Predicting Bug-Fix Time: Using Standard Versus Topic-Based Text Categorization Techniques

Pasquale Ardimento[1](✉), Massimo Bilancia[2], and Stefano Monopoli[3]

[1] Department of Informatics, University of Bari Aldo Moro, Via Orabona, 4, 70125 Bari, Italy
pasquale.ardimento@uniba.it

[2] Ionian Department of Law, Economics and Environment, University of Bari Aldo Moro, Via Lago Maggiore angolo Via Ancona, 74121 Taranto, Italy
massimo.bilancia@uniba.it

[3] Everis Italia S.p.A., Via Gustavo Fara, 26, 20124 Milano, Italy
stefano.monopoli@everis.com

Abstract. In modern software development, finding and fixing bugs is a vital part of software development and quality assurance. Once a bug is reported, it is typically recorded in the Bug Tracking System, and is assigned to a developer to resolve (bug triage). Current practice of bug triage is largely a manual collaborative process, which is often time-consuming and error-prone. Predicting on the basis of past data the time to fix a newly-reported bug has been shown to be an important target to support the whole triage process. Many researchers have, therefore, proposed methods for automated bug-fix time prediction, largely based on statistical prediction models exploiting the attributes of bug reports. However, existing algorithms often fail to validate on multiple large projects widely-used in bug studies, mostly as a consequence of inappropriate attribute selection [2]. In this paper, instead of focusing on attribute subset selection, we explore an alternative promising approach consisting of using all available textual information. The problem of bug-fix time estimation is then mapped to a text categorization problem. We consider a multi-topic Supervised Latent Dirichlet Allocation (SLDA) model, which adds to Latent Dirichlet Allocation a response variable consisting of an unordered binary target variable, denoting time to resolution discretized into FAST (negative class) and SLOW (positive class) labels. We have evaluated SLDA on four large-scale open source projects. We show that the proposed model greatly improves recall, when compared to standard single topic algorithms.

Keywords: Bug triage · Bug-fix time prediction · Text categorization · Supervised topic models · Supervised Latent Dirichlet Allocation (SLDA)

The authors equally contributed to this paper.

T. Calders et al. (Eds.): DS 2016, LNAI 9956, pp. 167–182, 2016.
DOI: 10.1007/978-3-319-46307-0_11

1 Introduction

In recent years, with the increasing complexity of software systems, the task of software quality assurance has become progressively more challenging. In modern software development, software repositories are specialized database storing the output of the development process. The large-scale and partially unstructured data stored in these facilities are often not fully suited to traditional analysis methods [27]. A major role is played by finding and fixing bugs, which is a vital part of software development and quality assurance to such an extent that it has been estimated that software companies spend over 45 percent of their costs in fixing bugs [21,27]. The largest and most complex software projects are most often supported by a specialized database known as Bug Tracking System (BTS), used by quality assurance personnel and programmers to keep track of software problems and resolutions. Once a bug is reported, it is typically recorded in the BTS, and is assigned to a developer to resolve (bug triage). Current practice of bug triage is largely a manual collaborative process, in that the triager first examines whether a bug report contains sufficient or duplicated informations, then she/he confirms the bug and sets severity and priority, and finally decides who has the expertise in resolving it. Any such process is expected to be costly and inaccurate when the number of bug reports is large. For example, [12] report that empirical studies on Eclipse and Mozilla show that 37 %–44 % of bugs have been re-assigned (tossed) at least once to another developer. As there are many bug reports requiring resolution and potentially many developers working on a large project, it is non-trivial to assign a bug report to the appropriate developers.

For the reasons above, predicting on the basis of past data the time to fix a newly-reported bug has been shown to be an important target to support the whole triage process (and, in particular, to make the assignment more effective), and hence to help project managers to better estimate software maintenance efforts and improve cost-effectiveness [29]. Broadly speaking, bug fix-time is defined as the calendar time from the triage of a bug to the time the bug is resolved and closed as fixed [19]. Many researchers have proposed methods for automated bug-fix time prediction, largely based on machine learning techniques. Most of existing approaches are building prediction models based on the attributes of bug reports. For example, [20] used a historical portion of the Eclipse Bugzilla database. The predictor variables consisted of selected fields included, at time of confirmation, in the textual description of bugs reports. Several data mining models were then built and tested, using a nominal target class based on discretized time to resolution on a logarithmic scale. A logistic regression classifier provided the best classification accuracy of 34.5 %. In a similar fashion, [11] combined the attributes of the initial bug report with post-submission information. Bugs reports in the training set were classified into fast and slowly fixed. Not surprisingly, they found that post-submission data of bug reports improved their prediction model based on decision tree analysis. See also [29], where further studies are reviewed in greater depth.

Despite these apparently positive findings, [2] showed how existing models fail to validate on multiple large projects widely-used in bug studies, indicating a poor predictive accuracy comprised between 30 % and 49 %. In addition, it was found that there was no correlation between the probability that a new bug will be fixed, bug-opener's reputation and the time it takes to fix a bug. These findings show that we must be able to identify attributes which are effective in predicting bug-fix time. On the other side, instead of focusing on attribute subset selection, an alternative promising approach consists of using all available textual information. The problem of bug-fix time estimation is then mapped to a text categorization problem. A new bug report is classified to a set of discretized time to resolution classes (discretized bug-fix time), based on a classifier which is trained using historical data.

Traditional text categorization techniques establish the relationship between a set of predefined categories and their respective documents by analyzing document contents. This class of models can be unified under the assumption that each textual bug report exhibits exactly one probability distribution over strings drawn from some vocabulary of terms [17]. This assumption is often too limiting to model a large collection of textual bug reports. In standard unsupervised Latent Dirichlet Allocation (LDA) each word in a document is generated from a Multinomial distribution conditioned on its own topic, which is a probability distribution over the terms in the vocabulary representing a particular underlying semantic theme [3]. However, as our main objective is to predict bug-fix time, we consider a supervised Latent Dirichlet Allocation (SLDA) model [4,28], which adds to LDA a response variable consisting of an unordered binary target variable, denoting time to resolution discretized into FAST (negative class) and SLOW (positive class) class labels. We have evaluated SLDA on four large-scale open source projects (see Sect. 4 for in-depth details). We show that the proposed model greatly improves recall, when compared to single topic algorithms.

The remainder of the paper is organized as follows. Section 2 describes how our model collects the data. Section 3 deals with the methods used in our prediction models. Section 4 presents the empirical study and the results. Section 5, finally, draws the conclusions.

2 Data Collection

Data collection is the first step of bug-fix time prediction process, whose overall conceptual design is shown in Fig. 1. Our design is largely application independent, albeit we will use the open source BTS Bugzilla for the actual implementation [7] (see also Sect. 4 for further details). First, data gathering consists of selecting only those historical bug records which are sensible to predict discretized time to resolution of previously unseen bugs. In other words, we select textual reports of resolved and closed bugs only, whose `Status` field has been assigned to VERIFIED, as well as `Resolution` field has been assigned to FIXED.

As we said before, our approach maps the prediction problem into a text categorization task. Hence, once the relevant textual bug reports have been selected, we consider the content of the following fields (which are first extracted and then re-collapsed into a single text identified by a unique `Id`, and used as input for

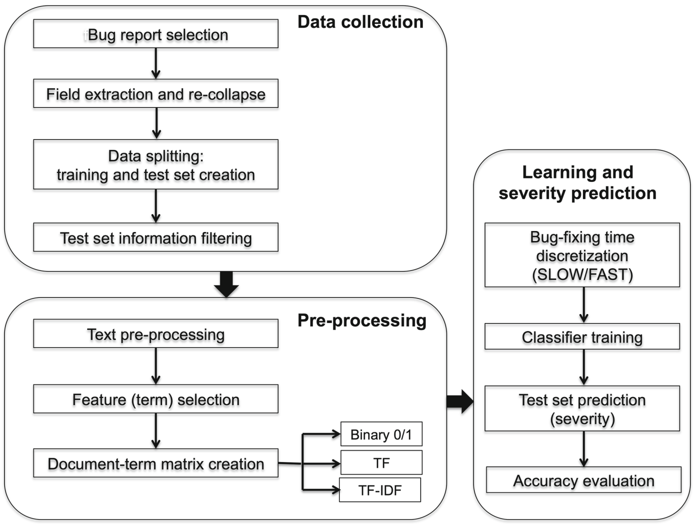
Data collection
Bug report selection
Field extraction and re-collapse
Data splitting: training and test set creation
Test set information filtering
Pre-processing
Text pre-processing
Feature (term) selection
Document-term matrix creation
Binary 0/1
TF
TF-IDF
Learning and severity prediction
Bug-fixing time discretization (SLOW/FAST)
Classifier training
Test set prediction (severity)
Accuracy evaluation

Fig. 1. Conceptual design of bug-fix time prediction process.

the subsequent phase described in Subsect. 3.1): `Product` (a real-world product, identified by a name and a description, having one or more bugs). `Component` (a given subsection of a Product, having one or more bugs). `Short_desc` (a one-sentence summary of the problem). `First_priority` (priority set by the user who created the report. The default values of priority are from P1, highest, to P5, lowest). `First_severity` (severity set by the user who created the report. This field indicates how severe the problem is, from blocker when the application is unusable, to trivial). `Reporter` (the account name of the user who created the report). `Assigned_to` (the account name of the developer to which the bug has been assigned to by the triager, and responsible for fixing the bug). `Days_resolution` (the calendar days needed to fix the bug). `Priority` (priority set either by the triager or a project manager). `Severity` (severity set either by the triager or a project manager). `First_comment` (the first comment posted by the user who created the report, which usually consists of a long description of the bug and its characteristics). `Comments` (subsequent comments posted by the `Reporter` and/or developers endowed with appropriate permissions, which can edit and change all bugs fields, and comment these activities accordingly).

We discarded a few fields, such as `Number_of_activities`, `CC_list`, `Status` and `Resolution`. For example, `Number_of_activities` is an integer value that would surely be removed during pre-processing steps, required to transform a raw text into a bag-of-word representation (see Sect. 3.1 and Fig. 2). Both `Status` and `Resolution` have been discarded because these fields have already been used for bug report selection, and they have always been assigned values VERIFIED and FIXED respectively. Finally, field `CC_list` contains a list of account names who get mail when the bug changes, and it is has been discarded because it has not been assigned any value in the great majority of cases. We also want to highlight that `Days_resolution` has been calculated as the number of days between the date the bug report has been assigned to a developer, and the date the `Resolution` field has been assigned to FIXED for the last time. This datum may be a very inaccurate estimate of the actual time spent on bug fixing, especially when developers of open source projects are considered (because of their discontinuous work patterns, for example during the weekend and/or their free time). Unfortunately, we cannot trace the actual time spent on a bug fixing, and provide a more accurate estimate expressed in person-hours.

When data collection and field extraction are complete, we randomly split the dataset into a training and a test dataset (given a fixed split percentage). This operation must necessarily take place after field extraction, because we have to filter out all post-submission information from test set. In fact, test instances simulate newly-opened and previously unseen bugs, and this makes compulsory to delete some of previously extracted fields that were not actually available before the bug was assigned. In particular, the deleted fields were `Priority`, `Severity` and `Comments`. On the contrary, fields `First_priority`, `First_severity` were not deleted, as they have been assigned a value by the user who created the report.

We want also to highlight that our design envisages that historical bug-fix times are discretized into two classes, conventionally labelled as SLOW and FAST. The SLOW labels indicates a discretized bug-fix time above some fixed threshold in right-tail of the empirical bug-fix time distribution. We also assume that SLOW indicates positive class, hence SLOW being the target class of our prediction exercise. In fact, we are interested in increasing the number of true positives for the positive class. In other words, over-estimation of bug-fix times can be considered as a less severe error than under-estimation.

3 Methods

3.1 Pre-processing Textual Description of Bug Reports

The forecasting models that we will use to predict bug-fix times are based on the representation of a document in terms of a bag-of-words. In this simplified representation, both the grammar and the order of occurrence of the words are not relevant. It is only relevant whether a term occur or not, as well as how many times it occurs in a textual description reproducing the bug.

A number of pre-processing steps are therefore necessary to convert the raw text into a bag-of-words representation. The steps followed in this paper, shown in Fig. 2, are quite common and well described in the literature on text categorization and Natural Language Processing (NLP; see, for example, [17]). The final goal is to precisely define a vocabulary of terms V, for example by eliminating those words that are very commonly used in a given language and focusing on the important words instead (stop word removal), or 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).

Two points appears to be particularly interesting and worthy of further elucidation. The first is that probabilistic text modeling is often done ignoring multi-word expressions that in specific contexts are given specific meanings. This issue is particularly relevant for textual descriptions of software defects, and ignoring it can lead to sub-optimal outcomes for a numbers of reason that are well described in [6]. The algorithm we have followed for detecting multi-words is a modification of a very simple heuristic introduced by [14]. The detection consists of the following steps:

1. Tokenize the text into bigrams (sequences of two adjacent words) and store candidate bigrams whose frequency of occurrence in the text is ≥ 3.
2. Pass the set of candidate bigrams through a part-of-speech (POS) filter.
3. Only let through the POS filter those part-of-speech patterns that are likely to be phrases, such as JJNN (Adjective+Noun, singular or mass, using the syntactic annotation scheme implemented by the Penn Treebank Project [18]). Ten predetermined patterns have been used to identify likely multi-words.

Surprisingly, the proposed algorithm has shown a reasonable accuracy in finding multi-words of more than two words (results will be published elsewhere).

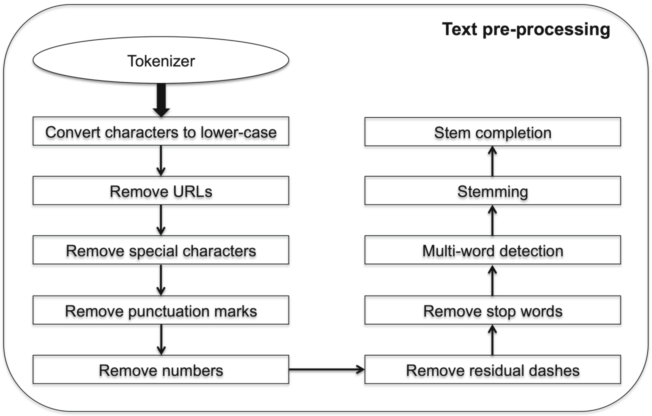

Fig. 2. The process of transforming a raw textual bug description into a bag-of-words.

Once text pre-processing has been completed, term selection is often necessary as it may even result in a moderate increase in predictive accuracy, depending on the classifier used and other factors [25]. In our approach, terms are sorted (from best to worst) according to their estimated normalized expected mutual information (NEMI, [17]) with the discretized time to resolution (SLOW/FAST). We only include terms which have a NEMI greater than the average NEMI, ensuring that terms almost approximately independent with the target class label are omitted.

The final pre-processing step consist of building, under different weighting schemes (see Fig. 1), the documents-terms arrays that will be used as inputs for the classification algorithms described in the following sections. Any global pre-processing parameters determined on the training data (such the number of documents that contain a given word) were subsequently applied to the test documents.

3.2 Prediction of Bug-Fix Times Based on Standard Text Categorization Models

As we said before, automated approaches to bug-fix time prediction, based on text-mining and machine-learning techniques, are not new [29]. One common generative model for a textual content is the Multivariate Bernoulli model (MB), which generates an indicator for each term of the vocabulary $|V|$, either 1 indicating presence of the term in the text or 0 indicating absence [17,26]. This amounts to assume that each document is represented as a binary vector $e_{1:|V|} = (e_1, \ldots, e_{|V|})$ of dimensionality $|V|$. For each bug report and independently of each other, the MB model assumes the following generative process, based on a Naïve Bayes conditional independence assumption:

1. Choose a discrete unordered label variable y (SLOW/FAST) from a discrete probability distribution.
2. For each word in V (i.e. for $t = 1, \ldots, |V|$) and independently of each other, choose a value of the indicator e_t from a Bernoulli distribution, $e_t|y \sim \mathsf{Bernouilli}(\pi_t)$, where $\pi_t = P(e_t = 1|y)$ is the probability that the word represented by e_t will occur at least once in any position, in a bug report labelled with severity y.

It is worth noting that we have suppressed writing the document index d. MB model implicitly relies on a bag-of-word assumption, and the natural input used by the model consists of a binary document-term incidence matrix (DTM). Hence, documents are modeled as a realization of a stochastic process, which is then reversed by standard machine learning techniques that return maximum-likelihood estimates of the posterior probabilities of a document (bug) d being in class y. Such estimates are in turn used for predicting discretized time to resolution y of newly-opened bugs. Despite its improved performance compared to the MB model, due to the incorporation of frequency information, we will not explicitly consider the Multinomial Naïve Bayes (MNB) model. Indeed, the supervised multi-topic model described in some detail in the next section, contains as a special case a supervised unigram model which is largely equivalent to the MNB model.

As an alternative way of considering the frequency of term occurrence in a document, we will use the Vector Space model (VS), with documents being represented as vectors in $\mathbb{R}^{|V|}$ [24]. The DTM can be weighted using the term frequency TF_{td} of word t in document d, as well as the term frequency–inverse document frequency $\mathsf{TF\text{-}IDF}_{td} = \mathsf{TF}_{td} \times \mathsf{IDF}_t$, which dampens the effects of local term-document counts. As usual, the inverse document frequency of word t is defined as $\mathsf{IDF}_t = \log(|\mathcal{D}|/\mathsf{DF}_t)$, where $|\mathcal{D}|$ is the number of documents in the training collection, and DF_t is the number of documents in $\mathcal{D}$ that contain the word t (document frequency). Using the vector-space approach, the documents of the training set correspond to a labeled set of points in an $|V|$-dimensional space. As state-of-the-art vector-based text classification algorithm [13,23], we will consider in Sect. 4 a non-linear Support Vector Machine (SVM) with soft margin classification.

3.3 Prediction Based on Supervised Latent Dirichlet Allocation

Models introduced so far limits each textual report to a single topic. This assumption may often be too limiting to model a large collection of textual bug reports, as any report typically concerns multiple topics and specific sub-issues in different proportions. As our objective is to discover this hidden thematic structure, and use it to predict the discretized time to resolution, we consider Supervised Latent Dirichlet Allocation (SLDA) as introduced in [4,28], which adds to standard Latent Dirichlet Allocation a target unordered binary variable.

We now briefly introduce the necessary notation to SLDA, as the representation of textual bug reports is different from that introduced in Sect. 3.2. As before, the target variabile is the unordered binary variable y denoting SLOW and FAST labels (also in this case, we suppress writing the document index d):

- A document d is a stream of N words, $d = (w_1, \ldots, w_N)$. Using superscripts to denote components, the νth word in $|V|$ is represented as a unit-basis vector w such that $w^\nu = 1$ and $w^u = 0$ for $u \neq \nu$.
- For each document we have K underlying semantic themes (topics), $\beta_{1:K} = (\beta_1, \ldots, \beta_K)$, where each β_k is a $|V|$-dimensional vector of probabilities over the elements of V, for $k = 1, \ldots, K$.
- $z_{1:N} = (z_1, \ldots, z_N)$ is a vector of K-dimensional vectors indicating, for $n = 1, \ldots, N$, the topic which has generated word w_n 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 $j \neq k$.

The topic indicator uniquely selects a probability distribution in $\beta_{1:K}$, as $\beta_{z_n} \equiv \beta_k$ when $z_n^k = 1$. Independently of each other, the generative process of each pre-processed bug report d is the following:

1. Draw topic proportions from a symmetric Dirichlet distribution over the K-dimensional simplex, $\theta|\alpha \sim \mathsf{Dirichlet}_K(\alpha)$.
2. For each word w_n, $n = 1, \ldots, N$, and independently of each other:
   (a) Choose a topic from a Multinomial distribution with probabilities θ, $z_n|\theta \sim \mathsf{Multinomial}_K(\theta)$.
   (b) Choose a word from a Multinomial distribution with probabilities dependent on z_n and $\beta_{1:K}$, $w_n|z_n, \beta_{1:K} \sim \mathsf{Multinomial}_{|V|}(\beta_{z_n})$.

3. Draw response variabile y (SLOW/FAST, with SLOW $\equiv 1$) from a logistic Generalized Linear Model (GLM) of the form:

$$y|z_{1:N}, \eta \sim \text{Bernouilli}\left(\frac{\exp(\eta^\top \overline{z})}{1+\exp(\eta^\top \overline{z})}\right), \tag{1}$$

where $\overline{z} = (1/N)\sum_{n=1}^{N} z_n$ is the vector of empirical topic frequencies.

In this hierarchical specification, we model discretized bug-fix time as a non-linear function of the empirical topic frequencies via the linear predictor $\eta^\top \overline{z}$ (readers unfamiliar with GLMs may profitably consult [9]). A graphical model representation of SLDA is depicted in Fig. 3, where observable stochastic nodes are represented as gray circles and latent stochastic variables are white circles. Parameters $\alpha, \beta_{1:K}$ and η (white boxes) are treated as unknown fixed hyper-parameters rather than latent stochastic nodes.

It is apparent from Fig. 3 that the response y and the words share a common ancestor (the latent topic variables), and hence they are not conditionally independent. Documents are generated as a bag-of-words under full word exchangeability, and then topics are used to explain the response. Other specifications are indeed possibile, for example y can be regressed as a nonlinear function of topic proportions θ, but [4] claim that the predictive performance degrades as a consequence of the fact the topic probabilistic mass does not directly predicts discretized bug-fix times.

Posterior inference of latent model variables is not feasible, as the conditional posterior distribution $p(\theta, z_{1:N}|w_{1:N}, y, \alpha, \beta_{1:K}, \eta)$ has not a closed form. Consequently, we use a standard variational Bayes (VB) parameter estimation approach under a mean-field approximation, say $q(\theta, z_{1:N}|\phi)$, of the conditional posterior distribution [5,16]. In the variational E-step of the algorithm, the variational parameters ϕ are optimized to minimize the Kullback-Leibler divergence

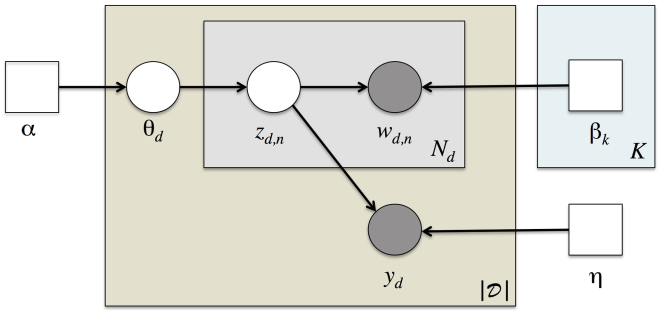

Fig. 3. A graphical model representation of Supervised Latent Dirichlet Allocation (SLDA). Parameters $\alpha, \beta_{1:K}$ and η are treated as unknown hyper-parameters to be estimated, rather than random variables.

between $q(\theta, z_{1:N}|\phi)$ and the conditional posterior distribution, given the hyper-parameters $\alpha, \beta_{1:K}$ and η. In the M-step, the hyper-parameters are optimized to maximize the evidence lower bound (ELBO) given the variational parameters.

Let $\{\tilde{\alpha}, \tilde{\beta}_{1:K}, \tilde{\eta}\}$ denote the hyper-parameters of a fitted model. As our main objective is to apply logistic SLDA to predict discretized bug-fix time of a newly-opened bug, whose words are $w^{\text{new}}_{1:N}$, this amounts to approximating the marginal posterior expectation of the response variable:

$$E(y^{\text{new}}|w^{\text{new}}_{1:N}, \tilde{\alpha}, \tilde{\beta}_{1:K}, \tilde{\eta}) = E(\mu(\tilde{\eta}^{\top}\bar{z}^{\text{new}})|w^{\text{new}}_{1:N}, \tilde{\alpha}, \tilde{\beta}_{1:K}), \qquad (2)$$

with $\mu(\tilde{\eta}^{\top}\bar{z}^{\text{new}}) = \exp(\tilde{\eta}^{\top}\bar{z}^{\text{new}})/(1+\exp(\tilde{\eta}^{\top}\bar{z}^{\text{new}}))$. The identity (2) easily follows from the law of iterated expectations. Using the multivariate Delta method and a suitable variational procedure, the RHS of (2) can be easily approximated (we defer specifics to [4,8]). Of course, if $E(y^{\text{new}}|w^{\text{new}}_{1:N}, \tilde{\alpha}, \tilde{\beta}_{1:K}, \tilde{\eta}) > 0.5$ then $y^{\text{new}} \equiv 1(\equiv \text{SLOW})$.

4 Results

We have obtained bug report information from Bugzilla repositories of four large open source software projects: Eclipse, Gentoo, KDE and OpenOffice. Data were automatically extracted from Bugzilla data sources, using a suitable scraping routine written in PHP/JavaScript/Ajax. Raw textual reports were pre-processed and analyzed using the R software system [22]. The table below shows the total number of textual bug reports extracted for each project (n_1), having both `Status` field assigned to VERIFIED and `Resolution` field assigned to FIXED. Next, we deleted some textual reports because of corrupted and unrecoverable records, or missing XML report, or dimension being too large (the resulting sample sizes are in column n_2). If $n_2 > 1500$ (resp.: $n_2 \leq 1500$) we randomly selected $n_3 = 1500$ bug reports (resp.: we retained the whole set of $n_3 = n_2$ bug reports), in order to train the classifiers and test the proposed models.

	n_1	n_2	n_3	n_4	n_5
Eclipse	44435	44347	1500	1200	300
Gentoo	3704	2466	1500	1200	300
KDE	1275	1270	1270	1016	254
Open Office	3057	3057	1500	1200	300

These subsets were randomly divided into a training and a test part, using an 80:20 split ratio. The resulting sample sizes are indicated by n_4 (training set) and n_5 (test set). Before splitting, for each experiment we binned bug reports into FAST and SLOW using the third quartile $q_{0.75}$ of the empirical distribution of bug resolution times. The table below shows the resulting binned distribution.

We can see that the time needed to fix the bugs exhibits large variations, and that the underlying continuous-time effort distribution is long-tailed, with a large percentage of bugs being fixed within a relatively short time.

	Eclipse	Gentoo	KDE	Open Office
FAST	0–52	0–42	0–248	0–107
SLOW	53–3696	43–1854	249–2188	108–2147

As shown below, the SLOW/FAST ratio is approximately preserved after each dataset is split into a training and a test part. Hence, the two class are moderately imbalanced in both the training and the test set, with a low ratio of positive to negative class (as we said before, SLOW identifies the positive class).

	n_3	Training set			Test set		
		SLOW	FAST	ratio	SLOW	FAST	ratio
Eclipse	1500	299	901	25.00 %	82	218	27.00 %
Gentoo	1500	298	902	25.00 %	79	221	26.00 %
KDE	1270	254	762	25.00 %	67	187	26.00 %
Open Office	1500	300	900	25.00 %	65	235	22.00 %

As we highlighted in Sect. 2, we are interested in increasing the number of true positives for the positive class, considering over-estimation of bug-fix times as a less severe error than under-estimation. We used accuracy, precision, recall and false positive rate (FPR) for measuring the performance of prediction models. Accuracy denotes the proportion of correctly predicted bugs: Accuracy = (TP+TN)/(TP+FP+TN+FN). Precision denotes the proportion of correctly predicted SLOW bugs: Precision = TP/(TP+FP). Recall denotes the proportion of true positives of all SLOW bugs: Recall = TP/(TP+FN). Finally, false positive rate measure the proportion of false positive of all FAST bugs: FPR = FP/(FP+TN). The following classification models were trained over the training set, and evaluated over the test set:

– Multivariate Bernoulli (MB), with either no posterior class probabilities smoothing, or Laplace λ smoothing parameter respectively set to $1, 2, 3$ [17].
– Support Vector Machines (SVM), with sigmoid kernel and soft-margin classification, cost parameter C respectively set to $0.1, 1, 10, 100$ and precision parameter γ respectively set to $0.001, 0.01, 0.1, 1$ [15].
– Supervised Latent Dirichlet Allocation (SLDA), with number of topics K being respectively set to 1, 2, 5, 10, 15, 20, 25, 30, 35, 40, 50, 100. When $K = 1$, the SLDA model becomes essentially equivalent to the Multinomial Naïve Bayes (MNB) model.

In what follows, we optimize over the test set and show only the best models (along with the corresponding parameter settings), i.e. the model with the highest predictive accuracy among the models of the same class. The full result set is shown below. We look at the first table (Eclipse), as it exhibits a typical pattern. The highest achievable accuracy with SLDA ($K = 25$ topics) is lower than the best achievable accuracies with both MB and SVM. However, accuracy provides reliable comparisons only when the two target classes have equal importance. In our setting, we assume that the minority class (SLOW) is more important, because of its larger impact in terms of cost/effectiveness. If, consequently, we assume that the main goal is increasing the recall, the logistic SLDA model has the best performance. FPR increases too, as expected, because of the larger number of FAST bugs that are classified SLOW under the SLDA model. However, costs incurred in false positive are generally very low. On the contrary, both MB and SVM classifiers have high accuracy, but they failed to correctly predict most of SLOW bugs. On the whole, these result clearly show that the use of a supervised topic model greatly improves the recall of bug-fix time prediction.

Eclipse.

	Parameters	Accuracy	Precision	Recall	FPR
MB	$\lambda = 2$	0.73	0.60	0.04	0.01
SVM	$\gamma = 0.001$, $C = 10$	0.67	0.23	0.09	0.11
SLDA	$K = 25$	0.57	0.32	0.48	0.40

Gentoo.

	Parameters	Accuracy	Precision	Recall	FPR
MB	$\lambda = 2$	0.74	0.50	0.13	0.05
SVM	$\gamma = 0.001$, $C = 100$	0.74	0.67	0.03	0.00
SLDA	$K = 30$	0.43	0.27	0.70	0.67

KDE.

	Parameters	Accuracy	Precision	Recall	FPR
MB	$\lambda = 2$	0.83	0.64	0.79	0.02
SVM	$\gamma = 0.001$, $C = 100$	0.60	0.03	0.01	0.19
SLDA	$K = 40$	0.41	0.29	0.84	0.74

Open Office.

	Parameters	Accuracy	Precision	Recall	FPR
MB	$\lambda = 2$	0.78	0.00	0.00	0.00
SVM	$\gamma = 0.001$, $C = 100$	0.58	0.22	0.38	0.38
SLDA	$K = 10$	0.51	0.23	0.55	0.55

5 Discussion and Conclusion

Manual bug triage is expensive both in time and cost. But even more importantly, manual triage is error-prone due to the large number of daily newly-opened bugs and the lack of knowledge about all bugs by the developers. We have, therefore, proposed a novel model for automatic prediction of bug-fix time of newly opened bugs, in order to support the whole triage process and, in particular, the assignment of any new bug to a developer who will try to fix it. Our prediction models use text categorization techniques, mapping each textual bug description into a bag-of-words after a suitable pre-processing stage. Each bug in the training data set is classified as SLOW or FAST (discretized time to resolution). The trained prediction model is then used to predict the discretized time to resolution of each bug in the test set. Any post-submission information, which was not actually available before the bug was assigned, has been removed from the test set. We compared two single-topic supervised learning algorithms, multivariate Bernoulli Model (MB) and Support Vector Machines (SVM), with a multi-topic model known as Supervised Latent Dirichlet Allocation (SLDA), recently introduced in [4].

To evaluate the forecasting accuracy of the proposed predictive model against that of MB and SVM algorithms, we used bug datasets on bug repositories of four large open source projects: Eclipse, Gentoo, KDE and OpenOffice. Results show that the proposed model greatly improves recall, when compared to single topic algorithms. On the other hand, the loss of accuracy of our method is quite significant. However, predictive accuracy provides meaningful and reliable comparisons only when the two target classes have equal importance. In our experimental setting the negative class (FAST) plays a minor role. Therefore, we may assume that the main goal is increasing the recall, that is the true positives for the positive class (SLOW). In this case, the number of false positives can also be increased, even though costs incurred in false positives are generally very low. Finally, a comparison with previously reported literature values shows a marked improvement of the predictive accuracy of the two single topic algorithms. However, a direct comparison is not possible and further validation will be needed, as different software projects were examined, and different attributes and/or parts of textual bug reports were selected, as well as different pre-processing methods were involved in extracting the bag-of-words representations.

In conclusion, the proposed method seems promising for implementing a large-scale bug-fix time prediction system. In the future, we plan to investigate the following threatens to internal validity:

- Using the quantile q_a with $a = 0.75$ to separate positive and negative instances is arbitrary. Letting a vary has an impact on both the imbalance ratio and the predictive accuracy. A sensitivity analysis is therefore needed.
- We need to assess the performance on a larger and independent validation set. Each presented method is indeed trained for a number of parameter settings and tested on the test set, but only the best results are presented. In the future, the methods will be optimized on a separate validation set, instead of the test set.
- Another issue consists of identifying the potential outliers of the distribution of bug-fix times, and using them to filter out the data sets. Many authors have demonstrated that filtering these outliers can improve the accuracy of the prediction models (see, for example, [1]).
- Most of defect tracking systems are just ticketing systems, that cannot keep track of actual person-hours spent to resolve a bug. This threat to internal validity of bug-fix time prediction models has not been investigated yet. In the same way, similarities and differences between open and non-open source software projects need to be investigated further.

Finally, some recent techniques tackle the bug-fix time prediction problem using a radically different (yet very promising) approach, based on predictive Process Mining, in which each ticket gives rise to a series of activities viewed as an instance of some ticket handling process, and which can be handled along with textual informations and descriptive fields, in order to assign a performance value to any partial process instance, and monitoring the resolution status during its enactment as well (see [10]). It is therefore highly desirable to compare (at least experimentally) our solution with this kind of approach. The future work will explore this interesting task.

References

1. AbdelMoez, W., Kholief, M., Elsalmy, F.M.: Improving bug fix-time prediction model by filtering out outliers. In: 2013 The International Conference on Technological Advances in Electrical, Electronics and Computer Engineering (TAEECE), pp. 359–364 (2013)
2. Bhattacharya, P., Neamtiu, I.: Bug-fix time prediction models: can we do better? In: Proceeding of the 8th Working Conference on Mining Software Repositories, MSR 2011, pp. 207–210. ACM Press, New York (2011)
3. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent Dirichlet allocation. J. Mach. Learn. Res. **3**, 993–1022 (2003)
4. Blei, D.M., McAuliffe, J.D.: Supervised topic models. In: NIPS (2007)
5. Blei, D.M., Kucukelbir, A., McAuliffe, J.D.: Variational inference: a review for statisticians, pp. 1–33 (2016). http://arxiv.org/abs/1601.00670

6. Boyd-Graber, J., Mimno, D., Newman, D.: Care and feeding of topic models: problems, diagnostics, and improvements. In: Airoldi, E.M., Blei, D., Erosheva, E.A., Fienberg, S.E. (eds.) Handbook of Mixed Membership Models and Their Applications. CRC Press, Boca Raton (2014)
7. The Bugzilla Team: Bugzilla Documentation 5.0.3+ (2016). https://www.bugzilla.org/docs/
8. Chang, J., Blei, D.M.: Hierarchical relational models for document networks. Ann. Appl. Stat. **4**(1), 124–150 (2010)
9. Dobson, A.J., Barnett, A.: An Introduction to Generalized Linear Models: Chapman & Hall/CRC Texts in Statistical Science, 3rd edn. Taylor & Francis (2008)
10. Folino, F., Guarascio, M., Pontieri, L.: An approach to the discovery of accurate and expressive fix-time prediction models. In: Hammoudi, S., Maciaszek, L., Teniente, E., Camp, O., Cordeiro, J. (eds.) ICEIS 2015. LNBIP, vol. 241, pp. 108–128. Springer, Heidelberg (2015). doi:10.1007/978-3-319-22348-3_7
11. Giger, E., Pinzger, M., Gall, H.: Predicting the fix time of bugs. In: Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering, RSSE 2010, pp. 52–56. ACM Press, New York (2010)
12. Hu, H., Zhang, H., Xuan, J., Sun, W.: Effective bug triage based on historical bug-fix information. In: 2014 IEEE 25th International Symposium on Software Reliability Engineering, pp. 122–132. IEEE (2014)
13. Joachims, T.: Text categorization with support vector machines: learning with many relevant features. In: Nédellec, C., Rouveirol, C. (eds.) ECML 1998. LNCS, vol. 1398, pp. 137–142. Springer, Heidelberg (1998). doi:10.1007/BFb0026683
14. Justeson, J.S., Katz, S.M.: Technical terminology: some linguistic properties and an algorithm for identification in text. Nat. Lang. Eng. **1**(01), 9–27 (1995)
15. Karatzoglou, A., Meyer, D., Hornik, K.: Support vector machines in R. J. Stat. Softw. **15**(1), 1–28 (2006)
16. Lakshminarayanan, B., Raich, R.: Inference in supervised latent Dirichlet allocation. In: 2011 IEEE International Workshop on Machine Learning for Signal Processing, pp. 1–6 (2011)
17. Manning, C.D., Raghavan, P., Schütze, H.: Introduction to Information Retrieval. Cambridge University Press, New York (2008)
18. Marcus, M., Kim, G., Marcinkiewicz, M.A., MacIntyre, R., Bies, A., Ferguson, M., Katz, K., Schasberger, B.: The Penn Treebank: annotating predicate argument structure. In: Proceedings of the Workshop on Human Language Technology, pp. 114–119. Association for Computational Linguistics, Stroudsburg (1995)
19. Marks, L., Zou, Y., Hassan, A.E.: Studying the fix-time for bugs in large open source projects. In: Proceedings of the 7th International Conference on Predictive Models in Software Engineering, Promise 2011, pp. 1–8. ACM Press, New York (2011)
20. Panjer, L.D.: Predicting eclipse bug lifetimes. In: Fourth International Workshop on Mining Software Repositories, MSR 2007: ICSE Workshops 2007, pp. 29–32. IEEE, Washington, DC (2007). doi:10.1109/MSR.2007.25
21. Pressman, R.S., Maxim, B.R.: Software Engineering: A Practitioner's Approach, 8th edn. McGraw-Hill Higher Education (2014)
22. Core Team, R.: R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria (2016). https://www.R-project.org/
23. Rennie, J.D.M., Shih, L., Teevan, J., Karger, D.R.: Tackling the poor assumptions of Naïve Bayes text classifiers. In: Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003), Washington DC, pp. 616–662 (2003)

24. Salton, G., Wong, A., Yang, C.S.: A vector space model for automatic indexing. Commun. ACM **18**(11), 613–620 (1975)
25. Sebastiani, F.: Machine learning in automated text categorization. ACM Comput. Surv. **34**(1), 1–47 (2002)
26. Wilbur, W.J., Kim, W.: The ineffectiveness of within-document term frequency in text classification. Inf. Retr. **12**(5), 509–525 (2009)
27. Xuan, J., Jiang, H., Hu, Y., Ren, Z., Zou, W., Luo, Z., Wu, X.: Towards effective bug triage with software data reduction techniques. IEEE Trans. Knowl. Data Eng. **27**(1), 264–280 (2015)
28. Zhang, C., Kjellström, H.: How to supervise topic models. In: Agapito, L., Bronstein, M.M., Rother, C. (eds.) ECCV 2014, Part II. LNCS, vol. 8926, pp. 500–515. Springer, Heidelberg (2015). doi:10.1007/978-3-319-16181-5_39
29. Zhang, J., Wang, X., Hao, D., Xie, B., Zhang, L., Mei, H.: A survey on bug-report analysis. Sci. China Inf. Sci. **58**(2), 1–24 (2015)