Bugs Assignment for Workload Distribution

Argaman Mordoch, Avraham Natan, Amir Elmishali and **Meir Kalech**
Software and Information Systems Engineering
Ben Gurion University of the Negev
e-mail: {mordocha, natanavr, amirelm}@post.bgu.ac.il, kalech@bgu.ac.il

Abstract

In the software industry, it is common that projects that are developed over time tend to grow and become more convoluted, which naturally serves as a fertile ground for bugs. These bugs are solved by developers that work on these projects. Efficient assigning bugs to developers is a key factor in order to reduce the average time to fix a bug. However, devising such assignment is non-trivial. The process introduces many challenges such as time consumption, optimizations in assignments, and resource limitations. To address these challenges, we suggest an automated system for assigning bugs to the developers who are most likely to solve them. Our bug assignment methods make the process of software maintenance easier and more reliable. We present two algorithms to tackle this problem. The first is inspired by Matrix Factorization, and the second is Machine Learning based. We compare the results of these algorithms to both an oracle assignment and baseline assignment methods. We evaluate the algorithms using data from 137 Apache open-source projects and show that our assignment methods well outperform the baseline methods.

1 Introduction

In the software industry, projects that are developed for a long period of time tend to grow in volume and become convoluted. To create the best possible product it is best to aspire to create code that is devoid of bugs. Since creating bug-free code is not possible, it is important to be able to assign developers with the bugs that they are most suited in solving.

Bug assignment is a process that entails many difficulties. Before assigning bugs to developers a process of bug triaging is needed. This process is time-consuming and is better invested in the solving of the bug.

Many works study the issue of automating the process of bug triaging as well as developers' bugs assignment. Some works [1; 2] suggest new methods to label bugs according to their required fields of expertise. Other works [3; 4] propose automated tools to assign bugs to developers based on previous bug reports. These works collect information on both the developers and the bugs from the bug reports. Using this method couples the knowledge collected on the developers to the bugs. Furthermore, this knowledge is project-specific and thus might be limited.

In addition to the issues previously presented, most previous works do not address the issue of workload. They try to optimize the assignment according to the developers' skills and expertise while creating a bottleneck of specific developers that solve most of the available bugs.

To address the issues presented above, we propose automated methods that assign bugs to developers. Our goal is to create an algorithm that can predict as accurately as possible the best developers group to assign a bug to. We aspire to assign the bugs while considering the workload and the size of the developers' groups.

In this paper, we present two methods for matching bugs to developers. Both methods use a clustering algorithm to cluster developers into groups based on their common skills and expertise. The first matching algorithm (MATRON) uses a clustering algorithm to cluster the bugs. In the second algorithm (MANGOLD) we match the bugs to their respected developers' group using a supervised machine learning classification algorithm.

To test our algorithms, we use these matching methods to distribute the workload between the developers. We tested our workload distribution algorithm on 137 Apache open-source projects with 8569 different developers. We trained our models on 40504 bugs and tested the trained models on 2414 bugs[1]. We compared our results to baseline matchers as well as to an assignment

[1] the reports were extracted from Jira

that assigns each of the bugs to the actual developer that was logged as the one that solved the bug, with the actual Jira logged time that took to solve it, i.e. an oracle matcher.

2 Related Work

Bug assignment is a well-studied field. Some proposed methods enable the developers to select bugs to solve according to different approaches. [3] present a theoretical method to assign bugs to developers where the developers rank their preferences and the system suggests bugs that comply with their preferences. In another paper by [4] the writers create a method based on auctions. Developers that want to solve a certain bug bid on it and the winner gets to solve it. Both of these methods reduce the total amount of time that it takes to solve bugs since developers are assigned to their preferred bugs, but both of them do not provide methods to identify the workload of the winners.

To solve the imbalance workload problem, [5] propose a method that distributes bugs to developers according to similarities between bugs (according to the bug report) and divides the bugs so that the workload is balanced between the available developers. This paper differs from our approach since it prefers a balanced distribution of bugs over an expertise-based distribution. Our method aspires to moderate the two approaches and create a semi-optimized distribution where the developers' expertise is taken into consideration.

A major difference between our approach and all of the previously presented bug assignment approaches is the fact that these methods rely solely on the data acquired from the bug report as their source of knowledge on the developers' skills and expertise. On the other hand, our approach collects data from the projects that the developers had worked on, which is independent of the information collected from the bug reports, and uses this information as the knowledge base about the developers' skills. This way, our approach is not project-specific and the knowledge regarding the developers can be applied to multiple projects at once. Using this approach, we can assign open-source developers to a variety of bugs without being limited to a single project.

3 Problem Description

In this section, we formalize the problem that this research intends on solving. As introduced, this study seeks to assign a bug to a group of developers. We expect the developers in the group to have the ability to solve the bug assigned to them. We would, therefore, like to define a developer by her development expertise and skills. To this end, we present the following terminology and entities.

Definition 1 (Developer). A DEVELOPER d is defined by a set of development expertise and abilities $d = \{pe^1, pe^2, ...pe^t\}$ where pe is a specific expertise. A set of developers is denoted by $D = \{d_1, ..., d_n\}$.

In reality, these development expertise and abilities are not usually known in advance, or not explicitly given, however, they could be inferred from past experiences of the developer. In particular, in this work, we gathered the development expertise and abilities by parsing the README files and other documentation files of the repositories that the developers had worked on. This parsed data is then being transformed into topics, using topic modeling ([6]).

When writing code and developing a program, developers often create and solve bugs. A bug indicates a fault in the software. A bug is usually described in an issue tracker tool like Jira, Bugzilla, and so on. It can be defined as follows:

Definition 2 (Bug). A BUG is an entity that represents characteristics of the software defect as can be seen in the issue tracker. It contains features such as: priority, comment count, description length, summary length and project. We denote a bug $b = \{f^1, ..., f^k\}$ is a set of features. A set of bugs is denoted by $B = \{b_1, ..., b_m\}$.

Sometimes, we would like to get bugs that are assigned to a group of developers. We denote $assigned(D')$ as the function that returns the set of bugs B' that were assigned to the developers that belong to D', i.e. $assigned(D') = \{b \in B : \exists d \in D' \; s.t. \; d \; is \; assigned \; to \; solve \; b\}$.

In this research, we assume that the more accurate the matching between the expertise set of a developer d and the information set of bug b, the faster the developer will solve the bug. For example, a developer with expertise in databases is more likely to solve a database typed bug faster than a developer with front-end expertise. We estimate the time to solve a bug by the degree of matching between the candidate bug and the candidate developer.

Determining the match between a bug and a developer is hard. Learning this matching process by using historical bug assignments could be achieved by solving a multi-class classification problem. In these problems, each bug's data is extracted to features and the class is the developer (the assignee). Solving such a multi-class classification problem, where the number of classes, i.e. developers is very large, is very hard. Therefore, in this research, we divide the developers into groups, by grouping developers with similar development expertise and abilities. In this way, we reduce the number of available classes (developer groups) and increase the number of solved bugs (instances) for each group.

Next, we define the estimated time to solve a bug by a group of developers, corresponding to the degree of matching between the bug and the developer group.

Definition 3 (Bug solving time). BUG SOLVING TIME $t(b, D')$ is the estimated time to solve bug b by a group of developers $D' \subseteq D$, and $t(B', D')$ is the estimated time

to solve a set of bugs $B' \subseteq B$ by a group of developers $D' \subseteq D$, which is:

$$t(B', D') = \sum_{b \in B'} t(b, D') \quad (1)$$

In this research, we would like to divide the bugs and the developers into disjoint subsets. Then, assign each subset of bugs to its most matching developer subset so that the total bugs solving time will be **minimized**.

Definition 4 (**Workload distribution).** Given a set of developers D and a set of bugs B, WORKLOAD DISTRIBUTION WD includes assignments of bug subsets to developer subsets. We denote the pair $\langle D^i, B^i \rangle$ to represent the subset of bugs B^i assigned to developer sub D^i. Given q disjoint subsets of D: $WD = [\langle D^i, B^i \rangle | 1 \leq i \leq q]$

Note, that D is divided into subsets D^i, and that bugs from B are assigned to the different D^i, and for that reason there are exactly q developer subsets and q bug subsets.

A subgroup of developers is capable to solve a certain number of bugs. We define this constraint in the following definition:

Definition 5 (**Load constraint function).** Given a set of developers $D' \subseteq D$, $Load$ is a function that limits the number of bugs that could be assigned to a set of developers: $Load : 2^D \rightarrow \mathbb{N}$.

This constraint limits the possibility of overloading a certain group of developers with many bugs.

Our goal is to optimize the workload distribution so that we can **minimize** the total amount of time it takes to solve a set of bugs.

Definition 6 (**Workload distribution optimization problem).** Given a set of developers D, a set of bugs B and a load constraint function $Load : 2^D \rightarrow \mathbb{N}$, a WORKLOAD DISTRIBUTION OPTIMIZATION PROBLEM is the problem of computing an optimal Workload distribution, as follows: $WD^* = argmin_{WD} \sum_{1 \leq i \leq q} t(assigned(D^i), D^i) s.t. \forall i : 1 \leq i \leq q, |assigned(D^i)| \leq Load(D^i)$

An optimal solution of the 'Workload Distribution Optimization Problem' can infer an optimal bug solving time. To achieve this, we formalize the following research question:

RQ: Which algorithm for assigning bugs to developer groups minimizes the value of WD?

In the next section we present algorithms to address the optimization problem.

4 Method Description

As described in Definition 6, the optimized workload distribution depends on the estimated time a candidate bug could be solved by a candidate developer. This time depends on the accuracy of the matching between the candidate bug and the candidate developer. Therefore, to compute the optimized workload distribution, we should first match between the bugs and developers who will potentially solve them. To this end, we use two different approaches. The first is inspired by *Matrix Factorization* [7] and the second is *Machine Learning* based.

Both approaches group the developers into subsets. The grouping mechanism is based on creating clusters that aggregate developers with similar expertise and abilities together. In the next section, we present the developers clustering method.

4.1 Developers Clustering

As a preliminary step, we gathered the development expertise and abilities of the developers by parsing the README files and other documentation files of the repositories that the developers had worked on. To divide the developers into clusters, we used a natural language processing algorithm - topic modeling. The algorithm extracts frequent development expertise for the developers, such as common knowledge in databases or DevOps. The repositories' descriptions, gathered from their README file and official website description, contain the main technologies and methods that the repositories have as well as the required expertise and abilities that a developer needs to be able to contribute to them.

The topic modeling algorithm in which we used to extract the expertise and abilities of the developers is Latent Dirichlet Allocation (LDA) [6]. We denote T as a topic that was created by the LDA algorithm. We define the distribution of the topics among the different repositories as follows:

Definition 7 (**Repository topic distribution).** Given a set of repositories R and a topic T, for each repository $r \in R$, we define REPOSITORY TOPIC DISTRIBUTION T_r to be the proportional part that is represented by the topic T in the repository r.

The method in which we acquire the value T_r is by generating the topics across all of the 'documents', i.e. the repositories description and README data. Then we use the algorithm to backtrack and calculate the proportional value that each of the repositories had contributed to the creation of the topic.

Each developer d contributes to repository r by submitting commits. The number of commits the developer had submitted is denoted as $\#commits_d(r)$ where the total number of commits that all of the contributors of the repository had submitted is denoted by $\#commits(r)$. For each developer, we also denote $R_d \subseteq R$ as the set of repositories that developer d had submitted commits to. Using the above definitions we define the topic contribution of a developer.

Developer's contribution is the proportional part of the contribution of the developer to the repository, which

is in direct correlation to the number of commits the developer submits. We denote $C_{d,r}$ as the contribution of the developer d to the repository r as follows:

$$C_{d,r} = \frac{\#commits_d(r)}{\#commit(r)} \quad (2)$$

Definition 8 (Developer's topic contribution). Given developer d and REPOSITORY TOPIC DISTRIBUTION T_r, we denote the DEVELOPER TOPIC CONTRIBUTION of d as TC_d, and define it by the following equation: $TC_d = \sum_{r \in R_d} T_r \cdot C_{d,r}$

After acquiring the developers' topics, the developers are clustered using the K-Means clustering algorithm, which enables the creation of developer groups where each group has its own specific required expertise and abilities, as defined in Definition 1. We denote the created $i'th$ developer cluster as $C_{dev}[i] \subseteq D$ (note that the following holds: $\bigcup C_{dev}[i] = D$ and $\bigcap C_{dev}[i] = \emptyset$).

The method of creating the developers' clusters, is the first building block in the process of creating an algorithm that is able to properly assign bugs to developers. Next, we present two methods to match bugs to developers' clusters.

4.2 Matrix Factorization Based Matching Algorithm (MATRON)

MATRON uses the developers clusters presented in the previous section, as well as the bug clusters. To create the bug clusters we use the K-Means clustering algorithm where the features for the clustering algorithm are the ones presented in Section 2. The created $i'th$ bug cluster is denoted as $C_{bugs}[i] \subseteq B$, and we note that the following holds: $\bigcup C_{bugs}[i] = B$ and $\bigcap C_{bugs}[i] = \emptyset$.

Recall that the developers are clustered into developer clusters, the next step is to define the connection between the assignee of a bug (i.e. the developer that solved the bug) and its cluster. We use the assignment table $Tab_{assigned}$ to represent this mapping:

$$Tab_{assigned}[i,j] = 1 \iff b_j \in assigned(C_{dev}[i])$$

Table 1 shows an example for such assignments.

After both the developers and the bugs are divided into clusters, we start the matching process. The method we use in order to match the developers to the bugs is inspired by the co-clustering matrix factorization algorithm. To this end, we define the Matching Truth Table:

Definition 9 (Matching truth table). Given a set of bugs, divided into clusters C_{bugs}, and a set of developers divided into clusters C_{dev}, MATCHING TRUTH TABLE ($Matches$) is a table that contains the sum of the 'true' assignments of bugs from a specific bug cluster, to developers from a specific developers cluster.

The matching truth table is generated by accumulating the assigned bugs according to a task management tool. For each cell $[i,j]$ in $Matches$ the value of a cell is defined as follows:

$$Matches[i,j] = \sum_{d \in C_{dev}[i], b \in C_{bugs}[j]} Tab_{assigned}[i,j] \quad (3)$$

Table 2 shows an illustration of the MATCHING TRUTH TABLE.

In Definition 3 we defined the bug solving time as the estimated time that is needed for a set of developers to solve a set of bugs. To be able to solve the workload distribution optimization problem as defined in Definition 6, we can compute the probability of a bug to be assigned to a developer using the matching truth table.

Definition 10 (Probabilities table). Given a set of bug clusters C_{bugs} and developer clusters C_{dev}, PROBABILITIES TABLE ($Probs$) is a table that contains the probability of bug $b \in C_{bugs}[i]$ to be assigned to developer $d \in C_{dev}[j]$. Each cell in the table is calculated using the MATCHING TRUTH TABLE $Matches$ as follows:

$$Probs[i,j] = P(b \in assigned(C_{dev}[i]) | b \in C_{bugs}[j])) = \frac{Matches[i,j]}{\sum_{k=1}^{|C_{dev}|} Matches[k,j]} \quad (4)$$

Table 3 shows an illustration of the PROBABILITIES TABLE. The probabilities are calculated by normalizing the values in $Matches$ by the sum of the cell's column.

Using $Probs$, we can calculate the assignment probability of bug b_i and developers cluster k as: $P(b_i, k) = Probs[k,j],\ s.t.\ b_i \in C_{bugs}[j]$

Given a bug b_i, we can use the above equation, to calculate for each developers' cluster, the probability of the bug being assigned to it. $prediction_{MATRON}(b_i)$ represents a set of these probabilities.

$$prediction_{MATRON}(b_i) = \{P(b_i, cd) | cd \in C_{dev}\} \quad (5)$$

4.3 Machine Learning Based Matching Algorithm (MANGOLD)

A second approach to assign bugs to developers' clusters that are best suited for solving them, is using machine learning techniques. Specifically, classification. Classification algorithms aim to classify an instance to the most suitable target. Generating a classification model is done by training a model with a training set that includes instances composed of features that characterize the instance, and a label (class) that characterizes the target. In our case, the instances of the training set are bugs. For each bug, we extract features, where the label of the bug is the cluster of the developer assigned to solve it. This training set is used to train a prediction model. Given a new bug, this model could recommend the best-suited developer cluster.

	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}	b_{16}
$C_{devs}[1]$			1	1				1						1	1	
$C_{devs}[2]$		1			1	1					1	1				
$C_{devs}[3]$	1						1		1	1			1			1

Table 1: An assignment table. All bugs marked by 1 in a row are assigned to developers in a cluster that corresponds to that row.

	$C_{bugs}[1]$	$C_{bugs}[2]$	$C_{bugs}[3]$
$C_{devs}[1]$	3	2	2
$C_{devs}[2]$	2	2	1
$C_{devs}[3]$	1	3	2

Table 2: The matching truth table. Contains the sum of all of the assigned bugs as mentioned in equation 3.

	$C_{bugs}[1]$	$C_{bugs}[2]$	$C_{bugs}[3]$
$C_{devs}[1]$	0.5	0.2857	0.4
$C_{devs}[2]$	0.3333	0.2857	0.2
$C_{devs}[3]$	0.1667	0.4285	0.4

Table 3: The probabilities table. The table contains the probability of bugs from a specific bug cluster to be solved by a developer from specific developer cluster, as mentioned in equation 4.

The features we use, are divided into the following groups:

1. **Shallow numerical data:** bug's priority, comments count, description length, summary length, and its project index as defined in Definition 2.
2. **Topic modeling based features:** these features were created using Latent Dirichlet Allocation (LDA) algorithm [6], similarly to the way we extracted features for the developers. We extracted topics from the description of the bugs and the comments commented on each bug in Jira. Using trial and error on a range of 3 to 11 topics, we picked 8 topics for the description topic modeling and 5 topics for the comments topic modeling.

For each one of the developers clusters ($C_{dev}[i]$), we also consider the confidence value of the classification as the likelihood of belonging to the cluster. For a bug b_i, we denote the algorithm's predicted cluster assignment probabilities as follows:

$$prediction_{MANGOLD}(b_i) = \{P(b_i, cd) | cd \in C_{dev}\} \quad (6)$$

Where $P(b_i, cd)$ is the probability of the developer cluster cd to solve the bug b_i, returned by the classification algorithm.

4.4 Workload Distribution Algorithm

To solve the workload distribution optimization problem (described in Definition 6), we propose a greedy algorithm, inspired by the knapsack solver [8]. Our greedy algorithm distributes the bugs across the developers' clusters in a way that the total time it takes to solve the bugs is minimized.

For each new bug that arrives, the algorithm invokes the function $prediction$. This function is defined as follows: $prediction(matcher, b_i) = prediction_{matcher}(b_i)$, where $prediction_{matcher}(b_i)$ is defined in equations 5 and 6 for MATRON and MANGOLD, correspondingly. This function returns the probability of b to be assigned to each developer cluster, organized in a set. From this set, we extract the maximal probability from the algorithms' predictions and the index of the corresponding developer cluster that had the maximal probability to solve the bug. If the load function for the selected cluster is full, the algorithm eliminates the highest probability, allowing the next highest probability to be chosen. Contrarily, if the load function is not full, the algorithm assigns the bug to the developer cluster and proceeds to the next bug.

5 Evaluation

In this section, we describe the experimental setup used to evaluate the matching algorithms as well as the workload distribution algorithm.

5.1 Data Collection

We used information that is accessible in **Jira** and **Github**. We divided the data collection into two parts, the data relevant to developers and the data relevant to bugs. We used the information available from 137 Apache open-source projects[2], in which 8569 developers work (some of them work on more than one project) and 100K+ Jira issues.

Developers Data

We collected the developers' data by fetching the information of each repository in the following manner:

1. For each of the tested Apache repositories, the main programming language was attached as a language that the repository's contributors know.
2. The repository's README.md file was analyzed and the data regarding the technologies that were used to create the repository was aggregated as well.

[2]https://issues.apache.org/jira

3. In cases where the README.md data was insufficient, we collected the above information from the repository's website or attached info links.

The above data is accessible through Github's REST API and through manually going over the textual information and validating its quality.

Bugs Data

We used information that is publicly available in **Jira**. We focused on collecting information that best characterizes and distinguishes bugs, as defined in Definition 2. We gathered the information in two stages: First, we collected shallow data of the bugs which can be acquired directly from the Jira API. This data includes information about the bug's priority and its description. We then collected the comments for each bug in a second and more exhaustive iteration. The types of data that we collected are mostly categorical (such as bug priority, assignee, project name, etc.) and textual (bug comments and bug description). We also collected information about the bugs' creation and resolution dates.

5.2 Experimental Process

In the following section we show the process the above data had to undergo in order to be used in our assignment algorithms.

Developers Data Processing

To process the data of the developers, the textual README, as well as the manually gathered repository description, was inserted into a topic modeling algorithm where it was distributed into topics. To select the number of topics, we used the coherence metric according to the implementation of the method suggested in [9]. The chosen amount of topics is the one that has the highest value of coherence which is 6.

After the creation of the topics, for each developer and each topic i we calculated the topic contribution TC_i (Definition 8) and normalized these values for each developer across all of their repositories. This normalization process ensures that all of the developer's topic contributions are summed up to 1.

The primary programming language of most of the projects used is Java. Therefore, we did not consider the programming language as a feature. Furthermore, in preliminary experiments, we noticed that using **only** the TC values created a massive cluster that contains most of the developers. To avoid this, we improved our features by using Principal Component Analysis (PCA) algorithm. With PCA, we managed to modify the features in a way that created a better representation of the developers, i.e. better representing the inter-connections between the features and we succeeded to create relatively well-distributed clusters.

Bugs Data Processing

For the MATRON algorithm, we had to cluster the bugs into bug clusters. To this end, we created feature families for each bug:

1. Shallow data as presented in Definition 2
2. Bug description features: topics extracted from the description of the Jira issues by topic modeling algorithm.
3. Bug comment features: topics extracted from the issue comments by topic modeling algorithm.

We clustered the bugs using the feature families described above and used the PCA technique to get 20 balanced clusters.

5.3 Competitive Algorithms

To test the performance of our bug assignment algorithm, we need to measure the degree of compatibility between the bug and its assigned developer. Optimally we would aspire that bugs would be solved by the developer that is most suited to the task. However, it is impossible to acquire information about the best-suited developer to solve a bug.

When viewing the bug reports, the assigned developer for a specific bug might not have been the best selection for the task. Sometimes, an assignment is set, due to manpower limitations rather than professional considerations. Nonetheless, if we neglect foreign influences that we could not consider anyway, we may assume that the developers that were reported as the ones that solved the bugs can be considered as an **optimal** assignment. Thus, an '**Oracle**' assignment is the one that assigns each of the bugs to the actual developer that was reported as the one that solved the bug. The bug solving time of the Oracle assignment is the time that the developer logged as the time to complete the Jira issue.

In addition to the Oracle assignment algorithm, we define two baseline methods to compare our results to. The first method uniformly distributes the bugs across the clusters according to the bug's index, i.e. the distribution of the bugs was conducted in an ordinal fashion.

The second method is based on features reduction. For this method, we used a set of features and distributed the bugs according to a predefined condition. We have experimented on several features and compared their performance and accuracy. The experimented features were: priority, comments count, description length, and summary length. We measured their performance in terms of accuracy and selected the feature that had the best performance which was the bug's **priority**. In total there are 8 different priority values. For each bug, the assigned developers' cluster was selected based on the modulo results of the priority, i.e. $selected\_cluster(bug_i) = priority(mod|C_{dev}|)$

Table 4 summarizes the evaluated algorithms - MATRON, MANGOLD, and the baselines.

Algorithm Name	Description
MANGOLD	Machine Learning based assignment algorithm
MATRON	Matrix factorization based assignment algorithm
Priority	Assignment based on the modulo value of the bug's priority
Uniform	Uniformly distributed bugs

Table 4: The different bug assignment algorithms.

5.4 Experimental Parameters

To define a point of reference for our algorithm, we denote $ACT(b)$ as the actual reported time needed to solve bug b, i.e. the Oracle assignment's bug solving time. Since our algorithm tries to assign the bugs to the most suited developers, we needed to find a method to penalize the results in case of an error. The penalization for the workload distribution generated by the assignment method, is defined as the following function:

Definition 11 (Penalty function). PENALTY FUNCTION is a function that penalizes the algorithm in correlation with the probability of the matching algorithm to match the bug to one of the developers' clusters. For a given bug b that was solved by the developers' cluster $C_{dev}[i]$ we denote:

$$Penalty(b, C_{dev}[i]) = \frac{1}{P(b, C_{dev}[i])} \tag{7}$$

We denote the penalized time for the completion of the bug solving task as: $t(b, C_{dev}[i]) = ACT(b) \times Penalty(b, C_{dev}[i])$

Penalization strategies

In Definition 11 we defined the penalty function that is used to penalize the workload distribution algorithm. We denote two penalization methods:

1. '$penalty\ \ miss$' - penalizing upon miss-classification. According to this penalization method, for each miss-classification, the total amount of time needed to solve a bug will be **multiplied** by the penalty value.
2. '$penalty\ all$' - penalizing each classification regardless of the correctness of the assignment. This method is tested as well since there is an uncertainty that the oracle solution is the actual optimal solution. In this method, **each assignment** is multiplied by the penalty value.

Bounding the developers clusters

In Definition 5 we defined a load function constraint on the number of bugs assigned to each developers' cluster ($Load : 2^D \rightarrow \mathbb{N}$). We evaluate two methods of setting this load function:

1. The number of bugs allowed for a given developer cluster **is bounded** by the proportional size of that developer cluster:

$$Load(C_{dev}[i]) = \frac{|C_{dev}[i]|}{|D|} \cdot |B|$$

2. The number of bugs allowed for a given developer cluster **is not bounded**: $Load(C_{dev}[i]) = \infty$

5.5 Evaluation Metrics

To evaluate our algorithms, we measure their *accuracy* and *F1* scores. To that end, we measured the receiver operating characteristics (TPR, FPR, TNR, and FNR) scores.

We assume that an algorithm with higher accuracy and F1 values will result (for the unbounded scenario) in a lower bug solving time. To measure these results, we denote the difference in percentile from the total time of the oracle algorithm for each algorithm A as $diff(A)$. $diff(A)$ is calculated as follows:

$$diff(A) = \left(\frac{\sum_{1 \leq i \leq q} t(B^i, D^i)}{\sum_{1 \leq i \leq q} ACT(B^i)} \right) \tag{8}$$

Since our goal is to create an algorithm that optimizes the workload distribution time, we aspire our results to be the closest as possible to the Oracle assignment algorithm. Due to this, we surmise that $diff(A)$ is an adequate measurement of the algorithm's performance.

6 Results

In this section, we present the results of the workload distribution algorithms. First, we focus on the machine learning assignment algorithm (MANGOLD) and examine the algorithm that performs the best. Then we compare this algorithm with the matrix factorization assignment algorithm (MATRON) and the baselines.

6.1 Machine Learning Assignment Algorithms

We tested MANGOLD with two machine learning classifiers - XGBoost and AdaBoost. Table 5 shows the accuracy and F1 score of these algorithms across the different feature sets. According to Table 5 the accuracy of the XGBoost algorithm is much higher than that of the AdaBoost algorithm.

	Accuracy	F1 Score
XGBoost - All Features	0.911	0.733
XGBoost - Shallow Data	**0.920**	**0.761**
XGBoost - Topics	0.773	0.320
AdaBoost - All Features	**0.854**	**0.562**
AdaBoost - Shallow Data	0.845	0.535
AdaBoost - Topics	0.768	0.303

Table 5: Accuracy and F1 score for XGBoost and AdaBoost across the different feature groups.

Figure 1a shows a comparison between XGBoost and AdaBoost in terms of $diff(XGBoost)$ and $diff(AdaBoost)$ (Equation 8), for different sets of bug features. The figure shows the results for Load function $Load(C_{dev}[i]) = \infty$ (i.e. the unbounded scenario). We can see that XGBoost significantly reduces the required

time for every feature subset. These results are explained by the high accuracy value (0.92) of XGBoost (Table 5). Since the algorithm rarely misses its classification, the aggregated time for each hit is identical to that of the oracle algorithm.

Figure 1b shows the same experiment with a load function $Load(C_{dev}[i]) = \frac{|C_{dev}[i]|}{|D|} \cdot |B|$ (i.e. the bounded scenario). Contrary to the previous experiment, we can see that the time that was calculated using XGBoost is much higher than AdaBoost, whereas AdaBoost retains its relatively low bug solving time. This assessment is consistent for all of the feature sets we have tested. An explanation for this result is that while using bounded load function, XGBoost is forced to choose one of the lower probability clusters and thus the resulting penalty is exceptionally high. On the other hand, AdaBoost distributes the probabilities between the clusters in a somewhat evenly manner. This creates a scenario where even if the algorithm is forced to make a wrong choice, i.e. not choosing the predicted cluster, the resulting penalty remains low. The results in Figures 2 are presented for the different configurations with $penalty\ miss$. We got the same trends for similar configurations for $penalty\ all$.

6.2 Comparison Between All Algorithms

In this section, we present a comparison between the different assignment algorithms. As a result of the experiments shown in the previous section, we consider only the best machine-learning algorithm in each configuration. In particular:

1. XGBoost with 'shallow data' features for the unbounded load function ($Load(C_{dev}[i]) = \infty$).
2. AdaBoost with 'all features' features for the bounded load function ($Load(C_{dev}[i]) = \frac{|C_{dev}[i]|}{|D|} \cdot |B|$).
3. MATRON - matrix factorization based algorithm.
4. Priority - distributes bugs according to the values of the priority of the bug.
5. Uniform - assigning bugs across the developers clusters uniformly according to the index.

	Accuracy	**F1**
XGBoost - Shallow Data	0.920	0.761
AdaBoost - All Features	0.854	0.562
MATRON	0.774	0.324
Priority	0.736	0.208
Uniform	0.720	0.161

Table 6: Accuracy and F1 scores of the different bug assignment algorithms

Table 6 shows the performance results of the listed algorithms in terms of accuracy and F1 score. The machine-learning algorithms outperform the matrix-factorization-based algorithm (MATRON), while MATRON performs better than the priority and uniform algorithms. Figures 2a, 2b show that XGBoost and AdaBoost outperform the other algorithms when using the unbounded and bounded load functions, respectively. This happens, however, when the penalty function is applied only to missed assignments. When the penalization strategy is penalty-all, and the number of assignments is bounded, then MATRON outperforms the AdaBoost algorithm as shown in Figure 3. The following results present the following phenomenon: MATRON's accuracy is lower than the other algorithms, but its bug solving time is lower in the scenario shown in figure 3. We deduce that this phenomenon is caused since MATRON's selected clusters have higher probabilities which result in the penalty being lower.

7 Conclusions and Future Work

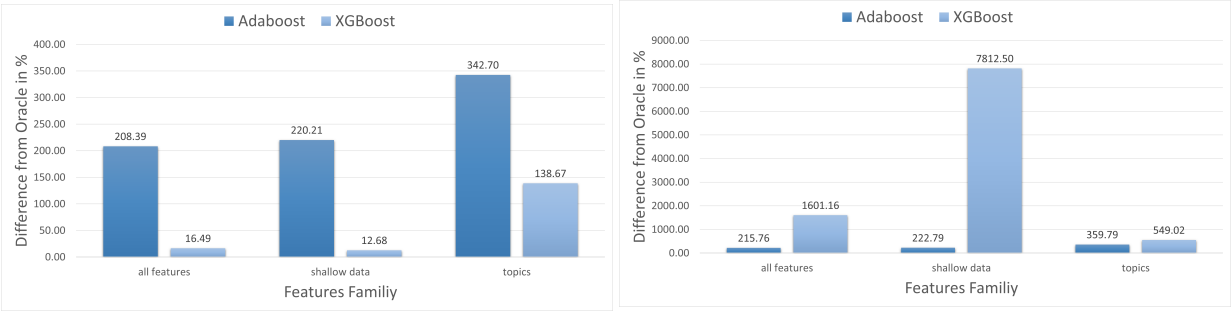
In this paper, we introduced the problem of workload distribution among developers. We presented two bug assignment algorithms. The first algorithm, MATRON, assigns bugs in a method inspired by the matrix-factorization algorithm. The second, MANGOLD, assigns bugs using machine-learning algorithms that classify the correct developers' cluster for each bug. The underlying machine learning algorithms that were used are XGBoost and AdaBoost.

We investigated 137 open-source projects that contained 42,918 bugs tasks, and with a total of 8569 developers. Our experiments show that for unbounded load function, XGBoost outperforms the other algorithms. But for the bounded load function AdaBoost performs the best when penalizing only on miss-classifications, while when penalizing all the assignments MATRON has the best performance.

In this work we regarded the developers as a uniform group that can contribute to the projects on with zero preparation time. This of-course, is a naive assumption since developers that are unfamiliar with a project need time to adjust themselves. As future work we intend to rank the developers in the groups and adjust the penalty based on the familiarity of the developers with the repository that the task belongs to. Another future work includes further investigation into the distribution of work within the selected developers cluster, i.e. the selection of the designated developer for the task.

References

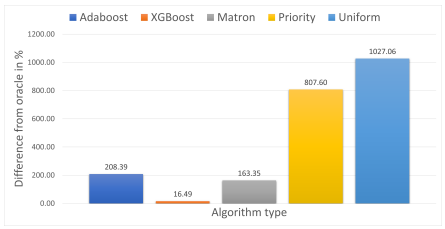
[1] Muhammad Younus Javed, Hufsa Mohsin, et al. An automated approach for software bug classification. In *2012 Sixth International Conference on Complex, Intelligent, and Software Intensive Systems*, pages 414–419. IEEE, 2012.



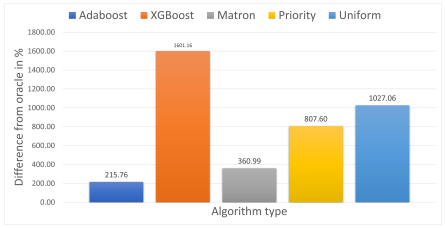
(a) $Load(C_{dev}[i]) = \infty$.

(b) $Load(C_{dev}[i]) = \frac{|C_{dev}[i]|}{|D|} \cdot |B|$.

Figure 1: Machine-Learning algorithms: $diff(A)$ for each algorithm as a function of features family, when missed predictions are penalized. Figure (a) - unbounded developer clusters, and Figure (b) - bounded developer clusters.



(a) $Load(C_{dev}[i]) = \infty$.



(b) $Load(C_{dev}[i]) = \frac{|C_{dev}[i]|}{|D|} \cdot |B|$.

Figure 2: All algorithms: $diff(A)$ when missed predictions are penalized. Figure (a) - unbounded developer clusters, and Figure (b) - bounded developer clusters.

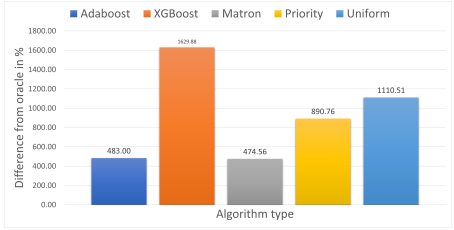


Figure 3: All algorithms: $diff(a)$ when all predictions are penalized, for bounded developer clusters ($Load(C_{dev}[i]) = \frac{|C_{dev}[i]|}{|D|} \cdot |B|$).

[2] Nachai Limsettho, Hideaki Hata, Akito Monden, and Kenichi Matsumoto. Unsupervised bug report categorization using clustering and labeling algorithm. *International Journal of Software Engineering and Knowledge Engineering*, 26(07):1027–1053, 2016.

[3] Olga Baysal, Michael W Godfrey, and Robin Cohen. A bug you like: A framework for automated assignment of bugs. In *2009 IEEE 17th International Conference on Program Comprehension*, pages 297–298. IEEE, 2009.

[4] Chetna Gupta and Mário M Freire. A decentralized blockchain oriented framework for automated bug assignment. *Information and Software Technology*, page 106540, 2021.

[5] Jaekwon Lee, Dongsun Kim, and Woosung Jung. Cost-aware clustering of bug reports by using a genetic algorithm. *J. Inf. Sci. Eng.*, 35(1):175–200, 2019.

[6] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[7] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[8] K. W. Ross and D. H. K. Tsang. The stochastic knapsack problem. *IEEE Transactions on Communications*, 37(7):740–747, 1989.

[9] Michael Röder, Andreas Both, and Alexander Hinneburg. Exploring the space of topic coherence measures. In *Proceedings of the eighth ACM international conference on Web search and data mining*, pages 399–408, 2015.