Classification Algorithms for Detecting Duplicate Bug Reports in Large Open Source Repositories

Sarah E. Ritchey

May 8, 2014

Abstract

Software maintenance is an important part of software development. Of the hundreds of defects, or bugs, reported each day, up to a third are duplicates of another report. Processing these reports is a time consuming task. Therefore, automated techniques are being developed to recognize these duplications. This paper describes an improved method for automatic duplicate bug report detection based on new textual similarity features and binary classification. Using a set of new textual features, inspired from recent text similarity research, several binary classification models were trained. A case study was conducted on three open source systems: Eclipse, Open Office, and Mozilla to determine the effectiveness of the improved method. A comparison is also made with current state-of-the-art approaches highlighting similarities and differences. Results indicate that the accuracy of the proposed method is better than previously reported research with respect to all three systems.

1 Introduction

As software develops and becomes more complex, it is more likely that defects or bugs will occur in the source code. Many software development teams have implemented bug-tracking systems to keep track of bugs reported by users. When a new bug report is submitted, a triage must assign it to a developer to fix. It is very common for multiple users to find the same defect and write multiple reports about the same bug. This adds a tremendous amount of work for the triage, since they must determine if a new report is describing the same defect as a previously submitted report. If the triage fails to detect duplicates, multiple developers could fix the same problem. Not only is this an enormous waste of time and resources, but it also leads to inconsistent code. Some larger software projects can receive over 100 bug reports in a single day. Some software repositories have even reported that up to 30% of all reports in their systems are duplicates. Therefore it is essential to detect duplicates automatically to keep software development fiscally feasible.

Recently, research has been conducted to develop a way to help detect if a new report is a duplicate at the time it is submitted. A few software companies have already implemented some of these techniques. There are two major schools of thought on how to solve this problem. The first approach is to create a list containing the k most similar reports to the new report. This method reduces the work required by triage, since they only need to compare the new bug report to k other bug reports instead of the entire repository. It does not eliminate human error though, since the triage still needs to take the time to check this list and recognize if the report is a duplicate.

This is where the next approach, automatic classification, comes into play. Automatic classification systems automatically determine if an incoming report is a duplicate of another report. This is done by measuring the similarity between two reports and then comparing that to a threshold generated from test datasets. If it is above the threshold, it will be labeled as a duplicate and deemed to be resolved. This system would require no work by the triage. The main issue with this system is that a non duplicate report could potentially be mislabeled as duplicates (false positive) if the threshold is too low. Depending on the severity of the defect, this could have disastrous results for a project since the mislabeled bug would never be looked at or fixed.

This project looks into several state of the art automated classification systems and develops an improved method for measuring the similarity between two reports. The textual similarity features are derived using a system called *TakeLab* proposed by Saric et al. [5]. *TakeLab* automates measuring of semantic similarity of short texts using supervised machine learning. Several categorical variables are also used to generate additional features. After all the features are calculated several binary classification methods including Naïve Bayes and Support Vector Machines are run

to classify bugs as duplicate or non-duplicate. The new textual features derived by *TakeLab* work well with classification methods to detect duplicate pairs of bugs and outperform previous work.

2 Sample Bug Report

Although different software repositories use slightly different terminology, the basic components of a bug report are in all systems. Table 1 is an actual duplicate bug report from the Eclipse software repository. The first field of the bug_{id}, which is a unique identifier for that particular report. The next two fields, *description* and $short_{desc}$, give detailed accounts of the defect and how to recreate it. They are concatenated in Table 1. Next, *priority* and $bug_{severity}$ contain how urgent it is to fix this particular bug. The *product*, *version*, and *component* tell the specific piece of software that is having problems. The date the report was created and the last time it was edited are in $creation_{ts}$ and $delta_{ts}$ respectively. The *resolution* tells if the report was a a duplicate of another report (DUPLICATE), could not be reproduced (WORKSFORME), not yet resolved (OPEN), or if the defect was fixed (FIXED). If a report is a duplicate of another report (master), the bug_{id} of the master report is contained in the dup_{id} field. If a report does not have a DUPLICATE resolution, then the bug_{id} field will be blank. Finally the bug_{status} just states whether the report is open or closed.

Table 1: Sample Duplicate Defect Report

bug_{id}	214068
description	Failed to preview Chart Viewer ...
$short_{desc}$	Failed to preview ...
priority	P3
$bug_{severity}$	critical
product	BIRT
version	2.3.0
component	Build
$creation_{ts}$	1/2/08 1:35
$delta_{ts}$	1/2/08 4:32
resolution	DUPLICATE
dup_{id}	214069
bug_{status}	CLOSED

3 Datasets Used

The three systems used in the case study presented here are Eclipse, Open Office, and Mozilla. Using web scraping techniques, bug reports were collected from the Bugzilla websites of the three systems. Table 2 contains the the time interval when the bugs collected were submitted, the number of initial bugs collected ($Bugs_i$), the number of initial duplicates ($Dupl_i$), the number of bugs after preprocessing ($Bugs_f$), the number of duplicates after preprocessing ($Dupl_f$), and the total number of duplicate pairs ($DuplPairs$).

Table 2: Details of Bug Datasets

Dataset	From	To	$Bugs_i$	$Dupl_i$	$Bugs_f$	$Dupl_f$	$DuplPairs$
Eclipse	1/2008	12/2008	45,746	4,386	39,020	2,897	6,024
Open Office	1/2008	12/2010	31,333	4,549	23,108	2,861	6,945
Mozilla	1/2010	12/2010	78,236	10,777	65,941	6,534	16,631

After the bug reports were collected, reports that had an OPEN resolution were removed from the datasets. This was done because their status cannot be confirmed from the information available (i.e. the defect has not been fixed, therefore it could end up being a duplicate). Removing them would help prevent training the model with mislabeled data. It is also important to note that this could easily be done in an industrial setting by using historical data. The datasets were trimmed further by removing DUPLICATE resolution bugs that did not have masters in the dataset.

It is important to note that each group of duplicate reports could contain more than just 2 duplicate reports. The largest grouping in this data actually contained 15 separate reports. Therefore, each group of duplicates needed to be calculated. This was a tedious task, since the bug with the DUPLICATE resolution contains the master reports bug_{id} in the dup_{id} field. It is impossible to tell that the master report is a duplicate unless it is a duplicate of an additional report. It is possible to get long chains of duplicate bugs that branch in both directions. The number of reports in each group was then used to generate all the duplicate bug report pairs. Therefore, the following algorithm was written to find all the duplicate bugs in its group and calculate the total number of duplicate pairs.

```
data = allBugReports
dupData = allBugReportsWithDuplicateResolution
dictionary = {}
for bug in dupData:
    equivBugs[] = list of reports equivalent to bug
    duplicateID = Id of report that bug is marked a duplicate of
    bugID= Id of bug
    add duplicateID and bugID to equivBugs list
```

```
    wereNewBugsAdded=true
    while(wereNewBugsAdded)
        if reports in equivBugs are duplicates
            add master to equivBugs if it is not already in equivBugs
        if reports in equivBugs are master
            add duplicates to equivBugs if it is not already in equivBugs
        else
            wereNewBugsAdded=false
    dictionary[bugID]=equivalentBugs
dupGroups[]=unique sets of equivalent bugs from dictionary
numberOfPairs=0
for group in dupGroups:
    lenChoose2=factorial(len(group))/factorial(len(group)-2)/factorial(2)
    numberOfPairs+=lenChoose2
```

Our model takes pairs of duplicate bug reports in addition to randomly generated pairs and determines if they are duplicate or not. To be able to check the results, an additional feature, *decision*, was added to each pair of bug report to indicate whether or not it is a duplicate. The *decision* field was set to 1 and -1 for duplicate and non-duplicate pairs respectively. To be consistent with original ratios in our test data sets, about four times as many non-duplicate pairs of bugs were generated for the training dataset.

These are the same datasets that are used in Sun et al. [6]. The Eclipse dataset is referred to as Eclipse2008 in Sun et al. However, open bugs were not removed from their datasets. Whereas, our datasets do not contain open bug reports.

4 Method

This section describes our method for deriving the features and training our models. We first explain how the features are generated, followed by the classification method and the evaluation measures.

4.1 Generating Features

An important assumption made in this research is that duplicate reports will have similar entries in most fields. Therefore, 25 features were created to measure this similarity. The following report fields were selected as the base for the classification features: bug_{id}, $short_{desc}$, $description$, $product$, $component$, $bug_{severity}$, $priority$, $version$, $creation_{ts}$. The most important of these fields are the $short_{desc}$ and $description$. Because they contain detailed descriptions of the defect, they contain a great deal of information that needs to be quantified. To this end, the two fields are concatenated

together and then used to generate 18 different numeric features. These features, inspired by Saric et al. [5], are generated using the *simple* TakeLab system and include: n-gram word overlap for unigrams, bigrams and trigrams, n-gram word overlap for unigrams, bigrams and trigrams after lemmatization, WordNet based augmented word overlap, weighted word overlap, normalized differences for sentence length and aggregate word information content, shallow named entity, and numbers overlap. Specifics for each feature can be found in the sections below. The TakeLab system was designed to generate a similarity score between 1 and 5 for pairs of sentences or short text. After a set of features are calculated, Support Vector Regression predicts the similarity score.

4.1.1 N-gram Word Overlap

Note an n-gram is a set of n consecutive words from the concatenated $short_{desc}$ and *description* fields. Let S_1 and S_2 be the sets of consecutive n-grams in the first and the second report, respectively. The n-gram overlap is the harmonic mean of the degree to which the second report covers the first and the degree to which the first report covers the second.

Definition The ngram overlap is defined as follows:

$$ngo(S_1, S_2) = 2\left(\frac{|S_1|}{|S_1 \cap S_2|} + \frac{|S_2|}{|S_1 \cap S_2|}\right)^{-1} \tag{1}$$

The overlap, defined by eq. (1), is computed for unigrams (1-gram), bigrams (2-grams), and trigrams (3-grams).

4.1.2 N-gram Word Overlap After Lemmatization

The next set of features are very similar to n-gram word overlap and can be calculated using eq. (1). The only difference is that n-grams are created using only content words. Content words are nouns, verbs, adjectives, and adverbs. Intuitively, the function words (prepositions, code, conjunctions, articles) carry less semantic information than content words. Therefore, removing them might eliminate the noise and provide a more accurate estimate of semantic similarity. N-grams created using this process are then called lemmas. The overlap after lemmatization is calculated for unigrams, bigrams, and trigrams.

4.1.3 WordNet Based Augmented Word Overlap

WordNet is a hierarchical network of words linked by word relations. Therefore, similar words will be closer together than unrelated words. One can expect a high unigram overlap between very similar sentences only if exactly the same words (or

lemmas) appear in both sentences. To allow for some lexical variation, we use WordNet to assign partial scores to words that are not common to both sentences.

Definition We define the WordNet augmented coverage:

$$PWN(S_1, S_2) = \frac{1}{|S_2|} \sum_{w_1 \in S_1} score(w_1, S_2)$$

$$score(w, S) = \begin{cases} 1, & \text{if } w \in S \\ \max_{w' \in S} sim(w, w'), & \text{otherwise} \end{cases}$$

where $sim(w, w')$ represents the WordNet path length similarity.

Definition The WordNet-augmented word overlap feature is defined as a harmonic mean of $PWN(S_1, S_2)$ and $PWN(S_2, S_1)$ or

$$wnba(S_1, S_2) = 2 \left(\frac{1}{PWN(S_1, S_2)} + \frac{1}{PWN(S_2, S_1)} \right)^{-1}.$$

4.1.4 Weighted Word Overlap

When measuring sentence similarities we give more importance to words bearing more content, by using the information content

$$ic(w) = \ln \left(\frac{\sum_{w' \in C} freq(w')}{freq(w)} \right)$$

where C is the set of words in the corpus and $freq(w)$ is the frequency of the word w in the corpus. We use the Google Books Ngrams to obtain word frequencies because of its excellent word coverage for English. Let S_1 and S_2 be the sets of words occurring in the first and second sentence, respectively.

Definition The weighted word coverage of the second sentence by the first sentence is given by:

$$wwc(S_1, S_2) = \frac{\sum_{w \in S_1 \cap S_2} ic(w)}{\sum_{w' \in S_2} ic(w')}$$

The weighted word overlap between two sentences is calculated as the harmonic mean of the $wwc(S_1, S_2)$ and $wwc(S_2, S_1)$, or

$$wwo(S_1, S_2) = 2 \left(\frac{1}{wwc(S_1, S_2)} + \frac{1}{wwc(S_2, S_1)} \right)^{-1}$$

This measure proved to be very useful, but it could be improved even further. Misspelled frequent words are more frequent than some correctly spelled but rarely used words. Hence dealing with misspelled words would remove the inappropriate heavy penalty for a mismatch between correctly and incorrectly spelled words.

4.1.5 Normalized Differences

Intrinsically, longer reports with more words are more likely to be similar to other reports, since the probability that the same words are used is higher. Therefore the several features are included in this model to measure the normalized differences in a pair of reports according to both sentence length and the aggregate word information content. This prevents longer reports from having an advantage over shorter reports.

4.1.6 Shallow Named Entity

Proper nouns convey a huge amount of information. Therefore, several Shallow Named Entity similarity features were created to measure the similarity of these named entities. This model calculates the overlap of capitalized words and the overlap of stock index symbols. Note that only capitalized words longer than one character are considered to be named entities, and words in all capital letters are considered to be stock index symbols. Named entities are also classified into the following types: persons, organizations, locations, dates, and rudimentary temporal expressions. The overlap of each class are calculated separately for each pair of reports. The absence of an entity class from both sentences is indicated by a separate binary feature (one feature for each class).

4.1.7 Numbers Overlap

Another set feature, Numbers Overlap, helps to remove bias against reports containing different sets of numbers.

Definition Let N_1 and N_2 be sets of number in two sentences: Define three separate numbers overlap features to be

$$nof1(N_1, N_2) = \log(1 + |N_1| + |N_2|)$$

$$nof2(N_1, N_2) = \frac{2|N_1 \cap N_2|}{|N_1| + |N_2|}$$

and

$$nof3(N_1, N_2) = \begin{cases} 1, & \text{if } N_1 \subseteq N_2 \text{ or } N_2 \subseteq N_1 \\ 0, & \text{otherwise} \end{cases}.$$

Additionally, the numbers that differ only in the number of decimal places are treated as equal (e.g., 65, 65.2, and 65.234 are treated as equal, whereas 65.24 and 65.25 are not).

4.1.8 Categorical Features

Again, assume that duplicate bug reports will have similar entries in in the most fields of the report. Therefore, several features were included to quantify these similarities.

Definition Let b_1 and b_2 be bug reports. Then $b_i.property$ is the field called property of b_i. Let f_n be the nth feature of the model defined by:

$$
\begin{aligned}
f_{19}(b_1, b_2) &= \begin{cases} 1, & \text{if } b_1.product = b_2.product \\ 0, & \text{otherwise} \end{cases} \\
f_{20}(b_1, b_2) &= \begin{cases} 1, & \text{if } b_1.component = b_2.component \\ 0, & \text{otherwise} \end{cases} \\
f_{21}(b_1, b_2) &= \begin{cases} 1, & \text{if } b_1.bug_{severity} = b_2.bug_{severity} \\ 0, & \text{otherwise} \end{cases} \\
f_{22}(b_1, b_2) &= \frac{1}{1 - |b_1.priority - b_2.priority|} \\
f_{23}(b_1, b_2) &= \frac{1}{1 - |b_1.version - b_2.version|}
\end{aligned}
$$

Features 19 - 23 were adapted from Sun et. al's paper [6]. When calculating feature number 23 (version related), if the version is 'unspecified' for at least one of the bugs in the pair, the value of the feature is set to 0.5.

Runeson [4] concluded that 53% of all duplicates were submitted in an interval of 20 days after the master bug was submitted. Based on Runeson findings, we decided to calculate a new feature as the absolute value between the open dates of a pair of bugs.

$$f_{24}(b_1, b_2) = |b_1.creation_{ts} - b_2.creation_{ts}|$$

The last feature, inspired from Sureka et al. [7], computes the absolute difference between the bug_{ids} of the two bugs in the pair.

$$f_{25}(b_1, b_2) = |b_1.bug_{id} - b_2.bug_{id}|$$

Feature 25 is represented in Figure 1 as a group histogram that shows the difference between the duplicate pairs as light gray and non-duplicate pairs represented as black. The feature discriminates the two categories well. Most duplicate bugs have a smaller bug_id difference compared with non-duplicate bugs.

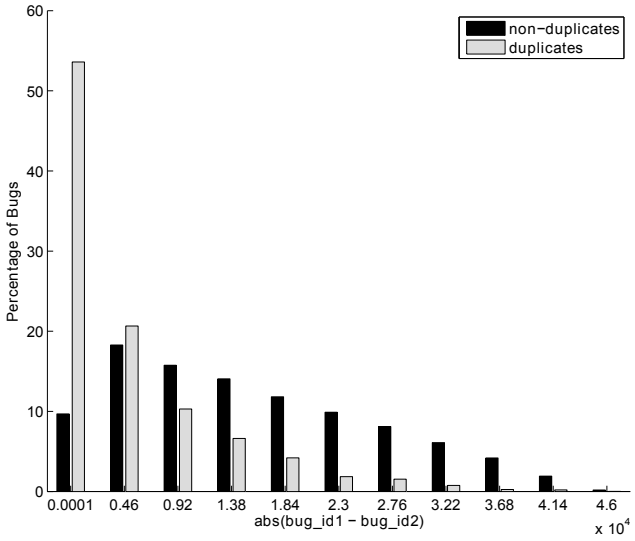

Figure 1: A histogram for feature 25. The x-axis represents intervals for the bug_id differences

4.2 Building the Training and Testing Sets

After we generate the features for all the bugs in the datasets, we divide the initial dataset into a training set and a testing set. The training set contains 5,000 bug pairs and all the other pairs are put into the testing set. The instances are divided into the subsets using stratified sampling, so the percentage between the two classes is preserved. That means out of the 5,000 pairs, 1,000 pairs are duplicate and 4,000 are non-duplicate. Next, all the values in the training and the testing datasets are scaled to the $[-1, 1]$ interval. First the training set is scaled and the ranges are then applied to scale the testing set. All the data is available at `http://www.csis.ysu.edu/~alazar/msr14`.

4.3 Binary Classification

At this point, the datasets contain pairs of duplicate and non-duplicate bug reports. To be able to automatically identify future bugs as duplicates of existing bugs, classification methods are used. The machine learning field provides several methods. One of the most popular algorithms today is the support vector machine and its implementation called *LibSVM* [2]. The *SVM* classification model discriminates well between pairs of duplicate and non-duplicate bugs, provides excellent results in terms of accuracy and runs in an acceptable amount of time. For the SVM method, the two best parameters C and γ are found performing a grid search done with cross-validation on the training set. After the model predicts the class for the each instance in the training set, the evaluation measures are computed.

See Figure 2 for the grid search on the Eclipse dataset. Other classification meth-

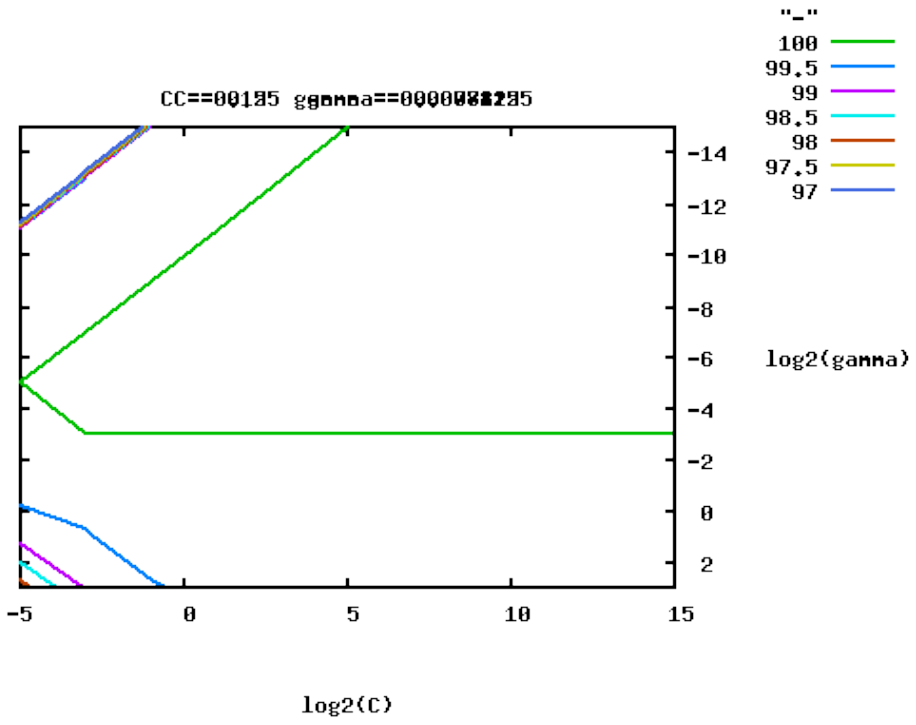

Figure 2: Grid Search for LibSVM

ods are implemented in the Python package, *scikit-learn* [3]. We ran the following methods from *scikit-learn*: K Nearest Neighbours, Linear Support Vector Machine, RBF Support Vector Machine, Decision Tree, Random Forest and Naïve Bayes.

4.3.1 K Nearest Neighbors

K Nearest Neighbors (KNN) classification is a type of instance-based learning. This means that a general model is not constructed from the training data. Instead, all instances in the training dataset are stored and used to classify a new query. When a new query is tested, the closest k neighbors are calculated using Euclidean distance. If the majority of those neighbors belong to one class, then the new query is predicted to be in that class as well. So for this system, if the majority of the k neighbors are duplicates, then the test pair is labeled as a duplicate. In the scikit-learn implementation, k is an integer value specified by the user. The optimal choice of the value k is highly data-dependent: in general a larger k suppresses the effects of noise, but makes the classification boundaries less distinct.

4.3.2 Suport Vector Machines

The first support vector machine (SVM) used in this study is a Linear SVM.

Definition Given some training data $\mathcal{D}$, a set of n points of the form

$$\mathcal{D} = \{(\mathbf{x}_i, y_i) \mid \mathbf{x}_i \in \mathbb{R}^m,\ y_i \in \{-1, 1\}\}_{i=1}^{n}$$

where the y_i is either 1 or -1, indicating the class to which the point $\mathbf{x}_i$ belongs. Each $\mathbf{x}_i$ is a m-dimensional real vector. If the training data are linearly separable, two hyperplanes can be found that split the data such that all cases where $y_i = 1$ are on one side, all cases where $y_i = -1$ are on the other side, and no points are between the two planes. These hyperplanes can be described by the equations

$$\mathbf{w} \cdot \mathbf{x} - b = 1 \text{ and } \mathbf{w} \cdot \mathbf{x} - b = -1.$$

The largest such planes are called maximum-margin hyperplanes. The region bounded by them is called "the margin". A third hyperplane, between the maximum-margin hyperplanes, is then used to divide the the margin in half. Test pairs are then mapped into m-space. If the point is on the side of the third hyperplane as all points with $y_i = 1$, it is predicted to be a duplicate and if $y_1 = -1$ it is predicted to be a non-duplicate.

The second support vector machine used in this study is a Radial Basis Function Support Vector Machine (RBF SVM). This algorithm is similar to Linear SVM, except that the dot products used to define the hyperplane are replaced by a nonlinear kernel function, gaussian radial basis function.

Definition Let $r = \|\mathbf{x} - \mathbf{x}_i\|$ and $\epsilon > 0$. Then, define the gaussian radial basis function to be:

$$\phi(r) = e^{-(\varepsilon r)^2}$$

This allows the algorithm to fit the maximum-margin hyperplane in a transformed feature space. The transformation may be nonlinear and the transformed space high dimensional; thus though the classifier is a hyperplane in the high-dimensional feature space, it may be nonlinear in the original input space.

When the Gaussian radial basis function is used, the corresponding feature space is actually a Hilbert space of infinite dimension! A Hilbert space is an abstract vector space possessing the structure of an inner product that allows length and angle to be measured.

4.3.3 Decision Tree and Random Forest

Decision trees are created by having a series of linked binary check points (decision) that determine the probability that a new report is a duplicate. So in these tree

structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels. In decision analysis, a decision tree can be used to visually and explicitly represent decisions and decision making.

A tree can be "learned" by splitting the source set into subsets based on the value of each feature. This process is repeated on each derived subset in a recursive manner called recursive partitioning. The recursion is completed when the subset at a node has all the same value of the target variable, or when splitting no longer adds value to the predictions. This process of top-down induction of decision trees is an example of a greedy algorithm, and it is by far the most common strategy for learning decision trees from data.

A random forest classifier uses a number of decision trees, in order to improve the classification rate. It begins by selecting a random sample from the training set and develops a decision trees to that particular set of data. This process is repeated and the resulting trees are then averaged together to create a random forest.

4.3.4 Naïve Bayes

Naïve Bayes classifiation assumes that features are independent of other features. Therefore, it is unrelated to the presence or absence of any other feature, given the class variable. It also takes into account the probability of a test case belonging to one class over another. In this experiment, there were 4 times an many non-duplicates compared to duplicates in the training data. Therefore, prior probability would be 4/5 for non-duplicates and 1/5 for duplicates. When a new query is introduced, the proportions of neighboring duplicates and non duplicates are calculated. These proportions are compared with the expected probabilities. The new query is then labeled accordingly (i.e. if more that 1/5 of the neighboring reports are duplicates, then it is classified as a duplicate and vise versa).

4.4 Measures

Usually, classification methods are evaluated using the accuracy measure which is calculated as the percentage of correctly classified instances. However, the accuracy does not paint the entire picture, especially in case of unbalanced datasets. There are fewer duplicate bugs than non-duplicates in the datasets intrinsically. In the dataset constructed, there were 4 times less pairs of duplicate bugs than non-duplicates. The accuracy can still be high, even if a significant number of instances from the positive class (duplicate pairs in our case) were classified incorrectly. To avoid this problem, we consider three other measures: precision, recall, and the area under the curve (AUC). The standard definitions for these measures are shown below.

Definition Let t_p and t_n be the number of instances where duplicates and non-duplicates were correctly identified respectively. Then, let f_p be the number of in-

stances where non-duplicates were mislabeled as duplicates and f_n be the number of instances where duplicates were mislabeled as non-duplicates. Then define the following measures:

$$\text{Accuracy} = \frac{t_p + t_n}{t_p + t_n + f_p + f_n}$$

$$\text{Precision} = \frac{t_p}{t_p + f_p}$$

$$\text{Recall} = \frac{t_p}{t_p + f_n}$$

Area Under the Curve (AUC) is simply a graphical plot which illustrates the performance of a binary classifier system as its discrimination threshold is varied. It is created by plotting the fraction of true positives out of the total actual positives vs. the fraction of false positives out of the total actual negatives, at various threshold settings.

5 Preliminary Results and Observations

In this section we show that the textual features extracted by TakeLab together with categorical features provide very good classification results. Accuracy is over 99% for all combinations of datasets and classification algorithms. A recall of 100% was obtained for all datasets. This means that all the positive instances (duplicate pairs of bugs) were correctly classified. Precision is also high, but not 100%. This means that sometimes few pairs of non-duplicate bug pairs were classified as duplicates. See Tables 3, 4, and 5 for the results of Eclipse, Open Office, and Mozilla respectively.

With respect to the Eclipse dataset, Naïve Bayes provides the best accuracy of 100%, which means that all the bug pairs were classified correctly. For Open Office, the highest accuracy is given by multiple algorithms, but recall is still 100%. From the confusion matrices we see only two pairs of bug reports were misclassified. The last table contains results related to the largest dataset in this case study: Mozilla. Linear SVM and Nearest Neighbors returned the best results and recall of 100%.

We also ran experiments using all 25 features, only TakeLab features (18) and only categorical features (7). All three experiments give almost identical results. We also experimented with the first three features from *TakeLab* and the five categorical features used by Sun et al.[6] and Alipour et al. [1] but this did not give good results.

The proposed approach is similar with the one described by Alipour et al. [1], with the exception of the features used. Three of the classification algorithms are common, but the additional algorithms we used may work better for larger datasets. The difference in results between the two approaches are reported in Table 6. The new set of textual features presented in this paper improves the accuracy between 3.25% and 6.32% over the contextual approach proposed by Alipour et al.

Sun et al. [6] were the first ones to propose the set of five categorical features (features 19 - 23), in addition of two textual measures based on the *BM25F* measure. The results in [6] are reported in terms of top-k recall rates and are not directly comparable with our results. However, no more than 80% of the duplicates were correctly identified compared with the 100% classification recall rates we obtained.

Table 3: Eclipse Results

	NK Neighbors	LSVM	RBF SVM	Decision Tree	Random Forest	Naïve Bayes
Accuracy:	0.99992	0.99996	0.999841	0.999841	0.99996	1
Precision:	0.999605	0.999803	0.999211	0.999211	0.999803	1
Recall:	1	1	1	1	1	1
AUC:	0.99995	0.999975	0.9999	0.9999	0.999975	1

Table 4: Open Office Results

	NK Neighbors	LSVM	RBF SVM	Decision Tree	Random Forest	Naïve Bayes
Accuracy:	0.999933	0.999933	0.999697	0.999798	0.999899	0.999933
Precision:	0.99966	0.99966	0.998471	0.99898	0.99949	0.99966
Recall:	1	1	1	1	1	1
AUC:	0.999958	0.999958	0.999811	0.999874	0.999937	0.999958

Table 5: Mozilla Results

	NK Neighbors	LSVM	RBF SVM	Decision Tree	Random Forest	Naïve Bayes
Accuracy:	0.999949	0.999949	0.999936	0.999347	0.999936	0.999808
Precision:	0.999745	0.999745	0.999681	0.996943	0.999808	1
Recall:	1	1	1	0.999808	0.999872	0.999042
AUC:	0.999968	0.999968	0.99996	0.99952	0.999912	0.999521

6 Conclusions and Future Work

The paper presents an improved method to detect duplicate bug reports based on textual similarity measures. *TakeLab*, a text similarity system, is used to generate a majority of the features. A total of 25 new textual features are used. After determining the features, binary classification methods were run to categorize bugs into two classes: duplicate or non-duplicate. We tested this method on bug reports from Eclipse, Open Office, and Mozilla. Our method improves duplicate bug report detection by 6.32% even without the use of context based features as reported by Alipour et al. [1]. These preliminary results are very promising. In future work, we plan on using 10 times the size of the current datasets used to see if the current results hold.

Table 6: Comparison with Alipour's Results [1]

	New Features		Alipour	
	Accuracy	AUC	Accuracy	AUC
Eclipse	100.0000	1.0000	96.75	0.9900
Open Office	99.9899	0.9999	93.67	0.9660
Mozilla	99.9930	0.9999	94.78	0.9430

7 Bibliography

[1] A. Alipour, A. Hindle, and E. Stroulia. A contextual approach towards more accurate duplicate bug report detection. *Proceedings of the Tenth International Workshop on Mining Software Repositories*, pages 183–192, 2013.

[2] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.

[3] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[4] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. *29th International Conference on Software Engineering*, pages 499–510, 2007.

[5] F. Saric, G. Glavas, M. Karan, J. Snajder, and B. Basic. Takelab: Systems for measuring semantic text similarity. In *Proceedings of the First Joint Conference on Lexical and Computational Semantics*, pages 441–448, Montreal, Canada, June 2012.

[6] C. Sun, D. Lo, S. Khoo, and J. Jiang. Towards more accurate retrieval of duplicate bug reports. *Proceedings of the 26th IEEE/ACM Automated Software Engineering*, pages 253–262, 2011.

[7] A. Sureka and P. Jalote. Detecting duplicate bug report using character n-gram-based features. *Asia Pacific Software Engineering Conference*, pages 366–374, 2010.

8 Appendix

8.1 Acknolegements

Thanks to Dr. Lazar and Dr. Sharif for advisement!

8.2 Project Website and Blog

- http://www.csis.ysu.edu/∼ creu/
- http://ysu-creu13-14.blogspot.com

8.3 Publications

- Lazar, A., Ritchey., S., Sharif, B., Improving the Accuracy of Duplicate Bug Report Detection using Textual Similarity Measures, The 11th Working Conference on Mining Software Repositories (MSR), Hyderabad, India, 2014, 4 pages to appear.
- Lazar, A., Ritchey., S., Sharif, B., Generating Duplicate Bug Datasets, The 11th Working Conference on Mining Software Repositories (MSR), Hyderabad, India, 2014, 4 pages to appear.