Towards an Improvement of Bug Severity Classification

Nivir Kanti Singha Roy
Free University of Bozen-Bolzano
39100, Bozen-Bolzano, Italy
nivir.roy@gmail.com

Bruno Rossi
Department of Computer Systems and Communications
Masaryk University, Brno, Czech Republic
brossi@mail.muni.cz

***Abstract*—Predicting the severity of bugs has been found in past research to improve triaging and the bug resolution process. For this reason, many classification/prediction approaches emerged over the years to provide an automated reasoning over severity classes. In this paper, we use text mining together with bi-grams and feature selection to improve the classification of bugs in severe/non-severe classes. We adopt the Naïve Bayes (NB) classifier considering Mozilla and Eclipse datasets commonly used in related works. Overall, the results show that the application of bi-grams can improve slightly the performance of the classifier, but feature selection can be more effective to determine the most informative terms and bi-grams. The results are in any case project-dependent, as in some cases the addition of bi-grams may worsen the performance.**

***Keywords*—Bug Severity Classification; Text Mining; Feature Selection;**

I. Introduction

Issue reports are used within software teams to track and organize the work in software projects. In general, issue reporting systems differentiate between bug reports and request for new features (or changes), even though in many cases the structure of the two types of reports is the same[1]. A bug report contains both natural language and structured information that reports about a software failure and can be useful for developers to understand the cause to work towards the resolution. From the academic point of view, the vast amount of information available has led to the emergence of a large research community to mine information available and to reason on the bug reports. Many studies focused on bug triaging - assignment of bugs to developers (e.g., [1]), bug duplication automatic detection (e.g., [17]), effort/time estimation (e.g., [12]), failures occurrences (e.g., [14]).

In the current paper, we deal with one problem at the basis of the way in which bugs are handled by development teams: the prediction of bug severity. Bug severity is an indicator that the users provide to developers to flag the criticality of a bug - developers can then use it to set their own priority for the resoution process. This is a fundamental information needed when scheduling a bug report for resolution.

The importance of predicting severity has been shown in many related papers, for example comments, severity, and version were used in this precise order to predict the time taken to fix a defect, meaning typically that more severe bugs lead to more effort/time for the resolution process [13]. Another point was that *"...when bugs require more conversation, greater than four comments, the resolution times become dependent on severity"*[13] or that *"high severity/priority defects detected in system testing required remarkably greater mean effort (staff-hours)"*[12].

To support these statements, one of the main points is that very often reporters do not know how to deal with the severity, and as such they provide the standard severity value. For this reason, being able to suggest a severity level that complements the one given by users can be seen as invaluable for developers. From the point of view of the techniques used, the prediction of severity is done in the majority of the papers in the area by means of the analysis of the natural language text submitted as report. Prediction models are then applied to classify issues in the different severity classes to make possible the classification of new instances based on the past bug reports.

The contribution that the current paper makes is to reason beyond uni-gram models that are normally applied and provide the application of n-gram models. Uni-gram models are based on the assumption that all the terms are independent one from each other: such models will not differentiate, for example, one text containing the word *"user"* in the following sentences: *"one user reported faulty behaviour"* and *"there has been a crash in the user interface"*. A model based on uni-gram models can potentially consider the two sentences similar while for a 2-gram model, the terms *"user"* and *"user interface"* are different. This kind of reasoning goes more towards giving semantics to bug reports and the hypothesis we are testing in the current paper is that it can contribute to improve the accuracy of the results. A more formal discussion is provided in the theoretical foundations section.

Another contribution is to include feature selection also in n-gram models (as suggested by the research in [18]), to evaluate the terms that represent more information for the classifiers and improving their performance.

Our approach is to consider the application of a subset of *n-grams* (*bi-grams* or *2-grams*) in comparison to uni-grams with the further application of χ^2 feature selection for the selection of the most informative terms. To support the term representation, we consider the Naïve Bayes (NB) classifier, that has been used with success in previous research.

As such, the two main research questions we answer in the current paper are:

RQ1. Do bi-gram models provide an improvement over uni-grams in severity classification?

RQ2. What is the impact of feature selection added to bi-gram models for severity classification?

[1] in fact, in system such as TRAC (http://trac.edgewall.org), they are differentiated just by a different label

The current paper is structured as follows: Section II provides a review of related works that deal with the prediction of bugs severity and motivates the current work. Section III gives a background on the theoretical instruments that will then be applied in the implementation part of Section IV. The analysis part of the paper - Section V - provides the analysis of the Eclipse and Mozilla projects to study the application of bi-grams and feature selection with the NB classifier. Threats to validity of the study are included as well. The last part of the paper - Section VI - provides the final conclusions and lists future works.

II. Related Works

There are several works that deal with the prediction of the severity of bugs. In the seminal paper from Menzies and Marcus from 2008, authors proposed to use text mining approaches to determine the severity levels for NASA projects [11]. The approach proposed (SEVERIS, SEVERity ISsue assessment) is based on entropy and information gain, supported by a rule learner. Overall, 775 bug reports composed of 79,000 terms were used. By considering top-terms, results were in the range of 65%-98% in terms of F-Measure, depending on the different severity level. For cases with less than 30 issues results reported an F-Measure of 14% - not really usable in practice.

Another approach was based on the use of the Naïve Bayes classifier to classify issues considered as severe, non-severe with a case study based on Eclipse, Mozilla and GNOME projects [9]. The main result was that - given a training set large enough[2] - the prediction of severity levels can be quite accurate with precision and recall from 65% to 75% (Mozilla and Eclipse) and 70%-85% (GNOME).

Shorter descriptions are found to be as good in prediction as more verbose ones and the most important terms for the classification have been found to be project-specific. As well, cross component prediction seems to be as effective as intra-component prediction.

In a successive study, Lamkanfi et al. also provided a comparison of several classifiers by using their previously proposed approach [8]. Authors compared Naïve Bayes (NB), Naïve Bayes Multinomial (NBM), K-Nearest Neighbor (K-NN) and Support Vector Machines (SVM) to evaluate the performance of the classifiers within Eclipse and GNOME projects. Confirming the results of the previous study, severity-relevant terms are typically component specific. In this study, the lower bound for number of bug reports for accurate prediction is reduced to 250 reports, necessary for NB and NBM to start providing accurate predictions. Overall, the best classifier selected for each component was NBM [3] with an AUC from 0.59 to 0.93. The 1-NN classifier was the less reliable classifier. NBM was generally faster in the prediction process, relevant apect if online evaluation is considered.

As a related study, Gegick *et al.* studied the prediction of security related bug reports (SBR) versus Non-SBRs [4]. The approach was based on the SAS text miner with the aid of Singular Value Decomposition (SVD) applied to Cisco dataset. Overall, the approach allowed to identify 77% SBRs misclassified as Non-SBRs by bug reporters.

A more recent study dealt with the usage of feature selection to evaluate the benefits for severity prediction algorithms [18]. Authors compared three different feature selection schemes: Information Gain, Chi-Square, and Correlation Coefficient supported by an NBM classifier. Authors considered four open-source components from Eclipse and Mozilla projects taking into consideration the cross-component prediction difficulties. The results showed that the application of feature selection can improve the results of the prediction. Considering different weighting schemes, starting from a baseline AUC of 0.744 authors reached 0.767 (Eclipse JDT-UI) and starting from 0.763 they reached 0.776 (Eclipse UI).

In 2012, Chaturvedi and Singh performed a comparison of NB, k-NN, NBM, Support Vector Machines, J48 and RIPPER for severity prediction by using the dataset used originally in [11] and available from the PROMISE repository [2]. Authors found that the accuracy of most of the classifiers stabilizes at around 125 terms, as well the prediction accuracy fluctuates heavily according to the severity level. Contrary to the results in [8], k-NN reaches accuracy levels comparable to the ones of NB and NBM.

Still in 2012, another study used k-NN solutions to predict the severity categories of defects [15]. An interesting aspect of the approach was the usage of duplication detection to identify defects that are duplicate to potentially determine attributes that could be important for the prediction. This information was used during the application of k-NN with a similarity function based on BM25F. The experiments were performed on more than 65,000 defects from Eclipse, OpenOffice, and Mozilla. Results were up to 72% (precision), up to 76% (recall), and up to 74% (F-Measure). Another relevant feature of the study was the online evaluation, mimicking what a recommender system would do when suggesting severity of upcoming defect reports.

The latest work published in the area in 2013 - to our knowledge - had a different focus, as the prediction was made on bug priority and not severity [16]. However, the overall issues and approaches are consistent with research in the severity prediction area. Differently from other research, the approach uses linear regression with thresholding to overcome the imbalance of data. Also, features considered are not only derived by means of text mining. The final results showed that the proposed approach can give an improvement over the baseline approach in terms of F-Measure by a relative improvement of 58.61%.

As reported, there are many commonalities among the reviewed studies, like the usage of text mining techniques in all of them. One of the differences of the aforementioned approaches is the level of granularity of the severity levels, for example, some use fine-grained categories of severity [11], while others use just binary (aggregated) categories [9], [8]. There are in general few differences related to the techniques that are applied. Mostly of the studies use text mining techniques, but exceptions are also present [7].

One limitation that we address in this work is that all the approaches do not go beyond considering uni-gram models.

III. Theoretical Background

In this section, we present the theoretical foundations that are relevant for this study. The feature set of a bug report is derived by processing the textual description by means of text

[2] around 500 bug reports is the number reported by authors

[3] 11 components out of 12 - in only one case NB outperforms NBM

mining, and is the focus on this section, while the description of the NB classifier follows.

In this research, we deal with the problem of bug severity classification, that we can represent as follows. Given a binary severity class for bug reports: s_0 and s_1, where s_0 is a severe class and s_1 is non-severe, and given a training set of bug reports $b_t = (x, s_i)$ represented by a feature set x and a severity class s_i, we want to classify upcoming bug reports by means of a classifier f so that:

$$s_i = f_j(b_e) \tag{1}$$

In our case, the severity s_i is represented by a binary category, by mapping the severity levels from issue trackers[4].

The features on which we base the classification process are the terms present within the bug reports. Before applying any text mining algorithm, one major step is to represent the textual information in such a way that it can be more easily processed and information can be derived.

The term-document matrix - also known as count matrix - represents the frequency of terms that occur in a collection of documents. The ff-idf schema determines the value that each entry in the matrix should take. Each row in the matrix corresponds to a unique word or term and each column corresponds to a document. We can consider a matrix in which element *(i,j)* describes the occurrence of term *i* in document *j*. The matrix can be represented as follows [10], [6]:

$$\begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_n \end{array} \begin{array}{c} \begin{array}{cccc} br_1 & br_2 & & br_m \end{array} \\ \begin{pmatrix} a_{11} & 0 & \dots & a_{1m} \\ 0 & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & a_{nm} \end{pmatrix} \end{array} \tag{2}$$

In the count matrix, each term (t_i) is associated to the presence in bug reports (br_j) with the value of each cell (a_{ij}) to mark the number of times that word occurs in the bug report. In our case, we use a Naïve Bayes classifier, as such we can simply use the presence/absence of terms in form of a sparse matrix, so that the information stored is not the frequency, rather a boolean value.

A. Co-locations and n-Grams

A co-location is a typical expression constituted by two or more terms within the text corpora. Examples can be *"user interface"* or *"open source software"* in which the terms taken in isolation have a different meaning compared to their appearance within the tuple (or triple).

An n-gram model is based on the sequence of terms from a given text corpora that can be used to identify co-locations of terms, in which n denotes the number of terms to consider together. While the identification of the n-grams in a text can follow a so-called "moving window approach" in which new n-grams are determined by moving through the text, a more relevant problem is how to determine the most significant n-grams.

In fact, not all the co-locations are relevant and we can generally infer those that are relevant by means of a statistical test. To exemplify, considering the bi-grams case (used in this paper) and two terms T_1 and T_2, we can make a general independence assumption of the terms and their co-location:

$$p(T_1T_2) = p(T_1)p(T_2) \tag{3}$$

in which we express as $p(T_1T_2)$ the observed frequency of co-locations and as $p(T_1)p(T_2)$ the expected chance of having the appearance of the single terms. Terms that appear more often together have more chances of being co-locations of terms, rather than terms that appear rarely together.

There are different tests that can be applied to identify co-locations. In our case we considered the χ^2 (chi-square) test of independence that takes the non-parametric assumption of data distributed non-normally - very often the case with textual documents.

The χ^2 test is based on the analysis of observed and expected values, taking into account the independence assumption discussed earlier.

$$X^2 = \sum_{i,j} \frac{(O_{i,j} - E_{i,j})^2}{E_{i,j}} \tag{4}$$

Looking at the case of bi-grams, this means using a contingency matrix in which we compute the terms and their co-occurence:

	T_1	$\neg T_1$
T_2	n_{ii}	n_{oi}
$\neg T_2$	n_{io}	n_{oo}

Computing then the χ^2 value as:

$$\chi^2 = \frac{N * (n_{ii} * n_{oo} - n_{io} * n_{oi})^2}{(n_{ii} + n_{io}) * (n_{ii} + n_{oi}) * (n_{io} + n_{oo}) * (n_{oi} + n_{oo})} \tag{5}$$

Generally, a χ^2 value near to zero indicates independence among the terms. We can then compare the χ^2 value with the χ^2 distribution with one degree of freedom and eventually reject the null hypothesis of independence of the terms, so that the bi-gram can be added to the list of significant bigrams. Thresholding can be used to determine the most relevant bi-grams we want to keep for the classification.

B. The Naïve Bayes Classifier

We use the Naïve Bayes (NB) classifier as the base classifier for comparison. NB is a discriminative model that learns the conditional probability distribution $p(y|x)$, that is the probability of event y given event x. The NB classifier applies *Bayesian statistical inference rules* commonly used in *bag of words models* for text classification. Very often, the NB classifier is based on the probability of the presence and absence of a word in a textual document. In order to classify a new item, the NB classifier examines the item features independently and compares those against previous item features. For learning purposes there are two model parameter: the priori probability and the posterior probability or likelihood.

[4]as in related works, the neutral category (the standard one offered by issue trackers) is not mapped to any of the two severity levels because reporters may use it just if undecided

In the prior probability we consider the probability of an event before the evidence is observed (this means simply $p(x)$, the probability of an event x), while in the posterior probability or likelihood we compute the conditional probability of each feature value given a certain class ($p(y|x)$, that is the probability of an event after the evidence is observed).
For simplification purposes the NB classifier takes the assumption that all the features are independent, hence it is called "Naïve" Bayes classifier.

C. Measuring Performance of Classifiers

There are different ways to measure the performance of a classifier, and each metric used provides different aspects on the final results of a classifier. In the current paper, we measure the performance of the classifier by means of five different metrics: accuracy, precision, recall, Receiver Operating Curve (ROC) and Area Under the Curve (AUC) to provide a wider set of metrics for comparison.
Accuracy shows the overall correctness of the model and is calculated as the sum of correct classifications divided by the total number of classifications. Precision shows the proportion of relevant items which are retrieved, in contrast to recall that presents the fraction of recalled relevant items that are retrieved. The ROC curve shows the performance of classification from the point of view of true positive rates versus false positive ones. For classifiers like NB that assign a probability of belonging to a class to an instance, such information can be used to plot a curve that indicates the performance of the classifier over a "random-guess" model that is represented through the diagonal of the diagram. If cross-fold validation is used - as in our case - the ROC curve plotted considers the overall set of test data available from all the runs. To summarize information from ROC curves, a common metric is AUC that provides information about the size of the area under the curve. ROC curves are in general preferred over precision and recall curves as they are independent from the dataset class distribution, so that they can provide more comparable results [3].

	Evaluation focus
$accuracy = \frac{tp+tn}{tp+fn+fp+tn}$	Effectiveness of a classifier in terms of true positives and true negatives
$precision = \frac{tp}{tp+fp}$	Agreement of the classification of positive labels within the dataset
$recall = \frac{tp}{tp+fn}$	Effectiveness in the identification of positive classes
ROC	Receiver Operating Curve, plot of true positive rates versus false positive rates
AUC	Area under the ROC curve

TABLE I: Measures and diagrams used in the current paper to evaluate the performance of binary classification

IV. Implementation

In this section, we describe the operationalization of the learning infrastructure that was set-up to process and input data to the classifiers.

A. Supervised Classification Framework

The supervised classification framework we adopted is based on the properties of observed history data for the purpose of training a classifier model (Figure 1, our reference framework). *Supervised learning* is divided into two major phases: i) a training phase and ii) a testing phase. In the training phase the classifier model applies a learning algorithm to analyze the training data and produce an inferred function, which is used for mapping new unseen examples in testing phase. Key phases of the framework are also data processing, like the extraction of the features, and the application of feature selection.

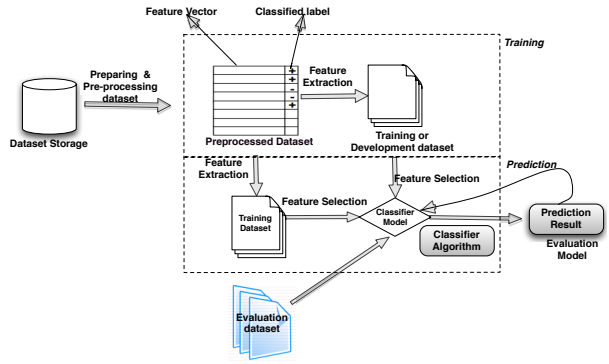

Fig. 1: Supervised Classification Framework

1) Training Phase: **Dataset Sampling, Extracting & Organizing Process.** Our text-mining application requires to access large datasets and needs to arrange the content in a structured manner. During the training phase we prepare the dataset for the analysis, extracting the features and performing the optional feature selection step.
Dataset Preprocessing involves transforming raw data into a structured format. We are running a set of standard steps for dataset pre-processing: tokenization, stop-words removal, stemming (figure 2).
Tokenization. It is a common preprocessing step to identify

Fig. 2: Dataset Preprocessing

each word (that is a token) by removing also the white spaces, punctuation, and converting the text to lower case. In our case, we used the NLTK tokenizer[5] to identify the tokens contained within a bug's description.
Stop-words Removal. In natural language processing, frequently used common constructive terms like conjunctions, verbs, adverbs, prepositions, provide very low value for semantic and statistical analysis. This is the main reason for the removal. In our case we filter the tokens by using the standard NLTK English stop-words list.
Stemming. In many cases, due to context sensitivity and

[5]http://www.nltk.org

precise interpretations in natural languages, the same word is used in different forms and different conjugations, such as *"failure, failing and failed"*, although the meaning would be the same. The objective of performing stemming is to port all the words in a common base form. Stemming is based on an heuristic process that removes the suffix of words. In our implementation, we used the Porter's algorithm to perform stemming.

Lemmatization. In contrast with Stemming, Lemmatization performs morphological analysis of words, to remove the inflectional endings of words and thus return the base of words like as in a dictionary. So, while after performing a stemming process, the base words may not maintain meaning in the language - the lemmatization ensures that lemmatized words have a common word root. In our case we used a lemmatizer based on the WordNet lexical database[6].

Splitting the dataset in Training and Testing sets. To train the classifiers we used k-fold cross validation by dividing the dataset in k=10 sets of size n/10 - where n is the number of items in the dataset. Training was performed on 9 sets and testing the remaining set, repeating this process k times[7].

There is also possibility that the training and/or testing datasets are imbalanced leading to problems with the accuracy of the results. In our case, we implemented a stratified cross-validation sampling, to ensure that the ratio between positive and negative classes is the same in each fold and the same as in the overall data set.

Another reason to use stratified sampling is that the training dataset shall include all probable dataset sample including the so-called rare items. Rare class items put heavy emphasis on learning minority classes [5].

In our case, the training and testing datasets are composed of features derived from the natural language processing of the bugs descriptions and the class is derived from the level of severity of a specific instance. The feature vector is represented by an idf matrix.

As in the related research, when mapping the categories of severity to severe and non-severe classes, we discarded the "normal" category that the submitters typically use when undecided about a certain severity level. This filtering action reduces heavily the number of bugs available for analysis.

Feature selection. A step that we performed is the selection of the features that best characterize the dataset and that are then fed into the machine learning technique. In our case, the features are the terms that more significantly distinguish among classes. At this step, we applied the χ^2 statistic to determine the terms that are more informative for the prediction process. The considerations about the calculations of the statistic are the same of the previous section, by applying feature selection we consider the most informative terms according to a given threshold, that is either the χ^2 number or the number of terms to be included. The feature selection calculation can be performed for both bi-grams and/or for single tokens.

2) Testing Phase: The testing phase deals with the application of the trained model to the part of the dataset that has been devoted to testing. The performance is evaluated by means of a confusion matrix that contains results of actual and predicted classifications done by a classifier with true positives, false positives, true negatives and false negatives. The confusion matrix is used to derive the metrics used for the evaluation of the results, namely accuracy, precision and recall.

For the creation of ROC curves, one additional information needed from a binary classifier is the probability of classification of each item in the different classes, so that true positive rates and false positive rates can be plotted. The AUC value is then computed by means of the trapezoidal approximation based on the data derived from ROC curves.

V. Analysis

We run the comparison of three different models of term-representation: *a) uni-grams*, *b) uni-grams+bi-grams*, *c) uni-grams+bi-grams+χ^2 feature selection*. For all the models we provide the results by using the NB classifier.

Dataset	Severe (s_0)			Non-severe (s_1)			Discarded
	Bugs	Words	Density	Bugs	Words	Density	Bugs
Eclipse							
CDT	680	3,523	5.18	349	1,870	5.35	4,598
JDT	1,228	6,570	5.35	982	5,725	5.82	9,649
PDE	554	2,942	5.31	288	1,657	5.75	4,809
Platform	3,674	19,198	5.22	1,573	8,541	5.42	19,472
Mozilla							
Thunderbird	4,364	26,529	6.07	1,576	10,030	6.36	13,243
Firefox	15,121	89,300	5.90	5,557	33,679	6.06	49,124
Bugzilla	960	4,961	5.16	735	3,951	5.37	2,887
Core	13,064	68,288	5.22	2,796	14,788	5.28	58,372

TABLE II: Descriptive statistics of the datasets used: number of bugs, words, density and discarded bugs

We considered the datasets from Mozilla and Eclipse study that were used in [9] even though we analyzed the whole products rather than single components. We did not consider GNOME data, as it has been found in later studies to be influenced by a possible bias in the automatic generation of issues (see [18]). We considered datasets from the following projects:

- **Mozilla[8]:** non-profit open-source software project. The Mozilla dataset contains approximately 400.000 reported bugs over the period of time from 1997-2008.
- **Eclipse[9]:** open-source project that provides an Integrated Development Environment (IDE). The Eclipse dataset contains approximately 200.000 reported bugs over the period of time from 2001-2008[10].

Table II shows the descriptive statistics of the datasets considered in our case study. The first column shows the name of the project, while the severe and non-severe columns show the count of bugs included in the analysis together with the total number of words and the density of the words per bug report. The last column includes the bugs that have been excluded from the analysis because they were in the normal

[6]http://wordnet.princeton.edu

[7]most of the related works use this approach - even though there are exceptions that use holdout validation just divide the overall set in testing and training set by a given proportion, e.g. [9]

[8]http://bugzilla.mozilla.org

[9]http://bugs.eclipse.org/bugs

[10]Both Mozilla and Eclipse datasets are available at https://github.com/ansymo/msr2013-bug_dataset

	Ug						Ug+BIg						Ug+BIg+χ^2						
Projects	AUC	acc	s_0_prec	s_0_rec	s_1_prec	s_1_rec	AUC	acc	s_0_prec	s_0_rec	s_1_prec	s_1_rec	AUC	acc	s_0_prec	s_0_rec	s_1_prec	s_1_rec	top
Eclipse CDT	0.764	0.711	0.787	0.773	0.570	0.590	0.764	**0.724**	0.803	0.773	0.587	0.627	0.856	**0.809**	0.830	0.897	0.759	0.639	150
Eclipse JDT	0.797	0.722	0.775	0.706	0.670	0.742	0.802	**0.733**	0.785	0.718	0.682	0.753	0.866	**0.789**	0.808	0.815	0.768	0.756	500
Eclipse PDE	0.847	**0.768**	0.845	0.797	0.647	0.711	0.837	0.757	0.847	0.776	0.626	0.722	0.910	**0.841**	0.875	0.888	0.776	0.749	150
Eclipse Platform	0.824	**0.769**	0.847	0.819	0.608	0.654	0.822	0.755	0.854	0.784	0.577	0.686	0.857	**0.807**	0.849	0.881	0.698	0.635	400
Mozilla Thunderbird	0.791	**0.757**	0.845	0.819	0.539	0.584	0.790	0.735	0.857	0.767	0.501	0.647	0.830	**0.796**	0.847	0.882	0.632	0.560	400
Mozilla Bugzilla	0.824	0.735	0.776	0.748	0.687	0.718	0.830	**0.743**	0.782	0.758	0.697	0.725	0.896	**0.815**	0.825	0.855	0.802	0.763	350
Mozilla Core	0.909	**0.855**	0.934	0.886	0.574	0.710	0.904	0.842	0.940	0.862	0.538	0.746	0.924	**0.880**	0.932	0.921	0.652	0.685	750
Mozilla Firefox	0.807	**0.768**	0.855	0.822	0.562	0.622	0.805	0.746	0.866	0.772	0.522	0.677	0.825	**0.787**	0.859	0.847	0.600	0.621	900

TABLE III: Results of uni-gram models compared to uni-grams+bi-grams and uni-grams+bi-grams+χ^2 feature selection

Eclipse CDT					**Eclipse JDT**			
R	**Term**	χ^2	Term	freq	**Term**	χ^2	Term	freq
1	index	57.82	icon	neg : pos = 8.4 : 1.0	eclips	120.12	hang	pos : neg = 10.9 : 1.0
2	crash	42.78	duplic	neg : pos = 8.4 : 1.0	compil	116.76	help	neg : pos = 10.7 : 1.0
3	highlight	32.29	highlight	neg : pos = 8.1 : 1.0	dialog	92.6	deadlock	pos : neg = 9.9 : 1.0
4	wrong	27.83	outlin	neg : pos = 6.6 : 1.0	javadoc	78.46	templat	neg : pos = 9.2 : 1.0
5	deprec	26.41	popup	neg : pos = 5.8 : 1.0	error	77.81	could	neg : pos = 8.7 : 1.0
6	comment	26.41	item	neg : pos = 5.8 : 1.0	messag	58.47	page	neg : pos = 8.7 : 1.0
7	project	26.03	format	neg : pos = 5.8 : 1.0	page	54.06	quickfix	neg : pos = 7.9 : 1.0
8	dialog	24.63	activ	neg : pos = 5.8 : 1.0	mnemon	52.89	keyword	neg : pos = 7.1 : 1.0
9	eclips	23.48	info	neg : pos = 5.8 : 1.0	java	52.6	deprec	neg : pos = 7.1 : 1.0
10	build	23.42	explor	neg : pos = 5.8 : 1.0	warn	51.64	crash	pos : neg = 6.6 : 1.0
Eclipse PDE					**Eclipse Plaform**			
R	**Term**	χ^2	Term	freq	**Term**	χ^2	Term	freq
1	sheet	82.02	sheet	neg : pos = 24.9 : 1.0	javadoc	366.3	javadoc	neg : pos = 25.7 : 1.0
2	cheat	82.02	cheat	neg : pos = 24.9 : 1.0	crash	305.72	deprec	neg : pos = 24.1 : 1.0
3	page	62.44	field	neg : pos = 14.7 : 1.0	eclips	281.83	descript	neg : pos = 23.8 : 1.0
4	export	58.94	section	neg : pos = 9.6 : 1.0	typo	180.08	crash	pos : neg = 19.6 : 1.0
5	dialog	36.94	locat	neg : pos = 9.6 : 1.0	prefer	159.74	constructor	neg : pos = 15.4 : 1.0
6	build	36.13	dialog	neg : pos = 9.0 : 1.0	page	134.72	guid	neg : pos = 13.2 : 1.0
7	mnemon	35.59	warn	neg : pos = 8.3 : 1.0	mnemon	119.09	cheat	neg : pos = 13.0 : 1.0
8	typo	35.59	icon	neg : pos = 8.3 : 1.0	descript	90.61	hang	pos : neg = 12.8 : 1.0
9	field	32.32	argument	neg : pos = 7.3 : 1.0	cheat	79.35	mnemon	neg : pos = 12.3 : 1.0
10	fail	31.47	text	neg : pos = 7.3 : 1.0	sheet	72.39	leak	pos : neg = 12.1 : 1.0

TABLE IV: Eclipse Projects - best terms identified by χ^2 feature selection and pos/neg frequency of terms

Mozilla Thunderbird					**Mozilla Bugzilla**			
R	**Term**	χ^2	Term	freq	**Term**	χ^2	Term	freq
1	crash	313.03	resiz	neg : pos = 17.5 : 1.0	bugzilla	255.98	messag	neg : pos = 16.1 : 1.0
2	icon	181.98	javascript	neg : pos = 15.7 : 1.0	releas	196.17	releas	pos : neg = 15.2 : 1.0
3	button	152.25	shortcut	neg : pos = 15.4 : 1.0	note	109.2	note	pos : neg = 13.2 : 1.0
4	thunderbird	143.34	strict	neg : pos = 13.8 : 1.0	fail	73.1	secur	pos : neg = 12.1 : 1.0
5	email	138.29	context	neg : pos = 13.3 : 1.0	secur	59.88	crash	pos : neg = 8.7 : 1.0
6	context	106.27	crash	pos : neg = 13.3 : 1.0	crash	50.43	warn	neg : pos = 8.3 : 1.0
7	menu	100.08	freez	pos : neg = 12.2 : 1.0	advisori	47.95	space	neg : pos = 8.3 : 1.0
8	shortcut	97.35	acceler	neg : pos = 12.0 : 1.0	link	45.16	dont	neg : pos = 7.0 : 1.0
9	dialog	90.29	area	neg : pos = 9.4 : 1.0	updat	41.71	uniniti	neg : pos = 6.5 : 1.0
10	remov	81.42	lost	pos : neg = 9.1 : 1.0	word	37.73	variabl	neg : pos = 6.5 : 1.0
Mozilla Firefox					**Mozilla Core**			
R	**Term**	χ^2	Term	freq	**Term**	χ^2	Term	freq
1	crash	1763.36	ring	neg : pos = 31.7 : 1.0	crash	2657.93	nsresult	neg : pos = 51.4 : 1.0
2	firefox	1050.41	strict	neg : pos = 22.7 : 1.0	warn	1291.92	crash	pos : neg = 50.6 : 1.0
3	button	622.96	misalign	neg : pos = 21.2 : 1.0	remov	870.18	useless	neg : pos = 42.0 : 1.0
4	icon	600.16	inconsist	neg : pos = 17.2 : 1.0	failur	725.68	deprec	neg : pos = 38.9 : 1.0
5	menu	320.46	border	neg : pos = 16.2 : 1.0	assert	696.18	cleanup	neg : pos = 34.6 : 1.0
6	text	289.96	crash	pos : neg = 13.9 : 1.0	unus	449.42	unreach	neg : pos = 32.7 : 1.0
7	hang	284.39	align	neg : pos = 13.6 : 1.0	useless	376.5	comparison	neg : pos = 31.8 : 1.0
8	favicon	259.59	typo	neg : pos = 12.5 : 1.0	roundtrip	332.63	warn	neg : pos = 26.2 : 1.0
9	titl	254.61	pad	neg : pos = 12.0 : 1.0	firefox	311.16	parenthes	neg : pos = 23.3 : 1.0
10	remov	245.59	favicon	neg : pos = 11.8 : 1.0	comparison	311.01	sign	neg : pos = 21.7 : 1.0

TABLE V: Mozilla Projects - best terms identified by χ^2 feature selection and pos/neg frequency of terms

severity level and as such they were not considered as a reliable assignment by the users. To take one exemplification from one project, Eclipse CDT, in such case we considered 2,550 bugs out of 7,148 bugs: 4,598 were discarded as they were rated by users as in the "normal" severity category. The 2,550 bugs were divided in severe (680) and non-severe (1,870). From the table we can notice how the Mozilla dataset is generally larger in terms of number of bugs (apart for the Bugzilla project). The level of density of words per bug report is generally comparable across the projects.

Performance of the models. The main results by running the NB classifier on all projects are shown in Table III. For each project we report the AUC, accuracy, positive and negative precision and severe (s_0) and non-severe (s_1) recall of all the three models. For the feature selection model, we report in the last column the number of features that have been selected by the model as the best ones. We run each model starting from 50 features and incrementing them based on increments of 50 features.

Using bi-grams and adding this information to the uni-gram model brings an improvement in two projects out of four (Eclipse), and one out of four (Mozilla). For Eclipse projects the change in accuracy is of the order of $\pm$ 1.40-1.90%, for Mozilla $\pm$ 1.12-2.82%. Projects that improve are Eclipse CDT (accuracy=0.711) that reaches an accuracy of 0.724 and Eclipse JDT (accuracy=0.722) that reaches an accuracy of 0.733, while Eclipse PDE (accuracy=0.768) and Eclipse Platform (accuracy=0.769) worsens to 0.757 and 0.755, respectively. For Mozilla projects, Bugzilla (0.735 to 0.743) is the only one to improve, while Mozilla Thunderbird (0.757 to 0.735), Core (0.855 to 0.842) and Firefox (0.768 to 0.746) worsen.

As such, for some of the projects adding bi-grams does not provide additional information to the classifier and in fact may worsen the results. The ROC curves of all the models and can help in identifying the performance according to the sensitivity to true positives and false positives ratio[11].

The inspection of the most informative bi-grams, showed many terms that are ranked according to the same χ^2 score. It can become as such difficult for stakeholders in this case to review a list of one hundred of equally ranked bi-grams. The evaluation of the results based on the change of the threshold of χ^2 score used showed also that the results improve to a certain level that keeps constant and worsens then for larger χ^2 scores. Such threshold is not the same across the projects and needs to be fine-tuned according to the project.

Looking at the ranking of the most informative uni-grams and bi-grams, this lead us to consider feature selection as the way to introduce the most significant terms for the classifier. The improvement over the uni-gram model is of the order of +4.9-13.8% for Mozilla and +2.5-10.8% for Eclipse.

We are getting good performance in the prediction levels for the whole products that were analyzed (e.g compared to [9]). Generally, in our case we reach better results when classifying severe items rather than non-severe ones, as the non-severe classes had less characterizing terms for the classification. Results are in general worse in both terms of precision and recall, while generally models based on feature selection outperform the other models.

[11]The ROC curves as well as additional materials for the current paper are available online: http://goo.gl/gZFL5i

By running stratified k-fold cross validation we observed less variance in the results compared to holdout validation that we were using in early experiments with bi-grams and the NB classifier. One aspect that we report also in the threats of validity is that the temporal distribution of the dataset can have great impact on the classification results, as the terminology used for the description of bugs may change during different periods of time.

Most informative terms. Tables IV,V show the results of the χ^2 statistic for all the projects. Together with the χ^2 values we also include the frequency within the severe (positive) and non-severe (negative) classes, providing the terms that are more frequent. This is an information that the χ^2 statistic does not provide being based on the independence of the terms.

As discussed in past research, the table shows that the terms are very likely project-specific. Giving significance to specific terms can also be difficult for projects' stakeholders: to exemplify, the name of the project is very often one of the terms that can be used to discern among classes (e.g. "Eclipse"). Interpretation about this fact can be difficult. Other interpretations are easier, like the fact that terms such as *"crash"*, *"deadlock"* can have a very high χ^2 statistic and very likely to identify severe classes.

Results for Mozilla projects are also similar: *"crash"* is very often an important term that is always more present in severe classes - as well it is easier to find similarities among most informative terms between Mozilla Thunderbird and Mozilla Firefox rather than with Mozilla Core.

We can summarize the main findings of the current paper as follows. These points answer the RQ1, RQ2 that we set-up in the introduction:

- using bi-grams in addition to uni-grams for the modelling of the textual resources brings benefits to the performance of the classifiers only in some cases - adding bi-grams may in fact reduce performance. This is context dependent and depends on the bi-grams added to the classification model;
- feature selection brings wider benefits to the performance of the classifiers, with the added benefit of a list of terms that can be used for stakeholders. We suggested to introduce bi-grams with the aid of feature selection, so that only the most informative terms are added. In the case of feature selection, as expected, the improvements are for all the projects considered independently of the features considered;
- the list of terms derived from feature selection needs some interpretations. Still, interpreting the reasons of the most informative terms can be useful to spot new issue reports as they are inserted within the system, while each project will have its own list of significant terms. For the most informative bi-grams the list can be even more difficult to use, as many of the bi-grams are tied with the same level of χ^2 statistics;

A. Threats to Validity

We have similar threats to validity to the ones reported in the related research. In particular, construct validity, internal validity, external validity and reliability are very similar to ones reported in [9] as we based our work on the same set of

assumptions, e.g. that descriptions of bugs are related to the severity level. To deal with external validity, we included the dataset used in [11], while - as already mentioned - we did not consider the GNOME data due to problems of automatic generation of some reports underlined in [18]. One limitation common to all the studies is the consideration of time within the dataset: we may build the classifier with training and test data from very different periods. We plan to investigate this effect in future works.
We have then some technique-specific threats: first of all there are alternative feature selection schemes that can be applied, [18] provides a discussion of the alternatives and performance under different conditions. A more relevant internal threat is the way in which bi-grams are computed: in our case we used standard association measure based on χ^2 statistic - different measures can yield different results. By inspection of the values derived from the different association measures we believe that in any case such differences are minimal and do not constitute a serious threat.

VI. Conclusions & Future Works

In this paper, we evaluated the introduction of bi-grams for severity prediction supported by the usage of feature selection. We observed that the introduction of bi-grams can improve the prediction accuracy, while in some cases it can worsen it. As it has been noticed in previous research, results are very often dataset-specific, and generalization is a major issue. Feature selection greatly enhances the accuracy results, and has the added benefit of providing a list of terms that can be useful for domain experts when evaluating the severity of reported bugs. Our suggestion is to introduce bi-grams with the support of feature selection to add only the most informative terms and bi-grams.
The evaluation of the most informative terms showed that there are often many bi-grams that are ranked equally, this makes even more difficult than for single terms to build a list of terms that can be useful for stakeholders. Also in this work, we evidence how the terms that derive from feature selection can be difficult to interpret for the stakeholders (e.g. the name of the project as one of the characterizing terms), and as well the list is very likely to be project-specific.
There are several future works that can be derived. In particular, we are currently evaluating the implementation of additional classifiers: Naïve Bayes Multinomial (NBM), Naïve Bayes (NB), Support Vector Machines (SVM) and Nearest Neighbour (NN) in the presence of bi-grams and feature selection. As well, we plan to evaluate the influence of temporal considerations in cross-fold validation, a factor that has not yet been considered in related research.

VII. Acknowledgments

We are thankful to Neda Nasiriani, Serge Demeyer and Ahmed Lamkanfi for sharing the datasets used in the current research work, providing also useful suggestions.

References

[1] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 361–370. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134336

[2] K. Chaturvedi and V. Singh, "Determining bug severity using machine learning techniques," in *2012 CSI Sixth International Conference on Software Engineering (CONSEG)*, 2012, pp. 1–6.

[3] T. Fawcett, "An introduction to roc analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006.

[4] M. Gegick, P. Rotella, and T. Xie, "Identifying security bug reports via text mining: An industrial case study," in *MSR*, 2010, pp. 11–20.

[5] S. Han, B. Yuan, and W. Liu, "Rare class mining: Progress and prospect," in *Pattern Recognition, 2009. CCPR 2009. Chinese Conference on*, 2009, pp. 1–5.

[6] D. Hull, "Improving text retrieval for the routing problem using latent semantic indexing," in *Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, ser. SIGIR '94. New York, NY, USA: Springer-Verlag New York, Inc., 1994, pp. 282–291. [Online]. Available: http://dl.acm.org/citation.cfm?id=188490.188585

[7] M. Iliev, B. Karasneh, M. R. V. Chaudron, and E. Essenius, "Automated prediction of defect severity based on codifying design knowledge using ontologies," in *2012 First International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, 2012, pp. 7–11.

[8] A. Lamkanfi, S. Demeyer, Q. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *2011 15th European Conference on Software Maintenance and Reengineering (CSMR)*, 2011, pp. 249–258.

[9] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *MSR*, 2010, pp. 1–10.

[10] T. K. Landauer, P. W. Foltz, and D. Laham, "An Introduction to Latent Semantic Analysis," *Discourse Processes*, no. 25, pp. 259–284, 1998.

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

[12] S. Morisaki, A. Monden, T. Matsumura, H. Tamada, and K. ichi Matsumoto, "Defect data analysis based on extended association rule mining," in *MSR*, 2007, p. 3.

[13] L. D. Panjer, "Predicting eclipse bug lifetimes," in *MSR*, 2007, p. 29.

[14] B. Rossi, B. Russo, and G. Succi, "Modelling failures occurrences of open source software with reliability growth," in *Open Source Software: New Horizons*, ser. IFIP Advances in Information and Communication Technology, P. Ågerfalk, C. Boldyreff, J. Gonzlez-Barahona, G. Madey, and J. Noll, Eds. Springer Berlin Heidelberg, 2010, vol. 319, pp. 268–280. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-13244-5_21

[15] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *2012 19th Working Conference on Reverse Engineering (WCRE)*, 2012, pp. 215–224.

[16] ——, "DRONE: predicting priority of reported bugs by multi-factor analysis." IEEE, Sep. 2013, pp. 200–209. [Online]. Available: http://pubzone.org/dblp/conf/icsm/TianLS13

[17] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 461–470. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368151

[18] C.-Z. Yang, C.-C. Hou, W.-C. Kao, and I.-X. Chen, "An empirical study on improving severity prediction of defect reports using feature selection," in *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, vol. 1, 2012, pp. 240–249.