Empir Software Eng (2016) 21:1533–1578
DOI 10.1007/s10664-015-9401-9

Automated bug assignment: Ensemble-based machine learning in large scale industrial contexts

Leif Jonsson[1,2] · **Markus Borg**[3] · **David Broman**[4,5] · **Kristian Sandahl**[2] · **Sigrid Eldh**[1] · **Per Runeson**[3]

Published online: 10 September 2015

Abstract Bug report assignment is an important part of software maintenance. In particular, incorrect assignments of bug reports to development teams can be very expensive in large software development projects. Several studies propose automating bug assignment techniques using machine learning in open source software contexts, but no study exists for large-scale proprietary projects in industry. The goal of this study is to evaluate automated bug assignment techniques that are based on machine learning classification. In particular,

Communicated by: Sunghun Kim

✉ Leif Jonsson
leif.jonsson@ericsson.com

Markus Borg
markus.borg@cs.lth.se

David Broman
dbro@kth.se

Kristian Sandahl
kristian.sandahl@liu.se

Sigrid Eldh
sigrid.eldh@ericsson.com

Per Runeson
per.runeson@cs.lth.se

1 Ericsson AB, Torshamnsgatan 35 Kista, Stockholm, Sweden

2 Department of Computer and Information Science, Linköping University, SE-581 83 Linköping, Sweden

3 Department of Computer Science, Lund University, Box 118, 221 00 Lund, Sweden

4 KTH Royal Institute of Technology, 164 40 Kista, Sweden

5 UC Berkeley, Berkeley, CA 94720, USA

we study the state-of-the-art ensemble learner Stacked Generalization (SG) that combines several classifiers. We collect more than 50,000 bug reports from five development projects from two companies in different domains. We implement automated bug assignment and evaluate the performance in a set of controlled experiments. We show that SG scales to large scale industrial application and that it outperforms the use of individual classifiers for bug assignment, reaching prediction accuracies from 50 % to 89 % when large training sets are used. In addition, we show how old training data can decrease the prediction accuracy of bug assignment. We advice industry to use SG for bug assignment in proprietary contexts, using at least 2,000 bug reports for training. Finally, we highlight the importance of not solely relying on results from cross-validation when evaluating automated bug assignment.

Keywords Machine learning · Ensemble learning · Classification · Bug reports · Bug assignment · Industrial scale; Large scale

1 Introduction

In large projects, the continuous inflow of bug reports[1] challenges the developers' abilities to overview the content of the Bug Tracking System (BTS) (Bettenburg et al. 2008; Just et al. 2008). As a first step toward correcting a bug, the corresponding bug report must be assigned to a development team or an individual developer. This task, referred to as *bug assignment*, is normally done manually. However, several studies report that manual bug assignment is labor-intensive and error-prone (Baysal et al. 2009; Jeong et al. 2009; Bhattacharya et al. 2012), resulting in "bug tossing" (i.e., reassigning bug reports to another developer) and delayed bug corrections. Previous work report that bug tossing is frequent in large projects; 25 % of bug reports are reassigned in the Eclipse Platform project (Anvik and Murphy 2011) and over 90 % of the fixed bugs in both the Eclipse Platform project and in projects in the Mozilla foundation have been reassigned at least once (Bhattacharya et al. 2012). Moreover, we have previously highlighted the same phenomenon in large-scale maintenance at Ericsson (Jonsson et al. 2012).

Several researchers have proposed improving the situation by automating bug assignment. The most common automation approach is based on *classification* using supervised Machine Learning (ML) (Anvik et al. 2006; Jeong et al. 2009; Alenezi et al. 2013) (see Section 2 for a discussion about machine learning and classification). By training a classifier, incoming bug reports can automatically be assigned to developers. A wide variety of classifiers have been suggested, and previous studies report promising prediction accuracies ranging from 40 % to 60 % (Anvik et al. 2006; Ahsan et al. 2009; Jeong et al. 2009; Lin et al. 2009). Previous work has focused on Open Source Software (OSS) development projects, especially the Eclipse and Mozilla projects. Only a few studies on bug assignment in proprietary development projects are available, and they target small organizations (Lin et al. 2009; Helming et al. 2011). Although OSS development is a relevant context to study, it differs from proprietary development in aspects such as development processes, team

[1]Other common names for bug report include *issues*, *tickets*, *fault reports*, *trouble reports*, *defect reports*, *anomaly reports*, *maintenance requests*, and *incidents*.

structure, and developer incentives. Consequently, whether previous research on automated bug assignment applies to large proprietary development organizations remains an open question.

Researchers have evaluated several different ML techniques for classifying bug reports. The two most popular classifiers in bug assignment are Naive Bayes (NB) and Support Vector Machines (SVM), applied in pioneering work by Cubranic and Murphy (2004) and Anvik et al. (2006), respectively. Previous work on bug assignment has also evaluated several other classifiers, and compared the *prediction accuracy* (i.e., the proportion of bug reports assigned to the correct developer) with varying results (Anvik et al. 2006; Ahsan et al. 2009; Helming et al. 2011; Anvik and Murphy 2011; Bhattacharya et al. 2012). To improve the accuracy, some authors have presented customized solutions for bug assignment, tailored for their specific project contexts (e.g., Xie et al. (2012) and Xia et al. (2013)). While such approaches have the potential to outperform general purpose classifiers, we instead focus on a solution that can be deployed as a plug-in to an industrial BTS with limited customization. On the other hand, our solution still provides a novel technical contribution in relation to previous work on ML-based bug assignment by *combining* individual classifiers.

Studies in other domains report that *ensemble learners*, an approach to combine classifiers, can outperform individual techniques when there is diversity among the individual classifiers (Kuncheva and Whitaker 2003). In recent years, combining classifiers has been used also for applications in software engineering. Examples include effort estimation (Li et al. 2008), fault localization (Thomas et al. 2013), and fault classification (Xia et al. 2013). In this article, we propose using *Stacked Generalization* (SG) (Wolpert 1992) as the ensemble learner for improving prediction accuracy in automated bug assignment. SG is a state-of-the-art method to combine output from multiple classifiers, used in a wide variety of applications. One prominent example was developed by the winning team of the Netflix Prize, where a solution involving SG outperformed the competition in predicting movie ratings, and won the $1 million prize (Sill et al. 2009). In the field of software engineering, applications of SG include predicting the numbers of remaining defects in black-box testing (Li et al. 2011), and malware detection in smartphones (Amamra et al. 2012). In a previous pilot study, we initially evaluated using SG for bug assignment with promising results (Jonsson et al. 2012). Building on our previous work, this paper constitutes a deeper study using bug reports from different proprietary contexts. We analyze how the prediction accuracy depends on the choice of individual classifiers used in SG. Furthermore, we study *learning curves* for different systems, that is, how the amount of training data impacts the overall prediction accuracy.

We evaluate our approach of automated bug assignment on bug reports from five large proprietary development projects. Four of the datasets originate from product development projects at a telecom company, totaling more than 35,000 bug reports. To strengthen the external validity of our results, we also study a dataset of 15,000 bug reports, collected from a company developing industrial automation systems. Both development contexts constitute large-scale proprietary software development, involving hundreds of engineers, working with complex embedded systems. As such, we focus on software engineering much different from the OSS application development that has been the target of most previous work. Moreover, while previous work address *bug assignment to individual developers*, we instead evaluate *bug assignment to different development teams*, as our industrial partners report this task to be more important. In large scale industrial development it makes sense to assign bugs to a team and let the developers involved distribute the work internally. Individual developers might be unavailable for a number of reasons, e.g., temporary peaks of workload, sickness, or employee turnover, thus team assignment is regarded as more important by our industry partners.

The overall goal of our research is to *support bug assignment in large proprietary development projects using state-of-the-art ML*. The focus of the paper is not to compare how specific classifiers behave on OSS data and industry data. Instead, the aim is to see if ML strategies, which have been applied in OSS contexts, also are applicable in industrial settings. This is particularly important because there are many differences in the way people are working in OSS projects and large scale industry projects. We further refine this goal into four Research Questions (RQ):

RQ1	Does stacked generalization outperform individual classifiers?
RQ2	How does the ensemble selection in SG affect the prediction accuracy?
RQ3	How consistent learning curves does SG display across projects?
RQ4	How does the time locality of training data affect the prediction accuracy?

To be more specific, our contributions are as follows:

- We synthesize results from previous studies on automated bug assignment and present a comprehensive overview (Section 3).
- We present the first empirical studies of automated bug assignment with data originating from large proprietary development contexts, where bug assignments are made at team level (Section 4).
- We conduct a series of experiments to answer the above specified research questions (Section 5) and report the experimental results and analysis from a practical bug assignment perspective (Section 6), including analyzing threats to validity (Section 7).
- We discuss the big picture, that is, the potential to deploy automated support for bug assignment in the two case companies under study (Section 8).

2 Machine Learning

Machine learning is a field of study where computer programs can learn and get better at performing specific tasks by training on historical data. In this section, we discuss more specifically what machine learning means in our context, focusing on *supervised* machine learning—the type of machine learning technique used in this paper.

2.1 Supervised Machine Learning Techniques and Their Evaluation

In *supervised learning*, a machine learning algorithm is *trained* on a *training set* (Bishop 2006). A training set is a subset of some historical data that is collected over time. Another subset of the historical data is the *test set*, used for *evaluation*. The evaluation determines how well the system performs with respect to some metric. In our context, an example metric is the number of bug reports that are assigned to correct development teams, that is, the teams that ended up solving the bugs. The training set can, in turn, be split into disjoint sets for parameter optimization. These sets are called *hold-out* or *validation* sets. After the system has been trained on the training data, the system is then evaluated on each of the instances in the test set. From the point of view of the system, the test instances are completely new since none of the instances in the training set are part of the test set.

To evaluate the predictions, we apply cross-validation with stratification (Kohavi 1995). Stratification means that the instances in the training sets and the test sets are selected to be proportional to their distribution in the whole dataset. In our experiments, we use stratified *10-fold cross-validation*, where the dataset is split into ten stratified sets. Training and evaluation are then performed ten times, each time shifting the set used for testing. The final estimate of accuracy of the system is the average of these ten evaluations.

In addition to 10-fold cross-validation, we use two versions of timed evaluation to closely replicate a real world scenario: sliding window and cumulative time window. In the *sliding window* evaluation, both the training set and the test set have fixed sizes, but the time difference between the sets varies by selecting the training set farther back in time. Sliding window is described in more details in Section 5.5.4. The sliding window approach makes it possible to study how time locality of bug reports affects the prediction accuracy of a system.

The *cumulative time window* evaluation also has a fixed sized test set, but increases the size of the training set by adding more data farther back in time. This scheme is described in more details in Section 5.5.5. By adding more bug reports incrementally, we can study if adding older bug reports is detrimental to prediction accuracy.

2.2 Classification

We are mainly concerned with the type of machine learning techniques called *classification* techniques. In classification, a software component, called a *classifier*, is invoked with inputs that are named *features*. Features are extracted from the training data instances. Features can, for instance, be in the form of free text, numbers, or nominal values. As an example, an instance of a bug report can be represented in terms of features where the subject and description are free texts, the customer is a nominal value from a list of possible customers, and the severity of the bug is represented on an ordinal scale. In the evaluation phase, the classifier will—based on the values of the features of a particular instance—return the class that the features correspond to. In our case, the different classes correspond to the development teams in the organization that we want to assign bugs to. The features can vary from organization to organization, depending on which data that is collected in the bug tracking system.

2.3 Ensemble Techniques and Stacked Generalization

It is often beneficial to combine the results of several individual classifiers. The general idea to combine classifiers is called *ensemble techniques*. Classifiers can be combined in several differet ways. In one ensemble technique, called *bagging* (Breiman 1996), many instances of *the same type* of classifier are trained on *different versions of the training set*. Each classifier is trained on a new dataset, created by sampling with replacement from the original dataset. The final result is then obtained by averaging the results from all of the classifiers in the ensemble. Another ensemble technique, called *boosting*, also involves training several instances of the same type of classifier on a modified training set, which places *different weights* on the different training instances. The classifiers are trained and evaluated in sequence with subsequent classifiers trained with higher weights on instances that previous classifiers have misclassified. A popular version of boosting is called Adaboost (Freund and Schapire 1995). Both bagging and boosting use the same type of classifiers in the ensemble and vary the data the classifiers are trained on.

Stacked Generalization (SG) (Wolpert 1992) (also called *stacking* or *blending*) is an ensemble technique that combines several level-0 classifiers *of different types* with one level-1 classifier (see Fig. 1) into an ensemble. The level-1 classifier trains and evaluates all of the level-0 classifiers on *the same data* and learns (using a separate learning algorithm) which of the underlying classifiers (the level-0 classifiers) that perform well on different classes and data. The level-1 training algorithm is typically a relatively simple smooth linear model (Witten et al. 2011), such as logistic regression. Note that in stacking, it is completely permissible to put other ensemble techniques as level-0 classifiers.

In this study (see Sections 5 and 6), we are using stacked generalization because this ensemble technique meets our goal of combining and evaluating *different* classifiers.

3 Related Work on Automated Bug Assignment

Several researchers have proposed automated support for bug assignment. Most previous work can either be classified as ML classification problems or *Information Retrieval* (IR) problems. ML-based bug assignment uses supervised learning to classify bug reports to the most relevant developer. IR-based bug assignment on the other hand, considers bug reports as queries and developers as documents of various relevance given the query. A handful of recent studies show that specialized solutions for automated bug assignment can outperform both ML and IR approaches, e.g., by combining information in the BTS with the source code repository, or by crafting tailored algorithms for matching bug reports and developers. We focus the review of previous work on applications of off-the-shelf classification algorithms, as our aim is to explore combinations of readily available classifiers. However, we also report key publications both from IR-based bug assignment and specialized state-of-the-art tools for bug assignment in Section 3.2.

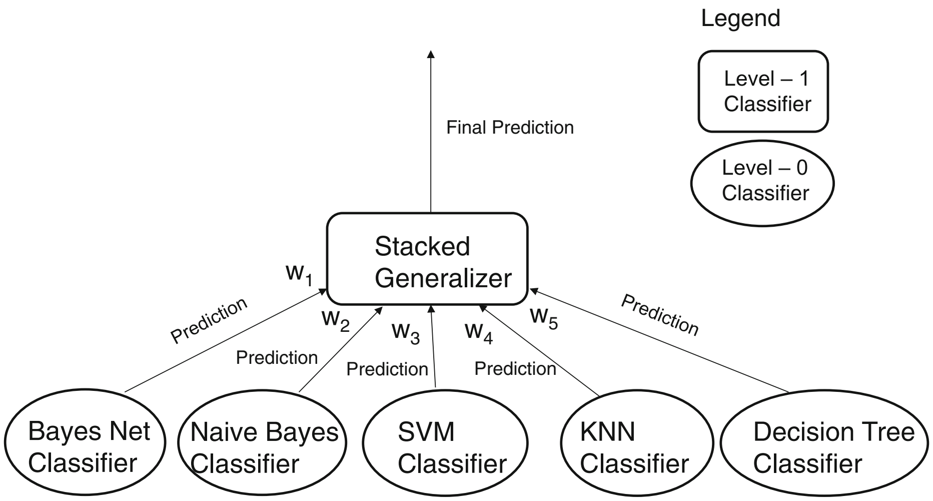


Fig. 1 Stacked Generalization

3.1 Automated Bug Assignment Using General Purpose Classifiers

Previous work on ML-based bug assignment has evaluated several techniques. Figure 2 gives a comprehensive summary of the classifiers used in previous work on ML-based bug assignment. Cubranic and Murphy (2004) pioneered the work by proposing a Naive Bayes (NB) classifier trained for the task. Anvik et al. (2006) also used NB, but also introduced Support Vector Machines (SVM), and C4.5 classifiers. Later, they extended that work and evaluated also rules-based classification and Expectation Maximization (EM) (Anvik 2007), as well as Nearest Neighbor (NN) (Anvik and Murphy 2011). Several other researchers continued the work by Anvik et al by evaluating classification-based bug assignment on bug reports from different projects, using a variety of classifiers. Ahsan et al. (2009) were the first to introduce Decision Trees (DT), RBF Network (RBF), REPTree (RT), and Random Forest (RF) for bug assignment. The same year, Jeong et al. (2009) proposed to use Bayesian Networks (BNet). Helming et al. (2011) used Neural Networks (NNet) and Constant Classifier (CC). In our work, we evaluate 28 different classifiers, as presented in Section 5.

Two general purpose classification techniques have been used more than the others, namely NB and SVM (cf. Fig. 2). The numerous studies on NB and SVM are in line with ML work in general; NB and SVM are two standard classifiers with often good results that can be considered default choices when exploring a new task. Other classifiers used in at least three previous studies on bug assignment are Bayesian Networks (BNET), and C4.5. We include both NB and SVM in our study, as presented in Section 5.

Eight of the studies using ML-based bug assignment compare different classifiers. The previously largest comparative studies of general purpose classifiers for bug assignment used seven and six classifiers, respectively, (Ahsan et al. 2009; Helming et al. 2011). We go beyond previous work by comparing more classifiers. Moreover, we propose applying ensemble learning for bug assignment, i.e., combining several different classifiers.

Figure 2 also displays the features used to represent bug reports in previous work on ML-based bug assignment. Most previous approaches rely solely on textual information, most often the title and description of bug reports. Only two of the previous studies combine textual and nominal features in their solutions. Ahsan et al. (2009) include information about product, component, and platform, and Lin et al. (2009) complement textual information with component, type, phase, priority, and submitter. In our study, we complement textual information by submitter site, submitter type, software revision, and bug priority.

Figure 3 shows an overview of the previous evaluations of automated bug assignment (including studies presented in Section 3.2). It is evident that previous work has focused on the context of Open Source Software (OSS) development, as 23 out of 25 studies have studied OSS bug reports. This is in line with general research in empirical software engineering, explained by the appealing availability of large amounts of data and the possibility of replications (Robinson and Francis 2010). While there is large variety within the OSS domain, there are some general differences from proprietary bug management that impact our work. First, the bug databases used in OSS development are typically publicly available; anyone can submit bug reports. Second, Paulson et al. (2004) report that defects are found and fixed faster in OSS projects. Third, while proprietary development often is organized in *teams*, an OSS development community rather consists of individual developers. Also, the management in a company typically makes an effort to maintain stable teams over time despite employee turnover, while the churn behavior of individual developers in OSS projects is well-known (Asklund and Bendix 2002; Robles and Gonzalez-Barahona 2006). Consequently, due to the different nature of OSS development, it is not clear to what extent

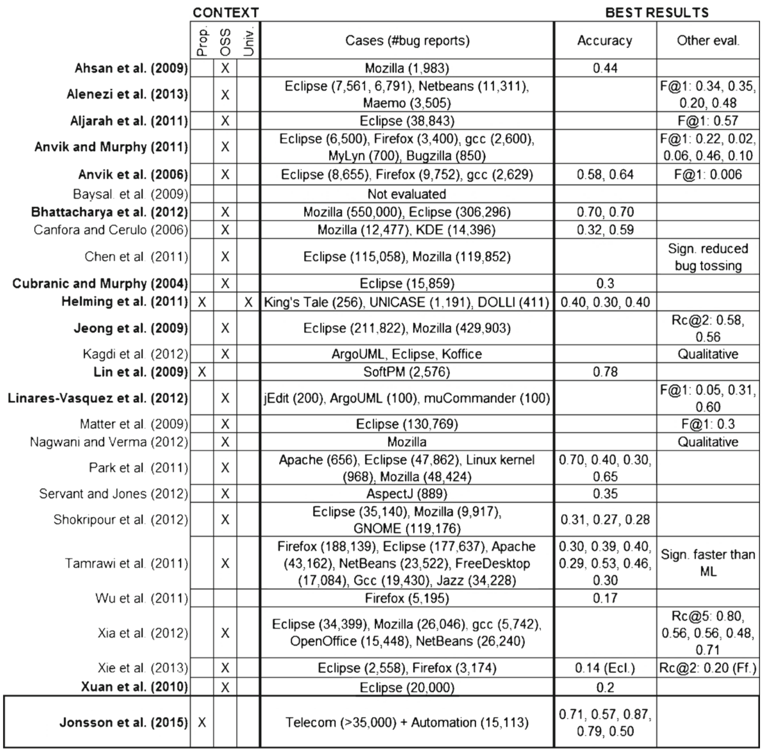
	ASSIGNMENT TECHNIQUE																FEATURES											
	Classification													IR			Textual				Nominal							
	Decision Trees	Naive Bayes	Bayesian NETworks	Nearest Neighbor	Neural Networks	RBF Network	Random Forest	REPTree	SVM	Rules	Expect. Maximization	C4.5	Constant Classifier	Algebreic	Probabilistic	RSSE	Title	Description	Comments	Identifiers	Product	Component	Platform	Version	Type	Phase	Priority	Submitter
Ahsan et al. (2009)	x	x				x	x	x	X			x					x				x	x	x					
Alenezi et al. (2013)		x															x											
Aljarah et al. (2011)			x														x											
Anvik and Murphy (2011)		x		x					X	x	x	x					x	x										
Anvik et al. (2006)		x							X			x					x	x										
Baysal. et al. (2009)																x	x	x										
Bhattacharya et al. (2012)		X	x						x			x					x	x										
Canfora and Cerulo (2006)															x		x	x	x									
Chen et al. (2011)														x			x	x										
Cubranic and Murphy (2004)		x															x	x										
Helming et al. (2011)	x	x		x	x				X				x				x	x										
Jeong et al. (2009)		x	X														x	x										
Kagdi et al. (2012)														x					x	x								
Lin et al. (2009)									x			X					x	x				x			x	x	x	x
Linares-Vasquez et al. (2012)									x					X			x	x	x	x								
Matter et al. (2009)																x		x										
Nagwani and Verma (2012)														x			x	x										
Park et al. (2011)																x		x					x	x		x		
Shokripour et al. (2012)														x					x									
Xuan et al. (2010)		x															x	x										
Jonsson et al. (2015)	x	x	x				x	x	X	x							x	x						x	x		x	x

Fig. 2 Techniques used in previous studies on ML-based bug assignment. Bold author names indicate comparative studies, capital **X** shows the classifier giving the best results. **IR** indicates Information Retrieval techniques. The last row shows the study presented in this paper

previous findings based on OSS data can be generalized to proprietary contexts. Moreover, we are not aware of any major OSS bug dataset that contains team assignments with which we can directly compare our work. This is unfortunate since it would be interesting to use the same set of tools in the two different contexts.

As the organization of developers in proprietary projects tend to be different from OSS communities, the bug assignment task we study differs accordingly. While all previous work (including the two studies on proprietary development contexts by Lin et al. (2009) and Helming et al. (2011)) aim at supporting assignment of bug reports to *individual developers*, we instead address the task of bug assignment to *development teams*. Thus, as the number of development teams is much lower than the number of developers in normal projects, direct comparisons of our results to previous work can not be made. As an example, according to Open HUB[2] (Dec 2014), the number of contributors to some of the studied OSS projects in Fig. 3 are: Linux kernel (13,343), GNOME (5,888), KDE (4,060), Firefox (3,187), NetBeans (893), gcc (534), Eclipse platform (474), Bugzilla (143), OpenOffice (124), Mylyn (92), ArgoUML (87), Maemo (83), UNICASE (83), jEdit (55), and muCommander (9). Moreover, while the number of bugs resolved in our proprietary datasets is somewhat balanced, contributions in OSS communities tend to follow the "onion model" (Aberdour 2007), i.e., the commit distribution is skewed, a few core developers contribute much source code, but most developers contribute only occasionally.

[2]Formerly Ohloh.net, an open public library presenting analyses of OSS projects (www.openhub.net).



	CONTEXT				BEST RESULTS	
	Prop.	OSS	Univ.	Cases (#bug reports)	Accuracy	Other eval.
Ahsan et al. (2009)		X		Mozilla (1,983)	0.44	
Alenezi et al. (2013)		X		Eclipse (7,561, 6,791), Netbeans (11,311), Maemo (3,505)		F@1: 0.34, 0.35, 0.20, 0.48
Aljarah et al. (2011)		X		Eclipse (38,843)		F@1: 0.57
Anvik and Murphy (2011)		X		Eclipse (6,500), Firefox (3,400), gcc (2,600), MyLyn (700), Bugzilla (850)		F@1: 0.22, 0.02, 0.06, 0.46, 0.10
Anvik et al. (2006)		X		Eclipse (8,655), Firefox (9,752), gcc (2,629)	0.58, 0.64	F@1: 0.006
Baysal et al. (2009)				Not evaluated		
Bhattacharya et al. (2012)		X		Mozilla (550,000), Eclipse (306,296)	0.70, 0.70	
Canfora and Cerulo (2006)		X		Mozilla (12,477), KDE (14,396)	0.32, 0.59	
Chen et al. (2011)		X		Eclipse (115,058), Mozilla (119,852)		Sign. reduced bug tossing
Cubranic and Murphy (2004)		X		Eclipse (15,859)	0.3	
Helming et al. (2011)	X		X	King's Tale (256), UNICASE (1,191), DOLLI (411)	0.40, 0.30, 0.40	
Jeong et al. (2009)		X		Eclipse (211,822), Mozilla (429,903)		Rc@2: 0.58, 0.56
Kagdi et al. (2012)		X		ArgoUML, Eclipse, Koffice		Qualitative
Lin et al. (2009)	X			SoftPM (2,576)	0.78	
Linares-Vasquez et al. (2012)		X		jEdit (200), ArgoUML (100), muCommander (100)		F@1: 0.05, 0.31, 0.60
Matter et al. (2009)		X		Eclipse (130,769)		F@1: 0.3
Nagwani and Verma (2012)		X		Mozilla		Qualitative
Park et al. (2011)		X		Apache (656), Eclipse (47,862), Linux kernel (968), Mozilla (48,424)	0.70, 0.40, 0.30, 0.65	
Servant and Jones (2012)		X		AspectJ (889)	0.35	
Shokripour et al. (2012)		X		Eclipse (35,140), Mozilla (9,917), GNOME (119,176)	0.31, 0.27, 0.28	
Tamrawi et al. (2011)		X		Firefox (188,139), Eclipse (177,637), Apache (43,162), NetBeans (23,522), FreeDesktop (17,084), Gcc (19,430), Jazz (34,228)	0.30, 0.39, 0.40, 0.29, 0.53, 0.46, 0.30	Sign. faster than ML
Wu et al. (2011)				Firefox (5,195)	0.17	
Xia et al. (2012)		X		Eclipse (34,399), Mozilla (26,046), gcc (5,742), OpenOffice (15,448), NetBeans (26,240)		Rc@5: 0.80, 0.56, 0.56, 0.48, 0.71
Xie et al. (2013)		X		Eclipse (2,558), Firefox (3,174)	0.14 (Ecl.)	Rc@2: 0.20 (Ff.)
Xuan et al. (2010)		X		Eclipse (20,000)	0.2	
Jonsson et al. (2015)	X			Telecom (>35,000) + Automation (15,113)	0.71, 0.57, 0.87, 0.79, 0.50	

Fig. 3 Evaluations performed in previous studies with BTS focus. Bold author names indicate studies evaluating general purpose ML-based bug assignment. Results are listed in the same order as the systems appear in the fourth column. The last row shows the study presented in this paper, even though it is not directly comparable

Bug reports from the development of Eclipse are used in 14 out of the 21 studies (cf. Fig. 3). Still, no set of Eclipse bugs has become the de facto benchmark. Instead, different subsets of bug reports have been used in previous work, containing between 6,500 and 300,000 bug reports. Bug reports originating from OSS development in the Mozilla foundation is the second most studied system, containing up to 550,000 bug reports (Bhattacharya et al. 2012). While we do not study bug repositories containing 100,000s of bug reports, our work involves much larger datasets than the previously largest study in a proprietary context by (Lin et al. 2009) (2,576 bug reports). Furthermore, we study bug reports from five different development projects in two different companies.

The most common measure to report the success in previous work is *accuracy*,[3] reported in 10 out of 21 studies. As listed in Fig. 3, prediction accuracies ranging from 0.14 to 0.78 have been reported, with an average of 0.42 and standard deviation of 0.17. This suggests that a rule of thumb could be that automated bug assignment has the potential to correctly assign almost every second bug to an individual developer.

[3]Equivalent to recall when recommending only the most probable developer, aka. the Top-1 recommendation or Rc@1.

3.2 Other Approaches to Automated Bug Assignment

Some previous studies consider bug assignment as an IR problem, meaning that the incoming bug is treated as a search query and the assignment options are the possible documents to retrieve. There are two main families of IR models used in software engineering: algebraic models and probabilistic models (Borg et al. 2014). For automated bug assignment, four studies used algebraic models (Chen et al. 2011; Kagdi et al. 2012; Nagwani and Verma 2012; Shokripour et al. 2012). A probabilistic IR model on the other hand, has only been applied by Canfora and Cerulo (2006). Moreover, only (Linares-Vasquez et al. 2012) evaluated bug assignment using both classification and IR in the same study, and reported that IR displayed the most promising results.

Most studies on IR-based bug assignment report *F-scores* instead of accuracy. In Fig. 3 we present F-scores for the first candidate developer suggested in previous work (F@1). The F-scores display large variation; about 0.60 for a study on muCommander and one of the studies of Eclipse, and very low values on work on Firefox, gcc, and jEdit. The variation shows that the potential of automated bug assignment is highly data dependent, as the same approach evaluated on different data can display large differences (Anvik and Murphy 2011; Linares-Vasquez et al. 2012). A subset of IR-based studies reports neither accuracy nor F-score. Chen et al. (2011) conclude that their automated bug assignment significantly reduces bug tossing as compared to manual work. Finally, Kagdi et al. (2012) and Nagwani and Verma (2012) perform qualitative evaluations of their approaches. Especially the former study reports positive results.

Three studies on automated bug assignment identified in the literature present tools based on content-based and collaborative filtering, i.e., techniques from research on Recommendation Systems (Robillard et al. 2014). Park et al. (2011) developed an RS where bug reports are represented by their textual description extended by the nominal features: platform, version, and development phase. Baysal et al. (2009) presented a framework for recommending developers for a given bug report, using a vector space analysis of the history of previous bug resolutions. Matter et al. (2009) matched bug reports to developers by modelling the natural language in historical commits and comparing them to the textual content of bug reports.

More recently, some researchers have showed that the accuracy of automated bug assignment can be improved by implementing more advanced algorithms, tailored for both the task and the context. Tamrawi et al. (2011) proposed Bugzie, an automated bug assignment approach they refer to as fuzzy set and cache-based. Two assumptions guide their work: 1) the textual content of bug reports is assumed to relate to a specific *technical aspect* of the software system, and 2) if a developer frequently resolves bugs related to such a technical aspect, (s)he is capable of resolving related bugs in the future. Bugzie models both technical aspects and developers' expertise as bags-of-words and matches them accordingly. Furthermore, to improve the scalability, Bugzie recommends only developers that recently committed bug resolutions, i.e., developers in the cache. Bugzie was evaluated on more than 500,000 bug reports from seven OSS projects, and achieved an prediction accuracies between 30 % and 53 %.

Wu et al. (2011) proposed DREX, an approach to bug assignment using k-nearest neighbour search and social network analysis. DREX recommends performs assignment by: 1) finding textually similar bug reports, 2) extracting developers involved in their resolution, and 3) ranking the developers expertise by analyzing their participation in resolving the similar bugs. The participation is based on developers' comments on historical bug reports, both manually written comments and comments automatically generated when source code changes are committed. DREX uses the comments to construct a social network, and approximated participation using a series of network measures. An evaluation on bug reports from the Firefox OSS project shows the social network analysis of DREX outperforms a purely textual approach, with a prediction accuracy of about 15 % and recall when considering the Top-10 recommendations (Rc@10, i.e., the bug is included in the 10 first recommendations) of 0.66.

Servant and Jones (2012) developed WhoseFault, a tool that both assigns a bug to a developer and presents a possible location of the fault in the source code. WhoseFault is also different from other approaches reported in this section, as it performs its analysis originating from failures from automated testing instead of textual bug reports. To assign appropriate developers to a failure, WhoseFault combines a framework for automated testing, a fault localization technique, and the commit history of individual developers. By finding the likely position of a fault, and identifying the most active developers of that piece of source code, WhoseFault reaches a prediction accuracy of 35 % for the 889 test cases studied in the AspectJ OSS project. Moreover, the tool reports the correct developer among the top-3 recommendations for 81.44 % of the test cases.

A trending technique to process and analyze natural language text in software engineering is *topic modeling*. Xie et al. (2012) use topic models for automated bug assignment in their approach DRETOM. First, the textual content of bug reports is represented using topic models (Latent Dirichlet Allocation (LDA) Blei et al. (2003)). Then, based on the bug-topic distribution, DRETOM maps each bug report to a single topic. Finally, developers and bug reports are associated using a probabilistic model, considering the *interest* and *expertise* of a developer given the specific bug report. DRETOM was evaluated on more than 5,000 bug reports from the Eclipse and Firefox OSS projects, and achieved an accuracy of about 15 %. However, considering the Top-5 recommendations the recall reaches 80 % and 50 % for Eclipse and Firefox, respectively.

Xia et al. (2013) developed DevRec, a highly specialized tool for automated bug assignment, that also successfully implemented topic models. Similar to the bug assignment implemented in DREX, DevRec first performs a k-nearest neighbours search. DevRec however calculates similarity between bug reports using an advanced combination of the *terms* in the bug reports, its *topic* as extracted by LDA, and the *product* and *component* the bug report is related to (referred to as BR-based analysis). Developers are then connected to bug reports based on multi-label learning using ML-KNN. Furthermore, DevRec then also models the affinity between developers and bug reports by calculating their distances (referred to as D-based analysis). Finally, the BR-analysis and the D-based analyses are combined to recommend developers for new bug reports. Xia et al. (2013) evaluated DevRec on more than 100,000 bug reports from five OSS projects, and they also implemented the approaches proposed in both DREX and Bugzie to enable a comparison The authors report average Rc@5 and Rc@10 of 62 % and 74 %, respectively, constituting considerable improvements compared to both DREX and Bugzie.

In contrast to previous work on specialised tools for bug assignment, we present an approach based on general purpose classifiers. Furthermore, our work uses standard features of bug reports, readily available in a typical BTS. As such, we do not rely on advanced operations such as mining developers' social networks, or data integration with the commit history from a separate source code repository. The reasons for our more conservative approach are fivefold:

1. Our study constitutes initial work on applying ML for automated bug assignment in proprietary contexts. We consider it an appropriate strategy to first evaluate general

purpose techniques, and then, if the results are promising, move on to further refine our solutions. However, while we advocate general purpose classifiers in this study, the way we combine them into an ensemble is novel in automated bug assignment.
2. The two proprietary contexts under study are different in terms of work processes and tool chains, thus it would not be possible to develop one specialized bug assignment solution that fits both the organizations.
3. As user studies on automated bug assignment are missing, it is unclear to what extent slight tool improvements are of practical significance for an end user. Thus, before studies evaluate the interplay between users and tools, it is unclear if specialized solutions are worth the additional development effort required. This is in line with discussions on improved tool support for trace recovery (Borg and Pfahl 2011), and the difference of *correctness* and *utility* of recommendation systems in software engineering (Avazpour et al. 2014).
4. Relying on general purpose classifiers supports transfer of research results to industry. Our industrial partners are experts on developing high quality embedded software systems, but they do not have extensive knowledge of ML. Thus, delivering a highly specialized solution would complicate both the hand-over and the future maintenance of the tool. We expect that this observation generalizes to most software intensive organizations.
5. Using general purpose techniques supports future replications in other companies. As such replications could be used to initiate case studies involving end users, a type of studies currently missing, we believe this to be an important advantage of using general purpose classifiers.

4 Case Descriptions

This section describes the two case companies under study, both of which are bigger than the vast majority of OSS projects. In OSS projects a typical power-law behavior is seen with a few projects, such as the Linux kernel, Mozilla etc, having large number of contributors. We present the companies guided by the six context facets proposed by Petersen and Wohlin (2009), namely *product*, *processes*, *practices and techniques*, *people*, *organization*, and *market*. Also, we present a simplified model of the bug handling processes used in the companies. Finally, we illustrate where in the process our machine learning system could be deployed to increase the *level of automation*, as defined by Parasuraman et al. (2000).[4]

4.1 Description of Company Automation

Company Automation is a large international company active in the power and automation sector. The case we study consists of a development organization managing hundreds of engineers, with development sites in Sweden, India, Germany, and the US. The development context is safety-critical embedded development in the domain of industrial control systems, governed by IEC 61511.[5] A typical project has a length of 12-18 months and follows an iterative stage-gate project management model. The software is certified to a Safety

[4]Ten levels of automation, ranging from 0, for fully manual work, to 10, when the computer acts autonomously ignoring the human.

[5]Functional safety - Safety instrumented systems for the process industry sector.

Integrity Level (SIL) of 2 as defined by IEC 61508[6] mandating strict processes on the development and maintenance activities. As specified by IEC 61511, all changes to safety classified source code requires a formal impact analysis before any changes are made. Furthermore, the safety standards mandate that both forward and backward traceability should be maintained during software evolution.

The software product under development is a mature system consisting of large amounts of legacy code; parts of the code base are more than 20 years old. As the company has a low staff turnover, many of the developers of the legacy code are still available within the organization. Most of the software is written in C/C++. Considerable testing takes place to ensure a very high code quality. The typical customers of the software product require safe process automation in very large industrial sites.

The bug-tracking system (BTS) in Company Automation has a central role in the change management and the impact analyses. All software changes, both source code changes and changes to documentation, must be connected to an issue report. Issue reports are categorized as one of the following: *error corrections* (i.e., bug reports), *enhancements*, *document modification*, and *internal* (e.g., changes to test code, internal tools, and administrative changes). Moreover, the formal change impact analyses are documented as attachments to individual issue reports in the BTS.

4.2 Description of Company Telecom

Company Telecom is a major telecommunications vendor based in Sweden. We are studying data from four different development organizations within Company Telecom, consisting of several hundreds of engineers distributed over several countries. Staff turnover is very low and many of the developers are senior developers that have been working on the same products for many years.

The development context is embedded systems in the Information and Communications Technology (ICT) domain. Development in the ICT domain is heavily standardized, and adheres to standards such as 3GPP, 3GPP2, ETSI, IEEE, IETF, ITU, and OMA. Company Telecom is ISO 9001 and TL 9000 certified. At the time the study was conducted, the project model was based on an internal waterfall-like model, but has since then changed to an Agile development process.

Various programming languages are used in the four different products. The majority of the code is written in C++ and Java, but other languages, such as hardware description languages, are also used.

Two of the four products are large systems in the ICT domain, one is a middleware platform, and one is a component system. Two of the products are mature with a code base older than 15 years, whereas the other two products are younger, but still older than 8 years. All four products are deployed at customer sites world-wide in the ICT market.

Issue management in the design organization is handled in two separate repositories; one for change requests (planned new features or updates) and one for bug reports. In this study we only use data from the latter, the BTS.

Customer support requests to Company Telecom are handled in a two layered approach with an initial customer service organization dealing with initial requests, called Customer Service Requests (CSR). The task of this organization is to screen incoming requests so that only hardware or software errors and no other issue, such as configuration problems, are

[6]Functional safety of Electrical/Electronic/Programmable Electronic safety-related systems.

sent down to the second layer. If the customer support organization believes a CSR to be a fault in the product, they file a bug report based on the CSR in the second layer BTS. In this way, the second layer organization can focus on issues that are likely to be faults in the software. In spite of this approach, some bug reports can be configuration issues or other problems not directly related to faults in the code. In this study, we have only used data from the second layer BTS, but there is nothing in principle that prevents the same approach to be used on the first layer CSR's. The BTS is the central point in the bug handling process and there are several process descriptions for the various employee roles. Tracking of analysis, implementation proposals, testing, and verification are all coordinated through the BTS.

4.3 State-of-Practice Bug Assignment: A Manual Process

The bug handling process of both Company Automation and Telecom are substantially more complex than the standard process described by Bugzilla (Mozilla 2013). The two processes are characterized by the development contexts of the organizations. Company Automation develops safety-critical systems, and the bug handling process must therefore adhere to safety standards as described in Section 4.1. The standards put strict requirements on how software is allowed to be modified, including rigorous change impact analyses with focus on traceability. In Company Telecom on the other hand, the sheer size of both the system under development and the organization itself are reflected on the bug handling process. The resource allocation in Company Telecom is complex and involves advanced routing in a hierarchical organization to a number of development teams.

We generalize the bug handling processes in the two case companies and present an overview model of the currently manual process in Fig. 4. In general, three actors can file bug reports: i) the developers of the systems, ii) the internal testing organization, and iii) customers that file bug reports via helpdesk functions. A submitted bug report starts in a *bug triaging stage*. As the next step, the Change Control Board (CCB) assigns the bug report to a development team for investigation. The leader of the receiving team then assigns the bug report to an individual developer. Unfortunately, the bug reports often end up with the wrong developer, thus *bug tossing* (i.e., bug report re-assignment) is common, especially between teams. The longer history the BTS stores about the bug tossing that takes place, the more detailed estimate of the savings of our approach can be done. With only the last entry saved in the BTS one can estimate the prediction accuracy of the system but to calculate the full saving, the full bug tossing history is needed.

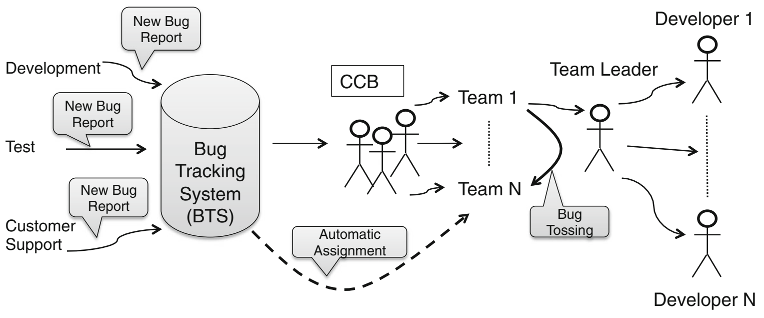


Fig. 4 A simplified model of bug assignment in a proprietary context

4.4 State-of-the-Art: Automated Bug Assignment

We propose, in line with previous work, to automate the bug assignment. Our approach is to use the historical information in the BTS as a labeled training set for a classifier. When a new bug is submitted to the BTS, we encode the available information as features and feed them to a prediction system. The prediction system then classifies the new bug to a specific development team. While this resembles proposals in previous work, our approach differs by: i) aiming at supporting large proprietary organizations, and ii) assigning bug reports to teams rather than individual developers.

Figure 4 shows our automated step as a dashed line. The prediction system offers decision support to the CCB, by suggesting which development team that is the most likely to have the skills required to investigate the issue. This automated support corresponds to a medium level of automation ("the computer suggests one alternative and executes that suggestion if the human approves"), as defined in the established automation model by Parasuraman et al. (2000).

5 Method

The overall goal of our work is to support bug assignment in large proprietary development projects using state-of-the-art ML. As a step toward this goal, we study five sets of bug reports from two companies (described in Section 4), including information of team assignment for each bug report. We conduct controlled experiments using Weka (Hall et al. 2009), a mature machine learning environment that is successfully used across several domains, for instance, bioinformatics (Frank et al. 2004), telecommunication (Alshammari and Zincir-Heywood 2009), and astronomy (Zhao and Zhang 2008). This section describes the definition, design and setting of the experiments, following the general guidelines by Basili et al. (1986) and Wohlin et al. (2012).

5.1 Experiment Definition and Context

The *goal* of the experiments is to study automatic bug assignment using stacked generalization in large proprietary development contexts, for the *purpose* of evaluating its industrial feasibility, from the *perspective* of an applied researcher, planning deployment of the approach in an industrial setting.

Table 1 reminds the reader of our RQs. Also, the table presents the rationale of each RQ, and a high-level description of the research approach we have selected to address them. Moreover, the table maps the RQs to the five sub-experiments we conduct, and the experimental variables involved.

5.2 Data Collection

We collect data from one development project at Company Automation and four major development projects at Company Telecom. While the bug tracking systems in the two companies show many similarities, some slight variations force us to perform actions to consolidate the input format of the bug reports. For instance, in Company Automation a bug report has a field called "Title", whereas the corresponding field in Company Telecom is called "Heading". We align these variations to make the semantics of the resulting fields the same for all datasets. The total number of bug reports in our study is 15,113 + 35,266 = 50,379. Table 2 shows an overview of the five datasets.

Table 1 Overview of the research questions, all related to the task of automated team allocation

	RQ1	RQ2	RQ3	RQ4
Description	Does stacked generalization outperform individual classifiers?	How does the ensemble selection in SG affect the prediction accuracy?	How consistent learning curves does SG display across projects?	How does the time locality of training data affect the prediction accuracy?
Rationale	Confirm the result of our previous work (Jonsson et al, 2012).	Explore which ensemble selection performs the best.	Study how SG performs on different data, and understand how much training data is required.	Understand how SG should be retrained as new bug reports are submitted.
Approach	Test the hypothesis: "SG does not perform better than individual classifiers wrt. prediction accuracy".	Based on RQ1: evaluate three different ensemble selections.	Using the best ensemble selection from RQ2: evaluate learning curves.	Using the best ensemble selection from RQ2 with amount of training data from RQ3: evaluate SG sensitivity to freshness of training data.
Related experiments	Exp A, Exp B	Exp B	Exp C	Exp D, Exp E
Dependent variable	Prediction accuracy			
Independent variables	Individual classifier	Ensemble selection	Size of training set	Time locality of training data (Exp D), size of training set (Exp E)
Fixed variables	Preprocessing, feature selection, training size		Preprocessing, feature selection, ensemble selection	Preprocessing, feature selection, ensemble selection

Each question is listed along with the main purpose of the question, a high-level description of our study approach, and the experimental variables involved

We made an effort to extract similar sets of bug reports from the two companies. However, as the companies use different BTSs, and interact with them according to different processes, slight variations in the extraction steps are inevitable. Company Automation uses a BTS from an external software vendor, while Company Telecom uses an internally developed BTS. Moreover, while the life-cycles of bug reports are similar in the two companies (as described in Section 4.3), they are not equivalent. Another difference is that Company Automation uses the BTS for issue management in a broader sense (incl. new feature development, document updates, and release management), Company Telecom uses the BTS for bug reports exclusively. To harmonize the datasets, we present two separate filtering sequences in Sections 5.2.1 and 5.2.2.

Table 2 Datasets used in the experiments

Dataset	#Bug reports	Timespan	#Teams
Automation	15,113	July 2000 – Jan 2012	67
Telecom 1	$>$ 9,000	$>$ 5 years	28
Telecom 2	$>$ 8,000	$>$ 5 years	36
Telecom 3	$>$ 3,000	$>$ 5 years	17
Telecom 4	$>$ 10,000	$>$ 5 years	64
Total	$>$ 50,000		

Note: At the request of our industry partners the table only lists lower bounds for Telecom systems, but the total number of sums up to an excess of 50,000 bug reports

5.2.1 *Company Automation Data Filtering*

The dataset from Company Automation contains in total 26,121 bug reports submitted between July 2000 and January 2012, all related to different versions of the same software system. The bug reports originate from several development projects, and describe issues reported concerning a handful of different related products. During the 12 years of development represented in the dataset, both the organization and processes have changed towards a more iterative development methodology. We filter the dataset in the following way:

1. We included only CLOSED bug reports to ensure that all bugs have valid team assignments, that is, we filter out bug reports in states such as OPEN, NO ACTION, and CHANGE DEFERRED. This step results in 24,690 remaining bug reports.
2. We exclude bug reports concerning requests for new features, document updates, changes to internal test code, and issues of administrative nature. Thus, we only keep bug reports related to source code of the software system. The rationale for this step is to make the data consistent with Company Telecom, where the BTS solely contains bug reports. The final number of bug reports in the filtered dataset is 15,113.

5.2.2 *Company Telecom Data Filtering*

Our first step of the data filtering for Company Telecom is to identify a timespan characterized by a stable development process. We select a timespan from the start of the development of the product family to the point in time when an agile development process is introduced (Wiklund et al. 2013). The motivation for this step is to make sure that the study is conducted on a conformed data set. We filter the bug reports in the timespan according to the following steps:

1. We include only bug reports in the state FINISHED.
2. We exclude bug reports marked as duplicates.
3. We exclude bug reports that do not result in an source code update in a product.

After performing these three steps, the data set for the four products contains in total 35,266[7] bug reports.

5.3 ML Framework Selection

To select a platform for our experiments, we study features available in various machine learning toolkits. The focus of the comparison is to find a robust, well tested, and comparatively complete framework. The framework should also include an implementation of stacked generalizer and it should be scalable. As a consequence, we focus on platforms that are suitable for distributed computation. Another criterion is to find a framework that has implemented a large set of state-of-the-art machine learning techniques. With the increased attention of machine learning and data mining, quite a few frameworks have emerged during the last couple of years such as Weka (Hall et al. 2009), RapidMiner (Hofmann and Klinkenberg 2013), Mahout (Owen et al. 2011), MOA (Bifet et al. 2010), Mallet (McCallum 2002), Julia (Bezanson et al. 2012), and Spark (Zaharia et al. 2010) as well as increased visibility of established systems such as SAS, SPSS, MATLAB, and R.

[7]Due to confidentiality reasons these numbers are not broken down in exact detail per project.

For this study, we select to use a framework called Weka (Hall et al. 2009). Weka is a comparatively well documented framework with a public Java API and accompanying book, website, forum, and active community. Weka has many ML algorithms implemented and it is readily extensible. It has several support functionalities, such as cross-validation, stratification, and visualization. Weka has a built-in Java GUI for data exploration and it is also readily available as a stand alone library in JAR format. It has some support for parallelization. Weka supports both batch and online interfaces for some of its algorithms. The meta facilities of the Java language also allows for mechanical extraction of available classifiers. Weka is a well established framework in the research community and its implementation is open source.

5.4 Bug Report Feature Selection

This section describes the feature selection steps that are common to all our data sets. We represent bug reports using a combination of textual and nominal features. Feature selections that are specific to each individual sub-experiment are described together with each experiment.

For the textual features, we limit the number of words because of memory and execution time constraints. To determine a suitable number of words to keep, we run a series of pilot experiments, varying the method and number of words to keep, by varying the built in settings of Weka. We decide to represent the text in the bug reports as the 100 words with highest TF-IDF[8] as calculated by the Weka framework. Furthermore, the textual content of the titles and descriptions are not separated. There are two reasons for our rather simple treatment of the natural language text. First, Weka does not support multiple bags-of-words; such a solution would require significant implementation effort. Second, our focus is not on finding ML configurations that provide the optimal prediction accuracies for our datasets, but rather to explore SG for bug assignment in general. We consider optimization to be an engineering task during deployment.

The non-textual fields available in the two bug tracking systems vary between the companies, leading to some considerations regarding the selection of non-textual features. Bug reports in the BTS of Company Automation contain 79 different fields; about 50% of these fields are either mostly empty or have turned obsolete during the 12 year timespan. Bug reports in the BTS of Company Telecom contain information in more than 100 fields. However, most of these fields are empty when the bug report is submitted. Thus, *we restricted the feature selection to contain only features available at the time of submission of the bug report*, i.e., features that do not require a deeper analysis effort (e.g., faulty component, function involved). We also want to select a small set of general features, likely to be found in most bug tracking systems. Achieving feasible results using a simple feature selection might simplify industrial adaptation, and also it limits the ML training times. Based on discussions with involved developers, we selected the features presented in Table 3. In the rest of the paper, we follow the nomenclature in the leftmost column.

A recurring discussion when applying ML concerns which features are the best for prediction. In our BTS'es we have both textual and non-textual features, thus we consider it valuable to compare the relative predictive power of the two types of features. While out previous research has indicated that including non-textual features improves the prediction

[8]Term Frequency-Inverse Document Frequency (TF-IDF) is a standard weighting scheme for information retrieval and text mining. This scheme is common in software engineering applications (Borg et al. 2014).

accuracy (Jonsson et al. 2012), many other studies rely solely on the text (see Fig. 2). To motivate the feature selection used in this study, we performed a small study comparing textual vs. non-textual features for our five datasets.

Figure 5 shows the results from our small feature selection experiment. The figure displays results from three experimental runs, all using SG with the best individual classifiers (further described in Section 6.1). The three curves represent three different sets of features: 1) textual and non-textual features, 2) non-textual features only, and 3) textual features only. The results show that for some systems (Telecom 1, 2 and 4) the non-textual features performs better than the textual features alone, while for some systems (Telecom 3 and Automation) the results are the opposite. Thus, our findings strongly suggest that we should combine both non-textual features and textual features for bug assignment.

As stated earlier, the main goal of this work is not to optimize classifier performance. We suspect, however, that more sophisticated text modeling techniques, such as LDA (Blei et al. 2003), may improve the overall classification result.

5.5 Experiment Design and Procedure

Figure 6 shows an overview of our experimental setup. The five datasets originate from two different industrial contexts, as depicted by the two clouds to the left. We implement five sub-experiments (c.f. A–E in Fig. 6), using the Weka machine learning framework. Each sub-experiment is conducted once per dataset, that is, we performed 25 experimental runs. A number of steps implemented in Weka are common for all experimental runs:

1. The complete dataset set of bug reports is imported.
2. Bug reports are divided into training and test sets. In sub-experiments A–C, the bug reports are sampled using stratification. The process in sub-experiments D–E is described in Sections 5.5.4 and 5.5.5.
3. Feature extraction is conducted as specified in Section 5.4.

We executed the experiments on two different computers. We conduct experiments on the Company Automation dataset on a Mac Pro, running Mac OS X 10.7.5, equipped with

Table 3 Features used to represent bug reports. For company Telecom the fields are reported for Telecom 1,2,3,4 respectively

	Company Automation	Company Telecom	Description
		Textual features	
Text	Title+Description	Heading+Observation	One line summary and full description of the bug report
		Nominal features	
SubmitterType #Possible values	SubmitterClass 17	Customer >170,>50,>120,>150	Affiliation of the issue submitter
Site #Possible values	SubmitterSite 14	SiteId >250,>60,>80,>200	Site from where the bug was submitted
Revision #Possible values	Revision 103	Faulty revision 547,1325,999,982	Revision of the product that the bug was reported on
Priority #Possible values	Priority 5	Priority 3,3,3,3	Priority of the bug

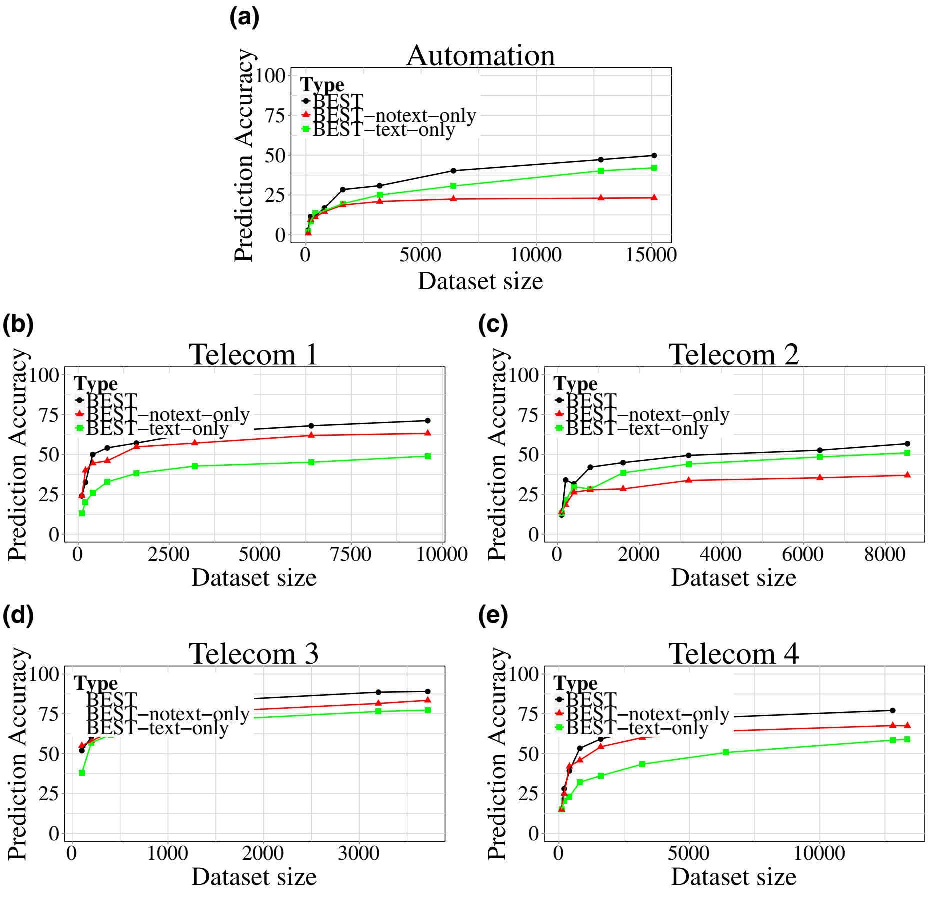


Fig. 5 The prediction accuracy when using text only features ("text-only") vs. using non-text features only ("notext-only")

24 GB RAM and two Intel(R) Xeon(R) X5670 2.93 GHz CPUs with six cores each. The computer used for the experiments on the Company Telecom datasets had the following specification: Linux 2.6.32.45-0.3-xen, running SUSE LINUX, equipped with eight 2.80 GHz Intel(R) Xeon(R) CPU and 80 GB RAM.

As depicted in Fig. 6, there are dependencies among the sub-experiments. Several sub-experiments rely on results from previous experimental runs to select values for both fixed and independent variables. Further details are presented in the descriptions of the individual sub-experiments A–E.

We evaluate the classification using a top-1 approach. That is, we only consider a correct/incorrect classification, i.e., we do not evaluate whether our approach correctly assigns bug reports to a set of candidate teams. In IR evaluations, considering ranked output of search results, it is common to assess the output at different cut-off points, e.g., the top-5 or top-10 search hits. Also some previous studies on bug assignment present top-X evaluations inspired by IR research. However, our reasons for a top-1 approach are three-fold: First, for fully automatic bug assignment a top-1 approach is the only reasonable choice,

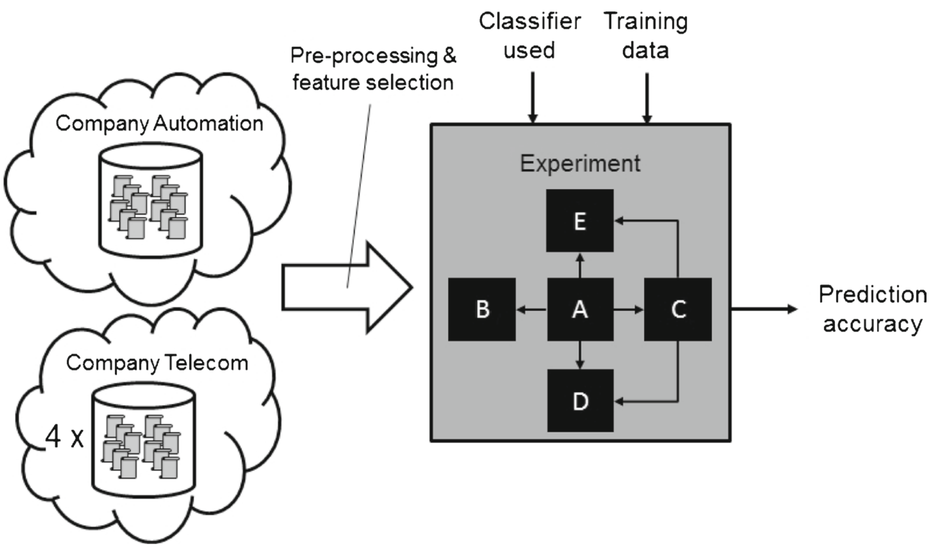


Fig. 6 Overview of the controlled experiment. *Vertical arrows* depict independent variables, whereas the *horizontal arrow* shows the dependent variable. *Arrows* within the experiment box depict dependencies between experimental runs A–E: Experiment A determines the composition of individual classifiers in the ensembles studied evaluated in Experiment B-E. The appearance of the learning curves from Experiment C is used to set the size of the time-based evaluations in Experiment D and Experiment E

since an automatic system would not send a bug report to more than one team. Second, a top-1 approach is a conservative choice in the sense that the classification results would only improve with a top-k approach. The third motivation is technical; to ensure high quality evaluations we have chosen to use the built-in mechanisms in the well established Weka. Unfortunately, Weka does not support a top-k approach in its evaluation framework for classifiers.

5.5.1 Experiment A: Individual Classifiers

Independent Variable: Choice Of Individual Classifier Experiment A investigates RQ1 and the null hypothesis “SG does not perform better than individual classifiers wrt. prediction accuracy”. We evaluate the 28 available classifiers in Weka Development version 3.7.9, listed in Table 4. The list of possible classifiers is extracted by first listing all classes in the corresponding .jar file in the “classifier” package and then trying to assign them one by one to a classifier. The independent variable is the *individual classifier*. For all five datasets, we execute 10-fold cross-validation once per classifier. We use all available bug reports in each dataset and evaluated all 28 classifiers on all datasets. The results of this experiment is presented in Section 6.1.

5.5.2 Experiment B: Ensemble Selection

Independent Variable: Ensemble Selection Experiment B explores both RQ1 and RQ2, i.e., both if SG is better than individual classifiers and which ensemble of classifiers to choose for bug assignment. As evaluating all combinations of the 28 individual classifiers in Weka is not feasible, we restrict our study to investigate three ensemble selections, each

Table 4 Individual classifiers available in Weka Development version 3.7.9

bayes.	functions.	lazy.	rules.	trees.	misc.
BayesNet	**Logistic**	IBk	DecisionTable	DecisionStump	InputMappedClassifier
NaiveBayes	**MultilayerPerceptron**	KStar	JRip	J48	
NaiveBayesMultinomial	**SimpleLogistic**	**LWL**	OneR	**LMT**	
NaiveBayesMultinomialText	SMO	ZeroR	PART	RandomForest	
NaiveBayesMultinomialUpdateable				RandomTree	
NaiveBayesUpdateable				REPTree	
net.BayesNetGenerator					
net.BIFReader					
net.EditableBayesNet					

Column headings show package names in Weka. Classifiers in bold are excluded from the study because of long training times or exceeding memory constraints combining five individual classifiers. We chose five as the number of individual classifiers to use in SG at a consensus meeting, based on experiences of prediction accuracy and run-time performance from pilot runs. Moreover, we exclude individual classifiers with run-times longer than 24 hours in Experiment A, e.g., MultiLayerPerceptron and SimpleLogistic.

Based on the results from Experiment A, we select three ensembles for each dataset (cf. Table 4). We refer to these as BEST, WORST, and SELECTED. We chose the first two ensembles to test the hypothesis "combining the best individual classifiers should produce a better result than if you choose the worst". The BEST ensemble consists of the five individual classifiers with the highest prediction accuracy from Experiment A. The WORST ensemble contains the five individual classifiers with the lowest prediction accuracy from Experiment B, while still performing better than the basic classifier *ZeroR* that we see as a lower level baseline. The ZeroR classifier simply always predicts the class with the largest number of bugs. No classifier with a lower classification accuracy than ZeroR is included in any ensemble, thus the ZeroR acts as a lower level cut-off threshold for being included in an ensemble.

The SELECTED ensemble is motivated by a discussion in Wolpert (1992), who posits that diversity in the ensemble of classifiers improves prediction results. The general idea is that if you add similar classifiers to a stacked generalizer, less new information is added compared to adding a classifier based on a different classification approach. By having level-0 generalizers of different types, they together will better "span the learning space". This is due to the fundamental theory behind stacked generalization, claiming that the errors of the individual classifiers should average out. Thus, if we use several similar classifiers we do not get the averaging out effect since then, in theory, the classifiers will have the same type of errors and not cancel out. We explore this approach by using the ensemble selection call SELECTED, where we combine the best individual classifiers from five different classification approaches (the packages in Table 4). The results of this experiment is presented in Section 6.1.

Some individual classifiers are never part of a SG. This depends on either that the classifier did not pass the cut-off threshold of being better than the *ZeroR* classifier, this case occurs for instance for the *InputMappedClassifier* (see Table 4). Alternatively the classifier was neither bad enough to be in the WORST ensemble nor good enough to be in the BEST or SELECTED, this is the case with for instance *JRip*.

In all of the ensembles we use SimpleLogistic regression as the level-1 classifier following the general advice of Wolpert (1992) and Witten et al. (2011) of using a relatively simple smooth linear model.

We choose to evaluate the individual classifiers on the whole dataset in favor of evaluating them on a hold-out set, i.e., a set of bug reports that would later not be used in the evaluation of the SG. This is done to maximize the amount of data in the evaluation of the SG. It is important to note that this reuse of data only applies to the selection of which individual classifiers to include in the ensemble. In the evaluation of the SG, all of the individual classifiers are completely retrained on only the training set, and none of the data points in the test set is part of the training set of the individual classifiers. This is also the approach we would suggest for industry adoption, i.e., first evaluate the individual classifiers on the current bug database, and then use them in a SG.

5.5.3 Experiment C: Learning Curves

Independent Variable: Amount of Training Data The goal of Experiment C is to study RQ3: *How consistent learning curves does SG display across projects?* For each dataset,

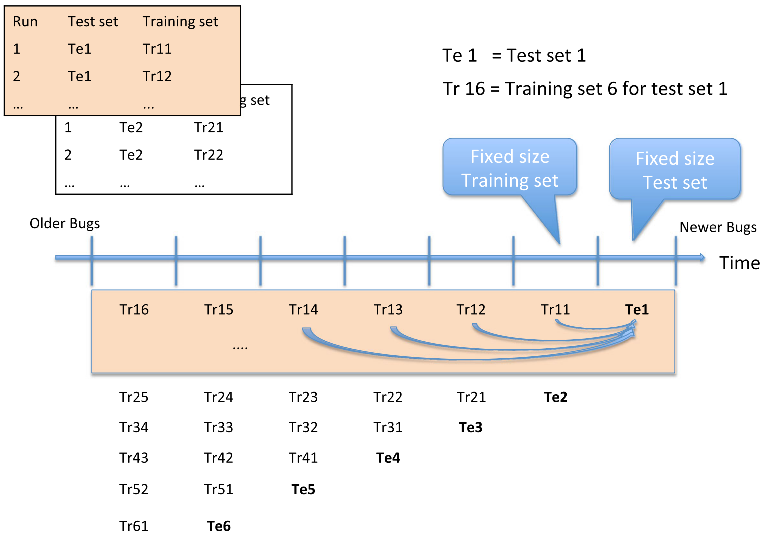


Fig. 7 Overview of the time-sorted evaluation. *Vertical bars* show how we split the chronologically ordered data set into training and test sets. This approach gives us many measurement points in time per test set size. Observe that the *time* between the different sets can vary due to non-uniform bug report inflow but the *number* of bug reports between each vertical bar is fixed

we evaluate the three ensembles from Experiment B using fixed size subsets of the five datasets: 100, 200, 400, 800, 1600, 3200, 6400, 12800, and ALL bug reports. All subsets are selected using random stratified sampling from the full dataset. As the datasets Telecom 1-3 contain fewer bug reports than 12800, the learning curves are limited accordingly. The results of this experiment is presented in Section 6.2.

5.5.4 *Experiment D: Sliding Time Window*

Independent Variable: Time Locality of Training Data Experiment D examines RQ4, which addresses how the time locality of the training set affects the prediction accuracy on a given test set. Figure 7 shows an overview of the setup of Experiment D. The idea is to use training sets of the same size increasingly further back in time to predict a given test set. By splitting the chronologically ordered full data set into fixed size training and test sets according to Fig. 7, we can generate a new dataset consisting of pairs (x, y). In this dataset, x represents the time difference measured in number of days (delta time) between the start of the training set and the start of the test set. The y variable represents the prediction accuracy of using the training set x days back in time to predict the bug assignments in the selected test set. We can then run a linear regression on the data set of delta time and prediction accuracy samples and examine if there is a negative correlation between delta time and prediction accuracy.

We break down RQ4 further into the following research hypothesis formulation: “Is training data further back in time worse at predicting bug report assignment than training data closer in time”? We test this research hypothesis with the statistical method of simple linear regression. Translated into a statistical hypothesis RQ4 is formulated as:

> Let the difference in time between the submission date of the first bug report in a test set and the submission date of the first bug report in the training set be the independent

variable x. Further, let the prediction accuracy on the test set be the dependent variable y. Is the coefficient of the slope of a linear regression fit on x and y statistically different from 0 and negative at the 5 % α level?

To create the training set and test sets, we sort the complete dataset in chronological order on the bug report submission date. To select suitable sizes to split the training set and test sets, we employ the following procedure. For the simple linear regression, we want to create enough sample points to be able to run a linear regression with enough power to detect a significant difference and still have as large training and test sets as possible to reduce the variance in the generated samples. Green (1991) suggests the following formula : $N \geq 50 + 8\,m$ as a rule of thumb for calculating the needed number of samples at α level of 5 % and β level of 20 %, where m is the number of independent variables. In our case we have one independent variable (delta time) so the minimum number of samples in our case is $58 = 50 + 8 * 1$. We use a combination of theoretical calculations for the lower and upper bounds on the number of training samples given that we want an 80/20 ratio of training to test data. We combine the theoretical approach with a program that calculates the number of sample points generated by a given training and test set size, by simulating runs. This combination together with Green's formula let us explore the most suitable training and test sets for the different systems.

We also know from Experiment C that the "elbow" where the prediction accuracy tends to level out is roughly around 1,000-2,000 samples, this together with the calculations for the linear regression guided our decision for the final selection of sample size.

We arrived at the following dataset sizes by exploring various combinations with the simulation program, the theoretical calculations and the experience from Experiment C. For the smallest of the systems, the maximum sizes of training and test sets that gives more than 58 samples amounts to 619 and 154 bug reports respectively. For the larger systems, we can afford to have larger data sets. For comparison we prioritize to have the same sized sets for all the other systems. When we calculate the set sizes for the smallest of the larger systems, we arrived at 1,400 and 350 bug reports for the training and test set sizes, respectively. These values are then chosen for all the other four systems. The results of this analysis is presented in Section 6.3.

5.5.5 *Experiment E: Cumulative Time Window*

Independent Variable: Amount of Training Data Experiment E is also designed to investigate RQ4, i.e., how the time locality of the training set affects the prediction accuracy. Instead of varying the training data using a fixed size sliding window as in Experiment D, we fix the starting point and vary the amount of the training data. The independent variable is the cumulatively increasing *amount of training data*. This experimental setup mimics realistic use of SG for automated bug assignment.

Figure 8 depicts an overview of Experiment E. We sort the dataset in chronological order on the issue submission date. Based on the outcome from Experiment C, we split the dataset into a corresponding number of equally sized chunks. We used each chunk as a test set, and for each test set we vary the number of previous chunks used as training set. Thus, the *amount of training data* was the independent variable. We refer to this evaluation approach as *cumulative time window*. Our setup is similar to the "incremental learning" that Bhattacharya et al. (2012) present in their work on bug assignment, but we conduct a more thorough evaluation. We split the data into training and test sets in a more comprehensive manner, and thus conduct several more experimental runs. The results of this experiment is presented in Section 6.4.

6 Results and Analysis

6.1 Experiment A: Individual Classifiers and Experiment B: Ensemble Selection

Experiment A investigates whether SG outperforms individual classifiers. Table 5 shows the individual classifier performance for the five evaluated systems. It also summarizes the results of running SG with the three different configurations BEST, WORST, and SELECTED, related to Experiment B. In Table 5 we can view the classifier "rules.ZeroR" as a sort of lower baseline reference. The ZeroR classifier simply always predicts the class with the highest number of bug reports.

The answer to RQ1 is that *while the improvements in some projects are marginal, using reasonable ensemble selection leads to a better prediction accuracy than using any of the individual classifiers*. On our systems, the improvement is 3 % better than the best of the individual classifiers on two of the systems. The best improvement is 8 % on the Automation system and the smallest improvement is 1 % on system Telecom 1 and 4, which can be considered negligible. This conclusion must be followed by a slight warning; mindless ensemble selection together with bad luck can lead to *worse* result than some of the individual classifiers. In none of our runs (including with the WORST ensemble) is the stacked generalizer worse than all of the individual classifiers.

Experiment B addresses different ensemble selections in SG. From Table 5 we see that in the cases of the BEST and SELECTED configurations the stacked generalizer in general performs as well, or better, than the individual classifiers. In the case of Telecom 1

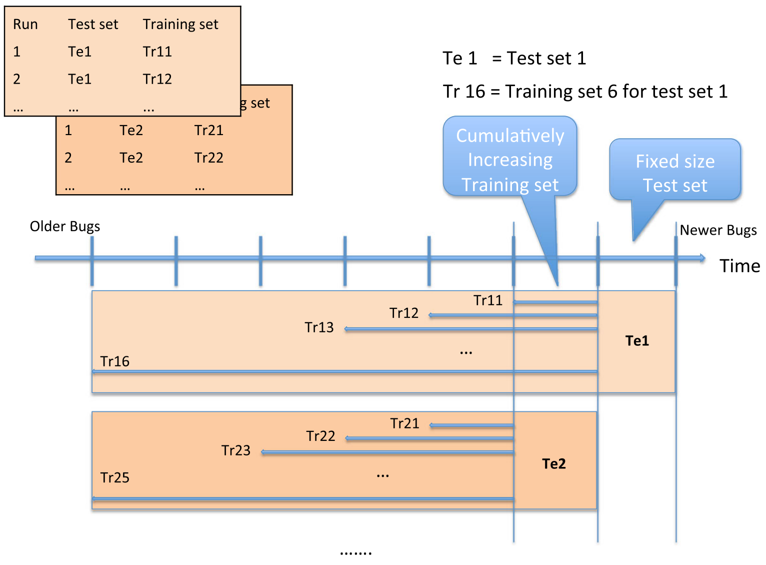


Fig. 8 Overview of the cumulative time-sorted evaluation. We use a fixed test set, but cumulatively increase the training set for each run

Table 5 Individual classifier results (rounded to two digits) on the five systems use the full data set and 10-fold cross validation

	Accuracy				
Classifier	Automation	Telecom 1	Telecom 2	Telecom 3	Telecom 4
bayes.BayesNet	35 % (B,S)	O-MEM	O-MEM	O-MEM	O-MEM
bayes.NaiveBayes	15 % (W)	25 % (W)	18 % (W)	35 %	17 %
bayes.NaiveBayesMultinomial	22 %	34 % (W)	32 %	53 % (W)	26 % (W)
bayes.NaiveBayesMultinomialText	6 %	13 %	16 %	43 %	19 %
bayes.NaiveBayesMultinomialUpdateable	26 %	34 %	32 % (S)	61 % (W)	28 % (W)
bayes.NaiveBayesUpdateable	15 %	25 %	18 %	35 %	17 %
bayes.net.BIFReader	35 %	O-MEM	O-MEM	O-MEM	O-MEM
bayes.net.BayesNetGenerator	35 %	41 % (S)	31 % (W)	66 % (S,W)	37% (S)
bayes.net.EditableBayesNet	35 %	O-MEM	O-MEM	O-MEM	O-MEM
functions.SMO	**42 %** (B,S)	**70 %** (B,S)	**54 %**(B,S)	**86 %** (B,S)	**78 %** (B,S)
lazy.IBk	38 % (B)	58 % (S)	44 % (B)	77 % (B)	63 %
lazy.KStar	42 % (B,S)	50 %	46 % (B,S)	77 % (S)	60 % (S)
lazy.LWL	9 % (W)	21 % (W)	O-MEM	O-MEM	O-MEM
misc.InputMappedClassifier	6 %	13 %	16 %	43 %	19 %
rules.DecisionTable	26 %	52 %	31 % (W)	65 % (W)	55 %
rules.JRip	23 %	51 %	36 %	73 %	55 %
rules.OneR	13 % (W)	43 % (W)	30 % (W)	71 %	50 % (W)
rules.PART	29 % (S)	61 % (B,S)	38 % (S)	76 % (S)	64 % (B,S)
rules.ZeroR	6 %	13 %	16 %	43 %	19 %
trees.DecisionStump	7 % (W)	21 % (W)	22 % (W)	44 % (W)	20 % (W)
trees.J48	30 %	62 % (B)	40 % (B)	78 % (B)	66 % (B)
trees.LMT	O-MEM	O-MEM	O-MEM	O-MEM	O-MEM
trees.REPTree	29 %	62 % (B)	34 %	79 % (B)	67 % (B)
trees.RandomForest	39 % (B,S)	63 % (B,S)	49 % (B,S)	84 (B,S) %	67 % (S)
trees.RandomTree	27 %	52 %	32 %	69 %	49 % (W)
functions.Logistic	O-MEM	O-MEM	O-MEM	O-MEM	O-MEM
functions.SimpleLogistic	40 %	O-TIME	52 %	O-TIME	O-TIME
functions.MultilayerPerceptron	20 % (W)	O-TIME	O-TIME	O-TIME	O-TIME
SG BEST (B)	50 %	71 %	57 %	89 %	77 %
SG SELECTED (S)	50 %	71 %	57 %	89 %	79 %
SG WORST (W)	28 %	57 %	45 %	83 %	62 %

Out of memory is marked O-MEM and an execution that exceeds a time threshold is marked O-TIME

and 4, there is a negligible difference between the best individual classifier SMO and the SELECTED and BEST SG. We also see that when we use the WORST configuration the result of the stacked generalizer is worse than the best of the individual classifiers, but it still performs better than some of the individual classifiers. When it comes to the individual classifiers we note that the SMO classifier performs best on all systems. The conclusion is that the SG does not do worse than any of the individual classifiers but can sometimes perform better.

Figure 9 shows the learning curves (further presented in relation to Experiment C) for the five datasets using the three configurations BEST, WORST, and SELECTED. The figures illustrate that the two ensembles BEST and SELECTED have very similar performance across the five systems. Also, it is evident that the WORST ensemble levels out at a lower prediction accuracy than the BEST and SELECTED ensembles as the number of training examples grows and the rate of increase has stabilized.

Experiment B shows *no significant difference in the prediction accuracy between BEST and SELECTED*. Thus, our results do *not* confirm that prediction accuracy is improved by applying ensemble selections with a diverse set of individual classifiers. One possible explanation for this result is that the variation among the individual classifiers in the BEST ensemble already is enough to obtain a high prediction accuracy. There is clear evidence that the WORST ensemble performs worse than BEST and SELECTED. As a consequence, *simply using SG does not guarantee good results—the ensemble selection plays an important role*.

6.2 Experiment C: Learning Curves

In Experiment C, we study how consistent the learning curves for SG are across different industrial projects. Figure 10 depicts the learning curves for the five systems. As presented in Section 5.5.3, the BEST and SELECTED ensembles yield similar prediction accuracy, i.e., the learning curves in Fig. 10a and c are hard to distinguish by the naked eye. Also, while there are performance differences across the systems, the learning curves for all five systems follow the same general pattern: the learning curves appear to follow a roughly logarithmic form proportional to the size of the training set, but with different minimum and maximum values.

An observation of practical value is that the learning curves tend to flatten out within the range of 1,600 to 3,200 training examples for all five systems. We refer to this breakpoint as where the graph has the maximum curvature, i.e., the point on the graph where the tangent curve is the most sensitive to moving the point to nearby points. For our study, it is sufficient to simply determine the breakpoint by looking at Fig. 10, comparable to applying the "elbow method" to find a suitable number of clusters in unsupervised learning (Tibshirani et al. 2001). Our results suggest that at least 2,000 training examples should be used when training a classifier for bug assignment.

We answer RQ3 as follows: *the learning curves for the five systems have different minimum and maximum values, but display similar shape and all flatten out at roughly 2,000 training examples. There is a clear difference between projects.*

6.3 Experiment D: Sliding Time Window

Experiment D targets how the time locality of the training data affects the prediction accuracy (RQ4). Better understanding of this aspect helps deciding the required frequency of retraining the classification model. Figure 11 show the prediction accuracy of using SG with the BEST ensemble, following the experimental design described in Section 5.5.4. The X axes denote the difference in time, measured in days, between the start of the training set and the start of the test set. The figures also depict an exponential best fit.

For all datasets, the prediction accuracy decreases as older training sets are used. The effect is statistically significant for all datasets at a 5 % level. We observe the highest

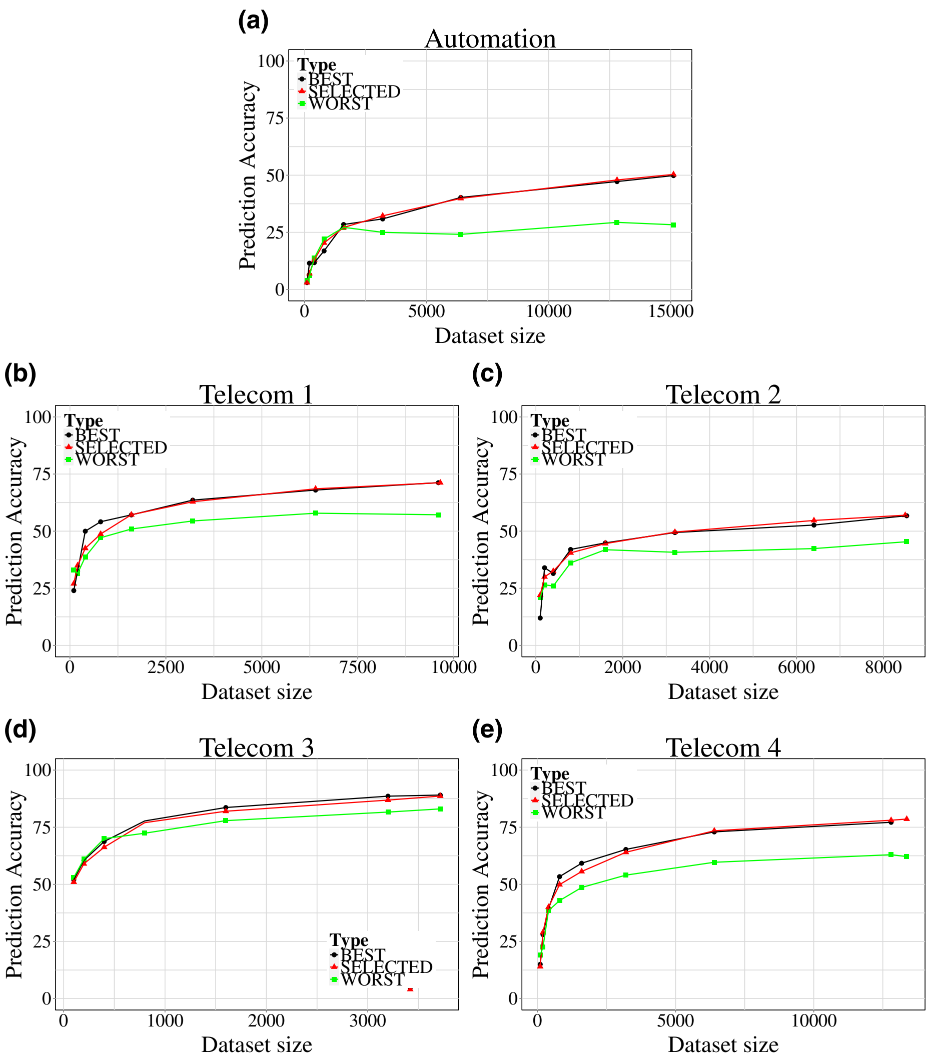


Fig. 9 Comparison of BEST (black, circle), SELECTED (red, triangle) and WORST (green, square) classifier ensemble

effects on Telecom 1 and Telecom 4, where the prediction accuracy is halved after roughly 500 days. For Telecom 1 the prediction accuracy is 50 % using the most recent training data, and it drops to about 25 % when the training data is 500 days old. The results for Telecom 4 are analogous, with the precision accuracy dropping from about 40 % to 15 % in 500 days.

For three of the datasets the decrease in prediction accuracy is less clear. For Automation, the prediction accuracy decreases from about 14 % to 7 % in 1,000 days, and Telecom 3, the smallest dataset, from 55 % to 30 % in 1,000 days. For Telecom 2 the decrease in prediction accuracy is even smaller, and thus unlikely to be of practical significance when deploying a solution in industry.

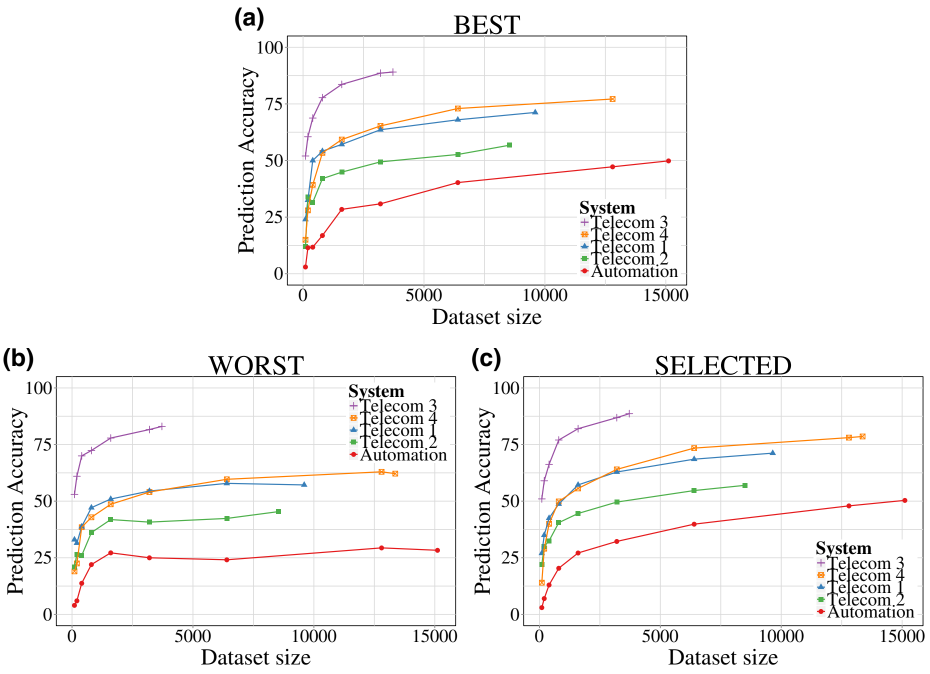


Fig. 10 Prediction accuracy for the different systems using the BEST (**a**) WORST (**b**) and SELECTED (**c**) individual classifiers under Stacking

A partial answer to RQ4 is: *more recent training data yields higher prediction accuracy when using SG for bug assignment.*

6.4 Experiment E: Cumulative Time Window

Experiment E addresses the same RQ as Experiment D, namely how the time locality of the training data affects the prediction accuracy (RQ4). However, instead of evaluating the effect using a fixed size sliding window of training examples, we use a cumulatively increasing training set. As such, Experiment E also evaluates how many training examples SG requires to perform accurate classifications. Experiment E shows the prediction accuracy that SG would have achieved at different points in time if deployed in the five projects under study.

Figure 12 plot the results from the cumulated time locality evaluation using SG with BEST ensembles. The blue curve represents the prediction accuracy (as fitted by a local regression spline) with the standard error for the mean of the prediction accuracy in the shaded region. The maximum prediction accuracy (as fitted by the regression spline) is indicated with a vertical line. The vertical line represents the cumulated ideal number of training points for the respective datasets. Adding more bug reports further back in time worsens the prediction accuracy. The number of samples (1589) and the prediction accuracy (16.41 %) for the maximum prediction accuracy is indicated with a text label (x = 1589 Y = 16.41 for the Automation system). The number of evaluations run with the calculated training set and test set sizes in each run is indicated in the upper right corner of the figure with the text "Total no. Evals".

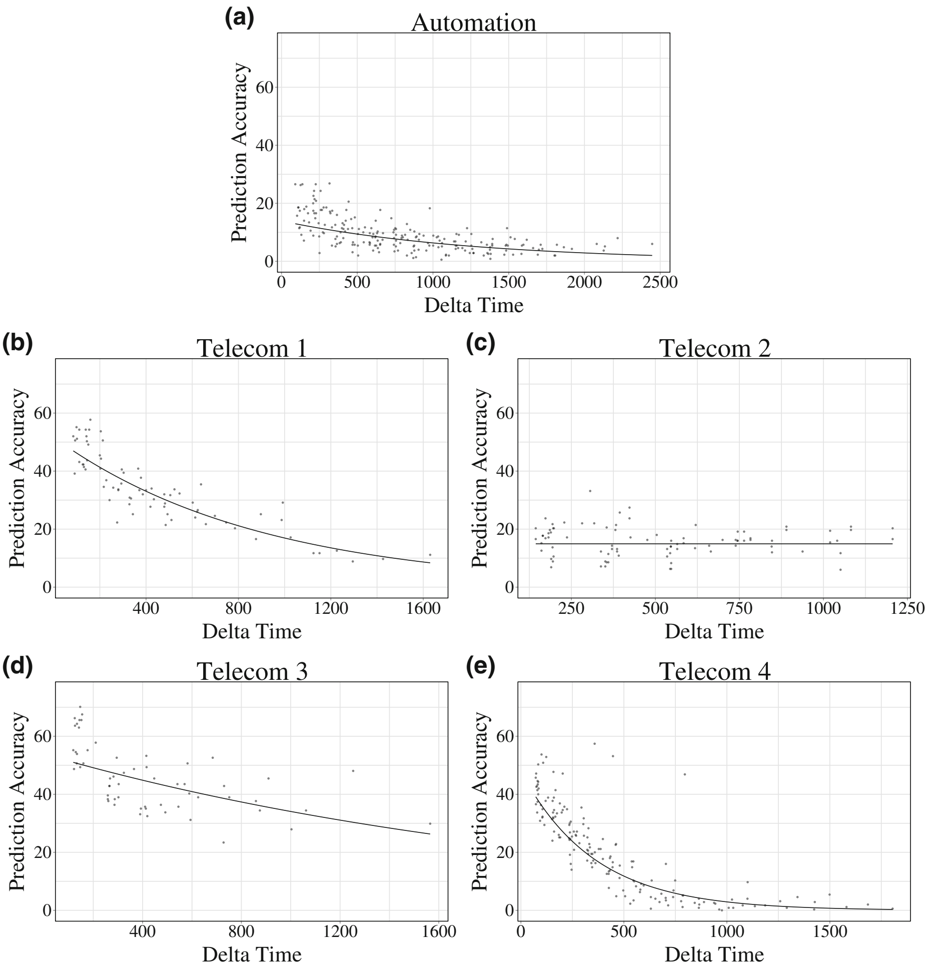


Fig. 11 Prediction accuracy for the datasets Automation (**a**) and Telecom 1-4 (**b-e**) using the BEST ensemble when the time locality of the training set is varied. Delta time is the difference in time, measured in days, between the start of the training set and the start of the test set. For Automation and Telecom 1,2, and 4 the training sets contain 1,400 examples, and the test set 350 examples. For Telecom 3, the training set contains 619 examples and the test set 154 examples

For all datasets in Fig. 12, except Telecom 3, the prediction accuracy increases when more training data is cumulatively added until a point where they reach a "hump" where the prediction accuracy reaches a maximum. This is followed by declining prediction accuracy as more (older) training data is cumulatively added. For Automation, Telecom 1, and Telecom 4, we achieve the maximum prediction accuracy when using about 1,600 training examples. For Telecom 2 the maximum appears already at 1,332 training examples. For Telecom 3 on the other hand, the curve is monotonically increasing, i.e., the prediction accuracy is consistently increasing as we add more training data. This is likely a special case for this dataset where we have not yet reached the point in the project where old data starts to introduce noise rather than helping the prediction.

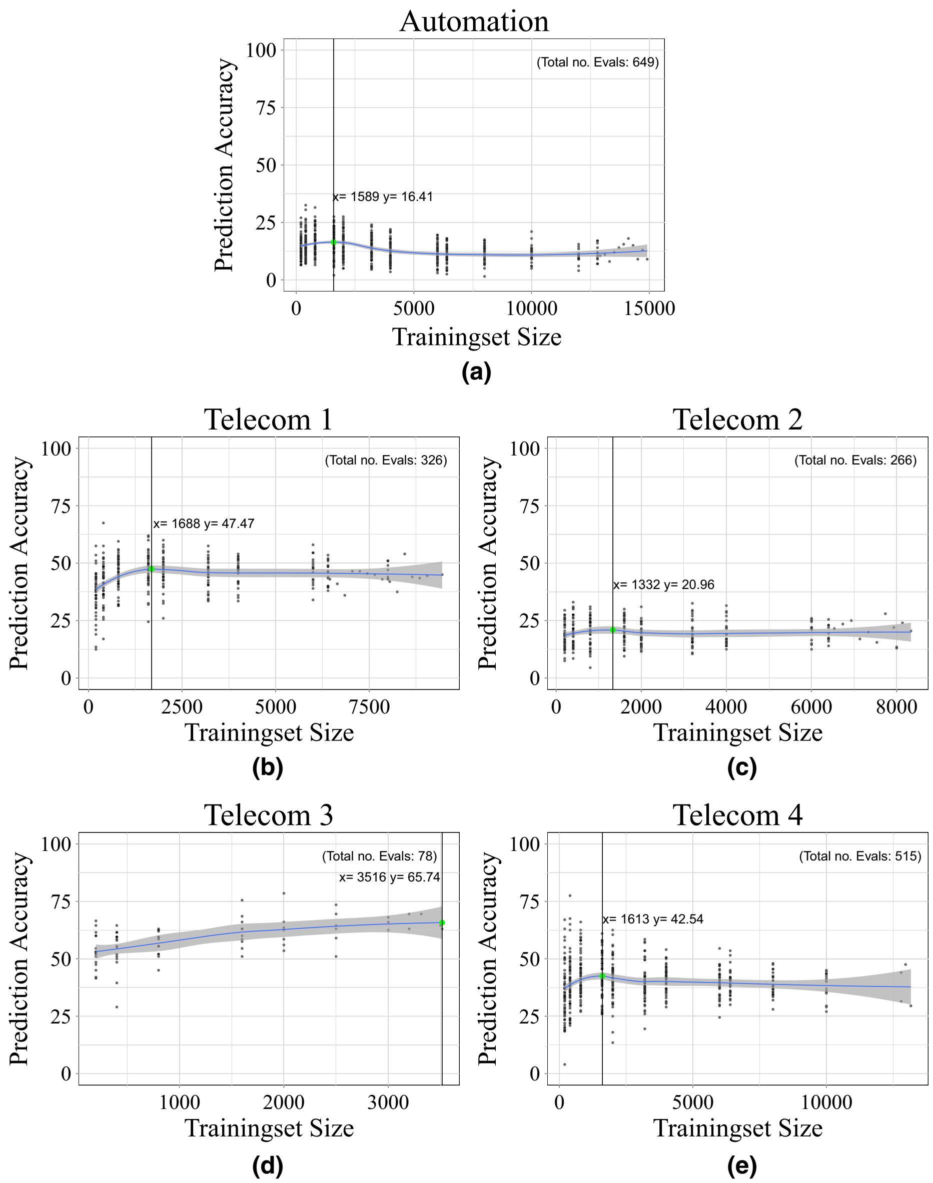


Fig. 12 Prediction accuracy using cumulatively (farther back in time) larger training sets. The *blue curve* represents the prediction accuracy (fitted by a local regression spline) with the standard error for the mean in the shaded region. The maximum prediction accuracy (as fitted by the regression spline) is indicated with a *vertical line*. The number of samples (1589) and the accuracy (16.41 %) for the maximum prediction accuracy is indicated with a text label (x = 1589 Y = 16.41 for the Automation system). The number of evaluations done is indicated in the upper right corner of the figure (Total no. Evals)

Also related to RQ4 is our observation: *there is a balancing effect between adding more training examples and using older training examples. As a consequence, prediction accuracy does not necessary improve when training sets gets larger.*

7 Threats to Validity

We designed our experiment to minimize the threats to validity, but still a number of decisions that might influence our results had to be made. We discuss the main validity threats to our study with respect to construct validity, internal validity, external validity, and conclusion validity (Wohlin et al. 2012).

7.1 Construct Validity

Construct validity involves whether our experiment measures the construct we study. Our aim is to measure how well automated bug assignment performs. We measure this using *prediction accuracy*, the basic measure for performance evaluations of classification experiments. As an alternative measure of the classification performance, we could have complemented the results with average F-measures to also illustrate type I and type II errors. However, to keep the presentation and interpretation of results simple, we decided to consistently restrict our results to contain only prediction accuracy.

The Weka framework does not allow evaluations with classes encountered in the test set that do not exist in the training set. For Experiments A-C (all based on cross-validation) Weka automatically *harmonizes* the two sets by ensuring the availability of all classes in both sets. However, for the time-sorted evaluations performed in Experiment D and E, we do not use the cross-validation infrastructure provided by Weka. Instead, we have to perform the harmonization manually. To simplify the experimental design and be conservative in our performance estimates in Experiment D and E, we always consider all teams present in the full dataset as possible classification results, i.e., regardless of whether the teams are present in the training and test sets of a specific run. This is a conservative design choice since it causes many more possible alternative classes available for classification, making it a harder problem. In practice, the team structure in an organization is dynamic. In the projects under study, teams are added, renamed, merged, and removed over the years as illustrated in Fig. 13. While the current team structure would be known at any given time in a real project, we do not use any such information in our experimental runs. Thus, there is a potential that the prediction accuracy of a deployed tool using SG could be higher.

In the projects we study, the teams are not entirely disjunct. Individual developers might be members of several teams, teams might be sub-teams of other teams, and certain teams are created for specific tasks during limited periods of time. Thus, as some teams overlap, more than one team assignment could be correct for a bug report. Furthermore, the correctness of a team assignment is not a binary decision in real life. Teams might be hierarchically structured and there are dependencies between teams. An incorrect bug assignment might thus be anything from totally wrong (e.g., assigning a bug report describing embedded memory management to a GUI team) to just as good as the team assignment stored in the BTS. Again, our evaluation is conservative as we consider everything not assigned to the same team as in the BTS as incorrect.

7.2 Internal Validity

Internal validity concern inferences regarding casual relationships. We aim to understand how SG performs compared to individual classifiers, and how its prediction accuracy is affected by different training configurations. Our experimental design addresses threats to internal validity by controlling the independent variables in turn. Still, there are a number of possibly confounding factors.

We conduct the same preprocessing for all classification runs. It is possible that some of the classifiers studied perform better for our specific choice of preprocessing actions than others. On the other hand, we conduct nothing but standard preprocessing (i.e., lower casing and standard stop word removal), likely to be conducted in most settings.

We use default configurations of all individual classifiers studied. While most classifiers are highly configurable, we do not perform any tuning. Instead, we consistently use the default configurations provided by Weka. The default configurations for some classifiers might be favorable for team assignment and others might underperform. Furthermore, we evaluated only one single level-1 classifier in SG, also using the default configuration. However, Wolpert (1992) argues that a simple level-1 classifier should be sufficient.

To facilitate future research, e.g., replications of our study and comparison to our results, we have restricted the features used by our classifiers to attributes that most BTSs will contain at *submission time*. Thus, we only consider general features that are set by the author of the bug report, without any extensive root cause analysis. However, our selection of features, based on our experience from industry, might have influenced the performance of the individual classifiers. Although this is a threat to the validity of our results, we argue that our choice is conservative; adding further informative features would potentially increase the performance, but should not decrease it. During industrial deployment, all features available in the specific BTS that could be of value should of course be used to train the system. Ideally high quality features should be collected by the production system and be automatically attached to the bug reports. We speculate that increasing the level of automation on both the submitting and receiving end of bug management, i.e., both for the author submitting bug reports to the BTS as well as the receiving analyst in the development organization, could create synergistic benefits for ML-based solutions. As a

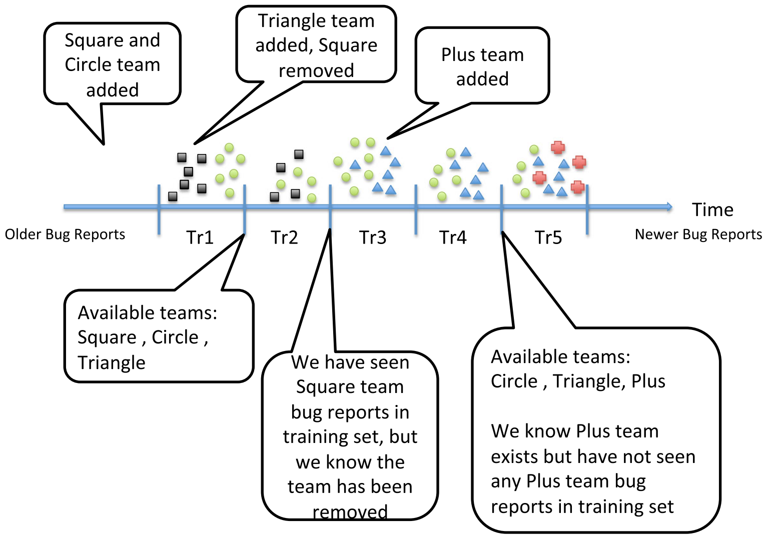

Square and Circle team added
Triangle team added, Square removed
Plus team added
Time
Older Bug Reports
Tr1
Tr2
Tr3
Tr4
Tr5
Newer Bug Reports
Available teams: Square , Circle , Triangle
We have seen Square team bug reports in training set, but we know the team has been removed
Available teams: Circle , Triangle, Plus
We know Plus team exists but have not seen any Plus team bug reports in training set

Fig. 13 This figure illustrates how teams are constantly added and removed during development. Team dynamics and BTS structure changes will require dynamic re-training of the prediction system. A prediction system must be adapted to keep these aspects in mind. These are all aspects external to pure ML techniques, but important for industry deployment

consequence we strongly encourage research in the area of feature selection for automatic bug assignment.

7.3 External Validity

External validity reflect the generalizability of our results. We study five large datasets containing thousands of bug reports from proprietary development projects. All datasets originate from development of different systems, including middle-ware, client-server solutions, a compiler, and communication protocols. However, while the datasets all are large, they originate from only two different companies. Furthermore, while the two companies work in different domains, i.e., automation and telecommunication, both are mainly concerned with development of embedded systems. To generalize to other domains such as application development, replications using other datasets are required.

The fraction of bug reports originating from customers is relatively low in all five datasets under study. In development contexts where end users submit more of the bug reports, different natural language descriptions and information content might be used. This might have an impact on the performance of SG for team assignment. However, as opposed to most previous work, we focus on proprietary development projects using closed BTSs.

We filtered the five datasets to contain only bug reports actually describing defects in the production software. It is possible that a BTS is used for other types of issues, as is the case in Company Automation, e.g., document changes, change requests, and changes to internal test code. We have not studied how well SG generalizes for these more generic types of issues. On the other hand, we assume that the most challenging team assignment involves defects in the production software where the location in source code and the related documentation are unknown.

Although our work focuses on large scale industrial settings, a relevant question is how well the approach generalizes to the OSS development context. Ideally, we would like to compare our results with similar OSS datasets. Unfortunately, we are not aware of any large dataset from OSS that uses team assignment. This makes a direct comparison very difficult. An alternative approach could be to use OSS datasets and created some type of after-the-fact team assignments. Applying such an approach would, however, be very hard to do in a fair manner.

7.4 Conclusion Validity

Conclusion validity is the degree to which conclusions we reach about relationships in our data are reasonable. For experiment A, B, and C we use 10-fold cross validation as conventional in machine learning evaluations. However, as argued by Rao et al. (2008), evaluations should also be performed using a *sequestered* test set. We accomplish this by performing experiment D and E on separate training and test sets. Moreover, we evaluate the performance in several runs as described in Sections 5.5.4 and 5.5.5.

The results from 10-fold cross-validation (using stratified sampling) and the evaluations conducted using defect reports submitted by submission date are different. Cross validation yields higher prediction accuracy, in line with warnings from previous research (Rao et al. 2008; Kodovsky 2011). To confirm the better results when using cross validation, we validated the results using RapidMiner (Hofmann and Klinkenberg 2013) for the two datasets Automation and Telecom 4. We trained a Naive Bayes classifier for Automation and an

SVM classifier for Telecom 4 and observed similar differences between evaluations using 10-fold cross validation and a sorted dataset.

8 Discussion

This section contains a discussion of the results from our experiments in the context of our overall goal: to support bug assignment in large proprietary development projects using state-of-the-art ML. Section 8.1 synthesizes the results related to our RQs and discusses the outcome in relation to previous work. Finally, Section 8.2 reports important experiences from running our experiments and advice on industrial adaptation.

8.1 Stacked Generalization in the Light of Previous Work

We conduct a large evaluation of using SG for bug assignment, extending our previous work (Jonsson et al. 2012). Our results show that *SG* (i.e., combining several classifiers in an ensemble learner) *can yield a higher prediction accuracy than using individual general purpose classifiers* (RQ1, ExpA). The results are in line with findings in general ML research (Kuncheva and Whitaker 2003). However, we show that simply relying on SG is not enough to ensure good results; some *care must be taken when doing the ensemble selection* (RQ2, ExpB). On the other hand, our results confirm the thesis by Wolpert (1992) that SG should on average perform better than the individual classifiers included in the ensemble.

We present the first study on bug assignment containing 10,000 s of bug reports collected from different proprietary development projects. Previous work has instead focused on bug reports from OSS development projects, e.g., Eclipse and Firefox, as presented in Section 3. A fundamental difference is that while bug assignment in OSS projects typically deal with individual developers, we instead assign bug reports to development teams. As this results in a more coarse assignment granularity, i.e., our output is a set of developers, one could argue that we target a less challenging problem.

We achieve prediction accuracy between 50 % and 85 % for our five systems using cross-validations, and between 15 % and 65 % for time-sorted evaluations. Thus, our work on bug team assignment does not display higher prediction accuracy than previous work on automated bug assignment to individuals, but is similar to what has been summarized in Fig. 3. Consequently, we show that *automated proprietary bug assignment, on a team level, can correctly classify the same fraction of bug reports as what has been reported for bug assignment to individual developers in OSS projects*. Bug assignment to teams does not appear to be easier than individual assignment, at least not when considering only the top candidate team presented by the ML system.

Cross-validation consistently yielded higher prediction accuracy than conducting more realistic evaluations on bug reports sorted by the submission date. The dangers of cross-validation have been highlighted in ML evaluation before (Rao et al. 2008), and it is a good practice to complement cross-validation with a sequestered test set. Our experiments show that evaluations on bug assignment can not rely on cross-validation alone. Several factors can cause the lower prediction accuracy for the time sorted evaluations. First, cross-validation assumes that the bug reports are independent with no distributional differences between the training and test sets (Arlot and Celisse 2010). Bug reports have a natural temporal ordering, and our results suggest that the dependence among individual bug reports can not be neglected. Second, we used stratified sampling in the cross-validation, but not in the time sorted evaluations. Stratification means that the team distributions in the training sets and test sets are the same, which could improve the results in cross-validation. Third, as we perform manual harmonization of the classes in the time sorted evaluation (see Section 7), all teams are always possible classifications. In cross-validation, Weka performs the harmonization just for the teams involved in the specific run, resulting in fewer available teams and possibly a higher prediction accuracy.

8.2 Lessons Learned and Industrial Adoption

Two findings from our study will have practical impact on the deployment of our approach in industrial practice. First, we studied how large the training set needs to be for SG to reach its potential. The learning curves from 10-fold cross-validation show that larger training set are consistently better, but the improvement rate decreases after about 2,000 training examples. The point with the maximum curvature, similar to an elbow point (Tibshirani et al. 2001) as used in cluster analysis, appears in the same region for all five systems. As a result, we suggest, as a rule of thumb, that *at least 2,000 training examples should be used when using SG for automated bug assignment* (RQ3, ExpC).

The second important finding of practical significance relates to how often an ML system for bug assignment needs to be retrained. For all but one dataset, our results show a clear decay of prediction accuracy as we use older training data. For two datasets the decay appears exponential, and for two datasets the decay is linear. Our conclusion is that the time locality of the training data is important to get a high prediction accuracy, i.e., *SG for bug assignment is likely to achieve a higher prediction accuracy if trained on recent bug reports* (RQ4, ExpD). Bhattacharya et al. (2012) recently made the same observation for automated bug assignment using large datasets from the development of Eclipse and Mozilla projects.

Finding the right time to retrain SG appears to be a challenge, as we want to find the best balance between using many training examples and restricting the training set to consist of recent data. In Experiment E, our last experiment, we try several different cumulatively increasing training sets at multiple points in time. This experimental setup mimics realistic use of SG for bug assignment, trained on different amounts of previous bug reports. We show that for four of our datasets, *the positive effect of using larger training sets is nullified by the negative effect of adding old training examples*. Only for one dataset it appears meaningful to keep as much old data as possible.

When deployed in industrial practice, we recommend that *the prediction accuracy of automated bug assignment should be continuously monitored to identify when it starts to deteriorate*. For four of our datasets, cumulatively increasing the amount of training data is beneficial at first (see Fig. 12), but then SG reaches a maximum prediction accuracy. For all but one dataset, the prediction accuracy starts to decay even before reaching the 2,000 training examples recommended based on the results from Experiment C. Furthermore, we stress that attention should be paid to alterations of the prediction accuracy when significant changes to either the development process or the actual software product are made. Changes to the team structure and the BTS clearly indicate that SG should be retrained, but also process changes, new tools in the organization, and changes to the product can have an impact on the attributes used for automated bug assignment. In practice, the monitoring of the prediction accuracy could be accomplished by measuring the amount of bug tossing taking place after the automated bug assignment has taken place. To facilitate careful monitoring of the bug tossing we suggest that BTSs should store the full bug assignment history, not only where the bug was finally solved. Unfortunately the full bug tossing history was not available for all BTSs in our study, thus we were unable to calculate a detailed estimate of the possible savings of our approach. Future work should introduce new bug tossing measurements to enable estimation of possible time savings from deploying our approach.

While we can measure the prediction accuracy of SG for bug assignment, it is not clear what this means for practical purposes. How accurate do the classifications have to be before developers start recognizing its value? Regnell et al. (2008) describe quality, in our case the prediction accuracy of automated bug assignment, as continuous and non-linear. Figure 14 shows what this means for bug assignment. *The perceived usefulness of SG is on a sliding scale with specific breakpoints*. The utility breakpoint represents when developers start considering automated bug assignment useful, any prediction accuracy below this level is useless. The saturation breakpoint indicates where increased prediction accuracy has no practical significance to developers. Figure 14 also displays the prediction accuracy of human analysts between the breakpoints. We argue that automated bug assignment does not have to reach the human accuracy to be perceived useful, as the automated process is much faster than the manual process. Our early evaluations indicate that the prediction accuracy of SG in Company Telecom is in line with the manual process (Jonsson et al. 2012). Even though this study also used 10-fold cross-validation, which we have seen can give overly optimistic estimates in this context; we believe that our prototype has passed the utility breakpoint before we have started any context specific tuning of SG.

When we implement automated bug assignment in an industrial tool, we plan to present a handful of candidate teams to the user for each bug report under investigation. While we could automatically assign the bug to the first candidate, our first step is to provide decision support to the CCB. Considering the automation levels defined by Parasuraman et al. (2000), this reflects an increase in automation from level 0 to level 3: "narrowing the selection down to a few". By presenting a limited selection of teams, possibly together with a measure of the confidence level from SG, an experienced developer can quickly choose the best target for the bug assignment. Note that the experimental design we used in the evaluations in this study are stricter as we only considered one single team assignment per bug report. Another recommendation is to plan from the start to run the new ML-based system in parallel with

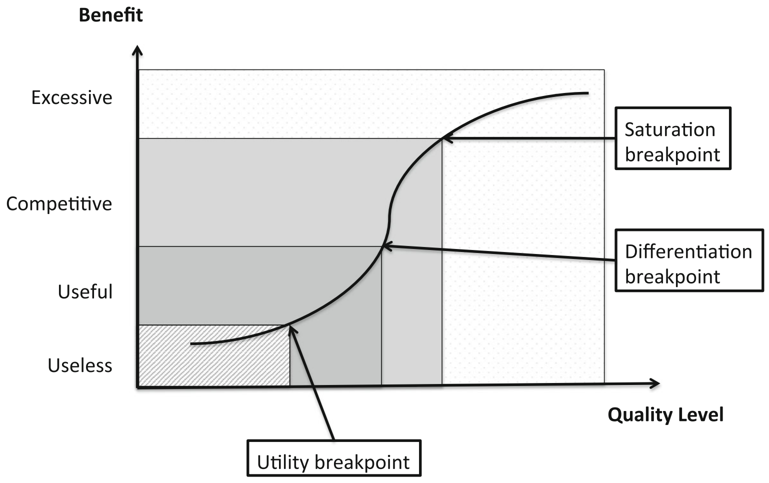


Fig. 14 Perceived benefit vs. prediction accuracy. The figure shows two breakpoints and the current prediction accuracy of human analysts. Adapted from Regnell et al. (2008)

the old way of working, to evaluate if the prediction accuracy is good enough for a complete roll over to a process supported by ML.

Furthermore, we must develop the tool to present the candidate teams to the user in a suitable manner. Murphy-Hill and Murphy (2014) presents several factors that affects how users perceive recommendations provided by software engineering tools. The two most important factors for our tool are transparency and the related aspect assessability. A developer must be able to see why our tool suggests assigning a bug report to a specific team, i.e., *the rationale leading to the SG classification must be transparent*. Also, our tool should *support developers in assessing the correctness of a suggested assignment*. We aim to achieve this by enabling interaction with the output, e.g., browsing previous bug assignments, opening detailed bug information, and comparing bug reports.

9 Conclusions and Future Work

We conduct the first evaluation of automated bug assignment using large amounts of bug reports, collected from proprietary software development projects. Using an ensemble learner, Stacked Generalization (SG), we train an ML system on historical bug reports from five different projects in two different, large companies. We show that *SG consistently outperforms individual classifiers with regard to prediction accuracy* even though the improvements are sometimes marginal (RQ1). Moreover, our results suggest that it is *worthwhile to strive for a diverse set of individual classifiers in the ensemble* (RQ2), consistent with recommendations in the general ML research field. Our results show that SG, with feasible ensemble selection, can reach prediction accuracies from 50 % to 89 % for the different systems, in line with the prediction accuracy of the current manual process. We also briefly study the relative value of textual vs. non-textual features, and conclude that the most promising results are obtained when combining both in SG. In future work we plan to improve the textual features with more advanced text modeling techniques such as Topic Modeling (LDA).

We study the SG learning curves for the five systems (RQ3), using 10-fold cross-validation. The learning curves for all five datasets studied display similar behaviour, thus we present an empirically based rule of thumb: *when training SG for automated bug assignment, aim for at least 2,000 bug reports in the training set*. Using time-sorted bug reports in the training and test sets we show that *the prediction accuracy decays as older training data is used* (RQ4). Consequently, we show that *the benefit of adding more bug reports in the training set is nullified by the disadvantage of training the system on less recent data*. Our conclusion is that *any ML system used for automated bug assignment should be continuously monitored to detect decreases in prediction accuracy*.

Our results confirm previous claims that *relying only on K-fold cross-validation is not enough to evaluate automated bug assignment*. We achieve higher prediction accuracy when performing 10-fold cross-validation with stratification than when analyzing bug reports sorted by the submission date. The differences we observe are likely to be of practical significance, thus it is important to report evaluations also using sorted data, i.e., mimicking a realistic inflow of bug reports. Several authors have proposed modifications to cross-validation to allow evaluations on dependent data, e.g., *h*-block cross-validation (Burman et al. 1994). Future work could try this for bug assignment evaluation, which means reducing the training set by removing *h* observations preceding and following the observations in the test set.

When deploying automated bug assignment in industry, we plan to present more than one candidate development team to the user of the ML system. By presenting a ranked list of teams, along with rationales of our suggestions, an experienced member of the CCB should be able to use the tool as decision support to select the most appropriate team assignment. Our current evaluation does not take this into account, as we only measure the correctness of the very first candidate. Future work could extend this evaluation by evaluating lists of candidates, opening up for measures from the information retrieval field, e.g., mean average precision and normalized discounted cumulative gain. Finally, to properly evaluate how ML can support bug assignment in industry, the research community needs to conduct industrial case studies in organizations using the approach. In particular, it is not clear how high the prediction accuracy needs to be before organizations perceive the system to be "good enough".

Future work could be directed toward improving our approach to automated bug assignment. A number of studies in the past show that tools specialized for bug assignment in a particular project can outperform general purpose classifiers (Tamrawi et al. 2011; Xie et al. 2012; Xia et al. 2013). It would be possible for us to explore if any enhancements proposed in previous work could improve the accuracy of SG, e.g., topic models, social network analysis, or mining the commit history of source code repositories. Also, we could further investigate if any particular characteristics of team assignment in proprietary projects could be used to improve automated bug assignment, i.e., characteristics that do not apply to OSS projects.

Another direction for future enhancements of our approach could explore how to adapt bug assignment based on the developers' current work load in the organization. The current solution simply aims to assign a bug report to development teams that worked on similar bug reports in the past. Another option would to optimize the resolution times of bug reports by assigning bugs to the team most likely to close them fast. For many bug reports, more than one team is able to resolve the issue involved, especially in organizations with a dedicated strategy for shared code ownership. Future work could explore the feature engineering required for SG to cover this aspect. Yet another possible path for future work, made possible by the large amount of industrial data we have collected, would be to conduct comparative studies of bug reports from OSS and proprietary projects, similar to what Robinson and Francis (2010) reported for source code.

Acknowledgements This work was supported in part by the Industrial Excellence Center EASE – Embedded Applications Software Engineering.[9]

References

Aberdour M (2007) Achieving quality in open-source software. IEEE Softw 24(1):58–64

Ahsan S, Ferzund J, Wotawa F (2009) Automatic software bug triage system (bts) based on latent semantic indexing and support vector machine. In: Proceedings of the 4th international conference on software engineering advances, pp 216–221

Alenezi M, Magel K, Banitaan S (2013) Efficient bug triaging using text mining. J Softw 8(9)

Alshammari R, Zincir-Heywood A (2009) Machine learning based encrypted traffic classification: Identifying SSH and Skype. In: Proceedings of the symposium on computational intelligence for security and defense applications, pp 1–8

[9]http://ease.cs.lth.se.

Amamra A, Talhi C, Robert JM, Hamiche M (2012) Enhancing smartphone malware detection performance by applying machine learning hybrid classifiers. In: Kim Th, Ramos C, Kim Hk, Kiumi A, Mohammed S, Slezak D (eds) Computer applications for software engineering, disaster recovery, and business continuity, no. 340 in communications in computer and information science. Springer, Berlin, pp 131–137

Anvik J (2007) Assisting bug report triage through recommendation. Thesis, University of British Columbia

Anvik J, Murphy GC (2011) Reducing the effort of bug report triage: recommenders for development-oriented decisions. Trans Softw Eng Methodol 20(3):10:1–10:35

Anvik J, Hiew L, Murphy GC (2006) Who should fix this bug? In: Proceedings of the 28th international conference on software engineering, New York, NY, USA, '06, pp 361–370

Arlot S, Celisse A (2010) A survey of cross-validation procedures for model selection. Statistics Surveys 4:40–79

Asklund U, Bendix L (2002) A study of configuration management in open source software projects. IEE Proceedings - Software 149(1):40–46

Avazpour I, Pitakrat T, Grunske L, Grundy J (2014) Dimensions and metrics for evaluating recommendation systems. In: Robillard M, Maalej W, Walker R, Zimmermann T (eds) Recommendation systems in software engineering. Springer, pp 245–273

Basili V, Selby R, Hutchens D (1986) Experimentation in software engineering. IEEE Trans Softw Eng SE 12(7):733–743. doi:10.1109/TSE.1986.6312975

Baysal O, Godfrey M, Cohen R (2009) A bug you like: A framework for automated assignment of bugs. In: Proceedings of the 17th international conference on program comprehension, pp 297–298

Bettenburg N, Premraj R, Zimmermann T, Sunghun K (2008) Duplicate bug reports considered harmful... really? In: Proceedings of the international conference on software maintenance, pp 337–345

Bezanson J, Karpinski S, Shah VB, Edelman A (2012) Julia: A fast dynamic language for technical computing. arXiv:1209.5145

Bhattacharya P, Neamtiu I, Shelton CR (2012) Automated, highly-accurate, bug assignment using machine learning and tossing graphs. J Syst Softw 85(10):2275–2292

Bifet A, Holmes G, Kirkby R, Pfahringer B, Massive online analysis (2010) J Mach Learn Res 11:1601–1604

Bishop CM (2006) Pattern recognition and machine learning. Springer, New York

Blei D, Ng A, Jordan M (2003) Latent dirichlet allocation. J Mach Learn Res 3:993–1022. http://dl.acm.org/citation.cfm?id=944919.944937

Borg M, Pfahl D (2011) Do better IR tools improve the accuracy of engineers' traceability recovery? In: Proceedings of the international workshop on machine learning technologies in software engineering, pp 27–34

Borg M, Runeson P, Ardö A (2014) Recovering from a decade: A systematic mapping of information retrieval approaches to software traceability. Empir Softw Eng 19(6):1565–1616. doi:10.1007/s10664-013-9255-y

Breiman L (1996) Bagging predictors. Mach Learn 24(2):123–140

Burman P, Chow E, Nolan D (1994) A cross-validatory method for dependent data. Biometrika 81(2):351–358

Canfora G, Cerulo L (2006) Supporting change request assignment in open source development. In: Proceedings of the symposium on applied computing, pp 1767–1772

Chen L, Wang X, Liu C (2011) An approach to improving bug assignment with bug tossing graphs and bug similarities. J Softw 6(3)

Cubranic D, Murphy GC (2004) Automatic bug triage using text categorization. In: Proceedings of the 16th international conference on software engineering & knowledge engineering, pp 92–97

Frank E, Hall M, Trigg L, Holmes G, Witten I (2004) Data mining in bioinformatics using Weka. Bioinformatics 20(15):2479–2481

Freund Y, Schapire RE (1995) A desicion-theoretic generalization of on-line learning and an application to boosting. In: Vitanyi P (ed) Computational learning theory, no. 904 in lecture notes in computer science. Springer, Berlin, pp 23–37

Green SB (1991) How many subjects does it take to do a regression analysis. Multivar Behav Res 26(3):499–510

Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH (2009) The WEKA data mining software: an update. SIGKDD Explor Newsl 11(1):10–18

Helming J, Arndt H, Hodaie Z, Koegel M, Narayan N (2011) Automatic sssignment of work items. In: Maciaszek LA, Loucopoulos P (eds) Proceedings of the international conference on evaluation of novel approaches to software engineering. Springer, Berlin, pp 236–250

Hofmann M, Klinkenberg R (2013) Data mining use cases and business analytics applications. CRC Press, Taylor & Francis Group, Boca Raton. ISBN: 1482205491, 9781482205497

Jeong G, Kim S, Zimmermann T (2009) Improving bug triage with bug tossing graphs. In: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, New York, NY, USA, pp 111–120

Jonsson L, Broman D, Sandahl K, Eldh S (2012) Towards automated anomaly report assignment in large complex systems using stacked generalization. In: Proceedings of the International conference on software testing, verification, and validation, pp 437–446

Just S, Premraj R, Zimmermann T (2008) Towards the next generation of bug tracking systems. In: Proceedings of the Symposium on visual languages and Human-centric computing, IEEE Computer Society, pp 82–85

Kagdi H, Gethers M, Poshyvanyk D, Hammad M (2012) Assigning change requests to software developers. J Softw: Evolution and Process 24(1):3–33

Kodovsky J (2011) On dangers of cross-validation in steganalysis. Tech. rep., Birmingham University

Kohavi R (1995) A study of cross-validation and bootstrap for accuracy estimation and model selection. In: Proceedings of the 14th International joint conference on artificial intelligence, vol 2, pp 1137–1143

Kuncheva LI, Whitaker CJ (2003) Measures of diversity in classifier ensembles and their relationship with the ensemble accuracy. Mach Learn 51(2):181–207

Li N, Li Z, Nie Y, Sun X, Li X (2011) Predicting software black-box defects using stacked generalization. In: Proceedings of the International conference on digital information management, pp 294–299

Li Q, Wang Q, Yang Y, Li M (2008) Reducing biases in individual software effort estimations: a combining approach. In: Proceedings of the 2nd international symposium on empirical software engineering and measurement, pp 223–232. doi:10.1145/1414004.1414041

Lin Z, Shu F, Yang Y, Hu C, Wang Q (2009) An empirical study on bug assignment automation using Chinese bug data. In: Proceedings of the 3rd international symposium on empirical software engineering and measurement, pp 451–455

Linares-Vasquez M, Hossen K, Dang H, Kagdi H, Gethers M, Poshyvanyk D (2012) Triaging incoming change requests: bug or commit history, or code authorship? In: Proceedings of the 28th international conference on software maintenance, pp 451–460

Matter D, Kuhn A, Nierstrasz O (2009) Assigning bug reports using a vocabulary-based expertise model of developers. In: 6th IEEE International working conference on mining software repositories, 2009. MSR '09, pp 131–140. doi:10.1109/MSR.2009.5069491

McCallum A (2002) A machine learning for language toolkit. Tech. rep. http://mallet.cs.umass.edu

Mozilla (2013) Life cycle of a bug. http://www.bugzilla.org/docs/tip/en/html/lifecycle.html. Accessed 28-October-2013

Murphy-Hill E, Murphy G (2014) Recommendation delivery: getting the user interface just right. In: Robillard M, Maalej W, Walker R, Zimmermann T (eds) Recommendation systems in software engineering. Springer, Berlin

Nagwani N, Verma S (2012) Predicting expert developers for newly reported bugs using frequent terms similarities of bug attributes. In: Proceedings of the 9th international conference on ICT and knowledge engineering, pp 113–117

Owen S, Anil R, Dunning T, Friedman E (2011) Mahout in action. Manning Publications, Shelter Island

Parasuraman R, Sheridan T, Wickens C (2000) A model for types and levels of human interaction with automation. IEEE Trans Syst Man Cybern 30(3):286–297

Park J, Lee M, Kim J, Hwang S, Kim S (2011) A cost-aware triage algorithm for bug reporting systems. In: Proceedings of the 25th AAAI conference on artificial intelligence

Paulson J, Succi G, Eberlein A (2004) An empirical study of open-source and closed-source software products. IEEE Trans Softw Eng 30(4):246–256

Petersen K, Wohlin C (2009) Context in industrial software engineering research. In: Proceedings of the 3rd international symposium on empirical software engineering and measurement, pp 401–404

Rao R, Fung G, Rosales R (2008) On the dangers of cross-validation. An experimental evaluation. In: Proceedings of the SIAM international conference on data mining, pp 588–596

Regnell B, Berntsson Svensson R, Olsson T (2008) Supporting roadmapping of quality requirements. IEEE Softw 25(2):42–47. doi:10.1109/MS.2008.48

Robillard M, Maalej W, Walker R, Zimmermann T (2014) Recommendation systems in software engineering. Springer, Berlin

Robinson B, Francis P (2010) Improving industrial adoption of software engineering research: A comparison of open and closed source software. In: Proceedings of the international symposium on empirical software engineering and measurement, pp 21:1–21:10

Robles G, Gonzalez-Barahona J (2006) Contributor turnover in Libre software projects. In: Damiani E, Fitzgerald B, Scacchi W, Scotto M, Succi G (eds) Open source systems, no. 203 in International federation for information processing. Springer, pp 273–286

Servant F, Jones J (2012) Automatic developer-to-fault assignment through fault localization. In: Proceedings. of the 34th international conference on software engineering (ICSE), pp 36–46

Shokripour R, Kasirun Z, Zamani S, Anvik J (2012) Automatic bug assignment using information extraction methods. In: Proceedings of the international conference on advanced computer science applications and technologies, pp 144–149

Sill J, Takacs G, Mackey L, Lin D (2009) Feature-weighted linear stacking. arXiv:0911.0460

Tamrawi A, Nguyen T, Al-Kofahi J, Nguyen T (2011) Fuzzy set and cache-based approach for bug triaging. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering, pp 365–375. doi:10.1145/2025113.2025163

Thomas S, Nagappan M, Blostein D, Hassan A (2013) The impact of classifier configuration and classifier combination on bug localization. IEEE Trans. Softw. Eng. 39(10):1427–1443

Tibshirani R, Walther G, Hastie T (2001) Estimating the number of clusters in a data set via the gap statistic. J R Stat Soc Ser B (Stat Methodol) 63(2):411–423

Wiklund K, Eldh S, Sundmark D, Lundqvist K (2013) Can we do useful industrial software engineering research in the shadow of lean and agile? In: Proceedings of the 1st international workshop on conducting empirical studies in industry, pp 67–68

Witten IH, Frank E, Hall MA (2011) Data mining. pub. Burlington, MA

Wohlin C, Runeson P, Host M, Ohlsson M, Regnell B, Wesslen A (2012) Experimentation in software engineering: A practical guide. Springer, Berlin

Wolpert D (1992) Stacked generalization. Neural Netw. 5(2):241–259

Wu W, Zhang W, Yang Y, Wang Q (2011) Developer recommendation with k-nearest-neighbor search and expertise ranking. In: Proceedings of the 18th Asia pacific software engineering conference, pp 389–396

Xia X, Lo D, Wang X, Zhou B (2013) Accurate developer recommendation for bug resolution. In: Proceedings of the 20th working conference on reverse engineering, pp 72–81

Xie X, Zhang W, Yang Y, Wang Q (2012) Developer recommendation based on topic models for bug resolution. In: Proceedings of the 8th international conference on predictive models in software engineering, pp 19–28

Zaharia M, Chowdhury NMM, Franklin M, Shenker S, Stoica I (2010) Spark: cluster computing with working sets. Tech. rep., EECS department, University of California, University of California at Berkeley, Berkeley, California

Zhao Y, Zhang Y (2008) Comparison of decision tree methods for finding active objects. Adv Space Res 41(12):1955–1959

Leif Jonsson received his MSc degree in Computer Science (1998) from Uppsala University; in the same year he started working at Ericsson AB's research division. In 2010 he started his PhD studies at Linköping University. His research interests include applying machine learning techniques to large-scale software development processes to automate traditionally hard to automate tasks. His current focus area is automatic fault localization.

Markus Borg received a MSc degree in Computer Science and Engineering (2007) and a PhD degree in Software Engineering (2015) from Lund University, where he is a member of the Software Engineering Research Group (SERG). His research interests are related to alleviating information overload in large-scale software development, with a focus on increasing the level of automation in the inflow of issue reports. Prior to his PhD studies, he worked three years as a development engineer at ABB in safety-critical software engineering. He is a student member of the IEEE.

David Broman is Associate Professor at KTH Royal Institute of Technology, Sweden. He also has a part time research position at the University of California, Berkeley, USA. David received his Ph.D. in Computer Science in 2010 from Linköping University, Sweden. He has been an Assistant Professor at Linköping University and a Visiting Scholar at the University of California, Berkeley. His research is focused on time-aware systems design, in particular programming and modeling language theory, and real-time systems. He has worked within the software security industry, co-founded the EOOLT workshop series, and is member of IFIP WG 2.4 and the TAACCS steering committee.

Kristian Sandahl is professor of Software Engineering at the Department of Computer and Information Science at Linköping University, Sweden. He is the leader of the Programming Environments Laboratory and his major research interests are: Requirements Engineering, Software Processes, Model-Based Software Development, and Empirical Research Methods. Professor Sandahl is a true enthusiast of transferring of knowledge between industry and academia and has worked both in spin-off companies and at Ericsson. He is an active member of IEEE and Swedish Association of Graduate Engineers.

Sigrid Eldh is a Senior Specialist and Researcher in Test & Debug Technology at Ericsson, Sweden with over 30 years of experience from the software industry. She holds a part time position at Mälardalens University, Sweden, where she also received her Ph.D. in Computer Science. She received her Masters of Computer Science from Uppsala University, Sweden. Her research focus is on test automation, test design and architecture using more automatic approaches for e.g. bug prevention and removal on large complex many-core systems. This has lead research towards self-healing systems, including other self*properties. Sigrid Eldh was a co-founder of test organizations such as ISTQB, SSTB and SAST.

Dr. Per Runeson is a professor of software engineering at Lund University, Sweden, head of the Department of Computer Science, and the leader of its Software Engineering Research Group (SERG) and the Industrial Excellence Center on Embedded Applications Software Engineering (EASE). His research interests include empirical research on software development and management methods, in particular for verification and validation. He is the principal author of "Case study research in software engineering", has coauthored "Experimentation in software engineering", serves on the editorial board of Empirical Software Engineering and Software Testing, Verification and Reliability, and is a member of several program committees.