in training. Same observations can be consistently made when we look at the prediction accuracies for *P3* in tests 2 and 3 as well as for *P4* in tests 4, 5 and 6. Results of test 11 for *P6* constitute the only exceptions for these observations. Naive Bayes classifier outperforms LSTM for this test. The overall best result for *P6* is again obtained with LSTM, but with test 12, where the *P4* dataset is used together with the *P5* dataset for training. We investigated the reasons for this and found out the highly imbalanced nature of the dataset regarding *P5* as the main reason. Only 25 bugs out of 3,305 in this dataset are associated with *S2* as the approval type. (See Table 5). The lack of enough instances in the training set leads to incorrect classifications, especially for deep learning models such as LSTM. The accuracy of test 11 was mainly reduced due to misclassification of bugs with *S2* as the approval type. Therefore, accuracy increases in test 12 when *P4* is involved. However, accuracy decreases in tests 13 to 15 as *P3*, *P2* and *P1* are also involved.

Table 7 lists the overall results regarding predictions that are performed only with textual descriptions of bugs. The table has the same structure as Table 6 except the last column. Predictions based on the *Summary* attribute were performed only with a Naive Bayes classifier. An LSTM model was not employed for this case since the accuracy of predictions are much lower as we can see in Table 7. The highest accuracy is obtained in test 9 with 60.66%. We examined many samples from the dataset to investigate the reasons for low accuracy values. It turns out that bug descriptions were provided in an inconsistent manner. Some of them were describing test steps for reproducing the bug, whereas some of them were describing the failure only. There was high variance regarding the level of technical detail in descriptions. There were inconsistencies even in terms of the language used. Some descriptions were in English, whereas some of them were in Turkish. As a result, we concluded that unstructured and inconsistent bug descriptions constituted a low-quality training data for the prediction model.

Table 8 lists the overall results regarding predictions when structured attributes of bugs are used together with prediction results obtained based on textual descriptions. The table has the same structure as Table 7. Hereby, some of the tests are *Not Applicable (NA)* since the use of two cascading classifiers requires at least two of the previous periods for training. The results are slightly better for tests 3 and 10, but worse in all the other tests when we compare them with the results obtained with only structured bug attributes (See Table 6). It is also not possible to observe a consistent trend and derive conclusions based on these results.

Table 7
Accuracy of predictions that are performed by a Naive Bayes classifier only with textual descriptions of bugs.

Test #	Training Data	Test Data	Accuracy
1	P1	P2	**0.167**
2	P2	P3	0.507
3	P1,P2	P3	**0.525**
4	P3	P4	**0.435**
5	P2,P3	P4	0.420
6	P1,P2,P3	P4	0.402
7	P4	P5	0.575
8	P3,P4	P5	0.562
9	P2,P3,P4	P5	**0.607**
10	P1,P2,P3,P4	P5	0.583
11	P5	P6	**0.558**
12	P4,P5	P6	0.555
13	P3,P4,P5	P6	0.533
14	P2,P3,P4,P5	P6	0.530
15	P1,P2,P3,P4,P5	P6	0.529

Table 8
Accuracy of predictions when structured attributes of bugs are used together with prediction results obtained based on textual descriptions.

Test #	Training Data	Test Data	Accuracy
1	P1	P2	NA
2	P2	P3	NA
3	P1,P2	P3	**0.562**
4	P3	P4	NA
5	P2,P3	P4	**0.531**
6	P1,P2,P3	P4	0.508
7	P4	P5	NA
8	P3,P4	P5	**0.723**
9	P2,P3,P4	P5	0.682
10	P1,P2,P3,P4	P5	0.692
11	P5	P6	NA
12	P4,P5	P6	0.604
13	P3,P4,P5	P6	0.578
14	P2,P3,P4,P5	P6	0.589
15	P1,P2,P3,P4,P5	P6	**0.613**

Table 9 lists prediction results for bugs that have *S1* as the approval type when the predictions are performed only with structured attributes of bugs. These bugs constitute the majority of the dataset and they represent the most important bugs. Prediction performance is measured with precision, recall and f-measure metrics. We observe that LSTM outperforms the Naive Bayes classifier in terms of recall and f-measure. The highest recall and f-measure are recorded as 0.97 and 0.87 for tests 11 and 7, respectively. We also observe that performance decreases as we incorporate data from earlier periods in training. However, this is not the case for precision, where the Naive Bayes classifier consistently outperforms LSTM. Investigation of classification results revealed that the LSTM model exploits the imbalance in the dataset. It learns that the majority of bugs have *S1* as the approval type. As a result, some of the bugs with other approval types are classified as *S1*, which in turn decreases the precision. However, the f-measure and the overall accuracy based on the whole dataset are still higher for LSTM since the ratio of

Table 9
Prediction results for bugs that have *S1* as the approval type when the predictions are performed only with structured attributes of bugs.

Test #	Naive Bayes			LSTM		
	Precision	Recall	F-Measure	Precision	Recalll	F-Measure
1	0.949	0.247	0.392	0.159	1	0.275
2	0.583	**0.775**	**0.665**	0.581	**0.946**	**0.72**
3	**0.626**	0.641	0.633	**0.592**	0.666	0.627
4	0.695	0.553	0.616	**0.699**	**0.740**	**0.719**
5	0.691	**0.559**	**0.618**	0.666	0.709	0.687
6	**0.727**	0.509	0.599	0.662	0.563	0.608
7	0.892	**0.839**	**0.865**	0.807	**0.959**	**0.876**
8	0.908	0.792	0.846	0.801	0.938	0.864
9	0.924	0.728	0.814	**0.821**	0.877	0.848
10	**0.929**	0.717	0.809	0.812	0.819	0.816
11	0.692	**0.907**	**0.785**	0.676	**0.971**	**0.797**
12	0.697	0.839	0.761	**0.695**	0.922	0.793
13	0.699	0.783	0.739	0.684	0.937	0.791
14	**0.732**	0.745	0.738	0.625	0.937	0.75
15	0.723	0.733	0.728	0.629	0.932	0.751

bugs with approval type *S1* is high. In the following, we provide answers for our research questions based on the results obtained. We conclude the section by discussing validity threats to our evaluation.

4.3.1. Using structural attributes, textual descriptions, or both (RQ1)

Overall, we observed that the prediction accuracy can be increased up to 79.36%. Despite the fact that we used an industrial dataset, our results do not diverge significantly from previous studies, where prediction accuracies generally lie between 60% and 80% although minimum and maximum accuracy levels have been reported as 30% and 90%, respectively for various other datasets (Lamkanfi et al., 2010; Xuan et al., 2012; Zhou et al., 2014). Accuracy is lower when only textual descriptions of bugs are used for prediction. Predictions based on structured attributes of bugs lead to better accuracy. Accuracy even decreases when structured attributes are used together with the prediction result based on textual descriptions. Hence, our results diverge from those of previous studies, which report highly accurate results for the prediction of bug priorities based on textual descriptions (Lamkanfi et al., 2010; Xuan et al., 2012; Ramay et al., 2019). Possible reasons for this difference are discussed further in Section 5.

4.3.2. Considering increasingly earlier defects as part of the training data (RQ2)

Our expectations regarding *RQ2* were confirmed by the results. Accuracy is not necessarily improved by training the prediction model based on increasingly earlier defects. Increasing the size of the training data even decreases the accuracy compared to the results, when we use data only regarding the recent defects reported in the last 6 months. This observation is subject to only a few exceptional cases. This result was expected because types of defects and their properties tend to vary throughout the lifetime of a project.

4.3.3. Performance of LSTM and naive Bayes classifier (RQ3)

Results showed that LSTM has a better overall accuracy when compared with a Naive Bayes classifier. However, it performs worse in terms of precision when applied on imbalanced datasets. Investigation of results showed that LSTM is more inclined to exploit the imbalance in the dataset. In case the majority of bugs are associated with a certain class, LSTM learns this overall tendency and favors this class more than others. This reduces the precision. Nevertheless, f-measure and accuracy measures based on the overall evaluation were still higher for LSTM.

4.4. Threats to validity

We have performed a case study in the context of a company. Therefore, it is subject to external validity threats (Wohlin et al., 2012). However, our study adds empirical evidence to the wider body of knowledge from an industrial context. More case studies can be conducted in different contexts to increase the generalizability of results. In our study, textual descriptions of bugs were not effective as a dataset for training a prediction model. Previous studies report better results based on text mining (Menzies and Marcus, 2008; Lamkanfi et al., 2010). This is due to the highly unstructured and inconsistent content we had, which might not be the case in other application domains. In any case, it is not realistic to expect formal descriptions in a fast moving business environment. However, descriptions might be based on a template and a set of rules or they can be provided in a semi-formal structure (Karagöz and Sözer, 2017).

One can question the way that measurements are performed, concerning internal validity threats. In our case study, we used real artifacts without any instrumentation or preprocessing. We only removed some bug attributes, which would not be available at the time of creating the issue. Results would be unfairly better if we did not exclude such attributes from our dataset. In the following section, we further elaborate on the results and lessons learned from our case study.

5. Discussion and lessons learned

The first take away message from our study is that one should consider the evolution of bug characteristics over time when using supervised machine learning techniques for predicting bug attributes. Our results showed that the priority of bugs can be predicted more accurately with less but more recent data. This consideration is especially relevant for long-lasting software development and maintenance projects that span several years. More training data will not necessarily improve the results. Less is more.

The second important consideration is related to the application context and constraints, which impact the contents of the utilized dataset. Consumer electronics market imposes a fast moving business environment being subject to hard deadlines and limited resources. It is not very easy to regulate the way that textual bug descriptions are provided in such an environment. The context of open source projects is not subject to such constraints. Probably that was the reason why all the previous studies reported highly accurate results for the prediction of bug priorities based on textual descriptions. We did not observe the same outcome in our study. In fact, textual descriptions were not reliable at all for performing prediction. The context of projects also impacts the prioritization decisions. It is not always possible to patch embedded systems like open source systems. Even when it is possible, it turns out to be very costly to send software updates to millions of products. Bug-fixing schedules must address such concerns as well. Types of data collected regarding bugs can also vary. In fact, many of the existing studies on bug prioritization rely on textual descriptions only. We rely mainly on structural attributes instead. Even these attributes differ in various contexts. We summarize existing related studies in the following section.

Finally, the choice of machine learning techniques should also be based on the application context and the structure of the dataset. Our results showed that LSTM performs better when compared with a Naive Bayes classifier. In particular, it is effective for handling defect characteristics that vary over a long period of software development life-cycle. It can automatically learn the characteristics of various project periods and filter out information that is no longer relevant for the current period. However, LSTM requires a large dataset to be effective and it might not be relevant for projects with shorter durations. One should also be careful about the distribution of classes in the dataset and the level of imbalance. We observed that LSTM leads to low precision as it tends to exploit imbalance to increase accuracy.

6. Related work

There have been several bug prioritization approaches that are mainly based on text classification. Majority of these approaches, including both earlier (Menzies and Marcus, 2008; Lamkanfi et al., 2010) and recent (Ramay et al., 2019) studies employ text mining techniques. Hereby, features of defects are derived from their textual descriptions. These features are used for estimating the severity of a bug. Mostly Naive Bayes algorithm has been used for classification (Lamkanfi et al., 2010; Zhou et al., 2016). Deep learning techniques have also been recently used for this purpose (Ramay et al., 2019). However, our study is different from all of these studies based on two aspects. First, previous studies (Lamkanfi et al., 2010; Xuan et al., 2012; Ramay et al., 2019) employ a dataset composed of textual bug descriptions issued for a set of open source projects (Mozilla, Eclipse, GNOME). To the best of our knowledge, there is no evaluation of such approaches in an industrial context. There is only one study that was applied on a NASA project (Menzies and Marcus, 2008). Second, all of these studies utilize solely the textual descriptions of bugs for estimating their severity. This approach did not turn out to be successful in our case study. The main reason was unstructured and inconsistent (even in terms of the language used) content, which constitute a low-quality training data for supervised models. In practice, information regarding detected bugs are managed via bug tracking systems like JIRA. Information provided for these bugs do not only include textual descriptions but also a set of predefined structured attributes like the related software component(s) and the type of test that revealed the bug. In this study, we aimed at exploiting such attributes as well.

There have also been studies, where a set of structured attributes regarding bugs are used together with their textual descriptions to determine their severity (Zhou et al., 2014, 2016). However, these attributes are not exactly the same as those used in our study. One basic difference is the definition of two types of priorities; one focusing on the severity of the failure caused by the bug (*Priority*), while the other one (*Approval Type*) actually representing its priority in bug-fixing schedule as decided by addressing a variety of concerns. Moreover, these studies ignored the timeline of the reported bugs and the impact of variance of bug types and severities over the project lifetime. This is also the case for other previous studies (Xuan et al., 2012; Lamkanfi et al., 2010; Ramay et al., 2019), where the dataset is used altogether for training and testing of prediction models. Hereby, cross validation is applied and the dataset is divided into training sets and test sets arbitrarily. First of all, this is not possible in a real-life setting. One can only employ information regarding bugs reported in the history to predict the severity of new bugs. Moreover, types of bugs can vary in time. We aimed at investigating the impact of this variation on the accuracy of classification. Therefore we collected over 15,000 bugs collected over 3 years and split them in 6 successive periods that are aligned with the actual periods of the project. We defined various training and test sets based on the ordering of these periods.

7. Conclusion

We evaluated the effectiveness of an automated defect prioritization approach that employs supervised machine learning. We used both a Naive Bayes classifier and a Long Short-Term Memory model. We performed an industrial case study with a real Smart TV project. We compiled 15,318 defect reports collected over 3 years. Types of defects that are revealed at the beginning of a software development project can be very different than those reported later on. We evaluated the impact of this variation on the accuracy of assignment. We also evaluated the impact of using defect attributes, textual descriptions, or both as training data. We trained our classifiers with both textual descriptions of defects and structured features collected from defect reports.

We could reach an accuracy level up to 79.36%. We observed that Long Short-Term Memory has a better overall accuracy when compared with a Naive Bayes classifier although precision turns out to be lower for imbalanced datasets. We also observed that structured features lead to better accuracy compared to textual descriptions if these descriptions are not standardized. Accuracy even decreases when structural features are combined with an additional feature derived from textual descriptions. Finally, we discovered that accuracy is not improved by considering increasingly earlier defects as part of the training data. Increasing the size of the training data even decreases the accuracy. We obtained better results when we use data regarding defects reported in the last 6 months only.

CRediT authorship contribution statement

Mustafa Gökçeoğlu: Methodology, Software, Data curation, Investigation, Validation, Writing - original draft, Writing - review & editing. **Hasan Sözer:** Conceptualization, Supervision, Writing - original draft, Writing - review & editing.

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

We would like to thank software developers at Vestel for sharing their JIRA records with us and supporting our case study.

References

Dirim, S., Sozer, H., 2020. Prioritization of test cases with varying test costs and fault severities for certification testing. In: Proceedings of the 15th Workshop on Testing: Academic and Industrial Conference Practice and Research Techniques. Porto, Portugal. pp. 386–391.

Gebizli, C.S., Sozer, H., 2016. Model-based software product line testing by coupling feature models with hierarchical Markov chain usage models. In: Proceedings of the 6th IEEE International Workshop on Model-Based Verification and Validation. pp. 278–283.

Gers, F., Schmidhuber, E., 2001. LSTM recurrent networks learn simple context-free and context-sensitive languages. IEEE Trans. Neural Netw. 12 (6), 1333–1340.

Greff, K., Srivastava, R.K., Koutník, J., Steunebrink, B.R., Schmidhuber, J., 2017. LSTM: A search space odyssey. IEEE Trans. Neural Netw. Learn. Syst. 28 (10), 2222–2232.

Hagan, M., Demuth, H., Beale, M., 1995. Neural Network Design. PWS Publishing, New York, NY, USA.

Herzig, K., Just, S., Zeller, A., 2013. It's not a bug, it's a feature: How misclassification impacts bug prediction. In: Proceedings of the International Conference on Software Engineering. pp. 392–401.

Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 9 (8), 1735–1780.

Karagöz, G., Sözer, H., 2017. Reproducing failures based on semiformal failure scenario descriptions. Softw. Qualilty J. 25 (1), 111–129.

Lamkanfi, A., Demeyer, S., Giger, E., Goethals, B., 2010. Predicting the severity of a reported bug. In: Proceedings of the 7th IEEE Working Conference on Mining Software Repositories. pp. 1–10.

Menzies, T., Marcus, A., 2008. Automated severity assessment of software defect reports. In: Proceedings of the IEEE International Conference on Software Maintenance. pp. 346–355.

Nair, V., Hinton, G.E., 2010. Rectified linear units improve restricted boltzmann machines. In: Proceedings of the 27th International Conference on Machine Learning. pp. 807–814.

Powers, D.M.W., 2011. Evaluation: From precision, recall and f-measure to roc., informedness, markedness & correlation. J. Mach. Learn. Technol. 2 (1), 37–63.

Ramay, W., Umer, Q., Yin, X., Zhu, C., Illahi, I., 2019. Deep neural network-based severity prediction of bug reports. IEEE Access 7, 46846–46857.

Strate, J.D., Laplante, P.A., 2013. A literature review of research in software defect reporting. IEEE Trans. Reliab. 62 (2), 444–454.

Wohlin, C., Runeson, P., Host, M., Ohlsson, M., Regnell, B., Wesslen, A., 2012. Experimentation in Software Engineering. Springer-Verlag, Berlin, Heidelberg.

Wu, Y., et al., 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. CoRR abs/1609.08144.

Xuan, J., Jiang, H., Ren, Z., Zou, W., 2012. Developer prioritization in bug repositories. In: Proceedings of the 34th International Conference on Software Engineering. pp. 25–35.

Zhou, Y., Tong, Y., Gu, R., Gall, H., 2014. Combining text mining and data mining for bug report classification. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution. pp. 311–320.

Zhou, Y., Tong, Y., Gu, R., Gall, H., 2016. Combining text mining and data mining for bug report classification. J. Softw.: Evol. Process 28 (3), 150–176.

Zoeteweij, P., Abreu, R., Golsteijn, R., van Gemund, A.J.C., 2007. Diagnosis of embedded software using program spectra. In: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems. pp. 213–220.

Mustafa Gökçeoğlu received his B.Sc. degree in computer engineering from Istanbul Technical University in 2017. He received his M.S. degree in computer science from Ozyegin University in 2021. Since his graduation from Istanbul Technical University, Mustafa Gökçeoğlu has been with Vestel Electronics, where he is currently working as a senior software specialist.

Hasan Sözer received his B.Sc. and M.S. degrees in computer engineering from Bilkent University, Turkey, in 2002 and 2004, respectively. He received his Ph.D. degree in 2009 from the University of Twente, The Netherlands. From 2002 until 2005, he worked as a software engineer at Aselsan Inc. in Turkey. From 2009 until 2011, he worked as a postdoctoral researcher at the University of Twente. In 2011, he joined the department of computer science at Ozyegin University, where he is currently working as an associate professor.