CosTriage: A Cost-Aware Triage Algorithm for Bug Reporting Systems

Jin-woo Park, Mu-Woong Lee, Jinhan Kim, Seung-won Hwang
POSTECH, Korea, Republic of
{jwpark85,sigliel,wlsgks08,swhwang}@postech.edu

Sunghun Kim
HKUST, Hong Kong
hunkim@cse.ust.hk

Abstract

'Who can fix this bug?' is an important question in bug triage to "accurately" assign developers to bug reports. To address this question, recent research treats it as a optimizing recommendation accuracy problem and proposes a solution that is essentially an instance of content-based recommendation (CBR). However, CBR is well-known to cause *over-specialization*, recommending only the types of bugs that each developer has solved before. This problem is critical in practice, as some experienced developers could be overloaded, and this would slow the bug fixing process. In this paper, we take two directions to address this problem: First, we reformulate the problem as an optimization problem of both accuracy and cost. Second, we adopt a content-boosted collaborative filtering (CBCF), combining an existing CBR with a collaborative filtering recommender (CF), which enhances the recommendation quality of either approach alone. However, unlike general recommendation scenarios, bug fix history is extremely *sparse*. Due to the nature of bug fixes, one bug is fixed by only one developer, which makes it challenging to pursue the above two directions. To address this challenge, we develop a topic-model to reduce the sparseness and enhance the quality of CBCF. Our experimental evaluation shows that our solution reduces the cost efficiently by 30% without seriously compromising accuracy.

1 Introduction

Bug reporting (or issue tracking) systems play an important role in the software development process. Through bug reporting systems, many bug reports are posted, discussed, and assigned to developers. For example, a big software project like Mozilla receives more than 300 reports per day (Anvik 2007).

Running a bug reporting system thus has many challenges including, and bug triage is one of the main challenges. Yet most triage tasks including bug assignment heavily rely on manual effort, which is labor intensive and potentially error prone (Jeong, Kim, and Zimmermann 2009). For example, miss-assignment causes unnecessary bug reassignments and slows the bug fixing process (Jeong, Kim, and Zimmermann 2009; Guo et al. 2010).

For this challenge, a state-of-the-art approach (Anvik, Hiew, and Murphy 2006) treats bug triage as a recommendation problem and proposes a solution that can be viewed as an instance of content-based recommendation (CBR). We will denote this as `PureCBR`, which learns from previous bug assignment patterns and estimates each developer's success possibilities for fixing reported bugs.

However, in a general recommendation problem, CBR is well-known to suffer from *over-specialization*. In other words, it recommends only the types of bugs that each developer has solved in the past. This problem is more critical in practice, as bug assignments will be skewed to few experienced developers, which will significantly delay the fix time for these bugs, as they will be assigned to overloaded developers. As an extreme example, consider a system that always assigns a bug to a single experienced developer who has fixed many types of bugs. This system would achieve high accuracy, but fail to delegate this load to other underutilized developers and thus slow down the bug fixing process.

This observation leads to two new goals: *First,* we need to reformulate the bug triage problem to optimize not only accuracy but also cost. *Second,* we need to consider a hybrid approach, more specifically, a content-boosted collaborative filtering (CBCF) approach (Melville, Mooney, and Nagarajan 2002), combining an existing CBR with a collaborative filtering recommender (CF). CBCF approach has been reported to perform better than either approach alone.

A key challenge in achieve both goals is *sparseness*. For the first goal of optimizing cost, we need to estimate the cost of developer $\mathcal{D}$ fixing bug $\mathcal{B}$. However, the past history only reports this cost for a single developer who solved $\mathcal{B}$, that is, one bug is fixed by one developer. This makes it challenging to estimate the cost for the remaining developers. For the second goal of leveraging a CF, sparseness is also an obstacle, as CF requires the collection of information from developers who have worked on the same bug (collaborating), which cannot be done using past history. As only one developer was selected to work on each bug, this approach is essentially infeasible.

To overcome this sparseness problem, we propose a cost-aware triage algorithm, CosTriage. CosTriage models "developer profiles" to indicate developers' estimated costs for fixing different types of bugs. "Types" of bugs are extracted by applying Latent Dirichlet Allocation (LDA) (Blei, Ng, and Jordan 2003) to bug report corpora. This addresses the sparseness problem and enhances the recommendation quality of CBCF.

Using developer profiles, whenever a new bug is reported, we first determine the type of the bug, and obtain each developer's estimated cost for the bug. Then by combining the cost with the possibilities estimated by `PureCBR`, we rank the developers and assign the first developer to the bug.

We experimentally evaluate CosTriage using four real-life bug report corpora[1,2,3,4], in comparison with two baselines: (1) `PureCBR`, and (2) `CBCF`. Our experimental results show that CosTriage significantly reduces cost, without seriously significantly sacrificing accuracy.

2 Approach

To pursue the dual goals of accuracy and cost, we use two historical datasets $\mathcal{H}_F$ and $\mathcal{H}_C$. $\mathcal{H}_F$ consists of the pairs of $\langle F(\mathcal{B}), \mathcal{D}\rangle$, where $F(\mathcal{B})$ is the feature vector of a given bug $\mathcal{B}$, and $\mathcal{D}$ is the developer who fixed $\mathcal{B}$. $\mathcal{H}_C$ is an m-by-n matrix of costs for m developers and n fixed bugs. As a proof-of-concept, we use bug fix time as cost.

For a reported bug, using $\mathcal{H}_F$, we identify *who can fix the bug*. Meanwhile, using $\mathcal{H}_C$, we find *how much it will cost for each developer to fix the bug*. Finally, combining the both results, we rank the developers and choose one to be assigned.

2.1 Baseline I: `PureCBR`

Anvik's algorithm (Anvik, Hiew, and Murphy 2006), `PureCBR`, considers bug triage as a multi-class classification problem, and utilizes only $\mathcal{H}_F$. For each data record in $\mathcal{H}_F$, $F(\mathcal{B})$ is the set of keywords extracted from the title, descriptions, and other metadata contained in the bug report $\mathcal{B}$. The developer $\mathcal{D}$, who actually fixed this bug, is the class of this data record.

After training a multi-class classifier, when a new bug is reported, `PureCBR` estimates each developer's score for the new bug by extracting its feature vector and applying the classifier. `PureCBR` simply picks the developer with the highest score to assign the reported bug. It thus neglects the costs of developers to fix the bug.

2.2 Baseline II: `CBCF`

Adopting `PureCBR` is a sufficient solution to utilize $\mathcal{H}_F$, but it disregards $\mathcal{H}_C$. We need to utilize both $\mathcal{H}_F$ and $\mathcal{H}_C$.

For a new reported bug $\mathcal{B}$, to be cost aware, we estimate each developer's cost for fixing $\mathcal{B}$. Even though this problem seems to be related to CF problems, pure CF algorithms do not work for bug triage data, because we have no known cost for fixing $\mathcal{B}$. However, by training one classifier of cost for each developer, CBR or CBCF can be applied. We choose to apply CBCF because it outperforms CBR in general (Melville, Mooney, and Nagarajan 2002).

To combine CBCF-based cost estimation with `PureCBR`, we introduce the following four-step hybrid procedure:

1. For each developer, a support vector machine (SVM) classifier is trained using the bug features from the bugs fixed by the developers and the time spent to fix that as the class.
2. When a new bug $\mathcal{B}$ is reported, developers' cost vector $\mathcal{L}_C$ is estimated. An empty column for $\mathcal{B}$ is inserted to $\mathcal{H}_C$, and each empty cell of $\mathcal{H}_C$ is filled with the pseudo cost estimated by the corresponding classifier trained in the first step. We then apply column-column CF (developer-developer CF) to $\mathcal{H}_C$, to revise the pseudo costs of $\mathcal{B}$.
3. `PureCBR` is performed to obtain the vector $\mathcal{L}_S$ of developers' success possibilities.
4. $\mathcal{L}_C$ and $\mathcal{L}_S$ are merged into one vector $\mathcal{L}_H$. Developers are ranked in descending order of $\mathcal{L}_H$, then the first developer is assigned to $\mathcal{B}$.

The first and the second steps of the procedure exactly follow Melville's CBCF algorithm (Melville, Mooney, and Nagarajan 2002).

In the rest of the paper, we will denote this four-step approach as `CBCF`.

Merging $\mathcal{L}_C$ and $\mathcal{L}_S$ When we reach the last step, $\mathcal{L}_C$ and $\mathcal{L}_S$ are merged as follows. Formally, $\mathcal{L}_C$ and $\mathcal{L}_S$ are:

$\mathcal{L}_C = \langle s_{\text{C}[1]}, s_{\text{C}[2]}, \cdots, s_{\text{C}[n]}\rangle$, $\mathcal{L}_S = \langle s_{\text{S}[1]}, s_{\text{S}[2]}, \cdots, s_{\text{S}[n]}\rangle$,

where $s_{\text{C}[i]}$ means the i^{th} developer's estimated cost to fix the given bug $\mathcal{B}$, and $s_{\text{S}[i]}$ denotes the i^{th} developer's success possibility for fixing $\mathcal{B}$, obtained by `PureCBR`.

To merge the vectors and obtain a hybrid score vector $\mathcal{L}_H$,

$$\mathcal{L}_H = \langle s_{\text{H}[1]}, s_{\text{H}[2]}, \cdots, s_{\text{H}[n]}\rangle,$$

a hybrid score $s_{\text{H}[i]}$ of a developer is computed as a weighted arithmetic mean of the normalized $s_{\text{S}[i]}$ and $s_{\text{C}[i]}$:

$$s_{\text{H}[i]} = \alpha \cdot \frac{s_{\text{S}[i]}}{\max(\mathcal{L}_S)} + (1-\alpha) \cdot \frac{1/s_{\text{C}[i]}}{1/\min(\mathcal{L}_C)}, \quad (1)$$

where $0 \leq \alpha \leq 1$. In Equation 1, we take the inverse of $s_{\text{C}[i]}$ because a small $s_{\text{C}[i]}$ value means high performance, *i.e.*, short estimated time to fix $\mathcal{B}$. Both $s_{\text{S}[i]}$ and $1/s_{\text{C}[i]}$ are normalized to have a maximum value of 1. This hybrid computation involves one control parameter, α, to trade-off accuracy with cost.

2.3 Proposed Approach: CosTriage

`CBCF` tries to utilize both accuracy and cost. However `CBCF` has room for improvement, because bug fix history is extremely sparse. We thus devise developer profiles, which are compact and unsparse. Our new triage algorithm, CosTriage, basically follows the four steps of `CBCF`, but we replace the first two steps with a new procedure using the developer profiles.

Constructing Developer Profiles A *developer profile* is a numeric vector, in which each element denotes the developer's estimated cost for fixing a certain bug type. The *bug type* means that if some bugs belong to the same type, these bugs share very similar features. More formally, a developer profile $\mathcal{P}_u$ is a $\mathcal{T}$-dimensional vector of numeric scores $p_{u[i]}$:

$$\mathcal{P}_u = \langle p_{u[1]}, p_{u[2]}, \cdots, p_{u[\mathcal{T}]}\rangle, \quad (2)$$

[1]Apache, `https://issues.apache.org/bugzilla/`
[2]Eclipse, `https://bugs.eclipse.org/bugs/`
[3]Linux kernel, `https://bugzilla.kernel.org/`
[4]Mozilla, `https://bugzilla.mozilla.org/`

Table 1: An LDA model of bug reports for Mozilla. $\mathcal{T} = 7$. Top-10 representative words with the highest probabilities.

Topic 1		Topic 2		Topic 3		Topic 4		Topic 5		Topic 6		Topic 7	
c	0.058	mozilla	0.047	js	0.028	window	0.023	html	0.026	windows	0.042	bug	0.036
line	0.058	firefox	0.024	error	0.018	page	0.019	text	0.018	mozilla	0.042	bugzilla	0.012
cpp	0.040	add	0.014	mozilla	0.018	click	0.016	table	0.013	gecko	0.029	cgi	0.009
int	0.032	version	0.010	file	0.014	menu	0.014	document	0.011	rv	0.028	code	0.008
mozilla	0.032	update	0.010	test	0.011	button	0.013	image	0.010	results	0.025	id	0.008
src	0.026	thunderbird	0.010	function	0.011	open	0.013	style	0.010	user	0.023	bugs	0.008
const	0.023	file	0.008	ns	0.010	text	0.012	content	0.009	build	0.020	time	0.007
unsigned	0.020	files	0.008	c	0.010	dialog	0.010	page	0.009	message	0.019	patch	0.007
bytes	0.019	added	0.007	chrome	0.009	select	0.009	type	0.009	nt	0.019	set	0.005
builds	0.018	install	0.007	content	0.009	search	0.009	id	0.009	firefox	0.018	fix	0.005

where $p_{u[i]}$ denotes the developer's cost for i^{th}-type bugs and $\mathcal{T}$ denotes the number of bug types.

To model developer profiles, we need to address two major problems: (1) how to categorize bugs to determine bug types, and (2) how to obtain the profile of each developer.

◇ *Categorizing Bugs*: One naive way to define types of bugs is to assume that each reported bug belongs to its own specific type. This means that if N distinct bugs have been reported, N types of bugs exist, *i.e.*, $\mathcal{T} = N$. Clearly this approach is not desirable for our problem, because the dimensionality of the profiles would be both too large and unbounded. Therefore, we devise a method to define a small and limited number of bug types, adopting LDA.

The basic idea of LDA is that a document is a mixture of latent *topics*, and that each topic is characterized by a distribution of words. For a given number of topics $\mathcal{T}$ and a corpus of documents, where each document is a sequence of words, LDA extracts $\mathcal{T}$ topics from the corpus. After the topics are extracted, the LDA model of these topics can be used to determine the distribution of topics for each document.

To apply LDA, one main question is "what is the natural number of topics in a corpus?" Although it may be explicitly answered, obtaining such information is often difficult. To address this challenge, we adopt Arun's divergence measure (Arun et al. 2010), because it has more robust behaviors than others (Cao et al. 2009; Zavitsanos et al. 2008).

In the context of bug reports, for using LDA, each bug report can be considered as a document, which consists of words that are explicitly stated in bug reports. Table 1 illustrates an LDA model of 48,424 Mozilla bug reports.

Each topic in an LDA model is represented as probabilities that words will occur. For instance, Topic 1 of the Mozilla project consists of 5.8% *c*, 5.8% *line*, 4.0% *cpp*, 3.2% *int*, and 3.2% *mozilla*. Then, to determine the type of a given bug report (document), we compute probabilities that words will occur in the report, and choose the topic with the most similar probability distribution. That is, for a given bug report, the bug is an i^{th}-type bug if its most related topic is Topic i in the LDA model.

◇ *Obtaining Profiles*: After determining bug types, for each developer $\mathcal{D}_u$, we quantify each value $p_{u[i]}$ of the developer's profile $\mathcal{P}_u$ as the average time to fix i^{th} type bugs. We only consider the bugs which were fixed by $\mathcal{D}_u$.

Table 2 shows example profiles constructed from Mozilla bug reports. On average, developer $\mathcal{D}_1$ took 14.28 days to fix 4^{th}-type bugs, and 5.44 days to fix 6^{th}-type bugs. In clear contrast, $\mathcal{D}_2$ took 1.75 days to fix 4^{th}-type bugs, and 12.90 days to fix 6^{th}-type bugs. This suggests that these developer profiles clearly reveal the differences between developers.

Table 2: Sparseness of developer profiles obtained from bug reports for Mozilla project. There are many blanks in the profiles, and this may incur some overspecialization problems. The 2^{th}-type bugs will always be assigned to Developer $\mathcal{D}_1$. Likewise, to $\mathcal{D}_4$, only 6^{th} or 7^{th}-type bugs are assigned.

	Bug types						
Dev	1	2	3	4	5	6	7
$\mathcal{D}_1$	7.73	1.71	8.59	14.28	7.54	5.44	8.45
$\mathcal{D}_2$	8.18	-	3.50	1.75	4.00	12.90	13.18
$\mathcal{D}_3$	11.56	-	60.50	23.50	-	2.67	19.20
$\mathcal{D}_4$	-	-	-	-	-	22.40	20.75

A key challenge in developer profile modeling is sparseness. LDA partially addresses sparseness by condensing the data. For the Mozilla corpus, we now know the cost of each bug type for 406 different developers (from 1165 developers in total) rather than just one cost per bug. However, the challenge still remains. Some values in $\mathcal{P}_u$ cannot be determined, if the developer $\mathcal{D}_u$ did not fix some types of bugs. The profiles in Table 2 thus have some blanks with undetermined costs. We thus discuss how to further address this sparseness problem by filling blanks.

Predicting Missing Values in Profiles To fill in the blanks in developers' profiles, we use CF. For a missing value $p_{u[i]}$ in a profile $\mathcal{P}_u$, we predict its value as an aggregation of the corresponding values in the profiles of other developers:

$$p_{u[i]} = F(\mathcal{P}_u) \times \frac{\sum_{\forall \mathcal{P}_v \in N_u} \mathcal{S}(\mathcal{P}_u, \mathcal{P}_v) \cdot \left(\frac{p_{v[i]}}{F(\mathcal{P}_v)} \right)}{\sum_{\forall \mathcal{P}_v \in N_u} \mathcal{S}(\mathcal{P}_u, \mathcal{P}_v)}, \quad (3)$$

where N_u is a set of neighborhood developers' profiles, and $\mathcal{S}()$ denotes the similarity between two profiles. $F(\mathcal{P}_u)$ is a scaling factor, which is computed as the maximum known value in $\mathcal{P}_u$, to normalize each profile. In this aggregation process, we only consider the profiles $\mathcal{P}_v \in N_u$ which have $p_{v[i]}$, *i.e.*, we ignore the profiles with missing $p_{v[i]}$.

The similarity between two profiles is computed as their cosine similarity:

$$\mathcal{S}(\mathcal{P}_u, \mathcal{P}_v) = \frac{\mathcal{P}_u \cdot \mathcal{P}_v}{||\mathcal{P}_u|| ||\mathcal{P}_v||} \times \omega(\mathcal{P}_u, \mathcal{P}_v), \quad (4)$$

where $||\cdot||$ denotes its $\mathcal{L}^2$ norm (Euclidean length). To compute the dot product, $\mathcal{P}_u \cdot \mathcal{P}_v$, and the $\mathcal{L}^2$ norms, $||\mathcal{P}_u||$ and $||\mathcal{P}_v||$, we only consider dimensions in which the values exist in both profiles.

In Equation 4, $\omega(\mathcal{P}_u, \mathcal{P}_v)$ is a *significance weighting factor*, which is a simple but very effective way to prevent over estimation of the similarities (Herlocker, Konstan, and Riedl 2002; Ma, King, and Lyu 2007). The significance weighting factor is computed as:

$$\omega(\mathcal{P}_u, \mathcal{P}_v) = \min\left(\frac{|\mathcal{P}_u \cap \mathcal{P}_v|}{\Theta}, 1\right), \quad (5)$$

where Θ is a given threshold, and $|\mathcal{P}_u \cap \mathcal{P}_v|$ denotes the number of dimensions in which both profiles have known values. If only a small number of dimensions have values in both profiles, considering only those dimensions would devalue the similarity between two profiles.

Based on this similarity notion, to determine the set N_u of neighborhood profiles, we pick the top-k most similar profiles to $\mathcal{P}_u$. Then by aggregating N_u, we can fill up the blanks in $\mathcal{P}_u$. To illustrate, Table 3 shows the example profiles introduced in Table 2, after we fill up the blanks using CF.

Table 3: Developer profile examples introduced in Table 2. Missing values were filled up using CF.

	Bug types						
Dev	1	2	3	4	5	6	7
$\mathcal{D}_1$	7.73	1.71	8.59	14.28	7.54	5.44	8.45
$\mathcal{D}_2$	8.18	6.40	3.50	1.75	4.00	12.90	13.18
$\mathcal{D}_3$	11.56	27.22	60.50	23.50	40.52	2.67	19.20
$\mathcal{D}_4$	13.18	12.99	10.97	11.41	15.14	22.40	20.75

After we fill up all blanks in the profiles, for a given bug report, we can find out all developers' estimated costs, $\mathcal{L}_C$, by determining the most relevant topic using the LDA model. Then `PureCBR` is also performed to obtain $\mathcal{L}_S$, and we merge $\mathcal{L}_C$ and $\mathcal{L}_S$ into $\mathcal{L}_H$, in the same way of `CBCF`.

3 Experimental Results

In this section, we discuss our experimental setup and present our results. We design our experiments to address two research questions:

Q1 How much can our approach improve cost (bug fix time) without sacrificing bug assignment accuracy?

Q2 What are the trade-offs between accuracy and cost (bug fix time)?

3.1 Setup

We evaluate our approach using bug reports from four open source projects: Apache, Eclipse, Linux kernel, and Mozilla.

To model developer profiles, we compute the time spent to fix a given bug, which is explicitly observable. We use a common equation (Kim and Whitehead 2006) to compute bug fix time for a fixed bug b in reports:

$$\text{bug fix time of } b \text{ (in days)} = \\ \text{fixed time of } b - \text{assigned time of } b + 1 \text{ day}, \quad (6)$$

where 'fixed time' is the time when fixing is complete, and 'assigned time' is the time of the first assignment to the fixer. These fix and assignment time data are recorded in bug report history.

However, we are not able to compute the fix time for the non-fixed bugs and bugs with no explicit assignment information. We consider them as *invalid bugs*, and only use *valid bugs* with assignment information in our experiments.

Like `PureCBR`, we consider only *active* developers. Mozilla has 1,165 developers, but about 637 developers have fixed less than 10 bug reports in 12 years. We do not consider these *inactive* developers, since they do not have sufficient information to build their profiles.

To filter out inactive developers, we use the interquartile range (IQR) of bug fix numbers of each developer (Han and Kamber 2006), *i.e.*,

$$IQR = (\text{the third quartile}) - (\text{the first quartile}).$$

Higher distributions than IQR can be considered as active because IQR is essentially the measure of central distributions. We thus considered developers whose bug fix number is higher than IQR as *active developers*.

In total, we extracted 656 reports and 10 active developers from Apache, 47,862 reports and 100 active developers from Eclipse, 968 reports and 28 active developers from Linux kernel, and 48,424 reports and 117 active developers from Mozilla as shown in Table 4.

Bug Feature Extraction We extract features from bug reports in a similar way to `PureCBR`. First we extract features from bug report metadata such as version, platform, and target milestone. We also extract features (*i.e.*, words) from the bug report text description. We preprocess the text description to remove unnecessary words such as stop words, numerics, symbols and words whose length is more than 20 characters.

To train an SVM model for these reports for `PureCBR`, we convert these bug reports into the pairs of $\langle F(\mathcal{B}), \mathcal{D}\rangle$ as mentioned in Section 2.1. Each report has a feature vector which consists of extracted words and their occurrence counters. We obtained 6,915, 61,515, 11,252 and 71,878 features from Apache, Eclipse, Linux kernel and Mozilla corpora respectively.

We adopt a widely used SVM implementation *SVM-light*[5] with a linear kernel. We empirically tuned parameter c to 1000 with the best performance. We set other parameters to default values.

Developer Profiles Modeling To model developer profiles, we first perform LDA on valid bug reports in each project. Since the LDA results vary based on the $\mathcal{T}$ value,

[5]http://svmlight.joachims.org/

Table 4: Subject systems

Projects	# Fixed bug reports	# Valid bug reports	# Total developers	# Active developers	# Bug types	# Words	Period
Apache	13,778	656	187	10	19	6,915	2001-01-22 - 2009-02-09
Eclipse	152,535	47,862	1,116	100	17	61,515	2001-10-11 - 2010-01-22
Linux kernel	5,082	968	270	28	7	11,252	2002-11-14 - 2010-01-16
Mozilla	162,839	48,424	1,165	117	7	71,878	1998-04-07 - 2010-01-26

it is important to select the "optimal" value. To choose $\mathcal{T}$, we adopt the divergence measure proposed in (Arun et al. 2010), as we discussed in Section 2.3. The optimal $\mathcal{T}$ minimizes the divergence between the document-word matrix and the topic-word matrix. Formally:

$$\mathcal{T} = \underset{t}{\operatorname{argmin}}\ KL\_Divergence(t) \tag{7}$$

Such $\mathcal{T}$ corresponds to the lowest $KL\_Divergence$ value. In our experiments, the optimal $\mathcal{T}$ are 19, 17, 7, and 7 for Apache, Eclipse, Linux kernel, and Mozilla projects' bug report corpora respectively.

After determining the optimal number of topics, $\mathcal{T}$, of each bug report corpus, we identify the bug type of each valid bug report. We evaluate the scores of each topic in a report, and then determine the report's bug type as the topic which has the highest score. Scores are the sum of extracted word distributions related to each topic respectively, from the title and the description in the bug report. For bug reports which have no topic words, we assign the reports' bug types randomly.

We then model the developer profiles of active developers. We first evaluate each developer profile (Table 2). However, we could not compute all values, because some developers had never been assigned certain types of bugs as mentioned in Section 2.3. In this case, we predict missing values using CF and complete modeling profiles of active developers (as illustrated in Table 3).

To evaluate the effectiveness of our profile modeling, we compute the relative error of expected bug fix time, using the four corpora. COSTRIAGE generally outperforms `CBCF`, especially for the case of the Apache corpus (Table 5).

Table 5: The relative error of expected bug fix time. We use the first 80% of bug reports in the four corpora as a training set and the rest as a test set to compute the error.

	Apache	Eclipse	Linux	Mozilla
CBCF	51.23	10.62	25.31	10.67
COSTRIAGE	19.18	16.66	9.25	8.76

Evaluation To evaluate the algorithms, we use two measures: $Accuracy = \frac{|W|}{|N|}$, and

$$\text{Average bug fix time} = \frac{\sum_{\forall w_i \in W} \{\text{bug fix time of } w_i\}}{|W|},$$

where W is the set of bug reports predicted correctly, and N is the number of bug reports in the test set. Because the "real" fix time for mismatched bugs is unknown, we only use the fix time for correctly matched bugs to compute the average fix time for a given set of bugs.

We use the first 80% of bug reports as a training set and the remaining 20% as a test set to conduct our experiments.

3.2 Results

This section presents our experimental results and addresses the research questions raised at the beginning of this section.

Improving Bug Fix Time We apply COSTRIAGE, `CBCF`, and `PureCBR` to our subjects, and compare accuracy and bug fix time of the three approaches.

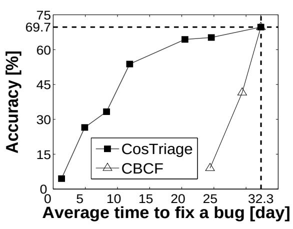
Table 6 shows the comparison results. Using `PureCBR`, the accuracy is 69.7% for Apache. On average, a developer takes 32.3 days to fix one bug for Apache. By controlling the parameter α for each of `CBCF` and COSTRIAGE, we match the accuracy of both `CBCF` and COSTRIAGE, to fairly compare their cost savings. For the Apache corpus, in one setting, both `CBCF` and COSTRIAGE sacrifice about 5.5% of accuracy, *i.e.*, the both methods show 65.9% accuracy, but COSTRIAGE significantly outperforms `CBCF` in terms of cost. While `CBCF` only reduces the cost by 2.2%, COSTRIAGE reduces it by more than 30%.

Table 6: Average bug fix time and accuracy using `PureCBR`, and reducing ratio of time and accuracy using `CBCF` and COSTRIAGE.

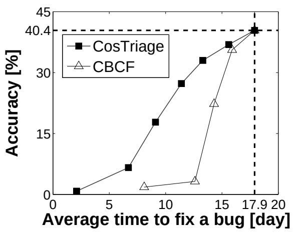
			Reducing ratio			
	PureCBR		CBCF		COSTRIAGE	
Project	Time	Acc	Time	Acc	Time	Acc
Apache	32.3	69.7	-2.2%	-5.5%	-31.0%	-5.5%
Eclipse	17.9	40.4	-5.6%	-5.0%	-10.6%	-5.0%
Linux	55.3	30.9	-24.9%	-4.9%	-28.9%	-4.9%
Mozilla	11.3	64.3	-4.4%	-5.0%	-7.1%	-5.0%

Trade-off between Accuracy and Bug Fix Time In this section, we observe the trade-offs between bug assignment accuracy and bug fix time using COSTRIAGE compared to `CBCF`.

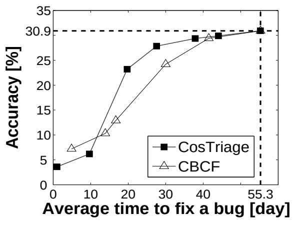
We apply COSTRIAGE and `CBCF` to the four subjects with varying α to observe the trade-offs between accuracy and average bug fix time. Figure 1 shows the relation graphs



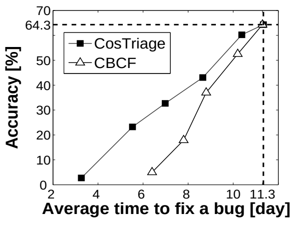
(a) Apache



(b) Eclipse



(c) Linux kernel



(d) Mozilla

Figure 1: The trade-offs between accuracy and bug fix time

between time and accuracy. The x-axis represents the average time to fix one bug, and the y-axis represents bug-assignment accuracy.

The range of the fix time is 1.3 to 32.3 days, while the accuracy is 4.5 to 69.7% for Apache. Eclipse has a fix time of 3.1 to 17.9 days, when the accuracy is 0.3 to 40.4%. The fix time is 1.0 to 55.3 days at an accuracy of 3.6 to 30.9% for Linux kernel. Mozilla has a fix time of 3.3 to 64.3 days, while the accuracy is 2.7 to 64.3%.

These relation graphs in Figure 1 show clear trade-offs between accuracy and average bug fix time for bug triage. As accuracy decreases, average bug fix time decreases. It is possible to sacrifice the accuracy to reduce the average bug fix time. However, we observe that the average bug fix time drops much more quickly than the accuracy as shown in these graphs. That is, COSTRIAGE significantly reduces cost without sacrificing much accuracy. Meanwhile, in comparison with COSTRIAGE, CBCF sacrifices accuracy much more to reduce cost in most cases.

These trade-offs of COSTRIAGE can be used in various scenarios in practice. For example, if a manager wants to fix bugs relatively quickly, she can tune COSTRIAGE to reduce average bug fix time. These clear trade-offs of COSTRIAGE provide flexibility for triagers who want to assign bug reports to developers accurately or efficiently.

4 Conclusions

We proposed a new bug triaging technique, COSTRIAGE, by (1) treating the bug triage problem as a recommendation problem optimizing both accuracy and cost and (2) adopting CBCF combining two recommender systems. A key challenge to both techniques is the extreme sparseness of the past bug fix data. We addressed the challenge by using a topic-model to reduce the sparseness and enhanced the quality of CBCF. Our experiments showed that our approach significantly reduces the cost without significantly sacrificing accuracy. Though we used a proof-of-concept implementation by using bug fix time as cost, our developer profile model is general enough to support other code indicators such as interests, efforts, and expertise to optimize for both accuracy and cost for automatic bug triage.

Our algorithm is not restricted to only bug triage problems. Any recommendation techniques involving two objectives can potentially adopt our algorithm. For example, identifying experts for a question in QA (Question & Answer) systems is a problem of matching a question to an expert, and our algorithm can utilize both side needs of questioner and answerer.

Acknowledgement

This research was supported by National IT Industry Promotion Agency (NIPA) under the program of Software Engineering Technologies Development, the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / National Research Foundation of Korea (NRF) (Grant 2011-0000979), and the 2010 Microsoft Software Engineering Innovation Foundation Awards.

References

Anvik, J.; Hiew, L.; and Murphy, G. C. 2006. Who should fix this bug? In *ICSE '06*.

Anvik, J. 2007. *Assisting Bug Report Triage through Recommendation*. Ph.D. Dissertation, University of British Columbia.

Arun, R.; Suresh, V.; Madhavan, C. E. V.; and Murthy, M. N. N. 2010. On finding the natural number of topics with latent dirichlet allocation: Some observations. In *PAKDD '10*.

Blei, D. M.; Ng, A. Y.; and Jordan, M. I. 2003. Latent dirichlet allocation. In *The Journal of Machine Learning Research '03*.

Cao, J.; Xia, T.; Li, J.; Zhang, Y.; and Tang, S. 2009. A density-based method for adaptive lda model selection. In *Neurocomputing*.

Guo, P. J.; Zimmermann, T.; Nagappan, N.; and Murphy, B. 2010. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *ICSE '10*.

Han, J., and Kamber, M. 2006. *Data Mining: Concepts and Techniques*.

Herlocker, J.; Konstan, J. A.; and Riedl, J. 2002. An empirical analysis of design choices in neighborhood-based collaborative filtering algorithms. In *Information Retrieval '02*.

Jeong, G.; Kim, S.; and Zimmermann, T. 2009. Improving bug triage with bug tossing graphs. In *ESEC/FSE '09*.

Kim, S., and Whitehead, Jr., E. J. 2006. How long did it take to fix bugs? In *MSR '06*.

Ma, H.; King, I.; and Lyu, M. R. 2007. Effective missing data prediction for collaborative filtering. In *SIGIR '07*.

Melville, P.; Mooney, R. J.; and Nagarajan, R. 2002. Content-boosted collaborative filtering for improved recommendations. In *AAAI '02*.

Zavitsanos, E.; Petridis, S.; Paliouras, G.; and Vouros, G. A. 2008. Determining automatically the size of learned ontologies. In *ECAI '08*.