2012 IEEE 14th International Symposium on High-Assurance Systems Engineering

Automated Duplicate Bug Report Classification using Subsequence Matching

Sean Banerjee, Bojan Cukic, Donald Adjeroh
Lane Department of Computer Science and Electrical Engineering
West Virginia University
Morgantown, WV, USA
{sean.banerjee, bojan.cukic, donald.adjeroh}@mail.wvu.edu

Abstract— **The use of open bug tracking repositories like Bugzilla is common in many software applications. They allow developers, testers and users the ability to report problems associated with the system and track resolution status. Open and democratic reporting tools, however, face one major challenge: users can, and often do, submit reports describing the same problem. Research in duplicate report detection has primarily focused on word frequency based similarity measures paying little regard to the context or structure of the reporting language. Thus, in large repositories, reports describing different issues may be marked as duplicates due to the frequent use of common words. In this paper, we present FactorLCS, a methodology which utilizes common sequence matching for duplicate report detection. We demonstrate the approach by analyzing the complete Firefox bug repository up until March 2012 as well as a smaller subset of Eclipse dataset from January 1, 2008 to December 31, 2008. We achieve a duplicate recall rate above 70% with Firefox, which exceeds the results reported on smaller subsets of the same repository.**

Keywords- String Algorithms, Duplicate Bug Reports, Documentation, Experimentation, Verification.

I. INTRODUCTION

Bug tracking systems play a vital role in software development and management. These systems allow users, developers and testers to report software problems. Problem reports are categorized as primary when it describes an unseen issue, or duplicate when it describes an existing problem. The proliferation of duplicates is due to a variety of reasons - poor search features, lack of duplicate detection [32], user impatience, and network connectivity issues being some of the major ones. The end result is an added workload on the software maintenance engineers. They have to ensure that each report is assigned a correct status and, if duplicate, associated with the appropriate group of similar reports.

Much of the work in duplicate detection has leveraged word frequency based similarity measures. Such methods weight the occurrences of each individual word in the report and compare their frequency across all documents. These measures assign a lower weight to frequent words and reduce the possibility of false matches. Several classical techniques exist for weighting word frequencies, including Term Frequency/Inverse Document Frequency (TF/IDF) and Group Centroids. Normalization of the language space is achieved through tokenization, stemming and stop word removal. The fundamental deficiency of word frequency based similarity measures is the lack of language context. A sentence in English, or any other language, is not a random clustering of words. Instead, it follows language structure rules with noun and verb phrases. While two documents may exhibit similar word frequencies, the issue described may not be similar and hence have a different context. As the size of the repository increases, so does the likelihood that two completely unrelated documents may be flagged as similar or duplicates.

The bioinformatics domain has been driving significant advances in string matching algorithms [33, 37], with successful applications of Rabin-Karp, Boyer-Moore, Needleman-Wunsch, Shift operations, and others. Ordered sequences play a critical role in bioinformatics, for example, the order of genes in sequences determines commonality [36]. In this paper, we develop *FactorLCS*, a new methodology to detect duplicate reports. We hypothesize that two reports that have the longest ordered sequence of common words are more likely to be duplicates than two documents that have the same frequency of unordered words. To reduce the number of false matches caused by the length of problem reports, we add in a factor - Match Size Within Group Weight (MSWGW) that determines impact of the matching document and longest matches within a group. Furthermore, the impact of MSWGW is weighted with the Product, Component and Classification of the test and matching report to enable more effective classification.

Our major contributions to the domain of bug report duplicate detection include:

- The development of a novel methodology *FactorLCS,* which accounts for the context of problem reports by analyzing and preserving word order, ignored by traditional information retrieval methods;
- We evaluate the performance of this algorithm on the entire Firefox repository (up until March 2012) and a subset of the Eclipse repository (January 1, 2008 to December 31, 2008), including duplicate reports associated with imported components, typically excluded in the related literature.

To fully appreciate problem size and its importance, we note that close to *100* reports are submitted to large bug repositories, similar to Firefox, every day [12]. As fresh reports are submitted, the triage team must devote significant resources to determine whether a submitted report reflects a yet unknown problem or is a duplicate of an existing report.

The remainder of the paper is organized as follows: Section 2 surveys the related work. Section 3 lists the drawbacks of word frequency as text similarity measures. In

1530-2059/12 $26.00 © 2012 IEEE
DOI 10.1109/HASE.2012.38

Section 4 we describe the Firefox repository, while Sections 5 and 6 explain the LCS algorithm. In the reminder of the paper we evaluate performance and statistical significance and draw appropriate conclusions.

II. RELATED WORK

Detecting duplicate bug report involves comparing newly submitted reports to an existing repository [3, 4, 5, 6]. One of the best classification results was reported by Wang et al. [3]. They leverage natural language processing techniques along with execution information to build a suggested list of known reports most similar to the new report. During submission, the reporter is presented with the list and can determine whether the report at hand matches any of the reports in the list. The authors achieve match rates between *67%* and *93%*. However, the paper uses a small subset of Firefox data, *77* duplicate bug reports submitted from Jan 1, 2004 to April 1, 2004, describing the release of Firefox 0.8. Uniform and small data sets typically boost performance. When applying this approach to a larger dataset the authors achieved a recall rate of approximately *53-69%* [19]. This data set included *3,207* duplicates, from April 4, 2002 to July 7, 2007. Similar recall rates have been achieved by others [18], but these experiments do not use problem reporting datasets of the complexity similar to Eclipse or Firefox. In [4] a text analysis approach achieved a *50%* recall rate on the Firefox repository. Bayesian learning and text categorization yielded a *30%* prediction rate in [5, 6]. In [34], BM25F textual similarity function and a gradient descent algorithm obtain a *67-68%* recall rate with Mozilla data from January 1, 2010 to December 31, 2010. Our previous work [22] utilizes group centroids as a weighting scheme leading to a *53%* recall rate when utilizing all Firefox reports submitted before June 2010.

Prior work on duplicate report classification puts emphasis on word frequency measures using samples of complete repository. String matching techniques have not been used. We notice, however, the application of string matching techniques for the detection of common usage patterns in execution logs [26]. Other related problems tackled through inexact string matching include the detection of duplicate records in databases [27, 16], name matching [10] and approximate duplicate detection in documents created by optical character recognition [11]. We note that the language found in these applications is far simpler than the free-form English in software problem reporting.

III. MOTIVATING EXAMPLE

Word frequency measures discussed in related work use cosine similarity to perform matching. Cosine similarity computes the dot product between the two vectors as follows:

$$\cos(\Theta) = \frac{v_1 \bullet v_2}{\|v_1\| \times \|v_2\|}$$

The vectors indicate specific reports, v_1 and v_2 from the corpus of Firefox reports. Each vector consists of distinct words found in each report and their associated frequencies. We use two actual Firefox problem reports, denoted *175655* and *171777*, as part of the motivating example. After pre-processing (discussed later) they look as follows:

171777: **bookmark separ** *misplac sidebar step reproduc 1 copi bookmarkshtml mozilla 12a applic data* **folder** *phoenix applic data folder 2 separ* **appear** *correctli pull-down menu manag bookmark window 3 view -> sidebar -> bookmark 4* **separ** *ar all top* **bookmark** *pane between item separ still appear* **correctli** *pull-down menu* **manag** *bookmark* **window** *---- addit test 5 open manag* **bookmark** *move separ 6 separ disappear* **bookmark** *pane correctli* **displai** *pull-down* **menu** *manag* **bookmark** *window 7 close* **bookmark** *view ->* **sidebar** *-> bookmark 8* **separ appear** *top pane shoulg three on disappear two remain ---- 9 add* **separ** *middl list 10 new separ appear* **bottom bookmark** *pane correctli* **displai** *pull-down menu manag* **bookmark** *window*

175655: drag url into **bookmark** *subfold drop just abov* **separ** *open tab caus new bookmark displai separ also occur if drop url onto name* **folder** *both case url* **appear** *new* **separ** *bottom list just abov separ open tab cannot select* **bookmark** *doe displai* **correctli** *bookmark* **manag window** *if* **bookmark** *wa creat by drag onto* **bookmark** *toolbar will* **displai** *bookmark* **menu bookmark** *sidebar similarli* **bookmark** *creat by drag into bookmark menu will displai correctli* **sidebar** *toolbar drag more url caus more* **separ appear** *these bookmark displai correctli after phoenix restart reproduc alwai step reproduc 1drag bookmark symbol address bar folder either within bookmark menu toolbar 2releas either name folder just abov* **separ bottom** *expand folder 3expand folder after* **bookmark** *been creat* **displai** *separ actual result new* **bookmark** *wa displai non-select separ expect result displai bookmark.*

Applying cosine similarity, the score is *0.68*. In problem report systems, this score typically implies that these reports are each other's duplicates. However, while they discuss issues pertinent to bookmarks, the actual problems are different. In the Firefox duplicate repository they are marked as distinct primary reports. The cosine similarity measure ignores one critical factor in documents written in natural language – the order of words. Therefore, a system that leverages not only the frequency of similar words but also the word ordering may create a more effective duplicate detection system.

Suppose, instead of taking the frequency of words we choose to find word positions in *175655* where it matches report 171777. We would obtain matches to report *171777* in the following word positions of *175655* (using a 0 starting number scheme): [*3, 8, 23, 27, 29, 39, 42, 44, 45, 47, 53, 56, 58, 59, 62, 72, 79, 80, 109, 110, 116, 119, 124*], or a chain of *23* matching words. (We highlight the matches in bold in the examples above.) Dividing this match score by the size of the matched document *171777* yields a similarity measure of *0.20.* These two reports do not appear likely to be duplicates.

The shortcomings of cosine similarity are not isolated to the ordering of words. The technique cannot account for spelling errors, especially when they affect many of the words in the string. But we learned that the low quality of English writing in many of the Firefox problem reports presents. Typographical errors also negatively affect TF/IDF and Group Centroids, which assign a lower weight to common words and a higher weight to uncommon words [13]. Words containing typos will be assigned higher weights as they are less frequent within the repository. Hence, documents containing common typos are more likely to match.

IV. DATASETS

The problem reporting data sets for our study come from Firefox [2] and Eclipse [1]. Both utilize the Bugzilla problem reporting system and have been often used in duplicate detection studies. Table 1 summarizes the contents of Firefox repository as of March 2012. Firefox shares common code with other Mozilla projects. The traditional duplicate detection approaches exclude from analysis all duplicates whose primary reports reside outside of Firefox. We do not inherit such an experimental design because it reduces the number of duplicates in the data set by almost *50%*. In our approach we consider only the 97,865 reports that constitute the Firefox repository until March 2012.

Of the 30,748 duplicate reports, a total of 3,565 reports had a singular duplicate residing within Firefox and the associated primary report existed outside the Firefox repository. It is impossible to detect these reports as we do not have a matching report within Firefox. Thus, a total of 3,565 duplicate reports within Firefox had to be marked as primary to enable proper duplicate detection without the need to bring in external primaries and their duplicates. Furthermore, a total of 2,138 reports were the first duplicate report in a duplicate group whose primary existed outside Firefox. These reports would also be impossible to detect. As a result, the total number of detectable duplicates within our repository was 25,045 reports. Finally, only 16,630 reports had a primary report found within the Firefox repository.

Table 1 - Firefox Summary (up to March 2012)

Total # Duplicates	30,748
Actual # of Detectable Duplicates	25,045
Total # Duplicates with Firefox Primary	16,630
Total # of Primaries Within Firefox Dataset	67,117
Total Reports	97,865

In order to ensure the approach described in this paper is not biased towards a particular dataset, a subset of the Eclipse dataset was also used for validation. We utilize one year of the Eclipse dataset from January 1, 2008 to December 31, 2008. We summarize this dataset in Table 2. The difference between the total number of duplicates and detectable duplicates is due to the existence of primary reports outside the window of observation.

Table 2 - Eclipse Summary (January 1, 2008 to December 31, 2008)

Total # Duplicates	4,059
Actual # of Detectable Duplicates	2,836
Total # of Duplicate Groups	2,054
Total # of Primaries	37,140
Total Reports	41,199

Occasionally, a report may be assigned as primary while it is, in fact, a duplicate of an existing report. For example, in Firefox bug 729936 is marked as a duplicate of bug 729932. But, bug 729932 is marked as a duplicate of 729935. Bug 729935 is a duplicate of bug 676001. Therefore, bug 729932, 729935 and 729936 should all be marked as duplicates of 676001. Such issues arise when the triage team incorrectly assigns a primary status, thereby creating a split in the repository. In such cases it will be necessary to merge the two groups to create one unified duplicate group. Through the course of this paper we identify such groups as "merged".

V. LONGEST COMMON SUBSEQUENCE

The longest common subsequence (LCS) algorithm operates on a set of strings [23]. A subsequence is a sequence that can be derived by deleting some elements of the sequence without changing the order of the remaining elements. LCS has been applied to several domains. For example, in molecular biology DNA sequences can be represented as combinations of four basic blocks – A, C, G and T. As new gene sequences are found, biologists are interested in determining what subsequences are common with some known genes. While LCS has generally been applied at a character level, it can be easily modified to work at the word level.

The longest common subsequence between report *171777* and *175655* includes of following words: {[bookmark], [separ], [folder], [appear], [separ], [bookmark], [correctli], [manag], [window], [bookmark], [bookmark], [displai], [menu], [bookmark], [bookmark], [sidebar], [separ], [appear], [separ], [bottom], [bookmark], [displai], [bookmark]} at a total length of 23 words. While the tokens in the LCS are important, we are more interested in finding the actual length of the LCS. A simple method for determining the length is described in Algorithm 1. In our experiments, we will use LCS function built in Perl within Algorithm::Diff library [24]. Our focus is not to improve the algorithm, but to leverage its characteristics to detect duplicate bug reports.

VI. DUPLICATE DETECTION METHODOLOGY

Prior to the application of any duplicate classification method the text found in the Title and Summary fields of each problem report needs to be brought into a form that enhances the chance to correctly recognize duplicates.

Algorithm 1: Length of Longest Common Subsequence

X[1..m] is a string and Y[1..n] is a string
C[m,n] is the length of the LCS of X and Y

```
function LengthLCS(X[1..m], Y[1..n])
  C = array(0..m, 0..n)
  for i := 0..m
        C[i,0] = 0
  for j := 0..n
        C[0,j] = 0
  for i := 1..m
        for j := 1..n
              if X[i] = Y[j]
                    C[i,j] := C[i-1,j-1] + 1
              else
                    C[i,j] := max(C[i,j-1], C[i-1,j])
  return C[m,n]
```

A. *Text Preprocessing*

Firefox bug reports are written in English. In addition to words, reports will contain characters not relevant to a software problem. The objective of preprocessing is to convert the document to a format that contains only relevant textual information in each report. Each report is tokenized to remove punctuation marks, and all upper case characters converted to lowercase. Next, we stem the reports using the classic Porter's Stemming algorithm [21]. Finally, all common words that do not add to the semantics are removed by using a stop word list [8]. However, the list in [8] is too broad for our chosen application. Moreover, given that negation terms and application specific terms are relevant to document semantics we chose to utilize a smaller stop word list containing *114* terms [15]. These basic steps are shared throughout the literature.

B. *FactorLCS Methodology*

We consider Title, Summary, Classification, Component and Product fields of Firefox bug report. While the Title has shown to be sufficient in word frequency based measures, it simply does not provide enough depth to be useful in generating common matches with LCS. Each report in the repository first goes through tokenization, stemming and stop word removal. Each report is compared to the ones submitted prior to it. LCS score is computed for each report pair.

Taking the highest LCS score as the indication of a match does not work well. Any short document is likely to have a higher LCS match scores with longer reports. Consequently, we weight the LCS score using Match Size Within Group Weight (MSWGW).

The MSWGW is defined as the sum of the ratios of the LCS value over the sizes of matching documents. The MSWGW is calculated as follows:

$$MSWGW = \frac{\sum_{n}\left(\frac{C[m,n]^2}{m}\right)}{N} * \sum \frac{C[m,n]}{N}$$

where N is the number of reports within the group, $C[m,n]$ is the number of words matched between report X of size m and report Y of size n, and m is the size of the matching document Y. For example, suppose Report A matches to reports B1, C1, D1 and E1 contained in duplicate groups X1, Y1, X1 and Z1, respectively, as shown in Table 3 below.

Table 3 - Classifying Report A

Report	Grp	Matches	Size of Report
B1	X1	20	100
C1	Y1	15	20
D1	X1	10	25
E1	Z1	10	20

We can compute the MSWGW values as follows:

$MSWGW(X1) = (20^2/100+10^2/25)/2 * (20+10)/2 = 60$
$MSWGW(Y1) = 15^2/20 * 15/1 = 168.75$
$MSWGW(Z1) = 10^2/20 * 10/1 = 50$

Without MSWGW we may have considered report A to be a part of group X1. With MSWGW we classify it as a part of group Y1. MSWGW factor allows us to reduce false matches to large groups and large reports.

Additionally, we determined that the Classification (Class), Component (Comp) and Product (Prod) fields played a critical role in determining which primary report a duplicate was associated with [35]. Reports describing similar problems had atleast one of the three fields similar. Thus, the likelihood of a match is can be increased by boosting the *MSWGW* score when one or more of the fields match. We exemplify this phenomenon in Table 4 below.

Rep	Grp	Comp (Rep)	Comp (Grp)	Class (Rep)	Class (Grp)	Prod (Rep)	Prod (Grp)
A	X1	1	2	C1	C1	P1	P2
A	Y1	1	1	C1	C2	P1	P1
A	Z1	1	3	C1	C3	P1	P3

Table 4 - Impact of Classification, Component and Product

From Table 4 we note that Group X1 has only one of the three fields (Comp, Class and Prod) similar to the test report. Group Y1 has two fields in common, while Group Z1 has only one field in common. Using the data in Table 4 we can boost the *MSWGW* scores when the Comp, Class and Prod values match by using the following equation:

FinalMSWGW =W1 MSWGW[if Comp(Rep) = Comp(Grp)] + W2 * MSWGW[if Class(Rep) = Class(Grp)] + W3 * MSWGW[if Prod(Rep) = Prod(Grp)] + MSWGW[if Comp(Rep) != Comp(Grp)] + MSWGW[if Class(Rep) != Class(Grp)] + MSWGW[if Prod(Rep) != Prod(Grp)]*

Suppose we assign weights to *W1, W2,* and *W3* of 2, 3 and 4 respectively. The *FinalMSWGW* values would be as follows:

FinalMSWGW(X1) = 300
FinalMSWGW(Y1) = 1181.25
FinalMSWGW(Z1) = 150

The *LikelyGroup* would be associated to group Y1 as it had the highest *FinalMSWGW* value. To generate the top 20 suggested list we simply take the highest 20 *FinalMSWGW* values and the groups they are associated with.

The *W1, W2* and *W3* weights associated with *FinalMSWGW* can be found using several techniques. We apply a pseudo Simulated Annealing (SA) method to optimize the weights [34]. The base case is set to the *MSWGW* results. Unlike a traditional SA where a model exists, we optimize until the best recall rate is achieved. Given the likelihood that an SA may not find an optimal solution, processing is terminated after 10,000 iterations. Testing with higher number of iterations did not yield significantly better results.

For the scope of this paper we optimize to the first year of Firefox data, for validation we tested on the Eclipse data with the same weights to determine whether the weights can be generalized. We obtain weight scores as follows (rounded to one decimal): **W1=17.2**, **W2=20.0**, and **W3=17.8**. The higher weight assigned to Classification is expected as it is a top level categorization within Bugzilla while Product and Component have nearly equal weights as they are second level categorizations [35]. However, the weights associated are likely to be application and search space dependent. The impact of the application and search space will be investigated as part of our future work.

Generating the *MSWGW* scores takes approximately 1.5 seconds on an AMD 3GHz machine with 1.75GB RAM. Computing the *FinalMSWGW* score, sorting and generating the *LikelyGroup* list takes approximately 0.5 seconds. Thus, a triager will be given a suggested top 20 list for each new incoming report in under 2 seconds. The size of the document also plays a critical role in the time taken to generate the final results. The median size of all reports found within the Firefox repository was 68 words, with 95% of all documents having less than 200 words.

C. *Evaluation Approach*

The Firefox repository offers the ground truth for both primary and duplicate reports. To evaluate the accuracy of *FactorLCS*, we will observe its ability to correctly flag the new bug report as a duplicate of the proper existing group. We measure the performance using a standard recall metric that computes the ratio of the number of duplicate bug reports correctly classified over the total number of duplicate reports in the repository. In all experiments we use Firefox problem reports dataset downloaded in March 2012. This dataset contains merged report groups, thus differing slightly from earlier snapshots.

In [22] we introduced the concept of a Time Window for bug report duplicate detection. We note from Figure 1 that in 95% of the cases a newly submitted report was likely to have a duplicate match within the *2,000* most recently updated duplicate groups. The time window approach improves the match time by reducing the search space to a manageable level. Nevertheless, we will report recall rates of FactorLCS obtained with and without the time window.

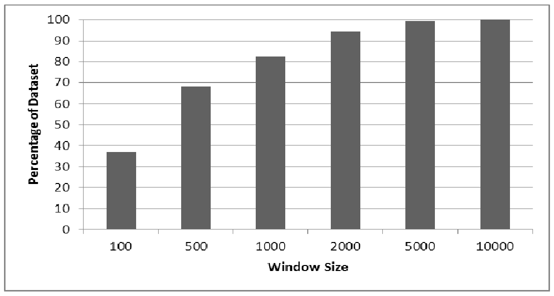

Figure 1 - Window Size Comparison (Firefox)

Our system does not use supervised learning. Therefore, in principle, it can be deployed to any emerging problem report repository, regardless of its initial size. The system would evaluate incoming reports against all prior reports until *2,000* distinct duplicate bug report groups emerge. That completes the size of the time window. At that time, the time window takes effect and finds matches within the *2,000* most recently updated groups.

VII. DUPLICATE DETECTION RESULTS

In line with previous work, *FactorLCS* returns a list of *20* reports / groups most likely to be duplicates of the report the user is trying to submit.

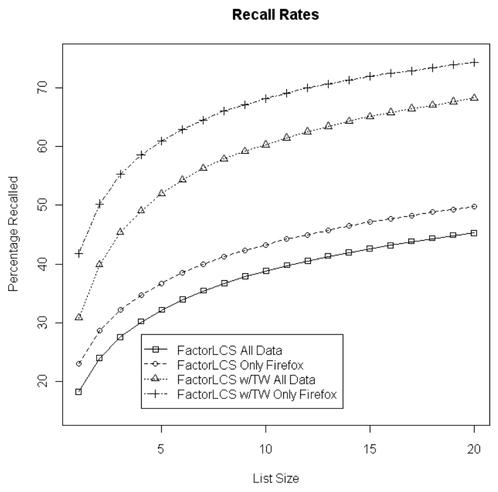

Figure 2 - Recall Rates by Experiments (Firefox)

Figure 2 summarizes the results of the application of *FactorLCS* on the March 2012 merged version of Firefox repository. As mentioned before, we applied the algorithm to the repository version that includes only Firefox problem reports and the version with the duplicates associated with the core reports too. We also ran duplicate detection experiments with and without the time window (marked TW in Figure 2).

To ensure that our procedure was not biased towards a particular dataset we repeated the same experiment with the smaller Eclipse bug repository from January 2008 to

December 2008. This dataset contains *2,836* total duplicate reports. We present the results of the partial Eclipse repository in Figure 3 and demonstrate the effects of each factor individually and when combined to create FactorLCS.

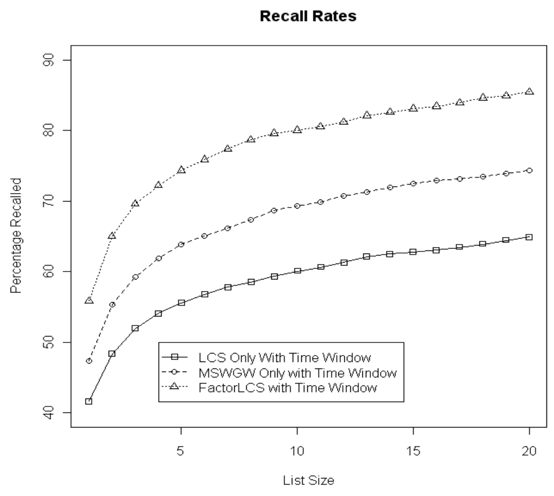

Figure 3 - Recall Rates (Eclipse)

Table 5 - Comparison of Results

Approach	**Results**	**Dataset**
Text Analysis [4]	~50% recall rate	Firefox (<2006)
Bayesian learning, text categorization [5]	Predicted 30% of duplicates	Eclipse
Text similarity clustering [6]	Recall 51% at list size 20	Firefox Feb 05 to Oct 05
NLP with execution information [3]	67-93% (43-72% with NLP)	Firefox Jan 04 to April 04
NLP with execution information [19]	~53% detection rate	Firefox Apr 02 to Jul 07
Discriminative model trained via SVM [19]	~70% recall rate at list size 20	Firefox Apr 02 to Jul 07
BM25F algorithm [20]	~68% recall rate at list size 20	Mozilla Jan 10 to Dec 10
BM25F algorithm [20]	~75% recall rate at list size 20	Eclipse Jan 08 to Dec 08
Time window with centroids [22]	~53% recall rate at list size 20	All Firefox up to June 2010
Time window with FactorLCS [This Paper]	~68% recall rate at list size 20	All Firefox up to March 2012
Time window with FactorLCS [This Paper]	~73% recall rate at list size 20	All Firefox up to March 2012
Time window with FactorLCS [This Paper]	~85% recall rate at list size 20	Eclipse Jan 08 to Dec 08

We offer Table 5 as a means to compare the performance of our approach to existing ones from the literature. From Table 3 we note that the recall rates obtained through *FactorLCS* match or outperform existing research results. Generally, *FactorLCS* with the time window (TW) typically offers a 5% performance improvement over related techniques. Performance increase is obtained in spite of the experimentation with the most recent version of an ever expanding Firefox problem report data set. The results of duplicate detection drop by ~6% when applied to which includes primaries outside of Firefox, but related literature did not tackle such data set extension either. We speculate a margin of improvement between *FactorLCS* and competing algorithms would remain the same.

VIII. STATISTICAL EVALUATION

In the experiments presented above, we relaxed some of the common simplifying assumptions. Therefore, it is important to understand whether the increase in performance is the result of changed assumptions or the improved duplicate detection algorithm.

A. Statistical Significance of Group Merging

One major artifact of organic repositories is the creation of branches in the life cycle of a bug report. A branch is created when a bug report is assigned a primary status while it is in fact the duplicate of another report or a group. To mitigate the problem one would have to merge the two groups and assign a single primary report. To determine whether the merging of some of the groups that occurred prior to March 2012 impacted the reported recall rate, we repeat the *FactorLCS* with Time Window experiment using the unmerged Firefox and Firefox plus Core data sets. We noted a *0.8%* to *1.2%* improvement in overall recall rates when using the merged dataset compared to the unmerged dataset. In order to test significance of this performance effect, we created the following hypotheses:

- ***H_0:*** *Merged repository recall scores are no different than unmerged recall scores*
- ***H_{alt}:*** *Merged repository recall scores are greater than unmerged recall scores*

To determine statistical significance of our results we employ Wilcoxon's Signed Rank Test. The Wilcoxon Signed Rank Test is a non-parametric test that ranks the absolute value of each difference between the merged and unmerged recall rates. It then reassigns the positive and negative sign and tests to determine whether the average rank is different from 0. Mathematically, the Wilcoxon Signed Rank Test z-value is defined as follows:

$$z = \frac{\bar{r} - 0}{\sqrt{(N+1)(2N+1)/6N}},$$

where $\bar{r}$ is the average rank and N is the total number of observations.

We can reject the NULL hypothesis at the 5% confidence interval (z=3.91) and infer that the recall rate for merged duplicate groups is higher than unmerged groups.

Obviously, the results of duplicate bug report classification are impacted by the consistency of the ground truth. Correct assignment of status to a bug report has a statistically significant effect on the recall rate. One may argue that we observed a minimal (yet statistically significant) performance improvement. Nevertheless, the process that leads to merging of report groups is difficult and, often, laborious. A large number of such changes may create even a better environment for a fully automated bug report triage.

B. *Impact of Excluding Non Firefox Primaries*

One major limitation in matching duplicate bug reports stems from the fact that not all existing problem reports may be available in the repository being searched. A primary report may be associated with another application which shares components with the current one. In Firefox example an issue in Core could propagate a problem which is now reported through Firefox.

Unless all primary reports from the Core development are included, it is not possible to match new reports with the primaries in the Core. Simply marking these duplicates from the Core as novel in Firefox [3] is not realistic either. We determined that *16,630* reports out of *30,748* had primaries within Firefox development. Results in Figure 1 confirm that opening the search to "all data", rather than Firefox – only repository impacts the performance of duplicate detection. Therefore, we formulated the following statistical test hypotheses:

- ***H_0*: *Recall scores from "All data" experiments are no different than those from Firefox only.***
- ***H_{alt}*: *Recall scores from "All data" experiments are lower than those from Firefox only.***

We were able to strongly reject the NULL hypothesis at a 5% confidence interval (z=3.91) and infer that there is a statistically significant improvement in recall rates when restricting the experiment exclusively to reports in Firefox repository. This behavior is expected as we may find more matches outside of the Firefox repository. But including all such reports makes the duplicate bug detection much more difficult, given the volume of reports.

C. *Threats to Validity*

The representativeness of the datasets used in experiments is an external validity threat. We believe Firefox and Eclipse are large enough and complicated enough to affirm relevance of the study. Similar results are expected from other projects where duplicate reports are submitted in natural language. Nevertheless, open source projects and open problem reporting policies allow anyone, regardless of their experience and understanding, to submit a report. Such reporting, including the absence of technical information and in many cases very poor use of language, imposes a burden on maintenance personnel, compared to proprietary projects. From this perspective, external threat to validity is significant.

Within the context of Firefox repository, several duplicate groups had been merged, status of several reports has been changed over time. Some primary bug reports were found to be duplicates. Changing ground truth makes it difficult to perform comparative analysis of our results with existing research. However, evolution of software artifacts is a known dynamic. Freezing the snapshot of a repository just to be able to compare performance of research algorithms does not seem to be the desirable approach either, except in cases where research community may organize benchmarks.

The next validity threat is related to the adequacy of the proposed technical approach. The subsequence based matching approach (LCS) relies on the presence of large bodies of text. When we used only the Title of reports to measure similarity, we failed to yield measurably improvements over existing methods. Titles are simply too short. Word frequency approaches are typically limited to the analysis and comparison of problem report titles only. We did not pursue detailed measurements of processing time, but having implemented both types of algorithms, it does not appear that LCS approach is prohibitively slower.

Lastly, the weights associated with *LikelyGroup* are likely to be application and search space dependent. For the scope of this paper the weights are generalized across both applications using only the first year of Firefox data for optimization purposes. We are currently investigating the impact of the application as well as the search space required to determine the weights.

IX. SUMMARY AND FUTURE WORK

Duplicate detection in bug tracking systems is not a new problem. Bugzilla itself employs Boolean full text search engine. Like many proposed research approaches, this tool is based on word frequency counts, which fail to account for the context of the report. In [22] we discussed the issues associated with the Bugzilla search tool wherein several reports did not yield a match even when searching with the exact phrase of an existing report.

This paper represents the first effort at applying a string matching approach to duplicate bug report detection. Our results exceed the recall rates of currently reported research approaches by at least 5% while using more recent release of Firefox bug report repository. Repository size has been shown to impact detection in [19, 20] where the authors demonstrated a loss in performance when using the larger repository. We show that longest common subsequences, and possibly other bioinformatics algorithms can be used to effectively detect duplicate reports. We also demonstrate the significance of proper primary bug report identification by demonstrating the long term impact of merging report branches, once they are recognized to represent the same underlying problem.

Natural language plays a critical role in the ability for systems to detect duplicates. Duplicate reports are submitted by users writing in British and American English. Typographical and alternate spellings are prevalent in duplicate reports. This raises an interesting area of future work. The breadth of the language in software problem reporting is a small subset of English. The prevalent terms in bug reports are very specific to the application under study. We believe that the development of application specific dictionaries may offer the new types of analyses that may improve report matching process [17]. Such an approach would further normalize individual reports and allow increased effectiveness of duplicate report detection and classification algorithms.

REFERENCES

[1] Eclipse – http://www.eclipse.org

[2] Firefox - http://www.mozilla.org/en-US/firefox/fx/

[3] 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 '08: Proc. of the 30th Int'l conference on Software engineering, pages 461–470, New York, NY, USA, 2008.

[4] L. Hiew. Assisted detection of duplicate bug reports. Master's thesis, The University of British Columbia, Vancouver, Canada, May, 2006.

[5] D. Cubranic. Automatic bug triage using text categorization. In In SEKE 2004: Proc. 16th Int'l Conf. on Software Engineering and Knowledge Engineering, pp. 92–97, 2004.

[6] N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In Dependable Systems and Networks With FTCS and DCC, IEEE International Conference on, pp. 52 –61, 2008.

[7] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In eclipse '05: Proc. of the 2005 OOPSLA workshop on Eclipse technology eXchange, pages 35–39, New York, NY, USA, 2005. ACM.

[8] Christopher Fox. 1989. A stop list for general text. *SIGIR Forum* 24, 1-2 (September 1989), 19-21.

[9] M. Bilenko and R. J. Mooney. 2003. Adaptive duplicate detection using learnable string similarity measures. In Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '03). ACM, New York, NY, USA, 39-48.

[10] W. Cohen, P. Ravikumar, and S. Fienberg. A comparison of string metrics for matching names and records. In Proc. Workshop on Data Cleaning and Object Consolidation at the Int'l Conf. on Knowledge Discovery and Data Mining (KDD), 2003.

[11] Lopresti, D.P. Models and algorithms for duplicate document detection. In Document Analysis and Recognition, 1999. ICDAR '99. Proceedings of the Fifth International Conference on , vol., no., pp.297-300, 20-22 Sep 1999.

[12] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Duplicate bug reports considered harmful ... really? In ICSM, pages 337–345, 2008..

[13] R. Feldman and J. Sanger. The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data. Cambridge University Press, 2006.

[14] T. Zimmermann, R. Premraj, J. Sillito, and S. Breu. Improving bug tracking systems. In Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on, pages 247 –250, May 2009.

[15] Stop Word List: http://assets.slate.wvu.edu/resources/167/1343313055.txt.

[16] Monge, A. E.: Matching Algorithm within a Duplicate Detection System. IEEE Techn. Bulletin Data Engineering 23(4), 2000

[17] S. Banerjee, J. Musgrove, and B. Cukic. Handling Language Variations in Duplicate Bug Detection Systems (under review)

[18] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In ICSE 07: Proceedings of the 29th international conference on Software Engineering, pages 499–510, Washington, DC, USA, 2007. IEEE Computer Society.

[19] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, pages 45–54, New York, NY, USA, 2010. ACM.

[20] C. Sun, D. Lo, S-C. Khoo, and J. Jiang. 2011. Towards more accurate retrieval of duplicate bug reports. In *Proc. of the 2011 26th IEEE/ACM Int'l Conference on Automated Software Engineering* (ASE '11). IEEE Computer Society, Washington, DC, USA, 253-262.

[21] Porter, M., "The Porter Stemming Algorithm," http://www.tartarus.org/-martin/PorterStemmer

[22] T. Prifti, S. Banerjee, and B. Cukic. Detecting bug duplicate reports through local references. In Proc. of the 7th International Conference on Predictive Models in Software Engineering (Promise 2011)

[23] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. Commun. ACM 18, 6 (June 1975)

[24] Perl Algorithm::Diff - http://search.cpan.org/dist/Algorithm-Diff/lib/Algorithm/Diff.pm

[25] S. Kim, T. Zimmerman, N. Nagappan. Crash Graphs: An Aggregated View of Multiple Crashes to Improve Crash Triage. In DSN '11: Proceeding of the 41st International Conference on Dependable Systems & Networks, pages 486-493, Hong Kong, China, 2011

[26] M. Nagappan, K. Wu, and M. A. Vouk. 2009. Efficiently extracting operational profiles from execution logs using suffix arrays. In *Proc. 20th IEEE Int'l Con. on Software Reliability Engineering* (ISSRE'09), Andrew Podgurski and Pankaj Jalote (Eds.). IEEE Press, Piscataway, NJ, USA, 41-50, 2009.

[27] A.K. Elmagarmid, P.G. Ipeirotis, V.S. Verykios. Duplicate Record Detection: A Survey. In *Knowledge and Data Engineering, IEEE Transactions on* , vol.19, no.1, pp.1-16, Jan. 2007

[28] M. Bilenko, R.J. Mooney, W.W. Cohen, P. Ravikumar, and S.E. Fienberg, Adaptive Name Matching in Information Integration. IEEE Intelligent Systems, vol. 18, no. 5, pp. 16-23, Sept./Oct. 2003.

[29] V.I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. Doklady Akademii Nauk SSSR, vol. 163, no. 4, pp. 845-848, 1965, original in Russian—translation in Soviet Physics Doklady, vol. 10, no. 8, pp. 707-710, 1966.

[30] M.A. Jaro. Unimatch: A Record Linkage System: User's Manual. technical report, US Bureau of the Census, Washington, D.C., 1976.

[31] L. Gravano, P.G. Ipeirotis, N. Koudas, and D. Srivastava, "Text Joins in an RDBMS for Web Data Integration," Proc. 12th Int'l World Wide Web Conf. (WWW12), pp. 90-101, 2003.

[32] Bugzilla Bug 22353 - https://bugzilla.mozilla.org/show_bug.cgi?id=22353

[33] D. Gusfield. Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. USA: Cambridge University Press. ISBN 0-521-58519-8.

[34] S. Kirkpatrick; C.D. Gelatt; M. P. Vecchi (1983). Optimization by Simulated Annealing. *Science* **220** (4598): 671–680.

[35] Bug Fields: https://bugzilla.mozilla.org/page.cgi?id=fields.html

[36] BLAST: http://blast.ncbi.nlm.nih.gov/Blast.cgi

[37] D. Adjeroh, T. Bell, and A. Mukherjee, The Burrows-Wheeler Transform: Data Compression, Suffix Arrays and Pattern Matching, Springer, 2008.