A Text-Based Regression Approach to Predict Bug-Fix Time

Pasquale Ardimento, Nicola Boffoli and Costantino Mele

Abstract Predicting bug-fixing time can help project managers to select the adequate resources in bug assignment activity. In this work, we tackle the problem of predicting the bug-fixing time by a multiple regression analysis using as predictor variables the textual information extracted from the bug reports. Our model selects all and only the features useful for prediction, also using statistical procedures, such as the Principal Component Analysis (PCA). To validate our model, we performed an empirical investigation using the bug reports of four well-known open source projects whose bugs are stored in Bugzilla installations, where Bugzilla is an online open-source Bug Tracking System (BTS). For each project, we built a regression model using the M5P model tree, Support Vector Machine (SVM) and Random Forests algorithms. Experimental results show the model is effective, in fact, they are slightly better than all the ones known in the literature. In the future, we will use and compare other different regression approaches to select the best one for a specific data set.

1 Introduction

In software maintenance, "a critical activity, which consumes the majority of the effort spent within the lifetime of a software system" [1], a significant amount of time is spent to investigate software bugs [2]. Generally, large-scale software projects use a Bug Tracking System (BTS) to report and manage a software bug. BTS management is relied on by team members, which can be developers and test engineers, and which have to fix bugs in the source code files. Each bug report must be triaged.

P. Ardimento (B) ·N. Boffoli ·C. Mele
Department of Informatics, University of Bari Aldo Moro, Via Orabona 4, Bari, Italy
e-mail: pasquale.ardimento@uniba.it

N. Boffoli
e-mail: nicola.boffoli@uniba.it

C. Mele
e-mail: c.mele22@studenti.uniba.it

The triager, who usually is a senior developer, selects the appropriate developer to fix the newly submitted bug. However, due to the large number of bug reports submitted daily for large-scale software projects, accurate bug triage is normally done manually. Furthermore, several studies demonstrate that bug-assignment is error-prone, expensive and that many times it is necessary to reassign a bug to another one ("bug tossing"). In recent years, several researchers analyzed bug-fixing time and its prediction. For example, Panjer [3] proposed to use classification techniques such as 0-R, 1-R, Decision Tree, Naive Bayes and Logistic Regression to predict the time to fix a bug for Eclipse project obtaining an accuracy of 34.9%. In [4] Kim et al. studied the life span of bugs in ArgoUML and PostgreSQL projects, and found that bug-fixing time had a median of about 200 days. Giger et al. [5] used Decision Tree to classify fast and slowly fixed bugs studying Eclipse, Mozilla, and Gnome projects.

The above-mentioned works, focused on bug-fixing time for open source projects, show a real need to improve the prediction accuracy results. The contribution of this paper is building a regression model, modifying the model already proposed in [6, 7], useful to predict the bug-fixing time, in order to solve this issue as a numerical regression problem. For this purpose, we extracted information contained in the Bugzilla bug reports relating to the Mozilla [8], FreeDesktop [9], NetBeans [10] and Eclipse [11] projects, to create a database on which the machine learning (ML) algorithms trains. The database chosen to host extrapolated data is MongoDB [12], a non-relational database that can easily handle collections of JSON documents. The environment used to create the data set and to perform the regression analysis is R [13], an open source software for statistical analysis and ML. We evaluated our model using M5P model tree, Random Forests and SVM algorithms comparing obtained results with that one's known in the literature.

Here below, Sect. 2 shows the background whereas Sect. 3 gives an overview of the literature found on the subject. Section 4 describes the proposed model and the results of the empirical investigation are presented in Sect. 5. Finally, Sect. 6 discusses results and provides conclusions.

2 Background

Each bug reported in a BTS follows a life cycle: it starts when the bug is discovered and ends when the bug is closed, after ensuring it has been fixed. Bug life cycle can be slightly different depending on the BTS used. To select bugs useful for prediction and, at the same time, to build a model independently from the BTS chosen, we studied both general bug life cycle and Bugzilla bug life cycle.

We selected Bugzilla as BTS basically for two reasons: first, it has a wide public installation base; on Bugzilla official page there is a list, whose last update is on May 3rd, 2017, of 137 companies, organizations, and projects that run "public" Bugzilla installations. Second, since version 5.0, Bugzilla installations offer a native well documented REST API [14] as a preferred way to interface with Bugzilla from external apps. Figure 1 shows life cycle of a bug in Bugzilla, as represented in the

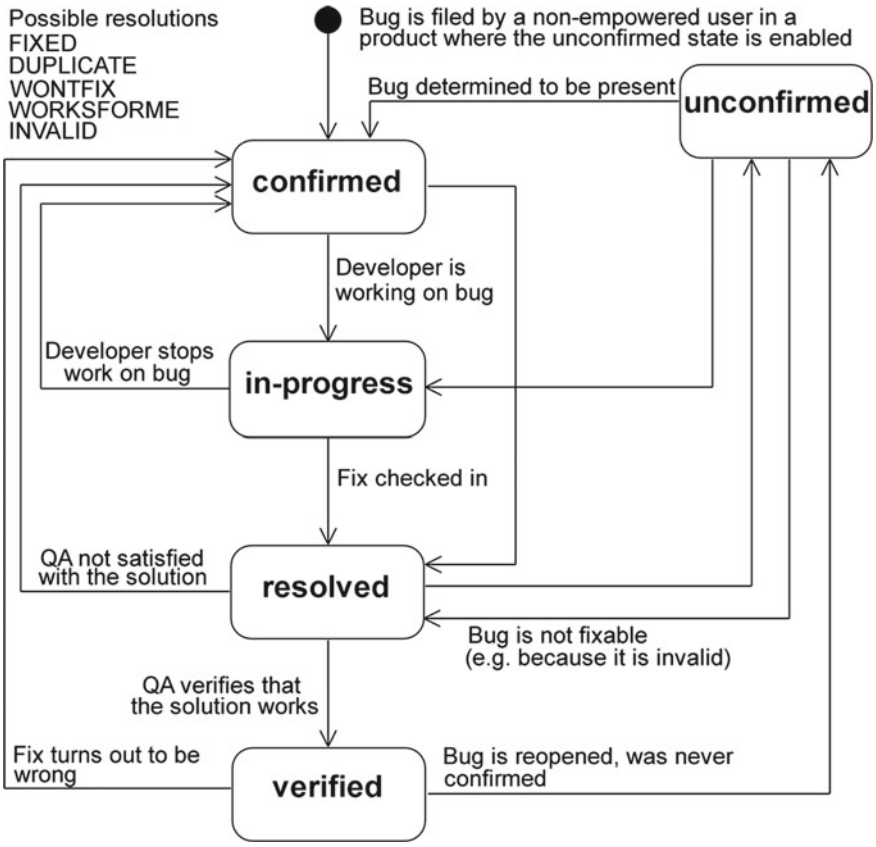

Fig. 1 Life cycle of a bug in Bugzilla

Bugzilla official documentation release 5.0.4 at 2.4.4 section [15], while Fig. 2 shows general bug life cycle.

General BTS as well as Bugzilla BTS allow users to report, track, describe, comment on and classify bug reports. A bug report is characterized by several predefined fields, such as the relevant product, version, operating system and self-reported incident severity, as well as free-form important text fields such as bug title, called summary in Bugzilla, and description. Moreover, users and developers add comments and submit attachments, which often take the form of patches, screenshots, test cases or anything else binary or too large to fit into a comment. When initially declared, a bug starts out in the unconfirmed pending state until a triager makes a first evaluation to see if the bug report corresponds to a valid bug, and that the bug is not already known, i.e., the submitted bug report is a duplicate of another bug report already stored in the defect reporting system. Bug reports can pass through several different stages before finally being resolved. Bug reports that are closed receive one of the following status: duplicate, invalid, fixed, wontfix, or worksforme.

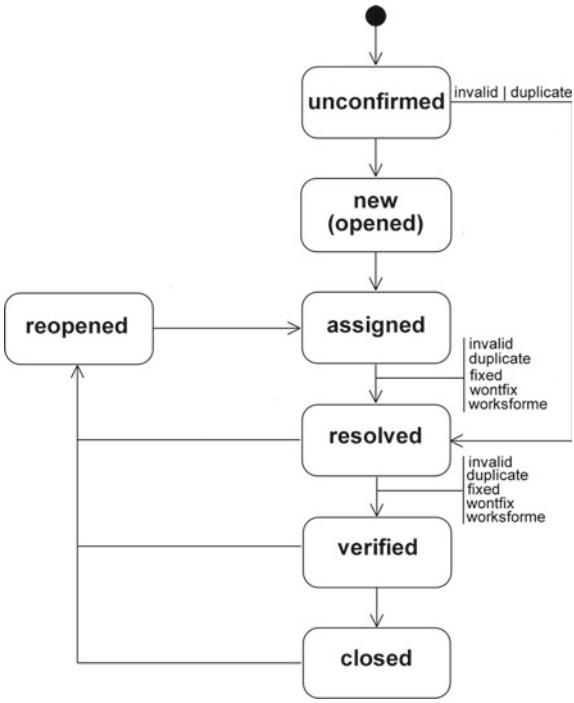

Fig. 2 General life cycle of a bug

These indicate why the report was closed; for example, worksforme and invalid both indicate that quality assurance was unable to reproduce the issue described in the report. Sometimes a bug report needs to be reopened and when it happens the normal defect lifecycle starts with status reopened.

Reopened status represents the most important difference between the two life-cycles because it is absent in Bugzilla. Anyway, differently from what shown in Fig. 1, reproducing trusty the image shown in Bugzilla documentation, it is also possible to add a reopened status in Bugzilla. This operation can be done simply adding a new status option, technically selecting "add option for Adding a new status", for the field value of status. As consequence, we decided to select only Bugzilla installations on where reopened status was added.

3 Related Work

According to our research, we focus on studies that propose models for predicting the overall time required for fixing bugs via classification and regression techniques.

In 2007, Lucas D. Panjer [3] focused his research on the bug reports of Eclipse project. He used machine learning algorithms as 0-R, 1-R, decision trees, Naive Bayes and logistic regression and he reported that his model is able to correctly

predict 34.9% of the bugs. Despite the results obtained by the logistic regression, the experimentation shows a lack in the classification phase, however the results obtained are in line with those obtained from other experiments in the literature. In the same year, Hooimeijer et al. [16] applied linear regression on 27.000 bug reports from the Firefox project in an attempt to identify an optimal threshold value by which the bug report may be classified as either "convenient" or "expensive". Experiments have shown that if there are many comments or if there are many attachments, it is very likely that the bug is classified as "expensive". The model was constructed using a statistical approach, as the text categorization is computationally more burdensome than a linear model, but using techniques based on text categorization could result in a significant increase in performance compared to the model presented.

In 2009, Anbalagan et al. [17] performed their study on 72.482 bug reports from Ubuntu. The experimentation showed that there is a strong linear relationship between the time to fix a bug and the number of developers involved in the correction, linear regression was used to estimate the coefficients of the predictive model. The results of this study are not satisfactory, since it has emerged that the predictive model achieved is able to predict the time to correct a bug, about with the same precision of the models already existing in the literature and at the same cost.

In 2011, Bhattacharya et al. [18] have trained a multiple regression model considering the severity of the bug, the number of attachments, the dependencies between the various bugs and the number of developers involved in the resolution process as independent variables. The results denote a low predictive power of the model. The results shown by these experiments should not surprise us, as previously the low predictive power of the models existing in the literature has been highlighted.

In 2016, Puranik et at. [19] have developed a predictive model by selecting the minimal set of best performing metrics used in the literature related to the bug prediction problem. To carry out the experiments, a data set already proposed in [20] was used. The model realized is based on multiple linear regression, considering as variables the optimal metrics selected by the authors, such as the number of bugs found up to that moment, the version number adopted at that time, the number of lines of code and the entropy. The results of this experimentation were not provided; however, the authors confirm that the proposed model behaves much better than the other two models considered, especially when the metrics used in the evaluation are calculated on the test set.

Finally, some researchers have applied Markov-based models. In 2018, Habayeb et al. [2] employed a hidden Markov model for predicting bug fixing time based on the temporal sequence of developer activities. This approach considers the temporal sequences of developer activities rather than frequency of developer activities used by previous approaches in [3, 5, 16]. They performed an experiment on Firefox projects and compared her model with popular classification algorithms to show that the model outperformed existing ones. In 2013 Zhang et al. [21] work on predicting bug fixing time. They used open source data from three commercial software projects from CA technologies and they apply a Markov-based model to predict the number of bugs that can be fixed monthly. In 2018, Akbarinasaji et al. [22] replicated Zhang

et al. [21] using open source data from Bugzilla Firefox. The results of this replication study are similar to the original experiment and confirm the original proposed model.

Starting from the results obtained from the various studies it is possible to state that the models based on the information retrieval, if used in a classification activity, are more predictive than the statistical models, the same cannot be said for regression analysis, because in the literature there is not a numerical regression model that exploits the text information contained in the bug reports. The experiments highlight that the selection of attributes contributes significantly to increase the predictive power of the model, especially when used to define attributes characterized by a stronger correlation with respect to the time of resolution of a bug. In some cases, the information on sampling is omitted, the chosen sampling could therefore largely influence the results obtained and there is no way to compare them appropriately. In the context of classification, we can affirm that at present the logistic regression, when compared with other algorithms, seems to obtain the best performances, very often due to the simplicity of the training phase compared to other models.

This work, to the best of our knowledge, is the first one to tackle the problem of predicting the bug-fixing time by a multiple regression analysis using as predictor variables the textual information extracted from the bug reports. To this regard, we used SVM, M5P model tree and Random Forests algorithms, all configured for regression analysis. Moreover, this work is also the first one to use a dimensionality-reduction method, a process until now never used even if, as stated by many authors, necessary in accordance to the intrinsic nature of the aforementioned problem. In our work, we used PCA as a dimensionality-reduction method.

4 Proposed Model

Our idea is to transform the prediction problem into a numerical regression problem, in which we extract significant textual information from bug reports in order to predict bug-fixing time.

The prediction model proposed is shown schematically in Fig. 3. It mainly consists of three phases, already proposed in [6, 7], that are Data Collection, Pre-processing, and Learning and severity prediction. The main differences of the model proposed in this work are the use of a dimensionality-reduction method and having dealt the problem as a numerical regression problem not more as a classification problem.

4.1 Data Collection

Data Collection phase involves data gathering and data analysis for the bug-fix time prediction from one or more Bug Tracking Systems. The model of this first phase is shown in the left side of Fig. 3. Our design is largely application independent but, anyway, for this work we decided to use the open source BTS Bugzilla.

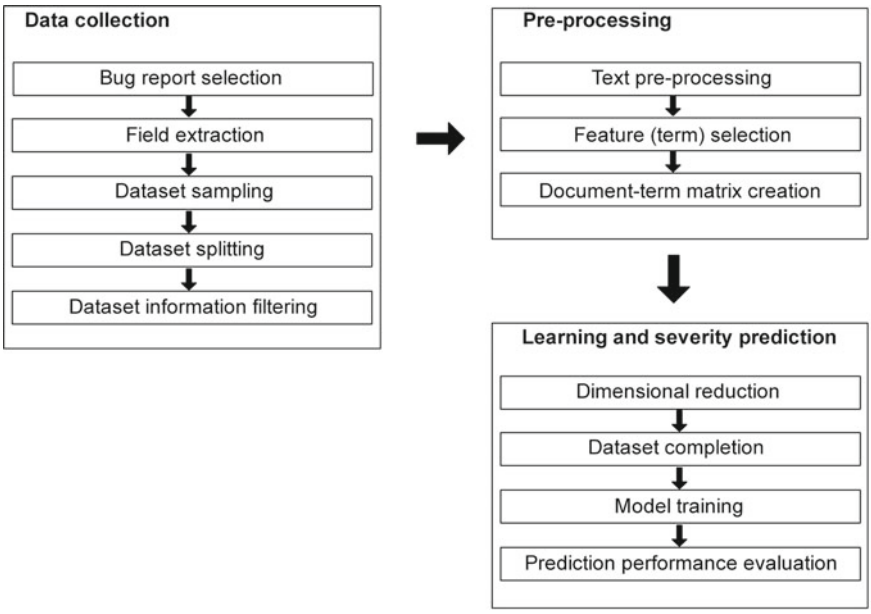

Fig. 3 Conceptual design of bug-fix time prediction process

Bug report selection consists of data gathering and data selection of only those historical bug reports from the BTS datastore whose Status field has been assigned to VERIFIED and Resolution field has been assigned to FIXED. These ones are the only useful for our regression analysis. For this purpose, we have used a web application able to carry out a web scraping process of bug reports from the Bugzilla platform. This process was made possible by exploiting some APIs made available by Bugzilla, collecting bug reports of each project adopted in separate JSON file. Our approach involves the use of the textual content of the bug reports extracted as independent variable, hence we selected those fields deemed significant for the prediction. Our choice includes the selection of the following fields:

- 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. Default values of priority are from P_1, highest, to P_5, 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/s of the developer/s to which the bug has been assigned to by the triager, and responsible for fixing the bug).

- Priority (priority set 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).
- Fixing-time was not available, so we introduce an additional field called Days_resolution, calculated as the time distance between the final time where bug field Status was set to RESOLVED and the date where the bug report was assigned for the first time. It is important to note that Days_resolution field is calculated in calendar days and not in working days, where usually a working day correspond to 8 h, because there is no accurate information about the actual time spent by developers responsible for fixing bugs. For this reason, Days_resolution field may be not very accurate and potentially affect the results.

We decided to discard some fields, because insignificant or unusable. The "Number of activities" field, for example, has been discarded because it is a numeric field, so, for this reason, it would have been any way removed in the pre-processing phase. Another field, the "CC list" field, containing the list of users interested in receiving an email notification each time the report update, was discarded because often not filled; fields "Status" and "Resolution" were not considered because already used for the selection of bug reports, hence not statistically valid for the prediction. After selecting the bug reports and extracting from them the relevant fields, we stored them in a non-relational database, our choice was the MongoDB database. We have chosen a non-relational database for the greater flexibility they offer for storing textual documents. Then we used a R script to access to the MongoDB database to import the bug reports as JSON objects in R environment. Due to hardware and software limitations it was not possible to use the entire set of bug reports stored in the Mongo DB database for the purpose of prediction. For this reason, we performed a random sampling for each data set, considering a sample composed of at most 2000 instances.

We split the resultant data sets into training, test and validation set, given a fixed split percentage. Data Collection involves also information filtering of those fields that are not generally present at the time of the insertion of a new report. In this activity, moreover, the Days_resolution field belonging to the bug reports is temporarily eliminated and kept for the purpose of prediction, given that this field does not require a pre-processing phase, being a numeric field. Initially we thought to use information filtering, denoted as IF_1 (Information Filtering n. 1), on the test set and validation set, as already performed in [22], because them instances simulate newly-opened and previously unseen bug reports, and this makes compulsory to delete some of the previously extracted fields that were not actually available before the bug was assigned. The deleted fields are: First_comment for instances belonging to the training set; Priority, Severity and Comments for instances belonging to the test and validation set.

We have also presented a further methodology of information filtering, denoted as IF_2 (Information Filtering n. 2), which provides for the uniform filtering of the information present in the instances belonging to the training, test and validation set, as we believe that a prediction based on textual content should be done using the same information for model training and for predicting information related to the bug-fixing time. In this case, the deleted fields are: Priority, Severity and Comments.

4.2 Pre-processing

The pre-processing phase, shown in Fig. 3, converts the original textual bug reports data in a data-mining-ready structure, where the most significant text-features that serve to build the regression model, are identified.

The model used to predict bug resolution time is based on the bug report representation in terms of bag-of-words. In this representation, the order of occurrence and the grammatical form of the words are not relevant while the presence or not of a term and its occurrence are discriminant. To represent the bug reports in terms of bag-of-words, it becomes necessary to do a text pre-processing: such activity is common to many works of text categorization and natural language processing and is well documented in the literature [23]. The goal is to define a vocabulary of terms representative of the context to classify, eliminating information that brings no benefit.

Text pre-processing tasks we used are the well-known ones such as: converting all words to lowercase; removing punctuation; removing URLs; removing stop words; text stemming, using Porter stemming algorithm, that is reducing each word to its stem.

The following code shows some of the principal activities performed during text pre-processing of the data corpus, using the R package "SnowballC" [26].

Text pre-processing

```
# remove extra white-spaces
corpus <- tm_map(corpus, stripWhitespace)
# convert to lower-case
corpus <- tm_map(corpus, content_transformer(tolower))
# remove numbers
corpus <- tm_map(corpus, removeNumbers)
# remove isolated dashes '-'
corpus <- tm_map(corpus, removePunctuation,
preserve_intra_word_dashes = TRUE)
# remove stopword
my.stopwords <- c(stopwords("english"),
break","else", "function", "next", "repeat")
```

```
corpus <- tm_map(corpus, my.removeWords, my.stopwords)
# stemming
corpus <- tm_map(corpus, stemDocument, language = "english")
```

Then we converted the content into a bag of words and in the feature selection activity we eliminated information that brings no benefit, such as words whose length is lower than 3 and higher than 20 characters. Finally, last activity aims to build the document-term matrices, weighed through term-frequency (TF) and term frequency—inverse document frequency (TF-IDF), whose terms will constitute the feature space to perform the regression analysis. These matrices, commonly used in natural language processing, contain the frequency of terms in documents. Rows correspond to documents (e.g. bug reports) and columns correspond to terms and each entry contains the frequency of the corresponding term in the respective document.

The following code shows how we performed feature selection and document-term matrix creation using the R package "tm" [27].

Feature term selection and document-term matrix creation

```
# build the training document-term matrix of the training set,
# setting the following global bounds:
# - term length: between 3 and 25
# - document frequency: >=5
dtmTraining <- DocumentTermMatrix(corpusTraining,
control = list(wordLengths = c(3, 25),
bounds = list(global = c(5, Inf))))
# tf document-term matrix of the training set
tfTrainingMatrix <- t(as.matrix(dtmTraining))
```

This procedure is also valid to build the document-term matrices for test and validation set.

4.3 Learning and Severity Prediction

The last phase of our proposed model, shown in the bottom right-side of Fig. 3 is Learning and severity prediction. Our idea is to use the document-term matrices, provided as output of the Pre-processing activity, to perform a multiple regression analysis.

Dimensional reduction aims at reducing the number of terms involved in the construction of the regression model, as a large number of features could introduce noise in the prediction. Our choice involves the use of the Principal Component Analysis (PCA), a standard tool in modern data analysis that analyze a data table in which observations are described by several inter-correlated quantitative dependent variables. PCA aims at extracting the important information from the data table, to represent it as a set of new orthogonal variables called principal components, and to display the pattern of similarity of the observations and of the variables as points in maps, as described in [24]. PCA was performed using the prcomp function in R.

PCA

```
# PCA using Mozilla data set
PCA <- prcomp(data = Mozilla)
```

where *data* parameter refers to the data to be used. We have adopted PCA also because this technique is completely non-parametric, this can be considered a positive feature as the output is unique and independent of the user. We have decided to reduce the number of dimensions of which the new data table, obtained by the application of the PCA on the data set, is composed. For this purpose, we have represented principal components and the correspondent variance expressed in a scree plot visualization. In multivariate statistics, a scree plot is a line plot of the eigenvalues[1] associated with the principal components (eigenvectors of the covariance data matrix) in an analysis. A scree plot is used to select the principal components to keep and it shows how much variation each principal component captures from the data. The y-axis is eigenvalues, which essentially stand for the amount of variation [25]. After an accurate analysis of the visualizations produced, we decided to adopt a threshold equal to 0.01, that is the 1% of the percentage of variance expressed by each principal component, as they are considered the most significant for the prediction purposes (Fig. 4).

As an alternative to PCA, we used a feature selection algorithm, the Recursive Feature Elimination (RFE), a recursive method that ranks features according to some measure of their importance; the measure we use is based on a chi-squared test.

After the dimensional reduction, we completed the resulting data tables belonging to the training and validation set, adding the Days_resolution field, extracted in the Data Collection activity, that will be used as response variable; the remaining variables will be used as independent variables.

Next activity provides for the training of the machine learning algorithms to be used. Our choice includes M5P model tree, Random Forests and Support Vector

[1]Eigenvalues are a special set of scalars associated with a linear system of equations (i.e., a matrix equation) that are sometimes also known as characteristic roots, characteristic values, proper values, or latent roots.

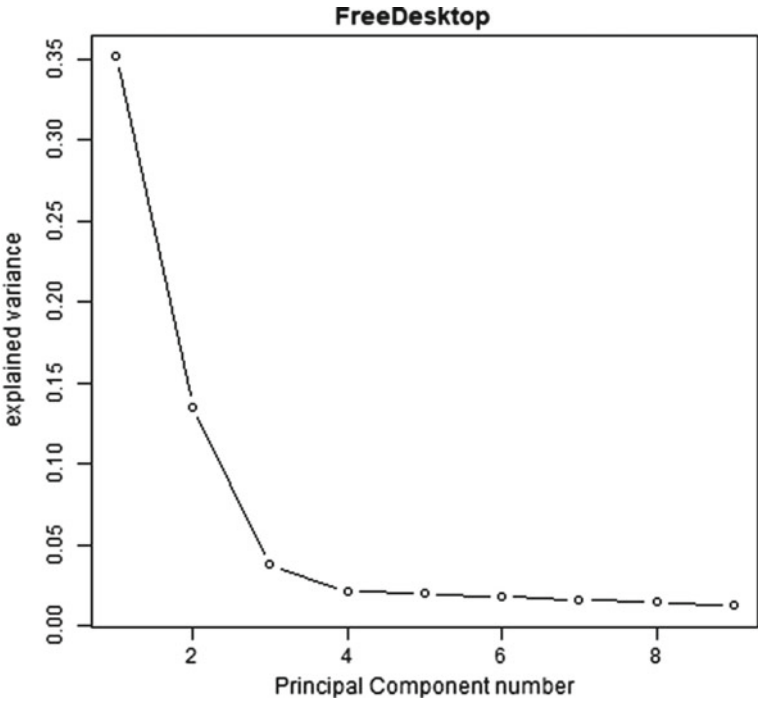

Fig. 4 Scree plot of the explained variance of the first nine principal components for FreeDesktop data set, threshold= 0.01

Machine (using polynomial and Gaussian kernels), configured for a regression analysis.

A tuning phase is necessary to estimate the optimal values to be associated with the algorithm's parameters for the prediction; for this purpose, we used the training set to train the models and the validation set to estimate the optimal values related to the models' parameters. Parameters that we estimate are the number of decision trees for the Random Forests model (values range from 500 to 1000, using 100 as an increment), the polynomial degree for the SVM with polynomial kernel (values range from 1 to 5, using 1 as an increment) and the γ coefficient for the SVM with Gaussian kernel (values range from $2^{(-3)}$ to 2^3, using 1 as an exponent increment). The M5P model tree algorithm does not provide for the use of parameters to be estimated. The criterion we have adopted, aimed at selecting the values to be associated with the related parameters, is based on the minimization of the RMSE on the validation set. Once predicted the parameters, we train the algorithms using the union of the training and validation set and we estimate the days resolution of bug reports belonging to the test set.

The following code provides instructions for M5P model tree training and prediction, using the R package "RWeka" [28].

M5P model tree training and prediction

```
#M5P model tree training
m5p <- M5P(formula = days_resolution ~ ., data = training)
# M5P model tree prediction
prediction <- predict(model = m5p, newdata = test)
```

where *formula* parameter in M5P( ) refers to a symbolic description of the model to be fitted, *model* parameter in predict( ) refers to the model to be used to carry out the prediction and *newdata* parameter refers to the data to be used to predict the response variable.

The following code provides instructions for Random Forest training and prediction, using the R package "randomForest" [29].

Random Forest training and prediction

```
# Random Forest training
rf <- randomForest(days_resolution ~ .,
data = training, ntree = value)
# Random Forest prediction
prediction <- predict(model = rf, newdata = test)
```

where, in this case, *ntree* parameter in randomForest( ) refers to the number of decision trees to be used for model training; this value, as discussed above, was estimated in the tuning phase.

The following code provides, instead, instruction for SVM with polynomial kernel training and prediction, using the R package "e1071" [30].

SVM with polynomial kernel training and prediction

```
# SVM polynomial kernel training
svr <- svm(days_resolution ~ ., data = training,
type = "eps-regression", kernel = "polynomial", degree = value)
# SVM prediction
prediction <- predict(model = svr, newdata = test)
```

where *type* parameter in svm( ) refers to the type of task for which the SVM is to be used (classification or regression), *kernel* parameter refers to the type of kernel to use (allowed values are linear, polynomial, radial and sigmoid), *degree* parameter refers to the polynomial degree to be used (exclusively for a polynomial kernel); this value, as discussed above, was estimated in the tuning phase.

Finally, the following code provides instruction for SVM with Gaussian kernel training and prediction, using the R package "e1071".

SVM with Gaussian kernel training and prediction

```
# SVM Gaussian kernel training
svr <- svm(days_resolution ~ ., data = training,
type = "eps-regression", kernel = "radial", gamma = value)
# SVM prediction
prediction <- predict(model = svr, newdata = test)
```

where *gamma* parameter (to be used exclusively for a Gaussian kernel), as discussed above, was estimated in the tuning phase. As further study we split the predicted values into Fast and Slow bins to perform a binary classification of bug reports. Metrics used for this experiment are described in Sect. 5.2.

5 Experiment

In this section, we provide the set-up of the experiment. First, we introduce the filtering step for bug reports, second, we present the metrics used, third, we show results obtained through the experiments.

5.1 Projects Selected

We extracted bug reports information from Bugzilla repositories of the following open source projects: Mozilla, FreeDesktop, NetBeans and Eclipse. Data were automatically extracted from the on-line front-end provided by Bugzilla installation, using ad-hoc web scraping routine written in PHP. Textual reports extracted were pre-processed and analyzed using the R software system [13].

Table 1 shows the total number of textual bug reports extracted for each project (F_1), having Status field assigned to VERIFIED and Resolution field assigned to FIXED. In F_2 we reported the number of discarded bug reports because corrupted and unrecoverable records, or missing XML report, or dimension being too large.

Table 1 Bug reports and projects selected

	F_1	F_2	F_3	F_4	F_5	F_6	Observation Period
Mozilla	120490	498	2000	1400	399	201	[Sept.1994–Nov.2016]
FreeDesktop	3461	2	2000	1400	399	201	[Jan.2003–Oct.2016]
NetBeans	42636	9	2000	1400	399	201	[June1999–Oct.2016]
Eclipse	45021	2	2000	1400	399	201	[Oct.2001–Nov.2016]

Table 2 Bug-fix time discretization

Label	Quantiles	Mozilla	FreeDesktop	NetBeans	Eclipse
Fast	0–0.75	0–50	0–72	0–48	0–41
Slow	0.76–1	51–2611	73–1961	49–2475	42–3795

We randomly selected, 2000 bug reports for each project (F_3), and then we split them into training, test and validation sets, considering the 70% of the sample size as training set (F_4), the 20% as test set (F_5) and the remaining 10% as validation set (F_6). Last column, Observation period, shows the data observation period for each project.

Once the prediction of the response variable related to the bug-fixing time was made, we decided to solve the same problem as a binary classification task: for this purpose, we have categorized predicted fixing-time and observed fixing-time of bug reports belonging to the test set into Fast or Slow classes, using the third quartile $q_{0.75}$ of the empirical distribution of bug-fixing time. We considered Slow as the positive class, because of its larger impact in terms of cost/effectiveness. Table 2 shows the results of binning.

It may be noted that the time taken to correct a bug shows large variations, with most of the bugs that are fixed in a relatively short time. Finally, in [32] the full collected dataset is available in JSON format.

5.2 Metrics Used

The reference metric used to evaluate the results provided by the regression analysis is the root-mean-square error (RMSE), which measure the differences between values predicted by a model and the values observed:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} (Predicted_i - Actual_i)^2}{n}} \quad (1)$$

One of the most popular performance measures in a binary classification problem is accuracy, which is defined by the ratio of the number of correct predictions and the total number of instances in the test sample. The accuracy denotes the proportion of bugs correctly predicted:

$$Accuracy = \frac{(TP + TN)}{(TP + FP + TN + FN)} \quad (2)$$

where TP (true positive) and FP (false positive) represents, respectively, the number of positive instances correctly predicted, and the number of negative instances incorrectly predicted as positive. The precision denotes the proportion of bugs correctly predicted for the positive class:

$$Precision = \frac{TP}{TP + FP} \quad (3)$$

Recall denotes the proportion of all the real positive bugs predicted for the positive class:

$$Recall = \frac{TP}{TP + FN} \quad (4)$$

5.3 Results

In the this section, we show the result set of the best configurations adopted in the experiments.

Table 3 shows, respectively, the result set for the projects Mozilla, FreeDesktop, NetBeans and Eclipse for the regression analysis experiment. The results showed that M5P model tree allows to minimize the RMSE on the relative test sets for the projects Mozilla, FreeDesktop and NetBeans. This means that currently M5P model tree allows us to estimate days resolution of bug reports belonging to that projects more accurately than the other models adopted. Regarding the Eclipse project, the model that has obtained the best performances, in terms of RMSE, is the SVM with Gaussian kernel, using parameter $\gamma = 4.00$. We can notice in most cases that the document-term matrix dimensional reduction allows to obtain satisfactory performance compared to when it is not used, in particular the Principal Component Analysis (PCA) seems to be the most effective procedure. The use of the main principal components that preserve most of the variance expressed by the data set, has contributed to obtaining more accurate estimates of days resolution. The Information Filtering procedure IF_1 has allowed to obtain better results in terms of accuracy of numerical values compared to the use of the IF_2, moreover the TF metric for the

Table 3 Regression results

Project	Inf. Filt.	Model	Parameters	Reduction	Metrics	RMSE
Mozilla	IF_1	$M5^r$	N.A.	PCA	TF	146.70
FreeDesktop	IF_1	$M5^r$	N.A.	PCA	TF	123.33
NetBeans	IF_2	$M5^r$	N.A.	N.A.	TF	152.02
Eclipse	IF_1	SVM	$\gamma = 4.00$	PCA	TF	147.13

Table 4 Classification results for SVM with polynomial kernel

Project	Inf. Filt.	Parameters	Reduction	Metrics	Accuracy	Precision	Recall
Mozilla	IF_1	degree = 1	PCA	TF-IDF	0.75	1.00	0.02
FreeDesktop	IF_1	degree = 4	RFE	TF	0.75	0.67	0.02
NetBeans	IF_2	degree = 1	PCA	TF-IDF	0.78	1.00	0.01
Eclipse	IF_1	degree = 2	RFE	TF	0.74	0.52	0.12

document-term matrices is certainly the metric that allows to obtain better performances in our regression experiment, being the one used in the result set.

Table 4 shows the result set of the same projects for the binary classification experiment, in which we have classified bug reports in Fast or Slow classes, depending on the estimated bug-fixing time during the previous experiment. Such table shows only the result set for the SVM with polynomial kernel model as it allows to obtain a higher accuracy score, metric that we consider most relevant in a classification task, on all the selected projects. Despite the good accuracy of the model (which ranges from 0.74 for Eclipse project to 0.78 for NetBeans project), we can notice a low recall, denoting that it failed to correctly predict most of Slow bugs, those bugs that require greater resources (in terms of time and human effort) for the resolution of the same. The use of the third quartile $q_{0.75}$ on the bug-fixing time discretization might have specialized models to predict Fast bugs. As we can see in the result set provided in Table 4 and in all the remaining ones here not included due to space reasons but available in [31], the accuracy score of models increases with a lower recall. The document-term matrix dimensional reduction seems to improve accuracy performance in our binary classification experiment; PCA allows to obtain a higher accuracy score when TF-IDF metric for document-term matrices is used, the same cannot be said for RFE algorithm, as it allows to obtain a higher accuracy score when TF metric is used. Moreover, also in this experiment, Information Filtering procedure IF_1 is the methodology used in the reported result set, which confirm that currently this procedure is the most effective for estimating bug-fixing time related to bug reports, for both experiments carried out.

6 Discussion and Conclusion

In this work, we tackled the problem of predicting the bug-fixing time as a result of a numerical regression problem using as predictor variables the textual information extracted from the bug reports. The data set was built with the bug reports stored in Bugzilla installations of four large open source projects like Mozilla, FreeDesktop, NetBeans and Eclipse; after the data set has been pre-processed, document-terms matrices have been generated; variables have been reduced using specific dimensional reduction techniques, considering only the most significant for the prediction purposes; they were used to train M5P model tree, Random Forest and Support Vector Machine models. Numerical estimates of bug-fixing time, provided by the regression analysis experiment, have been discretized into Fast and Slow classes, to perform a binary classification experiment of bug reports.

The obtained results show that dimensional reduction procedures, used to reduce the feature space to be included in our regression analysis, significantly affect the results. The regression experiment results are better in all the four projects involved using TF matrix, whose RMSE values ranging from 152.02 for NetBeans project, to 123.33 for FreeDesktop project. Model that provided the most accurate estimated in terms of bug resolution time was the M5P model tree. In binary classification experiment, it may be noted the unbalance of the labels in bug-fixing time discretization has allowed to achieve a high accuracy score, whose values ranging from 0.74 for Eclipse project, to 0.78 for NetBeans project; however, to these results correspond a low recall, ranging from 0.01 for NetBeans project to 0.12 for Eclipse project, which suggests that configurations characterized by a high accuracy score are not able to correctly classify bug reports belonging to the Slow class, those bugs characterized by a high-resolution time. However, it is possible to observe in the remaining result set some configurations with a discrete recall score, going to affect negatively to the general accuracy of the predictors. Model that provided the highest accuracy score of bug reports correctly classified was the SVM, using a polynomial kernel. Moreover, also in this experiment, dimensional reduction procedures significantly affect the results.

We can notice a slight improvement in evaluation measures in this work, compared with the best ones existing in similar works in the literature, where bug-fixing time has been solved as a binary classification problem: in work [6], using a sample of bug reports extracted from the Eclipse project, three classifiers have been trained to classify bug reports into Fast or Slow classes. Classifier that achieved the best performances in terms of evaluation measures was a Multivariate Bernoulli (MB) model, using a Laplace λ smoothing parameter equal to 2. This configuration achieved an accuracy score of 0.73, a precision of 0.60 and a recall of 0.04. In our work, using an innovative approach not yet tested in the bug-fixing time problem, SVM with polynomial kernel we have trained on the Eclipse data set reported an improvement in both accuracy and recall scores with respect to the mentioned work. Although the accuracy score has remained almost unchanged, we can notice an improvement in the recall, which suggests that our model is able to correctly classify more bug

reports belonging to the Slow class, those reports that have a larger impact in terms of resources spent to fix bugs.

Our proposed model, compared to the main approaches used in the literature, is able to provide numerical estimates about bug-fixing time, which constitutes one of the main advantages of our proposed approach. In Bug triage process accurate estimates on bug resolution time are fundamental for saving resources, in terms of both time and cost. However, days resolution estimates provided by our model are not yet accurate to the point of providing support for the bug triager, mainly due to the lack of some relevant information with-in bug reports. Number of developers involved in the bug-fixing process is an information not mentioned in bug reports extracted from Bugzilla; such information could help us in a better understanding of the amount of resources allocated for its resolution, moreover also fixing time in terms of working days is a missing information, for this reason it was necessary to estimate it in calendar days.

As future work, our model can also be refined, analyzing, for example, the content of the attachments in the reports, in order to enrich the feature space. Filtering outliers, that is bugs characterized by a long fixing time with respect to the average, could be a further improvement, particularly about errors due to a bad estimation of the values related to the bug-fixing time. Moreover, this process might help us to improve the recall score, as in a binary classification experiment these values are generally associated to the Slow class.

In conclusion, the prediction of the bug-fix time is a hot topic of software engineering and the results of this work, slightly better than those obtained from similar works in the literature, suggest that new and innovative ways should be explored to create a solution, reliable and highly precise, usable with extreme effectiveness in a real context.

6.1 Threats to Validity

To judge the quality of our work it is very important to consider the following threats to the validity of the study:

- **Bug-fixing time**. The actual time spent by developers as well as the distribution in terms of hours per day to fix a bug are information not publicly declared on Bugzilla. For this reason, we assumed a uniform distribution of developers work to calculate the effort spent in calendar days. These assumptions do not consider the real efforts spent by developers involved in bug-fixing.
- **Data set sampling**. Due to hardware and software limitations, data sampling was necessary. This approach could undermine the validity of the results, as it defines a sample that may not be representative with respect to bug reports extracted from Bugzilla.
- **The set of experimental projects**. We conducted our experiments on four data sets extracted from Mozilla, FreeDesktop, NetBeans and Eclipse Bugzilla BTS, respectively. These projects are not representative of the population of all open source software.

- **Non-open source projects**. We are not sure the proposed model can also be used effectively for non-open source projects, such as proprietary projects. This is because in proprietary projects a specific group is typically responsible for fixing given bugs based on corresponding features, so the current model may not apply to these kinds of projects.
- **Outliers management**. Samples extracted from collections are prone to contain bug-fix time outliers. This information can influence model training, therefore the prediction of the values related to the response variable. Removing outliers can improve the quality of the data and may have a positive impact on the model performance.
- **Field selection**. We are not sure that all the fields used to build the data set are significant, according to the bug-fixing time, so the proposed model may not be totally accurate. We decided not only to select those fields suggested by literature [3, 5] but also adding what are, in our opinion, useful to predict bug-fixing time.
- **Reopening of a closed bug**. The possibility that the same bug report can be reopened later is not considered. This event is treated as a new insertion of a report; this could invalidate, potentially, the calculation of each bug used for the data set.

Acknowledgements The research is partially supported by the POR Puglia FESR-FSE 2014-2020 - Asse prioritario 1 - Ricerca, sviluppo tecnologico, innovazione - Sub Azione 1.4.b bando innolabs - sostegno alla creazione di soluzioni innovative finalizzate a specifici problemi di rilevanza sociale - Research project KOMETA (Knowledge Community for Efficient Training through Virtual Technologies), funded by Regione Puglia.

References

1. Ardimento, P., Bianchi, A, Visaggio, G..: Maintenance-Oriented Selection of Software Components. In: Proceedings of the 8th Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04), Washington, DC, USA. IEEE Computer Society (2004)
2. Habayeb, M., Murtaza, S.S., Miranskyy, A., Bener, A.B.: On the use of hidden Markov model to predict the time to fix bugs. IEEE Trans. Softw. Eng. **44**, 1224–1244 (2018)
3. Panjer, L.D.: Predicting eclipse bug lifetimes. In: Proceedings of the 4th International Workshop on mining software repositories, p. 29 (2007)
4. Kim, S., Whitehead, Jr., E.J.: How long did it take to fix bugs? In: Proceedings of the 2006 International Workshop on Mining Software Repositories, pp. 173–174 (2006)
5. 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, pp. 52–56 (2010)
6. Ardimento, P., Bilancia, M., Monopoli, S.: Bug-fix, predicting, time: using standard versus topic-based text categorization techniques. In: Calders, T., Ceci, M., Malerba, D. (eds.) Discovery Science, DS 2016. Lecture Notes in Computer Science, vol. 9956. Springer, Cham (2016)
7. Ardimento, P., Dinapoli, A.: 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, Article 7, p. 9. ACM (2017)

8. Bugzilla installation for Mozilla. https://bugzilla.mozilla.org/. Accessed 19 July 2019
9. Bugzilla installation for FreeDesktop.org. https://bugs.freedesktop.org/. Accessed 19 July 2019
10. Bugzilla installation for NetBeans. https://netbeans.org/bugzilla/. Accessed 19 July 2019
11. Bugzilla installation for Eclipse. https://bugs.eclipse.org/bugs/. Accessed 19 July 2019
12. MongoDB, a cross-platform document-oriented database program. https://www.mongodb.com/. Accessed 19 July 2019
13. R Core Team. R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria (2018). https://www.r-project.org/. Accessed 19 July 2019
14. Bugzilla documentation. REST API Bugzilla. https://bugzilla.readthedocs.io/en/5.0/api/index.html. Accessed 19 July 2019
15. Life cycle of a bug. https://bugzilla.readthedocs.io/en/5.0/using/editing.html. Accessed 19 July 2019
16. Hooimeijer, P., Weimer, W.: Modeling bug report quality. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, pp. 34–43 (2007)
17. Anbalagan, P., Vouk, M.: On predicting the time taken to correct bug reports in open source projects. In: Proceedings of 2009 IEEE International Conference on Software Maintenance, pp. 523–526 (2009)
18. Bhattacharya, P., Neamtiu, I.: Bug-fix time prediction models: can we do better? In: Proceedings of the 8th Working Conference on Mining Software Repositories, pp. 207–210 (2011)
19. Puranik, S., Deshpande, P., Chandrasekaran, K.: A novel machine learning approach for bug prediction. Procedia Comput. Sci. **93**, 924–930 (2016)
20. D'Ambros, M., Lanza, M., Robbes, R.: An extensive comparison of bug prediction approaches. In: 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), pp. 31–41 (2010)
21. Zhang, H., Gong, L., Versteeg, S.: Predicting bug-fixing time: an empirical study of commercial software projects. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 1042–1051 (2013)
22. Akbarinasaji, S., Caglayan, B., Bener, A.: Predicting bug-fixing time: a replication study using an open source software project. J. Syst. Softw. **136**, 173–186 (2018)
23. 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, p. 11 (2011)
24. Abdi, H., Williams, L.J.: Principal component analysis. Wiley Interdiscip. Rev. Comput. Stat. **2**, 433–459 (2010)
25. Kanyongo, G.Y.: Determining the correct number of components to extract from a principal components analysis: a Monte Carlo study of the accuracy of the scree plot. J. Mod. Appl. Stat. Methods **4**(1), article 13 (2005)
26. Bouchet-Valat, M.: SnowballC: Snowball stemmers based on the C 'libstemmer' UTF-8 library. R package version 0.6.0. (2019). https://CRAN.R-project.org/package=SnowballC. Accessed 19 July 2019
27. Feinerer, I., Hornik, K.: tm: text mining package. R package version 0.7-6 (2018). https://CRAN.R-project.org/package=tm. Accessed 19 July 2019
28. Hornik, K., Buchta, C., Zeileis, A.: Open-source machine learning: R meets Weka. Comput. Stat. **24**(2), 225–232 (2009). https://doi.org/10.1007/s00180-008-0119-7
29. Liaw, A., Wiener, M.: Classification and regression by randomForest. R News **2**(3), 18–22 (2002)
30. Meyer, D., Dimitriadou, E., Hornik, K., Weingessel, A., Leisch, F: e1071: Misc functions of the department of statistics, probability theory group (Formerly: E1071), TU Wien. R package version 1.7-1. (2019). https://CRAN.R-project.org/package=e1071. Accessed 19 July 2019
31. The full result set of empirical experimentation. https://www.dropbox.com/s/s50o66pez76si7q/projectsResults.pdf?dl=0. Accessed 19 July 2019
32. The collected dataset is available, in JSON format, online under this link. https://drive.google.com/open?id=1gZP2yFYJ41xA6Vf_PQOKJrmJjNhaCFKa. Accessed 19 July 2019