

# A Supervised Generative Topic Model to Predict Bug-fixing Time on Open Source Software Projects

Pasquale Ardimento<sup>a</sup> and Nicola Boffoli<sup>b</sup>

Department of Informatics, University of Bari Aldo Moro, Via Orabona 4, Bari, Italy


**Keywords:** SLDA, Latent Topics, Bug-fixing, Repository Mining, Software Maintenance, Text Categorization, SLDA.


**Abstract:** During software maintenance activities an accurate prediction of the bug-fixing time can support software managers to better resources and time allocation. In this work, each bug report is endowed with a response variable (bug-fixing time), external to its words, that we are interested in predicting. To analyze the bug reports collections, we used a supervised Latent Dirichlet Allocation (sLDA), whose goal is to infer latent topics that are predictive of the response. The bug reports and the responses are jointly modeled, to find latent topics that will best predict the response variables for future unlabeled bug reports. With a fitted model in hand, we can infer the topic structure of an unlabeled bug report and then form a prediction of its response. sLDA adds to LDA a response variable connected to each bug report. Two different variants of the bag-of-words (BoW) model are used as baseline discriminative algorithms and also an unsupervised LDA is considered. To evaluate the proposed approach the defect tracking dataset of LiveCode, a well-known and large dataset, was used. Results show that SLDA improves recall of the predicted bug-fixing times compared to other BoW single topic or multi-topic supervised algorithms.

## 1 INTRODUCTION

“Most defects end up costing more than it would have cost to prevent them. Defects are expensive when they occur, both the direct costs of fixing the defects and the indirect costs because of damaged relationships, lost business, and lost development time” (Beck and Andres, 2005). The later a bug is found, the later it will be fixed and more time and resources will be wasted in the present and the form of opportunities lost in the future. For these reasons, an accurate bug-fixing time prediction can help software managers to better allocate resources and time. Software projects usually use bug tracking systems (BTS) to store and manage bug reports. A bug report contains a large amount of text information that can help software developers and maintainers understand bugs well and complete bug fixing. Many open-source software projects use BTS. Day by day, the number of bug reports submitted to BTS increases and, at the same time, the knowledge stored by BTS (Sun et al., 2017; Alenezi et al., 2018; Ardimento and Dinapoli, 2017). For Mozilla, by February 2022, the to-

tal number of bug reports was just under 1.8 million. In recent years, lots of work utilized information retrieval technology to explore these massive bug repositories to help developers understand, localize and fix bugs (Mohsin and Shi, 2021; Hamdy and El-Laithy, 2020; Meng et al., 2021). Most of the predictive models proposed to automatically predict bug-fixing time are based on traditional machine learning techniques. The basic idea is to extract text information by bug reports attributes, discarding only those that are empty or contain numerical values. Once a bug report is open (not re-open), it is categorized to a discretized time to resolution, based on the learned prediction model. This work is focused on an idea originally proposed in (Ardimento et al., 2016) and then applied and further elaborated in (Ardimento and Dinapoli, 2017; Ardimento et al., 2020). This idea involves using all the text attributes of a bug report and to treat the problem of bug-fixing time estimation as a text categorization problem. In this paper, conversely from previous experiences, two different variants of the bag-of-words (BoW) model are used as baseline discriminative algorithms. Then, supervised and unsupervised dimensionality reduction techniques are used on a corpus of textual bug reports extracted from Bugzilla. In standard unsupervised Latent Dirichlet

<sup>a</sup>  <https://orcid.org/0000-0001-6134-2993>

<sup>b</sup>  <https://orcid.org/0000-0001-9899-6747>

Allocation (LDA) each word in a document is generated from a Multinomial distribution conditioned on its own topic, which is a probability distribution over a fixed vocabulary representing a particular latent semantic theme (Blei, 2012). Once hyperparameters of the model are learned, posterior topic proportion estimates can be interpreted as a low dimensional summarization of the textual bug reporting. A linear classifier trained with a SVM with soft margins can then be used to learn discretized bug-fixing time class, using such posterior estimates as a reduced feature vector. It is also considered a supervised Latent Dirichlet Allocation (SLDA) generative model (Blei and McAuliffe, 2010), which adds the discriminative signal into the standard generative model of LDA. The response variable is an unordered binary target variable, denoting time to resolution discretized into FAST (negative class) and SLOW (positive class) class labels. The proposed models have been evaluated on one large-scale open-source project. Results show that SLDA improves recall. Due to the imbalance between the number of positive and negative cases and the importance assumed by the SLOW class for the prediction task, this measure is more useful than simply measuring the accuracy of classifiers. The rest of the paper is structured as follows. Section 2 briefly presents the lifecycle of bugs in a BTS while section 3 discusses the main related works of the bug-fixing time prediction problem. Section 4 describes the proposed prediction model and its main phases. Section 5 presents the empirical study and the results obtained, while section 6 discusses the main threats to the validity of the proposed model. Finally, section 7 draws the conclusions and sketches the future work.

## 2 BACKGROUND

The bug life cycle, also called defect workflow, is a process in which a defect goes through different stages in its entire life. These stages usually are: UNCONFIRMED, NEW, ASSIGNED, RESOLVED, VERIFIED, REOPENED, CLOSED. There are many popular implementations of a defect workflow such as Jira, Bugzilla, FogBugz, The Bug Genie, MantisBt, Request Tracker, and so on. The choice fell on Bugzilla as a data source for the following reasons:

- there is a consistent number of public Bugzilla installations, where a public installation is "one whose front or login page can be accessed and viewed over the Internet" (bugzilla.org, 2022). These installations permit anonymous users to browse and access bug information; on February 2021, Bugzilla Web official site lists 141 or-

ganizations and organizations that are running Bugzilla public installations (bugzilla.org, 2022);

- Bugzilla works for both free and open-source software and proprietary projects. In this way, it is possible to evaluate the feasibility of the proposed approach both in open source software and in commercial software;
- Bugzilla offers a native well-documented REST API (Bugzilla, 2022) to extract and put information from its installations to external programs and vice versa;
- Existing different implementations of the defect workflow can be considered equivalent to the one present in Bugzilla. In particular, a Bugzilla bug that is VERIFIED and FIXED represents a CLOSED bug.

## 3 RELATED WORK

Several scientific papers dealt with the bug fixing problem. A semi-automatic approach based on machine learning techniques to decrease the triaging time was presented in (Anvik et al., 2006). This method is based on learning the type of reported bug that can be fixed by a developer. The received bug is reported to this model and a list with a minimum number of developers who can fix the bug is reported as the output. They received up to 75 and 64% precision in Eclipse and Firefox projects, respectively. In their method, the text of 'Summary' and 'Description' fields are categorized and the name of developers who can fix the bug is reported as the output. The features are derived by applying information retrieval methods and the classification is completed through Naïve Bayes (NB), support vector machine (SVM), and C4.5 methods.

A binary prediction model was presented in (Giger et al., 2010), based on a Decision Tree algorithm, using both unchangeable fields, such as 'Reporter', and fields that can be changed over time, such as 'Severity'. They evaluated the proposed model on six of the systems of three open-source projects: Eclipse, Mozilla, and Gnome. Results obtained show a 0.65 precision and 0.62 recall performance in Eclipse. They also state that the inclusion of post-submission bug report data of up to one month promises to improve prediction models.

An analysis on how to calculate the time necessary to fix a bug is provided in (Kim and Whitehead, 2006). In this work, the authors identified, in ArgoUML and PostgreSQL projects, when the bugs are introduced and when the bugs are fixed.

Authors in (Marks et al., 2011) applied the Random Forest algorithm to predict bug-fixing time in Mozilla and Eclipse projects using a selected subset of bug reports fields. They obtained a 65% precision in classification. They also carried out a sensitivity analysis for parameters of both projects finding that bug report time and its location are the most important features in predicting the bug fix in Mozilla Project, while in Eclipse, the severity feature is more important. Priority, instead, is not very important in both projects.

Panjer (Panjer, 2007) proposed a method where they discretized the fix-time values using an equal frequency binning algorithm and identified seven classes for bug fix-time. The binary classification was carried out by applying the 0-R, 1-R, C4.5 DT, NB, LR algorithms, and 34% was obtained as the maximum value of precision.

A Markov-based model to predict the number of bugs that will be fixed in three consecutive months was proposed in (Zhang et al., 2013). The historical bug reports were used to predict the time necessary to fix the bug by applying the Monte Carlo Algorithm. Besides, a KNN-based classification model was proposed obtaining 72.45% as F-measure.

Bidirectional Encoder Representation from Transformers (BERT) was used to predict the fixing-time of a bug as slow and fast by (Ardimento and Mele, 2020). Even if "BERT makes use of a self-attention mechanism that allows learning the bidirectional context representation of a word in a sentence, which constitutes one of the main advantages over the previously proposed solutions" this method requires to be further applied and investigated before to be considered as mature to provide a significant contribution to face the bug-fixing prediction problem.

## 4 PROPOSED MODEL

The proposed prediction model consists of three sequential phases (shown in figure 1): Data Collection, Pre-processing, Learning and evaluation of bug-fixing time Prediction.

### 4.1 Data Collection

The Data Collection activity consists in gathering the bug reports from any BTS as the proposed model is BTS independent. The first step consists of selecting only those bug reports that have already been fixed, by one or more programmers, and successfully verified, by one or more testers. These bugs are the only ones useful to, first, train the classifier and, then, to

predict the newly unseen bug reports. After selecting this subset, for each bug in it, only the attributes sensible for both tasks were extracted, while residual attributes were discarded. In particular, the attributes containing only numeric or empty values in most cases were discarded. Since the bug-fixing time is rarely publicly available in a BTS, it also happens in Bugzilla installations, it was calculated as the number of days elapsed between the date where bug attribute Status was set to RESOLVED and the date on which the bug report was assigned for the first time. This measure, called Days Resolution, represents the bug-fixing time, from here on out. Since the Days Resolution measure does not represent the effective time spent to fix a bug it could affect the outcomes of this research. After completing data gathering and attribute extraction, the dataset was split into a training, test, and validation dataset using a fixed split percentage. This operation takes place after extraction attributes because all post-submission information has to filter out from both test and validation sets. Test and validation instances simulate newly opened and previously unseen bugs and, therefore, attributes that were not available before the bug was assigned must be removed from the information set. To this aim, it is necessary to retrieve the changes history of a bug. The history permits to know when and who assigned a value to the attributes. All the attributes whose values are not filled by the bug reporter are discarded. Severity and priority attributes represent an exception because their values can be filled by the bug reporter and, then, changed. For these attributes, only the initial value, if present, filled by the bug reporter is considered and not discarded. Finally, the model discretizes the bug-fixing times into two classes, conventionally labeled as SLOW and FAST. The SLOW label indicates a discretized bug-fixing time above some fixed threshold in the right-tail of the empirical bug-fixing time distribution. It is assumed that SLOW indicates the positive class, thus SLOW being the target class of the prediction model. Therefore, increasing the number of true positives for the positive class cause an overestimation of bug-fixing times. This situation is preferred to underestimation because it involves a less dangerous error.

### 4.2 Pre-processing and Learning Models

Before applying any probabilistic model is necessary to bring bug reports in a form that is analyzable for the classification task. The steps followed in this paper are the most commonly used in the literature on text classification and Natural Language Processing

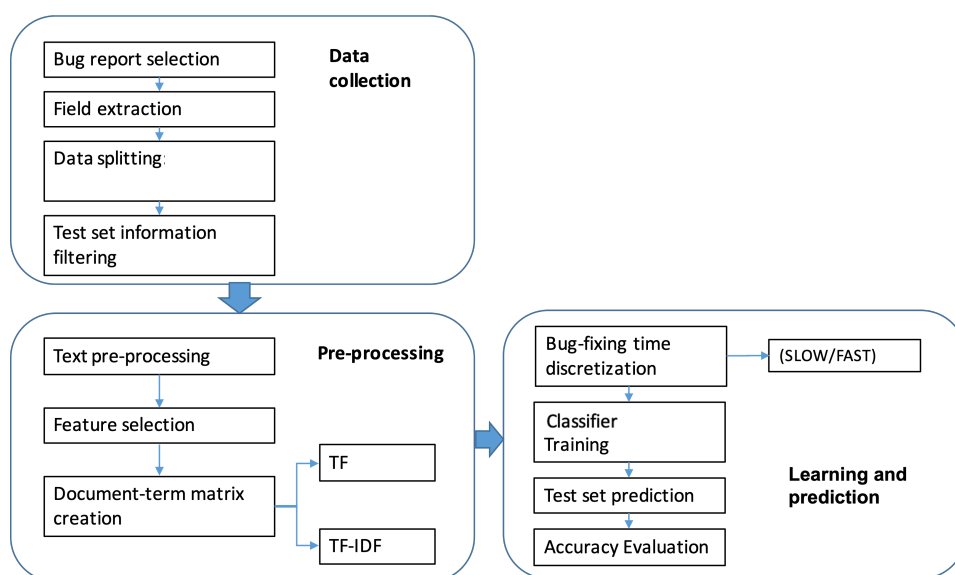


Figure 1: Proposed Prediction Model.

(Mogotsi, 2010). The objective is to determine a vocabulary of terms  $V$ , by:

- eliminating those words that occur commonly across all the documents in the corpus which means these words are not very discriminative, and focusing on the important words instead (stop word removal);
- reducing inflectional forms to a common base form and then heuristically complete stemmed words by taking the most frequent match as completion (stemming and stem completion);
- using a specific algorithm, proposed in (Ardimento et al., 2016), for detecting multi-word expressions. This algorithm consists of a suitable modification of a simple heuristic first introduced in (Justeson and Katz, 1995).
- reducing dimensionality by term space reduction as it may even result in a moderate increase in predictive accuracy depending "on the classifier used, on the aggressivity of the reduction, and on the term space reduction used." (Sebastiani, 2002). In the proposed approach, terms were sorted (from best to worst) according to their estimated normalized expected mutual information (NEMI, (Mogotsi, 2010)) with the discretized time to resolution (SLOW/FAST). Only terms showing a NEMI value greater than the overall average NEMI were included. In this way, the terms almost approximately independent with the target class label were omitted.

### 4.3 Baseline Model

Two different variants of the bag-of-words (BoW) model were used to be baseline algorithms. The BoW model is high-dimensional, as it represents a document with one dimension per word. Specifically, it was first considered a multivariate Bernoulli (MB) model with smoothing Laplace parameter  $\lambda$ , applied to the binary term-document incidence matrix (Mogotsi, 2010). For each word in  $V$  and independently of each other (Naive Bayes assumption) the MB model learns the probabilities  $\pi_{ty} = P(e_t = \frac{1}{y})$  that the word represented by  $e_t = (1, 2, \dots, |V|)$  will occur at least once in any position in any one of the documents of class  $y$  (i.e.  $y \equiv \frac{SLOW}{FAST}$ , with  $SLOW \equiv 1$ ). The second alternative was a linear classifier using unnormalized  $tf_{td}$  counts (for token  $t$  and document  $d$ ), trained with a support vector machine (SVM) with soft-margin classification, ruled by a cost-parameter  $C$ . Inverse document frequency weighting ( $tf-idf_{td}$ ) was also used.

### 4.4 Unsupervised Latent Dirichlet Allocation (LDA)

The most important thing to know about a document is the underlying semantic themes rather than words. Latent Dirichlet Allocation (LDA) is a generative model of a textual corpus  $D$ , that has the potential to discover topics during the training phase (Blei et al., 2003; Blei, 2012). Formally, a topic is the probability distribution over the elements of  $V$ . For each document we have  $K$  underlying semantic themes (topics),



$B_{1:K} = (B_1, \dots, B_K)$ , where each  $B_K (K = 1, \dots, K)$  is a  $|V|$ -dimensional vector of probabilities over the elements of  $V$ . We also have a vector of  $K$ -dimensional vectors  $Z_{d_{1:N_d}} = (z_{d,1}, \dots, z_{d,N_d})$  of topic assignments, where  $Z_{d,n}$  is the topic assignment for the  $n$ th word ( $n=1, \dots, N_d$ ) in document  $d$ . The indicator  $z$  of the  $k$ -th topic is represented as a  $K$ -dimensional unit-basis vector such that  $z^k = 1$  and  $z^j = 0$  for  $j \neq k$ . Similarly, words in a document  $W_{d_{1:N_d}} = (w_{d,1}, \dots, w_{d,N_d})$  are represented using superscripts to denote components, that is the  $n$ th word in  $|V|$  is represented as a unit-basis vector  $w$  in  $R^V$ , such that  $w^v = 1$  and  $w^v = 0$  for  $v \neq v$ . LDA can be summarized as the following generative procedure (independently over  $d$  and  $n$ ):

$$\theta_d \sim \text{Dirichlet}_K(\alpha) \text{ for } d \in D \quad (1)$$

$$z_{d,n} | \theta_d \sim \text{Multinomial}_k(\theta_d) \quad (2)$$

for  $D \in D$  and  $n \in \{1, \dots, N_d\}$

$$\omega_{d,n} | z_{d,n}, \beta_{1:K} \sim \text{Multinomial}_{|V|}(\beta_{z_{d,n}}) \quad (3)$$

for  $d \in D$  and  $n \in \{1, \dots, N_d\}$

where  $B_{z_{d,n}} \equiv B_j$  if  $z_{d,n}^j = 1$ . Parameters  $\alpha$  and  $\beta_{1:K}$  are treated as unknown hyper-parameters to be estimated, rather than random variables. In this way, documents are seen as a realization of a stochastic process, which is then reversed by standard machine learning techniques that return maximum-a-posteriori estimates of model parameters for each document (bug)  $d$ , given the value of hyper parameters  $\alpha$  and  $\beta$ . Posterior estimates of per-document topic proportions  $\theta_d$  can be interpreted as a low-dimensional summarization of the document. As LDA is an unsupervised algorithm, we used a linear classifier trained with an SVM with soft margins to learn bug-fixing time class  $y$ , using posterior  $\theta_d$  estimates as a reduced feature vector.

Unfortunately, exact inference in the LDA model is NP-hard for a large number of topics (Sontag and Roy, 2011), and posterior distribution of latent variables  $p(Z_{1:D}, \theta_{1:D} | \omega_{1:D}, \alpha, \beta_{1:K})$  has not a closed-form (here, per-word topic assignments  $z_{d:1:N_d}$  are collected into the  $z_{1:D}$  vector as  $d$  varies over  $1, \dots, D$ , with  $D = |D|$ , and the same definition applies to  $\theta_{1:D}$  and  $\omega_{1:D}$ ). In fact, the marginal likelihood is intractable and cannot be evaluated exactly in a reasonable time for all possible per-word topic assignments  $z_{d,n}$ . Consequently, we used a mean-field variational Bayes algorithm (VB) for approximate posterior inference (Blei et al., 2016). Because the number of topics  $K$  is in general not known, models with several different numbers of topics were fitted and the optimal number was determined in a data-driven way by minimizing the geometric mean per-word likelihood (perplexity)

on test documents (Wallach et al., 2009). Since the topic assignments for one document are independent of the topic assignments for all other documents, each test document can be evaluated separately. Once the hyper parameters  $\alpha$  and  $\beta_{1:K}$  are learned and  $K$  has been set, inference can be performed to compute a posterior  $\theta_d$  vector for each test document (as well as for each document in the validation set), to obtain the reduced representation in topic space, which is subsequently used to predict bug bug-fixing time  $y_d$  in a supervised fashion.

## 4.5 Supervised LDA

Supervised Latent Dirichlet Allocation (SLDA) adds the discriminative signal into the standard generative model of LDA. For each document  $d$ , response label  $y_d$  ( $y_d \in \text{SLOW/FAST}$ , with  $\text{SLOW} \equiv 1$ ) is sampled conditionally on the topic assignments:

$$y_d | Z_{d,1:N_d}, \eta \sim \text{Bernoulli}\left(\frac{\exp(\eta^T \bar{z}_d)}{1 + \exp(\eta^T \bar{z}_d)}\right) \quad (4)$$

where  $\bar{z}_d = \frac{1}{N_d} \sum_{n=1}^{N_d} z_{d,n}$  is the vector of empirical topic frequencies in document  $d$ . As previously said, parameters  $\alpha$ ;  $\beta_{1:K}$  and  $\eta$  are treated as unknown hyper parameters to be estimated, rather than random variables. Documents are generated under full word exchangeability, and then topics are used to explain the response. Other specifications are indeed possible, for example  $y_d$  can be regressed as a nonlinear function of topic proportions  $\theta_d$ , but (Blei and McAuliffe, 2007) claim that the predictive performance degrades as a consequence of the fact the topic probabilistic mass does not directly predict discretized bug-fix times.

Also in this case posterior inference of latent model variables is not feasible, as the conditional posterior distribution  $p(Z_{1:D}, \theta_{1:D} | \omega_{1:D}, \alpha, \beta_{1:K})$  has not a closed-form. Consequently, a variational Bayes (VB) parameter under a mean-field approximation was used. The best number of topics  $K$  was optimized through the test set (further details are in Section 5).

## 5 EXPERIMENT AND RESULTS

Bug report textual information was extracted from the Bugzilla repository of a large open-source software project: LiveCode. Data were automatically collected using software routines written in PHP/JavaScript/Ajax. Raw textual reports were pre-processed and analyzed using the R 4.0.5 software system (RProject, 2022). Five thousand bugs were randomly drawn and randomly divided into training,

Table 1: Results of best configurations for accuracy.

Test set - Best accuracy	Algorithm	Param.	TP	FN	FP	TN	Acc.	Prec.	Recall
<b>0,58</b>	NB	$\lambda = 0$	0	125	88	287	0,57	0	0
<b>0,55</b>	SVM	$C=0,01$	0	125	88	287	0,57	0	0
<b>0,61</b>	LDA+SVM	$C=0,01$	15	110	80	295	0,62	0,16	0,12
<b>0,79</b>	SLDA	$K=2$	112	13	206	169	0,56	0,35	0,89

test and validation subsets, using a 70:20:10 split ratio. Bug-fixing time was categorized into FAST and SLOW labels according to the third quartile,  $q_{0.75}$ , of the empirical distribution of bug resolution times in the training set. Accuracy, precision and recall were used to assess the performance of bug-fixing time prediction models. Accuracy indicates the proportion of correctly predicted bugs:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \quad (5)$$

where TP (True Positive) denotes the number of bugs correctly predicted as SLOW; FP (False Positive) denotes the number of bugs incorrectly predicted as SLOW, TN (True Negative) denotes the number of bugs reports correctly predicted as FAST, FN (False Negative) denotes the numbers of bugs incorrectly predicted as FAST.

Precision, instead, indicates the proportion of correctly predicted SLOW bugs:

$$Precision = \frac{TP}{TP + FP} \quad (6)$$

Recall denotes the proportion of true positives of all SLOW bugs:

$$Recall = \frac{TP}{TP + FN} \quad (7)$$

The trained set was used to train the classification models, reported in the list below. Then the specific free parameters of each model were assessed over the test set and, finally, performances measures were recalculated over the validation set using the best accuracy values obtained from the test set.

- Multivariate Bernoulli (MB), with either no smoothing, or Laplace  $\lambda$  varying in  $\{1, 2, 3\}$  (Mogotsi, 2010).
- Support Vector Machine (SVM), with soft-margin classification and cost parameter  $C$  varying in  $\{0.1, 10, 100\}$ .
- Latent Dirichlet Allocation plus Support Vector Machine (LDA+SVM), with  $K$  varying in  $\{2, 5, 10, 15, 20, 25, 30, 35, 40, 50\}$ , soft-margin classification and cost parameter  $C$  varying in  $\{0.1, 1, 10, 100\}$ .
- Supervised Latent Dirichlet Allocation (SLDA), with  $K$  varying in

$\{1, 2, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50\}$ . With  $K=1$  it obtains a single-topic model.

Table 1 presents the results obtained, it shows that SLDA does not achieve the best value for accuracy. However, it is important to note that accuracy is a misleading measure for imbalanced class distributions. In the case of our experimentation, the two classes are not equally distributed. Furthermore, it is evident that the minority class (SLOW) has more importance in the context of software maintenance, because of its larger impact in terms of cost/effectiveness. Consequently, if it is assumed that a higher recall (proportion of true positives of all SLOW bugs) is a more sensible target, the SLDA model has the best performance. It is also worth noting that a larger number of FAST bugs are classified as SLOW under the SLDA model. However, costs incurred in false-positive are generally very low. Conversely, NB, SVM and LDA+SVM classifiers showed higher accuracies, but they failed to correctly predict most of SLOW bugs. Overall, these results clearly show that the use of a supervised topic model highly improves the recall of bug-fix time prediction.

## 6 THREATS

The following threats have to be considered to judge the quality of our research.

- Dataset. The experiment was carried out on just one data set extracted from LiveCode. This project is not representative of the population of all open-source software.
- Non-open-source projects. There is no guarantee that the proposed model is effective for non-open-source projects. Typically, in fact, in proprietary projects a specific group is responsible for fixing given bugs based on corresponding features.
- Outliers and noises. Dataset could contain bug reports whose fixing time is not fit with other bug reports because, for example, it is extremely long. These bug reports, called outliers, should be removed to both improve the quality of the data and, maybe, generate a positive impact on the accuracy of the predictions. Moreover, a possible noise in the dataset could be represented by a bug

CLOSED more than one time. BTS, in fact, permits that a CLOSED bug could be reopened at any time. When this happens the calculation of the time resolution is not more coherent with the time necessary to fix a bug. Another threat in our experiment concerns the so-called problem of "proportion of inconsistent samples". Two bug reports are inconsistent if they have the same characteristics but, one is marked as positive class and the other is marked as negative class. A possible way to tackle this problem is reported in (Du et al., 2022), where the authors aim to reduce, by incorporating the activity information and time information of bug activity transfer, the proportion of inconsistent samples. This method of bug feature extraction and model construction is based on a LSTM-based deep learning model which can not only achieve sequence prediction, but also learn sequences interaction through the attention mechanism to improve prediction accuracy. The results are promising even if they are obtained on only one open data set Firefox.

- Bug-fixing time. A strong assumption about bug-fixing time was made. The actual amount of time spent by developers and the distribution in terms of hours per day to fix a bug are not publicly declared on Bugzilla. It is assumed, therefore, a uniform distribution of developers' work and calculated effort spent in calendar days. These assumptions hide and ignore the real efforts made by the developers to complete the fixing work.
- Number of developers involved. The number of developers involved in fixing a bug is not publicly known. It represents another limitation in calculating the bug-fixing time. For example, if different bugs have the same calendar days value, it does not imply that the time spent to fix them is the same.

## 7 CONCLUSION

This paper proposes an automatic binary bug-fixing time prediction model based on SLDA, a supervised generative topic model. This model significantly improves recall compared to other BoW single topic or multi-topic supervised algorithms. Accuracy, instead, is lightly lower and, considering all the models used, not so good. These results are aligned with the need to maximize recall to detect SLOW classes as much as possible and the acceptance that the negative class (FAST) plays a secondary role.

To improve the results obtained and to increase the external validity of the proposed approach, future

work will require: (i) to carry out a stability analysis to avoid that results depend on some purely random factor (in the experimentation carried out, if topic models were re-run with a different random seed, the topics will often change); (ii) to carry out a sensitivity analysis of quantiles to measure the impact of the inputs over the  $\alpha$ -quantile of the output distribution. (iii) to discover similarities and differences between open and non-open-source software projects; (iv) to tackle the problem of "proportion of inconsistent sample" to reduce the presence of inconsistent bugs.

## REFERENCES

- Alenezi, M., Banitaan, S., and Zarour, M. (2018). Using categorical features in mining bug tracking systems to assign bug reports. *CoRR*, abs/1804.07803.
- Anvik, J., Hiew, L., and Murphy, G. C. (2006). Who should fix this bug? In Osterweil, L. J., Rombach, H. D., and Soffa, M. L., editors, *28th International Conference on Software Engineering (ICSE 2006)*, Shanghai, China, May 20-28, 2006, pages 361–370. ACM.
- Ardimento, P., Bilancia, M., and Monopoli, S. (2016). Predicting bug-fix time: Using standard versus topic-based text categorization techniques. In Calders, T., Ceci, M., and Malerba, D., editors, *Discovery Science - 19th International Conference, DS 2016, Bari, Italy, October 19-21, 2016, Proceedings*, volume 9956 of *Lecture Notes in Computer Science*, pages 167–182.
- Ardimento, P., Boffoli, N., and Mele, C. (2020). A text-based regression approach to predict bug-fix time. In Appice, A., Ceci, M., Loglisci, C., Manco, G., Masciari, E., and Ras, Z. W., editors, *Complex Pattern Mining - New Challenges, Methods and Applications*, volume 880 of *Studies in Computational Intelligence*, pages 63–83. Springer.
- Ardimento, P. and Dinapoli, A. (2017). Knowledge extraction from on-line open source bug tracking systems to predict bug-fixing time. In *Proceedings of the 7th International Conference on Web Intelligence, Mining and Semantics, WIMS '17*, New York, NY, USA. Association for Computing Machinery.
- Ardimento, P. and Mele, C. (2020). Using BERT to predict bug-fixing time. In *2020 IEEE Conference on Evolving and Adaptive Intelligent Systems, EAIS 2020, Bari, Italy, May 27-29, 2020*, pages 1–7. IEEE.
- Beck, K. L. and Andres, C. (2005). *Extreme programming explained - embrace change, Second Edition*. The XP series. Addison-Wesley.
- Blei, D. M. (2012). Probabilistic topic models. *Commun. ACM*, 55(4):77–84.
- Blei, D. M., Kucukelbir, A., and McAuliffe, J. D. (2016). Variational inference: A review for statisticians. *CoRR*, abs/1601.00670.
- Blei, D. M. and McAuliffe, J. D. (2007). Supervised topic models. In Platt, J. C., Koller, D., Singer, Y., and Roweis, S. T., editors, *Advances in Neural Information Processing Systems 20, Proceedings*

- of the Twenty-First Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 3-6, 2007, pages 121–128. Curran Associates, Inc.
- Blei, D. M. and McAuliffe, J. D. (2010). Supervised topic models.
- Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022.
- Bugzilla (2022). Webservice api bugzilla. [Online; accessed 16-February-2022].
- bugzilla.org (2022). Bugzilla installation list. [Online; accessed 16-February-2022].
- Du, J., Ren, X., Li, H., Jiang, F., and Yu, X. (2022). Prediction of bug-fixing time based on distinguishable sequences fusion in open source software. *Journal of Software: Evolution and Process*, n/a(n/a):e2443.
- Giger, E., Pinzger, M., and Gall, H. C. (2010). Predicting the fix time of bugs. In Holmes, R., Robillard, M. P., Walker, R. J., and Zimmermann, T., editors, *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering, RSSE 2010, Cape Town, South Africa, May 4, 2010*, pages 52–56. ACM.
- Hamdy, A. and El-Laithy, A. (2020). Semantic categorization of software bug repositories for severity assignment automation. In Jarzabek, S., Ponsiszewska-Maranda, A., and Madeyski, L., editors, *Integrating Research and Practice in Software Engineering*, volume 851 of *Studies in Computational Intelligence*, pages 15–30. Springer.
- Justeson, J. S. and Katz, S. M. (1995). Technical terminology: some linguistic properties and an algorithm for identification in text. *Nat. Lang. Eng.*, 1(1):9–27.
- Kim, S. and Whitehead, E. J. (2006). How long did it take to fix bugs? In *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR '06*, page 173–174, New York, NY, USA. Association for Computing Machinery.
- Marks, L., Zou, Y., and Hassan, A. E. (2011). Studying the fix-time for bugs in large open source projects. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, pages 1–8.
- Meng, D., Guerriero, M., Machiry, A., Aghakhani, H., Bose, P., Continella, A., Kruegel, C., and Vigna, G. (2021). Bran: Reduce vulnerability search space in large open source repositories by learning bug symptoms. In Cao, J., Au, M. H., Lin, Z., and Yung, M., editors, *ASIA CCS '21: ACM Asia Conference on Computer and Communications Security, Virtual Event, Hong Kong, June 7-11, 2021*, pages 731–743. ACM.
- Mogotsi, I. C. (2010). Christopher d. manning, prabhakar raghavan, and hinrich schütze: Introduction to information retrieval - cambridge university press, cambridge, england, 2008, 482 pp, ISBN: 978-0-521-86571-5. *Inf. Retr.*, 13(2):192–195.
- Mohsin, H. and Shi, C. (2021). SPBC: A self-paced learning model for bug classification from historical repositories of open-source software. *Expert Syst. Appl.*, 167:113808.
- Panjer, L. D. (2007). Predicting eclipse bug lifetimes. In *Proceedings of the Fourth International Workshop on Mining Software Repositories, MSR '07*, page 29, USA. IEEE Computer Society.
- RProject (2022). The r project for statistical computing. [Online; accessed 16-February-2022].
- Sebastiani, F. (2002). Machine learning in automated text categorization. *ACM Comput. Surv.*, 34(1):1–47.
- Sontag, D. A. and Roy, D. M. (2011). Complexity of inference in latent dirichlet allocation. In Shawe-Taylor, J., Zemel, R. S., Bartlett, P. L., Pereira, F. C. N., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain*, pages 1008–1016.
- Sun, X., Zhou, T., Li, G., Hu, J., Yang, H., and Li, B. (2017). An empirical study on real bugs for machine learning programs. In Lv, J., Zhang, H. J., Hinchey, M., and Liu, X., editors, *24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4-8, 2017*, pages 348–357. IEEE Computer Society.
- Wallach, H. M., Murray, I., Salakhutdinov, R., and Mimno, D. M. (2009). Evaluation methods for topic models. In Danyluk, A. P., Bottou, L., and Littman, M. L., editors, *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, volume 382 of *ACM International Conference Proceeding Series*, pages 1105–1112. ACM.
- Zhang, H., Gong, L., and Versteeg, S. (2013). Predicting bug-fixing time: an empirical study of commercial software projects. In Notkin, D., Cheng, B. H. C., and Pohl, K., editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 1042–1051. IEEE Computer Society.