DURFEX: A Feature Extraction Technique for Efficient Detection of Duplicate Bug Reports

Korosh Koochekian Sabor, Abdelwahab Hamou-Lhadj
Software Behaviour Analysis (SBA) Research Lab
ECE, Concordia University, Montréal (QC), Canada
{k_kooche, abdelw}@ece.concordia.ca

Alf Larsson
Senior Specialist Observability
Ericsson, Stockholm, Sweden
alf.larsson@ericsson.com

Abstract—**The detection of duplicate bug reports can help reduce the processing time of handling field crashes. This is especially important for software companies with a large client base where multiple customers can submit bug reports, caused by the same faults. There exist several techniques for the detection of duplicate bug reports; many of them rely on some sort of classification techniques applied to information extracted from stack traces. They classify each report using functions invoked in the stack trace associated with the bug report. The problem is that typical bug repositories may have stack traces that contain tens of thousands of functions, which causes the curse of dimensionality problem. In this paper, we propose a feature extraction technique that reduces the feature size and yet retains the information that is most critical for the classification. The proposed feature extraction approach starts by abstracting stack traces of function calls into sequences of package names, by replacing each function with the package in which it is defined. We then segment these traces into multiple N-grams of variable length and map them to fixed-size sparse feature vectors, which are used to measure the distance between the stack trace of incoming bug report with a historical set of bug reports stack traces. The linear combination of stack trace similarity and non-textual fields such as *component* and *severity* are then used to measure the distance of a bug report with a historical set of bug reports. We show the effectiveness of our approach by applying it to the Eclipse bug repository that contains tens of thousands of bug reports. Our approach outperforms the approach that uses distinct function names, while significantly reducing the processing time.**

Keywords-Duplicate bug reports; data mining; stack traces; software maintenance; software reliability

I. Introduction

Software systems continue to suffer from bugs that go undetected during the verification phases, causing failures and crashes. When a system crashes, users have the option to report the crash using automated bug reporting systems such as the Windows Error Reporting tool[1], the Mozilla crash reporting system[2], and Ubuntu's Apport crash reporting tool[3]. These tools capture software crash and failure data (i.e., stack traces, memory dumps, etc.) from end users. This data is sent to the software development teams to uncover the causes of the crash and provide adequate fixes.

Analyzing bug reports (BRs), however, is not an easy task, especially for software systems that have a large user base. This is due to the large number of bugs that are reported every day. For example, Eclipse Bugzilla, the bug repository for Eclipse projects, receives more than 95 bugs a day [1] and Gnome Bugzilla receives more than 120 bugs. Fortunately, not all these bugs are unique. Studies have shown that many reported bugs are duplicates—the same fault causes different running instances of the same system to crash [2, 3, 4]. Automatic detection of duplicate bug reports at early stages of the bug handling process may reduce significantly the time and effort it takes to deal with bugs [2, 3, 4].

While some studies such as the one by Jalbert et al. [4] have been conducted on the premise that duplicate bug reports should be ignored during bug triaging, other studies such as the one by Bettenburg et al. [5] believe that duplicate reports provide additional information that can be used by developers to solve the problem faster. In both cases, the detection of duplicate bug reports is useful and needed.

There exist techniques for automatic detection of duplicate bug reports (e.g., [2, 3, 4]). They treat the problem as a classification problem, by building a model from historical bug reports, using machine learning techniques. The model is used to classify incoming bugs. These techniques can be grouped into two main categories based on the features they use to characterize bug reports. The first category uses the bug description provided by the submitter. They assume that bugs with similar descriptions are potential duplicates. While bug descriptions can be useful, they remain informal and not quite reliable [3]. The second category encompasses the techniques that use stack traces (also called crash traces) and assume bugs with similar stack traces are duplicate. Stack traces contain the functions that are executed until the crash occurred. Stack traces have shown to be more reliable than bug descriptions [2]. In addition, as noted by Schroter et al. in [6], bugs that have stack traces tend to be fixed sooner.

A recent technique for detecting duplicate bug reports using stack traces (also called stack traces) was proposed by Lerch et al. [2]. The authors developed a technique for comparing stack traces of different bug reports. They considered bugs with similar stack traces as potential duplicates. They used TF/IDF (Term Frequency-Inverse Document Frequency) as a weighting mechanism. Their approach, however, suffers from scalability problems due to

[1]http://msdn.microsoft.com/en-us/library/windows/hardware/gg487440.aspx
[2]http://crash-stats.mozilla.com
[3]https://wiki.ubuntu.com/Apport

the large number of distinct functions that exist in large bug repositories.

In this paper, we propose a more efficient approach for detecting duplicate bug reports using stack traces, called DURFEX (Detection of **Du**plication **R**eports Using **F**eature **Ex**traction). DURFEX uses a multi-step feature extraction algorithm that leverages the concept of trace abstraction. More precisely, we substitute functions in historical stack traces by their package names to reduce the number of features used for building the training model. We then use the varying length N-gram of package to build feature vectors from stack traces. To detect whether an incoming bug report is a duplicate of an existing one, first we simply measure the distance between the vector of an incoming bug and the bugs in the training model then we use the linear combination of the stack trace similarity and non-textual (categorical) features to retrieve a list of potential duplicate bug reports.

The remaining parts of this paper are organized as follows: We discuss our feature extraction technique in Section II. The overall approach is presented in Section III. The experiment protocol and the detailed implementation of the approach is presented in Section IV. We discuss the results of our approach in Section V. Threats to validity are presented in Section VI, followed by related work in Section VII. We conclude the paper in Section VII.

II. PRILIMINARY

This section starts by providing a formal description of both term vector weighting strategies, term frequency and inverse document frequency (considered in related work [2]). The proposed approach for extracting feature vectors from traces of package names based on variable length N-grams is described next.

Let $T = p_1, p_2, \ldots\ldots, p_L$ be a stack trace, T of length L, where the function calls are substituted by their defining packages. The stack trace T is generated by a bug in a system with an alphabet Σ of size $m = |\Sigma|$ (unique) package names. The collection of K traces that are generated by the process (or system) of interest and then provided for designing the duplicate detection system is denoted by $\Gamma = T_1, T_2, \ldots\ldots, T_K$.

The term vector maps each trace $T \in \Gamma$ into a vector of size m packages, $T \rightarrow \emptyset(T)_{o\epsilon\Sigma}$, where each package name $p_i \in \Sigma$ in the vector is assigned a binary flag depending on its appearance (one) or not (zero) in the trace T. The term vector can be weighted by the term frequency (*tf*):

$$\phi_{tf}(p, T) = freq(p_i); i = 1, \ldots, m \qquad (1)$$

where $freq(p_i)$ is the number of times the package p_i appears in T normalized by L (the total number of package calls in T).

The term frequency considers all terms as equally important across all documents or collection of traces (Γ). However, rare terms that appear frequently in a small number of documents convey more information than those that are frequent in most documents. The inverse document frequency (*idf*) is proposed to increase (or decrease) the weights of terms that are rare (or common) across all documents. The term vector weighted by the *tf.idf* is therefore given by:

$$\phi_{tf.idf}(p, T, \Gamma) = \frac{K}{df(p_i)} freq(p_i); i = 1, \ldots, m \qquad (2)$$

where the document frequency $df(p_i)$ is the number of traces T_k in the collection of Γ of size K that contains package name p_i. A high weight in *tf.idf* is thereby given to package names that are frequent in a particular trace $T \in \Gamma$, but appear in few or no other traces of the collection, Γ.

Equation (1) and (2), both weighting strategies, discard the temporal order of packages. The proposed approach, DURFEX, however, accounts for the temporal order of package occurrences by extracting and mapping variable length N-grams and their frequencies from each trace $T \in \Gamma$, to fixed-size feature vectors. Each N-gram is a sequence of contiguous package names of length n extracted from trace T.

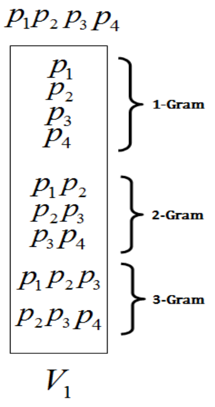

Figure. 1. Variable length N-gram

As shown in Figure 1, the feature extraction starts by sliding a window of N package names over the trace $T \in \Gamma$, shifted by one package name. For each sequence, the individual (or 1-gram) package names are first extracted, followed by all N-grams for $n = 1, \ldots, N$ that are rooted at the first package name inside the sliding window, which are then organized in vectors V_i (see Figure 1).

With the arrival of each new incoming bug report B_i, a dictionary of all variable length N-grams from the packages extracted from vectors $\Gamma = T_1, T_2, \ldots\ldots, T_K$ is constructed. The constructed dictionary has distinct alphabet $\sum$ with the size $m = |\sum|$. Each feature in the feature vector is weighted according to the frequency and inverse document frequency of that feature in N-gram features extracted from each $T \in \Gamma$, corresponding to the stack trace in the bug report.

The unique N-grams (for $n = 1, \ldots, N$) from all unique vectors V_i obtained by sliding the window over the available traces $T \in \Gamma$ are used as dictionary keys, while their accumulated frequencies are used as values. This dictionary of size, say *D*, is therefore the reference database representing the 1-grams, 2-grams, . . . , N-grams that occurred in the collection of traces *Γ*. Finally, each vector V_i (shown in Figure 1) is mapped to the space D of the reference dictionary and becomes a feature vector.

For a specific process, the size of the dictionary (D), which is the size of the unique feature vectors, depends on the alphabet size *m*, the sliding window size *N* and the regularity of the process. The value of *N* is a user-defined parameter that influences the detection power and the size of the feature vectors. A small *N* value is always desirable since it results in smaller feature vectors, and hence allows faster detection and response during operation.

For each $T \in \Gamma$, if we assume $|T|$=L then the maximum number of N-gram features or sequences of N package names that can be extracted is L– (Ngram - 1). For example the number of 3-gram features that can be extracted from a sequence of L packages is L-2. Considering that varying N-gram is the collection of features extracted by applying 1-gram, 2-gram up to N-gram, the maximum number of features that can be extracted from $|T|$=L denoted as *F* can be calculated as (3).

$$F = L + L - 1 + L - 2 + \ldots + L - (\text{N-gram} - 1)$$

$$F = N * L - \frac{N*(N-1)}{2} \qquad (3)$$

In practice, given that one package can appear more than once and overlaps of N-grams are common, the number of features is far less than what is stated in (3). According to (3), the number of features extracted from $|T|$=L is always less than the number of grams multiplied by the length of the sequence of packages in the stack trace. In our case studies, we found that the most suitable number of grams is at most 3. This shows that even by using varying length N-grams to extract feature vectors from packages, the number of features remains by far less than the number of unique functions.

The only non-null elements of the feature vectors are those that correspond to the N-grams of the original vector (V_i) before the mapping. These elements are weighted by their frequencies and inverse document frequencies that have been accumulated as values in the reference dictionary. We conducted experiments using TF-IDF.

III. PROPOSED APPROACH

DURFEX is as an online duplicate detection approach. Figure 2 shows an overview of how DURFEX is applied. For each trace in a bug report, a feature vector is constructed and the features are assigned weights.

To construct the feature vectors, we used the stack traces (by abstracting out function call into package names) as well as the categorical information such as *severity* and *component* that are associated with a bug. The use of categorical information stems from our analysis of several bug repositories (e.g., Eclipse, Gnome, Bugzilla, etc.). We observed that these two features can help identify duplicate bug reports belonging to the same group.

The output of the first phase is an adjacency matrix in which each trace is represented as a feature vector. For each incoming bug report, the stack trace is extracted and the feature vector is construed, then the constructed feature vector is compared to each existing feature vector in the adjacency matrix. The outcome of the cosine similarity is then used in a linear combination with severity and component features to retrieve the list of potential duplicate bug reports.

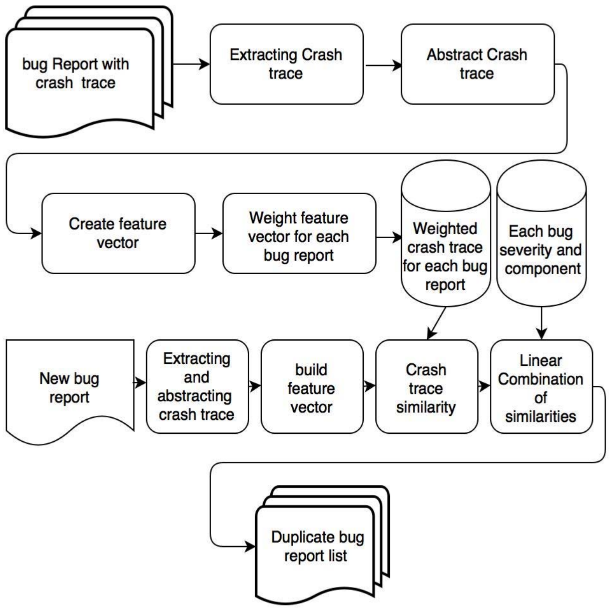
bug Report with crash trace
Extracting Crash trace
Abstract Crash trace
Create feature vector
Weight feature vector for each bug report
Weighted crash trace for each bug report
Each bug severity and component
New bug report
Extracting and abstracting crash trace
build feature vector
Crash trace similarity
Linear Combination of similarities
Duplicate bug report list

Figure 2. Proposed approach

IV. EXPERIMENT PROTOCOL

The objective of the experiment is to investigate whether or not the stack trace represented as package names instead of the raw trace of function calls can be used to detect duplicate bug reports, and if so, what would be the accuracy? More precisely, the experiment aims to answer the following questions:

RQ1. Can traces of package names alone be used to detect duplicate bug reports?

RQ2. What is the accuracy of detecting duplicate bug reports using traces of package names compared to traces of function calls?

A. The Dataset

The dataset used in this paper consists of bug reports of the Eclipse bug repository. In this repository, stack traces are embedded in the bug report description. To extract the content of the stack traces in Eclipse, we use the same regular expression presented by Lerch et al. [2].

For Eclipse, the extracted traces were preprocessed to remove noise in the data such as native methods, which are used when a call to the Java library is performed. There are also lines in the traces that are labelled ‘unknown source’. These occur due to the way debugging parameters were set.

The dataset contains Eclipse crash reports from October 2001 to February 2015. This dataset contains 455,700 reports of which 388,827 are bugs and 66,873 are labeled as enhancement (see Table 1). Among all these bugs, 42,853 had one or more stack traces in their description and, if labeled as duplicate, can be used to assess duplicate detection capability of our approach. We have a total of 8,834 duplicate bug reports that are distributed in 3,278 groups which contain at least two duplicates.

TABLE 1. CHARACTERISTICS OF THE DATASET

Data	Eclipse
Total number of reports	455,700
Total number of bug reports	388,827
Total number of enhancement reports	66,873
Total number of bug reports with stack traces	42,853
Total number of duplicate bug reports with stack traces	8,834
Total number of groups of duplicate bug reports with stack trace	3,278

B. *Dataset analysis*

In Eclipse, almost 68% of duplicate bug report groups contain only 2 bugs, while the other 32% are distributed among groups of 3 or more (see Figure 3). Pursuing this further, the fact that the duplicate bug report groups rarely contain more than 6 duplicate bugs shows that the problem of detecting duplicate bugs in the Eclipse bug repository is challenging, since, in most of the time, there exists only one duplicate bug for each bug in the repository.

To compare the difference when using package names instead of functions, we measure the total number of unique functions and packages. The result is shown in Figures 4. It shows the number of unique functions and unique packages in the Eclipse dataset between 2002 and 2014. In the bug reports submitted in 2002, the number of functions is 7670 compared to 534 packages (which represents only 7% of the number of functions). The bug reports submitted until 2104 contain 130483 unique functions defined in only 9733 packages (7.4%). Clearly, the use of package names instead of functions is a suitable abstraction level for detecting duplicate reports size-wise. We will show in the case study that the use of packages does not affect the accuracy of the approach.

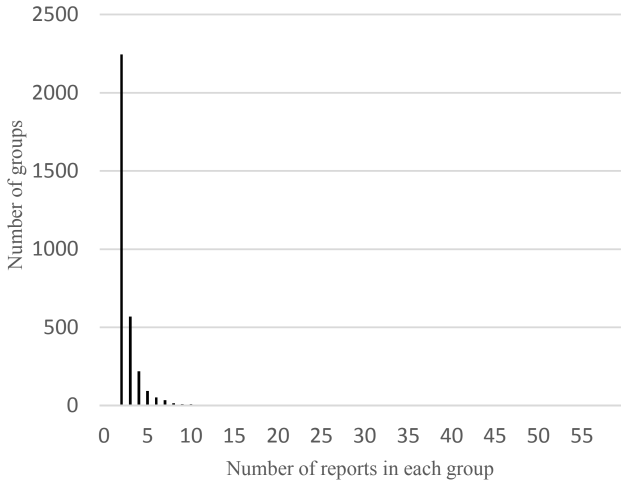

Figure. 3. The number of duplicate reports in all duplicate bugs groups in Eclipse

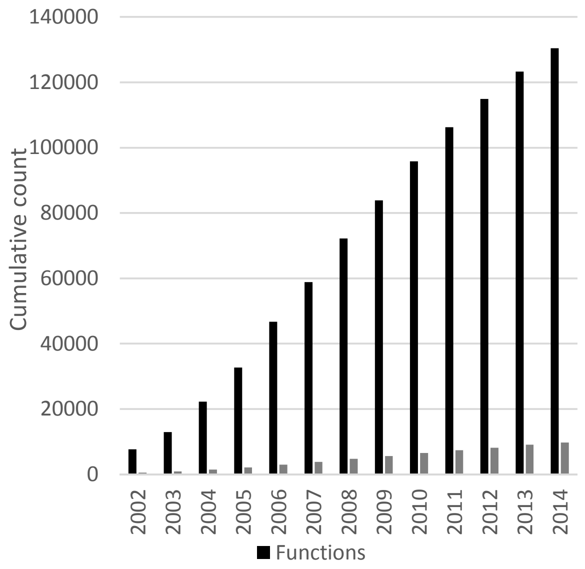

Figure 4. Cumulative number of distinct functions and packages in Eclipse

We also studied the time period in which most of the duplicate reports appear. This can help design an approach that considers only these time periods where most of the duplicate reports are submitted. For example, Runeson et al. [3] conducted a study on the Sony-Ericsson bug repository, and concluded that a time window of 50 days has the best recall rate for the Ericsson dataset. Their technique uses natural language processing of bug report descriptions.

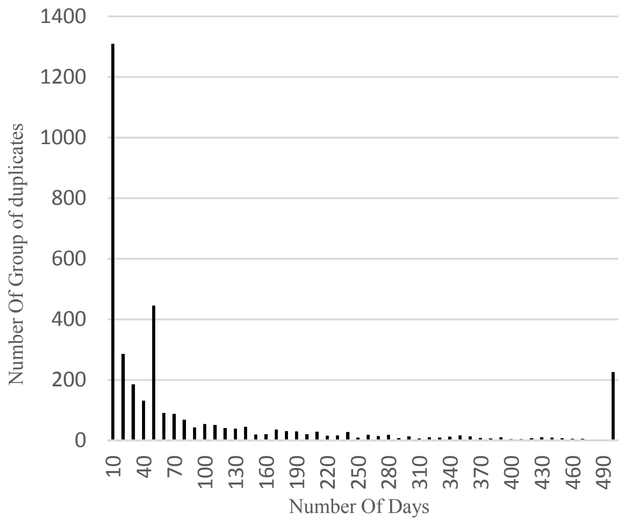

Figure. 5. Number of days between the first and last bug report in the duplicate group in Eclipse.

In the Eclipse dataset, the time period in which the duplicate bug reports are submitted is shown in Figure 5. In this figure, the time between the creation of the first and last bug report in the group is computed. As we can see, most duplicate bugs are reported within a time window of 100 days. Almost 81% of bug reports and their duplicates can be placed in a time window of 100 days. The last bar in the Figure 5 shows the number of bug reports which their distance to their duplicate is farther than 500 days.

C. Distance-based similarity

We define the similarity function between two bug reports (B_1, B_2) as follows:

$$SIM\,(B_1, B_2) = \sum_{i=1}^{3} w_i * feature_i \quad (4)$$

Where $feature_1, feature_2$ and $feature_3$ are defined as follows:

$$feature_1 = similarity\ of\ abstracted\ stack\ traces$$

$$feature_2 = \begin{cases} 1, if\ B1.components = B2.Component \\ 0, \quad otherwise \end{cases}$$

$$feature_3 = \begin{cases} 1, if\ B1.seveirity = B2.severity \\ 0, \quad otherwise \end{cases}$$

According to (4) the similarity of two bug reports is a linear combination of their stack trace and categorical features similarity.

The distance between two stack traces is measured as the distance between their corresponding vectors containing features weighted according to section 2. We use cosine similarity to measure similarity between two vectors.

Typically, given $V_1 =< w_{11}, w_{12}, \ldots.. w_{1n} >$ and $V_2 =< w_{21}, w_{22}, \ldots.. w_{2n}$, the cosine similarity is calculated as in [9]:

$$Cos(\theta) = \frac{V_1.V_2}{|V_1|.|V_2|} \quad (5)$$

The cosine similarity between two vectors, which is the cosine of the angle between two vectors, is equal to the dot product of the two vectors divided by the multiplication of their sizes.

The *SIM* function contains three free parameters (w_1, w_2, w_3) that represent the weight we assign to each feature. These weights are defined during training. More precisely, we used 10% of our dataset for training these parameters. Our training dataset format is consistent with the one presented by Sun et al. [19]. It contains triples in the form of *(q, rel, irr)* where *q* is the query bug report, *rel* is a duplicate bug report and *irr* is a bug report, which is not duplicate of *q*. The method used to create the training set is shown below (Algorithm 1) [19].

```
TS = ∅: training set
N>0 size of TS
For each Group G in the repository do
  R= all bug reports in Group G
  For each report q in R do
    For each report rel in R-{q} do
      For i=1 to N do
        Randomly choose a report irr out of R
        TS=TS ∪ {(q,rel,irr)}
      End for
    End for
  End For
End for
Return TS
```

Algorithm 1. Training technique

We need to define a cost function to optimize the free parameters (w_1, w_2, w_3) based on our training set. we define the RankNet cost function as follows [19]:

$$Y = Sim(irr,q) - sim\,(rel,q)$$

$$RNC(I) = Log(1 + e^{Y}) \quad (6)$$

The cost function is minimized when the similarity of a bug and its duplicate is maximized and the similarity of a bug report and the non-duplicate bug is minimized. We used gradient descent as shown below (Algorithm 2) [19] to minimize the above cost function.

The optimization algorithm adjusts each free parameter x in each iteration according to coefficient η and partial derivative of RNC with respect to each free parameter *x* [19].

```
TS = ∅: training set
N>0 size of TS
Ƞ: the tuning rate
For n=1 to N do
    For each instance I in TS in random order do
        For each free parameter x in sim() do
            x=x- Ƞ * $\frac{\partial RNC}{\partial x}$
        End for
    End for
End For
```

Algorithm 2. Optimization using gradient descent

D. Evaluation measure

Precision and recall as performance metrics for information retrieval techniques have been widely used. Precision and recall suit best two-class classification problems. In this study, we seek to find the duplicate of a given bug report among existing bug reports, precision and recall do not seem to be suitable metrics. Instead, similar to Runeson et al. [3] and Lerch et al. [2], in this paper, the duplicate detection is measured using the *recall rate* and *mean reciprocal rank*.

The recall rate can be defined as the percentage of duplicate bug reports from the whole duplicate bug reports set in our system, for which their duplicate bug report was found by the approach in the suggested list. If we assume that the total number of bug reports is *T*, and that the total number of duplicate bug reports for which their duplicate was in the suggested list provided by our approach as *D,* then recall rate is computed:

$$Recall\ Rate = \frac{D}{T} \qquad (7)$$

The recall rate, however, does no show the position of the duplicate BR in the suggested list of BRs. To evaluate the accuracy of the approach based on the rank of duplicate bugs in the suggested list, Lerch et al. [2] introduced the mean reciprocal rank (MRR), which is defined as:

$$MRR = \frac{1}{Q} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \qquad (8)$$

In MRR, Q is defined as the number of BRs, which have at least one duplicate in the list and $rank_i$ is the rank of the duplicate BR in the suggested list. MRR approaching 1 means that the duplicate is ranked among the first reports. MRR converges to 0 means that the detected duplicate in placed on the bottom of the list.

V. RESULTS AND DISCUSSION

In this section, we discuss the results of applying DURFEX to stack traces of the bug reports of the Eclipse bug repository. We also compare our approach to the one presented by Lerch et al. [2], where the authors used only function names with the term frequency and inverse document frequency as a weighting mechanism.

A. Comparison of packages with function calls

Figure 6 shows that in the Eclipse dataset, using only package names as features (weighed by their frequencies of occurrence and inverse document frequency) we have a recall rate of up to 84%. We can conclude that, using packages in stack traces, we are not only able to decrease the number of features (which yields lower processing time), but also improving the performance of the approach.

In Figure 6, the recall rate is best when using 2-gram sequences (86%) which outperforms using distinct function names only (81%) see Table 2.

The mean reciprocal rank (Table 3) is almost the same when using 1-gram packages and functions, but using 2-gram our approach outperforms using function calls only. Also in all case using abstracted stack traces outperforms using bug reports textual description which is consistent with Lerch et al. in [2].

TABLE 2. DURFEX RECAL RATE ON THE ECLIPSE DATASET

Approach	List Size					
	2	5	10	15	20	25
Packages (1-gram)	58	70	77	81	83	84
Packages (2-gram)	62	73	80	82	84	86
Packages (3-gram)	62	73	80	82	84	86
Packages (4-gram)	62	71	79	81	83	85
Packages (5-gram)	62	71	78	80	82	84
Functions Only	65	73	76	78	80	81
Words	44	52	58	63	65	68

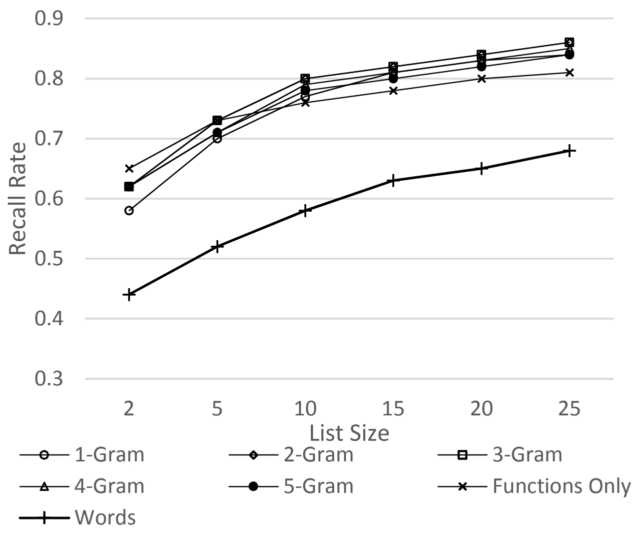

Figure. 6. Recall rate of DURFEX with different N-grams compared to the use of unique functions in Eclipse dataset.

B. Comparison of package with function calls with a time window

As we discussed earlier, the number of days between the submission of a bug report and its duplicate is less than 100 days in 81% of cases in the Eclipse dataset. This result encouraged us to study the effect of having a time window when detecting duplicate bugs. This was also used by Runeson et al. in [3].

TABLE 3. DURFEX MEAN RECIPROCAL RANK ON THE ECLIPSE DATASET

Approach	Eclipse without a time window
DUEFEX (1-gram)	0.65
DUEFEX (2-gram)	0.68
DUEFEX (3-gram)	0.68
DUEFEX (4-gram)	0.68
DUEFEX (5-gram)	0.68
Function calls	0.64
BR descriptions	0.58

Figure 7 (Table 4) shows the results of applying DURFEX to the Eclipse dataset comprising bug reports that appear only in the time window of 100 days before the submission of new bug report. The results show that DURFEX improved compared to when using the whole dataset. The recall rate obtained by using package names with 2-gram and 3-gram is 89%, which is better than the results obtained when using functions only.

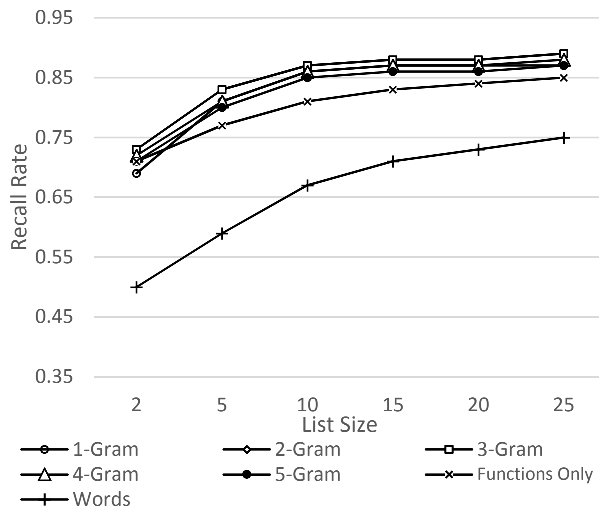

Figure. 7. Recall rate of DURFEX with different N-grams compared to the use of distinct functions in the 100-day-time period in Eclipse

In Table 5, the mean reciprocal rank is best when using 2-gram or 3-gram (73%) which outperforms using distinct functions only (59%) and using bug report description.

These findings are very promising since they suggest that we can detect duplicate bug reports using an abstract trace (in our case a trace of packages) instead of raw traces (this answers RQ1) with a better accuracy and lower processing time overhead. In the next subsection, we will show that the gain in terms of the execution time is huge. DURFEX is considerably more efficient that an approach that uses functions only.

C. Comparison of processing time using functions and packages (libraries)

To show the performance of DURFEX compared to an approach that only uses function calls such as the one from Lerch et al. [2], we measured the time it takes to find a duplicate bug report using DURFEX and compare it to the time it takes to find a duplicate report using Lerch's approach. Table 6 shows the average execution time when

TABLE 4. DURFEX RESULTS ON THE BUG REPORTS OF THE 100-DAY PERIOD ON ECLIPSE

Approach	List Size					
	2	5	10	15	20	25
Packages (1-gram)	69	81	86	87	87	87
Packages (2-gram)	73	84	87	88	88	89
Packages (3-gram)	73	83	87	88	88	89
Packages (4-gram)	72	81	86	87	87	88
Packages (5-gram)	71	80	85	86	86	87
Functions Only	71	77	81	83	84	85
Words	55	65	72	76	78	80

TABLE 5. DURFEX MEAN RECIPROCAL RANK ON THE BUG REPORTS OF THE 100-DAY PERIOD ON ECLIPSE

Approach	Eclipse without a time window
DUEFEX (1-gram)	0.70
DUEFEX (2-gram)	0.73
DUEFEX (3-gram)	0.73
DUEFEX (4-gram)	0.73
DUEFEX (5-gram)	0.73
Function calls	0.59
BR descriptions	0.58

applying DURFEX to Eclipse dataset (2002-2014), compared to an approach that uses functions (Figures 8 shows detailed results for each year of the dataset for Eclipse). As we can see, with 1-gram DURFEX execution time represents only 7% Eclipse of the execution time of an approach that uses functions as features. This is a gain of 93%. When used with 2-gram (that provide best detection accuracy as discussed in the previous section), DURFEX execution time represents 32% of the execution time of an approach that uses functions. In other words, DURFEX (with 2 gram) runs in average, around 70% faster than an approach that uses functions only. This addresses the research question RQ2. Finally, we want to note that when

DURFEX is used with 4 and 5 grams, we start noticing a decrease in processing time gap. This is expected since the larger the N-gram the more features we have, which results in longer time in processing duplicate bug reports.

Combined with the results shown in the previous subsection, we demonstrated that abstract traces of package (library) names can be used to detect bug reports with good accuracy (sometime up to 89% if a time window is used) while keeping the execution time considerably low. We believe that this makes DURFEX a practical approach for detecting duplicate bug reports.

TABLE 6. COMPARING THE AVERAGE EXECUTION TIME IN SECONDS OF DURFEX TO THE EXECUTION TIME OF AN APPROACH THAT USES FUNCTION CALLS APPLIED IN GNOME AND ECLIPSE. THE PERCENTAGE INDICATES THE RATIO OF THE EXECUTION TIME OF DURFEX TO THE EXECUTION TIME TAKEN BY AN APPROACH THAT USES FUNCTION CALLS ONLY

Approach	**Eclipse**
Use of function calls	5660.71
DURFEX (1-gram)	400.6 (7%)
DURFEX (2-gram)	1866.42 (32%)
DURFEX (3-gram)	4511.89 (79%)
DURFEX (4-gram)	8382.42 (148%)
DURFEX (5-gram)	13442 (237%)

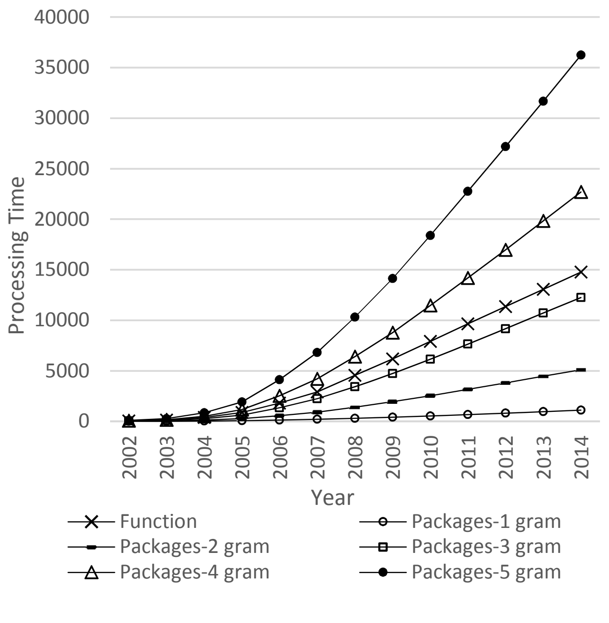

Figure. 8. Comparison of processing time of in Eclipse dataset

VI. THREATS TO VALIDITY

The regular expression used may miss some stack traces causing false negatives. Using a different parser or a better regular expression may affect the number of available stack traces and thus may change the recall rate.

In our dataset, the completeness of stack traces can be a threat to validity. On the ground that stack traces are copied by users who are not necessarily experienced, only a section of the traces that may seem to be important may have been copied by users in the description section of the bug report. Thus, relying on stack traces not provided by the system might have misled our approach at some points.

In this paper, we used Eclipse dataset. To generalize our approach, we need to apply it to more datasets (both public and proprietary).

VII. REALTED WORK

In this section, we present the main studies that focus on the detection of duplicate bug reports with an emphasis on those that use stack traces. Unlike existing approaches, DURFEX extracts abstractions from raw traces before using them which, as we showed in the case study, reduces considerably, the number of features used by a classifier.

Brodie et al. in [12] used call stacks to detect duplicate bug reports. They used a modified version of the Needleman algorithm, which is an algorithm for biological sequence matching to calculate similarity of frame sequences. They argued that although gaps or insertions are common among similar stack traces, substitution is very rare. They argued that if the top four functions are not the same, the probability of two stack traces being related to the same fault is small.

Modani et al. [7] presented an approach for detecting recurrent fault using stack traces. They reduce the size of traces by removing entry level function, error routines, and redundant recursive calls. They compared three different approaches including their own which was similar to Brodie et al. in [12], where the authors used the longest common subsequence and prefix matching algorithms. They used function frequencies and a supervised learning method to eliminate uninformative functions from stack traces. They showed that hashing the top J functions may cause reduction in the recall rate, so they proposed an alternative approach called inverse indexing.

Runeson et al. in [3] used term frequency and inverse document frequency to detect duplicate bug reports applied to the description part of the bug reports in the bug repository of the Sony-Ericsson company. They defined an accuracy metric for detecting duplicate bug reports called the recall rate, which we used in this paper.

Bettenburg et al. in [13] proposed a tool, called Infozilla, to extract structural information from the description of bug reports in Eclipse Bugzilla. Their tool is able to extract patches, stack traces, source codes and enumerations that are in the bug reports description. They showed that the accuracy of their tool can reach up to 97% in all cases. Infozilla can be used to extract features from bug reports that can later be used for the detection of bug reports.

Bartz et al. [14] extracted seven features from call stacks to calculate their similarity. They calculated similarity of bug reports as a whole by extracting four features from bug reports including call stack similarity as one feature. According to their results, insertions and deletions that caused to create or remove a new frame group should be penalized more than other types of insertions and deletions. Module substitution should be far more severely penalized than the other edits. They concluded that calls stack similarity is much more practical for detecting duplicate reports than other features.

Jalbert and Weimer [4] used a linear regression model to detect duplicate bug reports among 29000 bugs in Mozilla project bug repository. They used term frequency emphasizing that inverse document frequency does not increase the performance. Their features are composed of the bug report description, clustering results and categorical information. They trained a linear model on the first half of the data and chose a cut of value based on triage and miss effort. Having the linear model trained, they were able to successfully filter out 8% to 10% of the duplicate bug reports. They concluded that the title and description are the most important features.

Bettenburg et al. [5] challenged the theory of filtering bug reports. They quantified information provided by the master bug report and duplicate bug reports. They concluded that duplicate bug reports provide useful information to fix the master bug report and should not be filtered.

Wang et al. [15] used bugs extracted from bug repositories of Eclipse and Firefox. They criticized the sole use of description of the bug reports in duplicate detection process. They showed that two duplicate bug reports can contain less common words than two bugs that are not duplicate. To increase duplicate detection accuracy, they suggest combining description similarity with execution traces similarity.

Schroter et al. in [6] did a study on three bug repositories including Apache, Eclipse and Mozilla. They found that 60% of fixed bug reports that contain stack traces are fixed in one of the frames in the stack traces. They reported that 90% of bugs are fixed in the ten functions located in the top of stack traces.

Modani et al. [10] proposed an approach for the detection of duplicate bug reports using a character-based N-gram model. They studied the effect of using lower level features. They used the Eclipse bug repository and showed that, unlike the previous approaches, their approach is language independent. Their recall rate using only 40% of the dataset containing only bugs with similar titles was 40.22%.

Jeniffer et al. [1] studied 12 free and open source software projects. They reported that more active repositories do not necessarily contain more duplicates. They concluded that even being consumer-oriented does not mean that the software bug repository will have more duplicates.

Dang et al. [8] proposed a position dependent (PDM) model as a method for calculating call stack similarity. To calculate similarity among stack traces, the position dependent model considers two main metrics including the distance to the top frame and the alignment offset. PDM separates all shared function sequences between two call stacks, then for each it calculates the distance to the top and the alignment offset. For evaluation, they selected five Microsoft products and showed that their approach has slightly worse purity than the Windows Error Reporting (WER), but better inverse purity and f-measure for all of the Microsoft products. Using their approach they reduced the total number of buckets by 25% compared to WER.

Lerch et al. [2] analyzed the description of bug reports in Eclipse Bugzilla. They used a dataset the same dataset used in this study. They were able to detect 19,843 bug reports containing stack traces. To detect duplicate bug reports they applied term frequency and inverse document frequency to functions extracted from stack traces. They calculated the similarity of stack traces by considering the number of similar functions they have. They also studied the importance of the functions in stack traces and showed that the top 20 functions are the most important functions to be considered for detecting duplicate bugs. The challenge of their approach is the exponential increase in the number of distinct functions as many reports are submitted. The large number of functions creates sparse vectors for each and every stack trace corresponding to a bug.

Kim et al. in [16] used crash graphs to detect duplicate crashes and predict fixable crashes. They used the Microsoft Windows Error Reporting (WER) System, which receives crashes from users. They used Microsoft Windows and Microsoft Exchange server crash repositories. In WER, incoming crashes are assigned to buckets. If hit counts of a bucket exceed a certain threshold then the bucket will be reported as a bug and will be assigned to developers to solve the problem. By creating a crash graph for each bucket, they showed that crash graphs can help detect duplicate crash reports that are not detected by the 500 heuristics currently in use, by a rate of 40-60%.

Wang et al. [17] defined a diverse group of crash types that are related to the same or dependent underlying bug as Crash Correlation Group. Crash types are created by collecting crashes having similar stack traces not considering whether they are caused by the same bug or not. They used two popular bug repositories, Firefox Bugzilla and Eclipse Bugzilla. They concluded that using only crash type signature, which is the common top frame function among all stack traces in a crash type, one can identify the Crash Correlation Groups.

Bettenburg et al. [18] showed that stack traces and steps to reproduce the crash are the most important sources of information that if provided by the user when reporting a crash can be used extensively by developer to analyze the underlying faults of the crash reports and to detect duplicate bug reports. This motivated other researchers (e.g., [23]) to use stack traces for bug reproduction.

Sabor et al. [20] showed that stack traces could be used to predict the severity of bug reports. They indexed the functions in stack traces using term frequency and inverse document frequency. They showed that using stack traces, they can predict the severity of a bug report more accurately than when using textual description.

Ebrahmi et al. [21][22] used automata and Hidden Markov Models for the detection of duplicate bug reports in Firefox. The showed that their approach achieve in average 90% accuracy.

VIII. Conclusion

In this paper, we proposed an approach, called DURFEX that, if implemented in bug tracking systems, can facilitate the process of bug handling by automatically detecting duplicate reports. Our approach is based on the concept of trace abstraction where we turn stack traces of function calls to traces of packages (by replacing each function with its containing package). DURFEX keeps track of the temporal order of calls by creating feature vectors from package name traces using varying length N-grams.

When applied to the Eclipse bug repository, our approach achieves better result than an approach that uses function calls. In addition, the execution time of DURFEX can be 93% less than an approach that uses functions only with the 1-gram, up to 70% using 2-gram, which demonstrates the effectiveness of DURFEX.

In future work, we intend to study the role of the distance of packages to the top of the stack trace to see if this heuristic can further improve the detection accuracy. We also intend to conduct more empirical studies with industrial systems.

Acknowledgments

This work is partly funded by MITACS (Mathematics of Information Technology and Complex Systems) and Ericsson.

References

[1] J. Davidson, N. Mohan, and C. Jensen, "Coping with duplicate bug reports in free/open source software projects," Proc. of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2011, pp.101-108.

[2] J. Lerch and M. Mezini, "Finding duplicates of your yet unwritten bug report," Proc. of European Conference on Software Maintenance and Reengineering (CSMR), 2013, pp. 69-78.

[3] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," Proc of International Conference on Software Engineering (ICSE), 2007, pp. 499-510.

[4] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," Proc. International Conference on Dependable Systems and Networks With FTCS and DCC, 2008, pp. 52-61.

[5] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful... really?" Proc of International Conference on Software Maintenance, (ICSM) , 2008, pp. 337-345

[6] A. Schroter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?" Proc of Working Conference on Mining Software Repositories (MSR), 2010, pp. 118-121.

[7] N. Modani, R. Gupta, G. Lohman, T. Syeda-Mahmood, and L. Mignet, "Automatically identifying known software problems," Proc of International Conference on Data Engineering, 2007, pp. 43-441.

[8] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, "Rebucket: A method for clustering duplicate crash reports based on call stack similarity," Proc. of the 34th International Conference on Software Engineering (ICSE), 2012, pp. 1084-1093.

[9] C. D. Natwar, P. Raghavan, and H. Schutze. Introduction to Information Retrieval. New York, NY, USA: Cambridge University Press, 2008.

[10] A. Modani, A. Sureka and P. Jalote, "Detecting duplicate bug report using character n-gram-based features," Proc. of the 2010 Asia Pacific Software Engineering Conference, 2010, pp. 366-374.

[11] Rajeev G. Qian, S. Sural, Y. Gu, and S. Pramanik, "Similarity between euclidean and cosine angle distance for nearest neighbor queries," Proc. of the 2004 ACM Symposium on Applied Computing, 2004, pp. 1232-1237.

[12] M. Brodie, S. Ma, G. Lohman, L. Mignet, M. Wilding, J. Champlin, and P. Sohn, "Quickly finding known software problems via automated symptom matching," Proc of International Conference on Autonomic Computing, 2005, pp. 101-110.

[13] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Extracting structural information from bug reports," Proc. of the International Working Conference on Mining Software Repositories (MSR), 2008, pp. 27-30.

[14] K. Bartz, J. W. Stokes, J. C. Platt, R. Kivett, D. Grant, S. Calinoiu,and G. Loihle, "Finding similar failures using callstack similarity," Proc. of the 3rd Conference on Tackling Computer Systems Problems with Machine Learning Techniques, 2008.

[15] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," Proc. of the 30th International Conference on Software Engineering (ICSE), 2008, pp. 461-470.

[16] S. Kim, T. Zimmermann, and N. Nagappan, "Crash graphs: An aggregated view of multiple crashes to improve crash triage," Proc. of the IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN), 2011, pp. 486-493.

[17] S. Wang, F. Khomh, and Y. Zou, "Improving bug localization using correlations in crash reports," Proc of Working Conference on Mining Software Repositories (MSR), 2013, pp. 247-256.

[18] G. N. Bettenburg, S. Just, A. Schroter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2008, pp. 308-318.

[19] C. Sun, D. Lo, S. C. Khoo and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," Proc. of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2011, pp. 253-262.

[20] K .K .Sabor, M. Hamdaqa, and A. Hamou-Lhadj, “Automatic prediction of the severity of bugs using stack traces,” Proc. of the 26th Annual International Conference on Computer Science and Software Engineering (CASCON '16), 2016, pp. 96-105.

[21] N. Ebrahmi., Md. Shariful Islam, A. Hamou-Lhadj, and M. Hamdaqa, “An effective method for detecting duplicate crash reports using crash traces and hidden Markov models,” Proc. of the 26th Annual International Conference on Computer Science and Software Engineering (CASCON), 2016, pp. 75-84.

[22] N. Ebrahimi*, A. Hamou-Lhadj, "CrashAutomata: An Approach for the Detection of Duplicate Crash Reports Based on Generalizable Automata," Proc. of the 25th Annual International Conference on Computer Science and Software Engineering (CASCON), 2015, pp. 201-210.

[23] M. Nayrolles, A. Hamou-Lhadj, S. Tahar and A. Larsson, "JCHARMING: A bug reproduction approach using crash traces and directed model checking," Proc. of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2015, pp. 101-110.