AUTOMATED DEFECT PRIORITIZATION BASED ON DEFECTS RESOLVED AT VARIOUS PROJECT PHASES

A Thesis

by

Mustafa Gökçeoğlu

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in the
Department of Computer Science

Özyeğin University
January 2021

AUTOMATED DEFECT PRIORITIZATION BASED ON DEFECTS RESOLVED AT VARIOUS PROJECT PHASES

Approved by:

Assoc. Prof. Hasan Sözer, Advisor
Department of Computer Science
Özyeğin University

Asst. Prof. Furkan Kıraç
Department of Computer Science
Özyeğin University

Prof. Oya Kalıpsız
Department of Computer Science
Yildiz Technical University

Date Approved: 14 January 2021

To my family ...

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 recent defects reported in the previous phase of the project.

ÖZETÇE

Günümüzde, bir projedeki hataların önceliklendirmesi emek isteyen ve yanlışlıklara sebep olabilecek bir iştir. Bu çalışmada, makine öğrenimini kullanan otomatik bir yaklaşımın, bu iş için ne derecede etkin olduğu değerlendirilmiştir. Bu değerlendirme yapılırken, Naive Bayes sınıflandırıcısı ve uzun-kısa süreli bellek modeli (LSTM) uygulayan iki farklı teknik kullanılmıştır. Tüketici elektroniği alanından gerçek bir proje ile bir endüstriyel vaka çalışması gerçekleştirilmiştir. 3 yılda toplanan 15.000'den fazla örnek hata raporu derlenmiştir. Test sonuçlarında doğruluk değeri %79,36 seviyesine kadar ulaşmış ve bu sonuçlara göre 3 çıkarımda bulunulmuştur. İlk olarak, LSTM modeli, Naive Bayes sınıflandırmasına göre daha iyi doğruluğa sahiptir. İkinci olarak, yapılandırılmış özellikler, metinsel açıklamalara kıyasla daha iyi doğruluk sağlamaktadır. Üçüncüsü, eğitim kümemizde yakın zamanda raporlanan hatalara ek olarak, daha eskiden raporlanan hatalar kullanıldığında, doğruluk değeri artmamaktadır. Aksine, test kümemize ilişkin proje fazından sadece bir önceki fazın verileri eğitim kümesi olarak kullandığında, bütün fazlardaki verilerin eğitim kümesi olarak kullanıldığı duruma göre doğruluk seviyesinin daha yüksek olduğu görülmüştür.

ACKNOWLEDGEMENTS

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

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

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 2,764, 6,976 and 3,263, respectively [1]. It is necessary to systematically manage and trace these defects to reduce costs and improve system reliability [2]. 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 [3]. 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 [4]. 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 [5]. In fact, some defects might not be even pointing at actual bugs [3] 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 [6, 1, 7]. 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

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

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 early phases of a project can be very different than those reported at later phases. 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 mostly used and the most successful one for our classification purpose [1, 7]. Second, we employed a *Long Short-Term Memory (LSTM)* [8] 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 [9] 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 previous phase of the project, that is, the last 6 months only.

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

CHAPTER II

BACKGROUND

In the following sections, we provide background information on *Naive Bayes Classifier Algorithm* and *Long Short-Term Memory (LSTM)* networks that are utilized in our work.

2.1 Naive Bayes Classifier Algorithm

Naive Bayes classifier algorithm is a type of supervised learning algorithm which uses the Bayes Theorem. There are various types of this algorithm such as Multinomial Naive Bayes and Bernoulli Naive Bayes [10, 11]. The basic theory is based on the probability of events and their relationships (conditional probabilities) as formulated in Equation 1.

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)} \tag{1}$$

The set of terms taking place in Equation 1 can be summarized as follows:

- P(Y): Probability of occurrence of event Y
- P(X): Probability of occurrence of event X
- P(Y|X): Probability of occurrence of event Y given that event X occurs
- P(X|Y): Probability of occurrence of event X given that event Y occurs

Hereby, the event X represents evidence for the occurrence of event Y. As such, it is used as a feature for determining the probability of occurrence of event Y. Usually, there is more than one feature used for this prediction task. Therefore, the equation

takes the form as shown in Equation 2.

$$P(Y|X1, X2, ..) = \frac{P(X1, X2, ..|Y)P(Y)}{P(X1, X2, ..)} \tag{2}$$

Features are assumed to be conditionally independent of each other in Naive Bayes. So, the probability of co-occurrence of multiple features can be calculated by the multiplication of the probabilities of these features as shown in Equation 3.

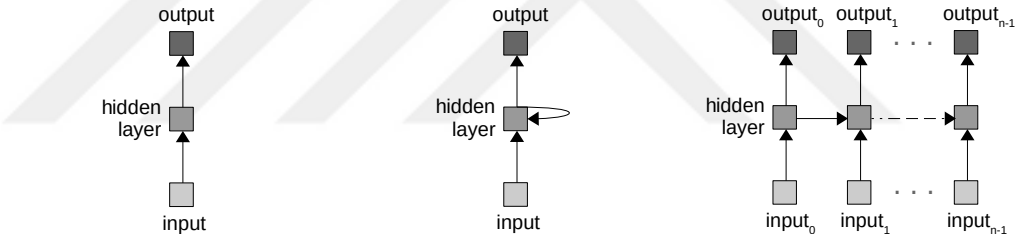
$$P(Y|Xn) = P(X1|Y) * P(X2|Y) * ... * P(Xn|Y) * P(Y) \tag{3}$$

Naive Bayes classifier algorithm explores probabilistic relationships among features taking place in a data set. One of these features is called the prior and its value is predicted based on the values of the other features, which are called evidence features. In this study, the set of features include properties of issue reports such as the approval type, priority, assignee, bug category, component, labels, test type and summary. Our prior feature is approval type, and other features are evidence features. Approval type feature will be classified according to probabilities of evidence features.

Naive Bayes is most commonly used in text-document classification tasks like spam filtering [10]. It has both some advantages and disadvantages. As an advantage, one can obtain accurate results with Naive Bayes classification even if the size of the training data is small. The main disadvantage of Naive Bayes is its assumption regarding the relations between features. Naive Bayes assumes that all the features are independent from each other. If some features are related to each other, such relationships are ignored during the classification process.

2.2 Long Short-Term Memory

LSTM [8] is a type of *Recurrent Neural Network (RNN)*. Figure 1(a) depict a traditional feed-forward neural network [12], whereas Figures 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. Figure 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.



(a) Traditional NN (b) RNN (c) RNN in unfolded form

Figure 1: Traditional Neural Network (NN) vs. Recurrent NN (RNN)

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 [13]. 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

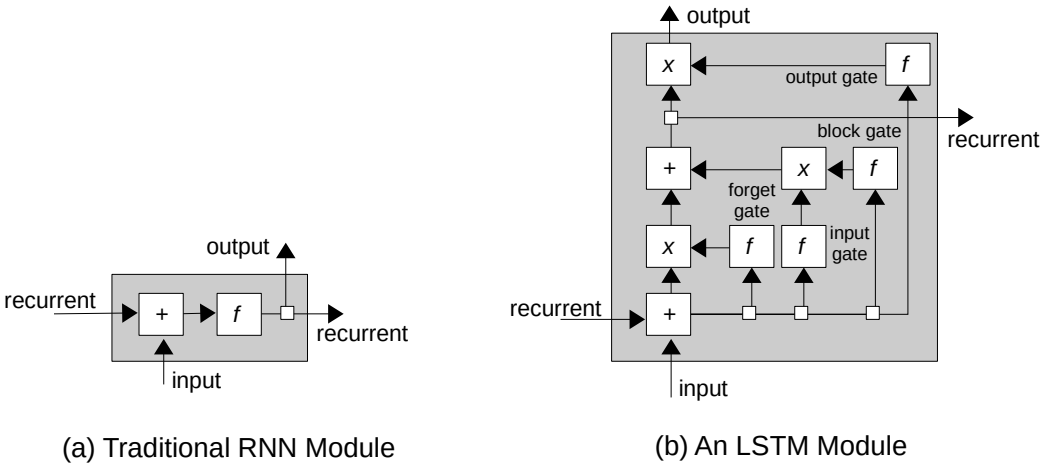


Figure 2: Traditional RNN vs. LSTM network

in LSTM is depicted in Figure 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 [12], or rectified linear units (ReLU) [14]. 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 [15] such as Google Translate [16]. 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 phases. In fact, LSTM has been recently used for predicting severities of bug reports based on their textual

descriptions [9] . We employed structural features such as *assignee* and *bug category* for the prediction task. We also used the Naive Bayes algorithm, which has been pointed out to be successful for bug prioritization [1, 7]. We trained our models with both unstructured, textual descriptions of defects and structured features collected from defect reports. In the following chapter, we describe our approach and design decisions.

CHAPTER III

THE OVERALL APPROACH

Figure 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 chapter. Therefore, they were not involved in our study with the LSTM model.

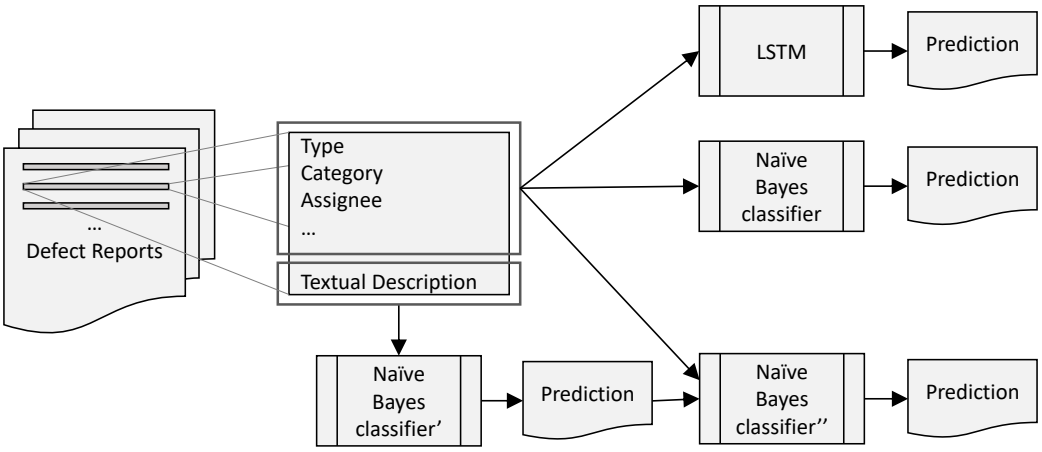


Figure 3: The overall approach.

Figure 4 depicts the approach we employed for combining structural features with an additional feature derived from predictions based on textual descriptions of previous phases. The steps taken for a prediction task regarding *Phase 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 *Phase n-2*;
2. Priorities of defects in *Phase 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 *Phase n-1*;
3. The dataset obtained in *Step 2* is used for training the second classifier;
4. Priorities of defects in *Phase 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 *Phase n*;
5. The dataset obtained in *Step 4* is used for predicting the priorities with the second classifier.

Figures 3 and 4 depict only a single instance of the process. We repeated this process for each phase multiple times. We used a different training set in each repetition. In the first test, the training set included data only from the previous phase. The second training was based on two previous phases. The last test involved all the data from the beginning of the project until the corresponding phase. The process depicted in Figure 4 employs two cascading Naive Bayes classifiers. Therefore, it requires data from at least two previous phases for the prediction task concerning a particular phase. For instance, the process depicted in Figure 4 represents a prediction task regarding phase n, based on data collected in phases n-1 and n-2.

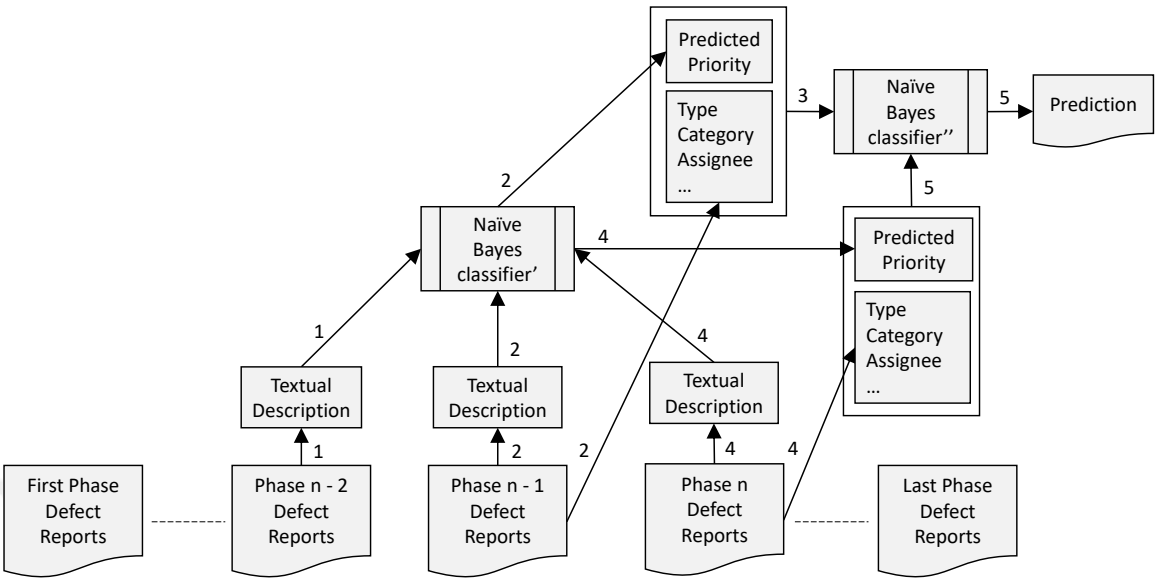


Figure 4: Combining structural features with an additional feature derived from predictions based on textual descriptions of previous phases.

Figures 5 depicts the approach we employed by using LSTM for classification. As the first step of this approach, structured attributes of all the defect reports are collected. One of these attributes (*Approval Type*) represents the severity of the defect. The value of this attribute is subject to prediction. Hence, it is used as the label in our dataset. The remaining attributes are provided as input for the neural network model. The overall dataset is divided into 6 successive periods to form the training and test sets. A particular test set is used for prediction with multiple models each of which is trained with a different dataset. First, the prediction task regarding phase n is performed after training the model with data from just the previous phase, i.e., phase n-1. Then the same prediction task is performed after the model is trained by using the data collected from phases n-1 and n-2. The task is repeated until the last test, where the data regarding all the previous phases are utilized for training.

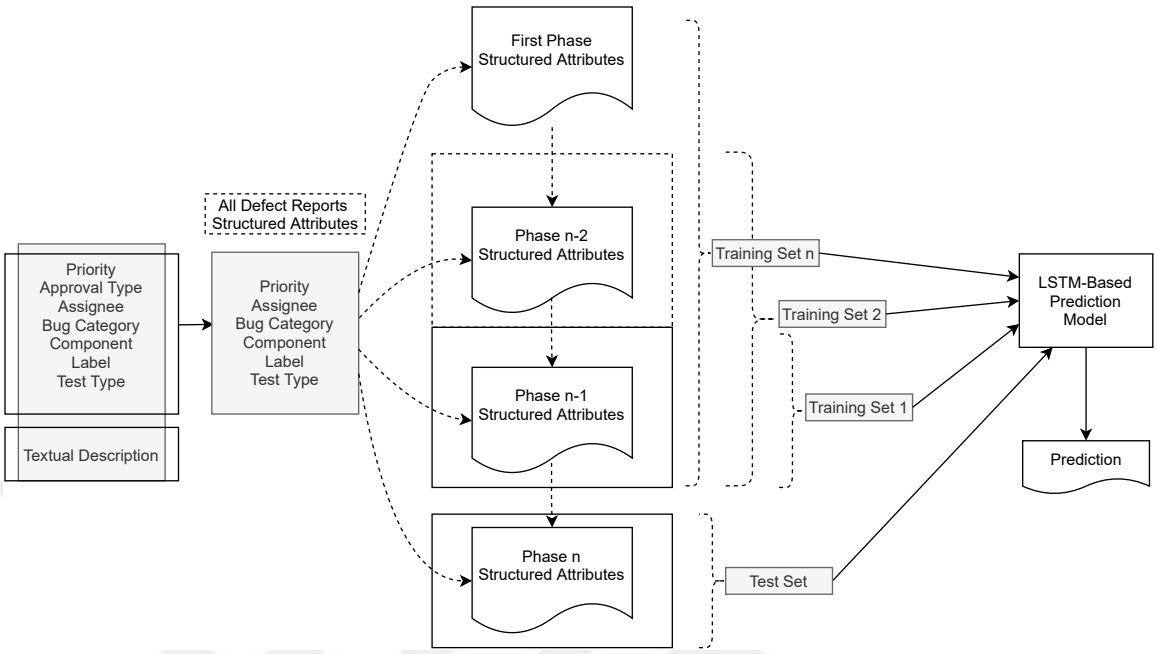


Figure 5: The overall process employed for prediction with the LSTM-based model.

Figure 6 depicts the LSTM model that is employed in our approach. We used 6 structural attributes in the input layer as explained in the following section. These attributes are encoded by using one-hot encoding since they are mainly in the form of nominal attributes. Further details regarding the types of attributes are provided in the following section. The hidden layer involves the LSTM modules as explained in the background section and depicted in Figure 2. Input layer elements are fully connected to these modules. We used the Weka[1] toolset for constructing the model. Hidden layer size is calculated automatically by this toolset based on the sizes of input and output layers. Hence, we did not include this structural property among the list of tuned parameters (See Table 1). The output of LSTM modules are connected to the elements of the output layer with using a soft sign activation. The output layer is where the the prediction results are obtained. The predicted attribute in our case study can take one of 4 values as explained in the following section. Therefore, we have 4 output class in our LSTM model.

[1]https://www.cs.waikato.ac.nz/ml/weka/

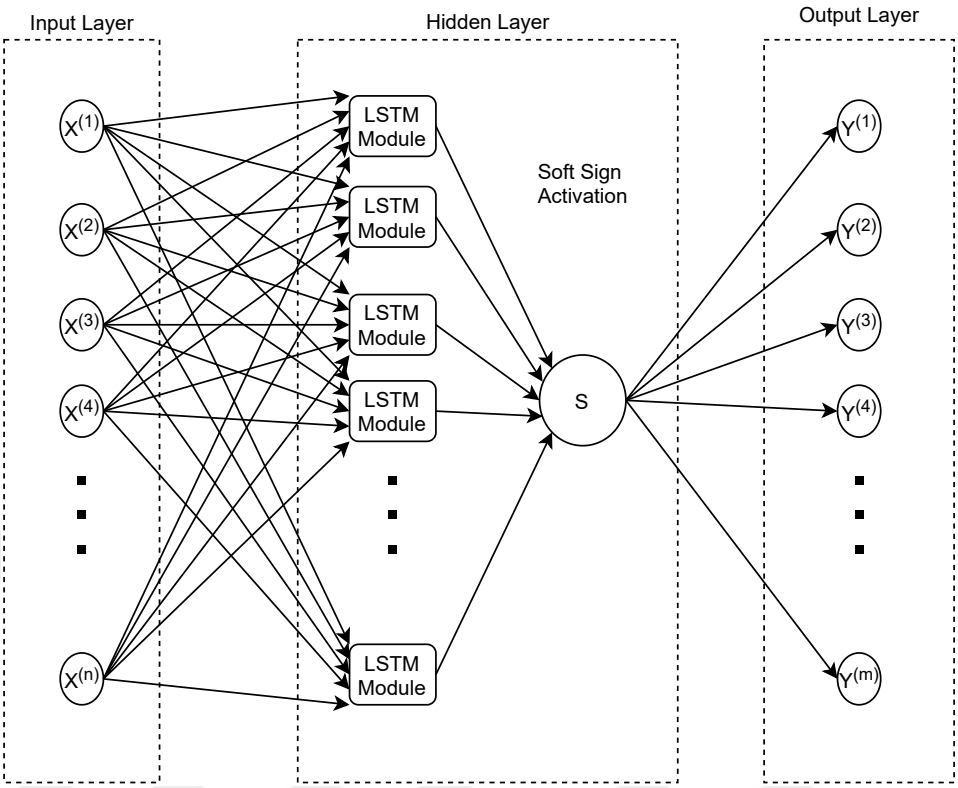


Figure 6: An overview of the constructed LSTM model.

LSTM models are subject to several parameters to be tuned. Table 1 lists the parameters and the set of values considered for each parameter while tuning our models. The actual values adopted for models were determined to achieve high accuracy by performing a grid search over possible combinations of the 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 considered ReLU, Soft Sign, Softmax and Sigmoid as candidate activation functions. We considered Adam, Adadelta, SGD, Adagrad as candidate optimization algorithms. We omitted additional layers like dropout layer since they did not have a significant impact on accuracy. Hidden layer size was calculated automatically by the toolset we employed, based on the sizes of input and output layers as mentioned above.

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

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

CHAPTER IV

INDUSTRIAL CASE STUDY

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 are also interested in the accuracy of all these alternatives at various project phases. In each phase, 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. Types of defects that are reported at early phases can be very different than those reported at later phases. 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. 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.You can see the example of Jira screenshot in appendix part. 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 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.

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

#	Name	Description
1	Priority	Priority of bug
2	**Approval Type**	Severity of bug for production
3	Assignee	One of the sotware 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

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.

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

Approval type	**Description**
$S1$	Very urgent; the bug can stop the production
$S2$	Urgent
$S3$	Not urgent but needs to be fixed
$S4$	No need to be fixed anytime soon

The first attribute listed in Table 2, *Priority* is assigned by test engineers based on how important they consider the registered bugs. However, they can not accurately estimate the severity of a bug in terms of its impact on the production process, i.e., *approval type*. Therefore, these two attributes do not represent the same property and they are commonly not consistent in practice. 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. *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 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.

Table 4 depicts the number of distinct values and types for each attribute in our dataset. We can see that there are two different types other than text: ordinal and nominal. Nominal attributes can take values that represent various categories without any particular ordering among these. On the other hand, the values of ordinal attributes are ordered. For instance, the *Approval Type* attribute can take 4 different values as listed in Table 3 and they are ordered from S1 to S4, representing the most severe and the least severe defects, respectively. This attribute is subject to prediction in our case. The only other attribute of ordinal type is *priority*. All the other attributes except *Summary* are of type nominal, meaning that there is no ordering defined among their values. One-hot encoding is used for representing the values of nominal attributes. The *priority* attribute is represented via two alternative approaches. First, we used one-hot encoding just like the other attributes. Second, we represented it as numeric data. One can consider a numeric representation more meaningful for an ordinal attribute; however, the number of distinct values were just 4 in our case. Therefore, the type and the corresponding representation of the attribute is not subject to a significant impact concerning the results. We also observed this fact empirically. In the following, we mainly share the results when all the attributes are represented via one-hot encoding. However, we share additional results in Appendix B, where we employed a numeric representation for the *priority* attribute.

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 5 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

Table 4: Number of used attributes and their types

#	Name	Data Type	Distinct Values
1	Priority	Ordinal	4
2	**Approval Type**	**Ordinal**	4
3	Assignee	Nominal	35
4	Bug Category	Nominal	7
5	Component	Nominal	1348
6	Labels	Nominal	235
7	Test Type	Nominal	2
8	Summary	Text	Number of Bugs

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.

Table 5: Bugs that have to be re-prioritized manually.

Period	# of re-prioritized bugs	# of bugs	%
P1	191	665	28.72
P2	1,995	2,207	90.39
P3	1,710	2,672	63.99
P4	3,714	4,424	83.95
P5	2,893	3,305	87.53
P6	1,461	2,045	71.44

In our study, our goal is to predict the approval type automatically and accurately when a bug is created. We use (supervised) machine learning for this purpose. In particular, we use a Naive Bayes classifier as many previous studies did [1, 7] as well as an 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 Chapter 3. We employed textblob[1] for text classification. We used the Naive Bayes classifier and LSTM model implemeted by the Weka[2] 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. Figure 7 depicts these enumerated periods from *P1* to *P6*.

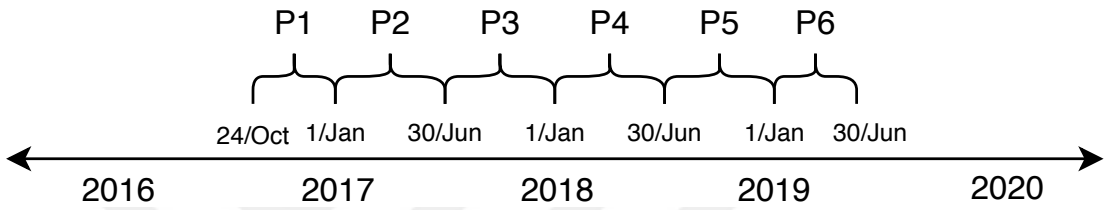


Figure 7: Periods over which the data collection is performed.

The way that the dataset is divided into periods was not based on our decision. Periods depicted in Figure 7 reflect successive project phases. The end of each phase is a milestone. The overall timeline including the phases and milestones are determined by the project management. Table 6 lists the duration and the number of bugs associated with each period.

Table 6: The duration and the number of bugs associated with each period.

Period Name	**Duration (months)**	**# of bugs per *Approval Type***				
		S1	**S2**	**S3**	**S4**	**Total**
P1	3	71	63	7	524	665
P2	6	1,444	311	100	352	2,207
P3	6	1,351	66	149	1,106	2,672
P4	6	2,537	130	356	1,401	4,424
P5	6	2,444	25	191	645	3,305
P6	6	1,139	197	170	539	2,045

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

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 phase. 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 [17], where *TN*, *TP*, *FN* and *FP* represent the number of *true negatives*, *true positives*, *false negatives* and *false positives*, respectively:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{4}$$

$$Precision = \frac{TP}{TP + FP} \tag{5}$$

$$Recall = \frac{TP}{TP + FN} \tag{6}$$

$$F = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{7}$$

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 (Equation 4) 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 data sets. For instance, the approval type for 2,444 out of 3,305 bug reports in *P5* is set as *S1* (See Table 6). 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* (Equation 5) evaluates to what extent bugs classified as *S1* are indeed among the most critical ones. Low precision leads to waste of resources. *Recall* (Equation 6) 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* (Equation 7) balances these two measures. We share and discuss the results in the following section.

4.3 Results and Discussion

Table 7 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. Each of the the listed accuracy values are in fact the average of results obtained after 10 successive training and test session. 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.

Table 7: 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	**33.34**	15.94
2	P2	P3	54.04	**<u>60.1</u>**
3	P1,P2	P3	55.23	**55.83**
4	P3	P4	52.57	**<u>58.11</u>**
5	P2,P3	P4	51.6	**57.84**
6	P1,P2,P3	P4	51.42	**52.93**
7	P4	P5	75.64	**<u>79.36</u>**
8	P3,P4	P5	73.1	**77.54**
9	P2,P3,P4	P5	68.74	**75.37**
10	P1,P2,P3,P4	P5	67.92	**71.28**
11	P5	P6	**66.41**	65.96
12	P4,P5	P6	65.52	**<u>67.18</u>**
13	P3,P4,P5	P6	63.17	**66.74**
14	P2,P3,P4,P5	P6	63.66	**66.50**
15	P1,P2,P3,P4,P5	P6	63.28	**66.20**

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 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 6). 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.

Recall that we repeated the training and test sessions for 10 times for each test to observe the variation in results. Figure 8 depicts a box plot that shows this variation regarding the results obtained with the LSTM model for *P5*. Likewise, Figure 9 shows the variation in results obtained with the LSTM model for *P6* after 10 successive

training and test sessions. We can see from figures that the amount of variation is very low. This is also the case for other tests. The highest difference between the minimum (48.12) and the maximum (59.31) accuracy values is observed for test 3 as 11.19. In fact, the minimum accuracy value for this test can be considered as an outlier. We had similar observations for the other tests as well. For instance, the second highest difference between the minimum (65.56) and the maximum (74.25) accuracy values is observed for test 10 as 8.69. We can see from the tail of the last box in Figure 8 that this difference is caused by an outlier. Detailed results for all the tests are shared in the Appendix.

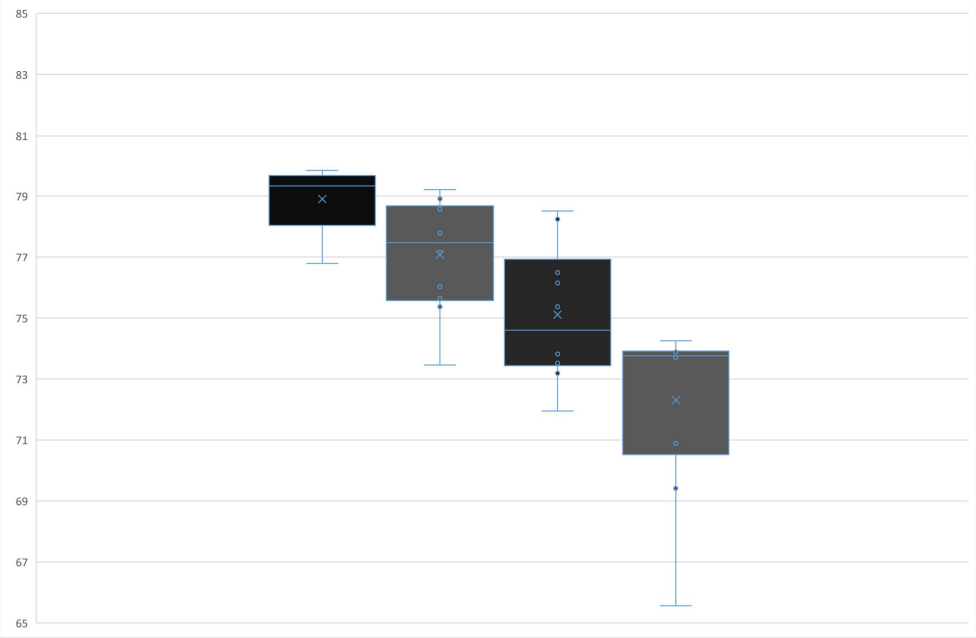


Figure 8: Variations in results obtained with the LSTM model for *P5* after 10 successive training and test sessions.

Table 8 lists the overall results regarding predictions that are performed only with textual descriptions of bugs. The table has the same structure as Table 7 except the last column. Predictions based on the *Summary* attribute were performed only

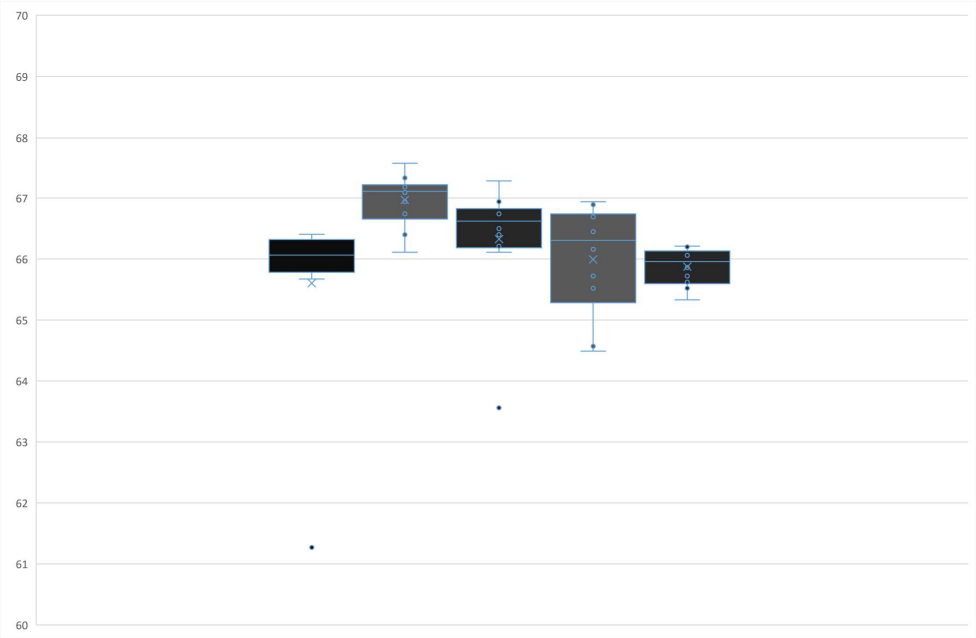


Figure 9: Variations in results obtained with the LSTM model for *P6* after 10 successive training and test sessions.

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 8. 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 9 lists the overall results regarding predictions when structured attributes of

Table 8: 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	**16.67**
2	P2	P3	50.74
3	P1,P2	P3	**52.47**
4	P3	P4	**43.53**
5	P2,P3	P4	42.04
6	P1,P2,P3	P4	40.23
7	P4	P5	57.48
8	P3,P4	P5	56.24
9	P2,P3,P4	P5	**60.66**
10	P1,P2,P3,P4	P5	58.33
11	P5	P6	**55.84**
12	P4,P5	P6	55.45
13	P3,P4,P5	P6	53.30
14	P2,P3,P4,P5	P6	52.95
15	P1,P2,P3,P4,P5	P6	52.86

bugs are used together with prediction results obtained based on textual descriptions. The table has the same structure as Table 8. 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 7). It is also not possible to observe a consistent trend and derive conclusions based on these results.

Table 10 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

Table 9: 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	**56.21**
4	P3	P4	NA
5	P2,P3	P4	**53.05**
6	P1,P2,P3	P4	50.18
7	P4	P5	NA
8	P3,P4	P5	**72.34**
9	P2,P3,P4	P5	68.16
10	P1,P2,P3,P4	P5	69.19
11	P5	P6	NA
12	P4,P5	P6	60.36
13	P3,P4,P5	P6	57.80
14	P2,P3,P4,P5	P6	58.85
15	P1,P2,P3,P4,P5	P6	**61.26**

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 bugs with approval type *S1* is high.

Overall, we conclude regarding *RQ1* that the prediction accuracy can be increased up to 79.36%. Despite the fact that we used an industrial data set, 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

Table 10: 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**	**Recall**	**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

been reported as 30% and 90%, respectively for various other data sets [1, 18, 19]. 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.

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.

Finally, we observed regarding *RQ3* that LSTM has a better overall accuracy when compared with a Naive Bayes classifier although precision turns out to be lower for imbalanced datasets.

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 [20]. 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 [6, 1]. 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 [21].

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.

CHAPTER V

RELATED WORK

There have been several bug prioritization approaches that are mainly based on text classification. Majority of these approaches, including both earlier [6, 1] and recent [9] 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 [1, 7]. Deep learning techniques have also been recently used for this purpose [9]. However, our study is different from all of these studies based on two aspects. First, previous studies [1, 18, 9] 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 [6]. 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 [19, 7]. However, these studies ignored the timeline of the reported bugs and the impact of

variance of bug types and severities in various project phases over a long period of time. This is also the case for other previous studies [18, 1, 9], 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 phases of the project. We defined various training and test sets based on the ordering of these periods.

CHAPTER VI

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 at later phases. 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 previous phase of the project only.

We have performed a case study in the context of a particular company. Therefore, an obvious future work direction would involve the realization of more case studies in

different contexts to increase the generalizability of results. Another research direction can consider the use of alternative machine learning models, in particular alternative recurrent neural network models like gated recurrent units, which are proven to be effective for other classification problems in various contexts.

APPENDIX A

A SAMPLE ISSUE REPORT

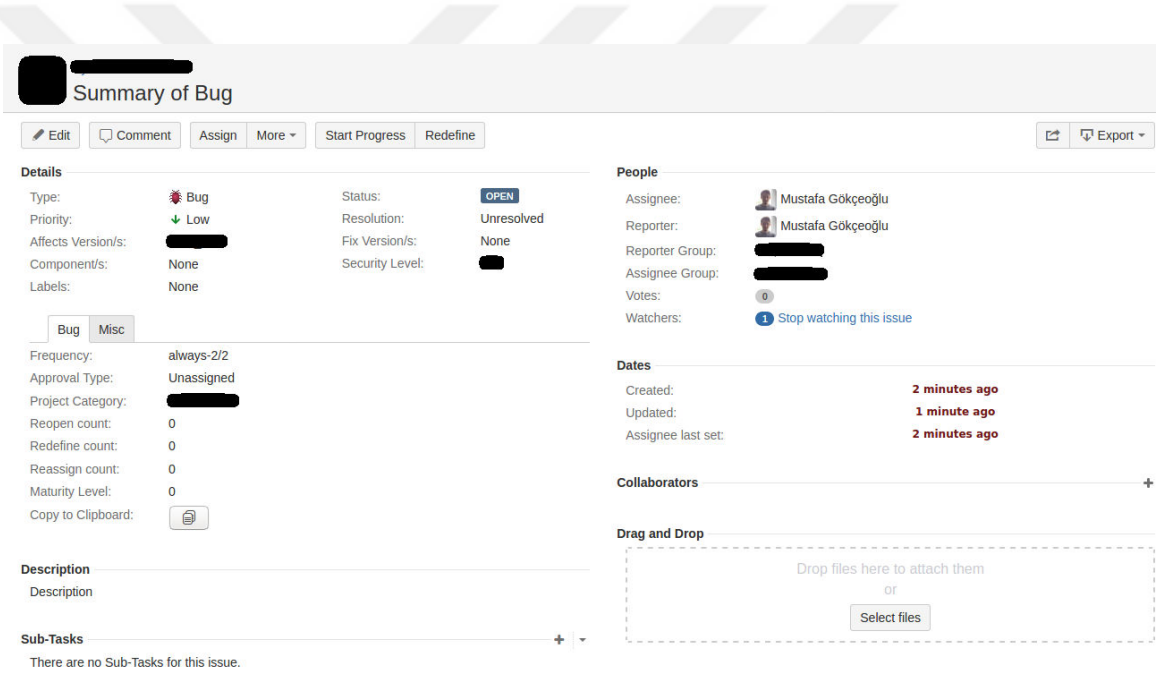


Figure 10: Example Jira Screenshot

APPENDIX B

RESULTS OBTAINED WITH LSTM MODELS

Table 11: Test results after 10 successive training and test sessions

	Trial 1	**Trial 2**	**Trial 3**	**Trial 4**	**Trial 5**	**Trial 6**	**Trial 7**	**Trial 8**	**Trial 9**	**Trial 10**
Test 1	15.94	15.94	15.94	15.94	15.94	15.94	15.94	15.94	15.94	15.94
Test 2	60.1	58.45	59.05	59.76	59.31	57.85	58.15	**60.32**	58.04	59.28
Test 3	55.83	59.16	50.56	58.30	58.08	57.29	48.12	**59.31**	52.80	57.97
Test 4	57.34	64.39	58.11	**64.89**	59.51	64.17	59.40	58.83	60.03	61.95
Test 5	57.80	55.71	**62.56**	61.27	56.71	62.40	59.85	58.45	56.66	56.82
Test 6	52.93	**59.96**	56.48	59.40	53.88	53.91	59.40	52.66	53.91	52.98
Test 7	79.72	79.36	76.79	79.30	79.42	78.06	**79.84**	77.97	78.88	79.66
Test 8	76.03	75.64	**79.21**	78.91	78.57	75.37	78.60	73.46	77.79	77.15
Test 9	75.37	**78.51**	76.15	78.24	73.52	76.49	73.83	73.82	73.19	71.95
Test 10	65.56	73.88	70.89	73.76	**74.25**	73.94	73.91	69.41	73.70	73.76
Test 11	65.96	65.82	61.27	65.67	**66.41**	66.31	66.36	65.87	66.16	66.21
Test 12	67.18	66.94	67.33	**67.57**	66.74	67.09	67.13	66.11	67.18	66.40
Test 13	66.74	66.11	66.40	66.79	66.94	63.56	66.50	66.21	66.74	**67.28**
Test 14	66.50	**66.94**	66.45	64.49	65.52	66.16	65.72	66.69	66.89	64.57
Test 15	**66.20**	66.11	66.21	65.72	65.86	65.62	66.06	65.52	66.11	65.33

Table 12: Test results after 10 successive training and test sessions when the *priority* attribute datatype is numerical

	Test 7	**Test 8**	**Test 9**	**Test 10**
Trial 1	79.33	**78.40**	73.40	72.61
Trial 2	78.66	76.73	**75.49**	**73.10**
Trial 3	78.97	76.21	73.49	71.13
Trial 4	78.82	73.94	74.01	72.61
Trial 5	79.15	73.76	74.46	70.31
Trial 6	78.09	73.49	74.85	68.71
Trial 7	78.97	76.01	72.88	71.73
Trial 8	**79.42**	74.09	72.34	68.89
Trial 9	78.88	75.43	73.94	69.74
Trial 10	77.73	77.06	72.52	71.37

Bibliography

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

[2] J. D. Strate and P. A. Laplante, "A literature review of research in software defect reporting," *IEEE Transactions on Reliability*, vol. 62, no. 2, pp. 444–454, 2013.

[3] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: How misclassification impacts bug prediction," in *Proceedings of the International Conference on Software Engineering*, p. 392–401, 2013.

[4] S. Dirim and H. Sozer, "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, 2020.

[5] P. Zoeteweij, R. Abreu, R. Golsteijn, and A. J. C. van Gemund, "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, 2007.

[6] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *Proceedings of the IEEE International Conference on Software Maintenance*, pp. 346–355, 2008.

[7] Y. Zhou, Y. Tong, R. Gu, and H. Gall, "Combining text mining and data mining for bug report classification," *Journal of Software: Evolution and Process*, vol. 28, no. 3, pp. 150–176, 2016.

[8] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computing*, vol. 9, no. 8, pp. 1735–1780, 1997.

[9] W. Ramay, Q. Umer, X. Yin, C. Zhu, and I. Illahi, "Deep neural network-based severity prediction of bug reports," *IEEE Access*, vol. 7, pp. 46846–46857, 01 2019.

[10] V. Metsis, I. Androutsopoulos, and G. Paliouras, "Spam filtering with naive bayes - which naive bayes?," in *CEAS*, 2006.

[11] A. McCallum and K. Nigam, "A comparison of event models for naive bayes text classification," 1998.

[12] M. Hagan, H. Demuth, and M. Beale, *Neural Network Design*. New York, NY, USA: PWS Publishing, 1995.

[13] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A search space odyssey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, pp. 2222–2232, 2017.

[14] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning*, pp. 807–814, 2010.

[15] F. Gers and E. Schmidhuber, "LSTM recurrent networks learn simple context-free and context-sensitive languages," *IEEE Transactions on Neural Networks*, vol. 12, no. 6, pp. 1333–1340, 2001.

[16] Yonghui Wu et al., "Google's neural machine translation system: Bridging the gap between human and machine translation," *CoRR*, vol. abs/1609.08144, 2016.

[17] D. M. W. Powers, "Evaluation: From precision, recall and f-measure to roc., informedness, markedness & correlation," *Journal of Machine Learning Technologies*, vol. 2, no. 1, pp. 37–63, 2011.

[18] J. Xuan, H. Jiang, Z. Ren, and W. Zou, "Developer prioritization in bug repositories," in *Proceedings of the 34th International Conference on Software Engineering*, pp. 25–35, 2012.

[19] Y. Zhou, Y. Tong, R. Gu, and H. Gall, "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, 2014.

[20] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer-Verlag, 2012.

[21] G. Karagöz and H. Sözer, "Reproducing failures based on semiformal failure scenario descriptions," *Software Qualilty Journal*, vol. 25, no. 1, pp. 111–129, 2017.

[22] D. E. Knuth, *The TEX Book*. Reading, Massachusetts: Addison-Wesley, 1984. Reprinted as Vol. A of *Computers & Typesetting*, 1986.

[23] D. E. Knuth, *TEX: The Program*, vol. B of *Computers & Typesetting*. Reading, Massachusetts: Addison-Wesley, 1986.

[24] D. E. Knuth, "The WEB system for structured documentation, version 2.3," Tech. Rep. STAN-CS-83-980, Computer Science Department, Stanford University, Stanford, California, Sept. 1983.

[25] D. E. Knuth, "Literate programming," *The Computer Journal*, vol. 27, pp. 97–111, May 1984.

[26] D. E. Knuth, "A torture test for TEX, version 1.3," Tech. Rep. STAN-CS-84-1027, Computer Science Department, Stanford University, Stanford, California, Nov. 1984.

[27] R. K. Furuta and P. A. MacKay, "Two TEX implementations for the IBM PC," *Dr. Dobb's Journal*, vol. 10, pp. 80–91, Sept. 1985.

[28] J. Désarménien, "How to run TEX in french," Tech. Rep. SATN-CS-1013, Computer Science Department, Stanford University, Stanford, California, Aug. 1984.

[29] A. L. Samuel, "First grade TEX: A beginner's TEX manual," Tech. Rep. SATN-CS-83-985, Computer Science Department, Stanford University, Stanford, California, Nov. 1983.

[30] L. Lamport, *LATEX: A Document Preparation System. User's Guide and Reference Manual*. Reading, Massachusetts: Addison-Wesley, 1986.

[31] M. D. Spivak, *The Joy of TEX*. American Mathematical Society, 1985.

[32] O. Patashnik, *BibTEXing*. Computer Science Department, Stanford University, Stanford, California, Jan. 1988. Available in the BibTEX release.

[33] O. Patashnik, *Designing BibTEX Styles*. Computer Science Department, Stanford University, Jan. 1988.

[34] D. Fuchs, "The format of TEX's DVI files version 1," *TUGboat*, vol. 2, pp. 12–16, July 1981.

[35] D. Fuchs, "Device independent file format," *TUGboat*, vol. 3, pp. 14–19, Oct. 1982.

[36] H. Naguib, N. Narayan, B. Brugge, and D. Helal, "Bug report assignee recommendation using activity profiles," in *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories*, pp. 22–30, 2013.

[37] X. Wei and W. Croft, "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*, pp. 178–185, 2006.

[38] X. Xie, W. Zhang, Y. Yang, and W. Qing, "DRETROM: Developer recommendation based on topic models for bug resolution," in *Proceedings of the 8th international conference on predictive models in software engineering*, pp. 19–28, 2012.

[39] J. Xuan, H. Jiang, Z. Ren, and W. Zou, "Developer prioritization in bug repositories," in *Proceedings of the 34th International Conference on Software Engineering*, pp. 25–35, 2012.

[40] W. Wu, W. Zhang, Y. Yang, and Q. Wang, "Drex: Developer recommendation with k-nearest-neighbor search and expertise ranking," in *Proceedings of the 18th Asia Pacific Software Engineering Conference*, pp. 389–396, 2011.

[41] A. Tamrawi, T. Nguyen, J. Al-Kofahi, and T. Nguyen, "Fuzzy set and cache-based approach for bug triaging," in *Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 365–375, 2011.

[42] F. Servant and J. Jones, "Whose fault: Automatic developer-to-fault assignment through fault localization," in *Proceedings of the 34th International Conference on Software Engineering*, pp. 36–46, 2012.

[43] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers," in *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, pp. 131–140, 2009.

[44] X. Xia, D. Lo, X. W. X, and B. Zhou, "Accurate developer recommendation for bug resolution," in *Proceedings of the 20th Working Conference on Reverse Engineering*, pp. 72–81, 2013.

[45] J. Xuan, H. Jiang, Z. Ren, J. Yan, and Z. Luo, "Automatic bug triage using semi-supervised text classification," in *Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering*, pp. 209–214, 2010.

[46] S. Ahsan, J. Ferzund, and F. Wotawa, "Automatic software bug triage system (BTS) based on latent semantic indexing and support vector machine," in *Proceedings of the 4th International Conference on Software Engineering Advances*, pp. 216–221, 2009.

[47] Z. Lin, F. Shu, Y. Yang, C. Hu, and Q. Wang, "An empirical study on bug assignment automation using Chinese bug data," in *Proceedings of the 3rd international symposium on empirical Software Engineering and Measurement*, pp. 451–455, 2009.

[48] G. Murphy and D. Cubranic, "Automatic bug triage using text categorization," in *Proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering*, pp. 92–97, 2004.

[49] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the ACM SIGSOFT symposium on the foundations of software engineering*, pp. 111–120, 2009.

[50] F. Gers and J. Schmidhuber, "Recurrent nets that time and count," in *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks*, pp. 189–194, 2000.

[51] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pp. 1724–1734, 2014.

[52] C. S. Gebizli and H. Sozer, "Model-based software product line testing by coupling feature models with hierarchical markov chain usage models," in *roceedings of the 6th IEEE International Workshop on Model-Based Verification and Validation*, pp. 278–283, 2016.

[53] R. Zhu, *Improving Software Defect Assignment Accuracy with the LSTM and Rule Engine Model*. PhD thesis, Seidenberg School of Computer Science and Information Systems, Pace University, New York, NY, United States, 2019.

[54] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers," in *Proceedings of the 6th International Conference on Mining Software Repositories*, pp. 131–140, 2009.