EMSE Major Revision Round 2

Response to Editor

We want to thank all reviewers (especially Reviewer 4) for their careful and insightful comments. Typos are fixed and readability is improved with careful proof-reading. According to reviewers' comments, we have revised the draft (shown as R1, R2 and R4, respectively).

"We will make an accept/reject decision based on the next revision. "

We appreciate the second opportunity to revise this paper. We also understand the above point since it does not make sense to have papers stuck in endless revision cycles. However, since

- We have seen a new reviewer (Reviewer 4) this round; and
- That reviewer has added several new added requirements not seen in the prior reviewers' requests

we want to ask, would it be possible to have more chances for the revisions? We ask this since, while we are trying our best to respond to those new comments, it would be very appreciated if more opportunities are granted to improve the current manuscript.

Response to Reviewer #1

Round2-R1-A: *I would prefer the author reply regarding task bias (regarding parameters from literature) to have a footprint in the manuscript.*

Thanks for the comments. We have added a footnote to explain the parameters using in Model Construction section. (please see R1A on page 8)

Round2-R1-B: *It could benefit from minor copy-edits which I list below; however, most are suggestions rather than corrections.*

Thanks for the suggestions, we have edited and polished the corresponding words and fixed typos as well.

Response to Reviewer #2

Round2-R2-A: *I am not convinced by the answer given to comment R1c. Since the authors indicated to have used FLASH for parameter tuning in the previous version of the manuscript, why not reporting the results? Or why not directly report that FLASH does not always work as believed and written by the same authors of this manuscript in prior studies?*

Thanks for the comments. Initially we were trying to prove that hyperparameter tuning can improve the prediction performance in our experiments. We tested a few optimizers and selected DE in the first manuscript. With your suggestion, we have added method tuned by FLASH in our benchmarks, updated the experiments results and reported that FLASH does not always work the best in the experiments. (please see R2A1 on page 11) (please see R2A2 on page 21)

Round2-R2-B: *GSCART (grid search) is much slower than DECART. This is true only if using a different number of trials/samples. Keeping the same number of trial configurations N, DECART requires by definition more running time due to the complexity of the evolutionary operator. Instead, grid search tries N configuration with no additional operators. How do DECART and GSCAR when using the same number of trials/samples?*

We agree with you on this that we made an unfair comparison between GSCART and DECART in the previous manuscript. To fix this issue, we have set the budget of GSCART (grid search) the same number of trials as other optimizers (e.g. DECART), re-run the experiments on whole dataset and updated the results. (please see R2B on page 11)

Round2-R2-C: *The authors used the non-parametric variant of the Scott-Knott statistical test introduced by Mittas et al. in TSE'13. However, as pointed out by H. Steffen, the correction made by Mittas et al. to this test "does not necessarily lead to the fulfillment of the assumptions of the original Scott-Knott test and may cause problems with the statistical analysis." Long story short, Scott-Knott is not a correct statistical test for non-normally distributed data. I strongly recommend using either the ANOVA (with permutation) or the Friedman tests instead. Suppose the authors insist on using the Scott-Knott statistical test. In that case, they MUST provide theoretical evidence that the concerns raised by H. Steffen do not hold in this case.*

Thanks for the suggestions. We have replaced the Scott-Knott statistical test by Friedman test with Nemenyi Post-Hoc test. In this new manuscript, for each project, we first use Friedman test to determine if there are statistical significant differences between the performance of predictors. If yes, we then use Nemenyi post-hoc test to determine which predictor(s) are statistically better. (please see R2C on page 16)

Round2-R2-D: *I'm afraid I have to disagree with the argument "Many feature selection mechanisms are somewhat arcane and, hence, make it hard to explaining*

to business users." This argument in the response letter is problematic in two ways: Arcane does not imply bad. If we go for that route, one can argue that the Turing machine is "bad" because it is 70+ years old. Of course, that is not the case. Many existing feature selection methods are straightforward to explain and apply. See, for example, unsupervised approaches based on simple correlation (to remove collinearity) My recommendation to use well-established feature selection methods remains unless the authors provide better arguments than those used in the current response letter.

Thanks for the comments. We agree that it will be more convincing if we could use well-established methods for feature importance analysis. In this round, we did some literature review and found that tree-based method is one way that widely used. To fix the problem, we applied the tree-based method from scikit-learn and evaluate the Gini importance of the features. The Gini importance algorithm computes impurity-based feature importances (the importance of a feature is computed as the normalized total reduction of the criterion brought by the feature). (please see R2D on page 19)

Round2-R2-E: *Based on the survey results, the "stargazers" indicator is marked as "irrelevant" by the majority of the participants (52%). Why then keeping this indicator? This indicator should be removed to be consistent with the results of the developers' interviews.*

Thanks for the suggestions. We have removed the stargazers prediction in the new experiments. (please see R2E on page 6)

Round2-R2-F: *The text added to answer the comment Rx2f should be revisited: lines 23-35 describe another work by the authors of this manuscript that is entirely unrelated. Unnecessary self-citations should be avoided: there is no need to sell unrelated work. On the other hand, the section does not provide a more compelling view of hyper-parameter optimization done in closely related fields. For example, evolutionary algorithms have been used in defect prediction or effort estimation to fine-tune the parameters of the same classifiers applied in this work (e.g., CART or J48 parameter tuning).*

Thanks for the comments. We have removed the unrelated text and added more recent works of hyperparameter optimization in defect prediction, systems configuration and effort estimation. (please see R2F on page 8)

Round2-R2-G: *The arguments justifying the usage of DE are unsatisfactory and mathematically incorrect (lines 24-38 on page 9). This paragraph is problematic for many reasons. Reason 1: DE is one of the many evolutionary algorithms in the literature. It is not the most recent nor the best performing in all contexts (as this manuscript wants us to believe). DE has indeed been shown to perform well on specific optimization problems. But it has also been shown to underperform on*

other optimization problems (e.g., non-linearly separable problem). I highly recommend the authors to look at Section VIII of the following survey: [2] S. Das and P. N. Suganthan, "Differential Evolution: A Survey of the State-of-the-Art," in IEEE Transactions on Evolutionary Computation, vol. 15, no. 1, pp. 4-31, Feb. 2011, DOI: 10.1109/TEVC.2010.2059031. Reason 2: I assume the authors are well familiar with the no-free lunch theorem. Hence, no algorithm is superior to all other algorithms on all optimization problems. This applies to DE and but also to different approaches (e.g., FLASH) the authors have proposed lately. "We found that DE works just as well as anything else, ran much faster [...]." Compared to what? In absolute terms? For all problems? This statement is mathematically incorrect (see the no-free lunch theorem) but also empirically inaccurate as the authors did not compare DE to "anything else" in this work. My final question, therefore, is: why not comparing DECART to other simpler evolutionary algorithms? If the authors insist on proposing DE as the best out there, they have to show that this is the case. The only argument the authors can use is that the DE has shown good performance regarding grid search and random search. Analysis of other evolutionary algorithms (probably even better than DE) can be part of future work. Finally, the authors should not claim that DE "is the best" but simply better than the two baselines used in the study.

Thanks for pointing out the inappropriate description of DE in our last draft, we have revised the paragraph to describe the power of DE objectively. In this new draft, we want to indicate that DE gets good performance compared to other methods in our experiment, and we would explore other evolutionary algorithms for further improvement in the future work. (please see R2G1 on page 3) (please see R2G2 on page 9) (please see R2G3 on page 12) (please see R2G4 on page 21) (please see R2G5 on page 25)

Round2-R2-H: *I highly appreciate that the authors improved Table six addressing the comments Rx1f1 Rx2h. However, the table might still include one word that should not be excluded: "CV." If CV stands for "cross-validation," then it should not be included in the blocklist.*

Thanks for the suggestion. We have removed "CV" from the blocklist. (please see R2H on page 13)

Round2-R2-I: *The authors should enrich the analysis of the results: In how many cases/projects are the results of DECART statistically better than those produced by the baselines? Are the percentages reported in Section 4.3 based on the number of statistically significant cases? Or based on whether the median/mean is larger?*

Thanks for the comments. With the new experiment and statistical methods (Friedman test and Nemenyi test), we have updated our experiment results and performance evaluations. The percentages reported in Section 4.3 are based on the method's numbers of "Rank=1" appears in all cases. "Rank=1" means that the related method is in statistically the best performance group (the group may have multiple methods if they don't have statistically significant difference). (please see R2I on page 20)

Response to Reviewer #4

Thank you for your detailed suggestions. For simplicity's sake, we have divided our replies to you into two parts.

Firstly, we list all the changes we made resulting from your specific individual comments.

Next, in Round2-R4-I, we offer a combined analysis of several of your comments which, to our reading, were aspects on one composite meta-issue.

Round2-R4-A: *Table 1 shows a mapping between the proposed features and Linux Kernel Practices, but how this mapping was determined was unclear to me... As far as I can tell, a mapping that maps everything to everything would be equally valid.*

You are quite correct. We realize that the added value of this table is not so much. With your suggestion, we have removed it from this new manuscript and added related content to the introduction (please see R4A on page 2).

Round2-R4-B: *The survey questions posed to the open source developers should be reported. In particular, it is unclear whether participants were given the task of rating the given properties or a more free-form survey where they were able to propose their own properties. The former would presumably introduce some bias ("choose the best from this menu of options").*

Thanks for the comments. We have added the details of our survey questions. (please see R4B1 on page 6) Regarding to the bias on "choose the best from given", we did give participants space to talk about their own opinions on additional health indicators if they have (Please see Question 4 in the survey). We also have added discussions about the bias in the section of Threats to validity, (please see R4B2 on page 24) and future work about exploring more health indicators based on developers' opinions. (please see R4B3 on page 25)

Round2-R4-C: *(My main concern with the evaluation is that) the authors are modeling and learning time-series data, but only evaluating at a few chosen points in time. A standard approach to evaluating these kinds of models in the machine learning literature is to apply a moving window, where models are trained on e.g. 6 months of data evaluated for the next month, then moved one month forward and evaluated again, and so on. This gives a much better picture of the reliability of the models and predictions at various points in time. This would also remove the need for repositories to be filtered by specific dates.*

Thanks for the comments. We have re-designed the model training process and re-run the experiments. We took your suggestions and applied a "sliding-window-style" way to build the models training and testing pipeline. (please see R4C on page 14)

Round2-R4-D: *In addition to the summary results across all considered projects, a case study would be insightful. An error of 10% is still relatively high, in particular if the numbers are used for project planning – 110 pull requests require more work and personnel than 100. The Linux kernel could be considered (I assume that it was included) to show how some of the prediction scenarios the authors outline at the beginning of the paper would play out (e.g. for personnel planning).*

Thanks for the suggestions. We have added a real-world use case to show how our methods can be used (predicting number of contributors next month for a set of open source OS projects). (please see R4D on page 18)

Round2-R4-E: *In the construction of the models, why is only 80% of available data selected for training?*

Thanks for the comments. In previous manuscript, we used 80% of data for training to test the reliability and performance variability of the models. In this new manuscript, since the model training process and performance evaluation methods have been changed, we have removed this rule when training the model. (please see R4E1 on page 14) (please see R4E2 on page 16)

Round2-R4-F: *The analysis of feature importance in section 4.2 should take the position of a feature split in the learned tree into account – a feature that is used high up in the tree is much more important than a feature far down that only helps to discriminate between a small number of data points.*

Thanks for the comments. With your permission, we wish to follow the spirit of your suggestion and replace the previous analysis method with something more standard. To that end, we applied the tree-based method from scikit-learn and evaluate the Gini importance of the features. The Gini importance algorithm computes impurity-based feature importances (the importance of a feature is computed as the normalized total reduction of the criterion brought by the feature). (please see R4F on page 19)

Round2-R4-G: *The criteria the authors used for choosing repositories to include in the study may also introduce unintended bias. For example, inactive repos are excluded even though they might provide a good historical snapshot of how a project failed – these are, by some definition, unhealthy projects, and could provide exemplars for the machine learning approaches to learn from.*

Thanks for the comments. You are quite correct, in the previous draft, we did not explicitly state the criteria by which we selected repositories.

In selecting repositories, we relied on advice from the literature. The criteria for repository selection has been extensively debated in the literature. We fear that if we adopt anything non-standard then that could be acceptable to you (for example) but would raise concerns for other reviewers. Hence we followed the advice on selection criteria from Kalliamvakou et al. and Munaiah et al. [**? ?**], use active repos for the current study. (please see R4G1 on page 12) We agree that inactive repositories may

provide additional historical information to help identify health issues. The current bias of repo collections have been added and discussed in Threats to Validity. (please see R4G2 on page 24) For the future work, after re-collecting massive data that also includes inactive repositories, we can move forward to see the influence of inactive repos. (please see R4G3 on page 25)

Round2-R4-H: *The comparison of the different machine learning approaches is unfair as the authors tune only one method. It is well-known that the relative performance of untuned methods is not predictive of the performance of tuned methods – a method that looks unpromising when not tuned could turn out to be the best when tuned. Further, the authors develop their own tuning approach instead of using one of the many tried-and-tested approaches from the literature (as the authors use scikit-learn, I recommend auto-sklearn, which is a drop-in replacement for any scikit-learn approach that automatically chooses the best machine learning algorithm and hyper-parameters). I hypothesize that a random forest, tuned properly, would achieve much better performance than the authors' approach.*

You raise an important point– we did not say why in the prior draft that we used DE and not other optimizers. Our main point is that hyper-parameter optimization (HPO) has not been applied previously to project health studies. Here, we show that such optimization significantly reduces the estimation error.

This is not to say that other optimizers might not do better– but HPO is a very active field of research. There will be many new optimizers proposed in the near future. In the future work, we will continue to explore more machine learning approaches, with different tuning techniques. (please see R4H1 on page 25)

But even if future work out-perform our current results, that would still does not violate the main conclusion of this paper; i.e. HPO is useful for predicting project health indicators. (please see R4H2 on page 3)

Just as a side note, like the reviewer, we have been advocated more use of hyperparameter optimization in SE for many years now (those results have been published in TSE [**?** **?**], ICSE [**?**], MSR [**?**], ASE [**?**], IST [**?**], and other venues [**?**]). Lamentably, despite all our work, hyperparameter optimization is not widely applied. Therefore we think that there is benefit in taking a "simplicity-first approach" to HPO as a tactic to encourage more use of this promising HPO technology. Such simplicity enables more refactoring and integration with other tools.

Round2-R4-I: *Several of your comments address our motivation and choice of indicators. Your list some other possible indicators, including some that are composites of more primitive indicators and some that change over time. We hope that the following response addresses the core issues of your concern in this regard.*

Your ideas for exploring temporal issues are very insightful and we have applied "sliding-window" for better model learning (please see R4C on page 14).

As to your comments on our selection of non-temporal indicators, we fully acknowledge your point that there are many questions that could be asked about this

selection. And, as you quite correctly pointed out, the previous draft was in error since it didn't explain why we explore certain indicators and not others.

To address this problem that your raise, we have added this note to the introduction (please see R4I1 on page 2). In summary:

- We explore these indicators since, initially, half a dozen of our industrial partners asked us to.
- We use our survey (please see R4B1 on page 6) to check if real-world software developers care about those indicators– which lead to the current set.

That said, your general point remains– that there are other indicators that might also be of interest, depending on your business goals. We have taken your list of suggested other indicators and added them to the future work section. (please see R4I2 on page 25) As to this current study, in our view, the question might not be "what is the best set of indictors" but rather, "given that the business users care about indicators A,B,C... (and that set will change over time and change from site to site), what is good technology for building predictors for those kinds of indicators?" On that matter, we refer you to a comment made in that introduction:

> "We make no claim that this is a complete set of project health indicators– after all, currently there is no unique and consolidated definition of project health [**?**]. However, what we can show is that we have prediction methods that work well, for thousands of projects, for all the indicators shown above. Hence we predict (but cannot absolutely prove) that when new indicators arose, our methods will be useful." (please see R4I3 on page 2)

Noname manuscript No.
(will be inserted by the editor)

Predicting Health Indicators for Open Source Projects (using the DECART Hyperparameter Optimizer)

Tianpei Xia · Wei Fu · Rui Shu · Rishabh Agrawal · Tim Menzies

Received: date / Accepted: date

Abstract Software developed on public platform is a source of data that can be used to make predictions about those projects. While the individual developing activity may be random and hard to predict, the developing behavior on project level can be predicted with good accuracy when large groups of developers work together on software projects.

To demonstrate this, we use 64,181 months of data from 1,159 GitHub projects to make various predictions about the current status of those projects (as of April 2020). We find that traditional estimation algorithms make many mistakes. Algorithms like *k*-nearest neighbors (KNN), support vector regression (SVR), random forest (RFT), linear regression (LNR), and regression trees (CART) have high error rates. But that error rate can be greatly reduced using the DECART hyperparameter optimization. DECART is a differential evolution (DE) algorithm that tunes the CART data mining system to the particular details of a specific project.

To the best of our knowledge, this is the largest study yet conducted, using recent data for predicting multiple health indicators of open-source projects.

Keywords Hyperparameter Optimization · Project Health · Machine Learning

Tianpei Xia, Wei Fu, Rui Shu, Rishabh Agrawal, Tim Menzies
Department of Computer Science, North Carolina State University, Raleigh, NC, USA
Email: txia4@ncsu.edu, fuwei.ee@gmail.com, rshu@ncsu.edu, agrawa3@ncsu.edu, timm@ieee.org

1 Introduction

In 2020, open-source projects dominate the software development environment [**? ? ? ? ?**]. Over 80% of the software in any technology product or service are now open-source [**?**]. With so many projects now being open-source, a natural next question is "which of these projects are any good?" or "which should I avoid?". In other words, we now need to assess the *health condition* of open-source projects before using them. Specifically, software engineering managers need *project health indicators* that assess the health of a project at some future point in time.

To assess project health, we look at project activity. Han et al. note that popular open-source projects tend to be more active [**?**]. Also, many other researchers agree that healthy open-source projects need to be "vigorous" [**? ? ? ? ? ? ?**]. In this paper, we use 64,181 months of data from GitHub to make predictions for the April 2020 activity within 1,159 GitHub projects. Specifically:

1. The number of contributors who will work on the project;
2. The number of commits that project will receive;
3. The number of opened pull-requests in that project;
4. The number of closed pull-requests in that project;
5. The number of opened issues in that project;
6. The number of closed issues in that project;
7. Project popularity trends (number of GitHub "stars").

R4A **R4I1** We explore these aspects since, after several months of weekly teleconferences with Linux developers, these were issues that many of them were concerned with. That said, **R4I3** we make no claim that this is a complete set of project health indicators– after all, currently there is no unique and consolidated definition of project health [**?**]. However, what we can show is that we have prediction methods that work well, for thousands of projects, for all the indicators shown above. Hence we predict (but cannot absolutely prove) that when new indicators arose, our methods will be useful.

A second result we offer is that, based on user studies with domain experts from those projects, we can assert that most of the above indicators are of active interest to the practitioner community. To be sure, some are seen to be of lesser importance (e.g. 52% of our experts didn't care about "stars") but others are deemed to be very important (e.g. "close issues" was described as "very important" or "somewhat important" by 93% of our experts). For the practical purposes, the domain experts consider the abnormal values of these indicators show issues that deserve a manager's attention (and we note that, in the literature, other researchers argue that such abnormal values offer useful guidance for directing manager intervention [**?**]).

A third result is that we demonstrate that it is possible to accurately predict health indicators for 1, 3, 6, and 12 months into the future. To the best of our knowledge, no such prediction has been shown possible for thousands of projects.

As to "how" we achieve these results, what we show below is that, using a special kind of learning, for predicting the future value of our indicators, we can achieve **low predictions error rates** (usually under 25%). We conjecture that our error rates are so low since we use **arguably better technology** than prior work. Most of the

prior work neglects to tune the control parameters of their learners. This is not ideal since some recent research in SE reports that such tuning can significantly improve the performance of models used in software analytics [**? ? ? ? ? ? ? ? ?**]. Here, we use a technology called "differential evolution" (DE, explained below) to automatically tune our learners. **R2G1** **R4H2** While future work might discover better optimizers (for health indicator prediction) than DE, what we can say about that algorithm in this paper is that it can find models that make better predictions than many other approaches (that are used widely in the literature).

This paper makes extensive use of recent data to predict the values of multiple health indicators of open-source projects. Looking at prior work that studied multiple health indicators, two closely comparable studies to this paper are *Healthy or not: A way to predict ecosystem health in GitHub* by Liao et al. [**?**] and *A Large Scale Study of Long-Time Contributor Prediction for GitHub Projects* by Bao et al. [**?**]. These papers studied project developing activities on hundreds of projects. On the basis of their work, we would like to take a further step, to explore more. In our study, we explore 1,159 projects, much of our data is current (as of April 2020) while some prior works use project data that is years to decades old [**?**].

One unique aspect of this work, compared to prior work, is that we try to explore more kinds of indicators than those seen in prior papers. For example, the goal of the Bao et al. paper [**?**] is to predict if a programmer will become a long term contributor to a GitHub project. While this is certainly an important question, it is all about *individuals* within a single project. The goal of our paper is to offer management advice at a *project level*.

Overall, the contribution of this paper is:

- Our user surveys show that the project health issues explored here relate to real-world user concerns for real projects (see our survey results §**??**).
- We show the viability of large scale project health analysis, for multiple indicators (prior work usually explores just one [**?**] or two [**?**] indicators; to the best of our knowledge, this is largest survey of projects yet explored for project health).
- This is the first paper to demonstrate that project health can be predicted into the future. Specially, when we compare the errors in our project health estimators 1,3,6 and 12 months into the future we find that (a) the error for predicting one month into the future is very low[1] and (b) that error does not increase much as we look further into the future[2].
- We show the viability of hyperparameter optimization across a very large number of projects. Prior to this work, the experience was that such optimization can be very slow and hence might not scale to a large population of projects. However, what was realized here is that the optimization problem for 1600 projects is 1600 tiny problems (13 columns by 60 rows, one for each month we sampled a project). Hence there is no need here for elaborate hyperparameter optimization[3]

[1] From Table **??**; median magnitude of relative error =25%.

[2] From Table **??**; that error increases, on average by less than a quarter to 31%

[3] e.g. not the 100 generations over 100 individuals recommended by Holland et a. [**?**]

Holland, John H. “Genetic Algorithms.” Scientific American, vol. 267, no. 1, 1992, pp. 66–73. JSTOR, http://www.jstor.org/stable/24939139. Accessed 10 Dec. 2021.

The rest of this paper is organized as follows: Section 2 introduces the current problems of open-source software development, the background of software project health, the related work on software analytics of open-source projects, and the difference between our work and prior studies. After that, Section 3 describes the researc:wh questions, explain our open-source project data mining, model constructions and the experiment setup details. Section 4 presents the experimental results and answers the research questions. This is followed by Section 5 and Section 6, which discuss the findings from the experiment and the potential threats in the study. Finally, the conclusions and future works are given in Section 7.

For the open source resources of this work, please see it at the Github repo[4].

2 Background and Related Work

In this section, we show the background and motivation of our study, current related works from other researchers, and the methods to use in our experiment.

2.1 Why Study Project Health Indicators?

There are many business scenarios within which the predictions of this paper would be very useful. This section discusses those scenarios.

We study open source software since it is becoming more prominent in the overall software engineering landscape. As said in our introduction, over 80% of the software in any technology product or service is now open-source [**?**].

As the open source community matures, so too does its management practices. Increasingly, open source projects are becoming more structured with organizing foundations. For example, as the largest software organizations, the Apache Software Foundation and Linux Foundation currently host 371 projects and 166 projects respectively [**? ?**]. For the promising projects, those organizations invest significant funds to secure seats on the board of those projects. Although the stakeholders of these projects have different opinions, they all need indicators of project status to make decisions. For example, project managers need information about upcoming activities in the development to decide where to put resources, and justifications to convince that it is cost effective to pay developers to work on specific tasks.

Some engineers might be eager to attract funding to the foundations that run large open source projects. Large organizations are willing to pay for the privilege of participating in the governance of projects. Hence, it is important to have a good “public profile” for the projects to keep these organizations interested. In the case of GitHub projects, the feature “number of stars” has been recognized as a success and popularity measurement to look at [**? ? ?**].

[4] `https://github.com/anonymousAuthor404/Health_Indicator_Prediction`

Also, many developers must regularly report the status of their projects to the governing foundations. Those reports will determine the allocations of future resources for these projects. For such reports, it is important to assess the projects compared to other similar projects. For example, some program managers argue that their project is scoring "better" than other similar projects since that project has more new contributors per month [**?**]. In our study, the feature "number of contributors" is a suitable measure for this way of scoring.

We also note that many commercial companies use open-source packages in the products they sell to customers. For that purpose, commercial companies want to use relatively stable packages (i.e. no massive abnormal developing activities) for some time to come. Otherwise, if the open-source community stops maintaining or changes those packages, then those companies will be forced into maintaining open-source packages which they may not fully understand.

Another case where commercial organizations can use predictions of project health indicators is the issue of *ecosystem package management*. For example, Red Hat are very interested in project health indicators that can be automatically applied to tens of thousands of projects. When they release a new version of their OS, more than 24,000 software packages included in the distribution are delivered to tens of millions of machines around the world. For this process, Red Hat seek project health indicators to help them:

- Decide what packages should not be included in the next distribution (due to abnormal behaviors in the development);
- Detect, then repair, falling health in popular packages. For example, in 2019, Red Hat's engineers noted that a particularly popular project was falling out of favor with other developers since its regression test suite was not keeping up with current changes. With just a few thousand dollars, Red Hat used crowd sourced programmers to generate the tests that made the package viable again [**?**].

Yet another use case where predictions of project health indicators would be useful is *software staff management*. Thousands of IBM developers maintain dozens of large open-source toolkits. IBM needs to know the expected workload within those projects, several months in advance [**?**]. Some indicators can advise when there are too many developers working on one project, and not enough working on another. Using this information, IBM management can "juggle" that staff around multiple projects in order to match the expected workload to the available staff. For example,

- If a spike is expected in a few months for the number of pull requests, management might move extra staff over to that project a couple of months earlier (so that staff can learn that codebase).
- When handling the training of newcomers, it is unwise to drop novices into some high stress scenarios where too few programmers are struggling to handle a large workload.
- It is also useful to know when the workload for a project is predicted to be stable or decreasing. In that use case, it is not ill-advised to move staff to other problems in order to accommodate the requests of seasoned programmers who want to either learn new technologies as part of their career development; or resolve personnel conflict issues.

Table 1: Importance of 7 Indicators to Project Health (based on the survey).

	Very Important	Somewhat Important	Not Important
contributors	33%	46%	21%
commits	57%	31%	12%
opened pull-requests	31%	56%	13%
closed pull-requests	46%	34%	20%
opened issues	33%	59%	8%
closed issues	40%	53%	7%
stargazers	12%	36%	52%

In our study, we focus on the 7 critical and easy to access GitHub features listed in Section 1 as our potential health indicators.

R4B1 To verify these features actually matter in real-world business-level cases, we have conducted extensive interviews with real-world open source software developers. From January to March, 2021, we selected 100 open source projects from our data collection list, and sent email surveys to their main developers to ask about their opinions on health indicators based on their development experience. We made the survey questions short and easy to reply to in order to get more responses and higher engagements. The transcript of our survey questions is shown in Figure 1. In the end, 112 core contributors from 68 projects provide valid responses. Table 1 shows the summary of whether our indicators matter to project health in their opinions.

Question 1
Are you a core contributor to this project?
(Yes / No)

Question 2
Did you notice your project was having a health problem in the previous development?
(Yes / No)

Question 3
Please rate the importance of the following features regarding project health:
(1:Very Important
2:Somewhat Important
3:Not Important)

 # contributors, # commits, # opened pull−requests, # closed pull−requests, # opened issues, # closed issues, # stargazers

Question 4
Besides the features mentioned in last question, what else developing features do you think are relevant to open source project health?

Fig. 1: Transcript of survey questions.

R2E Based on Table 1, we can say features like "contributors", "commits", "opened pull-request", "closed pull-request", "opened issues" and "closed issues" are all important as health indicators in the survey since very few developers treat them "Not Important"(mostly less than 20%). The only exception is "stargazers", which the majority of the participants (52%) consider as "Not Important". Hence, we conclude the first 6 features have great impacts on project health based on experienced software engineers' judgements and could be used as our health indicators in the experiment.

2.2 How to Study Project Health Indicators?

The results of our interviews and surveys give us hints that some activities in open source development can be treated as indicators of project health. In our literature

review, we find numerous studies and organizations are exploring the health or development features of open-source projects. For example:

- Jansen et al. [**?**] introduce an OSEHO (Open Source Ecosystem Health Operationalization) framework, using productivity, robustness and niche creation to measure the health of software ecosystem.
- Manikas et al. [**?**] propose a logical framework for defining and measuring the software ecosystem health consisting of the health of three main components (actors, software and orchestration).
- A community named "CHAOSS" (Community Health Analytics for Open Source Software) [**?**] contributes on developing metrics, methodologies, and software from a wide range of open-source projects to express open-source project health and sustainability.
- Weber et al. [**?**] use a random forest classifier to predict project popularity (which they define as the star velocity in their study) on a set of Python projects.
- Borges et al. [**?**] claim that the number of stars of a repository is a direct measure of its popularity, in their study, they use a model with multiple linear regressions to predict the number of stars to estimate the popularity of GitHub repositories.
- Wang et al. [**?**] propose a prediction model using regression analysis to find potential long-term contributors (through their capacity, willingness, and the opportunity to contribute at the time of joining). They validate their methods on "Ruby on Rails", on a large and popular project on GitHub. Bao et al. [**?**] use a set of methods (Naive Bayes, SVR, Decision Tree, KNN and Random Forest) on 917 projects from GHTorrent to predict long term contributors (which they determine as the time interval between their first and last commit in the project is larger than a threshold.), they create a benchmark for the result and find random forest achieves the best performance.
- Kikas et al. [**?**] build random forest models to predict the issue close time on GitHub projects, with multiple static, dynamic and contextual features. They report that the dynamic and contextual features are critical in such prediction tasks. Jarczyk et al. [**?**] use generalized linear models for prediction of issue closure rate. Based on multiple features (stars, commits, issues closed by team, etc.), they find that larger teams with more project members have lower issue closure rates than smaller teams, while increased work centralization improves issue closure rates.
- Other developing related feature predictions also include the information of commits, which is used by Qi et al. [**?**] in their software effort estimation research of open source projects, where they treat the number of commits as an indicator of human effort.
- Chen et al. [**?**] use linear regression models on 1,000 GitHub projects to predict the number of forks, they conclude this prediction could help GitHub to recommend popular projects, and guide developers to find projects which are likely to succeed and worthy of their contribution.

We explore the literature looking for how prior researchers have explored software developing activities. Starting with venues listed at Google Scholar Metrics "software systems", we searched for highly cited or very recent papers discussing

software analytics, project health, open source systems and *GitHub predicting*. We found:

- In the past six years (2014 to 2020), there were at least 30 related papers.
- 10 of those papers looked at least one of the seven features we listed in our introduction [**?** **?** **?** **?** **?** **?** **?** **?** **?** **?**].
- 3 of those papers explored multiple features [**?** **?** **?**].

Following the previous research, we consider these features as project health indicators, make a massive, systematic time-series data collection, and try to find a general method to predict the trends of these indicators.

As to the technology used in the related papers, the preferred predicting method was usually just one of the following:

- LNR: *linear regression* model that builds regression methods to fit the data to a parametric equation;
- CART: *decision tree learner* for classification and regression;
- RFT: *random forest* that builds multiple regression trees, then report the average conclusion across that forest;
- KNN: *k-nearest neighbors* that make conclusions by average across nearby examples;
- SVR: *support vector regression* uses the regressions that take the quadratic optimizer used in support vector machines and use it to learn a parametric equation that predicts for a numeric class.

Hence, for this study, we use the above learners as baseline methods with implementations from Scikit-Learn [**?**]. Unless being adjusted by hyperparameter optimizers (discussed below), all these learners run with the default settings.[5]

Of the above related work, a study by Bao et al. from TSE'19 seems close to our work [**?**]. They explored multiple learning methods for their prediction tasks. Further, while the other papers used learners with their off-the-shelf settings, Bao et al. took care to tune the control hyperparameters of their learners.

The idea of tuning the control hyperparameters to improve the prediction performance has been applied to many machine learning algorithms. For example, Bergstra et al. used random search (i.e. just pick parameters at random) and greedy sequential methods on finding the best configurations of their neural networks and deep belief networks models [**?**]. Snoek et al. applied Bayesian optimization to reduce the prediction errors of logistic regression and support vector machines [**?**].

R2F Much recent research in SE report that such hyperparameter tuning can significantly improve the performance of prediction methods used in many software analytic tasks [**?** **?** **?** **?** **?** **?**]. For example, for defect prediction, Fu et al. applied an evolutionary algorithm named Differential Evolution (DE) [**?**] on a set of tree-based defect predictors (e.g. CART). With the data from open source JAVA systems, their experiment results showed that hyperparameter tuning largely improves the precision

[5] We use default settings for the baselines to find if they can provide good prediction performance, and how much space hyperparameter-tuning can improve. Using a pre-selected parameter-settings from literature may bring bias because of different data format or prediction tasks. **R1A**

of these predictors [**?**]. Agrawal et al. also used Differential Evolution to automatically tune hyperparameters of a processor named "SMOTE". Based on the results of seven datasets, their experiments showed that Differential Evolution can lead to up to 60% improvements in predictive performance when tuning SMOTE's hyperparameters [**?**]. Tantithamthavorn et al. investigated the impact of hyperparameter tuning on a case study with 18 datasets and 26 learning models. They tuned multiple hyperparameters using a grid search (looping over all possible parameter settings) on 100 repetitions of the out-of-sample bootstrap procedure. With that approach, they found results led to improvements of up to 40% in the Area Under the receiver operator characteristic Curve [**?**].

In software system configurations, Nair et al. proposed FLASH, a sequential model-based optimizer, which explored the configuration space by reflecting on the configurations evaluated so far to find the best configuration for the systems, they evaluated FLASH on 7 software systems and demonstrated that FLASH can effectively find the best configuration [**?**]. Later, Xia et al. applied FLASH to tune the hyperparameters of CART in their study of software effort estimation, using data from 1,161 waterfall projects and 120 contemporary projects, the results showed that this sequential model-based optimization achieved better performance than previous state-of-the-art methods [**?**]. Also, in effort estimation, Minku et al. proposed an online supervised hyperparameter tuning procedure, which helps to tune the number of clusters in Dycom (Dynamic Cross-company Mapped Model Learning), a software effort estimation online ensemble learning approach. Using the ISBSG Repository, they showed that the proposed method was generally successful in enabling a very simple threshold-based clustering approach to obtain the most competitive Dycom results [**?**]. In this study, we do not use methods of Minku et al. since, at least so far, our work has been on within-company estimation (i.e. learning from the history of some *current project*, then applying what was learned to later points in that same project).

Recently, Tantithamthavorn et al. [**?**] extended their defect prediction study on more hyperparameter optimizers with genetic algorithms, random search, and Differential Evolution. They found that in the defect prediction domain, different hyperparameter tuning procedures led to similar benefits in terms of performance improvement; i.e. at least in that domain, it may not be necessary to perform extensive studies of across different hyperparameter optimizers. [**?**].

R2G2 For this paper, echoing the methods of the advice of Tantithamthavorn et al., we explore hyperparameter optimization using Grid Search, Random Search, FLASH and Differential Evolution. In our experiments, we will show Differential Evolution (DE) gets better results than other methods. In this regard, we offer the verification as other researchers who have found DE to be useful for tuning software analytic problems (e.g., defect prediction [**? ?**]). Also, since DE has shown good performance comparing other baselines like grid search and random search, further exploration of other evolutionary algorithms as hyperparameter optimizers may bring even better performance (this could be part of the future work).

The pseudocode of DE algorithm is shown in Figure 2. The premise of that code is that the best way to mutate the existing tunings is to extrapolate between current solutions (stored in the *frontier* list). Three solutions x, y, z are selected at random

from the *frontier*. For each tuning parameter j, at some probability cf (crossover probability), DE replaces the old tuning x_j with *new* where $new_j = x_j + f \times (y_j - z_j)$ where f (differential weight) is a parameter controlling differential weight.

The main loop of DE runs over the *frontier* of size np (population size), replacing old items with new candidates (if new candidate is better). This means that, as the loop progresses, the *frontier* contains increasingly more valuable solutions (which, in turn, helps extrapolation since the next time we pick x, y, z, we get better candidates.).

DE's loops keep repeating till it runs out of *lives*. The number of *lives* is decremented for each loop (and incremented every time we find a better solution).

Our initial experiments showed that out of all these "off-the-shelf" learners, the CART regression tree learner was performing best. Hence, we combine CART with differential evolution to create the DECART hyperparameter optimizer for CART regression trees. The choice of these parameters can have a large impact on optimization performance. Taking advice from Storn and Fu et al. [**?** **?**], we set DE's configuration parameters to $\{np, cf, f, lives\} = \{20, 0.75, 0.3, 10\}$. The CART hyperparameters we control via DE are shown in Table 2. For the tuning ranges of each hyperparameters, we follow the suggestions from Fu et al. [**?**].

3 Experiment Setup

In this section, we ask the related research questions, provide details of our experiment data, explain how to construct our experiment model, and how to measure our experiment results.

```
1   def DE(np=20, cf=0.75, f=0.3, lives=10): # default settings
2     frontier = # make "np" number of random guesses
3     best = frontier.1 # any value at all
4     while(lives-- > 0):
5       tmp = empty
6       for i = 1 to |frontier|: # size of frontier
7           old = frontier_i
8           x,y,z = any three from frontier, picked at random
9           new= copy(old)
10          for j = 1 to |new|: # for all attributes
11           if rand() < cf # at probability cf...
12                new.j = x.j + f * (z.j - y.j) # ...change item j
13          # end for
14          new = new if better(new,old) else old
15          tmp_i = new
16          if better(new,best) then
17              best = new
18              lives++ # enable one more generation
19          end
20      # end for
21      frontier = tmp
22      lives--
23    # end while
24    return best
```

Fig. 2: Differential evolution. Pseudocode based on Storn's algorithm [**?**].

Table 2: The hyperparameters to be tuned in CART.

Hyperparameter	Default	Tuning Range	Description
max_feature	None	[0.01, 1]	Number of features to consider when looking for the best split
max_depth	None	[1, 12]	The maximum depth of the decision tree
min_sample_leaf	1	[1, 12]	Minimum samples required to be at a leaf node
min_sample_split	2	[0, 20]	Minimum samples required to split internal nodes

3.1 Research Questions

The experiments of this paper are designed to explore the following questions.

RQ1: How to find the trends of project health indicators? R2A1 R2B We apply five popular machine learning algorithms (i.e., KNN, SVR, LNR, RFT and CART), and four hyperparameter-optimized methods: CART tuned by random search (RDCART), grid search (GSCART), FLASH and differential evolution (DECART) with the same trial budget[6] to 1,159 open-source projects collected from GitHub (For FLASH, we apply the same settings as used in the previous work [**?**]). For each project, once we collect N months of data, we make predictions for the recent status using a part of data from month 1 to month $N-j$ (for $j \in \{1,3,6,12\}$) in the past. DECART's median error in those experiments is under 25% (where this error is calculated from $error = 100 * |p-a|/a$ using the predicted p and actual a). Hence, we will say:

> **Answer 1**: Many project health indicators can be predicted, with good accuracy, for 1, 3, 6, 12 months into the future.

RQ2: What features matter the most in prediction? To find the most important features that have been used for prediction, we look into the internal structure of model with the best prediction and compute impurity-based feature importances (Gini importance). We will show that:

> **Answer 2**: In our study, "monthly_commits", "monthly_openPR", "monthly_openISSUE" and "monthly_closeISSUE" are the most important features, while "monthly_closePR" is the least used feature for all six health indicators' predictions.

RQ3: Which methods achieve the best prediction performance? We compare the experimental results of each method on all 1,159 open-source projects and prediction for 1, 3, 6, and 12 months into the future. After a statistical comparison between different learners, we find that:

[6] i.e. A maximum of 200 evaluations for Random Search, Grid Search, Flash and DE.

Answer 3: Hyperparameter optimization with DECART generates better prediction performance than the other methods studied here in 76% of our 1,159 projects.

R2G3 Just to say the obvious, we only show here that DECART performs best compared to the methods we took from prior work on project health (see Table **??**). What that is paper has not done is survey all possible methods (since that study would be much larger than any single paper). That said, we say Answer 3 is a significant result since it should serve to inspire much future work on finding yet better methods for optimizing health indicators,

3.2 Data Collection

R4G1 Many repositories on GitHub are not suitable for software engineering research [**? ?**]. We follow advice from Kalliamvakou et al. and Munaiah et al., apply related criteria (with GitHub GraphQL API) to find useful URLs of the projects [**? ?**]. As shown in Table 3, we select public, not archived, and not mirrored repositories as open sources, use a set of thresholds to ensure they are active and collaborative, with relatively popular profiles (Shrikanth et al. report that popular projects usually have stars around 1k to 20k [**?**]). In addition, to remove repositories with irrelevant topics such as "books", "class projects" or "tutorial docs", etc., we create a dictionary of "suspicious words of irrelevancy", and remove URLs that contain words in that dictionary (see Table 4). After applying the criteria of Table 3, Table 4 and a round of manual checking, we get 1,159 repositories which we treat as engineered software projects. From these repositories, we extract features of in total 64,181 monthly data across all projects.

At the point of writing, there is no unique and consolidated definition of software project health [**? ? ?**]. However, many researchers agree that healthy open-source projects need to be "vigorous" and "active" [**? ? ? ? ? ?**]. Based on our previous survey, we select 6 features as health indicators of open-source project on GitHub: number of commits, contributors, open pull-requests, closed pull-requests, open issues and closed issues. These features are important GitHub features to indicate the activities of the projects [**? ? ?**].

All the features collected from each project in this study are listed in Table 5. These features are carefully selected because some of them are used by other researchers who explore related GitHub studies [**? ? ?**].

To get the latest and accurate features of our selected repositories, we use the GitHub APIs for feature collection. For each project, the first commit date is used as the starting date of the project. Then all the features are collected and calculated monthly from that date up to the present date. For example, the first commit of the *kotlin-native* project was on May 16, 2016. After, we collected features from May, 2016 to April, 2020. Due to the GitHub API rate limit, we could not get some features, like "monthly_commits", which require a large amount of direct API calls. Instead, we clone the repo locally and then extracted features (this technique saved us much

Table 3: Repository selecting criteria.

Filter	Explanation
is:public	select open-source repo
archived:false	exclude archived repo
mirror:false	exclude duplicate repo
stars:1000..20000	select relatively popular repo
size:>=10000	exclude too small repo
forks:>=10	select repo being forked
created:>=2015-01-01	select relatively new repo
created:<=2016-12-31	select repo with enough monthly data
contributor:>=3	exclude personal repo
total_commit:>=1000	select repo with enough commits
total_issue_closed:>=50	select repos with enough issues
total_PR_closed:>=50	select repos with enough pull-request
recent_PR:>=1 (30 days)	exclude inactive repo without PR
recent_commit:>=1 (30 days)	exclude inactive repo without commits

Table 4: Dictionary of "irrelevant" words. We do not use data from projects whose URL includes the following keywords. **R2H**

Suspicious Keywords						
template	web	tutorial	lecture	sample	note	sheet
book	doc	image	video	demo	conf	intro
class	exam	study	material	test	exercise	resource
article	academic	result	output	resume	work	guide
present	slide	101	qa	view	form	course
thesis	collect	pdf	wiki	blog	lesson	pic
paper	camp	summit				

Table 5: Project health indicators. "PR"= pull requests. When predicting feature "X" (e.g. # of commits), we re-arrange the data such the dependent variable is "X" and the independent variables are the rest.

Dimension	Feature	Description	Predict?
Commits	# of commits	monthly number of commits	✓
Pull Requests	# of open PRs	monthly number of open PRs	✓
	# of closed PRs	monthly number of closed PRs	✓
	# of merged PRs	monthly number of merged PRs	
Issues	# of open issues	monthly number of open issues	✓
	# of closed issues	monthly number of closed issues	✓
	# of issue comments	monthly number of issue comments	
Project	# of contributors	monthly number of active contributors	✓
	# of stargazers	monthly increased number of stars	
	# of forks	monthly increased number of forks	

grief with API quotas). Table 6 shows a summary of the data collected by using this method.

Table 6: Summary of 64,181 monthly data across all 1,159 projects.

Feature	Min	Max	Median	IQR
monthly commits	0	10358	45	83
monthly contributors	0	312	6	7
monthly stars	0	6085	32	68
monthly opened PRs	0	3860	6	22
monthly closed PRs	0	14699	1	3
monthly merged PRs	0	1418	4	18
monthly open issues	0	3883	28	53
monthly closed issues	0	20376	24	50
monthly issue comments	0	97846	134	309
monthly forks	0	2789	7	13

3.3 Model Construction

In our experiment, we use five classical machine learning algorithms and four hyperparameter tuned methods for the prediction tasks. These five classical machine learning algorithms are Nearest Neighbors, Support Vector Regression, Linear Regression, Random Forest and Regression Tree (we call them KNN, SVR, LNR, RFT and CART). The configuration of those baselines follows the suggestions from Scikit-Learn.

Beyond the baseline methods, we build a hyperparameter optimized predictor, named "DECART". This method use differential evolution algorithm (with configuration settings to np=20, cf=0.75, f=0.3, lives=10) as an optimizer to optimize four hyperparameters (max_feature, max_depth, min_sample_leaf and min_sample_split) of regression tree (CART), and use this tuned CART to get predict results.

We also build three other hyperparameter optimized predictors named "RDCART", "GSCART" and "FLASH" as baselines of "tuned methods" to compare with DECART. The numbers of trials executed by these methods are identical to DECART.

R4C **R4E1** For each project, we have N monthly data and apply a "sliding-window-style" way to build the models. The methods use training set to construct the model (using goal feature as output and all other features as input), and do the prediction on testing set. We use previous 6 months' data to predict the current month, then make 4 consecutive predictions and return the median. For example, when predicting 1 month into the future, we use data from N-6 to N-1 to predict N, use data from N-7 to N-2 to predict N-1, use data from N-8 to N-3 to predict N-2, and use data from N-9 to N-4 to predict N-3, then we return the median of predictions on N, N-1, N-2 and N-3. In case of untuned predictors (KNN, SVR, LNR, RFT and CART), we use all 6 previous months' data for training; For hyperparameter optimized predictors (RDCART, GSCART, FLASH and DECART), we use the first 5 of previous months' data for training, and 6th month' data for validating to find the best configuration of CART's hyperparameters (the one that achieves the closest prediction value to the actual goal on validating set), then apply this configuration on CART to make prediction on the testing set.

We run each method 20 times to reduce the bias from random operators.

3.4 Performance Metrics

To evaluate the performance of learners, we use two performance metrics to measure the prediction results of our experiments: Magnitude of the Relative Error (MRE) and Standardized Accuracy (SA). We use them since (a) they are advocated in the literature [**?** **?**]; and (b) they both offer a way to compare results against some baseline (and such comparisons with some baselines is considered good practice in empirical AI [**?**]).

Our first evaluation measure metric is the magnitude of the relative error, or MRE. MRE is calculated by expressing absolute residual (AR) as a ratio of actual value, where AR is computed from the difference between predicted and actual values:

$$MRE = \frac{|PREDICT - ACTUAL|}{ACTUAL}$$

For MRE, there is the case when ACTUAL equals "0" and then the metric will have "divide by zero" error. To deal with this issue, when ACTUAL gets "0" in the experiment, we set MRE to "0" if PREDICT is also "0", or a value larger than "1" otherwise.

Sarro et al. [**?**] favors MRE since, they argue that, it is known that the human expert performance for certain SE estimation tasks has a MRE of 30% [**?**]. That is to say, if some estimators achieve less than 30% MRE then it can be said to be competitive with human level performance.

MRE has been criticized because of its bias towards error underestimations [**?** **?** **?** **?** **?** **?**]. Shepperd et al. champion another evaluation measure called "standardized accuracy", or SA [**?**]. SA is computed as the ratio of the observed error against some reasonable fast-but-unsophisticated measurement. That is to say, SA expresses itself as the ratio of some sophisticated estimate divided by a much simpler method. SA [**?** **?**] is based on Mean Absolute Error (MAE), which is defined in terms of

$$MAE = \frac{1}{N}\sum_{i=1}^{n}|PREDICT_i - ACTUAL_i|$$

where N is the number of data used for evaluating the performance. SA uses MAE as follows:

$$SA = (1 - \frac{MAE}{MAE_{guess}}) \times 100$$

where MAE_{guess} is the MAE of a set of guessing values. In our case, we use the median of previous months' values as the guessing values.

We find Shepperd et al.'s arguments for SA to be compelling. But we also agree with Sarro et al. that it is useful to compare estimates against some human-level baselines. Hence, for completeness, we apply both evaluation metrics. As shown below, both evaluation metrics will offer the same conclusion (that DECART's performance is both useful and better than other methods for predicting project health indicators).

Note that in all our results: For MRE, *smaller* values are *better*, and the best possible performance result is "0". For SA, *larger* are *better* , the best possible performance result is "100%".

3.5 Statistics

We report the median (50th percentile) and inter-quartile range (IQR=75th-25th percentile) of our methods' performance across all 1,159 projects.

R2C R4E2 To decide which methods do better than any other, we follow the suggestion by Demšar et al. and Herbold et al., use Friedman test with Nemenyi Post-Hoc test to differentiate the performance of each methods [**? ? ?**].

Friedman test is a non-parametric statistical test which determines whether or not there is a statistically significant difference between three or more populations. [**?**].

Nemenyi Post-Hoc test is used to decide which groups are significantly different from each other [**?**]. In case the Friedman test determines that there are statistically significant differences between the populations, the Nemenyi test uses Critical Distances (CD) between average ranks to define significant different populations. If the distance between two average ranks is greater than the Critical Distances (CD), these two populations will be treated as significantly different.

For each project in our experiments, we get performance populations for each method. We first run Friedman test across these populations, if the corresponding p-value is larger than *0.05*, we consider the difference of the methods are not significant (for this project), and we mark "Rank=1" for all methods.

If the corresponding p-value is less than *0.05*, we then run Nemenyi test to compare the performance populations with each method. We set the threshold of p-value in Nemenyi test to *0.05* and differentiate the methods into different groups, and mark "Rank=1" for the methods in group with the best performance.

4 Results

In this section, we answer the related research questions based on the experiment results.

4.1 How to find the trends of project health indicators? (RQ1)

We predict the value of health indicators for recent months using data from previous months. The median and IQR values of performance results in terms of MRE and SA are shown in Table 7, Table 8, Table 9, and Table 10, respectively.

In all these four tables, we show median and IQR of performance results across 1,159 projects. For MRE, *lower* values are *better*, the dark cells denote better results; For SA, *higher* values are *better*, and dark cells denote worse results.

In these results, we observe that our methods provide very different performance with these 6 health indicators' prediction. In Table 7, we see that some learners have errors over 100% (LNR, predicting for number of commits). For the same task, other learners, however, only have around half of the errors (CART, 57%). Also in that table, when predicting number of commits, the median MRE scores of the untuned learners (KNN, LNR, SVR, RFT, CART) are over 50%. That is, these estimates are often wrong by a factor of two, or more. Further, these tables show that hyperparameter optimization is beneficial. The DECART columns of Table 7 and Table 9 show that

Table 7: MRE median results: one month into the future.

	KNN	LNR	SVR	RFT	CART	RDCART	GSCART	FLASH	DECART
commit	53%	136%	92%	56%	57%	51%	39%	44%	37%
contributor	39%	42%	63%	54%	42%	36%	29%	23%	19%
openPR	41%	44%	66%	42%	30%	25%	21%	23%	23%
closePR	47%	65%	47%	48%	46%	36%	27%	32%	33%
openISSUE	59%	74%	59%	49%	43%	37%	24%	28%	21%
closedISSUE	54%	73%	78%	41%	48%	43%	27%	32%	24%

Table 8: MRE IQR results: one month into the future.

	KNN	LNR	SVR	RFT	CART	RDCART	GSCART	FLASH	DECART
commit	98%	156%	172%	133%	105%	91%	77%	77%	73%
contributor	74%	87%	92%	66%	63%	58%	57%	49%	49%
openPR	88%	111%	105%	81%	87%	81%	66%	75%	67%
closePR	104%	126%	87%	101%	89%	80%	79%	74%	74%
openISSUE	60%	83%	108%	61%	54%	45%	41%	41%	43%
closedISSUE	63%	64%	65%	52%	53%	53%	49%	50%	48%

Table 9: SA median results: one month into the future.

	KNN	LNR	SVR	RFT	CART	RDCART	GSCART	FLASH	DECART
commit	25%	-7%	-26%	29%	33%	37%	45%	41%	46%
contributor	37%	16%	18%	42%	36%	48%	57%	54%	57%
openPR	40%	31%	36%	44%	43%	44%	47%	51%	48%
closePR	19%	-24%	9%	25%	34%	34%	37%	41%	41%
openISSUE	33%	11%	-5%	36%	31%	42%	46%	45%	53%
closedISSUE	35%	14%	29%	40%	40%	45%	49%	48%	48%

Table 10: SA IQR results: one month into the future.

	KNN	LNR	SVR	RFT	CART	RDCART	GSCART	FLASH	DECART
commit	143%	195%	261%	132%	125%	113%	91%	109%	96%
contributor	106%	179%	148%	108%	113%	105%	107%	93%	97%
openPR	132%	117%	131%	120%	125%	121%	97%	104%	92%
closePR	109%	161%	167%	99%	103%	97%	91%	92%	89%
openISSUE	163%	206%	249%	152%	143%	110%	98%	119%	106%
closedISSUE	126%	139%	189%	132%	137%	110%	95%	88%	84%

this method has better median MREs and SAs than the untuned methods. As shown in the last column of Table 7, the median error for DECART is 24%. Additionally, the results of Table 8 and Table 10 also demonstrate the stability of DECART (with the lowest IQR when measuring the performance variability of all methods).

Turning now to other prediction results, our next set of results shows what happens when we make predictions over a 1, 3, 6, 12 months interval. Note that to simulate predicting the status of ahead $1st$, $3rd$, $6th$, $12th$ month, for a project with N months of data, the training sets need to be selected from month 1 to month $N-1$, $N-3$, $N-6$, $N-12$, respectively. That is, to say that the *further* ahead of our predictions, the *earlier* data we have for training. Hence, one thing to watch for is whether or not performance decreases as the training set ages.

Table 11 presents the MRE and SA results of DECART, predicting for 1, 3, 6, and 12 months into the future. By observing the median of ratio-changing (show in gray) from left to right across the table, we see that as we try to predict further and further

Table 11: MRE and SA results with DECART, predicting for 1, 3, 6, 12 months into the future.

	Health Indicator	1 month	3 month	6 month	12 month
Median, MRE	commit	0.37	0.39	0.44	0.51
	contributor	0.19	0.21	0.24	0.29
	openPR	0.23	0.26	0.27	0.37
	closePR	0.33	0.33	0.38	0.46
	openISSUE	0.21	0.21	0.24	0.29
	closedISSUE	0.24	0.26	0.29	0.36
	median ratio change		107%	112%	123%
Median, SA	commit	0.46	0.46	0.44	0.38
	contributor	0.57	0.51	0.46	0.40
	openPR	0.48	0.43	0.38	0.30
	closePR	0.41	0.41	0.33	0.27
	openISSUE	0.53	0.52	0.47	0.40
	closedISSUE	0.48	0.45	0.40	0.30
	median ratio change		96%	90%	83%
IQR, MRE	commit	0.73	0.75	0.87	0.96
	contributor	0.49	0.50	0.59	0.69
	openPR	0.67	0.76	0.91	1.07
	closePR	0.74	0.84	0.93	1.04
	openISSUE	0.43	0.47	0.50	0.55
	closedISSUE	0.48	0.48	0.55	0.61
	median ratio change		106%	115%	112%
IQR, SA	commit	0.96	0.98	1.16	1.38
	contributor	0.97	0.98	1.05	1.25
	openPR	0.92	1.05	1.13	1.27
	closePR	0.89	0.96	1.04	1.27
	openISSUE	1.06	1.14	1.31	1.60
	closedISSUE	0.84	0.87	0.92	1.08
	median ratio change		105%	108%	119%

into the future, (a) MRE degrades around 23% and (b) SA degrades only about 17%, or less. Measured in absolute terms, these changes are still relatively small. In any case, summarizing all the above, we say that:

Answer 1: Many project health indicators can be predicted, with good accuracy, for 1, 3, 6, 12 months into the future. For example, we can predict the number of contributors next month with an error of 19% (MRE).

R4D To give a scenario where this kind of prediction would be useful, we provide a real-world use case with a set of 14 real time OS projects currently supported by the Apache Software Foundation. As shown in Figure 3, one of these operating systems, Zephyr (the one in purple), is exhibiting the steepest growth in the number of its contributors. As to the other projects, most of these might be viewed as stagnant, perhaps even at risk of cancellation[7]. Hence, to the managers of the stagnant projects in Figure 3 (those with mostly flat curves), they are particularly interested in "catching

[7]In the Apache Software Foundation, projects can be canceled and "moved to the attic" (https://attic.apache.org) when they are unable to muster 3 votes for a release, lack of active contributors, or unable to fulfill their reporting duties to the Foundation.

up with Zephyr". Note that this requires increasing their number of contributors to the "flat" projects by 200% (for RIOT) to 3,000% (LiteOS). For that purpose, predictions with a mere 19% error (next month) would be useful to foretell improvements to the software.

4.2 What features matter the most in prediction? (RQ2)

In our experimental data, we have 10 numeric features for prediction. We use them since they are features with high importance, suggested by prior work (see Section 3.2). That said, having done all these experiments, it makes sense to ask which features, in practice, would be more useful when we predict health indicators. This information could help us to focus on useful features and remove irrelevancies when enlarging our research in future work. To work that out, we look into the trees generated by DECART (our best learners) in the above experiments. For each tree, we find impurity-based feature importances, which is computed as the normalized total reduction of the criterion brought by the feature (also known as the Gini importance). **R2D** **R4F**

For each predicting target, we calculate the mean scores for each feature, of all generated trees, the results are summarized in Table 12. In this table, "n/a" denotes the dependent variable which is not counted in the experiment. From this table, first of all, we find that some features are highly related to specific health indicators. For example, "commit", "openISSUE" and "star" have got scores 23%, 21% and 27% when we built trees to predict "contributor" indicator for 1,159 repositories. Secondly, some features are bellwethers that have been used as features for multiple indicator predictions, like "openPR" gets 16%, 23%, 19%, and 26% when predicting

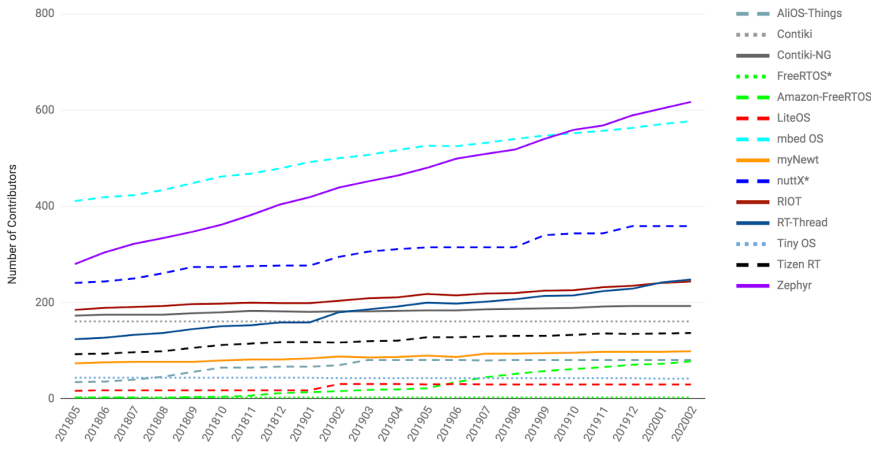

Fig. 3: The trends in number of contributors across 14 real-time OS projects since 2018. Zephyr (one in purple) has out-paced other similar software projects. (data source: The Apache Software Foundation)

"commit", "closePR", "openISSUE" and "closeISSUE". Thirdly, some features even though they belong to the similar type, like "openISSUE" and "closeISSUE", they are not highly related in the predictions. In our experiment, we find that "openISSUE" only gets 8%, way less than "ISSUEcomment" (27%), "openPR" (26%) and "closePR" (13%) when predicting "closeISSUE". Last but not least, some features are less used than others. According to our experiment, "closePR" is the least used feature for all predictions (the mean score of "closePR" is only 5%).

Answer 2: In our study, "monthly_commits", "monthly_openPR", "monthly_openISSUE" and "monthly_closeISSUE" are the most important features, while "monthly_closePR" is the least used feature for all six health indicators' predictions.

Note that none of these features should be abandoned. For feature "closePR", the least used feature in prediction, when predicting "closeISSUE", this feature still gets 13% score of these cases.

That said, it would be hard pressed to say that Table 12 indicates that only a small subset of the Table 5 features are outstandingly most important. While Table 12 suggests that some feature pruning might be useful, overall we would suggest that using all of these features might be the best practice in most cases.

4.3 Which methods achieve the best prediction performance? (RQ3)

To answer this question, we compared the experimental results of each method on all 1,159 open-source projects predicting for 1, 3, 6, and 12 months into the future.

R2I Across 1,159 projects, we report the "win rate", which are the percentages of one learner belongs to the group with the best predictions performance (Rank=1). To get "the group with the best prediction performance", for each project, we use the Friedman test with Nemenyi Post-Hoc test introduced in Section 3.5 to compare different learners in terms of MRE and SA, then differentiate the learners into different groups, group with lower MRE (or larger SA) is the best group. One note is, there

Target	commit	contributor	openPR	closePR	openISSUE	closeISSUE	star	ISSUEcomment	mergedPR	fork
commit	n/a	5%	16%	3%	7%	25%	4%	21%	5%	14%
contributor	23%	n/a	5%	4%	21%	3%	27%	5%	7%	5%
openPR	9%	7%	n/a	4%	22%	30%	2%	5%	18%	3%
closePR	15%	11%	23%	n/a	8%	24%	7%	5%	4%	3%
openISSUE	2%	3%	19%	2%	n/a	17%	5%	4%	25%	23%
closedISSUE	10%	3%	26%	13%	8%	n/a	3%	27%	6%	4%
mean	12%	6%	18%	5%	13%	20%	8%	11%	11%	9%

Table 12: The mean scores of Gini importance in trees generated by DECART (observed percentages in 1,159 cases).

Table 13: MRE results: the win rate of different learners, measured in terms of MRE.

Predicting Month	Health Indicator	KNN	LNR	SVR	RFT	CART	RDCART	GSCART	FLASH	DECART
1st month	commit	42%	18%	26%	38%	39%	45%	74%	61%	74%
	contributor	60%	31%	43%	46%	41%	44%	67%	56%	77%
	openPR	47%	46%	34%	49%	53%	57%	76%	62%	89%
	closePR	34%	25%	51%	48%	43%	43%	55%	58%	56%
	openISSUE	51%	28%	22%	42%	46%	48%	79%	63%	69%
	closedISSUE	41%	28%	19%	39%	44%	46%	54%	46%	55%
3rd month	commit	36%	27%	33%	45%	48%	45%	63%	61%	76%
	contributor	52%	25%	23%	41%	35%	38%	85%	54%	91%
	openPR	24%	42%	31%	41%	41%	46%	55%	59%	66%
	closePR	34%	25%	42%	36%	56%	50%	75%	64%	70%
	openISSUE	40%	22%	27%	43%	41%	58%	64%	54%	60%
	closedISSUE	36%	27%	25%	44%	42%	43%	92%	73%	85%
6th month	commit	42%	26%	29%	36%	48%	52%	77%	45%	90%
	contributor	46%	33%	22%	33%	51%	60%	57%	52%	68%
	openPR	41%	35%	30%	37%	48%	51%	76%	56%	86%
	closePR	26%	32%	34%	41%	47%	50%	73%	65%	61%
	openISSUE	45%	19%	30%	38%	44%	53%	65%	53%	76%
	closedISSUE	46%	28%	24%	38%	44%	48%	52%	43%	62%
12th month	commit	47%	17%	25%	50%	35%	58%	68%	56%	80%
	contributor	55%	24%	31%	36%	37%	54%	55%	60%	57%
	openPR	39%	28%	29%	52%	53%	55%	83%	69%	90%
	closePR	49%	24%	36%	40%	41%	44%	67%	62%	79%
	openISSUE	48%	27%	29%	53%	45%	55%	51%	67%	80%
	closedISSUE	34%	23%	21%	37%	43%	43%	91%	61%	89%
	median	42%	27%	29%	41%	44%	49%	67%	59%	76%

may be multiple methods belong to the best group (Rank=1). Table 13 and Table 14 show the results of learners' win rate in terms of MRE and SA.

The comparisons in these tables are for intra-row results, where the darker cells indicate the learning methods with higher win rate. For example, in the first row of Table 13 (except the header row), when predicting the number of commits in next month, DECART has the best MRE performance in 74% of all 1,159 cases.

As shown in Table 13, in terms of MRE, DECART achieves the best performance with winning rates from 55% to 91% for all predictions (the median win rate is 76%). Meanwhile, the winning rates of other learners, mostly range from 20% to 60%. **R2A2** For example, FLASH, the hyperparameter-optimized method used in the previous effort estimation study, no longer works the best and its median win rate is only 59%. In general, DECART outperforms other methods on almost all the predictions out of 1,159 projects by 10% $\sim$ 50%.

For SA results, as we see in Table 14, although the median win rate of DECART (69%) decreased a bit compared to MRE (76%), it still outperforms all the rest of methods (closest runner-up, GSCART gets 61%). Specifically, DECAERT wins from 47% to 84% out of 4 different prediction ways on 1,159 projects. Compare to KNN wins from 21% to 53%, LNR wins from 16% to 43%, SVR wins from 19% to 38%, RFT wins from 30% to 56%, CART wins from 33% to 56%, and tuned method RDCART wins from 41% to 57%, GSCART wins from 47% to 81% and FLASH wins from 40% to 75%, respectively. In most cases, the winning rates of the untuned methods are less than 40%. After we take a further look, SVR performs relatively worse, the median winning rate is only 25% compared to that of DECART, 69%.

Based on the results from our experiments, we conclude that:

Answer 3: DECART generates better prediction performance than other methods in 76% of our 1,159 projects (MRE, median).

R2G4 As state above, future work might discover better optimizers (for health indicator prediction) than DE. That said, these **RQ3** results tell us that DE can find models that make better predictions than many other approaches (that are used widely in the literature).

5 Discussion

In this section, we look into the efficiency of our methods, and discuss the potential issues observed from experiment results.

5.1 The efficiency of DECART

DECART is not only effective (as shown in Table 13 and Table 14), but also very fast. In our study, it took around 4.3 hours to run DECART on 1,159 projects (on a dual-core 4.67 GHz desktop); i.e. 13 seconds per datasets. This time includes optimizing CART for each specific dataset, and then making predictions. Note that, for these experiments, we made no use of any special hardware (i.e. we used neither GPUs nor cloud services that interleave multiple cores in some clever manner).

The speed of DECART is an important finding. In our experience, the complexity of hyperparameter optimization is a major concern that limits its widespread use. For example, Fu et al. report that hyperparameter optimization for code defect prediction requires nearly three days of CPU per dataset [**?**]. If all of our 1,000+ datasets required the same amount of CPU resources, then it would be a major blocker to the use of the proposed methods in this paper.

But why is DECART so fast and effective? Firstly, DECART runs fast since it works on very small datasets. This paper studies three to five years of project data. For each month, we extract the 10 features shown in Table 5. That is to say, DECART's optimizations only have to explore datasets up to $10 * 60$ data points per project. Fu et al. on the other hand, worked on more than 100,000 data points.

Secondly, as to why is DECART so effective, we note that many data mining algorithms rely on statistical properties that are emergent in large samples of data [**?**]. Hence they have problems reasoning about datasets with only $10 * 60$ data points. Accordingly, to enable effective data mining, it is important to adjust the learners to the idiosyncrasies of the dataset (via hyperparameter optimization).

5.2 DECART on other time predictions

In our experiment, we observe that when predicting specific health indicators, DECART can achieve 0% error in some cases. Such zero error is a red flag that needs to be investigated since they might be due to overfitting or programming errors (such as use the test value as both the predicted and actual value for the MRE calculation). What we found was that the older the project, the less the programmer activity. Hence, it is hardly surprising that good learners could correctly predict (e.g.) zero closed pull requests.

Table 14: SA results: the win rate of different learners, measured in terms of SA.

Predicting Month	Health Indicator	KNN	LNR	SVR	RFT	CART	RDCART	GSCART	FLASH	DECART
1st month	commit	40%	18%	25%	33%	36%	43%	78%	63%	70%
	contributor	53%	37%	33%	44%	35%	41%	59%	53%	70%
	openPR	46%	24%	32%	42%	50%	51%	73%	56%	68%
	closePR	35%	31%	38%	42%	44%	47%	51%	56%	52%
	openISSUE	45%	25%	19%	37%	42%	42%	78%	74%	69%
	closedISSUE	38%	19%	23%	37%	40%	45%	47%	40%	50%
3rd month	commit	32%	24%	33%	50%	45%	44%	56%	61%	73%
	contributor	50%	23%	21%	36%	41%	45%	79%	47%	67%
	openPR	21%	29%	25%	36%	40%	46%	51%	55%	60%
	closePR	32%	39%	37%	33%	56%	53%	68%	61%	74%
	openISSUE	35%	43%	23%	38%	45%	46%	59%	48%	47%
	closedISSUE	34%	24%	23%	41%	41%	56%	79%	63%	84%
6th month	commit	36%	22%	21%	32%	46%	45%	67%	54%	71%
	contributor	45%	31%	34%	30%	46%	52%	52%	45%	60%
	openPR	40%	41%	28%	37%	47%	46%	69%	53%	83%
	closePR	26%	29%	29%	35%	40%	53%	80%	65%	79%
	openISSUE	41%	19%	22%	36%	40%	52%	57%	50%	68%
	closedISSUE	35%	25%	24%	33%	41%	44%	49%	44%	54%
12th month	commit	44%	16%	19%	48%	33%	57%	58%	55%	69%
	contributor	54%	22%	27%	36%	35%	47%	51%	58%	56%
	openPR	36%	28%	27%	49%	48%	57%	81%	75%	63%
	closePR	48%	25%	31%	37%	37%	40%	59%	57%	67%
	openISSUE	41%	31%	25%	56%	44%	53%	76%	60%	74%
	closedISSUE	33%	30%	22%	35%	38%	42%	62%	56%	75%
	median	39%	25%	25%	37%	41%	46%	61%	56%	69%

But that raises another red flag: suppose *all* our projects had reached some steady state prior to April 2020. In that case, predicting (say) the next month's value of health indicator would be a simple matter of repeating last month's value. In our investigation, we have three reasons for believing that this is not the case. Firstly, prediction in this domain is difficult. If such steady state had been achieved, then all our learners would be reporting very low errors. As seen in Table 7, this is not the case.

Secondly, we looked into the columns in our raw data, looking for long sequences of stable or zero values. This case does not happen in most cases: our data contains many variations across the entire lifecycle of our projects.

Thirdly, just to be sure, we conducted another round of experiments. Instead of predicting for the most recent months, we do the prediction for an earlier period using data collected prior to that time point. Table 15 shows the results. In this table, if a project had (say) $N = 60$ months of data, we went to months $N/2$ and used DECART to predicted 12 months into the future (to $N/2 + 12$). The columns for Table 15 should be compared to the right-hand-side columns of Table 7, Table 8, Table 9, and Table 10. In that comparison, we see that predicting for months in mid period can generate comparable results as predicting for most recent months.

In summary, our results are not unduly biased by predicting just for the recent months. As the evidence, we can still obtain accurate results if we predict for earlier months.

6 Threats to validity

The design of this study may have several validity threats [**?**]. The following issues should be considered to avoid jeopardizing conclusions made from this work.

Parameter Bias: The settings to control the hyperparameters of the prediction methods can have a positive effect on the efficacy of the prediction. By using hyperparameter optimized method in our experiment, we explore the space of possible hyperparameters for the predictor, hence we assert that this study suffers less parameter bias than some other studies.

Survey Bias: R4B2 To verify whether our potential health indicators actually matter, we made a survey to open source project developers to ask about their opinions based on their development experience. While most features were considered to be relevant to the project health in the survey, we would not claim the result was a complete set of project health indicators. In our survey, the choice of features was limited to several options in order to keep it easy to respond. Although the participants could provide additional thoughts in the following question, this would still narrow down their opinions. In future, additional knowledge from participants will be added to reduce this bias.

Metric Bias: We use Magnitude of the Relative Error (MRE) as one of the performance metrics in the experiment. However, MRE is criticized because of its bias towards error underestimations [**? ? ? ? ? ?**]. Specifically, when the benchmark error is small or equal to zero, the relative error could become extremely large or infinite. This may lead to an undefined mean or at least a distortion of the result [**?**]. In our study, we do not abandon MRE since there exist known baselines for human performance in effort estimation expressed in terms of MRE [**?**]. To overcome this limitation, we set a customized MRE treatment to deal with "divide by zero" issue and also apply Standardized Accuracy (SA) as the other measure of the performance.

Sampling Bias: In our study, we collect 64,181 months with 12 features of 1,159 GitHub projects data for the experiment, and use 6 GitHub development features as health indicators of open-source project. While we reach good prediction performance on those data, it would be inappropriate to conclude that our technique always gets positive result on open-source projects, or the health indicators we use could completely decide the project's health status. Another confounding factor is, since the projects we collected have different sizes, domains, life-cycles, etc., they could have different factors regarding the predicting performance of health indicators. R4G2 Also, in the study, we focus on the active project for the data collection. Those excluded inactive repositories might also provide useful data about how projects failed then give more exemplars for model training. To mitigate these problems, we release

	Median MRE	IQR MRE	Median SA	IQR SA
commit	67%	124%	29%	112%
contributor	44%	83%	40%	133%
openPR	38%	77%	37%	106%
closePR	64%	104%	27%	87%
openISSUE	52%	73%	29%	164%
closedISSUE	37%	68%	34%	116%

Table 15: The performance of DECART, staring mid-way through a project, then predicting 12 months into the future.

open source resources of our work to support the research community to reproduce, improve or refute our results on broader data and indicators.

7 Conclusion and Future Work

Our results make a compelling case for open source software projects. Software developed on some public platforms is a source of data that can be used to make accurate predictions about those projects. While the activity of a single developer may be random and hard to predict, when large groups of developers work together on software projects, the resulting behavior can be predicted with good accuracy. For example, after building predictors for six project health indicators, we can make predictions with low error rates (median values usually under 25%).

Our results come with some caveats. The patterns of some activities are harder to be learned, for the law of large numbers. We know this since we cannot constantly get high accuracy in predictions. For example, across our six health indicators, the predicting performances of closePR and commit are not as good as when predicting the number of contributors (as shown in Table 7). Also, to make predictions, we must take care to tune the data mining algorithms to the idiosyncrasies of the datasets. Some data mining algorithms rely on statistical properties that are emergent in large samples of data. Hence, such algorithms may have problems reasoning about very small datasets, such as those studied here. Hence, before making predictions, it is vitally important to adjust the learners to the idiosyncrasies of the dataset via hyperparameter optimization. Unlike prior hyperparameter optimization work by Fu et al. [**?**], our optimization process is very fast (a few seconds per dataset). Accordingly, we assert that for predicting project health indicators, hyperparameter optimization is the preferred technology.

As to future work, there is still much to do. Firstly, we know many organizations such as IBM that run large in-house ecosystems where, behind firewalls, thousands of programmers build software using a private GitHub system. It would be insightful to see if our techniques work for such "private" GitHub networks. R2G5 R4H1 Secondly, our results still have large space to improve. Some prediction tasks are harder than others (e.g. commits, closed PR). In our study, DE has shown good prediction performance comparing to other methods, exploring other evolutionary algorithms [**? ?**] on different learners (e.g. random forest) or applying auto-sklearn [**?**] could be useful and might bring even better results.

Further, as to more indicators, there are more practices from real-world business-level cases to explore. R4B3 In our survey, some of the participants mentioned other features they think are relevant to open source project health, we will assess their opinions to find more indicators. R4I2 Also, it would be worth trying if we can derive more effective sophisticated health indicators, such as:

- The number of new joining/leaving contributors.
- The change in number of developing features (commits, contributors, openPR, etc.) over time.

R4G3 Lastly, an enriched data collection with more features from more types of repositories (e.g. inactive or archived projects) would be helpful for our model learning. With enough training data, we could also explore deep neural network methods (e.g. LSTM) on projects from thousands of repositories, and try to detect anomalies in their developments that may jeopardize the health of these software projects.

Acknowledgements

This work is partially funded by a National Science Foundation Grant #1703487.