Information and Software Technology 70 (2016) 68–84

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

KSAP: An approach to bug report assignment using KNN search and heterogeneous proximity

Wen Zhang [a,*], Song Wang [b], Qing Wang [c]

[a] School of Economics and Management, Beijing University of Chemical Technology, Beijing, 100019, P. R. China
[b] Department of Electrical and Computer Engineering at University of Waterloo, Ontario, Canada
[c] Laboratory for Internet Software Technologies, Institute of Software, Chinese Academy of Sciences, Beijing 100190, PR China

ARTICLE INFO

Article history:
Received 11 May 2015
Revised 16 October 2015
Accepted 16 October 2015
Available online 26 October 2015

Keywords:
Bug report assignment
Heterogeneous network
Heterogeneous proximity
Developer recommendation

ABSTRACT

Context: Bug report assignment, namely, to assign new bug reports to developers for timely and effective bug resolution, is crucial for software quality assurance. However, with the increasing size of software system, it is difficult to assign bugs to appropriate developers for bug managers.
Objective: This paper propose an approach, called KSAP (K-nearest-neighbor search and heterogeneous proximity), to improve automatic bug report assignment by using historical bug reports and heterogeneous network of bug repository.
Method: When a new bug report was submitted to the bug repository, KSAP assigns developers for the bug report by using a two-phase procedure. The first phase is to search historically-resolved similar bug reports to the new bug report by K-nearest-neighbor (KNN) method. The second phase is to rank the developers who contributed to those similar bug reports by heterogeneous proximity.
Results: We collected bug repositories of Mozilla, Eclipse, Apache Ant and Apache Tomcat6 projects to investigate the performance of the proposed KSAP approach. Experimental results demonstrate that KSAP can improve the recall of bug report assignment between 7.5–32.25% in comparison with the state of art techniques. When there is only a small number of developer collaborations on common bug reports, KSAP has shown its excellence over other sate of art techniques. When we tune the parameters of the number of historically-resolved similar bug reports (K) and the number of developers (Q) for recommendation, KSAP keeps its superiority steadily.
Conclusion: This is the first paper to demonstrate how to automatically build heterogeneous network of a bug repository and extract meta-paths of developer collaborations from the heterogeneous network for bug report assignment.

1. Introduction

Software projects commonly adopt a bug repository such as Bugzilla and JIRA to manage bug reports during development and maintenance. When a bug of the software was found, a bug report is created and submitted to the bug repository. A bug report usually describes the details of the found bug such as its reproduction procedure and use context. Then, a project member, called bug triager or bug manager, will examine the submitted bug report and make a decision as to whom the new bug report should be assigned for its resolution. This procedure, called bug report assignment, has great impact on quality of the software. An incorrect bug report assignment would increase the time taken for fixing a bug [1] and therefore increase the cost of the project [2].

On the one hand, with the increasing size and complexity of current software systems, more and more bugs were submitted to the bug repository of the project by developers and users. For example, about 200 bug reports were filed to the Eclipse bug repository per day close to its release dates, and for the Debian project, this number is about 150 [3]. On the other hand, with the prevalence of globally distributed software developer teams, it is difficult to find appropriate developers to resolve these bugs for bug triagers [8]. For instance, about two person-hours per day have to be spent on bug triage in the Eclipse project and nearly 25% of Eclipse bug reports were reassigned due to inaccurate manual bug assignment [3]. Thus, there is a rising interest in automating the process of bug report assignment [2–5].

After a bug was fixed, it will be marked as “resolved” or “fixed” in the open bug repository. Motivated by this observation, bug report assignment is typically modeled as a classification problem as labeling the target bug report with a developer as done by Anvik et al. [4] and Cubranic et al. [5]. However, we hold that in fact a bug resolution coming into being is with contribution of a group of developers.

* Corresponding author. Tel.: +86 13552250923.
E-mail addresses: zhangwen@mail.buct.edu.cn (W. Zhang), wangsong@uwaterloo.ca (S. Wang), wq@itechs.iscas.ac.cn (Q. Wang).

http://dx.doi.org/10.1016/j.infsof.2015.10.004
0950-5849/

Fig. 1. A snapshot of the bug report for bug #333160 of Mozilla project.

For instance, the Mozilla bug report #333160 "Document the Recently Closed Tabs menu and the keyboard shortcut"[1] as shown in Fig. 1 involves 4 developers' collaborative contribution with 19 comments in discussion. We observed that on average, each Mozilla bug report involves no less than 3 developers' contribution. Each Eclipse bug report involves no less than 5 developers' conversations on average [3]. Thus, we regard that bug reports are resolved with collaborative contribution of developers rather than "one developer dominating the whole bug report resolution" [6]. Following this line of motivation, we have conducted a previous study on using social network analysis to model the collaboration of developers in bug resolution [7].

However, two problems are still remaining for more work. First, a homogenous network is incapable of modeling developers' collaboration on bug resolution [9–11]. Second, the clique method, which links all developers who contributed to a common bug report, is unsuitable for network modeling of developer collaboration. The problem with the clique method is that many "exceptional" developers participated in crowding activities but they seldomly participated in non-crowding bug report resolution [11]. In this paper, we refer to a bug report as a crowding bug report if it has crowding (i.e. the upper three quartiles) participants and we refer it as a non-crowding bug report if it has a small number (usually the first quartile) of participants. Using the clique method, it seems that those "exceptional" developers contribute to resolution of a large number of bug reports because they have a large number of links to other developers. However, in fact, those "exceptional" developers merely contributed a small number of crowding bug reports. In real practice of bug report resolution, two developers (dev_1 and dev_2) can collaborate with each other via different activities [12]. For instance, one developer (dev_1) can toss a bug report to another developer (dev_2) [1] or make a comment for a bug report (br_1) which is under the same component as another bug report (br_2) meanwhile br_2 is commented by another developer (dev_2).

In this paper, we propose an approach, called KSAP, to improve the state of art bug report assignment approaches using KNN search and heterogeneous proximity. For each new incoming bug report, KSAP firstly searches similar bug reports from historically-resolved bug reports, and then ranks the developers who contributed to those similar bug reports for recommendation. We claim the contribution of this paper mainly in three aspects. Firstly, we propose a method to construct a heterogeneous network of a bug repository in Section 3. Secondly, we develop KSAP for automatic bug report assignment based on heterogeneous proximity in Section 4. Thirdly, we conduct experiments to examine the performances of KSAP compared with the state of art techniques on automatic bug report assignment in Section 5. Section 6 presents threats. Section 7 presents related work and Section 8 concludes the paper.

2. Background

2.1. Bug report

A bug report consists of many predefined meta-fields and free-form textual contents. Predefined meta-fields describe the basic attributes of a bug report such as "report id", "reporter", "product id", "component id", etc. "product id" stands for the product in which the bug was found, and "component id" stands for the specific component of the product. The free-form textual content of a bug report refers to the natural language description of the bug, including description of the bug report and comments posted by developers.

Fig. 1 shows Mozilla bug #333160. The meta-fields are surrounded with red dotted line. The developer who reported the bug and the description of the bug are surrounded with purple dotted line. The developers who made comments to the bug report are surrounded with green dotted line. The comments are surrounded with blue dotted line. We regard that the developers who reported and made comments to the bug report are interested in the bug report and have potential in resolving the bug. Thus, the problem of bug report assignment can be simplified as assigning new bug reports to the interested and potential developers in the project.

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=333160.

2.2. Problem formulation

With the above brief introduction of bug report and its components, we define four core concepts involved in bug report assignment as follows. We extend the previous work [7] by introducing meta-fields and differentiated bug reporter and commenters in the definition.

Definition 1. Bug report br_i: a bug report br_i can be represented by using a quadruple as $br_i = \{m_i, d_i, dev_i^{(r)}, dl_i^{(c)}\}$, where m_i denotes the meta-fields of br_i, d_i denotes the document of br_i, $dev_i^{(r)}$ denotes the developer who reported the bug and $dl_i^{(c)}$ denotes the commenter list of br_i containing the developers who commented to br_i.

Definition 2. Meta-fields m_i of br_i: we use its component c_i and product p_i as two elements of m_i i.e. $m_i = \{c_i, p_i\}$.

Definition 3. Commenter list $dl_i^{(c)}$ of bug report br_i: for a historical bug report br_i, its commenter list $dl_i^{(c)} = \{dev_{i,1}^{(c)}, \ldots, dev_{i,|dl_i^{(c)}|}^{(c)}\}$ contains the developers who make comments to the bug report br_i. For a new bug report, its developer list $dl_i^{(c)}$ is empty.

Definition 4. Document d_i of bug report br_i: for a historical bug report br_i, we use its description ds_i and comments $ct_i = \{ct_{i,1}, \ldots, ct_{i,|ct_i|}\}$ posted by its commenters $dl_i^{(c)}$ to denote its document d_i, i.e. $d_i = \{ds_i, ct_i\}$. Note that a developer $dev_{i,j}$ can make more than one comments for br_i. For a new bug report br_{new}, we used its description ds_{new} to denote its document, i.e. $d_{new} = \{ds_{new}\}$.

With the above definition, the problem of bug report assignment can be simplified as the following. Given historical bug reports $\{br_1, \ldots, br_i, \ldots, br_m\}$ (m is the size of bug report collection) in the open bug repository and a new bug report br_{new}, we need to find Q (a predefined number) developers$dl_{new} = \{dev_{new,1}, \ldots, dev_{new,Q}\}$ (where $dl_{new} \subseteq dl_1 \cup \ldots \cup dl_i \cup \ldots \cup dl_m$ and $|dl_{new}| = Q$) who are capable of contributing to br_{new} resolution.

3. Heterogeneous network of bug repository

Traditional homogenous network is composed of merely single type of nodes (referred to as entities hereafter) and single type of relations between the nodes, such as the nodes as papers and the relation as "cite" in citation network and the nodes as emails and the relation as "send to" in email network [22]. However, a heterogeneous developer network contains multiple types of nodes, such as developers, bugs, comments, components, and products, and multiple types of relations between these nodes [21]. In this section, we construct a heterogeneous network of a bug repository and extract developers' collaboration from the heterogeneous network. For consistence, we use nodes and entities in heterogeneous network interchangeably.

3.1. Entities and relations

We consider 5 types of entities in our heterogeneous network: developer, bug, comment, component, and product. For abbreviation, we use their first capital letters to denote these entities, namely D for developers, B for bug reports, C for components, and P for products. We use T to denote comments in order to distinguish the abbreviation of component and comment.

During software development, the entities in bug repositories interact with each other frequently. These interactions denote various relations between entities. Specifically, the interaction between developer and comment can be expressed by the relations "write" and "written by" (denoted as write^{-1}); bug and comment by "comment" and "commented by" (denoted as comment^{-1}); component and bug, product and component by "contain" and "contained in" (denoted as contain^{-1}). Previous studies [13,23] show that interactions between developers and bugs have multiple relations. We use "report/assign/toss/fix/close/reopen" to denote the multiple relations from developers to bugs and "report^{-1}/assign^{-1}/toss^{-1}/fix^{-1}/close^{-1}/reopen^{-1}" to denote the multiple relations from bugs to developers.

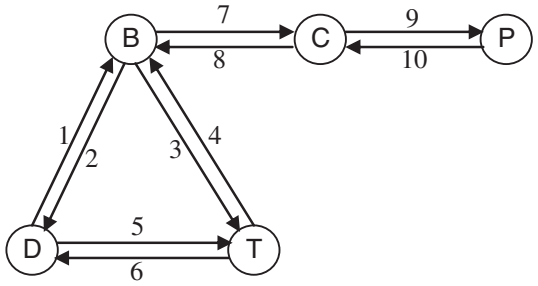

Fig. 2. Schema of entities and relations of a bug repository.

Table 1
The 10 types of relations between the 5 types of entities in the schema.

Type no.	Connotation list	Cardinality
1	report|assign|toss|fix|close|reopen|be assigned to|be tossed	1:n
2	report^{-1} |assign^{-1}|toss^{-1}|fix^{-1}|close^{-1}|reopen^{-1}|be assigned to|be tossed to	1:1
3	comment^{-1}	1:n
4	comment	1:1
5	write	1:n
6	write^{-1}	1:1
7	contain^{-1}	1:1
8	contain	1:n
9	contain^{-1}	1:1
10	contain	1:n

Fig. 2 illustrates the schema derived from our analysis of the entities and their relations in a bug repository. In the schema, nodes denote entities, and edges denote relations between entities. Entities can connect with each other via different meta-paths, e.g., two developers can be connected via a "developer–bug–developer" path (D–B–D), "developer–comment–bug–comment–developer" path (D–T–B–T–D) and so on.

Table 1 shows the 10 types of relations between the 5 types of entities in Fig. 2. The first column corresponds to the type number, the second column describes the connotation of the relation and the third column describes the cardinality of the two entities. For instance, the relation type No. 1 D→B, with size of the connotation list as 8 and the cardinality as 1:n, refers to that one (1) developer can report|assign|toss|fix|close|reopen|be assigned to|be tossed more than one (n) bug reports. The relation type No. 2 B→D, also with size of connotation list as 8 and the cardinality as 1:1, refers to that a bug report (1) can be (report^{-1} |assign^{-1}|toss^{-1}|fix^{-1}|close^{-1}|reopen^{-1}) by or be assigned |tossed to only one (1) developer. By analogy, all the other relations can be explained explicitly.

3.2. Building heterogeneous network

In order to build the heterogeneous network HN of a bug repository, we need to parse each bug report and link the 5 types of entities within it. Taking the bug report in Fig. 1 as an example, developers involved are "Mark Pilgrim", "Robert Strong", "Mike Beltzner", "Mike Conner" and "timeless". The bug report is "#333160", its component is "Installer" and its product is "Firefox". The comments are the 4 textual contents surrounded with blue dotted line. Table 2 shows the

Table 2
The activity log of bug report #333160 of Mozilla project.

Who	When	What	Removed	Added
Mark Pilgrim	2006-10-12 11:30:02	Assignee	Mark Pilgrim	Robert Strong
		Status	ASSIGNED	NEW
Robert Strong	2006-10-18 16:38:25	Assignee	Robert Strong	Mike Beltzner
Mike Beltzner	2006-10-26 12:02:22	Status	New	Assigned
Mike Beltzner	2007-01-03 16:44:14	Priority	P3	P5
Mike Beltzner	2007-03-01 16:27:31	Status	Assigned	Resolved
		Resolution	–	Fixed

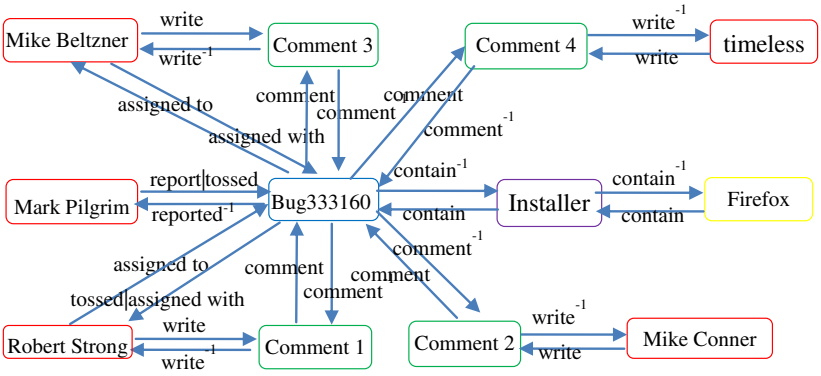

Fig. 3. The heterogeneous network of Mozilla bug report #333160.

activity log of Mozilla bug report #333160, we see that "Mark Pilgrim" assigned this bug to himself and then tossed it to "Robert Strong", and later "Robert Strong" tossed this bug to "Mike Beltzner".

From the network schema, we firstly parse the bug report to extract the developers who report and comment the bug report, as well as its component and product. We then parse its activity log to extract the relation "reports|tosses|assignes|fixes|closes| reopens|(is assigned to)|(is tossed) " between developers. Fig. 3 shows the constructed heterogeneous network using Mozilla bug report #333160. We use red boxes to denote developers, blue box for bug report, green boxes for comments, purple box for component and yellow box for product.

To formalize the above process, we represent a bug report br_i as $br_i = \{\{c_i, p_i\}, \{ds_i, \{ct_{i,1}, \ldots, ct_{i,|ct_i|}\}\}, dev_i^{(r)}, \{dev_{i,1}^{(c)}, \ldots, dev_{i,|dl_i^{(c)}|}^{(c)}\}\}$ based on the definitions in Section 2.2. Furthermore, we extract the developers from the activity log[2] of br_i as $dl_i^{(a)} = \{dev_{i,1}^{(a)}, \ldots, dev_{i,|dl_i^{(a)}|}^{(a)}\}$. With these expressions, the procedure of building a heterogeneous network for a bug repository can be described in Fig. 4.

The basic idea of the algorithm in Fig. 4 is that for each bug report, we enumerate all instances of the entities and their relations shown in Fig. 2 and append them into the heterogeneous network HN one by one. Lines 2–5 are used to add the developer who submitted the bug report to the heterogeneous network HN. Lines 6–10 add the developers who changed the bug report status to the heterogeneous network HN. Lines 12–17 add the comments of the bug report br_i and commenters to the heterogeneous network HN. Lines 19–24 add the component and product to the heterogeneous network HN.

3.3. Developers' collaboration extraction

All the two developers' collaboration can be described by a meta-path which starts from *D* and ends with *D* within the network schema shown in Fig. 2. For instance, D–B–D can be used to characterize the collaboration that one developer (D) reports|tosses|assignes |fixes|closes|reopens|(is assigned to)|(is tossed) a bug (B) which is reported|tossed|assigned|fixed| closed|reopened by or is assigned|tossed to another developer (D) .

With this intuition, we adopt graph traverse [17] to produce all the paths between D and D in the network schema. We remove paths that include trivial sub-paths such as B–T–B, C–B–C, etc., in building the heterogeneous developer network. The reason is that sub-path B–T–B stands for a bug (B) is commented by a comment (T) and meanwhile the same comment (T) also comments on another bug (B). That means the comment (T) is with two bugs at the same time. This is a rare case because a comment (T) can comment on only one bug. For the sub-path C–B–C, it is impossible that a bug report belongs to two components at the same time. Thus, we use Pruning Rule 1 to remove all the unlikely paths when producing all the paths starting from D and ending with D from the network schema.

Pruning Rule 1. If there are two relations X→Y and Y→X in a path, and the relation X→Y has size of connotation list as 1 and proportion as 1:*n*, and the relation Y→X has size of connotation list as 1 and proportion as 1:1, then the path X–Y–X is pruned for further consideration.

The algorithm for finding all the meta-paths between D and D in the network schema is illustrated in Fig. 5. The input of the algorithm is a starting node, an ending node and the network schema shown in Fig. 2. In this context, both starting and ending nodes are the same as D. Lines 1–7 are the function generatePath that generates all the paths that start from startNode and end with endNode in the network schema. Lines 8–14 are the function generatePathR that recursively generates paths from path to the endNode. Lines 15–26 are the function oneStepPath that returns all the paths which are of one step from the given path and startNode. We should notice that the function isPruned(temPath) was used to implement Pruning Rule 1 to decide whether or not a path is possible. Once all one-step paths from the startNode are generated, we need to decide whether or not each ending node of these paths is equal to endNode. If the answer is positive, then we find a path that starts from startNode and ends with endNode. Otherwise, we followed the path to walk on the

[2] We only retained the developers who changed status of the bug report in activity log.

```
==========================================
Input: the bug reports in the bug repository, i.e. { br1 ,…, bri ,…,
       brm } and its activity logs;
Output: a heterogeneous network HN;
==========================================
1   for each bri in { br1 ,…, bri ,…, brm } {
2      nodeB = new node( bri , B);
3      nodeD = new node( dev_i^(r) , D);
4      linkBD = new link(nodeB,nodeD,1);HN.add(linkBD);
5      linkDB = new link(nodeD,nodeB,2);HN.add(linkDB);
6      for each dev_{i,j}^(a) in {dev_{i,1}^(a),...,dev_{i,|dt_i^(a)|}^(a)} {
7         nodeD = new node( dev_{i,j}^(a) , D);
8         linkBD = new link(nodeB,nodeD,1);HN.add(linkBD);
9         linkDB = new link(nodeD,nodeB,2);HN.add(linkDB);
10     }
11for each ct_{i,j} in {ct_{i,1},...,ct_{i,|ct_i|}} {
12        nodeT = new node( ct_{i,j} ,T);
13        linkBT = new link(nodeB,nodeT,3);HN.add(linkBT);
14        linkTB = new link(nodeT,nodeB,4);HN.add(linkTB);
15        nodeD = new node( dev_{i,j}^(c) , D);
16        linkDT = new link(nodeB,nodeT,5);HN.add(linkDT);
17        linkTD = new link(nodeT, nodeB,6);HN.add(linkTD);
18     }
19     nodeC = new node( c_i ,C);
20     linkBC = new link(nodeB, nodeC,7);HN.add(linkBC);
21     linkCB = new link(nodeC, nodeB,8); HN.add(linkCB);
22     nodeP = new node( p_i ,C);
23     linkCP = new link(nodeC, nodeP,9); HN.add(linkCP);
24     linkPC = new link(nodeP, nodeC,10);HN.add(linkPC);
25 }
==========================================
```

Fig. 4. The algorithm of building a heterogeneous network for a bug repository.

```
==========================================
Input: startNode, endNode, schema
Output: pathList
==========================================
1   generatePath(startNode, endNode, schema) {
2      tempPathList = oneStepPath(null,startNode,schema);
3      for each path in tempPathList
4      if path.endNode==endNode
5            pathList.add(path);
6      else generatePathR(path, endNode, graph);
7}
------------------------------------------------------------------------
8   generatePathR(path,endNode, schema){
9    tempPathList = oneStepPath(path, path.endNode, schema);
10   for each path in tempPathList
11    if path.endNode==endNode
12                pathList.add(path);
13          else generatePath(path,endNode,schema);
14}
------------------------------------------------------------------------
15   oneStepPath(path, startNode, schema){
16      tempPathList = new List;
17      if path==null
18           for each link in schema
19                if link.startNode==startnode
20   tempPathList.add(link)
21      else for each link in schema
22           if link.endNode==path.startNode
23tempPath = combine(path,link);
24if(!isPruned(tempPath))
25    tempPathList.add(tempPath);
26      return tempPathList;}
==========================================
```

Fig. 5. The algorithm for generating meta-paths between entities from the network schema.

```
====================================
Input: meta-path D-B-D, developer pair (dev_i,dev_j),
       heterogeneous network HN
Output: A set of instances of meta path D-B-D InSet.
====================================
1 for each link l in HN
2   if l.startNode is dev_i and l.endNode.type is B
3     for each link l' in HN
4       if l.endNode is l'.startNode and l'.endNode is dev_j
5           InSet.add(new MPInstance(l,l',1));
====================================
```

Fig. 6. The algorithm of extracting instances of meta-path D-B-D from the heterogeneous network HN.

network step by step recursively until all the paths are either ending with endNode or pruned.

After finding all the paths starting from D and ending with D in the network schema, we also need to eliminate repetitive paths such as D–B–T–D and D–T–B–D because that a developer D reports|tosses|assignes|fixes|closes| reopens|(is assigned with)|(is tossed with) a bug report (B) that has a comment T written by another developer D is of the same meaning as that a developer D writes a comment T for a bug report B reported|tossed|assigned|fixed|closed| reopened by or assigned|tossed to another developer D. Thus we use the Pruning Rule 2 shown below to eliminate all the repetitive paths.

Pruning Rule 2. For two paths X–Y–…–Z and Z–…–Y–X, if they are of same length, and all links in one path X–Y–…–Z are reversely ordered in another path Z–…–Y–X, then we call the two paths repetitive and eliminate either of them in output path list.

Table 3 lists the 9 meta-paths output from the algorithm in Fig. 5 using D as startNode and endNode. Their meaning is explained in the second column. We use the 9 meta-paths to extract developer collaboration in a bug repository. Firstly, we regulate a developer as the starting node, and use one of the meta-paths in Table 3 for guidance. Secondly, we traverse the whole graph to find all the paths those are instances of the guided meta-path. Taking the network shown in Fig. 3 for an example, by the meta-path "D-B-D", we found the collaboration among "Mike Beltzner", "Mark Pilgrim" and "Robert Strong" because "Mark Pilgrim" reported Mozilla bug report #333160 and the bug report was then tossed to "Robert Strong" and next to "Mike Beltzner". By the meta-path "D–B–T–D", we found the collaboration between one of "Mike Beltzner", "Mark Pilgrim" and "Robert Strong", and one of "Mike Conner" and "timeless" because "Mike Conner" and "timeless" commented on the bug report. By the meta-path "D–T–B–T–D", we found the collaboration among "Mike Beltzner", "Robert Strong", "Mike Conner" and "timeless" because they four developers commented on the bug report.

Specifically, for the meta-path D–B–D, we firstly find dev_i (D) in HN, and then traverse all the entity nodes as bugs (B) adjacent to dev_i in HN, and finally find dev_j (D) adjacent to the bugs (B) in HN. Consequently, all the instances of meta-path D–B–D starting from dev_i and ending with dev_j are extracted from the heterogeneous network HN. We can describe this procedure in Fig. 6. Line 2 is used to find the links of D–B in the heterogeneous network HN. Line 3 to Line 4 is used to find the links of B–D in the heterogeneous network HN. Line 5 is used to compose the instances of meta-path D–B–D and add them to an instance set InSet. Here, new MPInstance(l, l',1) means an instance that is of type 1 in Table 3 and comprises two links l (D–B) and l' (B–D). By analogy, we can adapt the algorithm in Fig. 6 to extract all the instances from the heterogeneous network for all the meta-paths listed in Table 3.

Table 3
Meta-paths used in building heterogeneous developer network.

No.	Meta-path	Meaning of the relation
1	D–B–D	One developer (D) reports|tosses|assignes|fixes|closes| reopens|(is assigned with)|(is tossed with) a bug (B) that was reported|tossed|assigned|fixed|closed| reopened by or is assigned|tossed to another developer (D)
2	D–B–T–D	One developer (D) reports|tosses|assignes|fixes|closes| reopens|(is assigned with)|(is tossed with) a bug (B) that has comment (T) made by another developer (D)
3	D–T–B–T–D	One developer (D) make a comment (T) for a bug (B) which has another comment (T) made by another developer (D)
4	D–B–C–B–D	One developer (D) reports|tosses|assignes|fixes|closes|reopens|(is assigned with)|(is tossed with) a bug (B) of a component (C) and another bug (B) of the same component(C) was reported|tossed|assigned|fixed|closed|reopened by or is assigned|tossed to another developer (D)
5	D–B–C–B–T–D	One developer (D) reports|tosses|assignes|fixes|closes|reopens|(is assigned with)|(is tossed with) a bug (B) of a component (C) and another bug (B) of the same component(C) has a comment (T) made by another developer (D)
6	D–T–B–C–B–T–D	One developer (D) make a comment (T) for a bug (B) which is of the same component (C) as another bug (B) that has a comment (T) made by another developer (D)
7	D–B–C–P–C–B–D	One developer (D) reports|tosses|assignes|fixes|closes|reopens|(is assigned with)|(is tossed with) a bug (B) of a product (P) and another bug (B) of the same product(P) is reported|tossed|assigned|fixed closed|reopened by or is assigned|tossed to another developer (D)
8	D–B–C–P–C–B–T–D	One developer (D) reports|tosses|assignes|fixes|closes|reopens|(is assigned with)|(is tossed with) a bug (B) of a product (P) and another bug (B) of the same product(P) has a comment (T) made by another developer (D)
9	D–T–B–C–P–C–B–T–D	One developer (D) make a comment (T) for a bug (B) which is of the same product (P) as another bug (B) that has a comment (T) made by another developer (D)

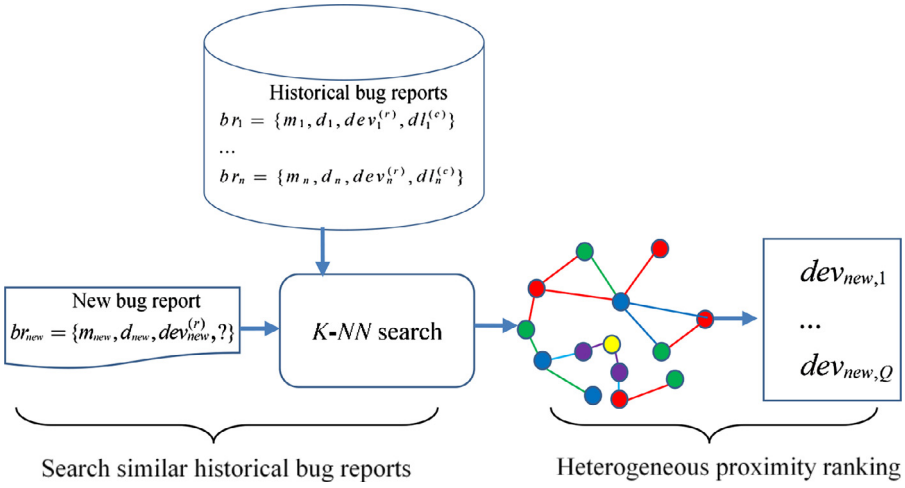

Fig. 7. The overview of KSAP.

4. KSAP – the proposed approach

This section proposes KSAP that utilizes K-nearest-neighbor search and heterogeneous proximity for automatic bug report assignment. The overview of KSAP is shown in Fig. 7.

4.1. K-nearest-neighbor search

K-nearest-neighbor (K-NN) classification is a type of instance-based lazy learning that classifies objects based on the top K similar training objects in the feature space [14]. For simplicity, we adopt the basic idea of K-NN classification to search the top K similar historical bug reports of a new bug report to establish the similar bug report set of the new bug report. The cosine similarity is used to measure similarity of the document vector of the new bug report d_{new} and the document vector of a historical bug report d_i as defined in Section 2.2.

Natural language processing (NLP) is employed to transfer documents of bug reports d_i into numeric vectors that can be processed by K-nearest-neighbor search. The processing includes two steps: document indexing and term weighting. Tokenization, stop word elimination and stemming are employed to preprocess the documents of both historical and new bug reports and vector space model [15] is employed to index the document contents using the terms occurring in the documents. The 100 stop words from USPTO (United States Patent and Trademark Office) patent full-text and image database[3] is used for stop word elimination. Porter stemming algorithm was used for English word stemming processing[4]. TF-IDF that is extensively studied in our previous research [16] is used for term weighting.

When a new bug report arrives, KSAP firstly transfers the new bug report into a document vector $d_{new} = \{w_{new,1}, \ldots, w_{new,|d_{new}|}\}$. Then, the documents of historical bug reports in the open bug repository are used to compute cosine similarities against the document vector of the new bug. Finally, the predefined number (K) of historical bug reports, whose documents are of the top K similarities with d_{new}, are extracted from the open bug repository to construct the similar bug report set of br_{new}. That is, $SimSet(br_{new}, k) = \{br_1^{similar}, \ldots, br_k^{similar}\}$.

4.2. Heterogeneous proximity ranking

To derive the Q developers as formulated in Section 2.2, we firstly extract the attached developers from the historical bug reports in $SimSet(br_{new}, k)$, i.e. to construct a candidate developer list $dl^{similar} = dl_1^{similar} \cup \ldots \cup dl_k^{similar}$. Those attached developers are the commenters in $dl_i^{(c)}$. Although in most cases all the developers in $dl_i^{(c)}$ and $dl_i^{(a)}$ together contributed to the bug resolution, we found

[3] http://ftp.uspto.gov/patft/help/stopword.htm.
[4] http://tartarus.org/~martin/PorterStemmer/.

that some developers who are wrongly assigned with the bug report tossed the bug report back to the bug manager. Thus, those wrongly assigned developers actually do not contribute to bug resolution. This is reason that we only use the developers in $dl_i^{(c)}$ as candidates for recommendation.

In addition to document similarity, we also consider matching meta-fields of new bug reports with that of historical bug reports. With the guidance of meta-paths shown in Table 3 and the Algorithm shown in Fig. 6, we can easily extract the instances of the meta-paths from the heterogeneous network HN. For each developer dev_i in the candidate developer list $dl^{similar}$, we enumerate the developer against all the other developers dev_j in $dl^{similar}$ to extract all the collaboration between dev_i and all the developers in $dl^{similar}$ using the 9 meta-paths listed in Table 3.

We can see from Table 3 that the meta-paths can be categorized into 3 types based on developers' collaboration. The first type is that two developers collaborate on common bugs (No. 1, 2 and 3). The second type is that two developers collaborate on common components (No. 4, 5 and 6). The third type is that two developers collaborate on common products (No. 7, 8 and 9). We indicate the type of meta-path instance in Line 5 of the algorithm shown in Fig. 6. Thus, after extracting instances of meta-paths among developers in $dl^{similar}$, we compute three types of heterogeneous proximity of two developers as follows:

$$HetPro^{(B)}(dev_i, dev_j) = Num_1(dev_i, dev_j) + Num_2(dev_i, dev_j) + Num_3(dev_i, dev_j) \quad (1)$$

Eq. (1) is used to measure the heterogeneous proximity of two developers dev_i and dev_j in $dl^{similar}$ on common bugs. Here, $Num_1(dev_i, dev_j)$ is the number of bug reports on which the two developers' collaboration relations belong to No.1 shown in Table 3. By analogy, $Num_2(dev_i, dev_j)$ and $Num_3(dev_i, dev_j)$ are the numbers of bug reports on which the two developers' collaboration relations belong to meta-paths No.2 and No.3, respectively.

$$HetPro^{(C)}(dev_i, dev_j) = \frac{1}{|S(c)|}(Num_4(dev_i, dev_j, c) + Num_5(dev_i, dev_j, c) + Num_6(dev_i, dev_j, c)) \quad (2)$$

Eq. (2) is used to measure the heterogeneous proximity of two developers dev_i and dev_j in $dl^{similar}$ on common components. Here, c is the component of the new bug report under consideration and $S(c)$ returns a set of bug reports belonging to component c in the bug repository. $Num_4(dev_i, dev_j, c)$, $Num_5(dev_i, dev_j, c)$ and $Num_6(dev_i, dev_j, c)$ denote the number of bugs on which the collaboration relations of dev_i and dev_j belong to meta-paths No.4, No.5 and No.6, respectively.

Compared to developers' collaboration on common bug reports, two developers dev_i and dev_j have higher possibility to collaborate on common components. Thus, we use $|S(c)|$ to normalize the importance of developers' collaboration on the common component c when measuring the heterogeneous proximity between dev_i and dev_j. We can deduce from Eq. (2) that, only under the condition that dev_i(D) reports|tosses|assignes|fixes|closes|reopens|(is assigned with)|(is tossed with) a bug report (B) of the component c and all the other bug reports (B) of the component c were reported|tossed|assigned| fixed|closed|reopened by dev_j(D), i.e. collaborated on all the reports of the component c resulting in$Num_4(dev_i, dev_j, c) = |S(c)|$, the heterogeneous proximity of dev_i and dev_j as $HetPro^{(C)}(dev_i, dev_j)$ is increased by 1.

$$HetPro^{(P)}(dev_i, dev_j) = \frac{1}{|S(p)|}(Num_7(dev_i, dev_j, p) + Num_8(dev_i, dev_j, p) + Num_9(dev_i, dev_j, p)) \quad (3)$$

Eq. (3) is used to measure the heterogeneous proximity of two developers dev_i and dev_j in $dl^{similar}$ on common products. Here, p is the product of the new bug report and $S(p)$ returns a set of bug reports belonging to product p in the bug repository. $Num_7(dev_i, dev_j, p)$, $Num_8(dev_i, dev_j, p)$ and $Num_9(dev_i, dev_j, p)$ denote the number of bug reports on which the collaboration relations of dev_i and dev_j belong to meta-paths No.7, No.8 and No.9 shown in Table 3, respectively. By analogy with Eq. (2), $|S(p)|$is used to normalize the importance of developers' collaboration on the common product p when measuring the heterogeneous proximity between dev_i and dev_j.

With the above equations to measure heterogeneous proximity of two developers on common bug reports, component c and product p, we compute the overall heterogeneous proximity of a developer dev_i in $dl^{similar}$ using Eq. (4). That is, we summarize the heterogeneous proximities of dev_i and all the other developers dev_j in $dl^{similar}$ on common bugs, component c and product$p$ to measure his or her overall heterogeneous proximity in $dl^{similar}$. Thus, heterogeneous proximity ranking for each developer in $dl^{similar}$ is derived by Eqs. (1)–(4) and used to rank developers' expertise on the incoming new bug report br_{new}.

The intuitive explanation of KSAP for bug report assignment is that when a new bug report is incoming, those developers assigned for its resolution should not only have similar expertise with the new bug report (using content matching using K-NN search) but also possess historical experience in collaboration with other developers in resolving bug reports of component c and product p of the new bug report.

$$HetPro(dev_i, dl^{similar}) = \sum_{dev_j \in dl^{similar}, dev_j \neq dev_i} HetPro^{(B)}(dev_i, dev_j) + \sum_{dev_j \in dl^{similar}, dev_j \neq dev_i} HetPro^{(C)}(dev_i, dev_j) + \sum_{dev_j \in dl^{similar}, dev_j \neq dev_i} HetPro^{(P)}(dev_i, dev_j) \quad (4)$$

5. Experiments

5.1. The datasets

To examine the effectiveness of KSAP in real practice, we collected the Mozilla[5] and Eclipse[6] bug repositories from MSR2011 website[7]. We also collected Apache Ant and Apache Tomcat6 bug reports (products are Ant and Tomcat6, respectively.) from ASF Bugzilla system[8]. The bug reports with status "resolved" and resolution as "fixed" and with no modification in the fields "Product" and "Component" are included in the experiments. We follow Guo et al. [18] to use 2.5 years as the time interval to collect bug reports in order to reduce possible changes in status and fields in the future. The used data extracted from Mozilla bug repository are the bug reports of Mozilla project from 30th July, 1999 to 2nd September, 2009. The Eclipse bug repository records bug reports of Eclipse project from 10th October, 2001 to 25th June, 2010. The Apache Ant bug repository records Ant bug reports from 07th, September, 2007 to 24th, December, 2012. The Apache Tomcat6 records Tomcat6 bug reports from 24th, August, 2003 to 28th, December, 2012. One threat here is the developer aliases [11]. That is, one developer may have more than one identifier in a project. By our manual checking randomly, we found very few developer aliases in our investigated projects.

Based our observation, we found that both the number of bug reports submitted by each developer and the number of comments attached with each bug report conforms to a power-law distribution with heavy tail [20] as shown in Fig. 8. The curve

[5] Online: http://bugzilla.mozilla.org/.
[6] Online: https://bugs.eclipse.org/bugs/.
[7] Online: http://2011.msrconf.org/msr-challenge.html.
[8] Online: https://issues.apache.org/bugzilla/.

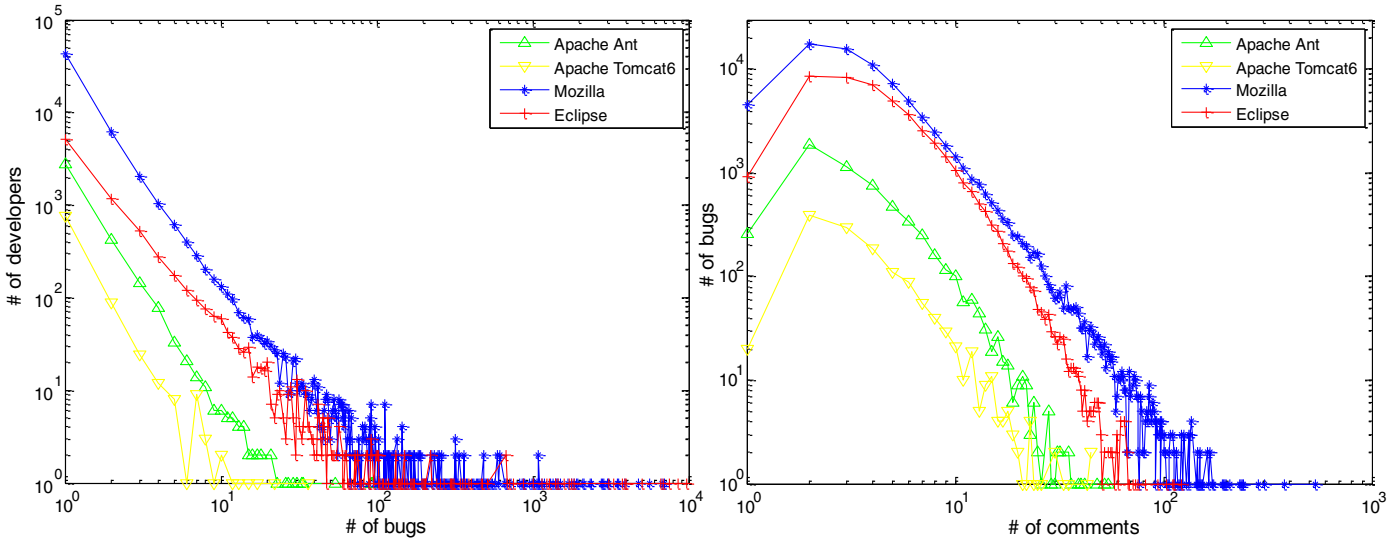

Fig. 8. The bug report-developer (above) and comment-bug report (below) distribution of Mozilla, Eclipse, Apache Ant and Apache Tomcat6 projects.

of Eclipse project is below the curve of Mozilla project because the former has a smaller number of bug reports than the latter. The same reason is explained for Apache Ant and Apache Tomcat6 projects. We can see that most bug reports are resolved with the contribution of a relatively small number of developers and most bug reports have a small number of comments. Here, we eliminate those developers who appeared only once in the bug repository from candidate bug resolvers as we speculate that they might have no interest in the project any more afterwards.

For Mozilla project, on average, each bug report has 6.27 comments and each developer contributes to 80.45 bug reports. For Eclipse project, on average, each bug report has 8.36 comments and each developer contributes to 70.56 bug reports. For Apache Ant project, on average, each bug report has 4.35 comments and each developer contributes to 1.64 bug reports. For Apache Tomcat6 project, on average, each bug report has 4.66 comments and each developer contributes to 1.46 bug reports.

We only use those bug reports whose components and products are not changed since its submission to the bug repository. In fact, we found that all those bug reports with changed components or products since its submission account for less than 5.0 percentages of all the investigated bug reports. Thus, we collected 74,100 bug reports from Mozilla project and 42,560 bug reports from Eclipse project. We collected 763 bug reports from Apache Ant project and 489 bug reports from Apache Tomcat6 project. Here, we combine "duplicate" bug reports to an extended one because they can provide supplement information for each other [26]. After data refinement, the basic information of the bug reports and its developers used for experiments are summarized in Table 4.

5.2. Experiment setup

We mimic the working scenario of a bug manager who has the historically-resolved bug reports and was confronted with new incoming bug reports. For all the investigated projects, we sort the whole dataset listed in Table 4 in chronological order of creation time and divide them into 5 subsets listed in Table 5.

Inside each subset, we further divide the sorted data into 11 non-overlapping folds (i.e., windows or frames) of equal sizes. We firstly train KSAP model using the bug reports from fold 1 and test the trained model using the bug reports from fold 2. Next, the model is trained using the data from folds 1 and 2, and is tested using the data from fold 3. By analogy, at the last step, we train KSAP model using the bug reports of folds from 1 to 10 and test the trained model using the bug reports from fold 11. Then, we compute the average precision and recall across the 10 folds. Finally, we further average the performances on the 5 subsets for each project.

It should be noted that for Apache Ant and Apache Tomcat6 projects, all the bug reports belong to the product "Ant" or "Tomcat6". Thus, we only use Eqs. (1), (2) and (4), i.e. heterogeneous proximity on common bugs and components, for ranking. For Mozilla and Eclipse project, they have more than one products and each product includes many components. In this case, we use all Eqs. (1)–(4)in ranking developers.

5.3. Baseline methods

We compare KSAP with the state of art techniques as ML-KNN [19,24], DREX [7], DRETOM [3], Bugzie [36], DevRec [24] and developer prioritization (DP method) [40] in bug report assignment. ML-KNN is used to transfer the bug triage problem to multi-labeled classification where each bug report is regarded as a data point and the developers who contribute to bug report resolution are regarded as its labels. This intuition is from Anvik's proposal to transfer bug fixer recommendation to a typical classification problem [4]. DREX is derived from our previous work and its difference with KSAP lies in that it adopts homogeneous network ranking. DRETOM is also derived from our previous work that uses LDA to construct topic models of bug reports to enhance matching between new incoming bug reports and developers' expertise.

Bugzie is proposed by Tamrawi et al. [36]. For each test bug report, we firstly extract its timestamp and fetch the candidate developers from the sorted training bug reports by timestamp. However, different from Tamrawi et al. [36], we extract technical terms by using not only the summary and description of bug reports but also the comments posted by the developers. The reason is that we are not to recommend a single developer to fix the bug but to recommend a group of developers to contribute ideas and advices in terms of comments to resolve bug reports.

DevRec is proposed by Xia et al. [24]. DevRec consists of two components: the one is BR-Based analysis using multi-labeled classification implemented by ML-KNN [19] and the other is D-Based analysis to measure the experience of a developer on the new incoming bug report by using four affinity scores of terms, topics, product and component. The same setting for LDA used in DRETOM was adopted by D-Based analysis to compute the affinity score of topics. For each

Table 4
The basic information of bug reports for experiments on Mozilla, Eclipse, Apache Ant and Apache Tomcat6 projects. U.D. abbreviates for "unique developers". B.D. abbreviates for "bug reports for each developer". U.D.B abbreviates for "unique developers for each bug report". C.B. abbreviates for "comments for each bug report".

Project	# of bug reports	# of U.D.	Average # of B.D.	Average # of U.D.B.	Average # of C.B.
Mozilla	74,100	51,571	84.78	5.35	6.67
Eclipse	42,560	5170	78.23	5.58	9.52
Apache Ant	763	587	3.75	2.17	4.35
Apache Tomcat6	489	170	6.21	2.23	4.66

Table 5
The 5 subsets with its training and test data for Mozilla, Eclipse, Apache Ant and Apache Tomcat6 projects. T.B. and S.B. abbreviate for "training bug reports" and "test bug reports", respectively.

Project	Subset no.	Time duration	# of bug reports
Mozilla	1	30/07/1999–29/07/2001	15,350
	2	30/07/2001–29/07/2003	14,776
	3	30/07/2003–29/07/2005	14,599
	4	30/07/2005–29/07/2007	14,617
	5	30/07/2007–02/09/2009	14,508
Eclipse	1	10/10/2001–09/08/2003	8521
	2	10/08/2003–09/06/2005	8604
	3	10/06/2005–09/04/2007	8582
	4	10/04/2007–09/02/2009	8518
	5	10/02/2009–25/06/2010	8085
Apache	1	07/09/2007–07/09/2010	281
Ant	2	01/06/2008–01/06/2011	264
	3	01/12/2008–01/12/2011	274
	4	10/01/2009–10/01/2012	268
	5	01/12/2010–24/12/2012	287
Apache	1	24/08/2003–24/08/2008	234
Tomcat6	2	25/08/2004–24/08/2009	224
	3	25/08/2005–24/08/2010	218
	4	25/08/2006–24/08/2011	221
	5	26/08/2007–28/12/2012	263

investigated project, we follow Xia et al. [24] using random exhaustive search in tuning the five parameters γ_1, γ_2, γ_3, γ_4 and γ_5 to weight each score in recommending developers for bug resolution.

The parameters of ML-KNN are K and s, where K is the number of nearest neighbors with the same meaning as K in Section 4.1, and s is used for probability smoothing. Following Zhang and Zhou [20], we set s as 1 and tune the parameter K together with KSAP. For DREX, its parameters includes K, N and Q where K has the same meaning as K in Section 4.1 and Q has the same meaning as Q in Section 2.2. N denotes the minimum number of bug reports for a developer if he/she was included in the candidates for resolving new bug reports and we set N as 80 according to our previous work [7]. We use the network metric degree to measure developer importance in homogeneous network ranking because it has been proved with the best performance in our previous research [7]. For DRETOM, it has three parameters: the number of topics T, θ to trade off developer's interest and expertise, and the number of developers Q for recommendation. Following our previous work [3], we set T as 200 and θ as 0.2 for optimal performance.

Two parameters need to be tuned in Bugzie as x%, i.e. the percentage of top fixers in developer cache and k, i.e. the number of technical terms for each developer. For all the investigated projects, we tune that when using all technical terms, 30% of most recent developers are enough to peak the top-5 accuracies. Moreover, we found that when we set x as 20, for Eclipse project, the parameter k should be set as 30 to peak the top-5 accuracy. For Mozilla project, the parameter k should be set as 35 for best performance. For Apache Ant and Tomcat6 projects, the parameter k should be tuned as 25 for best performance.

In DevRec, we need to tune five parameters as γ_1, γ_2, γ_3, γ_4 and γ_5. For Mozilla and Eclipse projects, we tune the parameters using evaluation criterion as Recall@5 and set the maximum number of iterations as 500 because we see from Table 4, there are about 5 unique developers for each bug report of both projects. With the same reason, for Apache Ant and Tomcat6 projects, we use Recall@3 as the evaluation criterion in parameter tuning and set the maximum number of iterations as 300.

We follow Xuan et al. [40] to introduce DP method as a baseline method for bug report assignment. However, we are different from them in that their method is used to find a single bug fixer for each bug report but ours is to locate a group of developers for collaboration. For each product and component, we construct a directed developer network as depicted in Fig. 1 in [40] using training data and compute the priority value for each developer. Then, we combine the probability of each developer predicted by SVM for ranking [41] and his or her priority score to produce the final score of the developer as done in [40]. Finally, for a given test bug report, the developers are sorted in descending order by their final scores.

5.3. Experimental results

Fig. 9 shows the precision and recall when recommending 10 ($Q = 10$) developers for each test bug report of Mozilla and Eclipse projects shown in Table 5. Fig. 10 shows the precision and recall when recommending 3 ($Q = 3$) developers for each test bug report of Apache Ant and Apache Tomcat6 projects shown in Table 5. The Precision and Recall are computed using Eqs. (5) and (6), respectively. Because the parameter K, i.e. the number of neighbors, is the only parameter needing to be tuned and, all the KSAP, DREX and ML-KNN have the common parameter K, we compare the three methods separately to tune parameter K for them.

$$Precision = \frac{|\{dev_{new,1}, \ldots, dev_{new,Q}\} \cap \{GroundTruth\}|}{|\{dev_{new,1}, \ldots, dev_{new,Q}\}|} \quad (5)$$

$$Recall = \frac{|\{dev_{new,1}, \ldots, dev_{new,Q}\} \cap \{GroundTruth\}|}{|\{GroundTruth\}|} \quad (6)$$

Here, $\{dev_{new,1}, \ldots, dev_{new,Q}\}$ denotes the recommended Q developers for resolving each test bug report. $\{GroundTruth\}$ denotes the

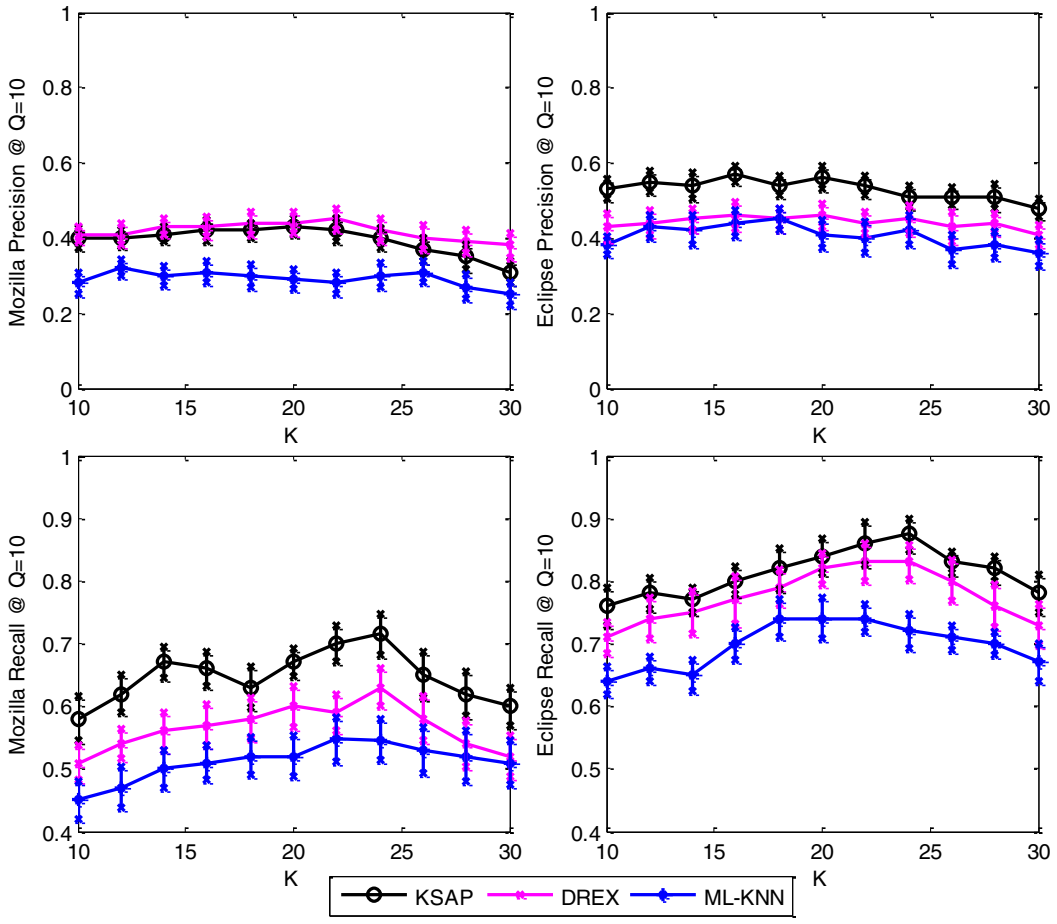

Fig. 9. The performances of KSAP in recommending 10 developers for each bug report of Mozilla and Eclipse projects compared with DREX and ML-KNN when tuning the parameter *K*.

set of developers who really contributed to the test bug resolution in real practice. For instance, the {*GroundTruth*} for bug report #333160 of Mozilla shown in Table 2 is {Mark Pilgrim, Robert Strong, Mike Beltzner}. The reason we set *Q* as 10 for Mozilla and Eclipse projects is that on average, about 10 developers collaborate with each other to resolve a bug report as shown in Table 4 in the column "Average # of U.D.B." The same reason is explained for setting *Q* as 3 for Apache Ant and Apache Tomcat6 projects. All the performances shown in Figs. 9 and 10 are averaged on all the test bug reports.

We can see from Fig. 9 that, KSAP and DREX produced higher precision than ML-KNN when the number of similar historical bug reports *K* is varied from 10 to 30 on both projects. This outcome means that developer ranking by based on developer collaboration by network is effective in bug report assignment. The frequency of a developer in historical bug resolution is not the solely important factor. The number of times that a developer collaborated with other developers is also decisive in developer recommendation for new bug reports.

It seems that there is no difference in precision between KSAP and DREX. However, when it comes to recall, we can see that KSAP produced the best performance among the three methods. In fact, in bug report assignment, recall is a more widely accepted metric for determining the performance because, the sizes of {*GroundTruth*} are different for different bug reports[9] [4]. The outcome illustrates that heterogeneous proximity ranking are more effective than traditional homogenous network ranking. In DREX, we model developers' collaborative behavior by single relation as commenting on common bugs. However, in KSAP, the developers' collaborative behaviors were captured by more than only one relation, i.e. the 9 meta-paths as shown in Table 3. When more information is utilized to characterize developers' collaborative behavior, the more effective the developers' collaboration is when it used in bug report assignment.

Moreover, we can see from Fig. 9 that when the number of similar historical bug reports *K* is varied from 10 to 30, there is an obvious trend for both KSAP and DREX that the Recall increases to a maxima peak value at first and then goes down. For Mozilla project, KSAP and DREX produce their maxima when *K* is equal to 24. For Eclipse project, they produce their maxima when *K* is equal to 22. We explain this outcome as that the number of training bug reports of Eclipser JDT project is relatively smaller than that of Mozilla project as shown in Table 5. Thus, it is not necessary to "query" a large number of similar bug reports when recommending appropriate developers for Eclipse bug reports. When *K* is small, KSAP and DREX "query" a small number of similar bug reports and the number of candidate developers is small so the recall is not maximized. When *K* is larger than a critical value (24 for Mozilla project and 22 for Eclipse project), KSAP and DREX "query" too many bug reports to produce a high recall due to "noise" in candidate developers. The best case is that when *K* is equal to 22, on Mozilla project, KSAP (0.70) improves 18.64% on recall in contrast to DREX (0.59). The worst case is that when *K* is equal to 14, on Eclipse project, DREX produces 0.75 on Recall and KSAP produce 0.77 on recall with 2.67% increase.

In Fig. 10, we see for Apache Ant project, KSAP produce better performances than other baseline methods on both precision and recall.

[9] Assuming that a bug report has 4 developers in its historical resolution, if we recommended 10 developers to the bug, the best precision we can derive from bug triaging is merely 40%. However, if the bug report has 10 developers in its historical resolution, then the precision can attain up to 100%.

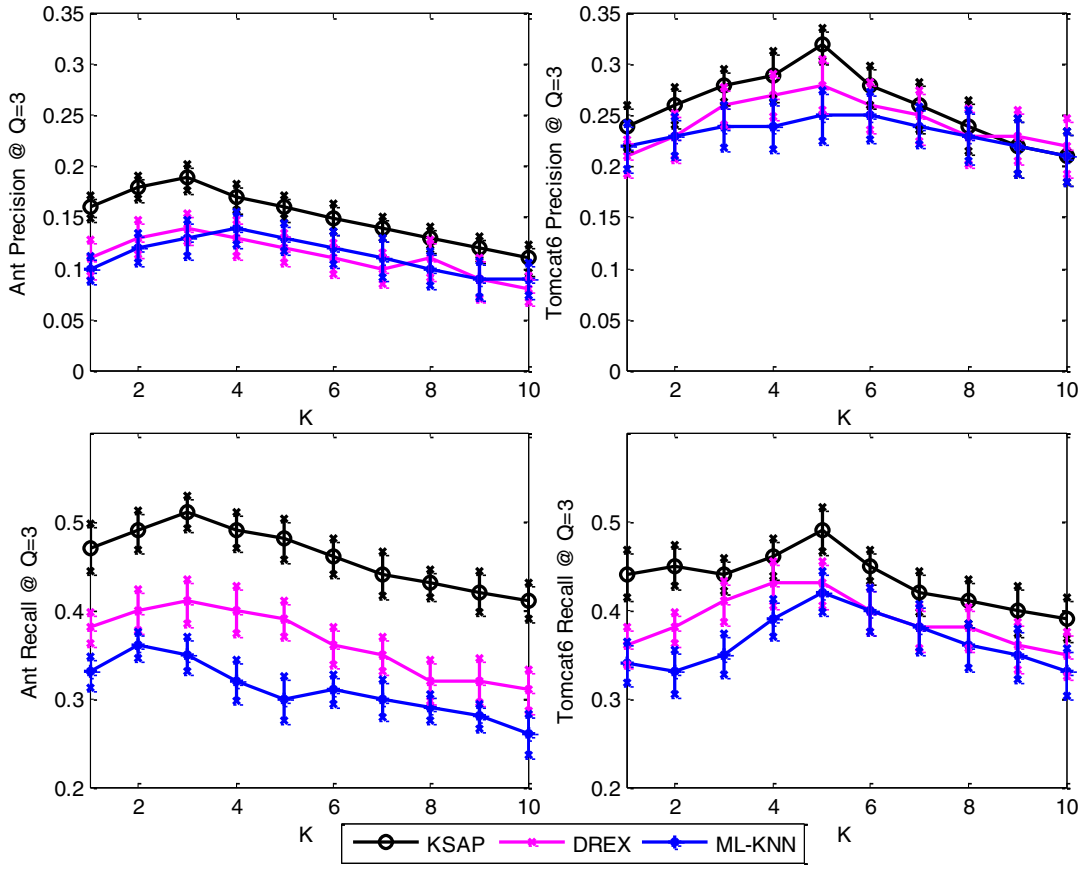

Fig. 10. The performances of KSAP in recommending 10 developers for each bug report of Apache Ant and Apache Tomcat6 projects compared with DREX and ML-KNN when tuning the parameter *K*.

When we vary *K* from 1 to 10, the best performance of KSAP is derived when *K* is 3. In this case, KSAP produce a precision as 0.19, which is of 35.71% increase than that of DREX (0.14). Meanwhile, KSAP produce a recall as 0.51 which brings about an increase as 24.39% compared with the recall of DREX (0.41). Moreover, when *K* is set as 10, KSAP has an approximately equal precision but an increase as 32.25% on recall (0.41) compared with DREX (0.31). For Apache Tomcat6 project, there seems no difference on precision between KSAP and other methods when we varied *K* from 1 to 10. However, on recall, KSAP produce the best performance as 0.49 when *K* is 5, which is of 13.95% increase than that of DREX (0.43).

It is very interesting to see from Figs. 9 and 10 that over all, the performances of KSAP and other methods on Apache Ant and Apache Tomcat6 projects are not good as those on Mozilla and Eclipse projects. We explain that this is caused by the fact that the number of bug reports and the number of developers of Apache Ant and Apache Tomcat6 project are much smaller than that of Mozilla and Eclipse projects, resulting in less collaboration among developers than the latter two projects.

Nevertheless, we see KSAP produced much better performances in Apache Ant project than that in Apache Tomcat6 project. We explain that the heterogeneous proximity on common component (i.e. $HetPro^{(C)}(dev_i, dev_j)$ in Eq. (2)) makes greater impacts on KSAP for Apache Ant project than for Apache Tomcat6 project. From Table 4, the average number of bug reports for each developer is 3.75 for Apache Ant project. That is to say, each developer participates in resolution of less than 4 bug reports, which is much smaller than that for Apache Tomcat6 project. In this case, the traditional methods such as DREX and ML-KNN that only take common bug reports into account would not produce a good performance due to the lack of enough common bug reports. However, this is not a problem for KSAP to make use of common components of developers.

Fig. 11 illustrates the performances of KSAP compared with DREX, ML-KNN, DRETOM, Bugzie and DevRec when we vary the number of recommended developers *Q* and fixed the number of similar historical bug reports *K* for Mozilla and Eclipse projects. Here, we set *K* as 24 for Mozilla project and 22 for Eclipse project, respectively, because we see from Fig. 9 that the recall is maximized with the two numbers. We can see from Fig. 11 that when *Q* is varied from 5 to 30, the precision is decreasing and the recall is increasing for both projects. This outcome can be easily understood as, when more developers are recommended for bug resolution, it brings about more developers both within and outside {*GroundTruth*}. Moreover, the developers with high ranking are more prone to be within {*GroundTruth*} than those with low ranking given by each method. The performances derived from Eclipse project are better than that derived from Mozilla project because we can see from Table 4 that the number of unique developers of Eclipse project is much smaller than that of Mozilla project. Also, we see no obvious difference among all the considered methods except ML-KNN on the precision. The same reason for the outcome in Fig. 9 can be explained here.

However, on the recall, we can see from Fig. 11 that KSAP outperforms DREX and DRETOM significantly ($P<0.05$ with Mann–Whitney *U* test [25]) for both Mozilla and Eclipse projects. In particular, the difference on the recall between KSAP and other methods derived from Mozilla project is larger than that derived from Eclipse project. We explain that KSAP is more effective when a project has larger number of unique developers. The best case is that when *Q* is equal to 30, on Mozilla project, KSAP (0.86) improves 7.5% on the recall compared with DREX (0.80). The worst case is that when *K* is equal to 2, on

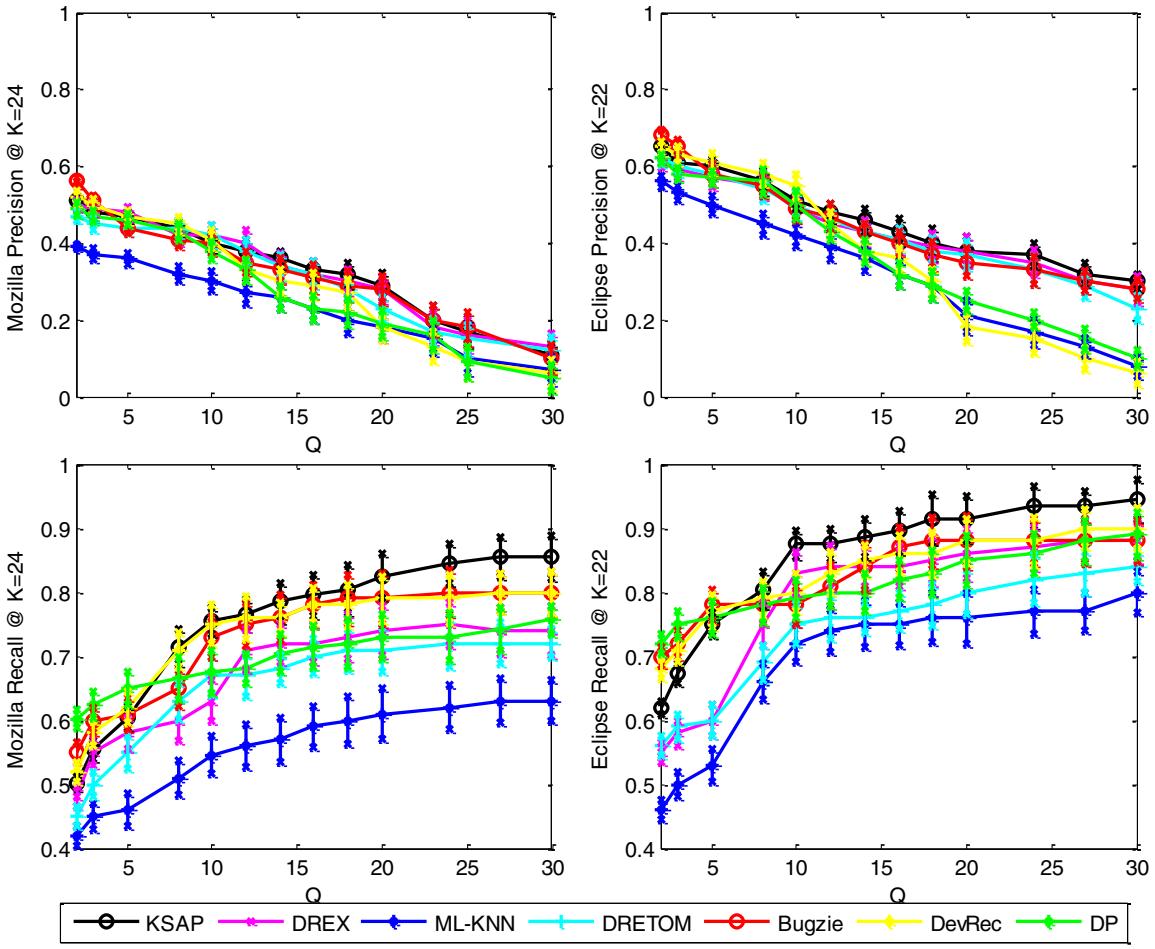

Fig. 11. The performances of KSAP in recommending different number of developers for each bug report of Mozilla and Eclipse projects with fixed number of *K*.

Eclipse project, Bugzie produce 0.70 on the recall and KSAP produce 0.62 on the recall.

When *Q* is varied from 2 to 10, the recall of KSAP and other methods increase steadily. However, it is not the case when *Q* is larger than 10 for both projects. We explain the outcome that developers provided by KSAP with high ranking are mostly in {*GroundTruth*} and those with low ranking are mostly out of {*GroundTruth*}. When we recommended more developers (that is, *Q* becomes larger), the recall of KSAP also increases. However, when it goes up to a critical number (approximately two times the average number of unique developers), KASP produce very few number of developers in {*GroundTruth*}. Considering that there are 51,571 developers in Mozilla project, when we recommend only 20 developers for a new bug report, KSAP can precisely hit more than 4 developers for most bug reports (remembering that the average number of unique developers is more than 5 in Table 4).

When we recommend less than 5 developers for Mozilla and Eclipse projects, the recall of Bugzie is much larger than that of KSAP and other methods. This outcome illustrates that when only a small number of (less than 5) developers are recommended, locality and recency are a decisive factors in producing a good performance. However, when we recommend more than 5 developers, the performances of Bugzie are worse than that of KSAP in the recall. We explain that in recommending a right fixer to fix the bug, Bugzie is a good choice to do this kind of job because "one of the recent fixers is likely to be the fixer of the next bug report" [36]. However, in recommending a group of developers who may contribute ideas to bug resolution, the locality or recency may be not as important as that in recommending a right fixer. By manual checking, we found that although some developers fixed bugs in a very short period, they actually make comments for bug resolution for a long time. For instance, the developer "Adam Schlegel" fixed seven bugs from 10th, June, 2002 to 29th, July, 2002 but he or she made 287 comments for 166 bug reports from 29th, October, 2001 to 23rd, August, 2002.

For DevRec method, on the one hand, the values of parameters γ_1, γ_2, γ_3, γ_4 and γ_5 are dependent on the sampled subset of training bug reports to a great extent. On the other hand, when we change the orders of parameters γ_1, γ_2, γ_3, γ_4 and γ_5 in searching space, the optimal parameter composition of each order setting is different from each other. It can be seen from Fig. 11 that, for Mozilla and Eclipse project, DevRec performs comparable to KSAP and better than other methods when we recommended 5–10 developers for bug resolution. However, the performance of DevRec deteriorate more drastically than other methods when *Q* is larger than 10. We explain this outcome as that in most cases, the parameter tuning method used by DevRec caused over fitting of developer recommendation. That is to say, it is not easy to generalize the parameter setting of DevRec derived in recommending 5 developers to other cases.

For DP method, when *Q* is small (not larger than 5), it is very effective in recommending potential developers for collaborative bug resolution. However, when *Q* becomes large (more than 10), its performance decreases drastically on both Mozilla and Eclipse projects. We manually checked the output of ranking by the DP method and found that for most developers, their final scores are approximately equal to that of their neighbors. For a sequence of 5 neighboring developers, their differences on final scores are less than 0.02.

Table 6
The performances on precision of KSAP compared with other baseline methods in bug report assignment on Mozilla and Eclipse projects.

Project	Method pair	Precision @1	Precision @5	Precision @10	Precision @20
Mozilla	KSAP vs ML-KNN	>>	>>	>>	>>
	KSAP vs DREX	~	~	~	>
	KSAP vs DRETOM	~	~	~	>
	KSAP vs Bugzie	~	~	~	~
	KSAP vs DevRec	~	~	>	>>
	KSAP vs DP	~	~	>	>>
Eclipse	KSAP vs ML-KNN	>>	>>	>>	>>
	KSAP vs DREX	~	~	~	>
	KSAP vs DRETOM	~	~	~	>
	KSAP vs Bugzie	<	~	~	>
	KSAP vs DevRec	<	<	<	>>
	KSAP vs DP	>	~	>	>>

Table 7
The performances on recall of KSAP compared with other baseline methods in bug report assignment on Mozilla and Eclipse projects.

Project	Method pair	Recall @1	Recall @5	Recall @10	Recall @20
Mozilla	KSAP vs ML-KNN	>>	>>	>>	>>
	KSAP vs DREX	>	>>	>>	>>
	KSAP vs DRETOM	>>	>>	>>	>>
	KSAP vs Bugzie	<<~	~	>	>>
	KSAP vs DevRec	<<	~	~	>>
	KSAP vs DP	~	<<	>>	>>
Eclipse	KSAP vs ML-KNN	>>	>>	>>	>>
	KSAP vs DREX	>>	>>	>>	>>
	KSAP vs DRETOM	>>	>>	>>	>>
	KSAP vs Bugzie	<<	<<	>>	>>
	KSAP vs DevRec	<<	<	>>	>>
	KSAP vs DP	<<	<	>>	>>

Moreover, the average difference on final scores of developers of Mozilla project are smaller (0.0037) than that of developers of Eclipse project (0.013). We explain that this outcome is caused by the larger number of developers involved in Mozilla project than that involved in Eclipse project as shown in Table 4 and the transmissive characteristic of scores from one developer to other adjacent developers in the network. Due to the small difference of final scores given by the DP method, it cause the probability produced by SVM for ranking [41] decisive in ranking developers when we recommend more than 5 developers. As a result, we see from Fig. 11 that the performance of the DP method are similar to that of ML-KNN when Q is larger than 5.

To better illustrate the effectiveness of each method, the classic non-parameter Mann–Whitney U test [1] is employed. Tables 6 and 7 demonstrate the results of Mann–Whitney U test of the performances of KSAP and other baseline methods on precision and recall, respectively. The following codification of the P-value in ranges was used: ">>" ("<<") means that P-value is lesser than or equal to 0.01, indicating a strong evidence that KSAP outperforms the compared baseline method; "<" (">") means that P-value is bigger than 0.01 and minor or equal to 0.05, indicating a weak evidence that KSAP outperforms the compared baseline method; "~" means that P-value is greater than 0.05, indicating that the compared methods do not have significant differences in performances.

Fig. 12 illustrates the performances of KSAP compared with DREX, ML-KNN, DRETOM, Bugzie, DevRec and DP method when we varied the number of recommended developers Q and fixed the number of similar historical bug reports K for Apache Ant and Apache Tomcat6 projects. Here, we set K as 3 for Apache Ant project and set K as 5 for Apache Tomcat6 project because, we see from Fig. 10 that the recall is maximized for at these two numbers. We can see from Fig. 12 that when Q is varied from 1 to 10, the precision is decreasing and the recall is increasing for both projects. The superiority of KSAP's performances is more obvious on Apache Ant project than that on Apache Tomcat6 project. The same reason as less communication and collaboration on common bug reports of Apache Ant project than that of Apache Tomcat6 project can be explained here. Moreover, we observed that when Q is attains up to 4, which is around two times the average number of developers for each bug report (i.e. 2.17 for Apache Ant and 2.23 for Tomcat6 in Table 4), both recall enters into an stable state with very small increase when we further increase Q.

For Apache Ant and Tomcat6 projects, it can be seen that when we recommended 3 developers for bug resolution, Bugzie and DevRec performs comparable to (even better than) KSAP in Recall. The DP method performs even better than KSAP. However, when we recommended more than 3 developers, the performances of Bugzie are much worse than that of KSAP. The same explanation of recommending bug fixer other than contributors in Mozilla and Eclipse projects can be used here. The performances of DevRec decreases drastically, even worse than ML-KNN in some cases in precision. The same explanation of possible over fitting in Eclipse and Mozilla projects can be also be employed here. We also see that when Q is larger than 3, the performances of the DP method is similar to that of ML-KNN method. We also conducted Mann–Whitney U test to examine the significance level of precision and recall of KSAP compared with other baseline methods as shown in Tables 8 and 9, respectively.

Table 10 lists the average values with standard deviation of MAP and MRR [42] measured on different projects of KSAP, Bugzie, DevRec and DP method. We can see that on the one hand, in Mozilla and Eclipse projects, the differences of MAP between KSAP and Bugzie are much larger than that of MRR. This outcome is caused by the fact that both KSAP and Bugzie can rank the developers in {*GroundTruth*} at the top places but, KSAP provides those developers with smaller rankings than Bugzie. By our manual inspection, we found that KSAP ranks those developers in {*GroundTruth*} within top 30 places in most cases. However, Bugzie needs to count 50 places to find all the developers in {*GroundTruth*}. Nevertheless, considering the large

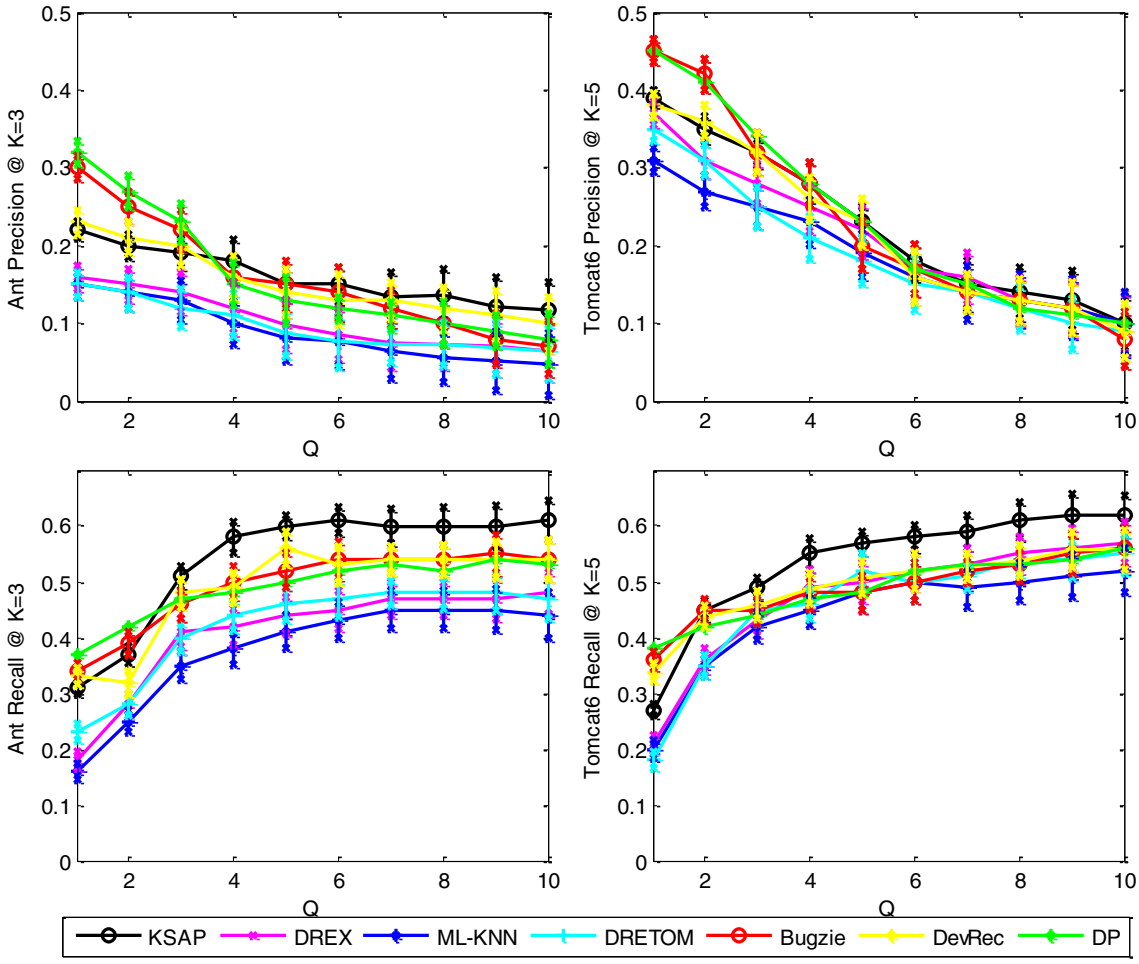

Fig. 12. The performances of KSAP in recommending different number of developers for each bug report of Apache Ant and Apache Tomcat6 projects with fixed number of *K*.

Table 8
The performances on precision of KSAP compared with other baseline methods in bug report assignment on Ant and Tomcat6 projects.

Project	Method pair	Precision @1	Precision @5	Precision @10
Ant	KSAP vs ML-KNN	>>	>>	>>
	KSAP vs DREX	>>	>>	>>
	KSAP vs DRETOM	>>	>>	>>
	KSAP vs Bugzie	<<	~	>>
	KSAP vs DevRec	<	~	>
	KSAP vs DP	<<	>	>>
Tomcat6	KSAP vs ML-KNN	>>	>>	~
	KSAP vs DREX	>>	~	~
	KSAP vs DRETOM	>>	~	~
	KSAP vs Bugzie	<<	~	>
	KSAP vs DevRec	~	~	~
	KSAP vs DP	<<	~	~

Table 9
The performances on recall of KSAP compared with other baseline methods in bug report assignment on Ant and Tomcat6 projects.

Project	Method pair	Recall @1	Recall @5	Recall @10
Ant	KSAP vs ML-KNN	>>	>>	>>
	KSAP vs DREX	>	>>	>>
	KSAP vs DRETOM	>>	>>	>>
	KSAP vs Bugzie	<<~	>>	>>
	KSAP vs DevRec	<	>>	>>
	KSAP vs DP	<<	<<	>>
Tomcat6	KSAP vs ML-KNN	>>	>>	>>
	KSAP vs DREX	>>	>>	>>
	KSAP vs DRETOM	>>	>>	>>
	KSAP vs Bugzie	<<	>>	>>
	KSAP vs DevRec	<<	>>	>>
	KSAP vs DP	<<	>>	>>

number of candidate developers (51,571 for Mozilla project and 5170 for Eclipse project), we can draw that both KSAP and Bugzie can produce acceptable performances in developer recommendation. On the other hand, in Apache Ant and Apache Tomcat6 projects, we can see that KSAP outperforms other methods significantly. We manually checked the outcome of each method and found that for all the methods, they can locate at least one developer in {*GroundTruth*} at top 10 places. For KSAP, it can approximately locate 2 developers in {*GroundTruth*} at top 30 developers. For other methods, the number is approximately 100. Moreover, if we want to locate all the developers in {*GroundTruth*}, KSAP needs to search top 150 and 100 places in Ant and Tomcat6 ranking, respectively. For other methods, they need to search at least top 350 and 200 places in Ant and Tomcat 6 ranking, respectively.

6. Threats

For external validity, there is a threat of generalization of our experiments, i.e. recommending developers for bug report resolution. The projects under investigation are all from open source community. Since we only used open source projects for evaluation, the results might not be generalizable to closed-source projects. The reason here is that the subject projects used in the paper have a history of more

Table 10
The average values with standard deviation of MAP and MRR of KSAP and other methods in bug report assignment on Mozilla, Eclipse, Ant and Tomcat6 projects.

Project	Method							
	KSAP		Bugzie		DevRec		DP	
	MAP	MRR	MAP	MRR	MAP	MRR	MAP	MRR
Mozilla	0.4449 ± 0.0673	0.2779 ± 0.0247	0.3740 ± 0.0583	0.2459 ± 0.0481	0.2322 ± 0.0598	0.1338 ± 0.0271	0.2201 ± 0.0439	0.1127 ± 0.0236
Eclipse	0.5642 ± 0.0597	0.2800 ± 0.0175	0.4253 ± 0.0572	0.2611 ± 0.0419	0.2884 ± 0.0372	0.1509 ± 0.0357	0.2074 ± 0.0378	0.0848 ± 0.0392
Ant	0.3648 ± 0.0785	0.3478 ± 0.0206	0.1796 ± 0.0485	0.1726 ± 0.0428	0.1731 ± 0.0515	0.1695 ± 0.0491	0.1727 ± 0.0577	0.1693 ± 0.0555
Tomcat6	0.3654 ± 0.0695	0.3475 ± 0.0176	0.2118 ± 0.0380	0.1879 ± 0.0517	0.1417 ± 0.0557	0.1251 ± 0.0340	0.1131 ± 0.0431	0.0972 ± 0.0172

than 5 years and accumulate a large number of resolved bug reports with complete records of developers' collaboration in their open bug repositories. These are all the prerequisites to examine the effectiveness of our proposed KSAP approach. We hold that if a closed-source project has enough historical records of bug report resolution in bug repository, KSAP can be generalized to this kind of cases without difficulty.

For construct validity, the first threat comes from that the criteria we set for experimental data inclusion are not rigorously examined. On the one hand, we merely investigate the bug reports whose components and products are never changed since its submission. We found only less than 5 percentages of all bug reports whose components and products would be changed after its submission. Moreover, if either component or product of a bug report was changed, then we should rerun KSAP on this changed bug report. On the other hand, we also removed those inactive developers (appearing only once in the bug repository) for bug report resolution because it would also be meaningless to recommend a developer who is absent from the project.

The second threat is that our evaluation method may be biased. We partitioned the collected bug reports into 11 folds according to different time duration. We split each fold into two sets to evaluate the KSAP approach. Different partitions and splits may yield different precision and recall due to concept drift [27]. Other types of measurements might yield different interpretation of the bug report triage results.

The third threat comes from the problem of over- specialization [36] of developers. As in indicated by our recent study [38], the time difference between a developer's latest activity in a project and the submission of the bug report is an important factor in considering whether or not the bug report will be handled by the developer. Moreover, a diverse recommendation of developers may accelerate the processing of bug report assignment.

7. Related work

The related work of the paper includes two aspects. The first aspect is network analysis of open source community. Singh [32] argued that open source community network of developers characterized by small-world properties have positive effect on the productivity of developers. They validated their assumption using the data from SourceForge.net using 4279 projects. Their experimental results showed that small-world properties closely correlated with the success of success of open source projects. We also observed the power-law phenomena on bug report-developer and comment-bug report distribution as shown in Fig. 8. However, it is not the focus of the paper to study the impact of small-world properties on success of subject projects.

Lim et al. [33] developed StakeNet using social network analysis to identify and prioritize stakeholders of RALIC project. Their network was constructed by recommendation of stakeholders of the project and an initial set of stakeholders and roles were identified by traditional search method. They reported that betweenness achieved the highest accuracy among all the social network measures in prioritizing stakeholder roles and PageRank produced the highest accuracy in prioritizing stakeholders. Bird et al. [34] mined email social networks in the point view of communication and co-ordination of OSS projects. Their analysis reported some interesting results. For instance, the in-degree and out-degree distribution of the social network exhibit typical long-tailed, small-world characteristics and there is a strong relationship between the level of email activity and the level of activity in the source code. Pinzger et al. [35] used social network measures to detect failure-prone modules of Microsoft Windows Vista. They reported that central modules are more failure-prone than those in the surrounding areas of the network. Zhang and Lee [39] used concept profile and social network analysis for bug report assignment.

The difference between our study in this paper and theirs in the above related work lies in that firstly, we consider heterogeneous network, not the homogenous network as used in their work. Secondly, we propose heterogeneous proximity to rank nodes in our heterogeneous network, not the traditional social network measures such as degree, betweenness and PageRank.

The second aspect of our related work is bug report resolution. Many studies have been conducted with the goal of recommending appropriate developers for resolving new bug reports. These studies can roughly be divided into two main streams. One is to assign bugs to developers based on text categorization, such as the studies conducted by Cubranic and Murphy [5] and Anvik et al. [4,28]. The other one is to model expertise of developers using historical to match bug contents such as activity records matching [29], noun phrase matching [6], fuzzy expertise caching [37] and vocabulary-based matching [30], including our previous study topic-based matching as DRETOM [3]. In the former, machine learning techniques are used to categorize new bug reports using historically assigned bug reports as training data. In the latter, historical bug report resolution records or (and) change history in source code repository are used to characterize the expertise of developers to match textual contents of new bug reports. In this study, we also use information retrieval to match textual contents of historical bug reports and the new bug report. Nevertheless, we further model the developers' collaboration using heterogeneous network analysis on historical bug reports.

In our previous study [31], we conducted heterogeneous network analysis of developer contribution in bug resolution. We considered 4 types of developer contribution to bug resolution as reporting new bugs, reopening bugs, making comments and changing source code. Those developer contributions are very different from developer collaboration shown in Table 3. The reason is that two developers' relation is considered when we talk about developer collaboration. However, we only consider what a developer has done for a bug report when we talk about developer contribution.

In another work [12], we proposed to use heterogeneous network analysis to improve bug report assignment. We firstly adopted supervised learning to produce candidate developers and then added other developers who frequently collaborate with those candidate developers under common components. We admit that this paper has the

similar idea as our previous work [12] in using heterogeneous network for bug report assignment. However, the paper is different from our previous work in that on the one hand, we proposed a complete solution to extract meta-path instances from heterogeneous network automatically, including heterogeneous network construction, meta-path extraction from schema and instance extraction from heterogeneous network. In [12], we only consider 5 types of collaboration whereas 9 types of collaboration in this paper. On the other hand, heterogeneous proximity is adapted entirely in the paper to recommend developers directly without any learning methods.

8. Conclusion

In this paper, we propose an approach called KSAP to recommend developers for bug report resolution using KNN search and heterogeneous proximity. When a new bug report is coming, KNN search is used to find historically-resolved similar bug reports in textual contents and heterogeneous proximity is used to rank developers extracted from similar historical bug reports. The main contribution of the paper can be summarized in three aspects.

- Firstly, we propose a method to build heterogeneous network of a bug repository using entities as developer (D), comment (T), bug report (B), component (C) and product (P) and its relations (see Sections 3.1 and 3.2). Then we present an algorithm to extract developers' collaborative behavior in bug resolution within bug repository based on graph traverse (see Section 3.3). We hold that the proposed algorithm can be extended to other area related with human collaboration provided that the behavior schema depicted in Fig. 2 can be come up with.
- Secondly, we propose heterogeneous proximity to rank developers' contribution in historical bug resolution and combined KNN search and heterogeneous proximity for bug triage (see Sections 4.1 and 4.2).
- Thirdly, we conduct extensive experiments to examine the proposed KSAP approach compared with existing methods including DREX [7], DRETOM [3], ML-KNN [19], BugZie [36] and DevRec [24] and DP method [40] using the datasets from Mozilla, Eclipse, Apache Ant and Apache Tomcat6 projects (see Section 5).

Experimental results demonstrate some promising aspects of KSAP. In the future, we will consider more entities in the bug repository such as version and platform of bug reports to make use of more heterogeneous information in recommending developers for bug report resolution. Also, we will plan to address the problem of over-specialization [36] in heterogeneous proximity ranking and we have already conducted an initial study on this topic [38].

Acknowledgment

This work is supported by the National Natural Science Foundation of China under Grant nos 71101138, 61379046 and 61432001; the Beijing Natural Science Fund under Grant no. 4122087; the Fundamental Research Fund for the Central Universities in BUCT. We would like to appreciate the reviewers' comments to improve the paper as well as the help of Ms. Mary Ann Mooradian to proofread the paper.

References

[1] G. Jeong, S. Kim, T. Zimmermann, Improving bug triage with bug tossing graphs, in: Proceedings of 17th ACM SIGSOFT Symposium on Foundations of Software Engineering, 2009, pp. 111–120.
[2] J. Anvik, G.C. Murphy, Reducing the effort of bug report triage: recommenders for development-oriented decisions, ACM Trans. Softw. Eng. Methodol. 20 (3) (2011) 1–35.
[3] X. Xie, W. Zhang, Y. Yang, Q. Wang, DRETOM: developer recommendation based on topic models for bug resolution, in: Proceedings of the 8th International Conference on Predictive Models in Software Engineering, 2012, pp. 19–28.
[4] J. Anvik, L. Hiew, G.C. Murphy, Who should fix this bug? in: Proceedings of the 28th International Conference on Software Engineering, 2006, pp. 361–370.
[5] D. Cubranic, G.C. Murphy, Automatic bug triage using text categorization, in: Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering, 2004, pp. 92–97.
[6] H. Naguib, N. Narayan, B. Brugge, Bug report assignee recommendation using activity profiles, in: Proceedings of 10th IEEE Working Conference on Mining Software Repositories, 2013, pp. 22–30.
[7] W. Wu, W. Zhang, Y. Yang, Q. Wang, DREX: developer recommendation with K-Nearest-Neighbor search and expertise ranking, in: Proceedings of 18th Asia Pacific Software Engineering Conference, 2011, pp. 389–396.
[8] C. Bird, N. Nagappan, Who? Where? What? Examining distributed development in two large open source projects, in: Proceedings of 9th IEEE Working Conference on Mining Software Repositories, 2012, pp. 237–246.
[9] C. Bird, D. Pattison, R. D'Souza, V. Filkov, P. Devanbu, Latent social structure in open source projects, in: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, pp. 24–35.
[10] W. Zhang, Y. Yang, Q. Wang, An empirical study on identifying core developers using network analysis, in: Proceedings of the 2nd International Workshop on Evidential Assessment of Software Technologies, 2012, pp. 43–48.
[11] W. Zhang, Y. Yang, Q. Wang, Network analysis of OSS evolution: an empirical study on ArgoUML project, in: Proceedings of the 12th International Workshop on Principles on Software Evolution and 7th ERCIM Workshop on Software Evolution, 2011, pp. 71–80.
[12] S. Wang, W. Zhang, Y. Yang, Q. Wang, DevNet: exploring developer collaboration in heterogeneous networks of bug repositories, in: Proceedings of 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2013, pp. 193–202.
[13] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter, C. Weiss, What makes a good bug report? IEEE Transactions on Software Engineering 36 (5) (2010) 618–643.
[14] N.S. Altman, An introduction to kernel and nearest-neighbor nonparametric regression, Am. Stat. 46 (3) (1992) 175–185.
[15] G. Salton, A. Wang, C.S. Yang, A vector space model for information retrieval, J. Am. Soc. Inf. Sci. 18 (1975) 613–620.
[16] W. Zhang, T. Yoshida, X. Tang, A comparative study of TF-IDF, LSI and multi-words for text classification, Expert Syst. Appl. 38 (3) (2011) 2758–2765.
[17] E. Shimon, Graph Algorithms, 2nd edition, Cambridge University Press, 2011.
[18] P.J. Guo, T. Zimmermann, B. Murphy, Characterizing and predicting which bugs get fixed: an empirical study of Microsoft windows, in: Proceedings of the 32th International Conference on Software Engineering, 2010, pp. 495–504.
[19] M. Zhang, Z. Zhou, ML-KNN: a lazy learning approach to multi-label learning, Pattern Recognit. 40 (7) (2007) 2038–2048.
[20] A. Barabási, R. Albert, Emergence of scaling in random networks, Science 286 (5439) (1999) 509–512.
[21] Y. Sun, J. Han, Mining heterogeneous information networks: principles and methodologies, Synthesis Lectures on Data Mining and Knowledge Discovery, Morgan & Claypool Publishers, 2012.
[22] J. Hopcroft, O. Khan, B. Kulis, B. Selman, Tracking evolving communities in large linked networks, Proc. Natl. Acad. Sci. USA 101 (2004) 5249–5253.
[23] T. Wolf, A. Schröter, D. Damian, L.D. Panjer, T.H.D. Nguyen, Mining task-based social networks to explore collaboration in software teams, IEEE Softw. 26 (1) (2009) 58–66.
[24] X. Xia, D. Lo, X. Wang, B. Zhou, Accurate developer recommendation for bug resolution, in: Proceedings of the 20th Working Conference on Reverse Engineering, 2013, pp. 72–81.
[25] H.B. Mann, D.R. Whitney, On a test of whether one of two random variables is stochastically larger than the other, Ann. Math. Stat. 18 (1) (1947) 50–60.
[26] N. Bettenburg, R. Premraj, T. Zimmermann, S. Kim, Duplicate bug reports considered harmful ... really? in: Proceedings of the 24th International Conference on Software Maintenance, 2008, pp. 337–345.
[27] G. Widmer, M. Kubat, Learning in the presence of concept drift and hidden contexts, Mach. Learn. 23 (1996) 69–101.
[28] J. Anvik, Automating bug report assignment, in: Proceedings of the 28th International Conference on Software Engineering, 2006, pp. 937–940.
[29] R. Shokripour, J. Anvik, Z.M. Kasirun, S. Zamani, Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation, in: Proceedings of 10th IEEE Working Conference on Mining Software Repositories, 2013, pp. 2–11.
[30] D. Matter, A. Kuhn, O. Nierstrasz, Assigning bug reports using a vocabulary-based expertise model of developers, in: Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories, 2009, pp. 131–140.
[31] W. Zhang, S. Wang, Y. Yang, Q. Wang, Heterogeneous network analysis of developer contribution in bug repositories, in: Proceedings of the 2013 IEEE International Conference on Cloud and Service Computing, 2013, pp. 98–105.
[32] P.V. Singh, The small-world effect: the influence of marco-level properties of developer collaboration networks on open-source project success, ACM Trans. Softw. Eng. Methodol. 20 (2) (2010).
[33] S.L. Lim, D. Quercia, A. Finkelstein, Stakenet: Using social networks to analyse the stakeholders of large-scale software projects, in: Proceedings of 32nd International Conference on Software Engineering, 2010, pp. 295–304.
[34] C. Bird, A. Gourley, P. Devanbu, M. Gertz, A. Swaminathan, Miningemail social networks, in: Proceedings of the 3rd International Workshop on Mining Software Repositories, Shanghai, China, May 2006.

[35] M. Pinzger, N. Nagappan, B. Murphy, Can developer-module networks predict failures? in: Proceedings ofACM SIGSOFT International Symposium on Foundations of Software Engineering,2008/FSE-16, Atlanta, USA, November 2008.
[36] A. Tamrawi, T.T. Nguyen, J.M. Al-Kofahi, T.N. Nguyen, Fuzzy set and cache-based approach for bug triaging, in: Proceedings of Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering ESEC/FSE'11, ACM, 2011, pp. 365–375.
[37] J. woo Park, M.-W. Lee, J. Kim, S. won Hwang, and S. Kim. CosTriage: a cost-aware triage algorithm for bug reporting systems, in: Proceedings of AAAI'11.
[38] S. Wang, W. Zhang, Q. Wang, FixerCache: unsupervised caching active developers for diverse bug triage, in: Proceedings of the 25thACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM, 2014.
[39] T. Zhang, B. Lee, An automated bug triage approach: a concept profile and social network based developer recommendation, in: Proceedings of the 8th International Conference on Intelligent Computing, 2012, pp. 505–512.
[40] J. Xuan, H. Jiang, Z. Ren, W. Zou, Developer prioritization in bug repositories, in: Proceedings of the ACM/IEEE International Conference on Software Engineering, ICSE'2012, 2012, pp. 25–35.
[41] T. Joachims, Training linear SVMs in linear time, in: Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (KDD), 2006.
[42] T. Andrew, S. Falk, User performance versus precision measures for simple search tasks, in: Proceedings of the 29th Annual international ACM Conference on Research and Development in Information Retrieval, 2006, pp. 11–18.