Information Retrieval Based Nearest Neighbor Classification for Fine-Grained Bug Severity Prediction

Yuan Tian[1], David Lo[1], and Chengnian Sun[2]
[1]Singapore Management University, Singapore
[2]National University of Singapore, Singapore
{yuan.tian.2011,davidlo}@smu.edu.sg, suncn@comp.nus.edu.sg

ABSTRACT

Bugs are prevalent in software systems. Some bugs are critical and need to be fixed right away, whereas others are minor and their fixes could be postponed until resources are available. In this work, we propose a new approach leveraging information retrieval, in particular BM25-based document similarity function, to automatically predict the severity of bug reports. We investigate similar bug reports reported in the past and assign severity labels to newly reported bug reports. Duplicate bug reports are utilized to determine what bug report features, be it textual, ordinal, or categorical, are important. We focus on predicting fine-grained severity labels, namely the different severity labels of Bugzilla including: `blocker`, `critical`, `major`, `minor`, and `trivial`. Compared to existing state-of-the-art study on fine-grained severity prediction, namely the work by Menzies and Marcus, our approach brings significant improvement.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Data Mining*; D.2.7 [**Software**]: Software Engineering—*Distribution, Maintenance, and Enhancement*

General Terms

Algorithms, Experimentation

Keywords

Severity Prediction, Software Defects

1. INTRODUCTION

Software systems usually contain defects that need to be fixed after releases, and in some projects users are allowed to feedback on these defects that they encounter through bug reporting systems such as Bugzilla. With Bugzilla, users can report not only the description of the bug but also estimate the severity of the reported bugs. Unfortunately, although guidelines exist on how severity of bugs need to be assigned, the process is inherently manual that is highly dependent on the expertise of the bug reporters in assigning correct labels. Novice bug reporter might find it difficult to decide the right severity level. Developers (aka. Bugzilla assignee) can later adjust the severity [1] and use this severity information to prioritize which bugs to be fixed first.

As the number of bug reports made is large, a number of past studies have proposed approaches to help users in assigning severity labels, and development team in validating bug report severity [19, 15, 16]. All these approaches combine text processing with machine learning to assign severity labels from the textual description of the reports. Menzies and Marcus develop a machine learning approach to assign the severity labels of bug reports in NASA [19]. More recently, Lamkanfi et al. develop another machine learning approach to assign severity labels of bug reports in several Bugzilla repositories of open source projects [15]. In a later work, Lamkanfi et al. have also tried many different classification algorithms and investigate their effectiveness in assigning severity labels to bug reports [16]. Menzies and Marcus assign fine-grained labels (5 severity labels used in NASA), while Lamkanfi et al. assign coarse-grained labels (i.e., binary labels: severe and non-severe).

The bug severity prediction tools are not perfect though and there are still rooms for improvement. Menzies and Marcus reported an F-measure score of 0.14 to 0.86 for the different severity labels [19]. Lamkanfi et al. reported an F-measure score of 0.65 to 0.75 on Bugzilla reports from different software systems [15]. Thus there is a need to improve the accuracy of the prediction tools further.

In this work, we propose an information retrieval (IR)-based nearest neighbor solution to predict the severity labels of bug reports. We first measure the similarity of different bug reports and based on this similarity we recover past bug reports that are most similar to it. There are various measures that have been proposed in the information retrieval community to measure the similarity between two textual documents [23, 31, 28, 27]. Some of the popular techniques are BM25 and its extensions [27]. BM25 technique and its extensions require some parameters to be learned. We leverage bug reports that have been marked as duplicate with one another to set these parameters. Our hypothesis is that duplicate bug reports would help us to identify what features are important and what are not to measure the similarity between two bug reports. Based on a set of k nearest neighbors, the labels of these k similar bug reports are then used to decide the appropriate severity label for a new bug report.

In this work, we focus on predicting fine-grained bug severity labels. We investigate the effectiveness of our proposed approach as compared to the past studies by Menzies and Marcus [19]. Since our approach requires duplicate bug reports, we do not use the NASA data investigated by Menzies and Marcus. Rather, we analyze a large number of bug reports stored in Bugzilla of Eclipse, OpenOffice, and Mozilla. We focus on predicting five severity labels of Bugzilla namely: `blocker`, `critical`, `major`, `minor`, and `trivial`. Following the work of Lamkanfi et al. [15, 16], we do not consider the severity label `normal` as this is the default option and "many reports just did not bother to consciously assess the bug severity" [15, 16]. Thus we treat this data as unlabeled data and do not use it for our testing.

Our experiments show that we could achieve a precision, recall, and F-score of up to 75%, 73%, and 74% for predicting a particular class of severity labels. Precision measures the amount of false positives, while recall measures the amount of false negatives. F-score is the harmonic mean of precision and recall. Comparing with the state-of-the-art work on fine-grained severity level prediction, we show that for most bugs and most severity label we could improve their approach by up to more than a hundred times more accurate, especially on hard-to-predict severity labels.

The following lists our contributions:

1. We propose an information retrieval based nearest neighbor solution, by leveraging duplicate bug reports, to predict fine-grained severity labels.
2. We have experimented our solution and compare it with the state-of-the-art work over a collection of more than 65,000 bug reports from three medium-large software systems: OpenOffice, Mozilla, and Eclipse.
3. We show that we can achieve an improvement of up to more than a hundred times more accurate for fine-grained bug severity prediction, especially on hard-to-predict severity label, over the state-of-the-art work.

The structure of this paper is as follows. In Section 2, we describe some background material related to bug reporting and text pre-processing. In Section 3, we elaborate our approach. We present our experiments and their results in Section 4. We discuss related work in Section 5. We conclude and describe future work in Section 6.

2. BACKGROUND

In this section, we describe the bug reporting process, then present standard approach to pre-processing textual documents, and finally highlight $BM25F_{ext}$ to measure the similarity between structured documents.

2.1 Bug Reporting

To help improve the quality of software systems, software projects often allow users to report bugs. This is true for both open-source and closed-source software developments. Bug tracking systems such as Bugzilla are often used. Users from various locations can log in to Bugzilla and report new bugs. Users can report symptoms of the bugs along with other related information to developers. These include textual descriptions of the bug either in short form or detailed form, product and component that are affected by the bug, and the estimated severity of the bug. The format of bug reports vary from one project to another, but they typically contain the fields described in Table 1.

Developers (in particular bug triagers) would then verify these symptoms and fix the bugs. They could make adjustment on the severity of the reported bug. There are often many reports that are received and thus developers would need to prioritize as to which reports are more important than others – the severity field is useful for this purpose. As bug reporting is a distributed process, often the same bug is reported by more than two people in separate bug reports. This is known as duplicate bug report problem. The developer/triager would also need to identify these duplicate bug reports so as not to waste different developers effort in fixing the same bug.

2.2 Text Pre-Processing

Tokenization. A token is a string of characters, and includes no delimiters such as spaces, punctuation marks, and so forth. Tokenization is the process of parsing a character stream into a sequence of tokens by splitting the stream at delimiters.

Stop-Word Removal. Stop words are non-descriptive words carrying little useful information for retrieval tasks. These include link verbs such as "is", "am" and "are", pronouns such as "I", "he" and "it", etc. Our stop word list contains 30 stop words, and also common abbreviations such as "I'm", "that's", "we'll".

Stemming. Stemming is a technique to normalize words to their *ground* forms. For example, a stemmer can reduce both "working" and "worked" to "work", thus for computers to capture the similarity between the two words. We used Porter's stemming algorithm [26] to process our text.

2.3 BM25F and Its Extension

We present *BM25F*, and $BM25F_{ext}$. The first is a standard document similarity function, the latter is the extended BM25F proposed in [27] to handle longer query documents.

BM25F **Similarity Function.** BM25F is a function to evaluate the similarity between two structured documents [22, 32]. A document is structured if it has a number of fields. A bug report is a structured document as it has several textual fields, i.e., *summary* and *description*. Each of the fields in the structured document can be assigned different weight to denote its importance in measuring the similarity between two documents.

Before we proceed further, let's define a few notations. Consider a document corpus D consisting of N documents. Also, each document d has K fields. Let's denote the bag of words in the f^{th} field as $d[f]$ for $1 \leq f \leq K$.

Table 1: **Fields of Interest in a Bug Report**

Field	Description
Summ	*Summary*: Short description of the bug which typically contains only but a few words.
Desc	*Description*: Detailed description of the bug. This includes information such as how to reproduce the bug, the error log outputted when the bug occurs, etc.
Prod	*Product*: The product that is affected by the bug.
Comp	*Component*: The component that is affected by the bug.
Sev	*Severity*: The estimated impact of the bug to the workings of the software. In Bugzilla, there are several severity levels: `blocker`, `critical`, `major`, `normal`, `minor`, and `trivial`. There is also another severity level, `enhancement` which we ignore in this work, as we are not interested in feature requests but only defects.

Table 2: Examples of Bug Reports from Mozilla Bugzilla

	ID	Summary	Product	Component	Severity
1	525359	replying to an HTML message which includes a contentedtable div leaves Thunderbird compose unusable until restart (from incredimail for example)	Thunderbird	Message Compose Window	major
	543032	Impossible to answer a mail from thunderbird 3.01 after viewing an e-mail sent by Incredimail	Thunderbird	Message Compose Window	critical
2	537897	No way to select engines when setting up to use an existing account	Mozilla Services	Firefox Sync, Backend	normal
	543686	Everything is synced when logging in to an existing account	Mozilla Services	Firefox Sync, UI	normal
3	538953	Using Search bar AND a proxy with password authentication ... keeps asking the password at any key entered	Firefox	Search	normal
	544836	Proxy authentication broken while typing in the search field	Firefox	Search	major

BM25F similarity function has two primary components which assign *global* and *local* importance to words. The *global* importance of a word t is based on its inverse document frequency (IDF). This IDF score is inversely proportional to the number of documents containing a word; it is defined in Equation 1.

$$IDF(t) = log\frac{N}{N_t} \tag{1}$$

In Equation (1), N_t is the number of documents containing the word t.

Another component prescribes the *local* importance of a word t in a document d. This local importance, denoted as $TF_D(d,t)$, is defined in Equation 2). This is the aggregation of the local importance of the word t for each of document d's field.

$$TF_D(d,t) = \sum_{f=1}^{K} \frac{w_f \times occurrences(d[f],t)}{1 - b_f + \frac{b_f \times size_f}{avg\_size_f}} \tag{2}$$

In Equation (2), w_f is the weight of field f, $occurrences(d[f],t)$ is the number of times the word t occurs in field f, $size_f$ is the number of words in $d[f]$, avg_size_f is the average size of $d[f]$ for all documents in D, and b_f, which takes the value between 0 to 1, is a parameter that controls the contribution of the size of the fields to the overall score.

Based on the global and local term importance weights, given two documents d and q, each of which is a bag of words, the BM25F score of d and q is:

$$BM25F(d,q) = \sum_{t \in d \cap q} IDF(t) \times \frac{TF_D(d,t)}{k + TF_D(d,t)} \tag{3}$$

In Equation (3), the word t is common in d and q, and k, whose value is greater or equal to zero, is a parameter that controls the contribution of $TF_D(d,t)$ to the overall score. We notice that BM25F has a set of free parameters that need to be tuned: w_f and b_f for each document's field, and k. Given a document containing K fields, BM25F requires $(1+2K)$ parameters to be tuned. An optimization technique based on stochastic gradient descent has been used to tune these BM25F parameters [30].

$BM25F_{ext}$ **Similarity Function.** *BM25F* is particularly developed to compute the similarity of a short document (i.e., query) with a longer document. It is typically used for search engines, where user queries are usually short and consist of only a few words. However, bug reports are longer textual documents – the description field of a bug report can contain a few hundred words. Thus, since we want to have a similarity function that measures the similarity of two bug reports each of which are relatively long textual documents, there is a need to extend *BM25F*. Sun et al. [27] address this need by proposing $BM25F_{ext}$ which considers the term frequencies in queries; it has the following form.

$$\begin{aligned} BM25F_{ext}(d,q) &= \sum_{t \in d \cap q} IDF(t) \times \frac{TF_D(d,t)}{k + TF_D(d,t)} \times W_Q \\ \text{where } W_Q &= \frac{(l+1) \times TF_Q(q,t)}{l + TF_Q(q,t)} \end{aligned} \tag{4}$$

$$TF_Q(q,t) = \sum_{f=1}^{K} w_f \times occurrences(q[f],t) \tag{5}$$

In Equation (4), for each common word t appearing in document d and query q, its contribution to the overall $BM25F_{ext}$ score has two components: one is the product of IDF and TF_D inherited from *BM25F*; and the other is the local importance of word t in the document q – denoted as W_Q. W_Q follows the word weighting scheme of Okapi BM25 [17]. Parameter l, whose value is always greater than or equal to 0, controls the contribution of the local importance of word t in q to the overall score – if $l = 0$, then the local importance of t in q is ignored, and $BM25F_{ext}$ becomes *BM25F*.

In Equation (5), the contribution of each word t is the summation of the product of w_f, which is the weight of field f, with the number of occurrences of t in the f^{th} field of q. Different from TF_D of Equation 2, we do not perform any normalization as retrieval is being done with respect to a single fixed query – we want to find rank bug reports based on their similarity to a given query bug report.

$BM25F_{ext}$ requires an additional free parameter l in addition to those needed by *BM25F*. This brings the number of total parameters for *BM25F* to $(2 + 2K)$. These parameters can be set by following a gradient descent approach presented in [27].

3. PROPOSED APPROACH

In this section, we describe our proposed approach. We first summarize our approach. We then highlight two major components of our approach.

3.1 Overall Framework

Our framework assigns severity label to a bug report BQ in question by investigating prior bug reports with known severity labels in the pool of bug reports *BPool*. The high-level pseudocode of our approach, named IR Based Nearest Neighbour Severity Prediction Algorithm, is shown in Figure 1. The algorithm would first find the top-k nearest neighbors (Line 1) and then predict label by considering the labels of these nearest neighbors (Lines 2-3).

Our framework thus consists of two major components: similarity computation, which is an integral part of finding nearest neighbors, and label assignment. In the similarity computation component, we measure the similarity between two bug reports. We leverage duplicate bug reports as training data to assign features that are important to measure how similar two reports are. We use an extended BM25 document similarity measure for the purpose. In the label assignment component, given a bug report whose severity is to be predicted, we take the nearest k bug reports based on the similarity measure. These k bug reports are then used to predict the label of the bug report.

```
Procedure INSPect
Inputs:
BQ: Bug report in question
BPool: Historical bug report pool
Output: Predicted bug report severity label
Methods:
1: Let NNSet = Find top-K nearest neighbors of BQ in BPool
2: Let PredictedLabel = Predict label from NNSet
3: Output PredictedLabel
```

Figure 1: IR Based Nearest Neighbour Severity Prediction Algorithm

3.2 Similarity Computation

A bug report contains more than textual features, it also contains other information such as *product*, *component*, etc. We want to make use of all these features, textual and non-textual, to detect the similarity among bug reports. To do this, given two bug reports d and q, our similarity function $REP(d,q)$ is a linear combination of four features, with the following form where w_i is the weight for the i-th feature $feature_i$.

$$REP(d,q) = \sum_{i=1}^{4} w_i \times feature_i \tag{6}$$

Each weight determines the relative contribution and the degree of importance of its corresponding feature. Features that are important to measure the similarity between bug reports would have a higher score. Each of the four features along with their definitions are given in Figure 2. There are two types of features: textual and non-textual; we elaborate them in the following paragraphs.

Textual Features. The first feature of Equation (7) is the textual similarity of two bug reports based on the *summary* and *description* fields as measured by $BM25F_{ext}$ similarity function described in Section 2. The second feature is the similar to the first one, except that *summary* and *description* fields are represented by bags of bigrams (a bigram is two words that appear consecutively one after the other) instead of bags of words (or unigrams).

Non-Textual Features. The other two features have binary values (0 or 1) based on the equality of the *product* and *component* fields of d and q.

$$\begin{aligned}
feature_1(d,q) &= BM25F_{ext}(d,q) \text{ //of unigrams} \quad (7)\\
feature_2(d,q) &= BM25F_{ext}(d,q) \text{ //of bigrams} \quad (8)\\
feature_3(d,q) &= \begin{cases} 1, & \text{if } d.prod = q.prod \\ 0, & \text{otherwise} \end{cases} \quad (9)\\
feature_4(d,q) &= \begin{cases} 1, & \text{if } d.comp = q.comp \\ 0, & \text{otherwise} \end{cases} \quad (10)
\end{aligned}$$

Figure 2: Features in the Retrieval Function

The similarity function *REP* defined in Equation (6) has 16 free parameters in total. For *feature₁* and *feature₂*, we compute textual similarities of d and q over *two* fields: *summary* and *description*. Computing each of two features requires $(2 + 2 \times 2) = 6$ free parameters. Also, we need to weigh the contributions of each of the 4 features in Equation (6). Thus overall, *REP* requires $(2 \times 6 + 5) = 16$ parameters to be set. Table 3 lists all these parameters.

Table 3: Parameters in *REP*

Parameter	Description
w_1	weight of $feature_1$ (unigram)
w_2	weight of $feature_2$ (bigram)
w_3	weight of $feature_3$ (product)
w_4	weight of $feature_4$ (component)
$w_{summ}^{unigram}$	weight of *summary* in $feature_1$
$w_{desc}^{unigram}$	weight of *description* in $feature_1$
$b_{summ}^{unigram}$	b of *summary* in $feature_1$
$b_{desc}^{unigram}$	b of *description* in $feature_1$
$k_1^{unigram}$	k_1 in $feature_1$
$k_3^{unigram}$	k_3 in $feature_1$
w_{summ}^{bigram}	weight of *summary* in $feature_2$
w_{desc}^{bigram}	weight of *description* in $feature_2$
b_{summ}^{bigram}	b of *summary* in $feature_2$
b_{desc}^{bigram}	b of *description* in $feature_2$
k_1^{bigram}	k_1 in $feature_2$
k_3^{bigram}	k_3 in $feature_2$

The above metric is similar to the one proposed by Sun et al. [27] except we remove several features: one is a binary feature that compares the type of the reports: defect, enhancement, etc., another is a feature that computes the difference between the reported severity, and the other is a feature that computes the difference between the versions. Since we only consider defects, and we assume that severity label is not available, we could not use the first two of the three omitted features to compute the similarity between bug reports. We do not use the last features as we do not have the complete data which require manual crawling of the web. The parameter tuning for *REP* is based on gradient descent. We take a training set consisting of duplicate bug reports, and follow the same approach as proposed in the work by Sun et al. [27]. We include the above description to ensure that our paper is self-explanatory.

3.3 Label Assignment

Leveraging the similarity measure, we locate the top-k nearest neighbors of a bug report in question. We then aggregate the contribution of each bug report to predict the label of the bug report. We compute the weighted mean of the labels of the neighbors as the predicted label. We map the labels into integers and order them from the most severe to the least severe. The labels `blocker`, `critical`, `major normal`, `minor`, and `trivial` are mapped to 0, 1, 2, 3, 4, and 5 respectively.

Consider a set of nearest neighbors *NNSet* of a bug report *BQ*. Also let $NNSet[i]$ be the ith nearest neighbor, $NNSet[i].Label$ be the label of the ith nearest neighbor (expressed in integer), and $NNSet[i].Sim$ be the similarity of *BQ* with $NNSet[i]$. The predicted label is computed by the following formula:

$$\left\lfloor \frac{\sum_{i=0}^{k}(NNSet[i].Sim \times NNSet[i].Label)}{\sum_{i=0}^{k}(NNSet[i].Sim)} + 0.5 \right\rfloor$$

The above formula aggregates the label of each neighbor based on its similarity with the target bug report *BQ*. The higher is a neighbor similarity with *BQ*, the more powerful it is in influencing the label of *BQ*. The formula ensures that the label would fall into the range. We use the floor operation and the "+ 0.5" to round the resultant label to the nearest integer.

Remember, that we ignore the bug reports with `normal` severity from the classification process. The neighbors returned however could belong to this category. Thus, we exclude all neighbors with `normal` severity from the above formula that predicts the label of a new unknown bug report. In case, *all* the k neighbors belong to the label `normal`, we simply assign label `major` to it. By default we set the value of k to be 20.

Example. To illustrate the above, we present an example. Consider a bug report *BQ*, with top-3 neighbors N_1, N_2, and N_3 with labels 5, 4, and 3 respectively. Let the *REP* similarity score of *BQ* with each of the neighbors to be:

$$\begin{gathered} REP(BQ, N_1) = 0.5 \\ REP(BQ, N_2) = 0.45 \\ REP(BQ, N_3) = 0.35 \end{gathered}$$

The assigned label for *BQ* would then be:

$$\begin{aligned}
&= \left\lfloor \frac{\sum_{i=0}^{3}(REP(BQ,N_i) \times N_i.Label)}{\sum_{i=0}^{k}(REP(BQ,N_i))} + 0.5 \right\rfloor \\
&= \left\lfloor \frac{(0.5 \times 5 + 0.45 \times 4 + 0.35 \times 3)}{(0.5 + 0.45 + 0.35)} + 0.5 \right\rfloor \\
&= \left\lfloor \frac{(2.5 + 1.8 + 1.05)}{1.3} + 0.5 \right\rfloor \\
&= 4
\end{aligned}$$

4. EXPERIMENTS

In this section, we highlight the datasets that we use in this study, followed by our experimental settings. We then present the measures used to evaluate the approaches, followed by our results. Finally, we also mention some threats to validity.

4.1 Datasets

We chose the bug repositories of three large open source projects: OpenOffice, Mozilla and Eclipse, as the three projects have different backgrounds, implementation languages and users, which can help generalizing the conclusions of our experiments. In particular, OpenOffice is a multi-platform and multi-lingual office suite. Mozilla is a non-for-profit community producing open-source software and technologies used by other applications, such as Firefox browser and Rhino JavaScript interpreter. Eclipse is a large project aiming to build a flexible development platform for all lifecycles of software development.

We extracted three datasets from them by choosing reports submitted within a period of time. Each dataset only contains defect reports, whereas feature requests and maintenance tasks are filtered away. Table 4 details the three datasets. We construct training set by selecting the first M reports of which 200 reports are duplicates, in order to tune the parameters in the retrieval function REP. Those M reports are also used to simulate the initial bug repository for all experimental runs. The rest of the reports are used for testing the prediction approach, shown in column *Testing Reports*.

4.2 Experimental Settings

We propose an online evaluation approach that mimics how severity prediction could be used in practice. At each experimental run, we iterate through the reports in the set of testing reports in chronological order. Once we reach a report R, we apply a severity prediction tool to predict the severity label of R. This would be the recommendation given to the user/developer on the severity of the bug report. At the beginning of the next iteration, we add R and its true report to the pool of bug report *BPool* in Figure 1. After the last iteration is done, we measure how good the recommendations are.

Unfortunately, the classification based approaches employed in [19] (i.e., Severis) is slow. For less than 4,000 bug reports of OpenOffice, employing the online evaluation approach would mean re-training the classification model for around 4,000 times. This took us more than 10 hours. As the number of bug reports increases the runtime increase in a super-linear fashion as at each step in the online evaluation approach more bug reports need to be investigated to train the model. Thus, we also evaluate the existing approach in an offline manner – we take a set of bug reports that we use to train REP to train Severis. We then use Severis to assign label to the remaining set of bug reports.

We perform both offline and online evaluation for Severis on OpenOffice bugs. We show that the results of these two evaluation approaches do not differ much for Severis. We only perform offline evaluation for Severis on the other two bug report datasets: Mozilla, and Eclipse. As our approach is fast and relies on nearest neighbors, we only do the online strategy.

Table 5: Precision, Recall, and F Measure for INSPect on OpenOffice

Severity	Precision	Recall	F-Measure
`critical`	54.1%	29.2%	37.9%
`major`	75.1%	73.0%	74.0%
`minor`	51.1%	24.4%	33.0%
`trivial`	46.7%	7.1%	12.3%

4.3 Evaluation Measures

We use the standard measures of precision, recall, and F-measure for each bug report category to evaluate the effectiveness of Severis and INSPect. The same measures were used by Menzies and Marcus to evaluate Severis [19]. The definitions of precision, recall, and F-measure are given below:

$$precision = \frac{Number\ of\ severe\ reports\ correctly\ labeled}{Number\ of\ reports\ labeled\ as\ severe}$$

$$recall = \frac{Number\ of\ severe\ reports\ correctly\ labeled}{Number\ of\ severe\ reports}$$

$$F1\ Score = 2 \times \frac{precision \times recall}{precision + recall}$$

4.4 Comparison Results

We compare INSPect and Severis on the three datasets. We present the results in the following sub-sections.

4.4.1 OpenOffice Results

The result of INSPect on bug reports of OpenOffice is shown in Table 5. Different from the other three programs in OpenOffice there are only five severity levels [20]. We map them to `critical`, `major`, `normal`, `minor`, and `trivial`. Again we drop `normal` from our analysis. We note that we can predict the `critical`, `major`, `minor`, and `trivial` severity labels by an F measure of 37.9%, 74.0%, 33.0%, and 12.3% respectively. The F measure is very good for `major` severity label but is poorest for `trivial` severity label.

The result for Severis (offline) is shown in Table 6. We note that Severis can predict the `critical`, `major`, `minor`, and `trivial` severity labels by an F measure of 25.6%, 75.1%, 20.5%, and 1.2% respectively. Comparing these with the result of INSPect (in Table 5, we note that we can improve the F measure for `critical`, `minor`, and `trivial` labels by a relative improvement of 48%, 60%, and 926% respectively. For the `major` label, INSPect lose out to Severis by only 1%. Thus for OpenOffice, in general our proposed approach INSPect performs better than Severis.

Although expensive (in terms of runtime; it takes more than 10 hours to complete), we also run Severis (online) and show the result in Table 7. We notice that the result using the

Table 4: Details of Datasets

Dataset	Period		Training Reports			Testing Reports	
	From	To	#Duplicate	#All	#All - #Normal	#All	#All - #Normal
OpenOffice	2008-01-02	2010-12-21	200	2,986	617	20,438	3,356
Mozilla	2010-01-01	2010-12-31	200	4,379	1,273	68,049	16,490
Eclipse	2001-10-10	2007-12-14	200	3,312	500	175,297	43,587

Table 6: Precision, Recall, and F Measure for Severis [Offline] on OpenOffice

Severity	Precision	Recall	F-Measure
`critical`	40.7%	18.6%	25.6%
`major`	63.9%	91.0%	75.1%
`minor`	39.0%	13.9%	20.5%
`trivial`	6.7%	0.7%	1.2%

Table 7: Precision, Recall, and F Measure for Severis [Online] on OpenOffice

Severity	Precision	Recall	F-Measure
`critical`	58.5%	15.4%	24.4%
`major`	63.2%	96.2%	76.3%
`minor`	42.2%	7.7%	13.0%
`trivial`	60.0%	1.0%	2.0%

online evaluation, although requires much more computation time, does not affect performance by much. There is a small improvement in F measure for `major` and `trivial`; However, for `critical` and `minor` there is a small reduction in F measure.

4.4.2 Mozilla Results

The result of INSPect on bug reports of Mozilla is shown in Table 8. We note that we can predict the `blocker`, `critical`, `major`, `minor`, and `trivial` severity labels by an F measure of 13.9%, 65.3%, 55.8%, 22.9%, and 20.6% respectively. The F measure is very good for `critical` severity label but is poorest for `blocker` severity label.

The result for Severis is shown in Table 9. Note that we only run the offline version of Severis as the online version takes much time, and our experiment with OpenOffice shows that it does not improve performance by much. We note that Severis can predict the `blocker`, `critical`, `major`, `minor`, and `trivial` severity labels by an F measure of 0.4%, 65.1%, 59.7%, 3.4%, and 2.2% respectively. Comparing these with the result of INSPect (in Table 8, we note that we can improve the F measure for `blocker`, `critical`, `minor`, and `trivial` labels by a relative improvement of 3380%, 0.3%,

Table 8: Precision, Recall, and F Measure for INSPect on Mozilla

Severity	Precision	Recall	F-Measure
`blocker`	53.4%	8.0%	13.9%
`critical`	69.6%	61.6%	65.3%
`major`	52.0%	60.2%	55.8%
`minor`	42.0%	15.7%	22.9%
`trivial`	60.5%	12.4%	20.6%

Table 9: Precision, Recall, and F Measure for Severis on Mozilla

Severity	Precision	Recall	F-Measure
`blocker`	100%	0.2%	0.4%
`critical`	82.6%	53.7%	65.1%
`major`	43.9%	93.1%	59.7%
`minor`	50.5%	1.8%	3.4%
`trivial`	19.7%	1.1%	2.2%

Table 10: Precision, Recall, and F Measure for INSPect on Eclipse

Severity	Precision	Recall	F-Measure
`blocker`	43.6%	4.8%	8.6%
`critical`	28.2%	35.3%	31.4%
`major`	58.8%	58.4%	58.6%
`minor`	51.1%	21.7%	30.6%
`trivial`	46.6%	10.0%	16.4%

572%, and 836% respectively. For the `major` label, INSPect lose out to Severis by 6.5%. Thus for OpenOffice, in general our proposed approach INSPect performs better than Severis.

4.4.3 Eclipse Results

The result of INSPect on bug reports of Eclipse is shown in Table 10. We note that we can predict the `blocker`, `critical`, `major`, `minor`, and `trivial` severity labels by an F measure of 8.6%, 31.4%, 58.6%, 30.6%, and 16.4% respectively. The F measure is very good for `major` severity label but is poorest for `blocker` severity label.

The result for Severis is shown in Table 11. We note that Severis can predict the `blocker`, `critical`, `major`, `minor`, and `trivial` severity labels by an F measure of 0.0%, 28.5%, 56.0%, 0.2%, and 0.0% respectively. The F measure scores of Severis are zeros for `blocker` and `trivial` as it does not assign any bug report to those severity label. Comparing these with the result of INSPect (in Table 8, we note that we can improve the F measure for `blocker`, `critical`, `major`, `minor`, and `trivial` labels by a relative improvement of infinity, 10%, 5%, 15,200%, and infinity, respectively. INSPect does not lose out to Severis for any label. Thus for Eclilpse, clearly our proposed approach INSPect performs better than Severis.

4.5 Sensitivity Analysis

Our proposed approach INSPect takes in one user defined parameter k whose default value is set to 20. We want to investigate the effect of changing the parameter k on the overall accuracy of our solution. We plot the effect of varying k from 1 to 20 on F measure for predicting severity labels of

Table 11: Precision, Recall, and F Measure for Severis on Eclipse

Severity	Precision	Recall	F-Measure
`blocker`	0.0%	0.0%	0.0%
`critical`	22.3%	39.7%	28.5%
`major`	48.2%	66.8%	56.0%
`minor`	7.6%	0.1%	0.2%
`trivial`	0.0%	0.0%	0.0%

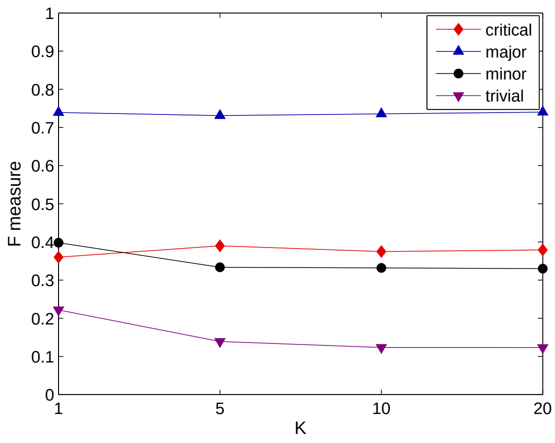

Figure 3: OpenOffice: Varying k and Its Effect on F1 Score

OpenOffice, Mozilla, and Eclipse in Figures 3, 4, & 5. When we increase k, we consider more nearest neighbors. This might increase accuracy as in effect we are tapping more to the "wisdom of the masses". However, this might also reduce accuracy as the neighbor might not be that similar anymore to the target bug report.

From the figures, for OpenOffice, the F measure scores are fairly stable as we increase k for all the severity labels. For Mozilla, the F measure scores of two severity labels: `blocker` and `trivial` decrease significantly as we increase k. This might be the case as there is more noise as more neighbors are considered. For Eclipse, similar to Mozilla, the F measure scores of two severity labels: `blocker` and `trivial` decrease significantly as we increase k.

4.6 Threats to Validity

We consider three threats of validity: threats to construct validity, threats of internal validity, and threats of external validity.

Threats to construct validity relates to the suitability of our evaluation metrics. We use standard metrics used in classification and prediction namely: precision, recall, and F measure. These measures have been used before by Menzies and Marcus to evaluate Severis [19].

Threats of internal validity refers to errors in our experiments. We extract the labels from the various Bugzilla repositories. We assume that except the `normal` label, the severity labels recorded in Bugzilla are the final severity labels that are deemed correct. We use these ground truth

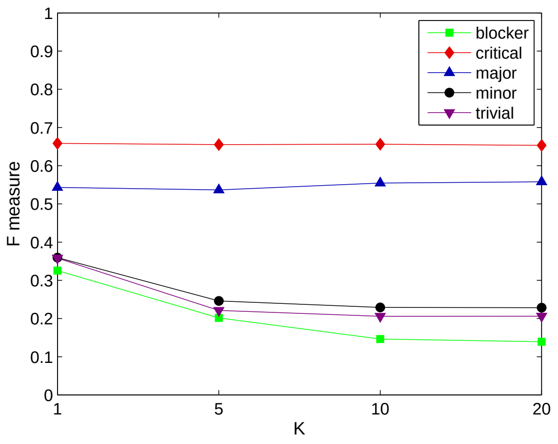

Figure 4: Mozilla: Varying k and Its Effect on F1 Score

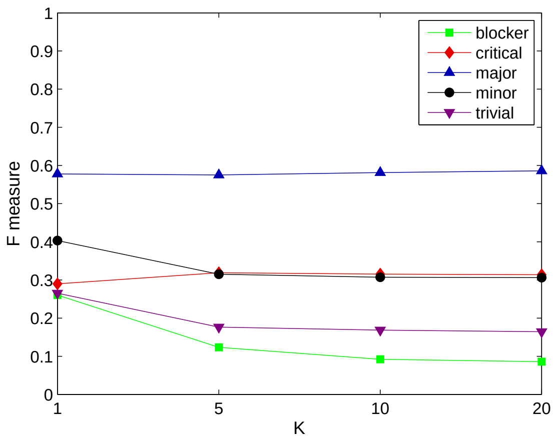

Figure 5: Eclipse: Varying k and Its Effect on F1 Score

labels to measure how good our predictions are. A similar assumption and experimental setting was also made in prior studies [15, 16].

Threats of external validity refers to the generalizability of our findings. We consider repositories of three large software systems: Eclipse, OpenOffice, and Mozilla. We consider a total of more than 65,000 bug reports. This is larger than the number of bug reports considered in prior studies [19, 15, 16]. Furthermore, the three projects are written in different programming languages, and have different background and user groups. Also, all our studies make use of open source repositories where data is publicly available. We do not use the datasets from NASA used in [19] and made available in Promise repository as they do not have information on duplicate bug reports which are typically available in Bugzilla repositories.

5. RELATED WORK

In this section, we highlight related studies on bug severity prediction, bug report analysis, and text mining for software engineering.

5.1 Past Studies on Bug Severity Prediction

There are a number of studies that predict the severity of bug reports [16, 15, 19]. We highlight these studies in the following paragraphs.

Menzies and Marcus predict the severity of bug reports in NASA [19]. They first extract word tokens from bug reports, and then perform stop word removal and stemming. Important tokens are then identified using the concept of term frequency-inverse document frequency, and information gain. These tokens are then used as features for a classification approach named Ripper rule learner [6]. Their approach is able to identify fine grained bug report labels, which are the the 5 severity levels used in NASA.

More recently, Lamkanfi et al. predict the severity of bug reports from various projects' Bugzilla [15]. They first extract word tokens and pre-process them. These tokens are then fed to a Naive Bayes classifier to predict the severity of the corresponding bug. Different from the work by Menzies and Marcus, the predict coarse grained bug severity: severe, and non-severe. Three of the six classes of severity in Bugzilla (`blocker`, `critical`, and `major`) are grouped as severe, two of the six classes (`minor`, and `trivial`) are grouped as non-severe, and the `normal` severity bugs are omitted from their analysis.

Extending the above work, Lamkanfi et al. also try out various classification algorithms to predict the severity of bugs [16]. They show that Naive Bayes perform better than other mining approaches on a dataset of 29,204 bug reports.

Our approach extends the above research studies. Similar to Menzies and Marcus's work, we detect fine grained bug report labels. Similar to the work by Lamkanfi et al. we consider bug reports on Bugzilla repositories of various open source projects. We compare our approach to their approaches on a dataset containing more than 65,000 bug reports and show that we could gain significant F measure improvements.

5.2 Other Studies Analyzing Bug Reports

In a related research area, recently a number of techniques are proposed for duplicate bug report retrieval [23, 31, 14, 28, 27]. Many of these approaches propose various ways to measure the similarity of bug reports to help developers in assigning bug reports as either duplicate or not. Runeson et al. propose a formula that considers the frequency of common words appearing in both documents as a similarity measure [23]. Wang et al. use both term frequency and inverse document frequency as a similarity measure [31]. They also consider a special situation where runtime traces are available and could be used to compute the similarity between bug reports. In practice, however, only a small minority of bug reports come with runtime traces. Jalbert and Weimer propose yet another term frequency based similarity measure [14]. Sun et al. propose a technique that leverages SVM for duplicate bug report detection [28]. In their later work, Sun et al. propose an approach to measuring the similarity of bug reports using an enhanced BM25F document similarity measure [27]. These recent advances in bug report similarity measurement could potentially be leveraged to categorize bug reports into various severity classes. Our work shows that they are indeed useful for this purpose.

Another line of research is categorization of bug reports for better maintenance. Anvik et al. [2], Cubranic and Murphy [7], Tamrawi et al. [29] propose various techniques based on text categorization to automatically assign the right developer for a new report. Huang et al. categorize bug reports into those related to capability, security, performance, reliability, requirement, and usability [12]. Pordguski et al [21] and Francis et al. [8] propose automated support for classifying reported software failures, by analyzing the corresponding execution traces, into groups, with reports of each group sharing the same or similar causes. Gegick et al. identify security bug reports using text mining [9]. The approach to some extent is similar to the work of Lamkanfi et al. [15], in that it categorizes bugs into two categories. However rather than categorizing bug into: severe and not-severe, it categorizes bug into: security-related and non-security-related.

Previous work also conducts empirical studies on bug repositories. The study [3] by Anvik et al. is focused on the characteristics of bug repositories and shows interesting findings on the number of reports that a person have submitted and the proportion of various resolutions. In the work by Sandusky et al. [24], the nature, impact and extent of a bug report network is investigated in one large F/OSS development community. Based on statistically analyzing surface features in over 27,000 bug reports in OSS projects, Hooimeijer and Weimer propose a novel descriptive model to predict the quality of bug reports [11]. Bettenburg et al. develop a standard of a good bug report by surveying the developers involved in Eclipse, Mozilla and Apache projects [4].

5.3 Text Mining for Software Engineering

There are many studies that utilize various forms of text analysis for software engineering purposes. Haiduc et al. propose a method to summarize source code to support program comprehension [10]. The work proposes an approach to extract informative yet succinct text to characterize source code entities so that developers can better understand a large piece of code. Sridhara et al. propose an approach to detect code fragments implementing high level abstractions and describe them in succinct textual descriptions [25].

Marcus and Maletic propose an approach to link documentation to source code traceability links via Latent Semantic Indexing [18]. Chen et al. has also proposed an approach to link textual documents to source code by combining several techniques including regular expression, key phrases, clustering and vector space model [5]. Huang et al. propose an approach to assess software system risk by using text mining [13]. In the paper, they utilize closed frequent itemset mining to recover risk association rules.

Similar to the above studies, we also extend text mining approach to solve problem in software engineering. Different from the above studies, we investigate a new problem namely to predict fine-grained bug report severity from its text. Our approach combines nearest neighbor search with an extension of BM25 document similarity function.

6. CONCLUSION AND FUTURE WORK

Severity labels are important for developers to prioritize bugs. A number of existing approaches have been proposed to infer these labels from the textual fields of bug reports. In this work, we propose a new approach to infer severity labels from various information available from bug reports: textual, and non-textual. We make use of duplicate bug reports to weigh the relative importance of each pieces of information or features to determine the similarity between bug reports. This similarity measure is then used in a nearest-neighbor fashion to assign severity labels to a bug report. We have compared our approach to the state-of-the-art approach on fine-grained severity prediction, namely Severis proposed by Menzies and Marcus. Extensive experiments on tens of thousands of bug reports taken from three large software systems: Eclipse, OpenOffice, and Mozilla, have been performed. The result shows that we can improve the state-of-the-art approach by up to more than a hundred times more accurate, especially on hard-to-predict severity labels.

As future work, we plan to improve the accuracy of the proposed approach further. We also plan to embed our solution into Bugzilla to let it be used by many people.

7. REFERENCES

[1] http://wiki.eclipse.org/WTP/Conventions_of _bug_priority_and_severity#How_to_set_Severity_and_Priority.

[2] J. Anvik, L. Hiew, and G. Murphy. Who should fix this bug? In *proceedings of the International Conference on Software Engineering*, 2006.

[3] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *ETX*, pages 35–39, 2005.

[4] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *SIGSOFT FSE*, pages 308–318, 2008.

[5] X. Chen and J. C. Grundy. Improving automated documentation to code traceability by combining retrieval techniques. In *ASE*, pages 223–232, 2011.

[6] W. Cohen. Fast effective rule induction. In *ICML*, 1995.

[7] D. Cubranic and G. C. Murphy. Automatic Bug Triage Using Text Categorization. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, pages 92–97, 2004.

[8] P. Francis, D. Leon, and M. Minch. Tree-based methods for classifying software failures. In *ISSRE*, 2004.

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

[10] S. Haiduc, J. Aponte, and A. Marcus. Supporting program comprehension with source code summarization. In *ICSE (2)*, pages 223–226, 2010.

[11] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *ASE*, pages 34–43, 2007.

[12] L. Huang, V. Ng, I. Persing, R. Geng, X. Bai, and J. Tian. AutoODC: Automated generation of orthogonal defect classifications. In *ASE*, 2011.

[13] L. Huang, D. Port, L. Wang, T. Xie, and T. Menzies. Text mining in supporting software systems risk assurance. In *ASE*, pages 163–166, 2010.

[14] N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In *DSN*, 2008.

[15] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals. Predicting the severity of a reported bug. In *MSR*, 2010.

[16] A. Lamkanfi, S. Demeyer, Q. Soetens, and T. Verdonck. Comparing mining algorithms for predicting the severity of a reported bug. In *CSMR*, 2011.

[17] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*, pages 232–233. Cambridge University Press, New York, NY, USA, 2008.

[18] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE*, pages 125–137, 2003.

[19] T. Menzies and A. Marcus. Automated severity assessment of software defect reports. In *ICSM*, 2008.

[20] www.openoffice.org/qa/ooQAReloaded/Docs/QA-Reloaded-ITguide.html#priorities.

[21] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering*, pages 465–475, 2003.

[22] S. Robertson, H. Zaragoza, and M. Taylor. Simple BM25 Extension to Multiple Weighted Fields. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 42–49, 2004.

[23] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *ICSE*, pages 499–510, 2007.

[24] R. J. Sandusky, L. Gasser, and G. Ripoche. Bug report networks: Varieties, strategies, and impacts in a f/oss development community. In *International Workshop on Mining Software Repositories*, pages 80–84, 2004.

[25] G. Sridhara, L. L. Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *ICSE*, pages 101–110, 2011.

[26] www.ils.unc.edu/∼keyeg/java/porter/PorterStemmer.java.

[27] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang. Towards more accurate retrieval of duplicate bug reports. In *ASE*, 2011.

[28] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *ICSE*, 2010.

[29] A. Tamrawi, T. T. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Fuzzy set-based automatic bug triaging (nier track). In *ICSE*, pages 884–887, 2011.

[30] M. Taylor, H. Zaragoza, N. Craswell, S. Robertson, and C. Burges. Optimisation methods for ranking functions with multiple parameters. In *Proceedings of the 15th ACM international conference on Information and knowledge management*, pages 585–593, 2006.

[31] 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 *ICSE*, pages 461–470, 2008.

[32] H. Zaragoza, N. Craswell, M. J. Taylor, S. Saria, and S. E. Robertson. Microsoft cambridge at trec 13: Web and hard tracks. In *TREC*, 2004.