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

Predicting Health Indicators for Open Source Projects (using Hyperparameter Optimization)

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 recent 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 hyperparameter optimization.

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 [31, 52, 53, 57]. Over 80% of the software in any technology product or service are now open-source [78]. 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 [28]. Also, many other researchers agree that healthy open-source projects need to be "vigorous" [16, 32, 43, 44, 69, 74]. 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").

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, 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 [43]. 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 [49]).

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 tuning, 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 [2–4, 26, 27, 65, 67, 68]. Here, we explore a set of optimization technologies such as Random Search, Grid Search, Differential Evolution, FLASH and auto-sklearn to automatically tune our learners. While future work might discover better optimization methods, what we can say in this paper is that for health indicator predictions, hyperparameter optimization (HPO) can help models to make better predictions.

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. [42] and *A Large Scale Study of Long-Time Contributor Prediction for GitHub Projects* by Bao et al. [6]. 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 [58].

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 [6] 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*.

1.1 Contributions and Structure of this Paper

The specific contributions of this paper are:

- Our developer surveys show that the project health issues explored here are related to real-world developer concerns for real projects (see Section 2.1), to the best of our knowledge, this is the largest developer survey yet explored for open-source project health.
- We show the viability of large scale project health analysis, for multiple indicators (prior work usually explores just one [1, 28, 36, 56, 71] or two [8, 33] indicators).
- This is the first paper to demonstrate that project health indicators 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 (25% MRE error, see Table 7) and (b) that error does not increase much as we look further into the future (less than 30%, see Table 11).
- 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 [26] and hence might not scale to a large population of projects. However, what was realized here is that the optimization problem for 1,000+ projects is thousand of small problems. Hence there is no need here for elaborate hyperparameter optimization. For example, in our experiments, we find that a very simple optimizer (differential evolution applied to a CART regression tree)

can reach similar prediction performance as a state-of-the-art optimizing system (auto-sklearn [20]), and does so much faster.

More generally, we offer here a somewhat novel perspective on the process of hyperparameter optimization and automatic tuning of learners for software analytics. In our view, it is very important to discuss and reduce the cost of hyperparameter optimization since it may not be a "one-off" cost. Rather, it may have to be repeated many times during a project. Recently we explored an environment where tunings from one application were applied to another. In that work we did the optimizations again for the new application and found that a combination of using (some) old information plus (new) tuning lead to best results [40]. While this paper does not explore tuning transfer, our lesson from that other work is that tuning needs to be specialized whenever the data changes; i.e it should be repeated many times along the life cycle.

But there is a problem: in many domains, tuning is CPU-intensive and needs cloud-based CPU to terminate quickly [26]. But just because that is true in many domains, it does not mean it is true in all domains. The lesson of this paper is that *the tuning of tiny regression trees on very small data sets is a specialized domain with its own properties.* Here, 1,159 times, we run optimizers to better extract a signal from data sets with around a dozen parameters and (at most) 60 rows of data. Hence, as we show below, such optimization can be complete very quickly (in less than 15 seconds). Further, at least for the studies we report here, longer optimization (that run 100 times longer) using more complex optimizers (auto-sklearn), perform no better that our very fast, very simple optimisers[1].

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 research 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 health indicators data used in this work, please see our Github repository[2].

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.

[1] In a recent TSE'21 article. we have explained by SE hyperparameter optimization can be so simple: SE data can be intrinsically simpler than other kinds of data and, hence simpler to explore (see Figure 6d of [5]).

[2] `https://github.com/arennax/Health_Indicator_Prediction`

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 [78].

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 [22, 24]. 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 [9, 10, 28].

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 [75]. 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 [63].

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 [39]. 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.

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

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.

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%

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. [32] 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. [44] 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) [23] 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. [71] 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. [9] 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. [70] 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. [6] 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. [36] 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. [33] 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. [56] 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. [13] 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 [1, 8, 9, 13, 28, 33, 36, 42, 56, 71].
- 3 of those papers explored multiple features [8, 33, 42].

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 [54]. Unless being adjusted by hyperparameter optimizers (discussed below), all these learners run with the default settings.[3]

Of the above related work, a study by Bao et al. from TSE'19 seems close to our work [6]. 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 [7]. Snoek et al. applied Bayesian optimization to reduce the prediction errors of logistic regression and support vector machines [61].

Much recent research in SE report that such hyperparameter tuning can significantly improve the performance of prediction methods used in many software analytic tasks [2–4, 26, 27, 76]. For example, for defect prediction, Fu et al. applied an evolutionary algorithm named Differential Evolution (DE) [64] 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 of these predictors [26]. 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 [3]. 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 [65].

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 [50]. 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 [76]. 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

[3]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.

results [45]. 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. [66] 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. [66].

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. We also add a state-of-the-art HPO technology "auto-sklearn" into our experiments to verify the perofrmance of hyperparameter optimization. "auto-sklearn" is an automated machine learning toolkit and a drop-in replacement for a scikit-learn estimator, it automatically searches for the right learning algorithm for a new machine learning dataset and optimizes its hyperparameters [20]. In our experiments, we will show that methods with hyperparameter optimization get better results than untuned methods. In this regard, we offer the verification as other researchers who have found HPO to be useful for tuning software analytic problems (e.g., defect prediction [26, 27]). Also, since some HPO methods (e.g. DE) has shown better performance comparing other methods like random search, further exploration of other algorithms as hyperparameter optimizers may bring even better performance (this could be part of the future work).

```
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 [64].

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. [26, 64], 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. [26].

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 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.

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? We apply five popular machine learning algorithms (i.e., KNN, SVR, LNR, RFT and CART), and

five hyperparameter-optimized methods: CART tuned by random search (RDCART), grid search (GSCART), FLASH, differential evolution (DECART) and auto-sklearn (ASKL) with the same trial budget[4] to 1,159 open-source projects collected from GitHub (For FLASH, we apply the same settings as used in the previous work [76]). 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. In the experiments, the median prediction errors of health indicators can be reached 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: How to improve the performance of health indicators prediction? 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:

Answer 3: Hyperparameter optimized methods generate better prediction performance than the other methods.

3.2 Data Collection

Many repositories on GitHub are not suitable for software engineering research [35, 48]. 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 [34, 48]. 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 [60]).

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

[4] i.e. A maximum of 200 evaluations for Random Search, Grid Search, Flash and DE; for ASKL, maximum runtime for each project is restricted to 15 seconds, please see Section 5.1 for details.

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

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 [32, 42, 43]. However, many researchers agree that healthy open-source projects need to be "vigorous" and "active" [16, 32, 43, 44, 69, 74]. 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 [1, 10, 28].

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 [14, 28, 77].

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,

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

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	

like “monthly_commits”, which need many direct API calls. Instead, we clone the repo locally and then extracted features (this technique saved us much grief with API quotas). Table 6 shows a summary of the data collected by using this method.

3.3 Model Construction

In our experiment, we use five classical machine learning algorithms and five 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 hyperparameter optimized predictors, named “DECART”, “RDCART”, “GSCART” and “FLASH”. “DECART” uses 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.

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

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

We also add a state-of-the-art HPO system named "auto-sklearn" into our experiments to compare with our hyperparameter optimized predictors. To make a fair comparison, for each project, the runtime budget of "auto-sklearn" is restricted to 15 seconds (which is the upper bound of mean runtime of other hyperparameter optimized predictors), please see Section 5.1 for details.

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 mean of prediction errors. 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 mean of prediction errors 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 [11, 58]; 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 [15]).

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. [58] favors MRE since, they argue that, it is known that the human expert performance for certain SE estimation tasks has a MRE of 30% [46]. 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 [21, 37, 38, 55, 59, 62]. Shepperd et al. champion another evaluation measure called "standardized accuracy", or SA [11]. 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 [11, 41] 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.

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 [18, 29, 30].

Friedman test is a non-parametric statistical test which determines whether or not there is a statistically significant difference between three or more populations. [25].

Nemenyi Post-Hoc test is used to decide which groups are significantly different from each other [51]. 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, each method gets a performance population when predicting an health indicator. We first run Friedman test across these populations, if the corresponding p-value is larger than *0.05*, we consider the difference of

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

	KNN	LNR	SVR	RFT	CART	RDCART	GSCART	FLASH	DECART	ASKL
commit	51%	148%	101%	64%	54%	48%	35%	43%	39%	36%
contributor	38%	38%	72%	61%	38%	36%	33%	21%	21%	23%
openPR	40%	39%	74%	43%	32%	29%	24%	26%	22%	22%
closePR	55%	66%	46%	56%	52%	40%	29%	30%	33%	32%
openISSUE	57%	78%	56%	56%	49%	35%	26%	28%	23%	25%
closedISSUE	60%	70%	89%	46%	50%	41%	28%	31%	28%	25%

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

	KNN	LNR	SVR	RFT	CART	RDCART	GSCART	FLASH	DECART	ASKL
commit	105%	152%	166%	122%	96%	91%	84%	75%	78%	81%
contributor	84%	94%	99%	67%	72%	53%	52%	53%	46%	50%
openPR	87%	104%	123%	77%	84%	95%	68%	83%	76%	73%
closePR	108%	117%	93%	92%	93%	88%	90%	85%	71%	72%
openISSUE	56%	84%	107%	67%	58%	47%	41%	46%	44%	40%
closedISSUE	62%	73%	69%	56%	61%	61%	51%	50%	52%	44%

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

	KNN	LNR	SVR	RFT	CART	RDCART	GSCART	FLASH	DECART	ASKL
commit	22%	-11%	-19%	31%	36%	34%	40%	43%	41%	45%
contributor	38%	18%	20%	45%	32%	44%	54%	52%	59%	54%
openPR	34%	27%	40%	44%	48%	42%	51%	54%	53%	55%
closePR	17%	-15%	9%	25%	31%	33%	34%	43%	39%	43%
openISSUE	28%	12%	3%	38%	29%	39%	49%	42%	51%	46%
closedISSUE	30%	13%	28%	39%	42%	49%	49%	46%	43%	45%

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

	KNN	LNR	SVR	RFT	CART	RDCART	GSCART	FLASH	DECART	ASKL
commit	160%	206%	296%	140%	115%	104%	91%	115%	98%	101%
contributor	102%	174%	155%	111%	106%	98%	94%	98%	89%	90%
openPR	151%	108%	117%	107%	124%	110%	98%	105%	91%	89%
closePR	100%	179%	184%	105%	120%	104%	103%	94%	102%	107%
openISSUE	150%	222%	291%	163%	136%	97%	93%	97%	96%	97%
closedISSUE	147%	160%	186%	142%	143%	116%	92%	86%	89%	94%

the methods are not significant (for this project), and we mark all methods belong to "first group".

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 the methods in group with the best performance as "first group".

4 Results

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

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	0.39	0.40	0.43	0.53
	contributor	0.21	0.21	0.23	0.26
	openPR	0.22	0.22	0.24	0.30
	closePR	0.33	0.33	0.35	0.40
	openISSUE	0.23	0.23	0.26	0.30
	closedISSUE	0.28	0.29	0.30	0.35
	median ratio change		101%	107%	117%
Median, SA	commit	0.41	0.40	0.38	0.30
	contributor	0.59	0.57	0.51	0.40
	openPR	0.53	0.52	0.48	0.38
	closePR	0.39	0.38	0.34	0.28
	openISSUE	0.51	0.50	0.44	0.36
	closedISSUE	0.43	0.43	0.39	0.32
	median ratio change		98%	91%	80%
IQR, MRE	commit	0.78	0.85	0.99	1.11
	contributor	0.46	0.49	0.57	0.66
	openPR	0.76	0.82	0.94	1.07
	closePR	0.71	0.76	0.90	1.02
	openISSUE	0.44	0.47	0.54	0.61
	closedISSUE	0.52	0.56	0.66	0.75
	median ratio change		108%	116%	113%
IQR, SA	commit	0.98	1.07	1.19	1.47
	contributor	0.89	0.97	1.10	1.35
	openPR	0.91	0.99	1.12	1.35
	closePR	1.02	1.12	1.28	1.57
	openISSUE	0.96	1.04	1.14	1.34
	closedISSUE	0.89	0.97	1.08	1.25
	median ratio change		109%	112%	122%

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 better 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, 54%). 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 GSCART, FLASH, DECART and ASKL columns of Table 7 and Table 9 show that these methods have better median MREs and SAs than the untuned methods. For example, as shown in Table 7, the median error for

DECART is usually at least 15% less than the CART. Additionally, the results of Table 8 and Table 10 also demonstrate the stability of HPO methods (with the lower 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 into the future, (a) MRE degrades around 17% and (b) SA degrades only about 20%, 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 21% (MRE).

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,

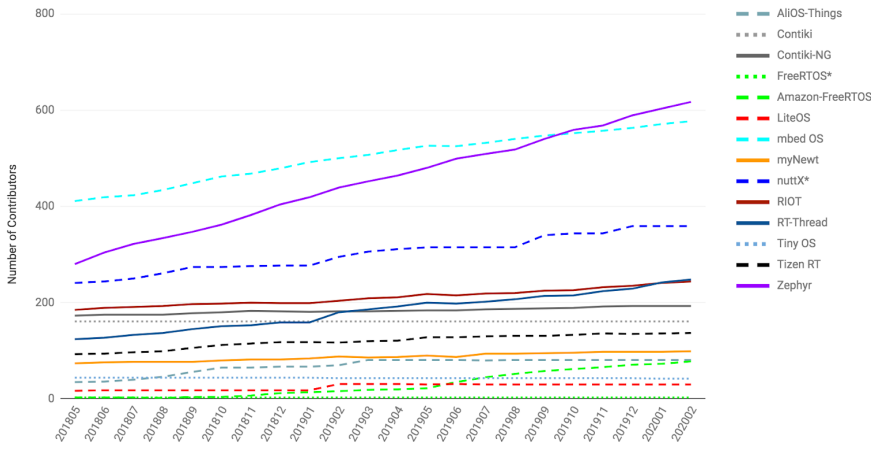

Fig. 3: The trends in number of contributors across 14 real-time OS projects since 2018. Zephyr (one in purple) has out-paced other similar software projects. (data source: The Apache Software Foundation)

Target	commit	contributor	openPR	closePR	openISSUE	closeISSUE	star	ISSUEcomment	mergedPR	fork
commit	n/a	7%	15%	4%	7%	22%	6%	19%	5%	15%
contributor	22%	n/a	4%	4%	20%	8%	23%	5%	8%	6%
openPR	10%	7%	n/a	5%	24%	29%	2%	4%	16%	3%
closePR	13%	12%	24%	n/a	9%	21%	7%	6%	4%	4%
openISSUE	3%	6%	16%	2%	n/a	16%	5%	5%	24%	23%
closedISSUE	9%	7%	24%	13%	8%	n/a	5%	21%	6%	7%
mean	11%	8%	17%	6%	14%	19%	8%	10%	11%	10%

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

perhaps even at risk of cancellation[5]. Hence, to the managers of the stagnant projects in Figure 3 (those with mostly flat curves), they are particularly interested in "catching up with Zephyr". Note that this means increasing their number of contributors to the "flat" projects by 200% (for RIOT) to 3,000% (LiteOS). For that purpose, predictions with a mere 21% 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 (one of 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).

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 22%, 20% and 23% 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 15%, 24%, 16%, and 24% when predicting

[5]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.

"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" (21%), "openPR" (24%) 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 6%).

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 How to improve the performance of health indicators prediction? (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.

Across 1,159 projects, for each health indicator and each of future month (1, 3, 6, 12), we report the "win rate", which are the percentages of one learner belongs to the group with the best prediction performance (i.e. in first group). As introduced in Section 3.5, when predicting one health indicator, for each project, learners get their own performance populations, we use the Friedman test with Nemenyi Post-Hoc test to compare different learners' performance populations in terms of MRE and SA, then differentiate the learners into different groups, group with lower MRE (or larger SA) is the first group. For example, when predicting "number of commits" in "1 month into the future", in terms of MRE, method "KNN" gets into first group in 510 out of 1159 cases, while method "DECART" gets into first group in 939 out of 1159 cases. Hence, their win rates for predicting "number of commits" in "1 month into the future" in terms of MRE are 44% and 81%, respectively.

One note is, there may be multiple methods belong to the first group. 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, KNN has the best MRE performance in 44% of all 1,159 cases, and DECART has the best MRE performance in 81% of all 1,159 cases.

As shown in Table 13, in terms of MRE, ASKL and DECART achieves the best performance with winning rates from 52% to 97% for all predictions (the median

Table 13: Statistical analysis of the MRE results: the win rate (ranked first using the Friedman Nemenyi test of Section 3.5 for the different treatments). Raw performance measured in terms of MRE for predictions *N* months into the future.

Months	Health Indicator	KNN	LNR	SVR	RFT	CART	RDCART	GSCART	FLASH	DECART	ASKL
1st	commit	44%	17%	28%	41%	36%	46%	79%	68%	81%	83%
	contributor	53%	33%	46%	44%	40%	45%	70%	51%	68%	74%
	openPR	41%	44%	32%	51%	50%	62%	69%	57%	90%	89%
	closePR	38%	27%	53%	46%	48%	47%	51%	59%	54%	60%
	openISSUE	52%	31%	22%	37%	44%	50%	81%	61%	69%	72%
	closedISSUE	39%	28%	17%	37%	48%	42%	59%	50%	58%	63%
3rd	commit	33%	29%	30%	47%	43%	50%	59%	61%	83%	78%
	contributor	54%	27%	21%	38%	32%	35%	80%	54%	91%	92%
	openPR	24%	41%	31%	43%	39%	48%	53%	61%	66%	69%
	closePR	37%	24%	43%	39%	54%	54%	80%	66%	67%	78%
	openISSUE	45%	20%	28%	41%	39%	61%	60%	50%	61%	65%
	closedISSUE	34%	24%	23%	43%	38%	40%	94%	70%	84%	76%
6th	commit	39%	27%	29%	36%	48%	58%	74%	48%	90%	84%
	contributor	46%	36%	20%	32%	55%	55%	58%	54%	62%	65%
	openPR	36%	32%	33%	38%	45%	47%	85%	61%	83%	84%
	closePR	26%	32%	33%	42%	47%	53%	75%	61%	54%	70%
	openISSUE	41%	17%	28%	42%	43%	58%	59%	55%	78%	77%
	closedISSUE	48%	28%	25%	40%	48%	49%	52%	47%	56%	52%
12th	commit	43%	17%	25%	46%	34%	56%	67%	51%	73%	79%
	contributor	50%	22%	30%	37%	38%	57%	58%	53%	53%	67%
	openPR	40%	28%	28%	58%	54%	56%	81%	64%	97%	71%
	closePR	50%	26%	40%	40%	39%	49%	74%	61%	85%	89%
	openISSUE	44%	27%	27%	51%	43%	52%	51%	75%	77%	73%
	closedISSUE	38%	21%	23%	39%	44%	43%	95%	66%	94%	88%
	median	41%	27%	28%	41%	44%	50%	69%	60%	75%	75%

win rate is 75%). Meanwhile, the winning rates of other learners, mostly range from 20% to 60%. 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 60%.

For SA results, as we see in Table 14, although the median win rate of hyperparameter tuned methods decreased a bit, they still outperform all the untuned methods. Specifically, ASKL wins from 57% to 82%, and DECAERT wins from 44% to 84% out of 4 different prediction ways on 1,159 projects. Compare to KNN wins from 21% to 57%, LNR wins from 15% to 48%, SVR wins from 19% to 41%, RFT wins from 27% to 58%, CART wins from 29% to 62%, and tuned method RDCART wins from 38% to 62%, GSCART wins from 45% to 84% and FLASH wins from 39% to 78%, respectively. In most cases, the winning rates of the untuned methods are less than 40%. After we take a further look, SVR and LNR performs relatively worse, the median winning rate is only 25% and 26%, respectively.

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

Answer 3: Hyperparameter optimized methods generate better prediction performance than the other methods.

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

Table 14: Statistical analysis of the SA results: the win rate (ranked first using the Friedman Nemenyi test of Section 3.5 for the different treatments). Raw performance measured in terms of SA for predictions *N* months into the future.

Months	Health Indicator	KNN	LNR	SVR	RFT	CART	RDCART	GSCART	FLASH	DECART	ASKL
1st	commit	44%	18%	28%	34%	40%	46%	71%	69%	77%	73%
	contributor	57%	35%	30%	45%	38%	42%	61%	50%	68%	78%
	openPR	47%	22%	28%	43%	47%	48%	69%	59%	70%	69%
	closePR	35%	29%	40%	42%	44%	45%	52%	57%	51%	61%
	openISSUE	48%	25%	19%	41%	42%	39%	71%	78%	77%	76%
	closedISSUE	41%	19%	24%	34%	38%	50%	49%	42%	47%	57%
3rd	commit	29%	22%	34%	55%	43%	44%	55%	68%	65%	71%
	contributor	49%	24%	21%	34%	41%	41%	70%	44%	72%	71%
	openPR	21%	30%	28%	34%	37%	41%	45%	60%	64%	65%
	closePR	28%	40%	41%	36%	62%	51%	65%	66%	78%	81%
	openISSUE	38%	48%	24%	41%	43%	47%	64%	45%	44%	59%
	closedISSUE	34%	24%	21%	43%	41%	57%	74%	65%	81%	66%
6th	commit	34%	24%	23%	32%	50%	43%	63%	53%	70%	74%
	contributor	43%	32%	33%	27%	47%	48%	54%	45%	62%	67%
	openPR	42%	40%	24%	38%	43%	43%	63%	50%	84%	78%
	closePR	25%	30%	27%	36%	41%	55%	80%	65%	71%	76%
	openISSUE	40%	17%	21%	33%	43%	49%	51%	51%	61%	74%
	closedISSUE	38%	27%	24%	33%	39%	43%	51%	39%	51%	63%
12th	commit	39%	15%	21%	45%	29%	62%	61%	53%	72%	68%
	contributor	57%	23%	25%	32%	39%	41%	51%	65%	58%	67%
	openPR	34%	26%	27%	45%	51%	51%	78%	72%	58%	75%
	closePR	44%	23%	35%	40%	37%	38%	58%	61%	66%	71%
	openISSUE	43%	28%	25%	58%	48%	48%	84%	55%	74%	73%
	closedISSUE	32%	31%	23%	36%	35%	46%	55%	62%	84%	82%
	median	39%	26%	25%	37%	42%	46%	62%	58%	69%	71%

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 hyperparameter optimization

In the experiments, our hyperparameter tuned methods are not only effective (as shown in Table 13 and Table 14), but also very fast. Table 15 shows the min, max, mean and median runtime of four hyperparameter tuned methods for a single project. The mean runtime of each project are similar for these methods (12 to 14 seconds). However, methods like FLASH and DECART use more time (in the median case, 27 and 24 seconds, respectively). This time includes optimizing CART for each specific dataset, and then making predictions. Note that, for these experiments, we make no use of any special hardware (i.e. we used neither GPUs nor cloud services that interleave multiple cores in some clever manner).

	RDCART	GSCART	FLASH	DECART
min	6 seconds	6 seconds	4 seconds	7 seconds
max	17 seconds	19 seconds	27 seconds	24 seconds
mean	13 seconds	12 seconds	13 seconds	14 seconds
median	11 seconds	10 seconds	11 seconds	12 seconds

Table 15: Runtimes of hyperparameter tuned methods for a single project.

Table 16: Runtime and performance comparison between ASKL with less restrictive runtimes and DECART, on 20 randomly selected projects.

method	DECART	ASKL
median MRE	0.35	0.34
runtime max	20 seconds	154 seconds
runtime mean	16 seconds	92 seconds
runtime total	324 seconds	1843 seconds

Also, as previously mentioned, for ASKL, we restrict its runtime for each project to 15 seconds (the upper bound of mean runtime of other hyperparameter optimized predictors). Without this restriction, ASKL can benefit from longer runtime to keep finding better solutions. We randomly select 20 from 1,159 projects and test the performance of ASKL without additional runtime restriction (its default time limit is 3,600 seconds) and DECART, and the result is summarized in Table 16. In this test, we find that DECART can reach similar prediction performance as ASKL, with much faster runtime (324 seconds vs 1,843 seconds). In a result that is somewhat troubling for ASKL, that algorithm seems to have occasionally large outlier runs, suggesting that it struggles sometimes to find solutions. Meanwhile, DECART's mean and max runtimes are very similar, suggesting that the performance of this algorithm might be more stable across a wider range of problems.

The efficiency of hyperparameter optimization in health indicators prediction is an important finding. In our experience, the complexity of hyperparameter optimization is a major concern that limits its widespread use in different domains. For example, in a defect prediction study, Fu et al. report that hyperparameter optimization for code defect prediction can use up to nearly three days of CPU per dataset [26]. If our 1,000+ projects required the similar amount of CPU resources, then it would be a major blocker to the use of the proposed methods in this paper.

But why is methods like DECART runs this fast and effective? Firstly, our methods work on relatively 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, hyperparameter tuned methods only have to explore datasets up to $10 * 60$ data points per project.

Secondly, as to why are hyperparameter tuned methods so effective, we note that many data mining algorithms rely on statistical properties that are emergent in large samples of data [72]. 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 On other time predictions

In our experiment, we observe that when predicting specific health indicators, some HPO methods 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

Table 17: The performance of DECART, staring mid-way through a project, then predicting 12 months into the future.

	Median MRE	IQR MRE	Median SA	IQR SA
commit	70%	119%	31%	98%
contributor	45%	87%	38%	147%
openPR	37%	85%	36%	112%
closePR	66%	93%	25%	93%
openISSUE	60%	67%	30%	154%
closedISSUE	34%	65%	38%	111%

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.

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 17 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 17 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 [19]. 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: 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 [21, 37, 38, 55, 59, 62]. 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 [12]. In our study, we do not abandon MRE since there exist known baselines for human performance in effort estimation expressed in terms of MRE [47]. 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 10 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. 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 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. [26], 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. 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 [17, 73] on different learners (e.g. random forest) or applying auto-sklearn [20] 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. 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. 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.

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.

References

1. Karan Aggarwal, Abram Hindle, and Eleni Stroulia. 2014. Co-evolution of project documentation and popularity within GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 360–363.

2. Amritanshu Agrawal, Wei Fu, Di Chen, Xipeng Shen, and Tim Menzies. 2019. How to" DODGE" Complex Software Analytics. *IEEE Transactions on Software Engineering* (2019).
3. Amritanshu Agrawal and Tim Menzies. 2018. Is" Better Data" Better Than" Better Data Miners"?. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 1050–1061.
4. Amritanshu Agrawal, Tim Menzies, Leandro L Minku, Markus Wagner, and Zhe Yu. 2018. Better Software Analytics via" DUO": Data Mining Algorithms Using/Used-by Optimizers. *arXiv preprint arXiv:1812.01550* (2018).
5. Amritanshu Agrawal, Xueqi Yang, Rishabh Agrawal, Rahul Yedida, Xipeng Shen, and Tim Menzies. 2021. Simpler Hyperparameter Optimization for Software Analytics: Why, How, When. *IEEE Transactions on Software Engineering* (2021), 1–1. `https://doi.org/10.1109/TSE.2021.3073242`
6. Lingfeng Bao, Xin Xia, David Lo, and Gail C Murphy. 2019. A large scale study of long-time contributor prediction for GitHub projects. *IEEE Transactions on Software Engineering* (2019).
7. James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*. 2546–2554.
8. Neda Hajiakhoond Bidoki, Gita Sukthankar, Heather Keathley, and Ivan Garibay. 2018. A Cross-Repository Model for Predicting Popularity in GitHub. In *2018 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, 1248–1253.
9. Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Predicting the popularity of GitHub repositories. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*. 1–10.
10. Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Understanding the factors that impact the popularity of GitHub repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 334–344.
11. M. C and S. MacDonell. 2012. Evaluating prediction systems in software project estimation. *IST* 54, 8 (2012), 820–827.
12. Chao Chen, Jamie Twycross, and Jonathan M Garibaldi. 2017. A new accuracy measure based on bounded relative error for time series forecasting. *PloS one* 12, 3 (2017).
13. Fangwei Chen, Lei Li, Jing Jiang, and Li Zhang. 2014. Predicting the number of forks for open source software project. In *Proceedings of the 2014 3rd International Workshop on Evidential Assessment of Software Technologies*. 40–47.
14. Jailton Coelho, Marco Tulio Valente, Luciano Milen, and Luciana L Silva. 2020. Is this GitHub project maintained? Measuring the level of maintenance activity of open-source projects. *Information and Software Technology* 122 (2020).
15. Paul R. Cohen. 1995. *Empirical Methods for Artificial Intelligence*. MIT Press, Cambridge, MA, USA.
16. Kevin Crowston and James Howison. 2006. Assessing the health of open source communities. *Computer* 39, 5 (2006), 89–91.

17. Swagatam Das, Sankha Subhra Mullick, and Ponnuthurai N Suganthan. 2016. Recent advances in differential evolution–an updated survey. *Swarm and Evolutionary Computation* 27 (2016), 1–30.
18. Janez Demšar. 2006. Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine Learning Research* 7 (2006), 1–30.
19. Robert Feldt and Ana Magazinius. 2010. Validity Threats in Empirical Software Engineering Research-An Initial Survey.. In *SEKE*. 374–379.
20. Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2019. Auto-sklearn: efficient and robust automated machine learning. In *Automated Machine Learning*. Springer, Cham, 113–134.
21. T. Foss, E. Stensrud, B. Kitchenham, and I. Myrtveit. 2003. A simulation study of the model evaluation criterion MMRE. *TSE* 29, 11 (2003), 985–995.
22. Apache Software Foundation. 2018. Apache Software Foundation Projects https://projects.apache.org/projects.html.
23. Linux Foundation. 2020. Community Health Analytics Open Source Software https://chaoss.community/.
24. Linux Foundation. 2020. Linux Foundation Projects https://www.linuxfoundation.org/projects/directory/.
25. Milton Friedman. 1940. A comparison of alternative tests of significance for the problem of m rankings. *The Annals of Mathematical Statistics* 11, 1 (1940), 86–92.
26. Wei Fu, Tim Menzies, and Xipeng Shen. 2016. Tuning for software analytics: Is it really necessary? *IST Journal* 76 (2016), 135–146.
27. Wei Fu, Vivek Nair, and Tim Menzies. 2016. Why is differential evolution better than grid search for tuning defect predictors? *arXiv preprint arXiv:1609.02613* (2016).
28. Junxiao Han, Shuiguang Deng, Xin Xia, Dongjing Wang, and Jianwei Yin. 2019. Characterization and Prediction of Popular Projects on GitHub. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 21–26.
29. Steffen Herbold. 2017. Comments on ScottKnottESD in response to" An empirical comparison of model validation techniques for defect prediction models". *IEEE Transactions on Software Engineering* 43, 11 (2017), 1091–1094.
30. Steffen Herbold, Alexander Trautsch, and Jens Grabowski. 2018. Correction of "A comparative study to benchmark cross-project defect prediction approaches". *IEEE Transactions on Software Engineering* 45, 6 (2018), 632–636.
31. Philipp Hohl, Michael Stupperich, Jürgen Münch, and Kurt Schneider. 2018. An assessment model to foster the adoption of agile software product lines in the automotive domain. In *2018 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC)*. IEEE, 1–9.
32. Slinger Jansen. 2014. Measuring the health of open source software ecosystems: Beyond the scope of project health. *Information and Software Technology* 56, 11 (2014), 1508–1519.
33. Oskar Jarczyk, Szymon Jaroszewicz, Adam Wierzbicki, Kamil Pawlak, and Michal Jankowski-Lorek. 2018. Surgical teams on GitHub: Modeling perfor-

mance of GitHub project development processes. *Information and Software Technology* 100 (2018), 32–46.
34. Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining GitHub. In *Proceedings of the 11th working conference on mining software repositories*. 92–101.
35. Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2016. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering* 21, 5 (2016), 2035–2071.
36. Riivo Kikas, Marlon Dumas, and Dietmar Pfahl. 2016. Using dynamic and contextual features to predict issue lifetime in GitHub projects. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 291–302.
37. B. A. Kitchenham, L. M. Pickard, S. G. MacDonell, and M. J. Shepperd. 2001. What accuracy statistics really measure. *IEEE Software* 148, 3 (2001), 81–85.
38. M. Korte and D. Port. 2008. Confidence in software cost estimation results based on MMRE and PRED. In *PROMISE'08*. 63–70.
39. Rahul Krishna, Amritanshu Agrawal, Akond Rahman, Alexander Sobran, and Timothy Menzies. 2018. What is the Connection Between Issues, Bugs, and Enhancements?. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 306–315.
40. Rahul Krishna, Vivek Nair, Pooyan Jamshidi, and Tim Menzies. 2021. Whence to Learn? Transferring Knowledge in Configurable Systems Using BEETLE. *IEEE Transactions on Software Engineering* 47, 12 (2021), 2956–2972. `https://doi.org/10.1109/TSE.2020.2983927`
41. W. B. Langdon, J. Dolado, F. Sarro, and M. Harman. 2016. Exact mean absolute error of baseline predictor, MARP0. *IST* 73 (2016), 16–18.
42. Zhifang Liao, Mengjie Yi, Yan Wang, Shengzong Liu, Hui Liu, Yan Zhang, and Yun Zhou. 2019. Healthy or not: A way to predict ecosystem health in Github. *Symmetry* 11, 2 (2019), 144.
43. Georg JP Link and Matt Germonprez. 2018. Assessing open source project health. (2018).
44. Konstantinos Manikas and Klaus Marius Hansen. 2013. Reviewing the health of software ecosystems-a conceptual framework proposal. In *Proceedings of the 5th international workshop on software ecosystems (IWSECO)*. Citeseer, 33–44.
45. Leandro L Minku. 2019. A novel online supervised hyperparameter tuning procedure applied to cross-company software effort estimation. *Empirical Software Engineering* 24, 5 (2019), 3153–3204.
46. Kjetil Molokken and Magne Jorgensen. 2003. A review of software surveys on software effort estimation. In *Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 International Symposium on*. IEEE, 223–230.
47. Kjetil Molokken and Magne Jorgensen. 2003. A review of software surveys on software effort estimation. In *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings.* IEEE, 223–230.

48. Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* 22, 6 (2017), 3219–3253.
49. Ahmed Nagy, Mercy Njima, and Lusine Mkrtchyan. 2010. A bayesian based method for agile software development release planning and project health monitoring. In *2010 International Conference on Intelligent Networking and Collaborative Systems*. IEEE, 192–199.
50. V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel. 2018. Finding Faster Configurations using FLASH. *IEEE Transactions on Software Engineering* (2018), 1–1. `https://doi.org/10.1109/TSE.2018.2870895`
51. Peter Bjorn Nemenyi. 1963. *Distribution-free multiple comparisons.* Princeton University.
52. Maria Paasivaara, Benjamin Behm, Casper Lassenius, and Minna Hallikainen. 2018. Large-scale agile transformation at Ericsson: a case study. *Empirical Software Engineering* 23, 5 (2018), 2550–2596.
53. Chris Parnin, Eric Helms, Chris Atlee, Harley Boughton, Mark Ghattas, Andy Glover, James Holman, John Micco, Brendan Murphy, Tony Savor, et al. 2017. The top 10 adages in continuous deployment. *IEEE Software* 34, 3 (2017), 86–95.
54. Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.
55. D. Port and M. Korte. 2008. Comparative studies of the model evaluation criterion mmre and pred in software cost estimation research. In *ESEM'08*. 51–60.
56. Fumin Qi, Xiao-Yuan Jing, Xiaoke Zhu, Xiaoyuan Xie, Baowen Xu, and Shi Ying. 2017. Software effort estimation based on open source projects: Case study of Github. *Information and Software Technology* 92 (2017), 145–157.
57. Alan R Santos, Josiane Kroll, Afonso Sales, Paulo Fernandes, and Daniel Wildt. 2016. Investigating the Adoption of Agile Practices in Mobile Application Development.. In *ICEIS (1)*. 490–497.
58. F. Sarro, A. Petrozziello, and M. Harman. 2016. Multi-objective software effort estimation. In *ICSE*. ACM, 619–630.
59. M. Shepperd, M. Cartwright, and G. Kadoda. 2000. On building prediction systems for software engineers. *EMSE* 5, 3 (2000), 175–182.
60. NC Shrikanth and Tim Menzies. 2021. The Early Bird Catches the Worm: Better Early Life Cycle Defect Predictors. *arXiv preprint arXiv:2105.11082* (2021).
61. Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical bayesian optimization of machine learning algorithms. *arXiv preprint arXiv:1206.2944* (2012).
62. E. Stensrud, T. Foss, B. Kitchenham, and I. Myrtveit. 2003. A further empirical investigation of the relationship of MRE and project size. *ESE* 8, 2 (2003), 139–161.
63. K. Stewart. 2019. Personnel communication.
64. R. Storn and K. Price. 1997. Differential evolution–a simple and efficient heuristic for global optimization over cont. spaces. *JoGO* 11, 4 (1997), 341–359.

65. Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. Automated parameter optimization of classification techniques for defect prediction models. In *Proceedings of the 38th International Conference on Software Engineering*. 321–332.
66. Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2018. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering* 45, 7 (2018), 683–711.
67. Huy Tu and Tim Menzies. 2021. FRUGAL: Unlocking SSL for Software Analytics. arXiv:cs.SE/2108.09847
68. Huy Tu, George Papadimitriou, Mariam Kiran, Cong Wang, Anirban Mandal, Ewa Deelman, and Tim Menzies. 2021. Mining Workflows for Anomalous Data Transfers. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 1–12. https://doi.org/10.1109/MSR52588.2021.00013
69. Dindin Wahyudin, Khabib Mustofa, Alexander Schatten, Stefan Biffl, and A Min Tjoa. 2007. Monitoring the "health" status of open source web-engineering projects. *International Journal of Web Information Systems* (2007).
70. Tao Wang, Yang Zhang, Gang Yin, Yue Yu, and Huaimin Wang. 2018. Who Will Become a Long-Term Contributor? A Prediction Model based on the Early Phase Behaviors. In *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*. 1–10.
71. Simon Weber and Jiebo Luo. 2014. What makes an open source code popular on git hub?. In *2014 IEEE International Conference on Data Mining Workshop*. IEEE, 851–855.
72. Ian H. Witten, Eibe Frank, and Mark A. Hall. 2011. *Data Mining: Practical Machine Learning Tools and Techniques* (3rd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
73. Guohua Wu, Xin Shen, Haifeng Li, Huangke Chen, Anping Lin, and Ponnuthurai N Suganthan. 2018. Ensemble of differential evolution variants. *Information Sciences* 423 (2018), 172–186.
74. Donald Wynn Jr. 2007. Assessing the health of an open source ecosystem. In *Emerging Free and Open Source Software Practices*. IGI Global, 238–258.
75. Tianpei Xia. 2021. Principles of Project health for Open Source Software.
76. Tianpei Xia, Rui Shu, Xipeng Shen, and Tim Menzies. 2020. Sequential Model Optimization for Software Effort Estimation. *IEEE Transactions on Software Engineering* (2020).
77. Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. 2016. Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? *Information and Software Technology* 74 (2016), 204–218.
78. J. Zemlin. 2017. If You Can't Measure It, You Can't Improve It. https://www.linux.com/news/if-you-cant-measure-it-you-cant-improve-it-chaoss-project-creates- tools-analyze-software/.