Software Component Prediction for Bug Reports

Wei Zhang WZHANG@ADOBE.COM
Chris Challis CHALLIS@ADOBE.COM
Adobe Inc., Mclean, USA

Editors: Wee Sun Lee and Taiji Suzuki

Abstract

In a software life cycle, bugs could happen at any time. Assigning bugs to relevant components/developers is a crucial task for software development. It is also a tough and resource consuming job. First, there are many components in a complex system and it is hard to understand their interactions and identify the root cause. Second, the list of components keeps growing for actively developed products and it is not easy to catch all updates. This task also faces several challenges from the machine learning point of view: 1) the ground truth is mixed with multiple levels of labels; 2) the data are severely imbalanced. 3). concept drift as future bugs are unlikely to come from the same distribution as the historical data. In this paper, we present a machine learning based solution for the bug assignment problem. We build component classifiers using a multi-layer Neural Network, based on features that were learned from data directly. A hierarchical classification framework is proposed to address the mixed label problem and improve the prediction accuracy. We also introduce a recency based sampling procedure to alleviate the data imbalance and concept drift problem. Our solution can easily accommodate new data and handle continuous system development/update.

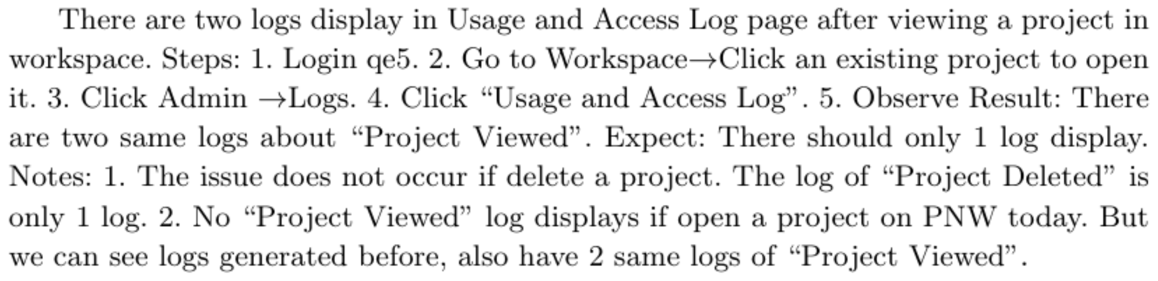
Keywords: Bug triage, Hierarchical classification, Machine Learning Application, Imbalanced Data, Recency based sampling

1. Introduction

As more and more features and functionalities are added to a software system, it is inevitable that software bugs will emerge. For example, nearly 1000 new bugs are reported each month for a commercial software product that is developed by our organization. To fix them timely, bugs have to be assigned to the right developers. In the industrial environment, personnel changes (internal transfer/leaving company) are frequent, but software components and their related developers are always well defined. Therefore, instead of directly assigning to individual developers, we try to assign each bug to the component which is the root cause of the bug. Then the bug can be resolved by the developers who are maintaining the component. However, assigning bugs to the right components is a very challenging task. First, there are usually many components ($\sim$ 200 for our product); just memorizing their names is not easy. It is much more difficult to understand the interactions between so many components and determine the root cause. In addition, the list of components keeps growing, especially when the system undergoes significant updates. For instance, we have observed new components almost every month during the past year. It is not trivial for bug assigners to keep the up-to-date knowledge on all components. All these challenges make the bug assignment process time consuming and error prone. According to our Quality Engineers, it often takes days or even weeks to get bugs assigned correctly. On the other hand, the ever-growing bug reports offer us a large amount of historical data that we can learn from. Therefore, we propose a machine learning based solution to alleviate the human burden and speed up the process.

1.1. Problem Definition

We use JIRA to track bug reports. Each bug comes with its unique symptom, which is elaborated in the "description" field in the JIRA ticket. We can also see the assigned component for historical bugs in the "component" field. Our JIRA bug data set spans over 14 years and exhibits huge variations. It includes inputs from numerous people with all kinds of reporting and writing styles. Since the software is used internationally, bug reports include multiple languages. In addition, the lengths of bug descriptions vary substantially, from one sentences to multiple paragraphs and hundreds of sentences. Figure 1 shows a relatively short example:



There are two logs display in Usage and Access Log page after viewing a project in workspace. Steps: 1. Login qe5. 2. Go to Workspace→Click an existing project to open it. 3. Click Admin →Logs. 4. Click "Usage and Access Log". 5. Observe Result: There are two same logs about "Project Viewed". Expect: There should only 1 log display. Notes: 1. The issue does not occur if delete a project. The log of "Project Deleted" is only 1 log. 2. No "Project Viewed" log displays if open a project on PNW today. But we can see logs generated before, also have 2 same logs of "Project Viewed".

Figure 1: An example bug description.

To understand the process of assigning bugs to related component, we introduce some background about the "component" field in the JIRA database. Typically, a component name contains two levels of information, e.g., "Reports & analytics: Segmentation". The component name suggests that it belongs to "Reports & analytics" in the group level component (also called super-component in this paper) and "Segmentation" in the level of fine-grained component (sub-component) within "Reports & analytics". Currently, our data contains 202 sub-components and 40 super-components. A super-component usually contains multiple sub-components. For instance, "Reports & analytics" has 17 subcomponents. Usually multiple teams of developers are connected to a super-component, while a sub-component is typically maintained by a small team or even one developer. Obviously, the more specific the component assignment, the more quickly the bug can reach the responsible developer and get resolved. Thus, it is desirable to assign the corresponding sub-component for each bug.

However, due to the challenging nature of the problem, sometimes human experts are uncertain about which sub-component to assign for a bug. Nevertheless, they can attribute the bug to a super-component because this is much easier with less choices (40 vs. 202 possible components). In our data set, over 1/3 cases were assigned super-component labels only, e.g. "Admin Console", instead of more specific component labels such as "Admin

Console: Code Manager". Consequently, the human labeled ground truth data is mixed with both group level labels and fine-grained level labels. Those labels are not mutually exclusive and can make the classification problem ill posed. Here is an example for explanation: assuming there are six bugs which are evenly split between two sub-components: "Platform: Billing" and "Platform: Export". If one bug from each sub-component is assigned the super-component label ("Platform" only), we will see 3 labels in the data set: "Platform: Billing", "Platform: Export" and "Platform". Notice the "Platform" component contains bugs from both sub-components. If we try to learn a classifier using these component labels directly, i.e., treating the group component the same as the two fine-grained components, the classification problem would be inherently flawed.

The task also involves severely imbalanced data as shown in Figure 2. From the machine learning point of view, each component is a class. There are 202 classes in the sub-component level and the numbers of data for different classes vary significantly. In general, standard learning algorithms expect balanced class distributions. For severely imbalanced data, machine learning algorithms tend to overfit training data and perform unfavorably on unseen data He and Garcia (2009). Another challenge from the machine learning perspective is the concept drift problem as the distribution of bugs is not stationary. For example, let's say that most bugs came from the "Data workbench" component in the previous quarter, but the "Admin Console" component has more bugs this quarter, and next quarter more bugs may arise from the "Platform" component. This is highly possible because components have different development schedule. Usually bugs are likely to emerge from the component that is actively evolving.

To use machine learning to solve the problem, we have to address these 3 issues: mixed labels, data imbalance and concept drift. The remainder of the paper is organized as follows. Section 2 describes our solution and how we address these machine learning challenges. Then we discuss experiments in Section 3 and related work in Section 4. We conclude the paper in Section 5.

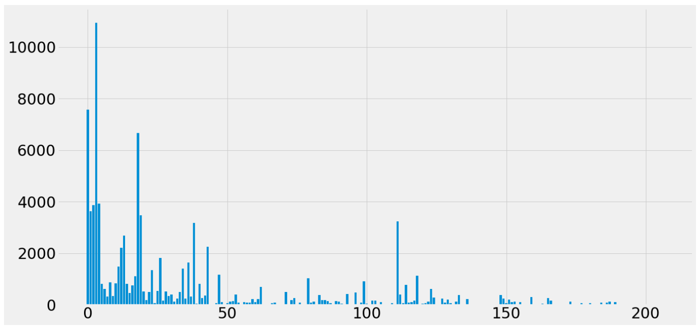


Figure 2: Distribution of the number of instances for all sub-component classes. The Y-axis is the number of instances per class. The X-axis is the indexed label of sub-components (the labels are omitted). The index is based on the first occurrence of each class. The most common class has 1000x more instances than the uncommon classes.

2. Our Solution

We propose a novel hierarchical classification framework to handle the problem of mixed ground truth labels. It contains a super-component classifier and a sub-component classifier. The two classifiers are trained separately, using the same training pipeline. In Section 2.2, we describe how to build these component classifiers. Then we present the hierarchical probability propagation and refining algorithm in Section 2.3. Note the word "hierarchical" refers to the semantic relationship between the super-component and sub-component classifiers, not that the two classifiers run in a hierarchical fashion like previous work. Figure 3 illustrates how the hierarchical classification works when doing prediction. We will discuss more on this in Section 4.

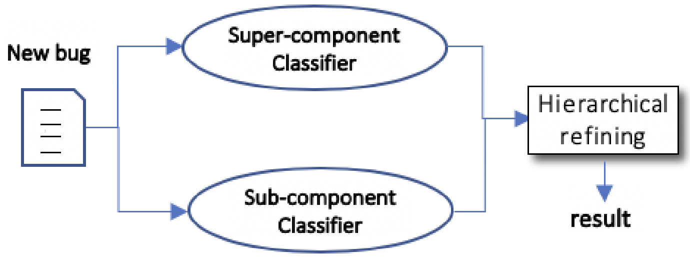


Figure 3: The prediction workflow: the two classifiers run in *parallel*, then their results are refined using Algorithm 2 and return to the user.

We also introduce a recency based sampling algorithm to tackle the problem of data imbalance and concept drift. This algorithm manipulates the training data that is used by the machine learning module.

2.1. Recency Based Sampling

As shown in Figure 2, the numbers of data instances vary significantly across different component classes. The majority classes have overwhelmingly more data instances than that of the minority classes. This problem has to be mitigated so the learned model can generalize to unseen data. The concept drift problem comes from the following facts: 1). Components that are under heavy development are likely to produce bugs. 2). Bugs are more likely to emerge from components that produced bugs recently than components that are more stabilized. These properties of bug reports have been demonstrated experimentally in Ye et al. (2014); Jonsson et al. (2016). We take it into consideration and propose Algorithm 1, a recency based sampling process to make the training data more balanced and adapt to concept drift. The "recency" of a bug measures the age of the bug, which is defined as the number of days between the training date and the date that it was reported.

In Algorithm 1, the parameter τ controls the probability of adding a data instance (bug in this case) to the sample. For example, if $\tau = A^*$, the most recently reported bug (age=0) has 50% probability of being added, while a median-aged bug has 25% probability and the oldest has 0 chance. The actually number of instances in S will vary due to different bug distributions. We set $\tau = A^*/100$ in our experiments. Thus the most recent bugs are almost

Algorithm 1 Recency based sampling

```
1. Initiate an sample $S = \{\}$
2. Determine the threshold $T_c$ for separating the majority/minority classes.
   $T_c = median(\{N_l\})$, where $N_l$ is the number of instances in class $C^l, l \in [1, M]$, where
   $M$ is the total number of classes.
3. For each data instance $d_i$ in the training data:
   (a) Find its corresponding class $C^i$. If $N_{C^i} < T_c$, add it to the sample $S$ (keep
       minority instances).
   (b) Otherwise, sample the majority instances:
       i. Calculate the probability $p_i$ of putting the data instance to $S$. $p_i = 1 - \frac{A_i + \tau}{A^* + \tau}$,
          where $A_i$ is the age of $d_i$, $A^*$ is the maximum age of all bugs, $\tau$ is a controlling
          parameter.
       ii. generate a random number $r \in [0, 1]$, if $r < p_i$, add it to the sample $S$.
4. Return $S$.
```

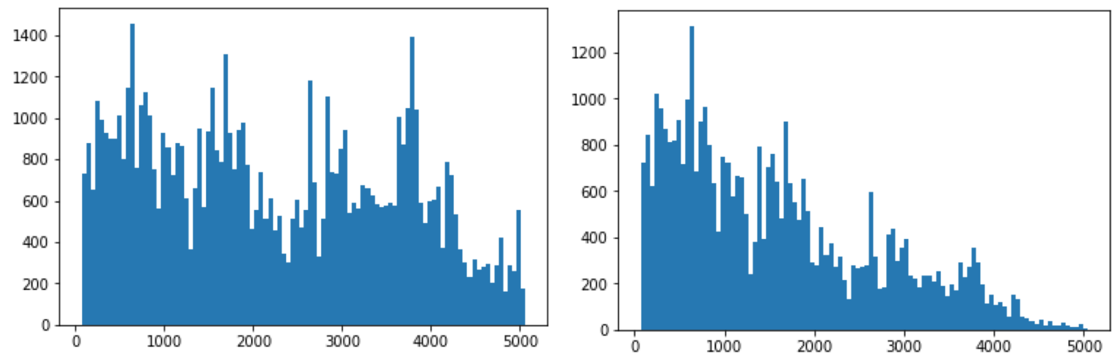
certain to be put to S, while old bugs are greatly reduced in the sample. Figure 4 shows the distribution of bug ages in the original data vs. the resulting sample. This sampling process will also change the ratio of data sizes between classes. Besides data instances of majority classes being reduced in general, if most instances of a class happened in the distant past, the sample S will contain few instances from the class. While for another class whose bug instances happened recently, S will retain almost all its instances. So this sampling process helps the learning module to focus more on components which have recent bug reports and are prone to new bugs. The components with old bugs are in more stable condition and less likely to produce new bugs; this property is implicitly encoded in the recency based sampling procedure.

2.2. Building The Component Classifier

Based on the sampled bug instances, we train classifiers to predict which component is responsible for a new bug. We first extract features from the bug description, and then feed the feature vector to the machine learning module for training/prediction. The difference is that training involves back propagation to minimize the training error, while prediction requires only the forward pass.

2.2.1. Feature extraction

First, a language model is built offline based on a large corpus of bug descriptions, which consists of JIRA bugs that are at least 180 days older than the current training date. This is done in the following steps: 1). preprocessing, including tokenization, removing stop-words and stemming. 2). build a word frequency matrix based on the processed text, which is specific to the bug data set. Then Latent Dirichlet Allocation (LDA) Blei et al. (2003) is



(a) Original data. (b) After sampling.

Figure 4: Distribution of bug ages before and after the recency based sampling. The X-axis is the number of days since the bug has been reported. The older the bug, the bigger the X value. The Y-axis is the number of instances for that time period.

used to extract a number of topics from this bug corpus. Each topic represents a distribution of words.

Once the set of topics is extracted, we can represent any bug description as a distribution of these topics. The weights of all topics form a fixed-size vector, which is used as the feature vector for the bug. We empirically set the number of topics to be 500, based on experimenting with different numbers. This compact yet discriminant feature enables us to build predictive models using various machine learning algorithms.

2.2.2. Build Predictive Model Based On Neural Network

To learn the map between a bug's feature vector and its associated human label (component), we use a Deep Neural Network (DNN) model. After trying different network structures, we adopted a DNN with 4 fully connected layers and Rectified Linear Unit (ReLU) as the activation function. It achieved the best tradeoff between bias and variance for our problem based on our comparative study. We implemented the network using PyTorch Paszke et al. (2017).

2.2.3. Training of super-component vs. sub-component classifiers

The two classifiers are trained using the same pipeline. The only difference is that their training data and labels are different. When training the sub-component classifier, we discard those instances which do not have a sub-component label. Thus, every label in its training data is a sub-component label and the mixed label problem is eliminated.

When training the super-component classifier, we aggregate sub-components to their group level (super-component) labels. For example, all "Data workbench: xxx" labels are converted to "Data workbench" (the corresponding super-component label). These cases which were only given the super-component labels are also included in this group. After aggregation, there are 40 group level super-component labels and their associated data instances. As opposed to the training of the sub-component classifier, the super-component classifier is learned based on the entire training data and their grouped labels.

The super-component classifier has the advantage that it has more instances per class. Usually the more training data, the more reliable the model. This is a clear advantage over the sub-component classifier. On the other hand, the symptoms of bugs from different sub-components might be quite dissimilar, although it is semantically meaningful to aggregate fine-grained labels. As a result, the learning itself is more challenging for the super-component classifier. The sub-component classifier has the exact opposite pros and cons. Thus, the two classifiers are complementary to each other and we combine them to improve the final result.

2.3. Hierarchical Refining

Denote the probability distribution of the super-component classification result as P_i, and the sub-component probability distribution as Q_i^j, where $i \in [1, k]$, k is the number of super-components, $j \in [1, f^i]$, f^i is the number of sub-components for the i^{th} super-component. Q_i^j is not on the same domain as P_i, but they are semantically related; more specifically there is a hierarchical relationship between the two classifiers. We improve the overall performance by propagating probabilities between them using Algorithm 2.

Algorithm 2 Probability propagation and refining

1. Aggregate results over sub-components to obtain the corresponding group-level probability distribution.

$$Q_i = \sum_{j=1}^{f^i} Q_i^j$$

   the result is a probability distribution over k super-components.

2. Update probability over super-components by stacking two probabilities P and Q.

$$\hat{P_i} = \alpha \times P_i + (1 - \alpha) \times Q_i$$

   α is a parameter learned from validation data, so that the resulting $\hat{P_i}$ achieves the best accuracy on the validation data.

3. Update the probability for sub-components:

$$\hat{Q_i^j} = \frac{\hat{P_i} \times Q_i^j}{Q_i}$$

We first refine the super-component probability by stacking the two classifiers, then propagate the probability down to the sub-component level. $\hat{Q_i^j}$ will be higher if $\hat{P_i}$ is bigger (because the super-component classifier assigns a higher probability to the i^{th} component). If $\hat{P_i}$ is smaller, $\hat{Q_i^j}$ will be lower as well. $\hat{Q_i^j}$ is still a probability distribution and sum to 1. The proof is as follows:

$$[h]\sum_{i=1}^{k}\sum_{j=1}^{f^i}\hat{Q_i^j} = \sum_{i=1}^{k}\sum_{j=1}^{f^i}\frac{\hat{P}_i \times Q_i^j}{Q_i} \qquad (1)$$
$$= \sum_{i=1}^{k}\frac{\hat{P}_i}{Q_i} \times \sum_{j=1}^{f^i} Q_i^j$$
$$= \sum_{i=1}^{k}\frac{\hat{P}_i}{Q_i} \times Q_i = \sum_{i=1}^{k}\hat{P}_i = 1$$

Our experiments confirm that this hierarchical framework improves system performance across all metrics (accuracy and recalls). Moreover, we can achieve better accuracy without speed loss: the super-component and the sub-component classifiers are independent of each other and can proceed in parallel. During test time, they share the same input feature vector - they only differ in the DNN model. The probability propagation and refining process in Algorithm 2 only involves a few hundred summation/multiplication/division operations. Thus the computational overhead is negligible and the performance is boosted without sacrificing speed.

3. Experiments

We use the JIRA data set of a commercial software product to train and evaluate our solution. It is a large data set with 91,626 JIRA bugs, dated from 08/2004 to 06/2018. The performance metrics that we use are: accuracy and recall@K. Recall@K is the percentage of test cases that have the ground truth within the top K ranked list of predicted components. The accuracy is equivalent to recall@1, the percentage of correct prediction when checking only the 1^{st} prediction result.

Our service is online and under regular update. Currently we follow a monthly updating/training cycle. During each training cycle, we pull all available data from the JIRA server, keep the latest month of data for testing and the previous data for training and validation. The language model is updated less often in a 6 month cycle, since the overall vocabulary and topics are not expected to change frequently. Three months of data are used as the validation data for estimating the parameter α used in Algorithm 2.

The performance evaluated using the last month is shown in Table 1. To demonstrate the benefit of the proposed hierarchical refining algorithm, we also list the original performance of each individual classifier. Clearly, we achieve better result with hierarchical refining. The running time for each bug prediction is virtually the same as that of the sub-component classification alone, since the running time of hierarchical refining is negligible. So we can conclude that the hierarchical refining framework efficiently resolved the mixed label problem, without sacrificing service time. Similar performance numbers were observed in the previous training cycles as well. For example, Table 2 shows the performance numbers when we started the first formal test in 2017.

To understand the robustness of our solution, we also investigated how our system would perform for bugs further away from the training period. Table 3 shows the results when

Table 1: Performance of the two individual classifiers vs. final refined results based on the last month of data ($\sim 1.1k$ bugs). All numbers increased after hierarchical refining in both component levels.

	Individual classifier		Hierarchical refined	
	Super-component	Sub-component	Super-component	sub-component
Accuracy	0.54	0.44	0.58	0.47
recall@3	0.73	0.61	0.76	0.63
recall@5	0.82	0.68	0.85	0.71
recall@10	0.90	0.76	0.92	0.80

Table 2: Performance for the first training cycle back in May. 2017.

	Accuracy	recall@3	recall@5	recall@10
Super-component	0.57	0.77	0.85	0.93
Sub-component	0.45	0.62	0.69	0.78

it was tested with 6 months of unseen bugs after May. 2017 (about 5.5k bugs reported after the 1^{st} training cycle). We can see that the performance numbers are lower than the numbers in Table 2. This is as expected since the testing data are further away from the training data in this case. The bug distribution is evolving continuously and getting more and more different over time (so called concept drift). Nevertheless, the performance is degrading gracefully and there is no catastrophic change. We believe that the recency based sampling algorithm helps to adapt to the drifting distribution.

We also look into the results on a per-class basis. In general, performance on classes with a large number of instances are fairly reliable, above 80%. Classes with particularly bad accuracy usually have much less instances in the training data. Table 4 and Table 5 show the prediction performance for each super-component and their corresponding numbers of instances. On the sub-component level, the problem is even worse. Many sub-components have only 10 or fewer instances, as Figure 2 has shown. The lack of data might be because they are not under active development, or they are the newly added components (bugs are just starting to emerge). Consequently, the performance on those classes are lower since we do not have enough data to cover them. Frequent updating is a way to alleviate the issue. Also, location based approaches Shokripour et al. (2013) may help to address the problem.

To verify the effectiveness of the proposed recency based sampling process, we compared it against the standard approach of undersampling the majority class Kubat et al. (1997). Both approaches were used to balance data and train classifiers. The resulting sample sizes were controlled to be virtually the same. We use 6 months of data for evaluation, so that

Table 3: Performance on the longer testing period.

	Accuracy	recall@3	recall@5	recall@10
Super-component	0.51	0.73	0.81	0.91
Sub-component	0.36	0.53	0.60	0.69

Table 4: Performance vs. number of instances for 5 components with highest accuracy.

Component	Ad hoc Analysis	Feedback	Workspace	Client Apps	Warehouse
Accuracy	0.95	0.89	0.88	0.85	0.84
recall@3	0.97	0.89	0.91	0.87	0.84
Data size	6734	860	5545	5685	4638

Table 5: Performance vs. number of instances for 5 components with lowest accuracy.

Component	Ad Analytics	Cache	Service	Core Platform	QE
Accuracy	0.25	0.2	0.11	0	0
recall@3	0.25	0.33	0.37	0.37	0.05
Data size	33	940	69	167	174

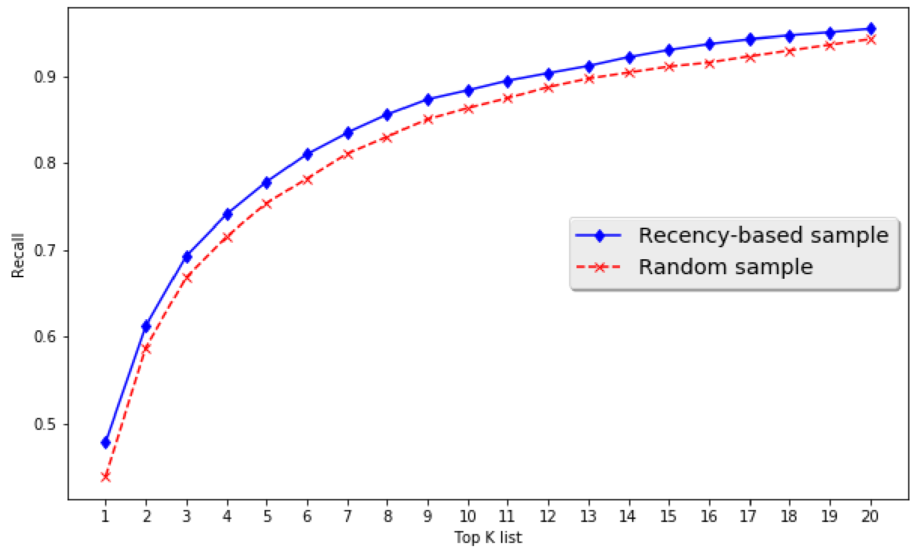
the test data is comprehensive and the results are reliable. The training data consists of bugs before that 6-month period. Figure 5 shows the classification results. The results based on the recency sampling procedure are clearly better than that of undersampling.

We also experimented with different numbers of topics and DNN layers, to find the best settings for our problem. We randomly selected 50% data for comparative experiments. The data was used to train a sub-component classifier. 10 fold cross-validation is used for performance evaluation. Figure 6 shows the performance comparison with varying numbers of topics. The performance first improves as the number of topics increases, then it saturates when the number of topics reaches 500. The performance with different numbers of DNN layers is shown in Figure 7. We can see that increasing layers does not necessarily bring better performance as it may overfit. Based on the experiments, we chose a four layer network. It contains 2 hidden layers with 40 nodes in each layer and a softmax output layer. The curves in both Figure 6 and Figure 7 are the averaged performance numbers over the 10-fold evaluation.

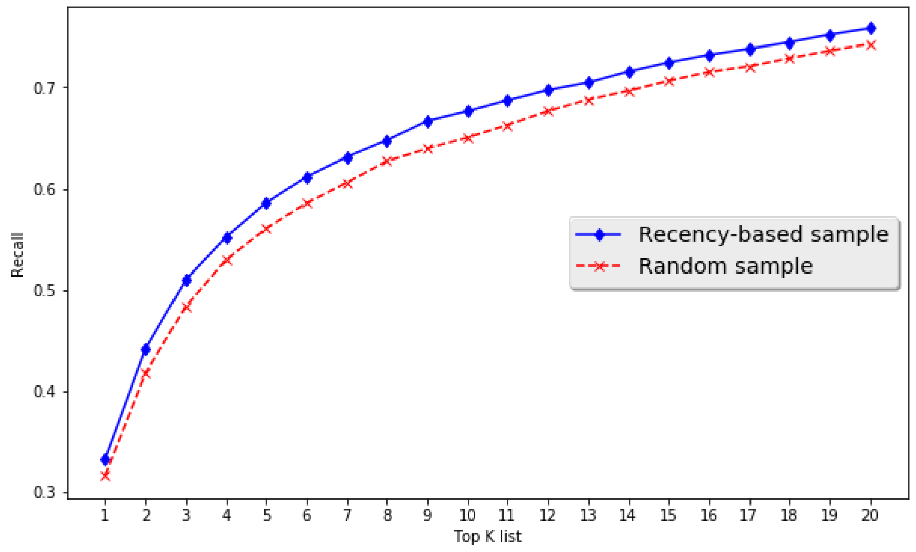
Our experiments are carried out using a Dell Precision T7810 workstation. The machine has Intel Xeon 2.4G CPU and 32GB memory, running with 64-bit Ubuntu 16.04. The average running time is about 1.1 seconds with the hierarchical classification. This ensures us to provide a service with 2 seconds turnaround time, even with some overhead in network and user interface interactions.

4. Related Work

Bug component prediction belongs to the general problem of bug triage, which deals with the problem of assigning bugs to appropriate developers. There has been considerable research on bug triage; they can be generally divided into two categories: location based and activity based Shokripour et al. (2013). Location based approaches Ye et al. (2014); Linares-Vásquez et al. (2012); Zhou et al. (2012) identify bug assignee by locating the source code which might produce the bug and the developers of the code pieces. They depend on bug localization, which may not be quite reliable. Lee et al. Lee et al. (2018) has reported a comprehensive study on the performance of six state-of-the-art bug localization techniques



(a) Super-component result.



(b) Sub-component result.

Figure 5: Performance using the recency based sampling (blue line) vs. the baseline random undersampling (red dashed line). We compare the performance on both individual classifiers (without hierarchical refining), measured by recall@K ranked list for $K = [1, 2, ..., 20]$. The Y-axis is the recall value and the X-axis is the value of K.

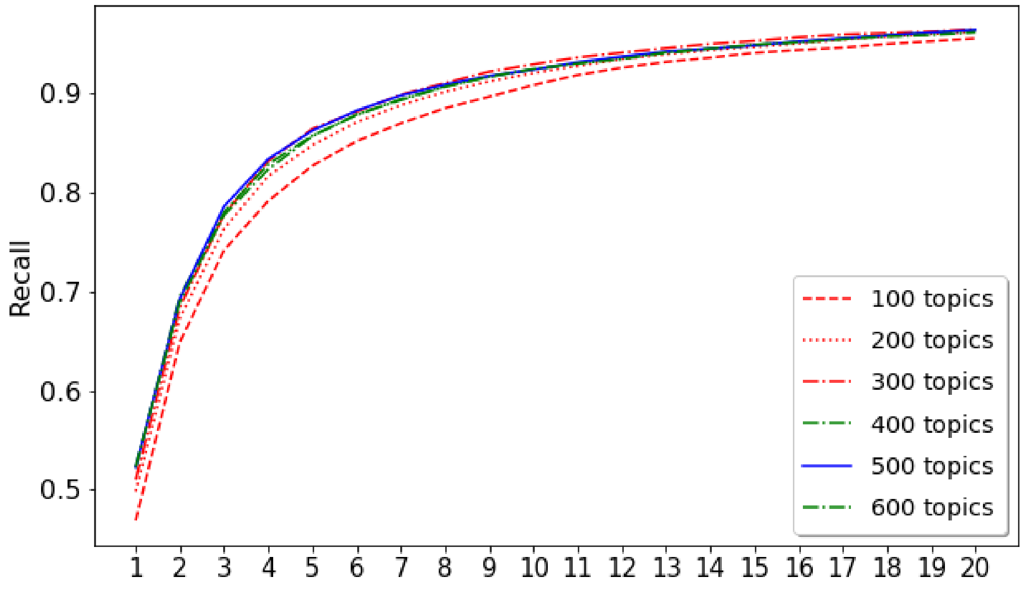


Figure 6: Performance comparison with different numbers of topics, measured by recall@K ranked list for $K = [1, 2, ..., 20]$. The X axis is the value of K.

based on information retrieval, using about 10k bug reports. Activity based approaches Xia et al. (2015); Anvik and Murphy (2011); Murphy and Cubranic (2004) assign bugs based on the developers' bug fixing history. It is treated as a text classification problem in Murphy and Cubranic (2004); Anvik and Murphy (2011). They build text feature based on the frequency of words and learn a predictive model using classification algorithms, e.g., Naive Bayes Classifier. Tian et al. Tian et al. (2016) instead treat is as a ranking problem based on the similarity between bug reports and documents. The activity based approaches do not utilize the source code updating history, which could be quite informative about who created the bug. Therefore, Tian et al. Tian et al. (2016) proposed combining the location based approach with the activity based approach. Those existing work were usually evaluated using relatively small data sets ($\sim 10k$ bug reports) from Open Source projects and were often based on manually designed features. The reported accuracy were $30\% \sim 45\%$ depending on data sets.

Jonsson et al. Jonsson et al. (2016) explored a large number of industry bugs. They used an ensemble of classifiers to achieve better results. In its largest "automation" data set with about 15k bugs, it obtained $\sim 50\%$ accuracy with 10-fold cross-validation. Lee et al. Lee et al. (2017) proposed to apply deep learning to bug triage and they worked with industrial bugs as well. They reported $\sim 31\%$ accuracy on a data set with 142 active developers (the number of classes is comparable with the number of sub-components in our data set), evaluated based on 10-fold cross-validation as well. Our colleagues have implemented this approach and tested for developer prediction as well, they also obtained $\sim 30\%$ accuracy based on 10 fold cross-validation. When trying to evaluate using future bugs (based on the model learned from past data), a significant accuracy drop (20%) is

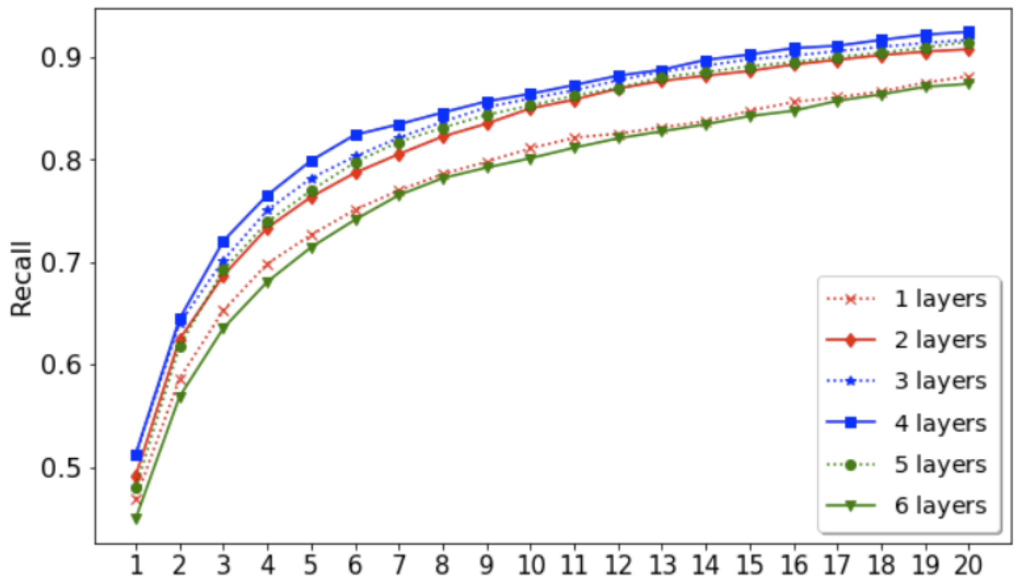


Figure 7: Performance comparison with different numbers of layers in the DNN network, measured by recall@K ranked list for $K = [1, 2, ..., 20]$.

reported by Jonsson et al. (2016). This is not surprising as previous research Rao et al. (2008) has warned about evaluating performance using cross-validation. When doing cross-validation, the training and test set are from the same period of time and obey the same distribution, while for bug reports data sets, future bug distribution usually will not be the same as the past. Consequently, cross-validation may overestimate the accuracy and may not be the suitable metric for the problem. We also found that the test accuracy based on future bugs is considerable lower than that based on 10 fold cross-validation. Nevertheless, we have achieved promising accuracy numbers when predicting components for future bugs.

For industry applications, assigning a bug to a component makes more sense than assigning to individual developers directly, as there are always personnel changes in a product life cycle. Nevertheless, the component based assignment also introduces the problem of mixed labels of different levels, which we addressed with the proposed hierarchical framework. The problem does not happen in the previous work, which only have one level of assignee.

As a fundamental challenge to machine learning, data imbalance has also been studied in the literature, mainly in the two class classification scenario. Kubat et al. Kubat et al. (1997) tackled the problem by down-sampling the majority class while keeping the minority class. Zhang et al. (2017) address the imbalance problem in large scale data by adaptive sampling the majority and minority classes. Ling et al. Ling and Li (1998) proposed to combine over-sampling of the minority class with under-sampling of the majority class. The Synthetic Minority Over-sampling Technique (SMOTE) introduced by Chawla et al. (2002) proposed to create synthetic positive instances using interpolation so the data could be more balanced. It is widely used due to effectiveness and simplicity. We didn't pursue this direction since it is not clear whether we can reliably create synthetic bug data. The problem of learning under concept drift has also been actively investigated. Last Last (2002) has proposed to rebuild the classification model with the latest examples. Wang et al. Wang et al. (2003) proposed to use weighted ensembles of classifiers to mine data streams. There are also several survey papers on concept drift Ditzler et al. (2015); Gama et al. (2014).

Hierarchical recognition Koller and Sahami (1997); Sebastiani (2002) has been used in the classification scenario for years. However, they usually follow a traditional coarse to fine pipeline: first do classification using the super-level (group) classifier and select the most likely group. Then using one or more sub-level (fine grained) classifiers to determine the final label within the selected group. Consequently, if the super-level classification is wrong, the sub-level classifiers cannot correct the mistake. Also they have multiple sub-level classifiers (at least one for each group). Our framework differs significantly: it is not a sequential process, and we only have two classifiers which run in parallel. We have a bi-directional probability propagation algorithm instead of one way filtering. The sub-level classification helps to improve the super-level classification, as opposed to solely depending on it. The novel probability propagation scheme helps to improve both the super and sub-level recognition, with virtually no extra cost in computation. The traditional sequential process incurs more classification (prediction) time, in addition to training of many more classifiers.

5. Conclusion and Future Work

Assigning bugs to the relevant component or developers has been a daunting task for Quality Engineer and customer care personnel. In this paper, we present an approach for automatically assigning the responsible component based on the bug description. We use Natural Language Processing and Deep Neural Network to build component classifiers. We also propose a hierarchical classification framework to address the mixed ground truth label problem. By combining two semantically related classifiers, which are complementary to each other and can run in parallel, we achieve better accuracy with negligible overhead in computation time. In addition, we introduce a recency based sampling procedure to handle the data imbalance and concept drift problem. It takes advantage of the time dependent nature of the bug occurrence process. We have built a service based on the proposed approach. It works in almost real time with ~ 2 seconds service time using a regular workstation. The service has been tested by internal users and received positive feedbacks.

Our goal is to provide an AI assistant for people who handle bug assignment, so they can save the time of going through a long list of possible components. Based on our experiments and user feedbacks, the service works fairly reliably for this purpose. In addition to the web interface, it could also be integrated into the JIRA reporting interface and serve as an automated bug assignment agent. The service is updated regularly (e.g. monthly or even weekly) to accommodate newly added components and new trends in bug occurrence. This constant updating could be a nightmare for human operator. In our case, we only need to pull JIRA data periodically, retrain the classifiers and push them to the server.

In the future, we plan to combine the location based approaches with our work to enhance its performance, as suggested in Tian et al. (2016). We also want to explore ideas in the research field of one shot learning Fei-Fei et al. (2006), to address the difficulty of dealing with new component. Another direction we will investigate is the history of component assignment. Currently we use only the last assigned component as the ground truth label. It is not rare for a bug to be re-assigned multiple times before being assigned to the right component. We could do some kind of hard example mining Shrivastava et al. (2016) using the re-assigning history to improve the performance.

References

John Anvik and Gail C Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(3):10, 2011.

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

Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16: 321–357, 2002.

Gregory Ditzler, Manuel Roveri, Cesare Alippi, and Robi Polikar. Learning in nonstationary environments: A survey. *IEEE Computational Intelligence Magazine*, 10(4):12–25, 2015.

Li Fei-Fei, Rob Fergus, and Pietro Perona. One-shot learning of object categories. *IEEE transactions on pattern analysis and machine intelligence*, 28(4):594–611, 2006.

João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM computing surveys (CSUR)*, 46 (4):44, 2014.

Haibo He and Edwardo A Garcia. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering*, 21(9):1263–1284, 2009.

Leif Jonsson, Markus Borg, David Broman, Kristian Sandahl, Sigrid Eldh, and Per Runeson. Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts. *Empirical Software Engineering*, 21(4):1533–1578, 2016.

Daphne Koller and Mehran Sahami. Hierarchically classifying documents using very few words. *International Conference on Machine Learning*, 1997.

Miroslav Kubat, Stan Matwin, et al. Addressing the curse of imbalanced training sets: one-sided selection. *ICML*, pages 179–186, 1997.

Mark Last. Online classification of nonstationary data streams. *Intelligent data analysis*, 6 (2):129–147, 2002.

Jaekwon Lee, Dongsun Kim, Tegawendé F Bissyandé, Woosung Jung, and Yves Le Traon. Bench4bl: reproducibility study on the performance of ir-based bug localization. In *ACM SIGSOFT*, pages 61–72. ACM, 2018.

Sun-Ro Lee, Min-Jae Heo, Chan-Gun Lee, Milhan Kim, and Gaeul Jeong. Applying deep learning based automatic bug triager to industrial projects. In *11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2017.

Mario Linares-Vásquez, Kamal Hossen, Hoang Dang, Huzefa Kagdi, Malcom Gethers, and Denys Poshyvanyk. Triaging incoming change requests: Bug or commit history, or code authorship? In *ICSM*, 2012.

Charles X Ling and Chenghui Li. Data mining for direct marketing: Problems and solutions. 1998.

G Murphy and D. Cubranic. Automatic bug triage using text categorization. In *Proceedings of the Sixteenth International Conference on Software Engineering and Knowledge Engineering*, 2004.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

R Bharat Rao, Glenn Fung, and Romer Rosales. On the dangers of cross-validation. an experimental evaluation. In *International Conference on Data Mining*. SIAM, 2008.

Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.

Ramin Shokripour, John Anvik, Zarinah M Kasirun, and Sima Zamani. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In *Mining Software Repositories (MSR)*, pages 2–11, 2013.

Abhinav Shrivastava, Abhinav Gupta, and Ross Girshick. Training region-based object detectors with online hard example mining. In *CVPR*, pages 761–769, 2016.

Yuan Tian, Dinusha Wijedasa, David Lo, and Claire Le Goues. Learning to rank for bug report assignee recommendation. In *ICPC*, 2016.

Haixun Wang, Wei Fan, Philip S Yu, and Jiawei Han. Mining concept-drifting data streams using ensemble classifiers. In *ACM SIGKDD*, pages 226–235, 2003.

Xin Xia, David Lo, Xinyu Wang, and Bo Zhou. Dual analysis for recommending developers to resolve bugs. *Journal of Software: Evolution and Process*, 27(3):195–220, 2015.

Xin Ye, Razvan Bunescu, and Chang Liu. Learning to rank relevant files for bug reports using domain knowledge. In *ACM SIGSOFT*, pages 689–699. ACM, 2014.

Wei Zhang, Said Kobeissi, Scott Tomko, and Chris Challis. Adaptive sampling scheme for learning in severely imbalanced large scale data. In *Asian Conference on Machine Learning*, pages 240–247, 2017.

Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *34th International Conference on Software Engineering (ICSE)*, pages 14–24. IEEE, 2012.