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

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

Tianpei Xia · Wei Fu · Rui Shu · Tim Menzies

Received: date / Accepted: date

Abstract Software developed on public platforms is a source of data that can be used to make 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.

To demonstrate this, we use 78,455 months of data from 1,628 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 (usually more than 50% wrong, sometimes over 130% wrong, median values). 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 the most recent data, for predicting multiple health indicators of open-source projects. Further, due to our use of hyperparameter optimization, the median predicting error of our predictions usually less than 10% which is much smaller than the errors seen in related work.

Our results are a compelling argument for open-sourced development. Companies that only build in-house proprietary products may be cutting themselves off from the information needed to reason about those projects.

Keywords Hyperparameter Optimization · Project Health · Machine Learning

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

1 Introduction

In 2020, open-source projects dominate the software developing environment[30, 46, 47, 52]. Over 80% of the software in any technology product or service are now open-source [67]. With so many projects now being open-source, a natural next question is “which of these project are any good?” and “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 [29]. Also, many other researchers agree that healthy open-source projects need to be “vigorous” and “active” [18, 31, 41, 42, 59, 64]. In this paper, we use 78,455 months of data from GitHub to make predictions for the April 2020 activity within 1,628 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 open pull-requests in that project;
4. The number of closed pull-requests in that project;
5. The number of open issue in that project;
6. The number of closed issue in that project;
7. Project popularity trends (number of GitHub “stars”).

We demonstrate that it is possible to accurately predict health indicators for 1, 3, 6, 12 months into the future. Predicting these indicators is important since they are used for many purposes within open source projects (for more on this see Section 2). Hence, accurately predicting these indicators is also critical.

What we show below is that, using a special kind of learning, we can generate results that are arguable better than those reported previously. For predicting the future value of our indicators, we achieve **highly predictions** error rates under 10%. It is hard to directly compare that number against many other results (due to differences in experimental conditions). But what is true is that prior researchers seen content with only **semi-approximate predictions**. For example, Bao et al.’s project health predictions for 12 months into the future were still 25% away from the best possible value (see Table 25 of [6]). Sarro et al.’s ICSE’16 paper argues for the superiority of their preferred techniques after seeing error rates in five datasets of 25, 30, 40, 45, 55% (see Figure 1a of [53]). And as for Boehm et al. [10], they had very low expectations for their COCOMO estimation system. Specifically, they declared success if estimations had error ranges less than 30% to 50%.

We conjecture that our error rates are so low since, we use **arguably better technology** than prior work. Most of the prior work neglect 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, 27, 28, 58]. Here, we use a technology called “differential evolution” (DE, explained below) to automatically tune our learners. In a result that endorses our use of this kind of hyperparameter optimization, we note that with DE, we achieve very low error rates (less than 10%).

To the best of our knowledge, this paper is **the largest study yet conducted**, **using the most recent data**, for predicting 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. [40] and *A Large Scale Study of Long-Time Contributor Prediction for GitHub Projects* by Bao et al. [6]. Those papers studied 52 and 917 projects, respectively, while we explore 1,628 projects. Further, much of our data is current (we predict for April 2020 values) while much prior work uses project data that is years to decades old [53].

We also **look at 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 GitHub project. While this is certainly an important question, it is a question about *individuals* within a project. The goal of our paper is to offer management advice at a *project level*.

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, explains 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 a replication package of this work, please see
`https://github.com/randompeople404/health_indicator_2020`.

2 Background and Related Work

2.1 Why Study Project Health?

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 are now open-source [67].

As the open source community matures, so to does its management practices. Increasingly, open source projects are becoming more structured with organizing foundations. For example, as the largest software organizations, the Apache Foundation and Linux Foundation currently host 371 projects and 166 projects respectively [23, 26]. For the promising projects, those organizations and investors invest significant funds to secure seats on the board of directors of those projects.

Although the stakeholders of these projects have different intentions, they all need indicators of project status to make decisions. For insiders, projects managers need metrics to learn where to put resources, and justifications to convince that it is cost effective to pay developers to work on specific tasks. On the other hand, for outsiders who are curious, and thinking about join these foundations, those curious organizations are more likely to invest and participate in “healthy” projects (and here, “health” is measured via the publicly available information and features on those projects).

We focus on the seven GitHub features listed in Section 1 for the predictions. Features like contributors and commits are essential parts of GitHub projects, commits record changes to files in development, while contributors make contributions in different ways, including make commits, create issues, or propose pull-request, etc. Projects cannot even exist without these features. Another measure of open source project health are the number of pull-requests and issues. Developers use pull-requests to propose changes and pull in their contributions, and use issues to track ideas, enhancements, tasks and bugs, etc.

To verify these features actually matter in the real-world business-level cases, we have conducted extensive interviews with domain experts in the field of open source projects [65]. Those experts often cite the “best development practices” published by the Linux foundation [24] as useful indicators of project success. In our interviews, these domain experts say that, as an early warning sign for project issues, they routinely monitoring trends related to these best practices. Table 1 maps key features of best development practices of the Linux kernel (as stated in [24]) to the metrics we list in the introduction. Based on Table 1, we cannot claim that any single indicator is “best”. Rather, we note the diverse range of indicators that could be used to assess different issues. Our engineering challenging, therefore, is to

Design a software analytics tool that works for whatever diverse range of indicators that might be useful for a diverse range of different business group.

To give a sense of that diversity, we offer the following examples. Some engineers might be eager to attract funding to the foundations that run large open source projects. Large organizations are willing 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 being interested. In case of GitHub projects, feature “number of stars” has been recognized as a success and popularity measurement to look at.

For another examples, other engineers 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, in our interviews, we find

GitHub Features	**Linux Practices** Short Release Cycles	Distributed Development Model	Consensus-Oriented Model
contributors	✔		
commits	✔		
open PRs		✔	
closed PRs		✔	
open issues			✔
closed issues			✔

Table 1: Some of the Linux Kernel Practices, mapped to our health indicators. For details of the practices, see http://tiny.cc/kernelpractices.

some program managers argue that their project is scoring “better” than other similar projects since that project has more new contributors per month [65]. In our study, feature “number of contributors” is a suitable measure for this way of scoring.

For other examples (for where our metrics are useful), we note that many commercial companies use open-source packages in the products they sell to customers. For that purpose, commercial companies want to use packages that are predicted to stay healthy for some time to come. If otherwise, the open-source community stops maintaining those packages, then those companies will be forced into maintaining open-source packages which that they did not build and, hence, may not fully understand.

Another case where commercial organizations can use project health predictions is the issue of *ecosystem package management*. Red Hat is very interested in project health indicators that can be automatically applied to tens of thousands of projects. When Red Hat releases a new version of systems, the 24,000+ software packages included in that distribution are delivered to tens of millions of machines, around the world. Red Hat seeks automatic project health indicators that let it:

- Decide what packages should not be included in the next distribution (due to falling health);
- Detect, then repair, falling health in popular packages. For example, in 2019, Red Hat’s engineers noted that a particularly popular project was falling from 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 [56].

Yet another use case where project health predictions 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 [38]. Predictions such as those discussed in this paper 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 expected workload to the available staff. For example,

- If a spike is expected 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 code base).
- 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 work load with too few personnel.
- It is also useful to know when the predicted 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 (a) accommodate the requests of seasoned programmers who want to either (b) learn new technologies as part of their career development; or (c) alleviate boredom; or (d) resolve personnel conflict issues.

2.2 Who Studies Project Health?

For all the above reasons, 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 [31].
- 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) [42].
- 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 help expressing open-source project health and sustainability [25].
- Weber et al. mine Python projects using a random forest classifier to predict project popularity (which they define as the star velocity in their study) [61].
- 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 estimation the popularity of GitHub repositories [11].
- Wang et al. propose a predicting 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", one a large and popular project on GitHub [60]. 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 them 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 [6].
- 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 predicting tasks [35]. 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 [32].
- 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 is an indicator of human effort [50].
- Also for the number of forks, which Chen et al. use linear regression models on 1,000 GitHub projects to predict, 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 [15].

2.3 How to Study Project Health?

We explore the literature looking for how prior researchers have explored project health. Starting with venues listed at Google Scholar Metrics "software systems"[1], 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 five years (2014 to 2019), there were at least 30 related papers.
– 10 of those papers looked at least one of the seven project health indicators we listed in our introduction [1, 9, 11, 15, 29, 32, 35, 40, 50, 61].
– 3 of those papers explored multiple indicators [9, 32, 40].
– None of those papers explored all the indicators explored in our study.

As to the technology used in that sample, of the above related papers, the preferred learners 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 makes conclusions by average across nearby examples;
– SVR: *support vector regression* uses the regressions that take the quadratic optimizer used in support vector machines and uses it to learn a parametric equation that predicts for a numeric class.

Hence, for this study, we use the above learners as baseline methods. The implementation of them are obtained from Scikit-Learn [48]. Unless being adjusted by differential evolution (discussed below), all these are run with the default settings from off-the-shelf Scikit-Learn.

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 predicting tasks. Further, while the other papers used learners with their off-the-shelf settings, Bao et al. took care to tune the control parameters of their learners. Much recent research in SE reports that such tuning can significantly improve the performance of models used in software analytics [2–4, 27, 28, 58]. The "grid-search-like" method they used was a set of nested for loops that looped over the various control parameters of the learners (so a grid search for, say, three parameters would contain three nested for loops).

We decide to explore some other different aspects to Bao et al. for several reasons:

– The data collecting procedures are different with incompatible settings.
– They explored one goal (predicting if a committer will be a long term contributor) while we want to see if it is possible to predict multiple project health indicators.
– Grid search is not recommended by the data mining literature. Bergstra et al. warn that grid search suffers from the *curse of dimensionality* [8]. That is, for any

[1] `https://scholar.google.com/citations?view_op=top_venues&hl=en&vq=eng_softwaresystems`

particular dataset and learner, the searching space of useful hyperparameters is a tiny fraction of the total space. A grid search that explores all the tuning options, which is in fine enough details to accommodate all learners and datasets, can be very slow. Hence, (a) most grid search algorithms take "large steps" in their parameter search; and (b) those large steps may miss the most useful settings of a particular learner/dataset [8].

The weaker performance of grid search is not just a theoretical possibility. Experimental results show that grid search can miss important options and performs worse than very simple alternatives [28]. Also, grid search can run needlessly slow since, often, only a few of the tuning hyperparameters really matter [7].

Accordingly, for this paper, we search control hyperparameters for our learners using another hyperparameter optimizer called Differential Evolution (DE) [57]. Initially we plan to study a large range of SBSE methods (as we have done in prior work) but given our initial results with DE are good and efficient, plus there is little improvement that might be achieved with other methods, we elect to explore other methods in future work. We use DE since prior work found it fast and comparatively more effective than grid search for other kinds of software analytic problems (e.g., defect prediction [27, 28]). Also, DE has a long history of successful application in the optimization research area, dating back to 1997 [57]. For example, Google Scholar reports that the original DE paper now has 22,906 citations (as of May 5, 2020) and that algorithm is still the focus of much on-going research [19, 20, 63]. Further, as part of this study, we spent months benchmarking DE against several other hyperparameter optimizers published since 1997. We found that DE work just as well as anything else, ran much faster, and its associated code base was much simpler to build and maintain.

```
def DE(np=20, cf=0.75, f=0.3, lives=10): # default settings
  frontier = # make "np" number of random guesses
  best = frontier.1 # any value at all
  while(lives-- > 0):
    tmp = empty
    for i = 1 to |frontier|: # size of frontier
       old = frontier_i
       x,y,z = any three from frontier, picked at random
       new= copy(old)
       for j = 1 to |new|: # for all attributes
         if rand() < cf # at probability cf...
            new.j = x.j + f(z.j - y.j) # ...change item j
       # end for
       new = new if better(new,old) else old
       tmp_i = new
       if better(new,best) then
          best = new
          lives++ # enable one more generation
       end
    # end for
   frontier = tmp
   lives--
  # end while
  return best
```

Fig. 1: Differential evolution. Pseudocode based on Storn's algorithm [57].

The pseudocode of DE algorithm is shown in Figure 1. 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, DE replaces the old tuning x_j with *new* where $new_j = x_j + f \times (y_j - z_j)$ where f is a parameter controlling differential weight.

The main loop of DE runs over the *frontier* of size np, 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 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 hyperparamter optimzier for CART regression trees. Taking advice from Storn and Fu et al. [27, 57], 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.

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

3.1 Research Questions

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

RQ1: Can we predict trends in project health indicators? We apply five popular machine learning algorithms (i.e., KNN, SVR, LNR, RFT and CART) and one state-of-the-art hyperparameter-optimized predictor (DECART) to 1,628 open-source projects collected from GitHub. Once we collected N months of data, we made predictions for the current status of each project (as of April 2020) using data from months one to $N - j$ for $j \in \{1, 3, 6, 12\}$ months in the past. DECART's median error in those experiments is under 10% (where this error is calculated from $error = 100 * |p - a|/a$ using the predicted p and actual a values seen after training on past months and testing for the April 2020 values). 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 the best predicting model, and count the number of times that each feature has been used when predicting the monthly trends. Hence we will show that:

Answer 2: In our study, "monthly_ISSUEcomments", "monthly_commit", "monthly_fork" and "monthly_star" are the most important features, while "monthly_PRmerger" is the least used feature for all seven health indicators' predictions.

RQ3: Which methods achieve the best prediction performance? We compare the performance results of each method on all 1,628 open-source projects and predicting for 1, 3, 6, and 12 months into the future. After a statistical comparison between different learners, we find that:

Answer 3: DECART generates better predicting performance than other methods in 91% of our 1,628 projects.

3.2 Data Collection

Kalliamvakou et al. warns that many repositories on GitHub are not suitable for software engineering research [33]. We follow their advice and apply a related criteria (with GitHub GraphQL API) for finding useful URLs of related projects (see Table 3). After that, 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 which contain words in that dictionary (see Table 4). After applying the criteria of Table 3 and Table 4, that left us with 1,628 repositories which we treat as engineered software projects. From these repositories, we extract features across 78,455 months of data.

Currently, there is no unique and consolidated definition of software project health [31, 40, 41]. However, most researchers agree that healthy open-source projects need to be "vigorous" and "active" [18, 31, 41, 42, 59, 64]. As Han et al. mentioned, popular open-source projects tend to be more active [29]. In our study, we select 7 features as health indicators of open-source project on GitHub: number of commits, contributors, open pull-requests, closed pull-requests, open issues, closed issues and stars. The first six features are important GitHub features to indicate the activities of the projects, while the last one is widely used as a symbol of GitHub project's popularity [1, 12, 29].

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 [16, 29, 66].

To get the latest and accurate features of our selected repositories, we use GitHub APIs for feature collection. For each project, the first commit date is used as the

Table 3: Repository selecting criteria.

Filter	Explaination
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.

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	cv	guide
present	slide	101	qa	view	form	course
org	collect	pdf	learn	blog	lesson	pic
paper	camp	summit	work	wiki	thesis	lang

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	
	# of PR mergers	monthly number of PR mergers	
	# of PR comments	monthly number of PR comments	
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	

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 in May 16, 2016. After after, we collected features from May, 2016 to April, 2020. Due to GitHub API rate limit, we could not get some features, like "monthly_commits", which require large amount of direct API calls. Instead,

Table 6: Summary of 78,455 monthly data across all 1,628 projects.

Feature	Min	Max	Median	IQR
monthly commits	0	2607	19	55
monthly contributors	0	176	3	5
monthly stars	0	6161	32	51
monthly opened PRs	0	341	2	9
monthly closed PRs	0	164	0	1
monthly merged PRs	0	329	1	7
monthly PR mergers	0	33	1	1
monthly PR comments	0	2785	3	28
monthly open issues	0	217	0	3
monthly closed issues	0	5943	10	28
monthly issue comments	0	30255	26	85
monthly forks	0	817	6	10

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 one hyperparameter optimized method 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 follow the default 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.

For each project, we have dozens of monthly data. We choose the last month as testing set and previous months as training set. 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. In case of DECART, it uses training set to find the best configuration of CART's hyperparameters (the one that achieves the closest predicting value to the actual goal on validation), then apply this configuration on CART to predict on the testing set.

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 these since (a) there are advocated in the literature [13, 53]; 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 [17]).

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. [53] favors MRE since, they argue that, it is known that the human expert performance for certain SE estimation tasks has a MRE of 30% [44]. 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 [22, 36, 37, 49, 54, 55]. Shepperd et al. champion another evaluation measure called "standardized accuracy", or SA [13]. 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 [13, 39] 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 large number (e.g., 1000 runs) of random guesses. Shepperd et al. observe that, over many runs, MAE_{guess} will converge on simply using the sample mean [13].

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 interquartile range (IQR=75th-25th percentile) of our methods' performance.

To decide which methods do better than any other, we could not use distribution-based statistics [5, 34, 43] since, for each project, we are making one estimate about the April 2020 status of a project. Hence, we need statistical methods that ask if two measurements (from two different learners) are in different places across the same distribution (the space of performance measurements across all our learners). For this purpose, we take the advice of Rosenthal et al. [51]. They recommend parametric methods, rather than non-parametric ones, since the latter have less statistical power than parametric ones. Rosenthal et al. discuss different parametric methods for asserting that one result is with some small effect of another (i.e. it is "close to"). They list dozens of effect size tests that divide into two groups: the *r* group that is based on the Pearson correlation coefficient; or the *d* family that is based on absolute differences normalized by (e.g.) the size of the standard deviation. Since Rosenthal et al comment that "none is intrinsically better than the other", we choose the most direct method. We say that one result is the same as another if their difference differs by less than Cohen's delta ($d = 30\% * standard\ deviation$). Note that we compute d separately for each different evaluation measure (SA and MRE).

4 Results

4.1 Can we predict trends in project health indicators? (RQ1)

We predict the value of health indicators for April 2020 by using data up until March 2020. That is, if a project is 60 months long (on April 2020), we predict for April 2020 using all data from its creation up until March 2020 (first 59 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 1,628 projects, using all but the last month to make predictions for April 2020. For MRE, *lower* values are *better*. Gray cells denote better results; For SA, *higher* values are *better*. In all these tables, for each row, the best learning scheme has the darkest background.

In these results, we observe that our methods provide very different performance with these 7 health indicators' prediction. In Table 7, we see that some learners have errors over 130% (LNR, predicting for number of commits). For the same task, other learners, however, only have around half of the errors (CART, 67%). Also in that table, the median MRE score of the untuned learners (KNN, LNR, SVR, RFT, CART) is over 50%. That is, these estimates are often wrong by a factor of two, or more. Another thing to observe is that untuned CART usually has lower MRE and higher SA values among those five untuned learners (5/7 in MRE, 4/7 in SA). Hence, we elect to use DE to tune CART. Also, these tables show that hyperparameter optimization

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

	KNN	LNR	SVR	RF	CART	DECART
commit	75%	139%	79%	72%	67%	17%
contributor	33%	44%	35%	33%	25%	5%
star	46%	47%	52%	44%	46%	11%
openPR	40%	52%	66%	31%	25%	4%
closePR	60%	100%	100%	50%	0%	0%
openISSUE	77%	72%	87%	74%	78%	49%
closedISSUE	40%	35%	55%	33%	33%	7%

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

	KNN	LNR	SVR	RF	CART	DECART
commit	201%	432%	223%	241%	178%	56%
contributor	55%	79%	53%	52%	62%	17%
star	56%	78%	58%	55%	55%	31%
openPR	70%	84%	69%	63%	67%	18%
closePR	100%	35%	35%	100%	100%	37%
openISSUE	44%	51%	29%	52%	63%	67%
closedISSUE	63%	77%	70%	54%	55%	18%

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

	KNN	LNR	SVR	RF	CART	DECART
commit	23%	20%	-28%	16%	35%	81%
contributor	0%	8%	-47%	35%	49%	81%
star	-54%	25%	-225%	-13%	-2%	63%
openPR	36%	60%	-69%	62%	70%	89%
closePR	0%	16%	-65%	32%	70%	92%
openISSUE	-398%	-160%	-977%	-249%	-200%	-79%
closedISSUE	18%	57%	-121%	51%	51%	86%

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

	KNN	LNR	SVR	RF	CART	DECART
commit	192%	109%	299%	166%	137%	60%
contributor	192%	139%	282%	150%	139%	64%
star	512%	112%	1210%	288%	254%	95%
openPR	192%	93%	439%	129%	120%	46%
closePR	317%	135%	397%	181%	141%	74%
openISSUE	1488%	685%	2846%	1040%	962%	424%
closedISSUE	197%	92%	649%	127%	125%	49%

is beneficial. The DECART columns of Table 7 and Table 9 show that this method has much better median SAs and MREs than the untuned methods. As shown in the last column of Table 7, the median error for DECART is under 10% (to be precise, 7%). On the other hand, the results of Table 8 and Table 10 also demonstrate the stability of DECART (with lowest IQR when measuring the performance variability of all methods).

Turning now to other prediction results, our next set of results show what happens when we make predictions over a 1, 3, 6, 12 months interval. Note that to simulate

Table 11: SA and MRE results with DECART, predicting for 1, 3, 6, 12 months into the future. All results are expressed as ratios of the predictions for one month.

	Health Indicator	1 month	3 months	6 months	12 months
Median, MRE	commit	100%	109%	111%	137%
	contributor	100%	116%	116%	137%
	star	100%	100%	110%	125%
	openPR	100%	110%	114%	133%
	closePR	100%	100%	100%	100%
	openISSUE	100%	111%	120%	133%
	closedISSUE	100%	94%	104%	115%
	median		109%	111%	133%
Median, SA	commit	100%	99%	95%	91%
	contributor	100%	100%	98%	95%
	star	100%	99%	97%	92%
	openPR	100%	100%	98%	97%
	closePR	100%	99%	97%	95%
	openISSUE	100%	82%	75%	70%
	closedISSUE	100%	98%	98%	97%
	median		99%	97%	95%
IQR, MRE	commit	100%	108%	126%	171%
	contributor	100%	113%	117%	133%
	star	100%	107%	117%	127%
	openPR	100%	121%	127%	149%
	closePR	100%	100%	100%	100%
	openISSUE	100%	100%	105%	101%
	closedISSUE	100%	98%	107%	125%
	median		107%	117%	127%
IQR, SA	commit	100%	104%	115%	125%
	contributor	100%	104%	114%	126%
	star	100%	97%	103%	102%
	openPR	100%	102%	110%	128%
	closePR	100%	99%	107%	130%
	openISSUE	100%	129%	142%	173%
	closedISSUE	100%	103%	110%	117%
	median		103%	110%	126%

predicting the status of ahead $1st$, $3rd$, $6th$, $12th$ month, for a project with N months of data, we must train on data collected from month 1 to month $N-1$, $N-3$, $N-6$, $N-12$, respectively. That is, to say that the *further* ahead our predictions, the *less* data we have for training. Hence, one thing to watch for is whether or not performance decreases as size of the training set decreases.

Table 11 presents the MRE and SA results of DECART, expressed as a ratio of the results seen after predicting one month ahead. By observing the median results (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) SA slightly degrades about 5% and (b) MRE degrades only around 33%, or less. Measured in absolute terms, this change is very small: recall that the median DECART MRE results in Table 7 for one-month-ahead predictions where less than 10%. This means that when Table 11 says that the median MRE for the 12 months predictions is worse by 133%, that translates to that $1.33 * 10\% \approx 14\%$ (which is still very low).

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.

The only counter result to **Answer 1** is when trying to predict the number of open issues. Table 7 and Table 9 show that DECART's worst MRE and SA predictions are for "openISSUE" health indicator. Additionally, in Table 9, all the SA predictions for openIssue are negative; i.e. we are performing very badly indeed when trying to predict how many issues will remain open next month. In retrospect, of course, we should have expected that predicting for how many new challenges will arise next month (in the form of new issues) is an inherently hard task.

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

In our experimental data, we have 12 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 is appropriate and fitting to ask which features, in practice, turned out to 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 the future work. To work that out, we look into the trees generated by DECART (our best learners) in the above experiments. We count the number of time of each feature has been used for prediction of every health indicator.

Those counts 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, "fork", "ISSUEcomment" and "commit" have been selected 100%, 99% and 98% when we built trees to predict "star" indicator for 1,628 repositories. Secondly, some features are bellwethers that have been used as features for multiple indicator predictions, like "commit" occurs 97%, 98%, and 93% times as features when predicting "contributor", "star" and "closeISSUE" indicators. "ISSUEcomment" has the similar pattern for "star", "openISSUE" and "closeISSUE". Thirdly, some features even

Target	commit	contributor	star	openPR	closePR	openISSUE	closeISSUE	ISSUEcomment	PRcomment	mergedPR	PRmerger	fork
commit	n/a	98%	96%	82%	68%	75%	94%	94%	81%	78%	54%	95%
contributor	97%	n/a	79%	67%	49%	55%	73%	75%	70%	65%	47%	75%
star	98%	93%	n/a	88%	69%	87%	97%	99%	85%	78%	60%	100%
openPR	74%	67%	77%	n/a	68%	58%	73%	72%	89%	83%	49%	73%
closePR	63%	50%	66%	68%	n/a	43%	66%	65%	79%	50%	32%	61%
openISSUE	78%	64%	87%	60%	46%	n/a	78%	88%	64%	51%	33%	83%
closeISSUE	93%	79%	94%	81%	69%	78%	n/a	99%	78%	79%	55%	91%
median	86%	73%	83%	75%	68%	67%	76%	88%	79%	78%	49%	83%

Table 12: How often was a feature found in the trees generated by DECART? (observed percentages in 1,628 cases)

though they belong to the same group as predicting indicator, like "openISSUE" v.s. "closeISSUE", they are not quite highly picked up by learners. In our experiment, we find that "openISSUE" was only selected 78% times, way less than "ISSUEcomment" (99%), "star" (94%) and "commit" (93%) for "closeISSUE" indicator. Last but not least, some features were less used than others. According to our experiment, "PRmerger" is the least used feature for all predictions (the median use-percentage of PRmerger is only 49%).

Answer 2: In our study, "monthly_ISSUEcomments", "monthly_commit", "monthly_fork" and "monthly_star" are the most important features, while "monthly_PRmerger" is the least used feature for all seven health indicators' predictions.

Note that none of these 12 features should be abandoned, even for "PRmerger", the least used feature in prediction (when predicting "star", this feature is used in 60% of cases).

That said, we would be hard pressed to say that Table 12 indicates that only a small subset of the Table 5 features are outstandingly the 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 policy in most cases.

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

To answer this question, we compared the performance results of each method on all 1,628 open-source projects and predicting for 1, 3, 6, and 12 months into the future.

Across 1,628 projects, we report the percent of times that one learner generating best or nearly best predictions (and the darker the cell, the more that count). To compute "nearly best" we used the Cohen's d measure introduced in Section 3.5 to compare different learning schemes in terms of MRE and SA in Table 13 and Table 14, respectively.

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 86% of all 1,628 cases.

As shown in Table 13, in terms of MRE, DECART achieves the best performance with winning rates from 86% to 99% for all predictions (the median win rate is 91%). However, the winning rates of other learners, KNN, LNR, SVR, RFT, and CART, mostly range from 20% to 50% with the exception of openISSUE where other methods have close win rate to DECART since none methods can predict it very well. That being said, our proposed method, DECART, outperforms other methods on almost all the predictions out of 1,628 projects by 25% $\sim$ 65%.

For SA results, as we see in Table 14, although the median win rate of DECART (72%) decreased a bit compare to MRE (91%), it still outperforms all the rest of methods (closest runner-up, CART only gets 44%). Specifically, DECAERT wins from 47% to 88% out of 4 different prediction ways on 1,628 projects. However, KNN wins from 10% to 56%, LNR wins from 10% to 77%, SVR wins from 4% to

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

Predicting Month	Health Indicator	KNN	LNR	SVR	RF	CART	DECART
1st month	commit	46%	33%	45%	48%	52%	86%
	contributor	46%	34%	39%	49%	52%	91%
	star	52%	53%	47%	55%	53%	91%
	openPR	44%	40%	25%	54%	55%	91%
	closePR	55%	31%	28%	61%	67%	93%
	openISSUE	87%	82%	78%	88%	86%	97%
	closedISSUE	42%	48%	29%	52%	49%	90%
3rd month	commit	47%	33%	45%	45%	51%	87%
	contributor	45%	33%	39%	48%	52%	91%
	star	52%	51%	47%	56%	52%	92%
	openPR	44%	41%	26%	55%	56%	90%
	closePR	56%	27%	29%	60%	67%	92%
	openISSUE	88%	82%	80%	89%	88%	98%
	closedISSUE	41%	47%	29%	51%	49%	91%
6th month	commit	49%	33%	48%	47%	52%	86%
	contributor	46%	32%	41%	48%	53%	91%
	star	55%	50%	50%	57%	54%	92%
	openPR	44%	40%	26%	53%	58%	91%
	closePR	57%	28%	31%	63%	66%	93%
	openISSUE	89%	82%	85%	90%	89%	97%
	closedISSUE	44%	45%	31%	50%	52%	91%
12th month	commit	49%	31%	51%	50%	52%	87%
	contributor	47%	32%	42%	49%	53%	91%
	star	57%	48%	52%	59%	56%	93%
	openPR	46%	40%	29%	54%	60%	91%
	closePR	59%	27%	35%	62%	70%	95%
	openISSUE	91%	82%	87%	92%	90%	99%
	closedISSUE	47%	45%	34%	55%	54%	92%
	median	48%	40%	40%	55%	54%	91%

43%, RFT wins from 12% to 64% and CART wins from 19% to 71%, respectively. Most of the time, all of the other methods wins rates are less than 50%. After we take a further look, SVR performs relatively worse, the median win rate is only 14% compared to the median of DECART win rate, 72%.

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

> **Answer 3**: DECART generates better predicting perfor-mance than other methods in 91% of our 1,628 projects (MRE, median).

5 Discussion

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 11,530 seconds to run DECART on 1,628 projects (on a dual core 4.67 GHz laptop); i.e. 7 seconds per datasets. This time includes optimizing CART for each specific dataset, and then making predictions. Note that, for these

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

Predicting Month	Health Indicator	KNN	LNR	SVR	RF	CART	DECART
1st month	commit	21%	19%	14%	25%	31%	63%
	contributor	15%	13%	7%	28%	43%	58%
	star	30%	47%	18%	36%	39%	73%
	openPR	46%	59%	23%	58%	62%	86%
	closePR	32%	19%	6%	41%	55%	64%
	openISSUE	55%	73%	34%	64%	68%	87%
	closedISSUE	31%	46%	14%	43%	43%	78%
3rd month	commit	17%	17%	12%	17%	27%	59%
	contributor	12%	10%	4%	25%	43%	53%
	star	31%	49%	19%	37%	39%	75%
	openPR	44%	56%	20%	54%	57%	83%
	closePR	32%	21%	7%	40%	57%	66%
	openISSUE	56%	73%	36%	63%	69%	87%
	closedISSUE	30%	45%	14%	41%	44%	77%
6th month	commit	10%	13%	9%	12%	19%	47%
	contributor	15%	14%	7%	27%	42%	57%
	star	32%	51%	20%	36%	42%	75%
	openPR	39%	52%	18%	51%	55%	80%
	closePR	32%	20%	7%	40%	53%	67%
	openISSUE	55%	75%	40%	64%	70%	87%
	closedISSUE	25%	39%	12%	36%	39%	71%
12th month	commit	13%	20%	14%	17%	26%	59%
	contributor	18%	21%	10%	28%	44%	63%
	star	31%	52%	23%	39%	43%	75%
	openPR	34%	46%	14%	42%	49%	75%
	closePR	34%	26%	10%	41%	55%	70%
	openISSUE	54%	77%	43%	64%	71%	88%
	closedISSUE	17%	30%	8%	25%	30%	65%
	median	31%	42%	14%	40%	44%	72%

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 reduction requires nearly three days of CPU per dataset [27]. If all of our 1,600+ datasets needed the same amount of CPU, then that would be a major deterrent to the use of the methods of 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 12 features shown in Table 5. That is to say, DECART's optimizations only have to explore datasets with $12 * 60$ 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 [62]. Hence they have problems reasoning about datasets with only $12 * 60$ 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

We observe that for the performance results in Table 7, while predicting the number of closed pull requests, CART and DECART achieve a 0% error for this indicator. Such zero error is a red flag that needs to be investigated since they might be due to a programming error (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.

But that raised another red flag: suppose *all* our projects had reached some steady state prior to April 2020. In that case, predicting (say) next month's health 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 much variation across the entire lifecycle of our projects.

Thirdly, just to be sure, we conducted another round of experiments. Instead of predicting for April 2020, we do the prediction for April 2019 using data collected prior to April 2018. 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 month $N/2+12$ generates comparable results as predicting for month N using all data from months 1 to $N-1$.

In summary, our results are not unduly biased by predicting just for April 2020. As the evidence, we can still obtain accurate results if we predict for April 2019 using data from before April 2018.

6 Threats to validity

The design of this study may have several validity threats [21]. The following issues should be considered to avoid jeopardizing conclusions made from this work.

	Median MRE	IQR MRE	Median SA	IQR SA
commit	14%	54%	76%	85%
contributor	6%	22%	68%	83%
star	12%	28%	54%	123%
openPR	4%	19%	89%	46%
closePR	6%	50%	85%	78%
openISSUE	24%	69%	0%	235%
closedISSUE	8%	22%	82%	56%

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

Parameter Bias: The settings to the control hyperparameters of the predicting 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.

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 [22, 36, 37, 49, 54, 55]. 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 [14]. In our study, we do not abandon MRE since there exist known baselines for human performance in effort estimation expressed in terms of MRE [45]. 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 78,455 months with 12 features of 1,628 GitHub projects data for the experiment. Also we use 7 GitHub development features as health indicators of open-source project. While we reach good predicting 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 to the project health. To mitigate these problems, we release a replicable package of our entire experiment 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 development. Companies that only build in-house proprietary products may be cutting themselves off from the information needed to reason about those 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 seven project health indicators, we can assert that usually (for 6/7 indicators), we can make predictions with less than 10% error (median values).

Our results come with some caveats. Some human activity is too random, for the law of large numbers. We know this since we cannot predict everything with high accuracy. For example, while we *can* predict how many issues will be *closed*, we were unsuccessful in building good predictions for how many will remain *open*. 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 [27], our optimization process is very fast (seven seconds per dataset). Accordingly, we assert that for predicting software project health, 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 are good but not perfect. Table 7 shows that while our median results are good, some prediction tasks are harder than others (e.g. open issues, commits, and star). Also, Table 9 shows that further improvements are possible. The DE algorithm used in this paper is essentially Storn's 1997 version and there are many more recent variants of that algorithm that could be useful [19, 63]. As to more indicators, there are more practices from real-world business-level cases to explore, collecting additional features would be helpful to find more useful indicators of more practices (e.g. "No Regression" and "zero internal boundry" in Linux Kernel Practices [24]). Another thing to try here might be deep learning. Normally we might not recommend slow algorithms like deep neural networks for reasoning over 1,600+ projects. But since our datasets are relatively small, that there might be ways to short cut the usual learning cycle. For example, suppose we found that our 1,600+ projects cluster into (say) just a handful of different project types. In that case, the target for deep learning models could be very small and fast to process.

Lastly, the GitHub project health literature offers many more targets for this kind of reasoning (e.g. the programmer assessment metrics used by Bao et al. [6]). Our results seem to indicate that the law of large numbers could apply to GitHub. If so, then there should be many more things we can readily predict about open source projects (not just the targets listed in Table 5).

Acknowledgements

This research was partially funded by blinded for review.

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. Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 1–10.
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 Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-parameter Optimization. *J. Mach. Learn. Res.* 13, 1 (Feb. 2012), 281–305.
8. 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.
9. 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.
10. Barry Boehm, Chris Abts, A Winsor Brown, Sunita Chulani, Bradford K Clark, Ellis Horowitz, Ray Madachy, Donald J Reifer, and Bert Steece. 2000. Cost estimation with COCOMO II. *ed: Upper Saddle River, NJ: Prentice-Hall* (2000).
11. 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.
12. 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.
13. M. C and S. MacDonell. 2012. Evaluating prediction systems in software project estimation. *IST* 54, 8 (2012), 820–827.
14. 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).
15. 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.
16. 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).
17. Paul R. Cohen. 1995. *Empirical Methods for Artificial Intelligence*. MIT Press, Cambridge, MA, USA.
18. Kevin Crowston and James Howison. 2006. Assessing the health of open source communities. *Computer* 39, 5 (2006), 89–91.
19. Swagatam Das, Sankha Subhra Mullick, and Ponnuthurai N Suganthan. 2016. Recent advances in differential evolution–an updated survey. *Swarm and Evolu-*

tionary Computation 27 (2016), 1–30.
20. Swagatam Das and Ponnuthurai Nagaratnam Suganthan. 2010. Differential evolution: A survey of the state-of-the-art. *IEEE transactions on evolutionary computation* 15, 1 (2010), 4–31.
21. Robert Feldt and Ana Magazinius. 2010. Validity Threats in Empirical Software Engineering Research-An Initial Survey.. In *SEKE*. 374–379.
22. 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.
23. Apache Software Foundation. 2018. Apache Software Foundation Projects https://projects.apache.org/projects.html.
24. Linux Foundation. 2018. Linux Kernel Development Best Practices https://tiny.cc/kernelpractices.
25. Linux Foundation. 2020. Community Health Analytics Open Source Software https://chaoss.community/.
26. Linux Foundation. 2020. Linux Foundation Projects https://www.linuxfoundation.org/projects/directory/.
27. Wei Fu, Tim Menzies, and Xipeng Shen. 2016. Tuning for software analytics: Is it really necessary? *IST Journal* 76 (2016), 135–146.
28. 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).
29. 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.
30. 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.
31. 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.
32. Oskar Jarczyk, Szymon Jaroszewicz, Adam Wierzbicki, Kamil Pawlak, and Michal Jankowski-Lorek. 2018. Surgical teams on GitHub: Modeling performance of GitHub project development processes. *Information and Software Technology* 100 (2018), 32–46.
33. 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.
34. Vigdis By Kampenes, Tore Dybå, Jo E Hannay, and Dag IK Sjøberg. 2007. A systematic review of effect size in software engineering experiments. *Information and Software Technology* 49, 11-12 (2007), 1073–1086.
35. 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.
36. 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.
37. M. Korte and D. Port. 2008. Confidence in software cost estimation results based on MMRE and PRED. In *PROMISE'08*. 63–70.
38. 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.
39. W. B. Langdon, J. Dolado, F. Sarro, and M. Harman. 2016. Exact mean absolute error of baseline predictor, MARP0. *IST* 73 (2016), 16–18.
40. 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.
41. Georg JP Link and Matt Germonprez. 2018. Assessing open source project health. (2018).
42. 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.
43. Nikolaos Mittas and Lefteris Angelis. 2012. Ranking and clustering software cost estimation models through a multiple comparisons algorithm. *IEEE Transactions on software engineering* 39, 4 (2012), 537–551.
44. 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.
45. 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.
46. 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.
47. 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.
48. 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.
49. 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.
50. 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.

51. Robert Rosenthal, Harris Cooper, and L Hedges. 1994. Parametric measures of effect size. *The handbook of research synthesis* 621, 2 (1994), 231–244.
52. 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.
53. F. Sarro, A. Petrozziello, and M. Harman. 2016. Multi-objective software effort estimation. In *ICSE*. ACM, 619–630.
54. M. Shepperd, M. Cartwright, and G. Kadoda. 2000. On building prediction systems for software engineers. *EMSE* 5, 3 (2000), 175–182.
55. 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.
56. K. Stewart. 2019. Personnel communication.
57. 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.
58. 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.
59. 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).
60. 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.
61. 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.
62. 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.
63. 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.
64. Donald Wynn Jr. 2007. Assessing the health of an open source ecosystem. In *Emerging Free and Open Source Software Practices*. IGI Global, 238–258.
65. Tianpei Xia. 2021. Principles of Project health for Open Source Software.
66. 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.
67. 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/.