Predicting the Lifetime of Pull Requests

Manoel Limeira de Lima Júnior[1], Daricélio Moreira Soares[1], Alexandre Plastino[2] and Leonardo Murta[2]

[1] *Centro de Ciências Exatas e Tecnológicas, Universidade Federal do Acre, Rio Branco – AC, Brazil*
Email: {limeira, daricelio}@ufac.br
[2] *Instituto de Computação, Universidade Federal Fluminense, Niterói – RJ, Brazil*
Email: {plastino, leomurta}@ic.uff.br

Abstract

Context: Having an estimate of the lifetime of pull requests may be useful for code integrators and project managers when performing tasks such as triaging, developers allocation, and release planning. Some works in the literature have explored pull request lifetime prediction by making use of regression techniques.
Objective: The first objective of our work was to reduce the average error rate of the prediction obtained by the regression techniques so far. The second objective was to obtain more effective and useful predictive models applying classification techniques. Thus, we switched to estimating discrete values, which represent time intervals (minutes, hours, days, weeks and months).
Method: We performed experiments with different regression techniques and achieved a significant decrease in the mean error rate. However, in cases where it is desired to know the magnitude of pull request lifetime, a prediction of the exact time with an error rate around thousands of minutes may not be appropriate. Thus, we decided to explore the adoption of classification techniques and discretized the lifetime into five time intervals. We also proposed new predictive attributes for the estimation of these time intervals and employed attribute selection strategies to identify subsets of attributes that could improve the predictive behaviour of the classifiers.
Results: Our classification approach achieved the best accuracy in 18 out of the 20 projects evaluated. The average accuracy was of 45.28% to predict pull request lifetime, with an average normalized improvement of 14.68% in relation to the majority class.
Conclusion: Our approach, which is represented by the Random Forest classifier applied over a subset of attributes identified by the attribute selection strategy based on the Information Gain metric, achieved superior results in comparison to other related approaches.

Keywords: pull request lifetime, lifetime prediction, pull-based software development, distributed software development

1. Introduction

The number of pull requests submitted by both external and internal developers (*i.e.*, requesters) in open-source projects which employ a pull-based software development methodology can be extremely high [1]. In this scenario, the core development team (*i.e.*, integrators) must divide its attention between a large number of pull requests and the remainder of the code development and maintenance tasks. As a result, the time interval between pull request submission and acceptance may increase in many cases. This interval, which we call pull request lifetime, involves the wait for the feedback of an integrator, the process of pull request review and discussion, and the actual integration of the submitted code.

A survey carried out with 749 integrators from projects hosted by GitHub revealed that the quality of the code to be integrated into the projects and the time available for the this task are two of the main concerns in projects that make use of the pull–based development methodology [2]. When a pull request exhibits a long lifetime, the project is kept from the benefits of a new functionality or a bug fix for that period of time. Furthermore, the delay in code integration might discourage the requester, who seeks to have his or her contribution integrated into the project.

In this context, having estimates for pull request lifetimes may be useful for integrators and project managers alike in order for them to triage pull requests, allocate code reviewers, and plan software releases. Moreover, pull request lifetime prediction enables transparency of the review process for the requester, providing a preview of the status regarding possible delays in the integration of his or her contributions. Lastly, end users might also benefit from such estimate, since they would have an indication of when a certain functionality or bug fix would be added to the product. However, predicting pull request lifetime is a complex task due to the number of variables that may influence the prediction.

Some recent works in the literature have explored pull request lifetime prediction scenarios. Gousios *et al.* [1] evaluated classification algorithms over open-source project data in order to predict pull request lifetime in three classes: $C_1 \leq 1\ \mathsf{hour} < C_2 \leq 1\ \mathsf{day} < C_3$. Yu *et al.* [3] employed the multiple linear regression technique to model the latency of pull requests from open-source projects. They showed that the predictive attributes used in their approach were able to achieve better results in the modeling of pull request latency than those used by Gousios *et al.* [1].

Focusing on improving on the results obtained by Yu *et al.* [3], we initially performed experiments with different regression techniques and achieved a significant decrease in the mean error rate. For instance, in project rosdistro, the mean error rate went down from 4,843 to 2,215 minutes using the SMOReg algorithm, and in project vagrant the mean error went down from 513,906 to 73,786 using the Random Forest algorithm.

However, in cases where it is desired to know the magnitude of pull request lifetime (*i.e.*, minutes, hours, days, weeks, or months), a prediction of the exact time with an error rate around the thousands of minutes may not be appropri-

ate. Thus, we decided to explore the adoption of classification techniques and discretized the lifetime into five time intervals: $C_1 \leq 60$ minutes $< C_2 \leq 24$ hours $< C_3 \leq 7$ days $< C_4 \leq 4$ weeks $< C_5$. In addition, another contribution of this work is the introduction of new predictive attributes, aside from those already explored in the literature, as a means to improve the predictive capabilities of the approaches. In comparison with the approach which uses the attributes proposed by Yu *et al.* [3], our approach, which is comprised of the combination of the Random Forest algorithm applied over a subset of attributes identified by the attribute selection strategy based on the Information Gain metric, achieved the best results for the totality of the 20 projects we evaluated. The mean accuracy was of 45.28% for predicting pull request lifetime, which represents a normalized improvement of 14.68% in relation to the majority class.

This work is organized in six sections, besides this introduction. Section 2 presents some concepts about the pull-based development methodology. Section 3 discusses the materials and methods utilized in the experiments. In Section 4, we detail the results obtained by the regression and classification algorithms employed to predict pull request lifetime. Section 5 presents limitations and threats to the validity of the obtained results. Section 6 compares our results to the results of related work. Finally, Section 7 concludes this work, highlighting our contributions and discussing some future work.

2. Background

In the context of distributed software development, mainly in open-source projects, the possibility of having external developers sending contributions to projects fostered the growth and popularization of the pull-based development model [4]. In this sense, the contribution process demands frequent interaction among developers, as external requesters do not have write permission to directly update the main repository [5]. Even so, this model has been also adopted by the members of the project's core development team, thus enabling a code inspection process by other core team members prior to the effective acceptance of the code into the main repository [1].

A pull request is basically comprised of a request associated with a branch forked from the main repository containing a number of commits. Each commit is composed of code modifications [1]. Pull requests also serve the purpose of a notification mechanism to the integrators about incoming contributions sent to the project. This enables integrators to be aware of the need to review the code, and consequently decide whether a pull request should be accepted or rejected. Additionally, project hosting platforms (*e.g.*, GitHub, GitLab, and BitBucket) have at their disposal tools that make the contribution process in distributed software development more seamless, such as automatic code integration, automated tests, discussion forums, and social networking among developers [6, 7].

Pull requests can exhibit three states: **open**, indicating that the pull request has not yet been reviewed or has been reopened; **merged**, meaning that the code has been accepted and merged into the main repository; and **closed**, when

the pull request has been rejected. During review, if the code is deemed unsatisfactory, an integrator can then demand changes or clarifications via comments. Requesters can reply to those comments and, if they choose to do so, they can update the forked branch with new commits so as to have their pull request considered once again [1].

3. Materials and Methods

The algorithms and datasets used in the pull request lifetime prediction experiments are presented in this section. In Subsection 3.1, we define the two sets of attributes adopted. The open-source projects, as well as their characteristics, are presented in Subsection 3.2. Subsection 3.3 describes the classification, regression, and attribute selection algorithms utilized in the experimental process. Lastly, the methods adopted for the evaluation of the algorithms are discussed in detail in Subsection 3.4.

3.1. Sets of Attributes

This subsection describes the two sets of attributes examined in the context of pull request lifetime prediction. Set A is comprised of the attributes proposed by Yu *et al.* [3]. Set B, the one that we propose in this work, is made up of attributes from Set A as well as some other attributes that we considered to be important to the prediction of pull request lifetime.

Table 1 showcases the identifier, category, type, name, and description of the attributes that make up Set A. These attributes are organized into four categories: Project, Pull Request, Requester, and Social. In the Project category, the attributes represent information about the age of the project ($A.1$), development team size ($A.2$), the quotient between the average number of commits over files modified by the pull request and the total number of commits in the last three months ($A.3$), and the number of open pull requests in the project ($A.4$).

Attributes under the Pull Request category identify a few features on the profile of a contribution, such as: the number of words in the title and description of the pull request ($A.5$), the number of commits ($A.6$), the number of lines added to and excluded from the code ($A.7$), the number of comments ($A.8$), the time interval between pull request submission and the first interaction the integrator makes towards such request ($A.9$), whether the pull request used continuous integration[1] ($A.10$), whether the pull request is comprised of test files ($A.11$), whether the pull request has been submitted on a Friday ($A.12$), and whether the title or the description of the pull request has some reference to issues ($A.13$) or developers ($A.14$).

[1]Continuous integration is a software development practice in which team members integrate their work frequently – generally at least daily –, with the possibility of multiple integrations happening over the course of one day. Integration is verified via automated tests so as to detect errors quickly [8].

It is important to note that attributes $A.8$ and $A.9$, proposed by Yu *et al.* [3], are not known upon pull request submission, i.e., their values are only updated after submission. As a consequence, it is inappropriate to employ these two attributes to predict pull request lifetime at the moment of submission. Therefore, we do not consider any of the two attributes in order to reduce threats to the validity of this study.

Table 1: Attributes of the Set A.

Id	Category	Type	Name	Description
$A.1$	Project	Numeric	*ProjectAge*	Project's age, in minutes, at the time of the pull request submission.
$A.2$	Project	Numeric	*TeamSize*	Number of active integrators (who integrated at least one pull request from other requester) in the three months prior to the submission of the pull request.
$A.3$	Project	Numeric	*AreaHotness*	Average number of commits in files changed by the pull request divided by the number of commits carried out in the project, considering the last three months.
$A.4$	Project	Numeric	*Workload*	Number of opened pull requests in each project at the time of the current pull request submission.
$A.5$	Pull Request	Numeric	*DescriptionSize*	Number of words in the title and description of the pull request.
$A.6$	Pull Request	Numeric	*NumCommits*	Number of commits in the pull request.
$A.7$	Pull Request	Numeric	*NumChurn*	Number of lines added and deleted by the pull request.
~~$A.8$~~	~~Pull Request~~	~~Numeric~~	~~*NumComments*~~	~~Number of comments in the pull request.~~
~~$A.9$~~	~~Pull Request~~	~~Numeric~~	~~*FirstResponse*~~	~~Time interval, in minutes, between the pull request submission and the first comment from reviewers.~~
$A.10$	Pull Request	Binary	*UseCI*	Indicates whether the pull request used continuous integration.
$A.11$	Pull Request	Binary	*TestInclusion*	Indicates whether the pull request has at least one test file (based on the file name and regular expressions of the path).
$A.12$	Pull Request	Binary	*FridayEffect*	Indicates whether the pull request was received on a Friday.
$A.13$	Pull Request	Binary	*IssueTag*	Indicates the presence of a issue (*e.g. Ticket* #7) reference in the title or description of the pull request.
$A.14$	Pull Request	Binary	*MentionTag*	Indicates the presence of a developers (*e.g.* @mlimeira) reference in the title or description of the pull request.
$A.15$	Requester	Binary	*RequesterType*	Indicates whether the requester is an integrator of the project.
$A.16$	Requester	Numeric	*RequesterAcceptRate*	Acceptance rate of pull requests submitted by the requester.
$A.17$	Social	Numeric	*RequesterFollowers*	Number of GitHub developers who are followers of the requester.

The Requester category comprises attributes that provide information on the expertise and reputation of the requester, that is: an attribute that indicates whether the requester has permission to write in the main repository, i.e., whether he or she can integrate the code sent via pull requests ($A.15$) and an attribute that represents the percentage of pull requests submitted by the requester whose final state is merged ($A.16$). Finally, the Social category is represented by the attribute which indicates the number of followers a requester has ($A.17$).

Table 2 displays other attributes which we also deem important for the prediction of pull request lifetime. Set B contains the attributes contained in Set A plus the attributes in this table. These attributes are organized in the following categories: Project, Pull Request, Requester, and Social. The attributes in the Project Category represent: the number of pull requests received in the project prior to the submission of the current pull request ($B.1$), the percentage of commits sent to the project in the form of pull requests ($B.2$), the number of lines of code in the project ($B.3$), and the number of Stars ($B.4$).

Table 2: Attributes added to Set *A* to form Set *B*.

Id	Category	Type	Name	Description
B.1	Project	Numeric	*PrevPullProject*	Number of pull requests received by the project until the submission of the current pull request.
B.2	Project	Numeric	*ExternalContribs*	Percentage of commits made by pull requests in the project in the last month.
B.3	Project	Numeric	*Sloc*	Number of lines of code in the project.
B.4	Project	Numeric	*Stars*	Number of stars in the project at the time the pull request was submitted.
B.5	Pull Request	Numeric	*NumAddedFiles*	Number of files added by the pull request.
B.6	Pull Request	Numeric	*NumDeletedFiles*	Number of files deleted by the pull request.
B.7	Pull Request	Numeric	*NumModifiedFiles*	Number of files modified by the pull request.
B.8	Pull Request	Numeric	*NumChangedFiles*	Number of files changed (added, deleted, or modified) by the pull request.
B.9	Pull Request	Numeric	*NumSrcFiles*	Number of source code files changed by the pull request.
B.10	Pull Request	Numeric	*NumDocFiles*	Number of documentation files changed by the pull request.
B.11	Pull Request	Numeric	*NumOtherFiles*	Number of other files changed by the pull request.
B.12	Pull Request	Numeric	*NumTestChurn*	Number of test lines of code changed (added or deleted) by the pull request.
B.13	Requester	Numeric	*NumPrevPullRequester*	Number of pull requests submitted by the requester until the submission time.
B.14	Requester	Numeric	*RequesterNumFollowing*	Number of developers followed by the requester.
B.15	Requester	Numeric	*RequesterAge*	Time, in minutes, since the requester became a GitHub user.
B.16	Requester	Binary	*WatcherProject*	Indicates whether the requester is a follower of the project.
B.17	Requester	Numeric	*IssueEvents*	Number of interactions of the requester in issues prior to the submission.
B.18	Requester	Numeric	*IssueComments*	Number of comments of the requester in issues prior to the submission.
B.19	Requester	Numeric	*PullEvents*	Number of interactions of the requester in pull requests prior to the submission.
B.20	Requester	Numeric	*PullComments*	Number of comments of the requester in pull requests prior to the submission.
B.21	Requester	Numeric	*NumPrevCommits*	Number of commits of the requester prior to the submission.
B.22	Requester	Numeric	*NumCommitComments*	Number of comments of the requester in commits prior to the submission.
B.23	Social	Binary	*RequesterFollowsCT*	Indicates whether the requester follows any integrator.
B.24	Social	Binary	*CTFollowsRequester*	Indicates whether some integrator follows the requester.

The other attributes of the Pull Request category represent pull request features, such as: the number of files added (*B*.5), deleted (*B*.6) and modified (*B*.7), the total of modified files (*B*.8), the number of source code (*B*.9), documentation (*B*.10) and other types of files modified in the pull request (*B*.11), and the number of test lines of code modified (*B*.12).

Some attributes in the Requester category characterize the developer profile regarding, for example, the number of pull requests submitted (*B*.13), the number of developers followed by the requester (*B*.14), how long he or she has been a GitHub user (*B*.15), and whether the requester is a Watcher of the project (*B*.16). The remaining attributes in this category indicate whether the requester has participated in any event (sending comments, identifying issues, submitting pull requests, and/or making commits) within the project prior to the submission of the current pull request: the number of interactions in issues (*B*.17), the number of comments on issues (*B*.18), the number of interactions in pull requests (*B*.19), the number of comments in pull requests (*B*.20), the number of commits (*B*.21), and the number of comments on commits (*B*.22).

In the *Social* category, the attributes designate whether the requester is a

follower of some integrator in the core development team (*B*.23) and whether any integrator of the project is a follower of the requester (*B*.24).

3.2. Project Corpus

The experiments for pull request lifetime prediction involved 97,603 pull requests collected from 30 open-source projects using the GHTorrent tool [9]. The selected projects received more than 1,000 pull requests, and displayed constant pull request integration activity. Further, these projects are predominantly developed in the most popular programming languages[2] on GitHub: Java, JavaScript, Python, Ruby, and Scala.

Figure 1 shows the distribution of pull request lifetime in boxplots in the scale of log_{10}, according to the final state of pull requests. The distribution of pull request lifetime is expressed in minutes. However, these values can also be interpreted in a discrete manner, by the use of other time representations. For example, we can observe the following distribution among the pull requests exhibiting the merged final state for the different time intervals: 23.88% are integrated within minutes (from 1 to 60 minutes), 34.37% within hours (from 1 to 24 hours), 25.30% within days (from 1 to 7 days), 10.98% within weeks (from 1 to 4 weeks), and 5.47% within months (over 1 month). In this discretization, each interval comprises a significant number of pull requests.

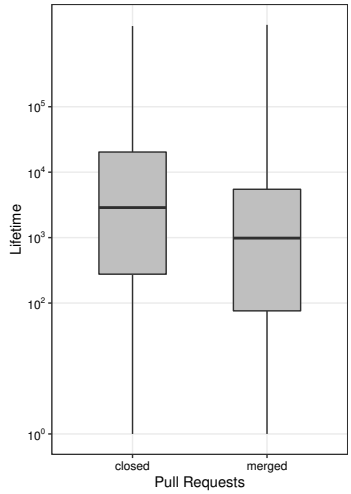

Figure 1: Lifetime in minutes of the pull requests in the selected projects (log scale).

Pull requests from the collected projects that had been accepted into the main repositories represented 76.50% of the total. Observing the boxplots of Figure 1, the pull requests displaying the merged final state were reviewed more

[2]GitHub allows the visualization and exploration of the popularity of programming languages among GitHub developers at `http://githut.info/`.

rapidly than those whose code was rejected. Among the accepted pull requests, the mean lifetime was of 11,735 minutes, approximately 8 days, and the median was of 941 minutes, that is, about 15 hours. On the other hand, rejected pull requests presented a mean lifetime of 42,442 minutes, approximately 29 days, and median of 2,716 minutes, that is, almost two days.

Figure 2 shows, for each of the 30 selected projects, the distribution of pull request lifetimes. Each boxplot represents the distribution of pull request lifetimes in the projects in a log_{10} scale. The red dots in the boxplots of some projects are considered to be outliers. All projects display pull requests within the five time intervals considered in our study (minutes, hours, days, weeks, and months), shown in the left portion of the figure. Besides, the majority of projects has a greater concentration of pull requests with time intervals of hours and days, except for projects appium and rosdistro, where about 45% and 70% of pull requests feature lifetimes in the order of minutes, respectively.

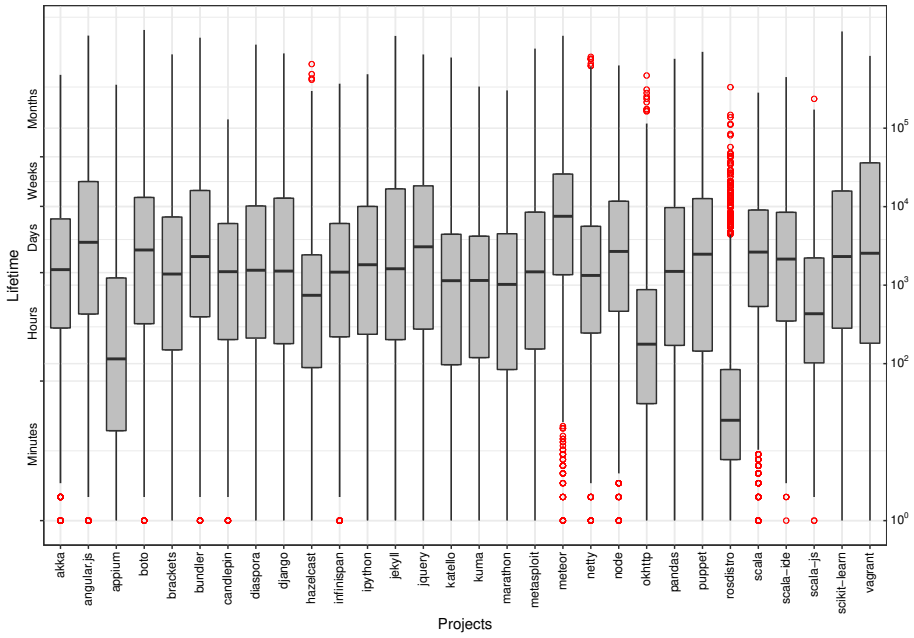

Figure 2: Lifetime in minutes of pull requests per project (log scale).

Table 3 contains the characteristics of each project. For each project, the column Language indicates the programming language that prevails across the projects' files. Column PRs represents, for each project, the total number of pull requests. Columns under the title Status indicate the percentages of accepted and rejected pull requests. Columns under the title Average Time show the mean pull request lifetime for each project sorted by final state, in days. Columns under the title Discrete Time represent, in percentage, the distribution of pull

requests in related to their lifetimes, in which each class is defined by a time interval. Moreover, the majority class of each project is displayed in boldface type.

Table 3: Characteristics of the projects.

			Status (%)		Average Time		Discrete Time (%)				
Language	Project	PRs	Merged	Closed	Merged	Closed	Minutes	Hours	Days	Weeks	Months
Java	candlepin	1,093	87.64	12.36	3.55	5.03	14.93	34.23	**36.42**	12.82	1.60
	hazelcast	1,832	91.27	8.73	2.13	18.79	20.92	**47.88**	23.92	4.87	2.41
	infinispan	3,990	67.14	32.86	5.14	4.25	13.58	**36.44**	33.87	13.02	3.09
	netty	2,028	49.70	50.30	6.20	15.17	15.34	**37.97**	29.73	10.50	6.46
	okhttp	1,194	88.27	11.73	1.38	17.47	35.37	**46.58**	10.89	4.61	2.55
	Average	2,027	76.80	23.20	3.68	12.14	20.03	40.62	26.97	9.16	3.22
JavaScript	angular.js	6,190	36.39	63.61	27.20	40.48	12.27	**26.89**	25.93	17.24	17.67
	appium	1,984	87.85	12.15	1.01	11.81	**45.77**	33.47	15.73	3.33	1.71
	brackets	4,575	87.23	12.77	7.54	31.53	19.21	**33.07**	27.41	12.66	7.65
	jquery	2,032	30.91	69.09	15.03	16.41	16.45	**25.74**	24.47	19.45	13.89
	meteor	1,299	57.51	42.49	22.75	51.30	10.55	16.86	**30.10**	22.79	19.71
	node	2,851	33.36	66.64	11.08	17.32	12.03	27.57	**33.22**	15.22	11.96
	Average	2,548	59.37	40.63	11.48	25.67	20.80	27.34	26.18	14.69	10.98
Python	boto	1,711	87.61	12.39	14.66	76.04	14.68	26.89	**29.94**	16.96	11.53
	django	6,006	68.88	31.12	15.72	38.54	15.88	**33.85**	22.88	12.29	15.10
	ipython	4,429	87.33	12.67	7.67	31.68	14.74	**31.56**	28.96	15.24	9.49
	kuma	3,565	89.20	10.80	2.69	9.23	19.24	**39.64**	30.52	8.92	1.68
	pandas	4,476	77.12	22.88	9.80	36.82	17.82	**32.49**	25.61	13.14	10.94
	rosdistro	10,155	93.75	6.25	0.20	1.40	**70.72**	26.12	2.49	0.55	0.12
	scikit-learn	3,213	80.82	19.18	17.37	87.24	14.63	**30.25**	25.15	14.60	15.38
	Average	5,168	85.64	14.36	7.55	33.27	27.43	32.01	22.55	10.49	7.52
Ruby	bundler	1,327	77.32	22.68	24.34	40.06	11.15	**33.38**	25.32	14.54	15.60
	diaspora	2,612	76.65	23.35	9.20	39.26	14.97	**34.11**	25.92	15.31	9.69
	jekyll	1,908	74.21	25.79	17.45	16.41	16.43	**31.99**	21.98	12.18	17.42
	katello	5,170	90.48	9.52	3.46	12.42	21.03	**37.90**	29.41	8.99	2.67
	metasploit-framework	5,341	86.20	13.80	10.62	35.33	17.66	**32.39**	27.45	12.08	10.43
	puppet	4,592	82.51	17.49	10.26	28.70	19.80	**26.24**	25.98	16.64	11.35
	vagrant	1,622	77.01	22.99	21.50	67.65	20.22	**25.40**	16.83	15.29	22.26
	Average	3,727	82.08	17.92	12.66	32.10	19.03	30.78	24.33	13.03	12.82
Scala	akka	4,175	71.62	28.38	4.32	9.14	9.08	**38.83**	34.92	13.56	3.62
	marathon	1,360	87.06	12.94	3.10	16.86	22.28	**36.99**	27.57	9.41	3.75
	scala	4,774	81.38	18.62	5.47	8.94	6.35	**35.13**	36.18	18.12	4.23
	scala-ide	1,050	86.95	13.05	5.93	15.62	9.45	**35.46**	34.32	15.18	5.59
	scala-js	1,049	94.19	5.81	1.61	15.59	16.97	**52.24**	25.17	4.19	1.43
	Average	2,482	84.24	15.76	4.09	13.23	12.82	39.73	31.63	12.09	3.72
	General Average	3,253	76.25	23.75	9.61	27.22	18.98	33.59	26.28	12.46	8.70

The majority of the selected projects has more accepted (*i.e.*, merged) pull requests than rejected (*i.e.*, closed) ones, except in the cases of projects netty, angular.js, jquery, and node. In average, all teams in the selected projects accepted the pull requests in less than a month. On the other hand, integration teams from projects angular.js, brackets, meteor, boto, django, ipython, pandas, scikit-learn, bundler, diaspora, metasploit-framwork, and vagrant take, in average, more than a month to reject pull requests. Projects jekyll and infinispan were the only ones which featured longer lifetimes for accepted pull requests than those of rejected pull requests.

In the lifetime distribution expressed in discrete values, we are able to observe

that integrators usually integrate pull requests between one and 24 hours, taking into account the selected projects. Conversely, in projects candlepin, meteor, node, and boto, the time period of 1 through 7 days encompasses the greatest percentage of the integrated pull requests.

The lifetime of pull requests may vary considerably. While most pull requests are integrated within minutes, a few hours, or even a few days, there is a significant number of pull requests with very large lifetimes, of over 30 days. In fact, pull request lifetimes may be impacted by different factors, such as the employment of continuous integration, code conflicts between the submitted code and the code already in the main repository, integration team size, work overload of integration tasks, and the lack of communication between the requester and the integration team.

3.3. Classification, Regression, and Attribute Selection Algorithms

In the experiments performed to provide estimates for the numeric values of pull request lifetimes, the following regression algorithms were evaluated: (i) Linear Regression [10] representing the multiple linear regression technique; (ii) M5P (M5Prime) [10] as an implementation of the regression trees algorithm; (iii) Random Forest [11] representing an ensemble method; and (iv) SMOReg (Sequential Minimal Optimization) [12] as the corresponding implementation of the SVM (Support Vector Machine) algorithm for regression.

As seen in the previous subsection, the distribution interval of pull request lifetimes in open-source projects is quite large. While a number of pull requests are integrated within minutes, several take weeks or even months to be thoroughly reviewed. As a result, the task of performing predictions by means of regression algorithms with numeric values for the target attribute may become susceptible to errors of large magnitude. In this context, instead of predicting a numeric value, the core development team may be interested in a more accurate lifetime prediction scheme. Thus, lifetime prediction can be carried out by making use of classification algorithms with discrete values representing time intervals.

In regards to the experiments where lifetime is expressed in discrete values, the following classification algorithms were evaluated: (i) IBk (Instance-Based) [13] as a representation of the k-NN algorithm; (ii) J48 [14], an implementation of the C4.5 algorithm of decision tree induction; (iii) Naive Bayes [15] as a correspondent of the Bayesian paradigm; (iv) Random Forest [11] as a representative of an ensemble method; and (v) SMO [16] representing the SVM algorithm for classification.

The regression and classification algorithms chosen for application in this study belong to different machine learning paradigms, and are often suggested for their competitive performance [17]. We utilized the implementations of such algorithms available in Weka[3] [18]. This tool offers a vast array of regression and classification algorithms, provides several preprocessing filters, attribute

[3]http://www.cs.waikato.ac.nz/ml/weka/

selection strategies, and the support necessary in the training and test phases of the experimental process.

Attribute selection strategies were employed to identify whether there is a subset of attributes with better performance in the classification task and, consequently, capable of improving the predictive capability of the algorithms. To that end, three strategies suggest by Hall and Holmes [19] were evaluated: CFS (Correlation-based Feature Selection) [20], IGAR (Information Gain Attribute Ranking) [10], and Relief-F [21].

The CFS strategy seeks a subset of attributes which present themselves to be more correlated to the class and less correlated with one another. The IGAR and Relief-F strategies make use of their own metrics to evaluate each attribute individually in relation to their capability of determining the class. Attributes are ranked following the metrics, and the n best attributes compose the set of returned attributes. The value of n is an input parameter for both strategies.

3.4. Experimental Process

The k-fold cross-validation method has been widely used to evaluate the accuracy of the classifiers [22]. This method randomly divides the dataset into k partitions of same size, and both the training and testing phases are carried out in k classification rounds. In each round, one of the k partitions is used for testing and the remaining ones for training. In this context, the accuracy is defined as the percentage of instances in the dataset whose classes have been correctly classified by the algorithm. However, this method is not appropriate for datasets exhibiting temporal dependency, since, due to the randomized generation of the test and training sets, future instances could be used to predict the past.

In order to prevent this problem from arising, we make use of the training-test sliding validation method [23], which preserves the chronological order of the instances when dividing the dataset into training and test sets. In this method, each classification round consists of 10% of the instances as training sets and the 1% immediately following this training set as the test set. For example, considering 100 ordered subsets of instances, in the first round, the training set comprises instances starting from the 1^{st} through the 10^{th} subset, and the test set is the 11^{th} subset. In the second round, the training set comprises instances starting from the 2^{nd} through the 11^{th} subset, and the test set is the 12^{th} subset, and so forth. The sliding window shifts 1% per round, which allows for the test of 90% of the instances. The accuracy of the method is estimated by the percentage of instances tested whose classes have been accurately classified by the algorithm in all rounds.

For the verification of the predictive performance of the regression algorithms, we made use of Spearman's nonparametric correlation coefficient (ρ) and mean error metrics: RMSE (Root Mean Squared Error) and NRMSE (Normalized Root Mean Squared Error).

To evaluate the predictive performance of the classifiers, we employed accuracy and normalized improvement metrics obtained by a classifier when com-

pared to the majority class (baseline). The idea behind the normalized improvement metric is to evaluate how much a classifier increases the accuracy in comparison with the baseline improvement range. This range is the difference between 100% and the accuracy of the baseline (*i.e.*, the majority class in our case). The normalized improvement of an evaluated classifier in some project p, that is, over the dataset representing the project p, is defined by Equation 1 [24].

$$f_p = \begin{cases} \dfrac{Accuracy_p - Baseline_p}{1 - Baseline_p}, if\ Accuracy_p > Baseline_p \\ \\ \dfrac{Accuracy_p - Baseline_p}{Baseline_p}, otherside \end{cases} \quad (1)$$

where $Accuracy_p$ is the accuracy obtained by the classification algorithm in project p and $Baseline_p$ represents the accuracy when the majority class is suggested in project p.

Finally, to calculate the statistical significance of the results [25], we adopted the Friedman [26] (nonparametric) and Nemenyi [27, 28] (post hoc) tests. In both tests, the significance level was of 0.05.

4. Results and Discussion

This section offers the results obtained as well as a discussion on the pull request lifetime prediction experiments, organized as follows. Subsection 4.1 brings the results achieved with regression algorithms using the attributes in Set A. Subsection 4.2 describes the input parameter tuning procedure for some classification algorithms taking into account the following discretization for the target attribute: from 1 through 60 minutes, from 1 through 24 hours, from 1 through 7 days, from 1 through 4 weeks, and more than a month. In Subsection 4.3, we present the accuracies achieved by the classification algorithms using the attributes in Set B. After that, Subsection 4.4 shows the results returned by the utilization of attribute selection strategies. Subsection 4.5 compares the results achieved by the Random Forest algorithm with three different approaches. Finally, Subsection 4.6 discusses the importance of the attributes for the lifetime prediction context in five projects.

4.1. Lifetime Prediction with Regression Algorithms

In this subsection, we evaluate the predictive power of the approach proposed by Yu *et al.* [3], which employs the linear regression algorithm. In addition, we also evaluated the performance of other regression algorithms in the prediction of pull request lifetime using the attributes of Set A. In order to do this, we considered the following algorithms: Random Forest, M5P, and SMOReg with `Polynomial` kernel. The regression algorithms were evaluated by means of the training-test sliding validation method. In this experiment, the target attribute value, which is represented by the pull request lifetime, is expressed in minutes.

In Table 4, for each project, the column Lang. contains the predominant programming language throughout the project's files. Column ρ represents the results for the Spearman correlation coefficient between the actual lifetime and the one predicted by the algorithm. Columns RMSE and NRMSE display the values for root mean square error and normalized root mean squared error, respectively, obtained by the algorithms: Linear Regression (LR), M5P, Random Forest, and SMOReg. The higher values for the Spearman coefficient indicate a greater correlation between the ranks of the actual and predicted values. The lower values for the RMSE and NRMSE metrics represent a greater accuracy rate achieved by the regression model. In each project, the best results for the NRMSE metric are displayed in boldface type.

Table 4: Results obtained by different regression algorithms using the Set *A*.

		LR			M5P			Random Forest			SMOReg		
Lang.	Project	ρ	RMSE	NRMSE	ρ	RMSE	NRMSE	ρ	RMSE	NRMSE	ρ	RMSE	NRMSE
Java	candlepin	0.406	9,677	397	0.413	60,055	1,162	0.360	7,610	456	0.408	9,043	**264**
	hazelcast	0.426	105,717	1,408	0.509	114,414	790	0.390	11,649	512	0.418	34,820	**475**
	infinispan	0.457	15,824	953	0.462	14,315	**360**	0.424	12,735	521	0.484	13,313	535
	netty	0.416	159,372	6,331	0.422	88,996	8,263	0.415	37,360	2,174	0.459	47,325	**771**
	okhttp	0.319	65.202	20,648	0.391	110,273	19,296	0.381	12,679	**316**	0.410	21,192	7,753
JavaScript	angular.js	0.712	105,052	5,286	0.714	78,579	**1,391**	0.550	80,849	2,172	0.730	105,671	2,527
	appium	0.370	12,579	11,184	0.397	26,343	560	0.407	6,515	443	0.408	7,244	**148**
	brackets	0.693	63,178	6,475	0.707	19,350	**395**	0.560	23,958	1,104	0.739	35,091	1,246
	jquery	0.444	57,530	3,725	0.458	54,675	2,238	0.406	45,896	2,091	0.479	47,834	**1,226**
	meteor	0.606	203,590	14,312	0.627	64,203	2,618	0.476	75,380	**2,613**	0.632	180,619	4,103
	node	0.779	29,079	1,928	0.797	75,082	**206**	0.519	25,694	1,126	0.798	28,023	961
Python	boto	0.512	67,149	3,519	0.545	48,022	**1,051**	0.351	55,895	1,590	0.521	55,970	1,885
	django	0.734	62,940	5,535	0.700	88,260	**1,805**	0.622	49,376	1,938	0.738	68,398	2,447
	ipython	0.694	130,832	2,104	0.671	202,679	**518**	0.604	23,346	688	0.714	93,862	770
	kuma	0.498	12,890	550	0.497	87,936	344	0.449	9,457	345	0.502	9,924	**187**
	pandas	0.645	60,869	4,719	0.663	86,800	**837**	0.621	39,438	999	0.674	57,813	2,259
	rosdistro	0.220	4,843	226	0.218	25,053	424	0.211	2,318	124	0.229	2,215	**35**
	scikit-learn	0.682	68,864	4,404	0.688	42,683	**562**	0.566	67,427	1,939	0.685	71,519	3,053
Ruby	bundler	0.425	344,875	16,959	0.485	195,259	**2,493**	0.390	87,145	3,563	0.441	437,675	16,845
	diaspora	0.499	72,067	3,783	0.518	57,068	**1,103**	0.422	45,350	1,853	0.534	59,010	1,385
	jekyll	0.506	83,058	4,114	0.528	78,415	2,049	0.459	72,924	2,090	0.540	77,779	**1,332**
	katello	0.574	18,257	1,107	0.550	414,327	1,215	0.460	14,089	569	0.557	15,354	**332**
	metasploit-framework	0.683	65,815	4,573	0.660	124,126	**1,207**	0.588	34,491	1,513	0.686	52,340	1,720
	puppet	0.713	47,522	5,214	0.681	62,011	4,568	0.566	33,806	2,699	0.722	39,184	**1,822**
	vagrant	0.437	513,906	11,284	0.461	138,044	**4,730**	0.409	73,786	5,665	0.451	145,508	6,138
Scala	akka	0.563	20,452	1,299	0.566	18,430	1,047	0.485	17,255	812	0.571	18,418	**449**
	marathon	0.413	40,960	1,534	0.436	219,106	818	0.460	14,686	**504**	0.461	18,528	759
	scala	0.425	157,333	803	0.563	56,204	564	0.479	10,980	608	0.600	12,637	**519**
	scala-ide	0.385	38,955	**139**	0.465	165,888	825	0.458	18,058	470	0.457	41,807	544
	scala-js	0.602	13,304	843	0.395	13,709	64	0.414	8,060	**37**	0.468	12,567	64
General Average		0.528	77,052	4,845	0.540	87,698	1,975	0.463	28,398	1,274	0.551	54,625	2,034

Even though values above 0.7 for the correlation coefficient have been observed in some projects, the RMSE and NRMSE values presented by the linear regression models are too high. The lowest root mean squared error obtained was in project rosdistro: RMSE = 4,843. This means that the mean difference between the actual and the predicted values is of 4,843 minutes. The evaluation of the linear models revealed narrow mean squared errors, which motivated the evaluation of other regression algorithms using Set *A*.

In most cases, the values obtained by M5P, Random Forest, and SMOReg

were superior to those exhibited by the Linear Regression algorithm employed by Yu *et al.* [3]. Taking into account the values of the NRMSE metric of Table 4, algorithms M5P and SMOReg achieved better performances. The M5P algorithm returned the best results for 13 out of the 30 projects. SMOReg achieved the best results for 12 projects, and Random Forest was able to reach the best results for 5 projects.

We can observe that the Linear Regression algorithm only obtained the best result in comparison to the other three algorithms for one project: scala-ide. The lowest NRMSE values were achieved in projects appium (148), node (206), rosdistro (35), scala-ide (139) and scala-js (37).

Even though the evaluated algorithms (M5P, Random Forest, and SMOReg) improved on the results obtained by the approach adopted by Yu *et al.* [3] (Linear Regression), the error rate is still high, which limits the applicability of the prediction. For instance, the lowest root mean squared error of linear regression (project rosdistro) dropped from 4,843 to 2,215 minutes when using SMOReg, but is still a high error rate. In the decision-making process, it may be simpler and more useful to requesters and integrators to estimate the lifetime of pull requests with discrete values, defining five time intervals in the order of: minutes (from 1 through 60 minutes), hours (from 1 through 24 hours), days (from 1 through 7 days), weeks (from 1 through 4 weeks), and months (more than a month). This discretization is intended to devise more useful and effective predictive models. In this sense, from the next subsection on, we evaluate the employed classification algorithms using the attributes of Set A and Set B to predict the lifetime in terms of those five time intervals.

4.2. Tuning the Input Parameters for the Classification Algorithms

Preliminary experiments were conducted to calibrate the input parameter of the IBk, SMO, and Random Forest classification algorithms. These experiments were executed with the attributes in Set B in 10 projects – the two with the lowest number of pull requests for each language: candlepin and okhttp (Java); appium and meteor (JavaScript); boto and scikit-learn (Python); bundler and vagrant (Ruby); scala-ide and scala-js (Scala). These projects have been set aside in the algorithms' performance evaluation and comparison that took place later on.

The following input parameters have been tuned prior to the actual execution of the next experiments: the k value (number of neighbors) in the IBk algorithm, the kernel of the SMO algorithm (Polynomial, Puk, or RBF), and the number of trees built during the execution of the Random Forest algorithm. The classification algorithms were evaluated via the training-test sliding validation method with lifetimes expressed in discrete values.

In the next experiments, each table shows four metrics to measure the performance of the algorithms: (i) Accuracy Average, defined by the quotient of the sum of the accuracies of each algorithm and the total of projects; (ii) average of the normalized improvement, or simply, Improvement Average, which represents the sum of the normalized gains (improvement in the accuracy compared to the majority class) obtained by the accuracies of each project divided by the total

number of projects; (iii) Ranking Average, calculated as follows: for each project, each algorithm receives an ordered rank number (1 for the best accuracy, 2 for the second best accuracy, and so on); then, the rank numbers of each algorithm are summed up and the result is divided by the number of projects; the lower the average is, the better the algorithm's performance is; and (iv) Number of Wins, defined by the number of times the algorithm achieved the best accuracy, considering all projects.

Table 5 shows the results obtained by ranging the k parameter in the IBk algorithm, with odd values ranging from 1 through 29. Algorithm IBk with a k value of 23 displayed the best results taking into consideration all the metrics.

Table 5: Results obtained by IBk with different values of k.

	k **Value**										
Metrics	**1**	**3**	**5**	**7**	**9**	**11**...	**21**	**23**	**25**	**27**	**29**
Accuracy Average (%)	35.06	35.20	36.11	36.41	37.14	37.20	38.30	**38.68**	38.16	37.90	37.94
Improvement Average (%)	-1.01	-0.76	2.12	2.63	4.28	3.68	4.73	**5.86**	4.75	4.38	4.03
Ranking Average	13.70	13.90	11.75	11.20	9.40	9.70	5.00	**2.70**	4.10	5.75	6.20
Number of Wins	0	0	0	0	0	0	0	**4**	0	0	1

Table 6 showcases the results achieved by the SMO algorithm using three kernels: Polynomial, Puk, and RBF. The Polynomial kernel showed the best performance considering all metrics.

Table 6: Results obtained by SMO with different kernels.

	SMO kernel		
Metrics	**Polynomial**	**Puk**	**RBF**
Accuracy Average (%)	**40.19**	38.41	36.76
Improvement Average (%)	**4.99**	2.20	-1.02
Ranking Average	**1.20**	2.00	2.80
Number of Wins	**9**	1	0

The results for the variation of the number of trees used in the Random Forest algorithm are presented in Table 7. For this parameter, we used 50 trees as minimum and a maximum value of 500, with an iterative increment of 50 trees. The value of 300 trees achieved the best accuracy in three projects, and the best results for the accuracy average, improvement average, and the ranking average.

Table 7: Results obtained by Random Forest with different numbers of trees.

	Number of trees									
Metrics	**50**	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
Accuracy Average (%)	42.64	42.78	43.09	43.00	43.03	**43.20**	43.01	42.91	42.93	42.90
Improvement Average (%)	6.21	6.75	7.46	7.35	7.27	**7.50**	7.24	7.07	7.00	7.13
Ranking Average	7.40	6.65	4.95	5.10	4.95	**3.25**	5.15	5.75	5.70	6.10
Number of Wins	0	2	**3**	0	1	**3**	0	1	0	0

Based on these results, $k = 23$, Polynomial kernel, and 300 decision trees were used as input parameters for the IBk, SMO, and Random Forest algorithms, respectively. The following subsection presents the comparison between these classification algorithms using the attributes of Set B.

4.3. Lifetime Prediction with Classification Algorithms

The experiment outlined in this subsection consists in the comparison of some classification algorithms using the predictive attributes in Set B to predict pull request lifetime. The input parameters for the IBk, Random Forest (RF), and SMO algorithms received the values found in the previous experiments. The J48 and Naive Bayes (NB) algorithms were evaluated with default values for the input parameters. We used the 20 projects that remained unused after the parameter tuning experiments. We performed the evaluation of the classification algorithms by making use of the training-test sliding validation method, and pull request lifetime assumed five discrete values ($C_1 \leq$ 60 minutes $< C_2 \leq$ 24 hours $< C_3 \leq$ 7 days $< C_4 \leq$ 4 weeks $< C_5$).

In Table 8, column Baseline represents, for each project, the percentage of the majority class, that is, the time interval with the most integrated pull requests. The columns under Accuracy display, for each project, the accuracy rates obtained with each of the classification algorithms for lifetime prediction using Set B. Further, for each project and metric, the best result is shown in boldface type.

Table 8: Results obtained by the classification algorithms using the Set B

			Accuracy (%)				
Language	**Project**	**Baseline**	**IBk**	**J48**	**NB**	**RF**	**SMO**
Java	hazelcast	47.01	47.52	41.55	41.12	**47.98**	46.28
	infinispan	35.37	38.60	36.89	32.09	**44.08**	43.89
	netty	38.88	36.85	38.37	31.14	**42.70**	41.36
JavaScript	angular.js	26.91	30.97	**43.96**	33.11	42.44	40.57
	brackets	31.03	32.90	40.39	31.26	**41.19**	39.54
	jquery	26.30	27.61	28.59	27.39	31.30	**32.28**
	node	34.80	34.94	**50.70**	31.91	48.64	45.42
Python	django	35.37	34.67	43.28	28.97	**46.74**	45.33
	ipython	31.83	31.14	38.01	30.04	**41.96**	40.51
	kuma	39.08	39.81	36.77	31.99	43.51	**44.13**
	pandas	32.39	34.98	42.79	37.85	**44.64**	43.01
	rosdistro	**71.52**	70.49	61.77	54.49	68.39	70.99
Ruby	diaspora	35.30	30.68	36.05	29.12	**38.06**	37.95
	jekyll	34.75	30.94	31.87	31.23	**37.42**	36.32
	katello	38.53	39.75	37.25	31.39	44.54	**45.72**
	metasploit-framework	31.34	32.14	39.25	31.29	**43.20**	42.59
	puppet	27.05	29.37	35.43	27.85	**38.17**	36.64
Scala	akka	38.11	39.58	38.92	35.05	44.75	**47.22**
	marathon	37.01	36.52	33.93	33.04	**37.73**	36.92
	scala	38.28	40.82	39.82	23.89	47.41	**47.62**
	Accuracy Average (%)		37.01	39.78	32.71	**43.74**	43.21
	Improvement Average (%)		-0.11	3.95	-9.94	**10.82**	10.00
	Ranking Average		3.65	3.15	4.85	**1.45**	1.90
	Number of Wins		0	2	0	**12**	5

The RF algorithm featured the best results taking into account all four metrics. The RF algorithm showed an average normalized improvement of 10.82%, obtaining the best accuracy rate in 12 out of the 20 projects. Based on these results, the RF algorithm was selected for a follow-up experiment now incorporating attribute selection strategies using Set *B*.

4.4. Identifying Better Attribute Subsets

In order to evaluate whether there is a subset of attributes capable of improving classification algorithm performance, the attributes of Set *B* were subjected to attribute selection strategies. We employed three strategies to that end: CFS, IGAR, and Relief-F. Strategies IGAR and Relief-F have as input parameter the size of the set of attributes which must be returned. As such, an experiment to tune this parameter was conducted. We evaluated the strategies using the 20 projects used in the previous experiment, considering, in all training sets, subsets of 1 through 40 attributes.

Table 9 brings the results obtained by the Relief-F strategy. The best performance was achieved with subsets of the 33 most relevant predictive attributes. Some columns in the table, which represent other numbers of attributes, presented inferior results and were omitted to emphasize the significant results.

Table 9: Results obtained with the Relief-F strategy.

	Number of Attributes								
Metrics	**1**	**2...**	**20**	**21**	**22...**	**32**	**33**	**34...**	**40**
Accuracy Average (%)	37.57	37.67	43.80	43.75	43.75	44.15	**44.27**	44.02	44.02
Improvement Average (%)	-0.18	-0.14	10.78	10.72	10.73	11.36	**11.57**	11.13	11.18
Ranking Average	37.55	38.98	14.53	15.40	16.25	12.25	**9.73**	11.85	11.65
Number of Wins	0	0	1	0	0	2	**3**	2	1

Table 10 shows the results achieved by the IGAR strategy. The best performance was featured again by the 33 most relevant predictive attributes. Once again, some columns were omitted to focus on significant results.

Table 10: Results obtained with the IGAR strategy.

	Number of Attributes								
Metrics	**1**	**2...**	**20**	**21**	**22...**	**32**	**33**	**34...**	**40**
Accuracy Average (%)	38.22	39.13	43.81	43.88	43.90	45.04	**45.28**	44.98	44.93
Improvement Average (%)	1.42	3.23	11.20	11.30	11.31	12.74	**13.10**	12.63	12.55
Ranking Average	38.68	38.93	19.88	18.63	18.10	8.38	**6.20**	8.73	8.78
Number of Wins	0	0	0	0	1	2	**5**	2	3

As subsets of 33 attributes for strategies Relief-F and IGAR displayed the best results, these values were used as input parameters in the next experiment, which compares the performance of the three attribute selection techniques. Table 11 shows, for each project, the accuracy rates obtained by the Random Forest algorithm following the utilization of attribute selection strategies with

Set *B*. Column Baseline represents the percentage of the majority class. Besides, for each project and metric, the best result is in bold.

Table 11: Results obtained with attribute selection using the Set *B*.

			Accuracy (%) Random Forest		
Language	**Project**	**Baseline**	**CFS**	**IGAR (33)**	**Relief-F (33)**
Java	hazelcast	47.01	47.28	47.58	**47.95**
	infinispan	35.37	41.80	**45.57**	42.06
	netty	38.88	40.11	**44.62**	43.67
JavaScript	angular.js	26.91	39.24	**45.46**	45.08
	brackets	31.03	41.81	**44.84**	43.92
	jquery	26.30	30.43	**31.96**	31.52
	node	34.80	50.50	52.95	**53.80**
Python	django	31.83	41.74	**47.38**	46.09
	ipython	31.83	39.69	**44.99**	42.86
	kuma	39.08	41.46	43.88	**43.92**
	pandas	32.39	44.32	**50.46**	47.04
	rosdistro	**71.52**	64.15	67.97	67.11
Ruby	diaspora	35.30	35.88	39.90	**40.24**
	jekyll	34.75	33.60	**38.86**	36.36
	katello	37.90	58.87	**60.36**	58.38
	metasploit-framework	31.34	41.09	**44.37**	42.98
	puppet	27.05	35.63	39.52	**39.83**
Scala	akka	38.11	43.16	**44.97**	43.85
	marathon	37.01	**38.54**	37.81	37.41
	scala	38.28	42.85	**47.86**	46.07
	Accuracy Average (%)		41.68	**48.28**	44.27
	Improvement Average (%)		7.58	**13.10**	11.57
	Ranking Average		2.90	**1.30**	1.80
	Number of Wins		1	**13**	5

The Random Forest algorithm in conjunction with the IGAR attribute selection strategy, using the 33 most relevant attributes, achieved superior results in comparison with the remaining strategies, considering all metrics employed.

4.5. Comparing Three Approaches

The focus of this experiment is to compare the performance of three approaches to predict pull request lifetime. Each approach takes into account the combination of a classification algorithm and a set of attributes. **Approach A** represents the Random Forest algorithm with a fixed value of 300 decision trees as input parameter, using the predictive attributes from Set *A* proposed by Yu *et al.* [3]. **Approach B** represents the combination of the Random Forest algorithm also having 300 decision trees as input parameter, using predictive attributes from Set *B*, proposed in this work. **Approach C** represents the Random Forest algorithm having again 300 decision trees as input, using the IGAR attribute selection strategy with Set *B*.

A summary of the results achieved by each of the three approaches is presented in Table 12. For each metric, the best result is in boldface type.

Table 12: Results obtained by the Approaches A, B, and C.

	Approaches		
Metrics	**Approach A**	**Approach B**	**Approach C**
Accuracy Average (%)	41.48	43.74	**45.28**
Improvement Average (%)	8.94	12.40	**14.68**
Ranking Average	2.90	2.00	**1.10**
Number of Wins	0	2	**18**

We can observe that the subsets of attributes of Approach C featured the best results for every metric. In the comparison of the three approaches, considering the accuracy values obtained for each of the 20 projects, the Friedman test rejected the null hypothesis with $p\text{-}value = 9.214 \times 10^{-8}$, implying that there is a significant difference in accuracy between the approaches. The Nemenyi test revealed that the results obtained by Approach C are statistically superior to those achieved by Approach A ($p\text{-}value = 3.8 \times 10^{-8}$) and Approach B ($p\text{-}value = 0.012$).

Table 13 presents the average normalized improvement for each project in order to provide more detail on the evaluation of the three approaches. These results show that the use of the IGAR attribute selection strategy with Set *B* (Approach C) was able to achieve superior average normalized improvement when compared to the other two approaches. For instance, in projects angular, brackets, node, and pandas, the normalized improvement were over 20%. Moreover, Approach C obtained the best normalized improvement in 18 out of the 20 projects. These results show that the identification of an appropriate subset of attributes allowed for an increase in the capability of predicting the pull request lifetime for the majority of the evaluated projects.

Table 13: Normalized improvement obtained by the three approaches, by project.

		Normalized Improvement (%)		
Language	**Project**	**Approach A**	**Approach B**	**Approach C**
Java	hazelcast	-3.68	**1.83**	1.08
	infinispan	6.48	13.48	**15.68**
	netty	3.17	6.25	**9.39**
JavaScript	angular.js	20.89	21.25	**25.38**
	brackets	13.53	14.73	**20.02**
	jquery	5.02	6.78	**7.68**
	node	24.03	21.23	**27.84**
Python	django	10.82	17.59	**18.58**
	ipython	10.09	14.86	**19.30**
	kuma	-1.77	7.27	**7.88**
	pandas	19.88	18.12	**26.73**
	rosdistro	-11.21	**-4.38**	-4.96
Ruby	diaspora	0.12	4.27	**7.11**
	jekyll	-4.63	4.09	**6.30**
	katello	4.20	9.78	**9.86**
	metasploit-framework	14.30	17.27	**18.98**
	puppet	13.75	15.24	**17.09**
Scala	akka	8.76	10.73	**11.08**
	marathon	-0.16	1.14	**1.27**
	scala	9.02	14.79	**15.52**

4.6. Importance of the Attributes

In this subsection, we identify the most relevant attributes of Set *B*. To that end, projects which displayed the highest normalized improvement for each language were selected: infinispan, metasploit-framework, node, pandas, and scala. For each training set used in the evaluation of the classifiers, the attributes were sorted according to the value provided by the Information Gain metric. The first rank represents the attribute of highest value, whereas the last rank represents the attribute of lowest value. Then, the ranks of each attribute were summed up, and this value was divided by the number of training sets evaluated. This quotient represents the average of the ranks of each attribute.

Table 14 displays the relevance of the attributes in the five evaluated projects. Each project has two columns representing attribute ranks. Column Attribute contains the label of each attribute, sorted by the value in column Average, which is the average of the ranks considering the Information Gain values for each attribute. As such, the most relevant attributes are found at the uppermost portion of the table, and the least relevant ones are in the bottommost portion. Columns under the title Legend contain the label and name of each attribute, solely to ease their identification.

In the five projects presented, attributes Workload (*A*.4) and MentionTag (*A*.14) appear among the best rankings. On the other hand, some attributes associated with project characteristics, such as ProjectAge (*A*.1), TeamSize (*A*.2), PrevPullProject (*B*.1), and ExternalContribs (*B*.2) are amongst the ones which contribute the least to the prediction of pull request lifetime.

Some attributes proposed in Set *B* proved to be relevant in predicting pull request lifetime, for instance: NumAddedFiles (*B*.5), NumTestChurn (*B*.12), and WatcherProject (*B*.16). In projects metasploit and pandas, attributes that characterize previous developer interactions in the project (*B*.17 through *B*.22) were also shown to be good for the prediction of pull request lifetime.

Furthermore, a few pieces of information on the size of the pull request such as attributes NumCommits (*A*.6) and NumChurn (*A*.7) also showed a significant correlation towards pull request lifetime. The attribute RequesterType (*A*.15) also seems to impact the time taken to tackle pull requests.

5. Threats to Validity

In this section, we point out some decision and aspects of this work which might have imposed threats to some results and conclusions. Due to the complexity of determining the time an integrator takes effectively performing a pull request review and integration, pull request lifetime was considered to be the time between submission and conclusion of the review process of a pull request, which culminates in its integration (acceptance) or closure (rejection). Therefore, the lifetime considered throughout this work is not exactly the integration time of a pull request. This is so due to the possibility of a pull request remaining open for reasons beyond the integration process, such as, for example, the integrator having an excessive workload, and the lack of interaction with the requester. Nevertheless, this overall lifetime is what requesters actually perceive,

Table 14: Importance of the attributes.

	Project											
	infinispan		metasploit		node		pandas		scala		Legend	
Rank	Attribute	Average	Attribute	Average	Attribute	Average	Attribute	Average	Attribute	Average	Id	Name
1	*A*.4	1.00	*A*.4	1.00	*A*.4	2.67	*A*.4	1.00	*A*.4	1.00	*A*.1	ProjectAge
2	*A*.14	6.94	*A*.14	4.88	*A*.7	5.74	*A*.14	2.77	*A*.14	3.87	*A*.2	TeamSize
3	*A*.15	9.33	*A*.7	5.04	*A*.14	5.86	*A*.6	3.72	*A*.15	8.89	*A*.3	AreaHotness
4	*B*.16	9.98	*A*.6	5.84	*B*.5	7.88	*A*.7	7.90	*B*.16	9.06	*A*.4	Workload
5	*A*.12	10.44	*B*.19	7.19	*B*.9	8.80	*B*.12	9.70	*B*.24	9.89	*A*.5	DescriptionSize
6	*A*.7	11.89	*B*.13	10.07	*A*.6	8.83	*B*.19	10.76	*A*.12	10.18	*A*.6	NumCommits
7	*B*.23	13.42	*B*.17	10.73	*B*.12	9.00	*B*.18	11.33	*B*.23	10.42	*A*.7	NumChurn
8	*A*.6	13.50	*B*.18	11.92	*B*.8	9.03	*B*.17	11.36	*A*.7	12.32	*A*.10	UseCI
9	*B*.24	14.56	*B*.5	13.47	*B*.7	11.00	*B*.13	12.90	*A*.6	14.32	*A*.11	TestInclusion
10	*B*.12	14.88	*A*.15	14.46	*A*.12	11.13	*B*.21	13.14	*A*.11	16.69	*A*.12	FridayEffect
11	*B*.11	16.16	*B*.20	14.99	*B*.23	12.23	*A*.17	14.7	*B*.12	17.00	*A*.13	IssueTag
12	*A*.11	16.36	*A*.16	15.32	*B*.16	12.57	*B*.20	14.81	*B*.22	17.79	*A*.14	MentionTag
13	*B*.10	17.50	*A*.17	15.67	*B*.24	12.66	*A*.5	16.37	*B*.11	18.39	*A*.15	RequesterType
14	*B*.5	17.98	*B*.21	16.23	*A*.15	12.70	*A*.15	17.01	*A*.3	18.47	*A*.16	RequesterAcceptRate
15	*B*.22	18.60	*B*.8	16.28	*B*.11	16.39	*B*.16	18.26	*A*.13	19.33	*A*.17	RequesterNumFollowers
16	*B*.19	18.94	*A*.5	16.36	*A*.3	16.80	*B*.8	18.36	*B*.5	19.36	*B*.1	PrevPullProject
17	*A*.3	19.43	*B*.24	17.98	*A*.11	16.96	*B*.9	18.69	*B*.1	19.69	*B*.2	ExternalContribs
19	*B*.6	19.46	*B*.16	18.96	*A*.13	17.78	*B*.7	18.81	*B*.6	19.97	*B*.3	Sloc
20	*B*.9	19.79	*B*.23	19.34	*B*.6	19.42	*B*.23	19.49	*B*.2	20.16	*B*.4	Stars
21	*B*.7	19.83	*A*.12	20.84	*B*.10	20.57	*B*.22	19.70	*B*.10	20.24	*B*.5	NumAddedFiles
22	*A*.13	19.84	*B*.22	21.59	*B*.22	20.86	*A*.12	20.48	*B*.7	20.90	*B*.6	NumDeletedFiles
23	*B*.3	20.38	*B*.9	23.90	*B*.1	22.29	*A*.16	20.81	*B*.8	20.96	*B*.7	NumModifiedFiles
24	*B*.8	20.92	*A*.11	23.94	*B*.2	26.01	*B*.24	22.14	*B*.9	21.22	*B*.8	NumChangedFiles
25	*B*.18	21.74	*B*.10	24.58	*B*.15	26.62	*B*.11	24.04	*B*.17	22.82	*B*.9	NumSrcFiles
26	*A*.10	23.14	*B*.7	24.60	*B*.18	27.03	*B*.5	24.49	*A*.17	23.32	*B*.10	NumDocFiles
27	*B*.1	23.33	*B*.6	25.20	*B*.19	27.10	*B*.10	25.07	*B*.18	25.74	*B*.11	NumOtherFiles
28	*B*.13	25.04	*B*.11	25.04	*B*.20	27.13	*B*.6	25.28	*B*.19	25.74	*B*.12	NumTestChurn
29	*B*.2	25.54	*A*.3	26.21	*B*.17	27.68	*B*.14	26.09	*B*.21	26.06	*B*.13	NumPrevPullRequester
30	*B*.20	26.12	*B*.15	26.27	*A*.17	28.46	*A*.11	26.64	*B*.15	26.10	*B*.14	RequesterNumFollowing
31	*B*.14	26.34	*B*.12	26.53	*A*.1	29.53	*B*.15	27.59	*A*.1	26.50	*B*.15	RequesterAge
32	*B*.17	27.41	*B*.14	29.11	*B*.14	30.21	*A*.10	31.11	*B*.14	27.54	*B*.16	WatcherProject
33	*B*.21	29.26	*B*.3	29.93	*B*.13	31.91	*A*.3	32.17	*B*.4	27.87	*B*.17	IssueEvents
34	*B*.15	29.49	*B*.4	30.99	*B*.3	32.14	*A*.2	32.29	*B*.20	28.30	*B*.18	IssueComments
35	*A*.1	29.92	*A*.2	31.49	*A*.16	32.48	*B*.3	32.49	*A*.16	28.68	*B*.19	PullEvents
36	*B*.4	30.00	*A*.10	32.17	*A*.5	33.24	*B*.4	32.78	*B*.3	29.12	*B*.20	PullComments
37	*A*.16	30.77	*B*.1	33.04	*B*.4	33.34	*B*.1	32.83	*A*.10	31.18	*B*.21	NumPrevCommits
38	*A*.17	32.51	*A*.1	33.72	*B*.21	34.62	*A*.13	33.20	*A*.2	31.94	*B*.22	NumCommitComments
39	*A*.2	34.83	*A*.13	34.30	*A*.2	36.49	*A*.1	33.32	*A*.15	32.82	*B*.23	RequesterFollowsCT
40	*A*.5	36.04	*B*.2	34.82	*A*.10	38.34	*B*.2	33.78	*A*.5	38.58	*B*.24	CTFollowsRequester

and the time that users miss a specific functionality or bug fix that could have already been integrated in the project.

Another threat to the validity of the results hereby presented is the fact that they were achieved through the use of open-source projects. Thus, it is not possible to guarantee that these results are also valid for industrial projects. Nonetheless, the projects selected to be included in the experiments feature a large number of pull requests (over 1,000) and are developed in some of the most popular languages amongst projects hosted by GitHub.

6. Related Work

In software engineering, predicting the time needed to perform coding tasks has become an important subject of research. Some studies in the literature already explore attempts to predict the duration of tasks, such as bug fix [29, 30, 31]; issue resolution [32, 33], pull request integration [1, 3], and continuous integration [34].

The pull-requests differ from bugs and issues, as pull requests already contain the source code needed to fix a problem or add a new functionality. Pull requests provide information both from changes control and version control, which include: request description, requester, commits, number of lines added and removed, and files included, modified, and deleted. Naturally, data from different contexts may be used to aid in pull request lifetime prediction.

The integration of a pull request may become a slow task for different reasons: extensive code, important functionality changes, and lack of knowledge by the reviewer about the files altered by the pull request [9]. In this scenario, predicting the time need to carry out pull request review may help in the process of triage, allowing for, for instance, the allocation of the available time of developers with more expertise to tackle high priority pull requests. Two recent studies available in the literature have already approached pull request lifetime prediction [1, 3], discussed in the following.

Gousios *et al.* [1] evaluated classification algorithms with 141,468 pull requests from 291 open-source projects in order to predict pull request lifetime, which was discretized into three classes ($C_1 \leq 1$ hour $< C_2 \leq 1$ day $< C_3$). Considering the explored projects, in average, a pull request takes 3.7 days to be closed. In the experiments, the authors used 12 predictive attributes and a cross-validation method achieving accuracies of 38% with the Naive Bayes algorithm and 59% with the Random Forest algorithm. Upon evaluating the relevance of the attributes for the Random Forest algorithm, in a data set of 83,442 pull requests, the authors stated that developer expertise, project size, existence of test files, and the percentage of commits submitted via pull requests in the project seem to play an important role in the quality of the pull request lifetime prediction. Moreover, pull request submitted by core team developers take precedence over the ones submitted by external developers. Finally, they suggest that the time a pull request takes to be accepted is inversely proportional to the requester's merge rate, that is, the higher the requester's merge percentage is, the lower will be the time taken for his or her pull requests to be integrated [1].

Yu *et al.* [3] explored the influence of three sets of attributes in the latency of pull requests in GitHub projects. In 103,284 pull requests extracted from a sample of 40 open-source projects which employ a continuous integration service[4], the authors used the multiple linear regression technique to model pull request review latency. Three linear models were built: the first one solely using the attributes used by Gousios *et al.* [1] and Tsay *et al.* [35]; the second one adding attributes associated with pull request review and integration processes; and the third one adding attributes about the use of continuous integration in

[4]When a pull request is received by a project that practices continuous integration, the code is integrated in a test branch, and tests are executed. If the contribution fails the tests, an integrator can reject the pull request or demand improvements in the code, informing the requester what is incorrect. If tests are successful, an integrator can review the code and decide about its integration [3]. One example of a continuous integration service is TravisCI (`https://travis-ci.org/`).

the process.

The third model, which features the largest number of attributes, including new and previously used attributes by former works, obtained the best determination coefficient ($R^2 = 0.587$). From the presented models, the authors stated that: (i) pull request latency is a complex issue, which demands the understanding of several independent variables to be appropriately explained; (ii) the practice of continuous integration is a strong positive predictor; (iii) the number of comments is the best predictive attribute; and (iv) pull request size (the number of added lines and the number of commits) is a relevant attribute to determine the latency of pull requests [3].

In this work, we used 97,603 pull requests collected from 20 open-source projects by means of the GHTorrent tool. Focusing on estimating pull request lifetime, we made use of classification and regression algorithms to evaluate two sets of attributes: the first one composed by the attributes proposed by Yu *et al.* [3], which presented themselves to be better than the ones used by Gousios *et al.* [1] and Tsay *et al.* [35] to model pull request latency; and the second one comprised of all the attributes from the first set plus a new subset we believe to be useful in the task of pull request lifetime prediction.

In comparison with the approach proposed by Yu *et al.* [3], following the unification of the experimental process and the project corpus in order to perform a fair comparison, our approach presented a significant reduction of the mean error rate. Regarding the experiments using discretized lifetime values, our approach featured the best normalized improvements, achieving the best accuracy in 18 out of the 20 projects considered. The average accuracy of our approach was of 45.28% for predicting pull request lifetime, which meant an average normalized improvement of 14.68% in relation to the majority class (the highest percentage of pull requests in the same time interval).

7. Conclusions

This work proposed an approach for predicting the pull request lifetime. Our approach was evaluated against other approaches, which represent the related work, over 97,603 pull requests collected from 30 open-source projects hosted at GitHub. All in all, our work considered two different sets of predictive attributes, represented by Set A and Set B, four regression algorithms (Linear Regression, M5P, Random Forest, and SMOReg) and five classification algorithms (IBk, J48, Naive Bayes, Random Forest, and SMO).

Regarding the regression techniques employed, even though we were able to reduce the mean error rate when contrasted with Yu *et al.* [3], the estimates had not achieved satisfactory values. For this reason, we turned to exploring classification algorithms to predict discrete values representing time intervals.

Three approaches were considered for the classification problem. Approach A is represented by the evaluation of the Random Forest algorithm with attributes from Set A proposed by Yu *et al.* [3]. Approach B corresponds to the attributes in Set B, proposed in this work, and the Random Forest algorithm. Approach C,

also proposed in this work, represents an evaluation of the Random Forest algorithm using the 33 most relevant attributes from Set B, in accordance with the attribute selection strategy based on the Information Gain metric.

In comparison with the related work [3], Approach C featured the best average normalized improvement, achieving the best accuracy in 18 out of the 20 projects evaluated. The average accuracy was of 45.28% to predict pull request lifetime, with an average normalized improvement of 14.68% in relation to the majority class. It is noteworthy that Approach C uses Set B, containing attributes that were introduced by this work.

Among the projects that displayed the best normalized improvements, we observed that some attributes proposed in Set B proved to be relevant in determining the pull request lifetime, such as: NumAddedFiles (B.5), NumTestChurn (B.12), WatcherProject (B.16) and the ones characterizing previous interactions of project developers (B.17 through B.22).

Acknowledgment

We would like to thank FAPERJ (grant E-26/201.523/2014), CNPq (grant 308369/2015-7 and 306137/2017-8), and CAPES (grant 2326/2011) for the financial support.

References

[1] G. Gousios, M. Pinzger, A. V. Deursen, An Exploratory Study of the Pull-based Software Development Model, in: Proceedings of the 36th International Conference on Software Engineering (ICSE), ACM, New York, NY, USA, 2014, pp. 345–355 (2014). doi:10.1145/2568225.2568260.

[2] G. Gousios, A. Zaidman, M.-A. Storey, A. v. Deursen, Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), Vol. 1, 2015, pp. 358–368 (May 2015). doi:10.1109/ICSE.2015.55.

[3] Y. Yu, H. Wang, V. Filkov, P. Devanbu, B. Vasilescu, Wait for It: Determinants of Pull Request Evaluation Latency on GitHub, in: Proceedings of the 12th Working Conference on Mining Software Repositories (MSR), IEEE, Florence, Italy, 2015, pp. 367–371 (May 2015). doi:10.1109/MSR.2015.42.

[4] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, P. Devanbu, Cohesive and Isolated Development with Branches, in: Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE), Springer-Verlag, Tallinn, Estonia, 2012, pp. 316–331 (2012). doi:10.1007/978-3-642-28872-2_22.

[5] S. Chacon, B. Straub, Pro Git, 2nd Edition, Apress, Berkely, CA, USA, 2014 (2014).

[6] BitBucket, Pull Requests (2014).
URL https://www.atlassian.com/git/tutorials/making-a-pull-request/how-it-works

[7] GitHub, Using pull requests - User Documentation (2014).
URL https://help.github.com/articles/using-pull-requests/

[8] M. Fowler, Continuous Integration (2017).
URL https://martinfowler.com/articles/continuousIntegration.html

[9] G. Gousios, The GHTorent Dataset and Tool Suite, in: Proceedings of the 10th Working Conference on Mining Software Repositories (MSR), IEEE, San Francisco, USA, 2013, pp. 233–236 (2013).

[10] I. H. Witten, E. Frank, M. A. Hall, Data Mining: Practical Machine Learning Tools and Techniques, 3rd Edition, The Morgan Kaufmann Series in Data Management Systems, Morgan Kaufmann, Boston, 2011 (2011).

[11] L. Breiman, Random Forests, Machine Learning 45 (1) (2001) 5–32 (Oct. 2001). doi:10.1023/A:1010933404324.

[12] S. K. Shevade, S. S. Keerthi, C. Bhattacharyya, K. R. K. Murthy, Improvements to The SMO Algorithm for SVM Regression, IEEE Transactions on Neural Networks 11 (5) (2000) 1188–1193 (Sep. 2000). doi:10.1109/72.870050.

[13] D. W. Aha, D. Kibler, M. K. Albert, Instance-Based Learning Algorithms, Machine Learning 6 (1) (1991) 37–66 (Jan. 1991). doi:10.1023/A:1022689900470.

[14] J. R. Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993 (1993).

[15] G. H. John, P. Langley, Estimating Continuous Distributions in Bayesian Classifiers, in: Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI), Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995, pp. 338–345 (1995).

[16] J. C. Platt, Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines, Tech. rep., Microsoft Research, Technical Report MSR-TR-98-14 (1998).

[17] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings, IEEE Transactions on Software Engineering 34 (4) (2008) 485–496 (Jul. 2008). doi:10.1109/TSE.2008.35.

[18] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I. H. Witten, The WEKA Data Mining Software: An Update, SIGKDD Explorations 11 (1) (2009) 10–18 (Nov. 2009). doi:10.1145/1656274.1656278.

[19] M. A. Hall, G. Holmes, Benchmarking Attribute Selection Techniques for Discrete Class Data Mining, IEEE Transactions on Knowledge and Data Engineering 15 (6) (2003) 1437–1447 (Nov. 2003). doi:10.1109/TKDE.2003.1245283.

[20] M. A. Hall, Correlation-based Feature Selection for Machine Learning, Ph.D. thesis, University of Waikato, Hamilton, New Zealand (1999).

[21] I. Kononenko, Estimating Attributes: Analysis and Extensions of RELIEF, in: Proceedings of the 7th European Conference on Machine Learning (ECML), Springer-Verlag, Catania, Italy, 1994, pp. 171–182 (Apr. 1994). doi:10.1007/3-540-57868-4_57.

[22] J. Han, M. Kamber, J. Pei, Data Mining: Concepts and Techniques, 3rd Edition, The Morgan Kaufmann Series in Data Management Systems, Morgan Kaufmann, Boston, 2011 (2011).

[23] M. L. de Lima Júnior, D. M. Soares, A. Plastino, L. Murta, Automatic assignment of integrators to pull requests: The importance of selecting appropriate attributes, Journal of Systems and Software 144 (2018) 181 – 196 (2018). doi:https://doi.org/10.1016/j.jss.2018.05.065.

[24] G. L. Pappa, A. A. Freitas, Automatically Evolving Rule Induction Algorithms, in: European Conference on Machine Learning (ECML), Springer Berlin Heidelberg, 2006, pp. 341–352 (Sep. 2006). doi:10.1007/11871842_34.

[25] J. Demšar, Statistical Comparisons of Classifiers over Multiple Data Sets, The Journal of Machine Learning Research 7 (2006) 1–30 (Dec. 2006).

[26] M. Friedman, A Comparison of Alternative Tests of Significance for the Problem of m Rankings, The Annals of Mathematical Statistics 11 (1) (1940) 86–92 (Mar. 1940). doi:10.1214/aoms/1177731944.

[27] P. Nemenyi, Distribution-free Multiple Comparisons, Ph.D. thesis, Princeton University (1963).

[28] T. Pohlert, The Pairwise Multiple Comparison of Mean Ranks Package (PMCMR), r package (2014).
URL https://CRAN.R-project.org/package=PMCMR

[29] L. Panjer, Predicting Eclipse Bug Lifetimes, in: Proceedings on the 4th International Workshop on Mining Software Repositories (MSR), IEEE, Minneapolis, Minnesota, USA, 2007, pp. 29–32 (2007). doi:10.1109/MSR.2007.25.

[30] L. Marks, Y. Zou, A. E. Hassan, Studying the Fix-time for Bugs in Large Open Source Projects, in: Proceedings of the 7th International Conference on Predictive Models in Software Engineering (PROMISE), ACM, Banff, AB, Canada, 2011, pp. 11:1–11:8 (2011). doi:10.1145/2020390.2020401.

[31] P. Bhattacharya, I. Neamtiu, Bug-fix Time Prediction Models: Can We Do Better?, in: Proceedings of the 8th Working Conference on Mining Software Repositories, ACM, Waikiki, Honolulu, HI, USA, 2011, pp. 207–210 (2011). doi:10.1145/1985441.1985472.

[32] M. Rees-Jones, M. Martin, T. Menzies, Better predictors for issue lifetime, arXiv preprint arXiv:1702.07735 (2017).

[33] R. Kikas, M. Dumas, D. Pfahl, Using dynamic and contextual features to predict issue lifetime in github projects, in: Proceedings of the 13th International Conference on Mining Software Repositories, ACM, 2016, pp. 291–302 (2016).

[34] E. Bisong, E. Tran, O. Baysal, Built to last or built too fast?: evaluating prediction models for build times, in: Proceedings of the 14th International Conference on Mining Software Repositories, IEEE Press, 2017, pp. 487–490 (2017).

[35] J. Tsay, L. Dabbish, J. Herbsleb, Influence of Social and Technical Factors for Evaluating Contribution in GitHub, in: Proceedings of the 36th International Conference on Software Engineering (ICSE), ACM, New York, NY, USA, 2014, pp. 356–366 (2014). doi:10.1145/2568225.2568315.