Duplicate Bug Tracker

Technical Report · August 2016

3 authors:

Manish Munikar
University of Texas at Arlington
8 PUBLICATIONS   **95** CITATIONS

SEE PROFILE

Sushil Shakya
Tribhuvan University
4 PUBLICATIONS   **86** CITATIONS

SEE PROFILE

Brihat Ratna Bajracharya
Tribhuvan University
6 PUBLICATIONS   **8** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Movie Review Mining and Recommendation System View project

Duplicate Bug Tracker View project

TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

A

MINOR PROJECT REPORT

ON

DUPLICATE BUG TRACKER

By:

Manish Munikar, 070-BCT-520

Sushil Shakya, 070-BCT-547

Brihat Ratna Bajracharya, 070-BCT-513

A PROJECT WAS SUBMITTED TO THE DEPARTMENT OF ELECTRONICS AND
COMPUTER ENGINEERING IN PARTIAL FULLFILLMENT OF THE
REQUIREMENT FOR THE BACHELOR'S DEGREE IN COMPUTER ENGINEERING

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
LALITPUR, NEPAL

AUGUST, 2016

ACKNOWLEDGMENT

We would like to convey our acknowledgement to our seniors and teachers for their support and encouragement. Without their sheer support this project wouldn't have been possible. We owe debt of gratitude to everybody involved directly or indirectly for bringing this project into existence.

We would like to take this opportunity to thank our supervisor Ms. Bibha Sthapit for her continuous support and guidance in every way possible. Also we want to thank our lecturer Mr. Basanta Joshi for his help on the topics of Artificial Intelligence and making the concept clearer to us.

We would like to thank our seniors for providing us the necessary materials required for the development of this software. The patience, support, help and approval provided by everybody in every way is simply unfathomable.

Every attempt has been made to include each and every aspects of the project in this documentation so that the reader can clearly understand about our project. We would be pleased to get the feedbacks on this project. Finally we would like to express our gratitude to our family, friends and well wishers for encouraging and supporting us.

Sincerely,
Brihat Ratna Bajracharya, `070-BCT-513`
Manish Munikar, `070-BCT-520`
Sushil Shakya, `070-BCT-547`

ABSTRACT

An open source project typically maintains an open bug repository so that bug reports from all over the world can be gathered. When a new bug report is submitted to the repository, a person, called a triager, examines whether it is a duplicate of an existing bug report. If it is, the triager marks it as duplicate and the bug report is removed from consideration for further work. To address the problem with duplicate bug reports, a person called a triager needs to manually label these bug reports as duplicates, and link them to their *master* reports for subsequent maintenance work. However, in practice there are considerable duplicate bug reports sent daily; requesting triagers to manually label these bugs could be highly time consuming. To address this issue, recently, several techniques have be proposed using various similarity based metrics to detect candidate duplicate bug reports for manual verification. Automating triaging has been proved challenging as two reports of the same bug could be written in various ways. There is still much room for improvement in terms of accuracy of duplicate detection process. In this paper, we leverage recent advances on using discriminative models for information retrieval to detect duplicate bug reports more accurately. We have consulted an ample number of research article for this project. We have used the approaches that they have mentioned but with a little twist of our own and we have also described the rationale for why we did so. We have validated our approach on two software bug repositories from Firefox and Linux.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

BFGS	Broyden-Fletcher-Goldfarb-Shanno
BTS	Bug Tracking System
DSF	Django Software Foundation
ERD	Entity Relationship Diagram
HTTP	Hyper Text Transfer Protocol
IDE	Integrated Development Environment
IDF	Inverse Document Frequency
ITS	Issue Tracking System
IR	Information Retrieval
L-BFGS	Limited-memory Broyden-Fletcher-Goldfarb-Shanno
MVC	Model-View-Controller
ORM	Object-Relational Mapper
OS	Operating System
RAM	Random Access Memory
TF	Term Frequency
UML	Unified Modeling Language
URL	Uniform Resource Locators
VRAM	Video Random Access Memory

1.   INTRODUCTION

Our project entitled as "**Duplicate Bug Tracker**" is a web based bug tracking system that uses the natural language information provided by the bug reporter in order to track the duplicate bugs and inform the reporter about them. Natural language information means the summary and description of the bug report entered by the user. Whenever a user reports about a new bug, the system compares the bug information with the ones already present in the database. The system uses natural language processing in order to compare the bug reports.

Due to complexities of systems built, software often comes with defects. Software defects have caused billions of dollars loss.[1] Many open source software projects incorporate open bug repositories during development and maintenance so that both developers and users can report bugs that they have encountered. There are at least two important advantages of using such a bug repository. The bug repository allows users all around the world to be *testers* of the software, so it can increase the possibility of revealing defects and thus increase the quality of the software.

Despite the benefits of a bug reporting system, it does cause some challenges. As bug reporting process is often uncoordinated and ad-hoc, often the same bugs could be reported more than once by different users. Hence, there is often a need for manual inspection to detect whether the bug has been reported before. If the incoming bug report is not reported before then the bug should be assigned to a developer. However, if other users have reported the bug before then the bug would be classified as being a duplicate and attached to the original first-reported *master* bug report. This process referred to as *triaging* often takes much time. For example, for the Mozilla programmers, it has been reported in 2005 that "Everyday, almost 300 bugs appear that need triaging. This is far too much for only the Mozilla programmers to handle."[2]

In order to ease the heavy burden of triagers, there have been recent techniques to automate the triaging process in two ways. The first one is automatically filtering duplicates to prevent multiple duplicate reports from reaching triagers.[3] The second is providing a list of similar bug reports to each incoming report under investigation.[3, 4, 5] With the help of these, rather than checking against the entire collection of bug reports, a triager could first inspect the top-k most similar bug reports returned by the systems. If there is a report in the list that reports about the same defect as the new one, then the one is a duplicate, the triager then marks it as a duplicate and adds a link between the two duplicates for subsequent maintenance work.

Instead of filtering duplicates, we choose the second approach as duplicate bug reports are not necessarily bad. As stated in [6], one report usually does not carry enough information for developers to dig into the reported defect, while duplicate reports can complement one another.

To achieve better automation and thus save triagers' time, it is important to improve the quality of the ranked list of similar bug reports given a new bug report. There have been several studies on retrieving similar bug reports. However, the performance of these systems is still relative low, making it hard to apply them in practice. The low performance is partly due to the following limitations of the current methods. First, all the three techniques in [3, 4, 5] employ one or two features to describe the similarity between reports, despite the fact that other features are also available for effective measurement of similarity. Second, different features contribute differently towards determining similarities. For example, the feature capturing similarities between summaries of two reports are more effective than that between descriptions, as summaries typically carry more concise information. However, as project contexts evolve, the relative importance of features might vary. This can cause the past techniques, which are largely based on absolute rating of importance, to deteriorate in their performance.

More accurate results would mean more automation and less effort by triagers to find duplicate bug reports. To address this need, we propose a probabilistic model based approach that further improves accuracy in retrieving duplicate bug reports by up to 35% on real bug report datasets. Different from the previous approaches that rank similar bug reports based on similarity score of vector space representation, we develop a probabilistic model to retrieve similar bug reports from a bug repository. We make use of the recent advances in information retrieval community that shows the probability of two documents being similar with each other. We strengthen the effectiveness of bug report retrieval system by introducing many more relevant features to capture the similarity between bug reports. Moreover, with the adoption of the probabilistic model approach, the relative importance of each feature will be automatically determined by the model through assignment of an optimum weight.

Consequently, as bug repository evolves, our probabilistic model also evolves to guarantee the all the weights remain optimum at all time. In this sense, our process is more adaptive, robust, and automated. We evaluate our probabilistic model approach on two sample bug report datasets from large programs including Firefox, an open source web browser, Linux, an open source operating system. But it is ineffective to compare the new report with each and every bug reports present in the database because this will degrade the system performance. For solving this, the level of similarity of each bug with every other bug present in the

database is measured on frequent basis and the results are stored in some form of a table.

It isn't always the case that the similarity measured by the algorithms used provides the expected results, i.e., the similar bug reports listed may not match the bug that the user is facing. So, it is necessary to tell the system that the results displayed by it are as per expectations or not. And for doing this, the system makes the use of machine learning. The system continuously takes in the user feedback for the results displayed by it and uses these feedbacks for displaying results for future purpose, i.e., the system learns on its own.

2. BACKGROUND

Basically, duplicate bug report retrieval involves information extraction from comparison between documents in natural language. This section deals with necessary background and foundation techniques to perform the tasks in our approach.

2.1. Duplicate Bug Reports

A bug report is a structured record consisting of several fields. Commonly, they include summary, description, project, submitter, priority and so forth. Each field carries a different type of information. For example, summary is a concise description of the defect problem while description is the detailed outline of what went wrong and how it happened. Both of them are in natural language format. Other fields such as project, priority try to characterize the defect from other perspectives. In a typical software development process, the bug tracking system is open for testers or even for all end users, so it is unavoidable that two people may submit different reports on the same bug. This causes the problem of duplicate bug reports. As mentioned in [4], duplicate reports can be divided into two categories. One describes the same failure, and the other depicts two different failures both originated from the same root cause. In our project, we only handle the first category. As an example, Table 2.1 shows some pairs of duplicate reports extracted from Issue Tracker of OpenOffice. Only the summaries are listed.

ID	Summary
85064	[Notes2] No Scrolling of document content by use of the mouse wheel
85377	[CWS notes2] unable to scroll in a note with the mouse wheel
85487	connectivity: evoab2 needs to be changed to build against changed api
85496	connectivity fails to build (evoab2) in m4

Table 2.1: Examples of Duplicate Bug Reports

Usually, new bug reports are continually submitted. When triagers identify that a new report is a duplicate of an old one, the new one is marked as duplicate. As a result, given a set of reports on the same defect, only the oldest one in the set is not marked as duplicate. We refer to the oldest one as master and the others as its duplicates. A bug repository could be viewed

as containing two groups of reports: masters and duplicates. Since each duplicate must have a corresponding master and both reports are on the same defect, the defects represented by all the duplicates in the repository belong to the set of the defects represented by all the masters. Furthermore, typically each master report represents a distinct defect.

2.2. Information Retrieval

Information retrieval (IR) aims to extract useful information from unstructured documents, most of which are expressed in natural language. IR methods typically treat documents as "bags of words" and subsequently represent them in a high-dimensional vector space where each dimension corresponds to a unique word or term. The following describes some commonly used strategies to pre-process documents and methods to weigh terms.

2.2.1. Pre-processing

In order to computerize retrieval task, sequence of actions should be taken first to pre-process documents using natural language processing techniques. Usually, this sequence comprises tokenization, stemming and stop-word removal. A word token is a maximum sequence of consecutive characters without any delimiters. A delimiter in turn could be a space, punctuation mark, etc. Tokenization is the process of parsing a character stream into sequence of word tokens by splitting the stream by the delimiters. Stemming is the process to reduce words to their ground forms. The motivation to do so is that different form of words derived from the same root usually have similar meanings. By stemming, computers can capture this similarity via direct string equivalence. For example, a stemmer can reduce both "playing" and "play" to "play". The last action is stop-word removal. Stop-words are those words carrying little helpful information for information retrieval task. These include pronouns such as "it", "he" and "she" link verbs such as "is", "am" and "are", etc. In our stop word list, in addition to removing 30 common stop words, we also drop common abbreviations such as "I'm", "that's", "we'll", etc.

2.2.2. Term weighting

TF-IDF (Term Frequency, Inverse Document Frequency) is a common term weighting scheme. It is a statistical approach to evaluating the importance of term in a corpus. TF is a

local importance measure. Given a term and a document, in general, TF corresponds to the number of times the term appears within the document. Different from TF, IDF is a global importance measure most commonly calculated by the formula within the corpus,

$$\text{IDF}(term) = \log\left(\frac{D_{all}}{D_{term}}\right) \tag{2.1}$$

where,

$$D_{all} = \text{number of documents in the corpus}$$
$$D_{term} = \text{number of documents containing } term$$

Given a term, the fewer documents it is contained in, the more important it becomes. In our approach, we employ both TF and IDF to weigh terms.

2.3.   Logistic Regression

In general, when we make a machine learning based software, we are basically trying to come up with a function to predict the output for future inputs based on the experience it has gained through the past inputs and their outputs. The past data is referred to as the *training set*.

Logistic regression (also known as *logit regression* or *logit model*) is one of the most popular machine learning algorithms used for classification problem. Given a training set having one or more independent (input) variables where each input set belongs to one of predefined classes (categories), what logistic regression model tries to do is come up with a probability function that gives the probability for the given input set to belong to one of those classes. The basic logistic regression model is a binary classifier (having only 2 classes), i.e., it gives the probability of an input set to belong to one class instead of the other. If the probability is less than 0.5, we can predict the inputs set to belong to the latter class. But logistic regression can be hacked to work for multi-class classification problem as well by using concepts like "*one-vs-rest*". What we basically do is create a classifier for each class that predicts the probability of an input set to belong to that particular class instead of all other classes. It is popular because it is a relatively simple algorithm that performs very well on wide range of problem domains.

Actually, logistic regression is one of the techniques borrowed by machine learning from the field of statistics. Logistic regression was developed by statistician David Cox in 1958. The binary logistic model is used to estimate the probability of a binary response based on one or more predictor (or independent) variables (called *features*).

The name "*logistic*" comes from the probability function used by this algorithm. The logistic function (also known as *sigmoid* function) is defined as:

$$\text{logistic}(x) = \text{sigmoid}(x) = \frac{e^x}{1 + e^x} = \frac{1}{1 + e^{-x}} \tag{2.2}$$

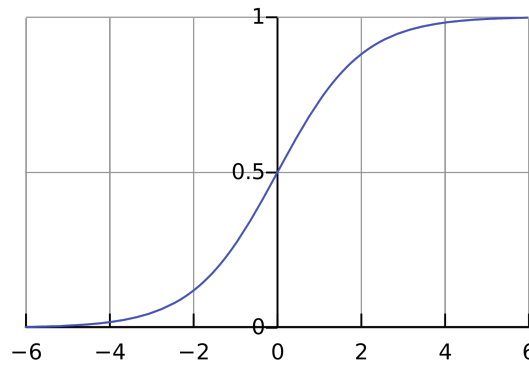The graph of this function is given in Figure 2.1.

Figure 2.1: Graph of logistic (sigmoid) function

The logistic regression classifier uses the logistic function of the *weighed* (as well as biased) sum of the input variables to predict the probability of the input set belonging to a class (or category). The probability function is already fixed. The only thing that we can change while learning from different training set is the set of weight parameters (θ) assigned to each feature.

2.3.1. The Algorithm

Let,

$$m = \text{number of samples in training set}$$

$$n = \text{number of features} \geq 1$$

$$x = \text{input feature vector} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad x_0 = 1 \text{ (always)}$$

$$y = \text{class} \in \{1, 0\} \text{ with 1 as primary class}$$

The training set for this machine learning algorithm is the set of m training samples (examples).

$$\text{training set} = \left\{ (x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)}) \right\} \tag{2.3}$$

The weight parameters for the $(n + 1)$ features are given by:

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix}$$

Then the hypothesis function used to predict the output y for input feature set x with parameter θ is given by:

$$h_\theta(x) = \text{sigmoid}\left(\sum_{i=0}^{n} \theta_i x_i \right) = \text{sigmoid}\left(\theta^T x \right) = \frac{1}{1 + e^{-\theta^T x}} \tag{2.4}$$

Now the aim of this machine learning algorithms is to adjust the parameters θ to fit the hypothesis $h_\theta(x)$ with the real output y of training set with minimum cost (error). For that, we need to define the cost function, preferably, a *convex* cost function. There are different types of cost functions. Linear regression, for instance, uses the sum of the squares of the errors as the cost function. But in logistic regression, since the output is not linear (even though the input is), this cost function turns out to be non-convex and there are not efficient algorithms that can minimize a non-convex function. Therefore, we define a logarithmic cost function $J(\theta)$ for logistic regression as follows:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{Cost}\left(h_\theta\left(x^{(i)}\right), y^{(i)}\right) \tag{2.5}$$

where,

$$\text{Cost}(h, y) = \begin{cases} -\log(h) & \text{for } y = 1 \\ -\log(1 - h) & \text{for } y = 0 \end{cases}$$
$$= -y\log(h) - (1 - y)\log(1 - h) \tag{2.6}$$

After we define an appropriate convex cost function $J(\theta)$, the machine learning algorithm basically boils down to finding parameter θ that minimizes $J(\theta)$.

$$\min_{\theta} \quad J(\theta) \tag{2.7}$$

This can be achieved using various optimization algorithms. Some notable ones are Gradient descent, BFGS, Quasi-Newton, L-BFGS, etc. The gradient descent is the simplest one. It is a hill-climbing optimization algorithm that tries finds the local optima from the start point. But since our cost function $J(\theta)$ is convex, there is only one minima and that is the global minima.

To find parameter θ that minimizes the cost function $J(\theta)$, we initial θ with small random values and then iterate the following until convergence:

$$\theta_j = \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j}; \quad j \in 0, 1, 2, \ldots, n; \quad \alpha = \text{learning rate}$$

where,

$$\frac{\partial J(\theta)}{\partial \theta_j} = \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)} \right) x_j^{(i)} \tag{2.8}$$

Note that all θ_j's must be updated simultaneously. This concept is called *batch learning*, contrary to *online learning* where the parameter is updated separately for every training example.

The resulting parameter θ that minimizes the cost function $J(\theta)$ is the parameter of the learning model. Then we can use the hypothesis $h_\theta(x)$ to predict the output y for any input feature set x. The output y will be a value in the range $(0, 1)$. The output can be interpreted as the probability of the given input set belonging to the class 1 (primary class).

Other advanced optimization algorithms such as BFGS, L-BFGS, Quasi-Newton, etc. are more efficient that the basic gradient descent and also has the advantage that we don't have to manually select the learning rate (α). These advanced optimization algorithms will automatically select the appropriate value of α to maximize efficiency.

2.3.2. Other Considerations

Feature Scaling is the process of scaling (or normalizing) all the features to a limit of $[-1, 1]$ or $[0, 1]$. This is required because unscaled features causes some features to get higher priority implicitly and it reduces the accuracy of the learning algorithm. Feature scaling can be done in following ways:

$$x_j^{(i)} = \frac{x_j^{(i)} - \min(x_j)}{\max(x_j) - \min(x_j)} \in [0, 1]$$

or

$$x_j^{(i)} = \frac{x_j^{(i)} - \overline{x_j}}{\max(x_j) - \min(x_j)} \in [-1, 1]$$

or

$$x_j^{(i)} = \frac{x_j^{(i)} - \overline{x_j}}{\sigma_{x_j}} \quad \text{(for normal distribution)}$$

Regularization is the process of scaling down the values of parameter θ to reduce the problem of over-fitting. Over-fitting is the condition when the learning algorithm satisfies the training set very precisely but doesn't satisfy test data (not included in training set). Regularization is done by introducing a regularization parameter λ term in the overall cost function $J(\theta)$.

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{Cost}\left(h_\theta\left(x^{(i)}\right), y^{(i)}\right) + \frac{\lambda}{2m} \sum_{j=0}^{n} \theta_j^2 \tag{2.9}$$

The corresponding first-order derivatives then becomes:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \begin{cases} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_j^{(i)} & \text{for } j = 0 \\ \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_j^{(i)} + \frac{\lambda}{m}\theta_j & \text{for } j \in 1, 2, \ldots, n \end{cases}$$

3. METHODOLOGY

Duplicate bug report retrieval can be viewed as an application of information retrieval (IR) technique to the domain of software engineering, with the objective of improving productivity of software maintenance. In classical retrieval problem, the user gives a query expressing the information he/she is looking for. The IR system would then return a list of documents relevant to the query. For duplicate report retrieval problem, the triager receives a new report and inputs it to the duplicate report retrieval system as a query. The system then returns a list of potential duplicate reports. The list should be sorted in a descending order of relevance to the queried bug report. Our approach adopts recent development on discriminative models for information retrieval to retrieve duplicate bug reports. Adapted from [7], we consider duplicate bug report retrieval as a binary classification problem, that is, given a new report, the retrieval process is to classify all existing reports into two classes: duplicate and non-duplicate. We compute 27 types of textual similarities between reports and use them as features for training and classification purpose. The rest of this section is structured as follows: Section 3.1 discusses about overall framework. Section 3.2 explains how existing bug reports in the repository are organized. Section 3.3 elaborates on how discriminative model is built. Section 3.4 describes how the model is applied for retrieving duplicate bug reports. Finally, Section 3.5 describes how the model is updated when new triaged bug reports arrive.
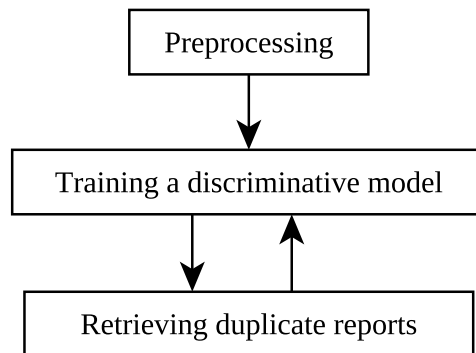
3.1. Overall Framework

Figure 3.1: Overall framework to retrieve duplicate bug reports

Figure 3.1 shows the overall framework of our approach. In general, there are three main steps in the system, preprocessing, training a discriminative model and retrieving duplicate bug reports. The first step, preprocessing, follows a standard natural language processing style—

tokenization, stemming and stop words removal—described in Section 2.2. The second step, training a discriminative model, trains a classifier to answer the question "How likely are two bug reports duplicates of each other?" The third step, retrieving duplicate bug reports, makes use of this classifier to retrieve relevant bug reports from the repository.

3.2.   Report Organization

Buckets	
Master 1	Duplicate 1.1, Duplicate 1.2, . . .
Master 2	Duplicate 2.1, Duplicate 2.2, . . .
Master 3	Duplicate 3.1, Duplicate 3.2, . . .
. . .	

Figure 3.2: Bucket structure

All the reports in the repository are organized into a bucket structure. The bucket structure is a hash-map-like data structure. Each bucket contains a master report as the key and all the duplicates of the master as its value. As explained in Section 2.1, different masters report different defects while a master and its duplicates report the same defect. Therefore, each bucket stands for a distinct defect, while all the reports in a bucket correspond to the same defect. The structure of the bucket is shown diagrammatically in Figure 3.2. New reports will also be added to the structure after they are labeled as duplicate or non-duplicate by triagers. If a new report is a duplicate, it will go to the bucket indexed by its master; otherwise, a new bucket will be created to include the new report and it becomes a master.

3.3.   Training a Discriminative Model

Given a set of bug reports classified into masters and duplicates, we would like to build a discriminative model or a classifier that answers the question: "How likely are two input bug reports duplicate of each other?" This question is essential in our retrieval system. As described in Section 3.4, the answer is a probability describing the likelihood of these two reports being duplicate of each other. When a new report comes, we ask the question for each pair between the new report and all the existing reports in the repository and then retrieve the duplicate reports based on the probability answers. To get the answer we follow a multi-step approach involving example creation, feature extraction, and discriminative model creation.

```
┌─────────────────────────┐
│         Buckets         │
└─────────────────────────┘
             │
             ▼
┌─────────────────────────┐
│    Creating examples    │
└─────────────────────────┘
```
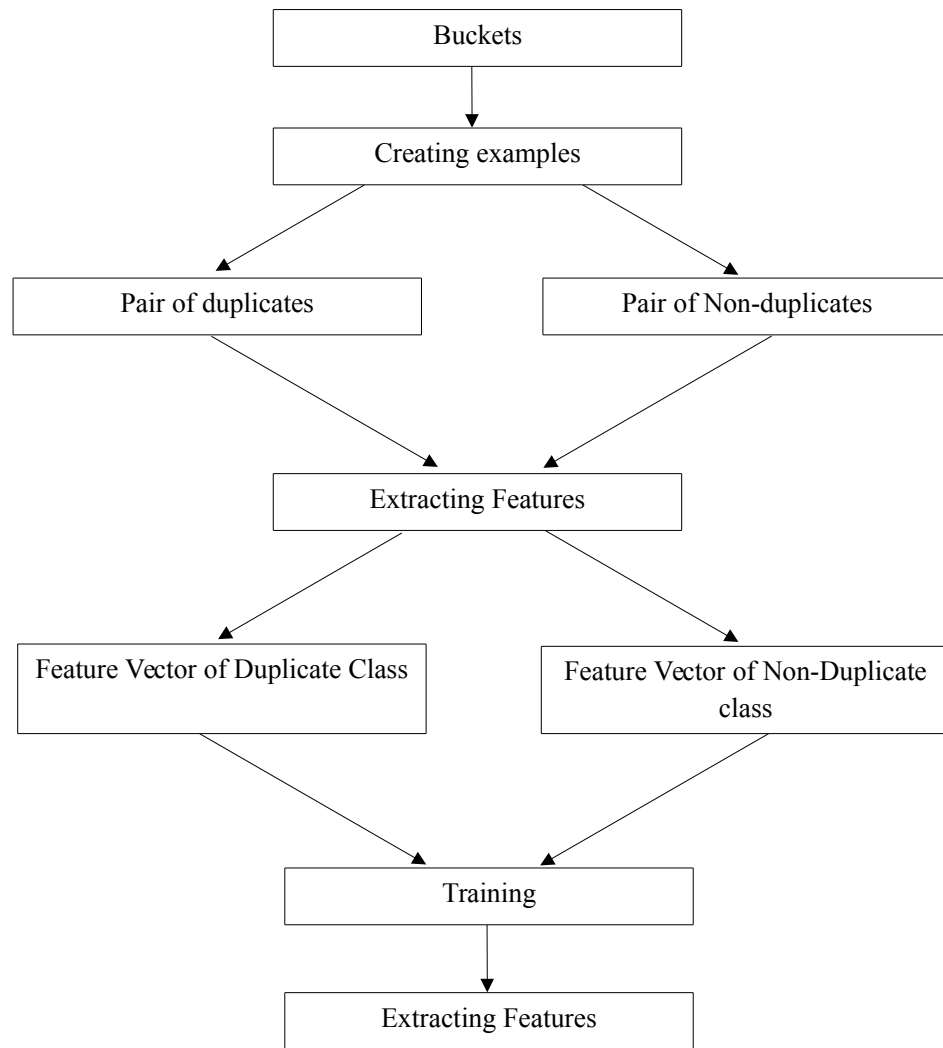
Figure 3.3: Training a discriminative model

The steps are shown in Figure 3.3. Based on the buckets containing masters associated with corresponding duplicates, we extract positive and negative examples. Positive examples correspond to pairs of bug reports that are duplicates of each other. Negative examples correspond to pairs of bug reports that are not duplicates of each other. Next, a feature extraction process is employed to extract features from the pairs of bug reports. These features must be rich enough to be able to discriminate between cases where bug reports are duplicate of one another and cases where they are distinct. These feature vectors corresponding to duplicates and non-duplicates are then input to a learning algorithm to build a suitable discriminative model. The following sub-sections describe each of the steps in more detail.

3.3.1. Creating Examples

To create positive examples, for each bucket, we perform the following:

- Create the pair (*master*, *duplicate*), where duplicate is one of the duplicates in the bucket and master is the original report in the bucket.

- Create the pairs (*duplicate*$_1$, *duplicate*$_2$) where the two duplicates belong to the same bucket.

To create negative examples, one could pair one report from one bucket with another report from the other bucket. The number of negative examples could be much larger than the number of positive examples. As there are issues related to skewed or imbalanced dataset when building classification models, we choose to under-sample the negative examples, thus ensure that we have the same number of positive and negative examples. At the end of the process, we have two sets of examples: one corresponds to examples of pairs of bug reports that are duplicates, and the other corresponds to examples of pairs of bug reports that are non-duplicates.

3.3.2. Feature Engineering and Extraction

At times, limited features make it hard to differentiate between two contrasting datasets: in our case, pairs that are duplicates and pairs that are non-duplicates. Hence a rich enough feature set is needed to make duplicate bug report retrieval more accurate. Since we are extracting features corresponding to a pair of textual reports, various textual similarity measures between the two reports are good feature candidates. In our approach, we employ the following formula as the textual similarity.

$$\text{Sim}(B_1, B_2) = \sum_{w \in B_1 \cap B_2} \text{IDF}(w) \tag{3.1}$$

In (3.1), $\text{sim}(B_1, B_2)$ returns the similarity between two bags of words B_1 and B_2. The similarity is the sum of *IDF* values of all the shared words between B_1 and B_2. The IDF value for each word is computed based on a corpus formed from all the reports in the repository. The rational why the similarity measure does not involve TF is that the measure with only

IDF yields better performance indicated by *Fisher score*. The Fisher score is a vector of parameter derivatives of log likelihood of a probabilistic model.

Generally, each feature in our approach can then be abstracted by the following formula,

$$f(R_1, R_2) = \text{Sim}(\text{Tokens}(R_1), \text{Tokens}(R_2)) \tag{3.2}$$

From (3.2), a feature is actually the similarity between two bags of words from two reports R_1 and R_2. One observation is that a bug report consists of two important fields: title and detail. Title is nothing but a one line summary of the bug report and the detail of the bug report includes different information about operating system, RAM, video memory of the system in which this bug was noticed. Moreover, the detail also includes the actual output and the expected output. So we can get three bags of words from one report, one bag from title, one from detail and one from both (title + detail). To extract a feature from a pair of bug reports, for example, one could compute the similarity between the bag of words from the title of one report and the words from the detail of the other. Alternatively, one could use the similarity between the words from both the title and detail of one report and those from the summary of the other. Other combinations are also possible. Furthermore, we can compute three types of IDF, as the bug repository can form three distinct corpora. One corpus is the collection of all the titles, one corpus is the collection of all the details, and the other is the collection of all the both (title + detail). We denote the three types of IDF computed within the three corpora by IDF_{title}, IDF_{detail}, and IDF_{both} respectively.

Algorithm 3.1 Calculate candidate reports for Q

   Procedure PROPOSE-CANDIDATES
   Input:
      Q: a new report
      R: the bug repository
      N: expected number of candidates
   Output:
      A list of N masters which Q is likely duplicate of
   Body
   1: $candidates \leftarrow$ an empty min-heap of maximum size N
   2: **for all** bucket $B \in \text{BUCKETS}(R)$ **do**
   3:    $similarity \leftarrow \text{PREDICT-BUCKET}(Q, B)$
   4:    $\text{MASTER}(B).similarity \leftarrow similarity$
   5:    Add $\text{MASTER}(B)$ **to** $candidates$
   6: **end for**
   7: Sort $candidates$ in descending order of $similarity$
   8: **return** sorted $candidates$

Algorithm 3.2 Calculate similarity between Q and a bucket.

Procedure PREDICT-BUCKET
Input:
    Q: a new report
    B: a bucket
Output:
    max: the maximum similarity between Q and each report of B
Body
1: $max \leftarrow 0$
2: $tests \leftarrow \{ f_{27}(Q, R) \mid R \in \text{REPORTS}(B) \}$
3: **for all** feature vector $t \in tests$ **do**
4:     $sim \leftarrow \text{LOGISTIC-REGRESSION-PREDICT}(t)$
5:     $max \leftarrow \text{MAX}(max, sim)$
6: **end for**
7: **return** max
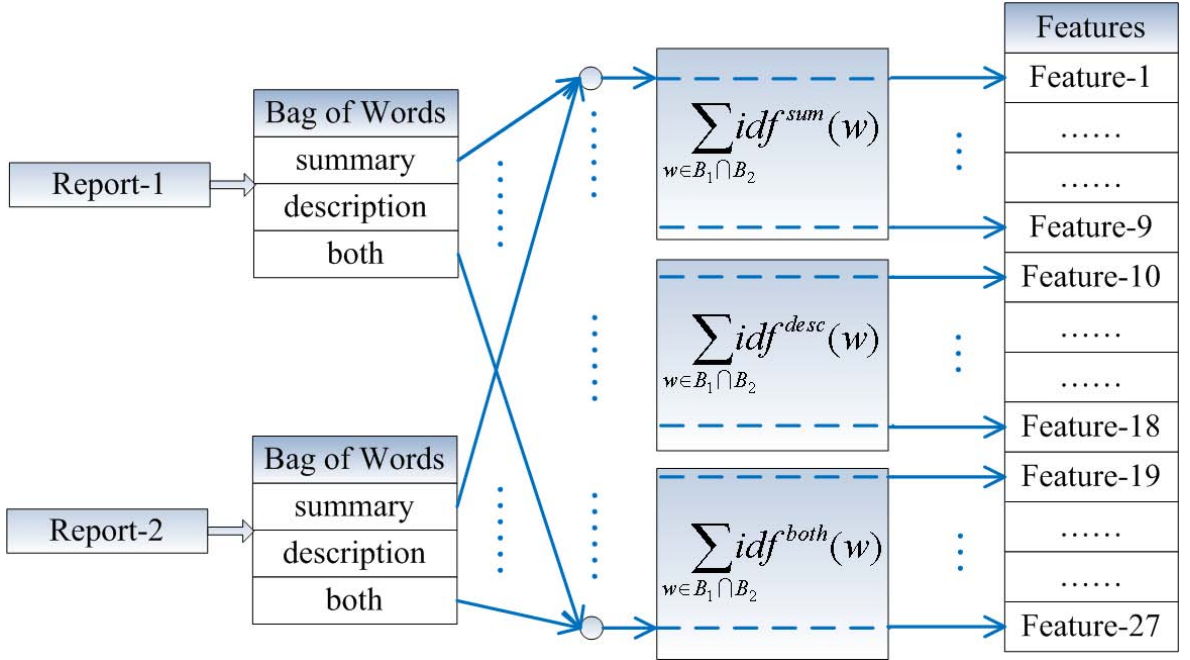
Figure 3.4: Feature extraction: 27 features

The output of function f defined in (3.2) depends on the choice of bag of words for R_1, the choice of bag of words for R_2 and the choice of IDF. Considering each of the combinations as a separate feature, the total number of different features would be $3 \times 3 \times 3 = 27$. Figure 3.4 shows how the 27 features are extracted from a pair of bug reports.

3.3.3. Training Models

After extracting 27 features it is time for training our model. By training we mean adjusting the value of theta parameters as we explained in Section 2.3 We make a training set of $m/2$ duplicate pairs and the rest $m/2$ non-duplicate pairs, resulting in a total of m training examples. The set the output $y = 1$ (interpreted as: the input sample belongs to class 1) for the positive samples and $y = 0$ for the negative samples. Then we'll have a training set of the form:

$$\text{training set} = \begin{bmatrix} x_0^{(0)} & x_1^{(0)} & x_2^{(0)} & \cdots & x_{27}^{(0)} & y^{(0)} \\ x_0^{(1)} & x_1^{(1)} & x_2^{(1)} & \cdots & x_{27}^{(1)} & y^{(1)} \\ x_0^{(2)} & x_1^{(2)} & x_2^{(2)} & \cdots & x_{27}^{(2)} & y^{(2)} \\ \cdots & \cdots & \cdots & \ddots & \vdots & \vdots \\ x_0^{(m)} & x_1^{(m)} & x_2^{(m)} & \cdots & x_{27}^{(m)} & y^{(m)} \end{bmatrix} \quad (3.3)$$

Then we feed the training set to the learning model which fits our hypothesis function to the training set with minimum error

3.4. Applying Model for Duplicate Detection

When a new report Q arrives, we can apply the trained model to retrieve a list of candidate reports of which Q is likely a duplicate. The retrieval details are displayed in Algorithm 3.1 and Algorithm 3.2. Algorithm 3.1 returns a duplicate candidate list for a new report. In general, it iterates over all the buckets in the repository and calculates the similarity between the new report and each bucket. At last, a list of masters whose buckets have the biggest similarity is returned. In Line 2, BUCKETS(R) returns all the buckets in the bug repository, and MASTER(B) in Line 4 is the master report of bucket B. The algorithm makes a call to Algorithm 3.2 which computes the similarity between a report and a bucket of reports. To make the algorithm efficient, we only keep top N candidate buckets in memory while the buckets in the repository are being analyzed. This is achieved by a minimum heap of maximum size N. If the size of candidates is less than N, new bucket will be directly added. When the size equals to N, if the new bucket whose similarity is greater than the minimum similarity in Candidates, it will replace the bucket with the minimum similarity; otherwise, the request of adding the bucket will be ignored by Candidates.

Given a new report Q and a bucket B, Algorithm 3.2 returns the similarity between Q and B. This algorithm first creates candidate duplicate pairs between Q and all the reports in the bucket B (Line 2). Each pair is represented by a vector of features, which is calculated by the function f_{27}. The trained discriminative model is used to predict the probability for each candidate pair. Finally, the maximum probability between Q and the bug reports in B is returned as the similarity between Q and B. Reports(B) denotes all the reports in B. The operation $f_{27}(Q, R)$ returns a vector of the 27 similarity features from the pair of bug reports Q and R, including 27 features based on single words and 27 features. The procedure Logistic-Regression-Predict() in Line 4 is the invocation to the discriminative model and it returns a probability in which the pair of reports the feature vector t corresponds to are duplicates of each other.

3.5. Model Evolution

As time passes, new bug reports will come and be triaged. This new information could be used as new training data to update the model. Users could perform this process periodically or every time after a new bug report has been triaged. In general, such newly created training data should be useful. However, we find that such information is not always beneficial to us for the following reason: our retrieval model is based on lexical similarity rather than semantic similarity of text. In another word, if two bug reports refer to the same defect but use different lexical representations, i.e., words, then our retrieval model does not give a high similarity measure to this pair of bug reports. Therefore, including a pair of duplicate bug reports that are not lexically similar in the training data would actually bring noise to our trained classifier. We therefore consider the following two kinds of update:

3.5.1. Light update

If our retrieval engine fails to retrieve the right master for a new report before the triager marks the report as a duplicate, we perform this update. When the failure happens, the new bug report could syntactically be very far from the master. For this case, we only perform a light update to the model by updating the IDF scores of the training examples and re-training the model.

3.5.2. Regular update

We perform this update if our retrieval engine is able to retrieve the right master for the new duplicate bug report. When this happens, the new bug report is used to update the IDF *and* to create new training examples. An updated discriminative model is then trained based on the new IDF and training examples.

3.6. Tools Used for Development

For the development of the website we first considered using PHP and CodeIgniter web framework. But due to lack of library support for CodeIgniter we had to shift to some other language. So we learned python programming language as it has an easy yet powerful web framework named Django. Moreover, Python is very popular programming language for data science. It has many helpful modules for information retrieval and learning model. Then we built the whole project based on them. Below is a brief description about Python and Django.

3.6.1. Python Programming Language

Python is a widely used high-level, general-purpose, interpreted, dynamic programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than possible in languages such as C++ or Java. The language provides constructs intended to enable clear programs on both a small and large scale. Python supports multiple programming paradigms, including object oriented, imperative and functional programming or procedural styles. It features a dynamic type system and automatic memory management and has a large and comprehensive standard library.

Python interpreters are available for many operating systems, allowing Python code to run on a wide variety of systems. Using third-party tools, such as Py2exe or Pyinstaller, Python code can be packaged into stand-alone executable programs for some of the most popular operating systems, so Python-based software can be distributed to, and used on, those environments with no need to install a Python interpreter. For more details refer to python official website at `https://www.python.org`.

The whole logic of our application is developed in Python.

3.6.2.   Django Web Development Framework

Bug tracking systems are usually online web applications where developers and users can submit or view submitted bug reports. Therefore, we needed a framework to develop websites quickly and properly. That's where Django web development framework comes.

Django is a free and open-source web framework, written in Python, "*for perfectionists with deadline*". It follows the model-view-controller (MVC) architectural pattern. It is maintained by the Django Software Foundation (DSF), an independent nonprofit organization. Django's primary goal is to ease the creation of complex, database-driven websites. Django emphasizes re-usability and "pluggability" of components, rapid development, and the principle of don't repeat yourself. Python is used throughout, even for settings files and data models. Django also provides an optional administrative create, read, update and delete interface that is generated dynamically through introspection and configured via admin models. Some well-known sites that use Django include Pinterest, Instagram, Mozilla, The Washington Times, Disqus, the Public Broadcasting Service, BitBucket, and Nextdoor.

Despite having its own nomenclature, such as naming the callable objects generating the HTTP responses "views", the core Django framework can be seen as an MVC architecture. It consists of an object-relational mapper (ORM) that mediates between data models (defined as Python classes) and a relational database ("Model"), a system for processing HTTP requests with a web templating system ("View"), and a regular-expression-based URL dispatcher ("Controller").

For developing a Django project, no special tools are necessary, since the source code can be edited with any conventional text editor. Nevertheless, editors specialized on computer programming can help increase the productivity of development, e.g. with features such as syntax highlighting. Since Django is written in Python, text editors which are aware of Python syntax are beneficial in this regard. Integrated development environments (IDE) add further functionality, such as debugging, refactoring, unit testing, etc. As with plain editors, IDEs with support for Python can be beneficial. Some IDEs that are specialized on Python additionally have integrated support for Django projects, so that using such an IDE when developing a Django project can help further increase productivity. For more details refer to Django official website at `https://www.djangoproject.com`.

4. CASE STUDIES

While developing this project we carried out the experiment on different sample bug repositories in order to test the efficiency of our approach and make necessary modification to it. In our experiment we use the sample bug repositories of Linux and Firefox. A brief overview of the experiment and its result in mentioned in the following subsections.

4.1. Experiment on Firefox Sample Bug Repositories

We carried out an experiment where we tested our system on sample bug repository of Firefox. There were total 33,000 bug reports in the sample repository. It contains fields such as project name, bug id, bug summary, status, etc. We calculated the textual similarity between two bugs just using the concept of IDF. We just used the project summary for the experiment. It took 96 seconds in order to compare a bug with every other bugs in the repository. In order to improve the calculation we carried out the following technique:

1. Tokenize, stem, remove stop-words for summary of the bug report in advance.
2. Calculate the frequency of every token obtained in Step 1.
3. Store the frequency of every token along with the bug id in the database

What this algorithm does is that it removes the overhead of preprocessing each and every bugs in the database time and again. All we need now is to get the value of term frequency from the database and then we can easily calculate the IDF and bugs similarity. This idea highly improved the performance of our system and it took just 32 seconds to calculate the similarity with the use of this algorithm. That's pretty satisfactory result. We hadn't implemented learning algorithm till now. The similarity based on text is 35–50% efficient. In order to improve efficiency, we applied machine learning.

4.2. Experiment on Linux Sample Bug Repositories

We carried out another experiment where we tested our system on sample bug repository of Linux. There were total 1000 bug reports in the sample repository. It also contains the similar fields to that of Firefox repository. Here also we calculated the textual similarity between bugs. We entered a search query and calculated similarity with every bug in the repository. The results were satisfactory in this repository as well. It took about 5 seconds in order to display the list of duplicate bugs for the search query entered by the user whereas once the above algorithm was used, once the calculation time was reduced to few milliseconds. The results of the experiment were enough for us to apply the above algorithm in our system.

5. SYSTEM DEVELOPMENT

5.1. System Modeling

Every complex software needs some abstract designing from a high level to understand the software better. Modeling is a way to create abstract representations of the system to better understand it. As software engineers, we know how important it is to design system models before writing code. System models can be made by using many types of visual languages such as UML (Unified Modeling Language).

Since our application is of moderate complexity, we made the following models to design our understand the requirements and design the system. The use case diagram is one of the UML diagrams. It was made to understand how our system could be used by different external actors such as customers and developers. Another model we made was of the database. Designing a sophisticated datbase schema without proper modeling cause lots of trouble. An ERD (Entity Relationship Diagram) is a high-level model of the actual database schema. We made it to understand the database structure and the relations we need to define to store the data efficiently.
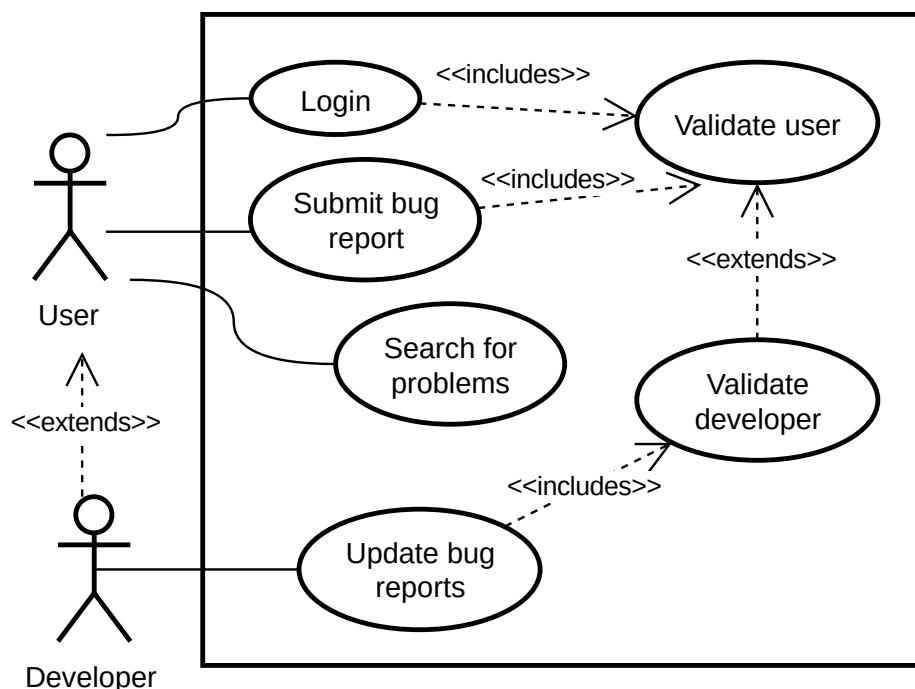
Figure 5.1: Use case diagram
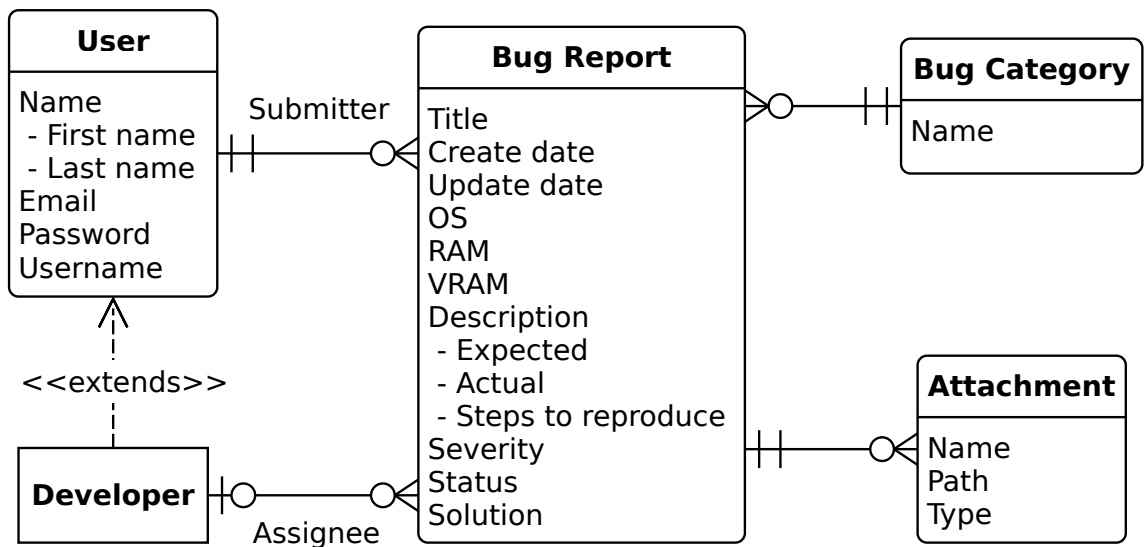
Figure 5.2: Entity relationship diagram

5.2. Project Schedule

This project started at the end of May, 2016 and ended in the middle of August, 2016. Therefore, it was completed in about 3 months' time. The timeline of how we progressed through these 3 months' time is given in the Gantt chart below. We had allocated enough time for initial analysis and system design. But as we know that most of software development time is spent in testing and debugging, we allocated more than half of the project schedule to coding, testing and debugging. Finally we documented everything and completed the project in time.
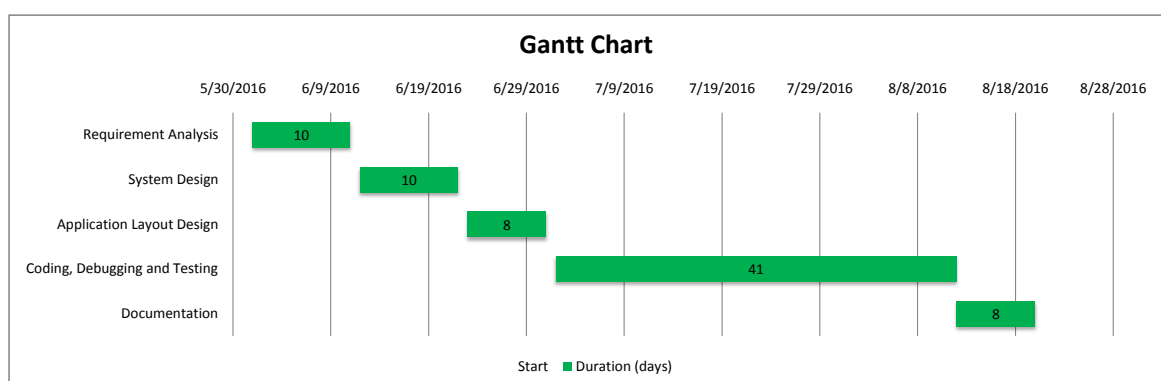
Figure 5.3: Gantt chart of our project schedule

6. FINAL PRODUCT
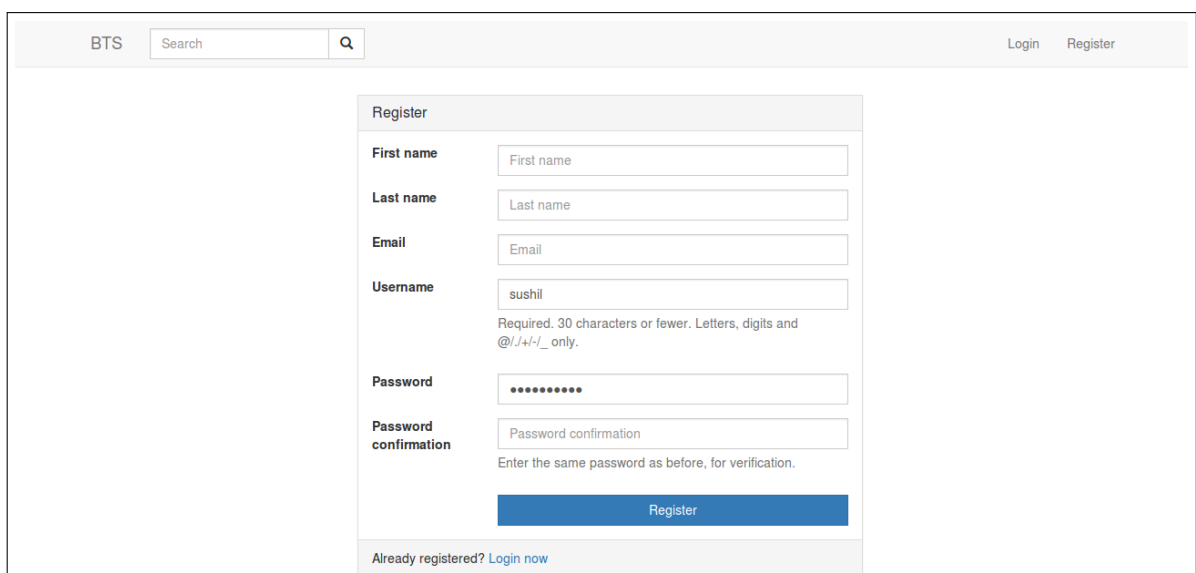
After all the fore-mentioned research and study, as the final product of the this project, we have developed a sophisticated bug tracking system, an online bug management web application where users can submit and search for bug reports and developers can track and manage them.

The main features of our product are described below.

6.1. User Registration and Login

As with any web application, for anyone to interact with an web application, he/she must be registered in that website. Although view bug reports doesn't require user validation, many of the features of our web application requires user credentials such as email and permissions. So user authentication is an indispensable part of our web application.

For that, our application provides a very simple user registration and login facilities. Figure 6.1 shows the registration form, whereas Figure 6.2 shows the login form.

Figure 6.1: User registration form

Figure 6.2: User login form

6.2. Types of User

Our application has three types of users: customer, developer, and administrator. They all login with the same login form, but they differ in their permissions and privileges.

6.2.1. Customer

A customer is a basic user. They can register and activate their account without the administrator's verification. They can submit bug reports but cannot update them.

6.2.2. Developer

They have all the privileges of the customer plus some more. When anyone submits a bug reports, they can assign the bug to one of the developers. The bug is said to been *assigned* to that developer. Then that developer is known as the *assignee* of that bug. Assignee of a bug has the responsibility to update the bug's status and fix the bug if possible.

Unlike customers, one cannot register as a developer themselves. The administrator has to mark the user as a developer. Only then he can have the privilege of being an assignee of bug reports.

6.2.3. Administrator

The administrator is the all-powerful controller of the system. Ideally, there is only one administrator of a system. The administrator is in charge of marking users as developers and controlling the content on the website and the web server from a lower level.

6.3. Search Bug Report by Keywords

All our data for the website are store in database in well-structured manner so querying information from database is very efficient. Out website has a search input field in the top navigation bar in every page. When a user enters some keywords and submits the search query form, the backend program compares the keywords to all the bug reports in the database and list them in descending order of relevance.

This is a very useful feature that can help customers solve some problems they may be facing because a similar problem could have been solved earlier. If not, one can obviously submit a new bug report.

6.4. Highly Filterable Bug List

The default (home) page of our website shows the list of latest submitted bug reports with their various fields such as submitter, severity, category, status, etc. as shown in Figure 6.3. The list can be sorted (both ascending and descending order) by every field and can also be filtered by many fields as well with the help of the filter form located just above the bugs list.

The filtered list is shown in Figure 6.4.

6.5. Bug Report Submission

As mentioned earlier, every authenticated user of the application can submit new bug reports. But if one if found submitting too much fraud reports, then the administrator may ban him from accessing the system. The bug submission form is sown in Figure 6.5.

Figure 6.3: Bugs list in home page

Figure 6.4: Filtered list

6.6. Assignee Notification

The developers don't have to log into website to check for new bug reports. Whenever someone submits a new bug report, they can directly choose the assignee and the assignee will be immediately notified via email about the new bug report. The assignee can then fix the bug or reassign to someone more appropriate.

Figure 6.5: Bug report submission form

6.7. Bug Status Update

Only assignees can modify bug reports, but not all fields. Assignee of a bug can modify the bug report's status fields such as severity, status, solution, master report, or the assignee itself. The bug status update form is shown in Figure 6.6.

Figure 6.6: Bug report update form

6.8.   Duplicate Bug Report Detection

This is the prime feature of our application. The homepage displays a list of latest bug reports. When they click on a bug report, it will open up the detail page of the bug report, which looks as in Figure 6.7.
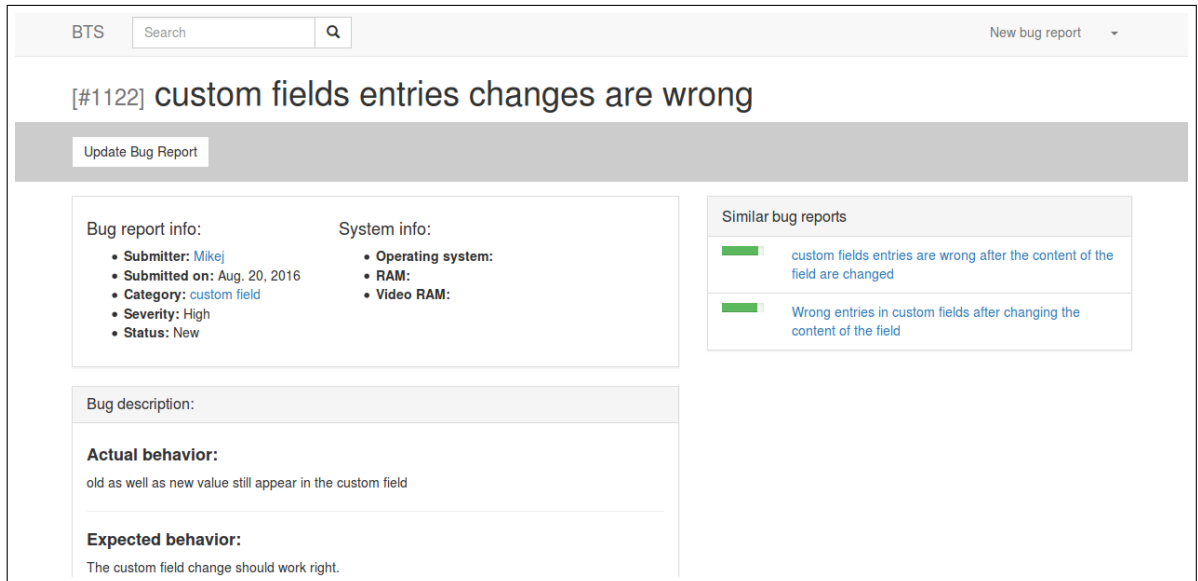
Figure 6.7: Bug report detail page

The page contains all the information about the specific bug report. In a sidebar, it displays a list of similar bug reports based on our machine learning algorithm. The list shows the most similar bug report at the top. This is to help the developers identify duplicate bug reports quickly to help them fix the bug faster and manage the bug repository better.

Our concept is that duplicate bug reports aren't a *bad* thing. Sometimes, one report doesn't describe a bug well. Multiple reports about the same bug will help developers fine tune the real cause of the bug and help them fix them faster.

7. LITERATURE REVIEW

One of the pioneer studies on duplicate bug report detection is by Runeson et al.[4] Their approach first cleaned the textual bug reports via natural language processing techniques—tokenization, stemming and stop-word removal. The remaining words were then modeled as a vector space, where each axis corresponded to a unique word. Each report was represented by a vector in the space. The value of each element in the vector was computed by the following formula on the TF value of the corresponding word.

$$\text{weight}(w) = 1 + \log_2(\text{TF}(w)) \tag{7.1}$$

After these vectors were formed, they used three measures—cosine, dice and jaccard—to calculate the distance between two vectors as the similarity of the two corresponding reports. Given a bug report under investigation, their system would return top-k similar bug reports based on the similarities between the new report and the existing reports in the repository. A case study was performed on defect reports at Sony Ericsson Mobile Communications, which showed that the tool was able to identify 40% of duplicate bug reports. It also showed that cosine outperformed the other two similarity measures.

In [5], Wang et al. extended the work by Runeson et al. in two dimensions. First they considered not only TF, but IDF. Hence, in their work, the value of each element in a vector corresponded to the following formula,

$$\text{weight}(w) = \text{TF}(w) \times \text{IDF}(w) \tag{7.2}$$

Second, they considered execution information to detect duplicates. A case study based on cosine similarity measure on a small subset of bug reports from Firefox showed that their appraoch could detect 67–93% of duplicate bug reports by utilizing both natural language information and execution traces.

In [3], Jalbert and Weimer extended the work by Runeson et al. in two dimensions. First, they proposed a new term-weighting scheme for the vector representation,

$$\text{weight}(w) = 3 + 2\log_2(\text{TF}(w)) \tag{7.3}$$

The cosine similarity was adopted to extract top-k similar 2 reports. Aside from the above, they also adopted clustering and classification techniques to filter duplicates. Similar to the above three studies, we address the problem of retrieving similar bug reports from repositories for duplicate bug report identification. Similar to the work in [3, 4], we only consider natural language information which is widely available. The execution information considered in [5] is often not available and hard to collect, especially for binary programs. For example, in the OpenOffice, Firefox and Eclipse datasets used in our experiment, the percentages of reports having execution information are indeed low (0.82%, 0.53%, and 12% respectively). Also, for large bug repositories, creation of execution information for legacy reports can be time consuming. Compared to the three approaches, there are generally three differences. First, to retrieve top-k similar reports, they used similarity measure to compute distances between reports while we adopt an approach based on discriminative model.

We trained a discriminative model via logistic regression to classify whether two bug reports are duplicates of one other with a probability. Based on this probability score, we retrieve and rank candidate duplicate bug reports.

Besides the effort on duplicate bug report detection, there has also been effort on bug report mining. Anvik et al. [8] and Cubranic and Murphy [6] and Lucca [9] all proposed semi-automatic techniques to categorize bug reports. Based on categories of bug reports, their approaches helped assign bug reports to suitable developers. Menzies and Marcus also suggested a classification based approach to predicting the severity of bug reports.[10] While the above works mentioned the need of duplicate bug detection or some of their techniques may benefit duplicate bug detection, none of them worked directly on the duplicate bug detection problem.

There have been several statistical studies and surveys of existing bug repositories. Anvik et al. reported a statistical study on open bug repositories with some interesting results such as the proportion of different resolutions and the number of bug reports that a single reporter submitted.[2] Sandusky et al. studied the relationships between bug reports and reported some statistic results on duplicate bugs in open bug repositories.[10] In general, none of these work proposed any approaches to duplicate-bug-report detection, but some of the work pointed out the motivation and effect of detecting duplicate bug reports.

8. DISCUSSIONS

As the major part of this project has already been described in many earlier sections, we don't think we need to elaborate about them again. Instead, in this section we discuss issues related to runtime overhead followed by the rationale behind the choice of the 27 similarity scores as features rather than other types of similarity scores.

8.1. Runtime Overhead

The major overhead is due to the fact that we consider different similarity features between reports. The runtime overhead is higher at the later part of the experiment as the number of bug report pairs in the training set is larger. Consequently, it will take more time to build a discriminative model. However, a higher runtime overhead does not mean that this approach is not practical. In fact, it is acceptable for real world bug triaging process for two reasons. First, we have experimented with real world datasets. The dataset from Firefox spans contains more than 33,000 reports in total. Second, new bug reports do not come every minute. Therefore, our system still has enough time to retrain the model before processing a new bug report. An approach that we have taken in order to minimize the overhead was explained earlier in the Section 4.1.

8.2. Feature Selection

Feature selection is the process to identify a set of features which can bring the best classification performance. A commonly used metric for feature selection is Fisher score. In our approach, we use 27 IDF-based formulas to calculate similarity features between two bug reports. It may come to one's head, why TF or TF×IDF measures are not used in the formulas. The reason to choose an IDF-only solution is that this setting yields the best performance during our empirical validation. To further demonstrate that IDF is a good measure, we replace the IDF measure in the 27 features with TF and TF×IDF measure respectively. We then calculate the Fisher score of each of the 27 features using IDF, TF ,and TF×IDF. The results shows that IDF-based formulas outperform TF-based and (TF×IDF)-based formulas in terms of Fisher score. This supports our decision in choosing the IDF-based formulas as feature scores.

9. CONCLUSION AND FUTURE WORK

For this project we considered many approaches given by several researchers in the field of duplicate bug detection and used the discriminative model approach with a slight modification of our own. The discriminative model answers the question "Are two bug reports duplicates of each other?". The model would report a score on the probability of A and B being duplicates. This score is then used to retrieve similar bug reports from a bug report repository for user inspection. We have investigated the utility of our approach as well as previous approaches on 2 sizable bug repositories from 2 large open-source applications including Firefox, and Linux and we have found a satisfactory result.

As a future work, we plan to investigate the utility of bi-grams in discriminative models, and include the execution information while submitting a bug report for the potential improvement in accuracy. The other interesting direction is adopting pattern-based classification [4, 11] to extract richer feature set that enables better discrimination and detection of duplicate bug reports.

REFERENCES

[1] G. Tassey. The economic impacts of inadequate infrastructure for software testing. In *National Institute of Standards and Technology. Planning Report 02-3.2002*, 2002.

[2] J. Anvik, L. Hiew, and G. Murphy. Coping with an open bug repository. In *Eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 35–39, 2005.

[3] N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2008.

[4] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the International Conference on Software Engineering*, 2007.

[5] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the International Conference on Software Engineering*, 2008.

[6] D. Cubranic and G. C. Murphy. Automatic bug triage using text categorization. In *Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, pages 92–97, 2004.

[7] R. Nallapati. Discriminative models for information retrieval. In *SIGIR '04: Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, 2004.

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

[9] L. G. An approach to classify software maintenance requests. In *ICSM '02: Proceedings of the International Conference on Software Maintenance*, page 93, 2002.

[10] T. Menzies and A. Marcus. Automated severity assessment of software defect reports. In *ICSM08: Proceedings of IEEE International Conference on Software Maintenance*, pages 346–355, 2008.

[11] Jason Brownlee. Logistic Regression for Machine Learning [Online tutorial]. `http://machinelearningmastery.com/logistic-regression-for-machine-learning`, 2016. Retrieved August 1, 2016.

[12] Andrew Ng. Machine Learning [Video lectures]. `https://www.coursera.org/learn/machine-learning`. Retrieved July, 2016.

[13] Logistic Regression. (n.d.). In *Wikipedia*. `https://en.wikipedia.org/wiki/Logistic_regression`. Retrieved July 23, 2016.

[14] R. J. Sandusky, L. Gasser, R. J. S, U. L. Gasser, and G. Ripoche. Bug report networks: Varieties, strategies, and impacts in a foss development community. In *International Workshop on Mining Software Repositories*, 2004.