A Market-Based Bug Allocation Mechanism Using Predictive Bug Lifetimes

Hadi Hosseini
School of Computer Science
University of Waterloo
Waterloo, ON, Canada
Email: h5hossei@uwaterloo.ca

Raymond Nguyen
School of Computer Science
University of Waterloo
Waterloo, ON, Canada
Email: r9nguyen@uwaterloo.ca

Michael W. Godfrey
School of Computer Science
University of Waterloo
Waterloo, ON, Canada
Email: migod@uwaterloo.ca

***Abstract*—Bug assignment in large software projects is typically a time-consuming and tedious task; effective assignment requires that bug triagers hold significant contextual information about both the reported bugs and the pool of available developers. In this paper, we propose an auction-based multiagent mechanism for assigning bugs to developers that is intended to minimize backlogs and overall bug lifetime. In this approach, developers and triagers are both modeled as intelligent software agents working on behalf of individuals in a multiagent environment. Upon receiving a bug report, triager agents auction off the bug and collect the requests. Developer agents compute their bids as a function of the developer's profile, preferences, current schedule of assigned bugs, and estimated time-to-fix of the bug. This value is then sent to the triager agent for the final decision. We use the Eclipse and Firefox bug repositories to validate our approach; our studies suggest that the proposed auction-based multiagent mechanism can improve the bug assignment process compared to currently practised methods. In particular, we found a 16% improvement in the number of fixed bugs compared to the historic data, based on a sample size of 213,000 bug reports over a period of 6 years.**

***Index Terms*—Multiagent system, market mechanism, bug lifetime, bug repositories**

I. Introduction and Motivation

Estimating the time and effort it takes to release a product is a key task for almost every software project. Post-release modification is inevitable, as all software systems have bugs, and eventually need new features and other improvements also. Maintenance is expensive; the planning and allocation of resources can be difficult if we are unable to estimate how long it takes to fix these bugs [1]. With good estimates, project managers can effectively schedule the resources for upcoming releases and development phases.

Most software projects, both industrial and open source, use an issue tracking system such as Bugzilla to help manage the maintenance process. These systems permit both end-users and developers to enter contextual information about observed failures, and to document the historical progress towards their resolution [2]. Once a bug has been reported, trusted team members must perform bug *triage*: they analyze the report, evaluate its merit, assess its seriousness and urgency, and decide if the bug should be assigned to a developer for fixing. The process of assigning bugs to suitable developers is done manually by the triagers; it is time consuming and tedious, and requires that the triager hold significant contextual information about the reported bugs, developers, priorities, preferences, and dependencies between bug reports (i.e., duplicates, complements, etc.) [3] [4]. This assignment is strongly influenced by the discretion and expertise of the triager; due to the various human factors involved, it is also prone to being subjective and sub-optimal. The triager should take the limitations and preferences of each developer into account, avoiding overloaded backlogs for one developer and increased idle times for others. However, it is unrealistic to require that triagers have perfect knowledge of every possible developer's expertise, experience, and availability. Mistakes in bug assignment can potentially delay the resolution of a bug report, and as a result, increase the overall bug fixing time.

Compounding the problem, developers may not accurately report their experience, domain knowledge, availability, etc. they may over- or under-sell their abilities according to their egos and commitment level to the project [5]. On the other hand, we don't want to bug developers too often; it would be best if their stated preferences could speak for them when bug assignment is being considered; the cost of bothering a user must be weighed against the expected utility of the result [6]. Developers strive to achieve "flow", a state of deep concentration, an experience of full immersion, of harmony between one's actions, skills, and goals [7]. It takes time and effort to get into flow, and it is a relatively fragile state, yet achieving it is extremely important for successful creative activities. Consequently, if a software agent is able to act effectively on behalf of a developer in representing their bug fixing preferences, this would likely be seen as a benefit, and that it is extremely important for successful creative activities.

In this paper, we consider the following research questions: Can an automated multiagent approach improve the quality of bug assignment in large software projects? How can predicted fix time improve the intelligent bug allocation process? Can bug allocation be improved through intelligent multiagent systems?

A. Framing the Idea

Automating the process of bug assignment can decrease the load on the triager and system as well as providing a more suitable allocation by 1) preventing huge backlogs,

2) minimizing the overall bug lifetime by finding the most appropriate allocation, and 3) increasing satisfaction by not overloading developers, as a side effect of intelligent bug assignment.

Our proposed approach works as follows: When a bug is reported to the tracking system, a preliminary analysis extracts basic information such as type, priority, severity, etc. using simple textual analysis. Having extracted the bug's basic information, the system then automatically assigns the bug to a triager based on the bug category. Here, bug triagers are software agents that are responsible for collecting bugs, holding auctions, and assigning the bug to the auction winner. Each developer is also represented by a software agent, which is responsible for placing bids on bugs on their behalf. The developer agents are embedded into the bug repository platform, and are responsible for proactively monitoring and gathering information about the developers' experience, expertise, and preferences based on their historic interactions. This information is combined with details of the open bugs in their current work queue to create their profile, which is continually updated as new actions are performed by the developer. Developer agents reveal their willingness-to-pay values on behalf of the developers using an internal pricing mechanism. This internal function can be developed to include simple or complex attributes of the developers. Every time a triager agent announces a new bug, the developer agent calculates the expected utility of the bug to place a bid over the new bug report. After collecting all bug requests, the triager makes the final decision and assigns the bug report to the winner developer.

Figure 1 summarizes our proposed approach. Incoming bugs are reported and stored in a bug tracking system. Once a record of the bug has been made, it is passed to the bug lifetime prediction engine which takes the stated characteristics of the bug such as severity, platform, priority, etc. and predicts how long it will take to fix the bug using data mining on the bug fixing history of the project. The estimated lifetime is used as part of our function for determining the "price" of a bug. The bugs are then split off based on their category and given to an auctioneer. For example, one auctioneer handles bugs in the UI category, another in the database category, and so on. Now developer agents are able to bid on the bugs they wish to fix. The bug is assigned to the developer with the highest bid at the time, which is based on aspects of the developer such as experience and current bug queue.

This paper is organized as follows: In Section II, we discuss recent research related to mining bug repositories and bug assignment methods. Section III describes our model using machine learning to predict bug lifetime. Section IV proposes an intelligent model for improving bug allocation using a market-based mechanism. In Section V, we validate our approach through empirical study of the Eclipse and Firefox projects. We consider threats to validity of our model in Section VI. Finally, we summarize our work, and discuss future directions in Section VII.

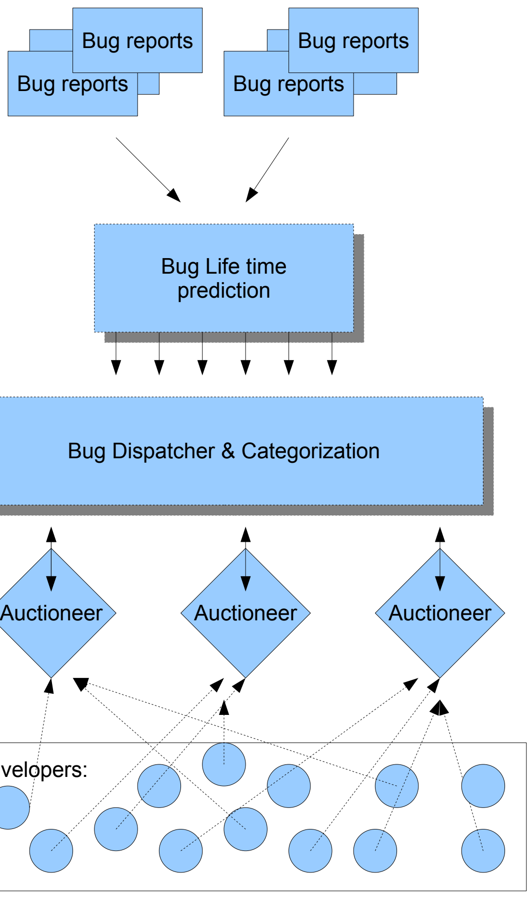


Fig. 1: Overview of our bug assignment approach

II. Related Work

A lot of of the research in the mining software repositories field has concerned bug prediction. Hooimeijer and Weimer [8] looked at how much time was required to triage a bug in the Mozilla Firefox project. Linear regression was used on the bug report data, and it was found to have performed significantly better than random chance in terms of precision and recall. Bettenburg et al. [9] investigated what makes a good bug report by sending out a survey to see the information mismatch between what developers require and what information users supply in a bug report. They developed a tool called CUEZILLA which measures the quality of new bug reports. It was found that well-written bug reports are likely to get more attention than poorly written ones. The first work that used data mining techniques on bug reports to directly predict bug lifetimes was done by Panjer [10], who predicted the life of a bug in Eclipse from confirmation to resolution into a discretized log scaled lifetime class. He used a number of different algorithms, including 0-R, 1R, decision trees, Naive Bayes, and logistic regression. Overall, logistic regression performed best with an accuracy of 34.9%.

Bougie et al. [11] reproduced Panjer's experiment using

FreeBSD instead of Eclipse as the target system. The same algorithms were used but in this instance, bug lifetimes could only be predicted with only 19.49% accuracy. This shows how the predictive accuracy can vary depending on the software project. Giger et al. [12] extended Panjer's study by comparing the predictive power of newly reported bugs to see if they would improve with post submission data within 1 to 30 days after. Classification is done only using decision tree analysis to classify bug fix time as either fast or slow. They found that between 60-70% of bugs could be correctly classified and that post-submission data improves the model by 5-10%. To develop an automated bug assignment system, there has been some work in finding the most useful bug assignment using the information extracted from bug repositories. Anvik et al. [13][14] suggested a semi-automated approach that recommends a set of suitable developers to whom the bug report can be assigned using supervised machine learning algorithms, clustering algorithms, and expertise networks [15]. The authors argued that the process of bug triage cannot be fully automated, since this process requires some sort of human interaction to include the required contextual knowledge for decision making.

Cubranic and Murphy [3] investigated the connection between bug reports and program features by applying supervised machine learning using a Naive Bayes classifier to automatically assign reports to the developers within Eclipse. By using text categorization, they were able to correctly predict 30% of 15,859 bug report assignments from large open-source projects. Baysal et al. [4] reported a theoretic framework for assigning bug fixing tasks to developers based on developers' level of expertise. The system recommends a list of suitable developers upon arrival of a bug report. They used a vector space model to extract developers' expertise from the history of previously fixed bugs by mining the bug repository. At the same time developers would provide their preferences while fixing the bugs. Our approach is novel because it allows the developers to request for suitable bugs from the bug triagers, letting the developers make decisions based on their preferences, expertise, and such. This approach prevents huge backlogs and minimizes the overall bug lifetime by finding the nearly optimal allocation.

III. Predicting Bug Lifetime

This section provides the details of our method in predicting bug lifetime.

A. Bug Repositories: Bugzilla

The data used to validate our model comes from a web-based bug tracker called Bugzilla. This tool was first released in 1998 as a piece of open source software to be used by developers in the Mozilla project, but since then has been adopted by many other organizations. At the core of Bugzilla is a screen that displays information about a particular bug. Each bug will have a number of fields that are used to describe the bug to help developers get a better understanding of the problem. There can be up to 19 different fields, with a number

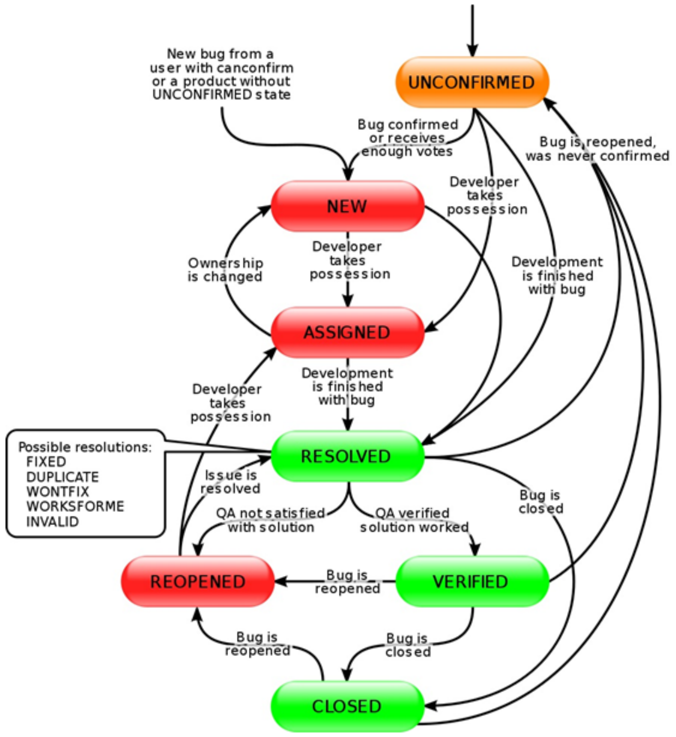

New bug from a user with canconfirm or a product without UNCONFIRMED state
UNCONFIRMED
Bug confirmed or receives enough votes
Bug is reopened, was never confirmed
Developer takes possession
NEW
Ownership is changed
Developer takes possession
Development is finished with bug
ASSIGNED
Developer takes possession
Development is finished with bug
Possible resolutions:
FIXED
DUPLICATE
WONTFIX
WORKSFORME
INVALID
RESOLVED
Issue is resolved
QA not satisfied with solution
QA verified solution worked
Bug is closed
REOPENED
Bug is reopened
VERIFIED
Bug is reopened
Bug is closed
CLOSED

Fig. 2: Bugzilla Bug Life Cycle [16]

of them being optional. For the purpose of predicting the length of time it takes for a bug to go from reported state to fixed state, certain fields were ignored. This is due to the fact that some information would not be available at the time a bug is reported (i.e., votes, # of comments, # of attachments). Once a bug is reported, it has a life cycle which is summarized in Figure 2.

When a bug is first reported, it is marked as UNCONFIRMED until triagers can confirm its existence, validity, and uniqueness. Once this is done or if the bug is submitted from a trusted user, it will now be classified as NEW where the bug will be triaged and assigned with a severity, priority, product, and a developer. A bug in this state is classified as ASSIGNED [10]. When a bug is RESOLVED, VERIFIED, or CLOSED, it can have a number of resolutions. The most common outcome is that the bug is FIXED; however, there are also the cases where a bug can be resolved as DUPLICATE, WONTFIX, WORKSFORME, or INVALID. Our dataset from the Eclipse Bugzilla database contains 213,000 records for bugs reported from 2001 to 2007 and 366,112 bugs from Firefox dated between 1997-2007.

B. Data Preparation and Pre-Processing

The data we used for the project was obtained from the MSR mining challenge 2008 website [17]. The file downloaded was the Eclipse Bugzilla export in an XML format containing bugs 1-213,000. This dataset is 3.2GB in size, split into 2130 files and was retrieved from Bugzilla repository on December 19, 2007 by Thomas Zimmermann. We parsed these large XML files and created a single CSV file in order be able to do more manipulations. We have developed a custom parsing program that converts XML files into CSV files, removing redundancies in XML files. For the purposes of data mining, the count of these optional or multiple element fields should be sufficient in helping to construct a model. For detailed steps of our data pre-processing approach please refer to Appendix A.

Not all the files were well-formed XML documents, and thus our tool encountered some errors within an element. In these cases, the bug was ignored and not copied over to the CSV file. There were 999 such errors encountered throughout the 213,000 $\langle bug \rangle$ elements. It also appeared that the first 4920 elements contained the same creation date along with some blank dates. This portion of the data was removed since it was most likely an issue of importing old bugs to the new Bugzilla database.

Having properly refined the data set, the final collection contains 207,080 bugs with the following fields considered: bug id, creation date, last updated date, classification id, product, component, version, rep platform, op sys, bug status, resolution, duplicate id, bug file location, keywords, priority, bug severity, target milestone, dependent bugs, blocked, votes, reporter name, assigned to name and number of comments. Moreover, the resolution of each bug was filtered and it was found that 29,479 records were either NEW or ASSIGNED and thus removed, since they have not been fixed yet. After filtering out the other resolution statuses, we are left with 106,187 uniquely fixed bugs for which we work with in our project.

In order to predict for the time it takes a bug to go from reported to FIXED, we calculate the time difference between the time a bug is reported and the time it is fixed. As it is impossible to figure out the exact amount of time a developer has actually spent fixing a bug, we decided to examine the lifetime of a bug and not the time it takes for a developer to actually fix a bug. Also, reopening a completed bug is not considered, since it is not a common practice and it is not easily detectable. Thus, we are strictly looking at start date and end date of the bug, that is, the time a bug is completely fixed.

C. Machine Learning: Classification

Data mining is defined as the extraction of implicit, previously unknown, and potentially useful information from the load of data in repositories. It is a set of processes performed automatically whose task is to discover and extract hidden features from large data sets [18]. In bug repositories, the large volume of the bug reports makes manual analysis impractical. Hence, by leveraging data mining techniques, we aim to devise good indicators to predict the length of time it takes for a bug to be resolved.

The prediction of bug lifetime is done using Weka [19], which is a collection of machine learning algorithms and preprocessing tools [18]. Weka supports only a single file or relation, so in order to import the data to Weka, the entire dataset must be processed into a single file, as described in the previous section. To normalize our dataset before running

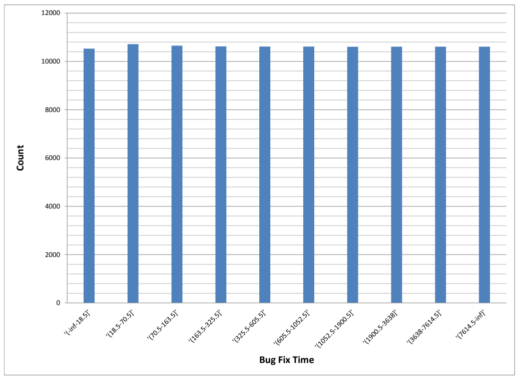


Fig. 3: Discretizing the bug fix time (hours) into 10 bins

Algorithm	FreeBSD Prediction	Eclipse Prediction
0-R	10.00%	29.10%
1-R	17.00%	31.00%
C4.5 Decision Tree	19.49%	31.90%
Naive Bayes	**18.45%**	**32.50%**
Logistic Regression	16.53%	34.90%

TABLE I: Naive Bayes vs. Other Classifiers [10][11]

our algorithm, we use the discretize function, which changes numerical data into nominal data by placing a range into a values into bins. We have considered 10 bins of equal frequency to classify bugs into different bins, predicating the bug fixing time (Figure 3). A size of 10 was chosen since it can be compared to previous related works and also because it is large enough that we can get a good estimate of how long a bug takes to get fixed. Next on the workbench is the classify panel which is used to apply the classification algorithms. The algorithms used for our prediction model are described below.

D. Applying a Machine Learning Algorithm

Bougie et al. [11] and Panjer [10] found no significant difference between the predictive ability of other well-established algorithms such as SVM [20], C4.5, logistic regression, 0-R, and 1-R. A simple Naive Bayes algorithm showed a difference around +/-2% compared to the other algorithms. The results of two studies are shown in Table I. Hence, we adopted Naive Bayes as our classification algorithm. Moreover, according to our experiments a decision tree with a reduced dataset gets results around 9-14% correctly predicted, which is not much better than randomly selecting 1 out of the 10 bins for selection. Hence, we decided to rely our experiment on Naive Bayes algorithm.

A Bayes classifier is a simple probabilistic classifier based on applying Bayes' theorem (from Bayesian statistics) with strong (naive) independence assumptions. In spite of their naive design and apparently over-simplified assumptions, Naive Bayes classifiers have worked quite well in many complex real-world situations. In 2004, analysis of the Bayesian classification problem has shown that there are some theoret-

Dataset	Firefox	Eclipse
10 bins	33.56%	25.18%
5 bins	48.26%	39.19%
10 bins with post submission data	34.39%	26.41%

TABLE II: Prediction Accuracy Using Different Datasets

ical reasons for the apparently unreasonable efficacy of Naive Bayes classifiers [21].

E. Evaluation and Results

A model produced from a training set of roughly 70,000 records or 66% of the total dataset was outputted along with the calculated predictions for the values in the testing set. Overall, we found that it correctly classified the duration it took for a bug to get fixed 25.14% of the time (Recall value). The results are better than those found in the FreeBSD system (Figure I), but not as good as the ones Panjer described in [10]. This is most likely because in Panjer's study, he only used 7 bins with the first bin having a size that is twice as large as the other bins. By having fewer bins and hence a smaller chance to make an incorrect prediction, the accuracy of the model will increase. The matrix in Figure 4 summarizes the predictions made by the Naive Bayes algorithm. We know that it predicts the correct bin 25% of the time, but how far off are the incorrect predictions? The rows represent the predicted bin while the column is the actual bin the instance belongs to. From this matrix, we can see that the classifier tends to get most of the bugs that are fixed quickly or fixed slowly correct. In addition, for these early and late bins; it seems that if a bin is classified incorrectly, then it is likely to appear closer to the correct bin. For example if we take a look at the row 'a', then the 2nd and 3rd most incorrectly classified result is 'c' and 'b' respectively. This means that if a mistake were to occur, it is likely to be close to the correct one. However for bins in the middle, this correlation is not noticeable. The precision and recall for each bin is also listed in Figure 4 and confirms that there is a weakness in predicting bugs particularly in bins 'b', 'd', 'e', and 'g'.

1) More on Validation: Besides the main experiment on running Naive Bayes on Eclipse data with 10 bins the Bugzilla reports for Firefox was parsed and classified as well to give a comparison.

The first comparison made is by shrinking the number of bins down from 10 to 5 (Table II). Doing so allows our model to be more accurate in guessing the correct bin; however, the trade-off is the bins become larger and thus we lose accuracy within a bin. Depending on the nature of the application, a less fine grain prediction might be a better choice. Post submission data such as number of votes or number of comments was included to the dataset. We wanted to see what kind of effect this extra information had in helping the bug get fixed. One can presume that if a bug report has more activity, then it would mean that more developers are working on fixing the problem. Thus, this extra information should help in predicting the fix time of a bug. Table II shows that although this extra information has a positive effect on precision rate, the improvement is not much significant. The accuracy of the prediction for Eclipse increased by 1.23%, while Firefox just increased by 0.83%.

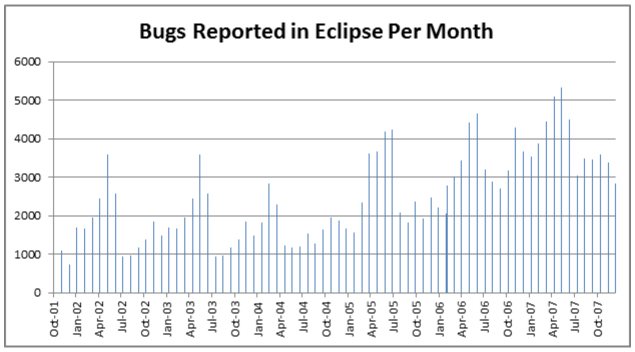


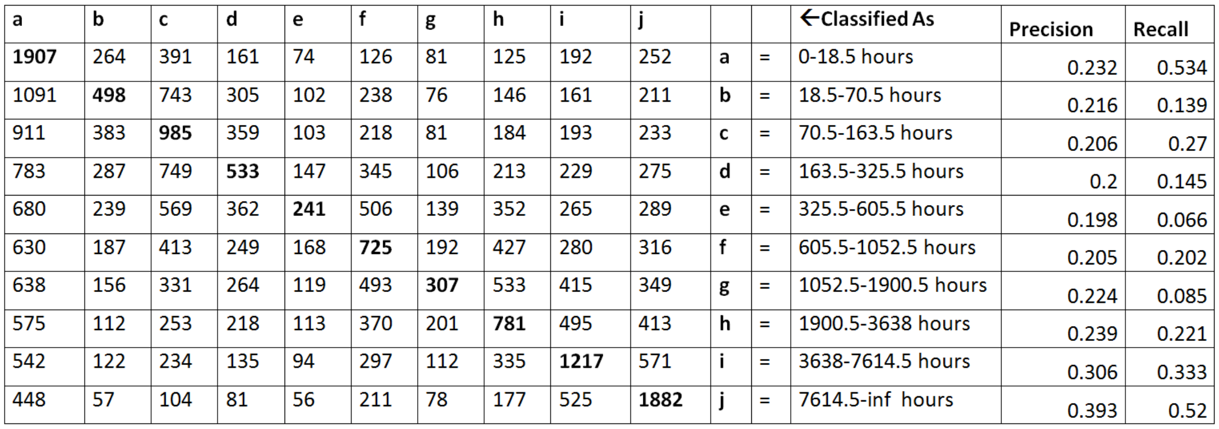
Fig. 5: Eclipse bug report distribution over a period of 6 years

In addition to the Eclipse data, 366,112 Firefox bugs were retrieved from 1999-2007. The same process was done to parse these bugs and have the data placed into the Naive Bayes classifier. We found that the bug report data of Firefox has a much better ability to predict bug fix time than does Eclipse. All the fields used were the same as Eclipse, so there was no extra information that the Firefox data provided. The only difference noted between the two datasets is the bugs in Firefox take longer to fix on average than do the ones in Eclipse. For example, the first bin in Eclipse has bugs whose fix time ranged from 0-18 hours while in Firefox the range was from 0-91 hours. However, the bins are still of equal numbers and so it is hard to tell why there is such a difference in prediction accuracy.

IV. Bug Allocation Problem

In large open-source development projects, bug allocation is a tedious and time-consuming task usually done by bug triagers. Therefore, the quality of these assignments depends heavily on the contextual knowledge of a triager about the reported bug, and knowledge about the different developers who are willing to contribute to the project in a timely manner. In open-source projects, there are usually two groups of developers involved: professional developers whose jobs are maintaining the system and fixing defects/bugs, and amateur developers who do not get paid but contribute as expert users occasionally. In both cases, it is the triager's responsibility to assign the most proper bug to the appropriate contributor who 1) is knowledgeable about that type of bug, and 2) has availability in his/her schedule to fix the assigned bug as quickly as possible.

In large open-source projects such as Eclipse and Firefox, there may be thousands of developers involved, each with different schedules, capabilities, expertise, etc. This makes it almost impossible for triagers to take all these constraints into account. Moreover, the number of bugs reported daily in these projects is huge according to our preliminary analysis in Figure 5. In fact, even the most updated systems are often



a	b	c	d	e	f	g	h	i	j			←Classified As	Precision	Recall
1907	264	391	161	74	126	81	125	192	252	a	=	0-18.5 hours	0.232	0.534
1091	**498**	743	305	102	238	76	146	161	211	b	=	18.5-70.5 hours	0.216	0.139
911	383	**985**	359	103	218	81	184	193	233	c	=	70.5-163.5 hours	0.206	0.27
783	287	749	**533**	147	345	106	213	229	275	d	=	163.5-325.5 hours	0.2	0.145
680	239	569	362	**241**	506	139	352	265	289	e	=	325.5-605.5 hours	0.198	0.066
630	187	413	249	168	**725**	192	427	280	316	f	=	605.5-1052.5 hours	0.205	0.202
638	156	331	264	119	493	**307**	533	415	349	g	=	1052.5-1900.5 hours	0.224	0.085
575	112	253	218	113	370	201	**781**	495	413	h	=	1900.5-3638 hours	0.239	0.221
542	122	234	135	94	297	112	335	**1217**	571	i	=	3638-7614.5 hours	0.306	0.333
448	57	104	81	56	211	78	177	525	**1882**	j	=	7614.5-inf hours	0.393	0.52

Fig. 4: Matrix for the Prediction Accuracy of the Naive Bayes Model Using Eclipse Data

unable to keep track of all the developers and the progress of bug fixing. The decentralized nature of bug assignment makes it an attractive candidate for using a distributed agent-based approach for bug assignment.

A. *Multiagent Systems*

A multiagent system is composed of several autonomous intelligent agents having different information and/or diverging interests. Multiagent systems can be used to solve problems that are difficult or computationally expensive for an individual agent or a centralized system to solve. Also, these systems are used in situations where there is diverse information in the system that is almost impossible for a centralized system to gather all needed data. We have chosen to adopt a multiagent based approach to tackle our problem of bug assignment, because such systems allow the representation of every single coordination object, i.e., the responsible entities, as single autonomous agents with their own goals. The agents can react with the needed flexibility to changes (as new bugs are reported or schedules are changed) through proactiveness and responsiveness [22].

In the bug allocation process, we model two types of active agents involved as intelligent software agents: bug triagers and developers. Triagers are responsible for assigning bugs to the most qualified developers. Developers are modeled as intelligent agents that are able to analyze information about their respective developer, arranging schedules, calculate preferences and limitations, and make decisions on their behalf.

B. *Market-Based Solution*

One of the mostly used, and yet simple, mechanisms for marketing goods and services are auction-based processes. Auctions are commonly used for allocating resources in multiagent settings. Agents express their desire of having a particular item by sending their bids to a central auctioneer who is responsible for making the allocation based on the received bids [23].

The basic interactions in any auction are included in two major phases: bidding process, where individuals reveal their willingness to buy an item by assigning a value, and winner determination process where auctioneer identifies the winner by applying specific judgment rules. In different types of auctions, these two major processes vary in details and implementation. Auction is a simple and effective mechanism as it keeps the communication level at minimum. Agents maximize their local utility function with minimum communication with the auctioneer. The auctioneer is in charge to receive bids from agents and coordinate them based on a global function.

In this paper, we have modeled a market inspired coordination mechanism based on a simple sealed first-price auction, also known as a first-price sealed-bid auction [24]. This type of auction is a single-round mechanism, as oppose to English auctions, where bidders simultaneously submit their bids to the auctioneer without revealing the assigned value to other bidders. As this is a single-round mechanism, bidders do not have a chance to adjust their bids according to other participants' bid values [23]. Then the auctioneer identify the highest bidder and announces the winner. Of course in this type of auction, the winner pays exactly the same amount of bid, as oppose to Vickrey auctions [25] where winner pays the second-highest bid. First-price sealed-bid auctions are not strategyproof[1], meaning that bidders may choose to be dishonest and lie about their private utility with the incentive of winning the auction while paying less.

In this paper, triagers are modeled as auctioneers that receive various bug reports at a time (as goods ready to be auctioned off). Bidders are developers modeled as agents trying to bid over a reported bug. The winner of each auction is the developer agent with the highest bid who stated the highest value for that particular bug. Upon receiving a bug report, the triager opens up an auction based on the characteristics of the reported bug such as bug category, severity, and priority.

[1]In the multiagent systems literature, strategyproof mechanisms refer to ones where agents are truthful and have no incentive to lie or hide their private information.

C. Market-Based Bug Assignment

The bug assignment mechanism consists of four phases: the broadcasting phase, the subscription phase, the bidding phase, and the awarding phase. This mechanism dictates the simple auction process in market mechanism. The triager agent receives bug reports and opens up an auction by broadcasting the bug's stated category (e.g., user interface, database, etc.). All developer agents receive this information, and will subscribe to the auction if their expertise includes the bug category. A developer agent has information about the developer's current status, schedules, and the number of bugs in the queue as well as the number of bugs he/she has fixed so far and the average fixing time of the bugs. The latter data is gathered dynamically by looking at the performance of the developer. In the bidding phase, developer agents compute the price they are willing to pay for the specific bug and submit their bids for the needed bug. Finally, the auctioneer (bug triager) announces the winner of the auction, assigns the bug to the winner, and closes the auction.

D. Pricing Mechanism

In the proposed auction-based bug allocation mechanism, the developer agents compete with each other over the bugs to achieve the objectives of their corresponding developers [26]. In contrast to commercial domains of marketing, the utility or pricing functions cannot be based upon monetary values. While in e-commerce scenarios, the human principals reveal their preferences through their willingness to pay, developers cannot reveal their preferences through their willingness to pay for a specific reported bug. The preferences rather have to be based upon developers' past history of fixing bugs and expertise on fixing particular type of bugs. To do this, we propose a pricing mechanism that takes all the history of a developer into account as well as their current schedule of assigned bugs. A developer's willingness to pay (η) is calculated as follows:

$$\eta_i = \kappa_i^{-1} \sum_{b \in B_i} \frac{priority_b}{T_b}, \quad \kappa_i = \sum_{j \in Queue} \hat{t}_j \times 100 \qquad (1)$$

where B_i is the set of past fixed bugs in the history for the ith developer, $priority_b$ denotes the assigned priority level (or severity level) of a certain bug report b, and T_b is the actual fix time of bug b which have been resolved in the past. $\hat{t}_j$ is the predicted time it takes for developer i to fix bug j extracted from the data mining algorithm. We use κ_i^{-1} as a normalizing factor for developer i. This normalizing factor is a linear decaying rate that ensures that developers will not be overloaded by many bugs in their queue. If a developer has already too many bugs in her schedule, she will not be willing to bid high, and as a result, will not win the auction. Hence, the more a developer contributes to the project in a timely fashion, the higher she can bid over a bug.

The "willingness to pay" value models a developer's experience and quickness in terms of response (fixing bugs) and emphasizes on the number of fixed bugs by the developer. This may be a simplistic representation of developers' experience; however, it encapsulates the necessary information to model the contribution and reliability of developers. Nonetheless, the pricing mechanism that outputs a developer's willingness (or reliability) to pay can be changed to any other utility function representing developers' utility when bidding on an item.

V. EVALUATION

We have implemented a Java program to simulate our proposed auction-based mechanism. We now discuss how we have used this tool to evaluate our ideas using real-life data from the Eclipse project.

A. Data

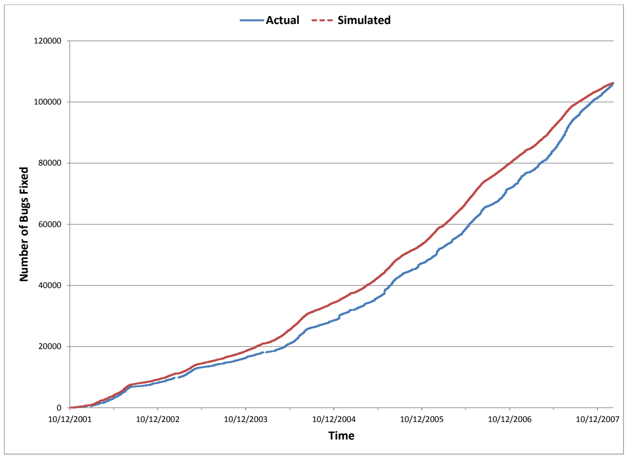
In this section, we discuss the evaluation of our proposed bug allocation mechanism. As data, we used the bugs reported in the Eclipse project in the period of 2001–2007. We ran our auction-based bug assignment mechanism on this data, and validated the proposed allocation mechanism by comparing it to the real-world data extracted from the Eclipse Bugzilla reports. For simplicity, we assume that all the developers can bid on all the reported bugs by just submitting their "willingness to pay" value to the bug triager. To simplify the allocation procedure, we do not consider multiple triagers for each bug category, but rather have one triager who opens up an auction upon receiving each bug report. Due to the dynamic nature of the bug reports and uncertainty about the fixing time of each reported bugs, we consider the time it gets for a bug to be fixed by looking at the bugs with status of FIXED or RESOLVED. Since the fixing time is derived from the real values reported for each bug, it assures that our market mechanism improves the structure of bug allocation. Later in the next section we will evaluate our results using the predictive time values obtained from our classifying approach in Section III-C.

B. Results

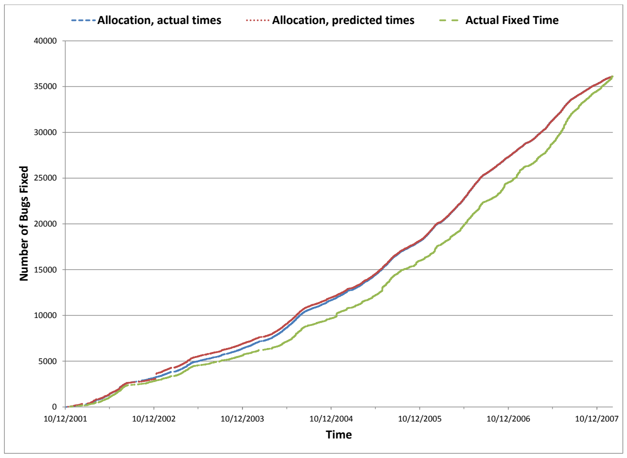
In this section, we compare our allocation method with observed industrial practice. As discussed before, Eclipse development over the period of 6 years (2001–2007) is considered as simulation data.

1) Bug Allocation Mechanism with Real Bug Lifetime: Figure 6a illustrates the results of simulation using real bug reports. To evaluate the effectiveness of the market-based allocation mechanism, we assume that the fixing time for each bug is equal to the observed real-world fixing times. For example, if a bug took 100 days to be fixed, our algorithm allocates 100 days for this bug to be completed. Although, in practice, the time it takes for a bug to be resolved may depend on various factors such as developer who fixes the bug, for simplicity we assume this time is equal for all the developers. This will help us to have a fixed baseline for validation purposes, since in this experiment we are interested to see the effectiveness of our allocation mechanism.

As shown in the figure, the proposed allocation mechanism outperforms the real practiced scenario in terms of having



(a) Real bug lifetime vs. current practices



(b) Cumulative number of bugs resolved: predicted bug lifetime

Fig. 6: The number of resolved bugs

more number of bugs fixed in each timestamp. Nevertheless, due to the fixed number of bugs and equal fixing times, both figures converge to the total number of bugs at the end of simulation. To show the improvement over the practiced data, we use percentage change based on relative difference calculated as follows:

$$\delta = \frac{\upsilon - \hat{\upsilon}}{\upsilon} \times 100 \qquad (2)$$

where υ is the actual practiced data, and $\hat{\upsilon}$ is the result from the simulation. The trend line shows improvement in all the stages of simulation over the real practiced data with maximum improvement of 74.73% and the average improvement of approximately 16%. Within the first 4–5 months, our simulated bug allocations appear to have bugs fixed at a rate that is up to 74% faster than that of historic data. While in the last 4–5 months the rate is below 5% quicker. This is likely caused by the fact that we are not taking into account the rate of which bugs are reported. In the beginning, there is an overload of bugs and so our system will allocate all of them, while near the end, there are no new bug reports and so our bug allocations converge with that of the historic data.

Moreover, Figure 7 demonstrates the number of allocated to bugs to all the developers by the end of the timeline. This shows that our algorithm respects the developers' degree of involvement in the project, i.e., the most focused and active contributors still get the more bugs to fix than less active developers. Moreover, if a developer becomes more active and contributes more to the project, our system dynamically takes this into account by increasing the developer's bidding power, resulting in more bugs to be assigned to the developer. This essentially ensures the fairness of the bug assignment process.

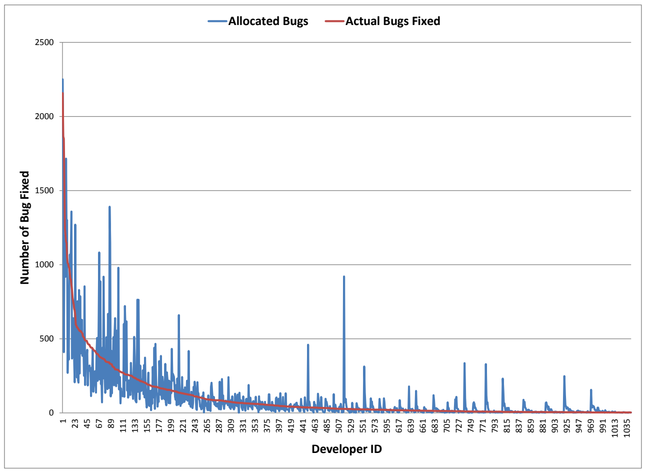


Fig. 7: Number of allocated bugs to each developer

2) Bug Allocation Mechanism with Predicted Bug Lifetime: In order to validate the usage of our allocation mechanism, we simulate the timeline on random bug reports. In this experiment, we use the predicted fixing time for each bug report estimated in Section III.

Figure 6b shows that allocation mechanism using predicted bug lifetime does as well as an allocation mechanism using real fixing times, and they both outperform the practiced scenarios in Eclipse project. The quality of bug allocation mechanism does not change even when adding predictive data.

VI. Threats to validity

In this section, we would like to express our awareness about some of the possible validity threats to our studies. First, we have used some simplifying assumptions for the valuation function that indicates the value of each bug for the developer. Although this will affect the final results in terms of not serving the most urgent bugs right away, it would have no impact on the proposed bug assignment mechanism as a concept. In fact, our market-based mechanism will improve by having more specific information about bug reports and developers.

Second, we have applied a linear decaying factor to weight out the developers with many bugs waiting in their job queue. There are some opportunities to use different decaying factors such as exponential decaying rates. However, we are unsure if this would affect our results. As possible future work, we will experiment with different approaches to find the most suitable decaying factors in the bug allocation domain.

VII. Conclusion and Future Work

Effective bug assignment in large software projects not only requires significant contextual information about both the reported bugs and the pool of available developers but also is a time-consuming and tiresome process. In this paper, we proposed an auction-based multiagent mechanism for assigning bugs to developers that is intended to minimize backlogs and overall bug lifetime. In this setting, developers and triagers are both modeled as intelligent software agents working on behalf of individuals in a multiagent environment.

We used a data mining technique to predict when a big might get fixed, with prediction accuracy of 25.14% for Eclipse and 33.54% for Firefox using 10 bins. We leveraged the predicted fix time of the reported bugs to augment the utility function in our bug assignment system. A preliminary look at our market-based allocation system shows a 16% gain against historic data based on a sample size of 213,000 bug reports over a period of 6 years. Our approach outperforms other attempts in automated bug assignment systems; however, many future adjustments can be made to get a more accurate bug allocation system.

As future work, we would like to examine the improvement of our bug allocation algorithm when there are multiple triagers involved in categorizing bugs and starting up the auctions. This might decrease the number of requests sent from the developers for a certain bug, and cause a faster (and probably more efficient) method of task allocation for bugfixes. Another interesting topic would be to study the current bug assignment mechanisms and devise a more detailed pricing mechanism that incorporates various preferences of the developers. This would, in theory, result in more developer satisfaction and in so doing would increase productivity. Another interesting variation would be to set up a mechanism where developers can bid for a bundle of desired bugs. This idea is mainly based on the auction systems literature and might improve the overall bug scheduling amongst individual developers.

References

[1] F. Heemstra, "Software cost estimation," *Information and Software Technology*, vol. 34, no. 10, pp. 627–639, 1992.

[2] N. Serrano and I. Ciordia, "Bugzilla, ITracker, and other bug trackers," *Software, IEEE*, vol. 22, no. 2, pp. 11–13, 2005.

[3] D. Čubranić, "Automatic bug triage using text categorization," in *In SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*. Citeseer, 2004.

[4] O. Baysal, M. Godfrey, and R. Cohen, "A bug you like: A framework for automated assignment of bugs," in *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*. IEEE, 2009, pp. 297–298.

[5] P. Paruchuri, P. Varakantham, K. Sycara, and P. Scerri, "Analyzing the impact of human bias on human-agent teams in resource allocation domains," in *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*. International Foundation for Autonomous Agents and Multiagent Systems, 2010, pp. 1593–1594.

[6] R. Cohen, M. Cheng, and M. Fleming, "Why bother about bother: Is it worth it to ask the user?" in *Proceedings of AAAI Fall Symposium*, 2005.

[7] M. Csikszentmihalyi, *Flow: The psychology of optimal experience: Steps toward enhancing the quality of life*. Harper Collins Publishers, 1991.

[8] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 34–43.

[9] N. Bettenburg, S. Just, A. Schroter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 308–318.

[10] L. Panjer, "Predicting Eclipse bug lifetimes," 2007.

[11] G. Bougie, C. Treude, D. German, and M. Storey, "A comparative exploration of freebsd bug lifetimes," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2010, pp. 106–109.

[12] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*. ACM, 2010, pp. 52–56.

[13] J. Anvik, "Automating bug report assignment," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 937–940.

[14] J. Anvik, L. Hiew, and G. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 361–370.

[15] J. Anvik and G. Murphy, "Determining implementation expertise from bug reports," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*. IEEE Computer Society, 2007, p. 2.

[16] (2011) Bugzilla documentation-life cycle of a bug. [Online]. Available: http://www.bugzilla.org/docs/tip/html/lifecycle.html

[17] (2011) MSR mining challenge 2008, retrieved march 2010. [Online]. Available: http://msr.uwaterloo.ca/msr2008/challenge/

[18] I. Witten and E. Frank, "Data mining: practical machine learning tools and techniques," 2010.

[19] (2011) Weka 3 - machine learning software in java. [Online]. Available: http://www.cs.waikato.ac.nz/ml/weka/index.html

[20] B. Scholkopf, C. Burges, and A. Smola, "Advances in kernel methods-support vector learning," 1998.

[21] H. Zhang, "The optimality of naive Bayes," *A A*, vol. 1, no. 2, p. 3, 2004.

[22] N. Jennings, "An agent-based approach for building complex software systems," *Communications of the ACM*, vol. 44, no. 4, pp. 35–41, 2001.

[23] Y. Shoham and K. Leyton-Brown, *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge Univ Pr, 2009.

[24] R. McAfee and J. McMillan, "Auctions and bidding," *Journal of economic literature*, vol. 25, no. 2, pp. 699–738, 1987.

[25] W. Vickrey, "Counterspeculation, auctions, and competitive sealed tenders," *The Journal of finance*, vol. 16, no. 1, pp. 8–37, 1961.

[26] E. Durfee, "Distributed problem solving and planning," *Multi-agent systems and applications*, pp. 118–149, 2006.