Characterizing and Triaging Change Points

Jie Chen
Hangzhou Dianzi University
Hangzhou, China
cjie@hdu.edu.cn

Haiyang Hu
Hangzhou Dianzi University
Hangzhou, China
huhaiyang@hdu.edu.cn

Dongjin Yu
Hangzhou Dianzi University
Hangzhou, China
yudj@hdu.edu.cn

ABSTRACT

Testing software performance continuously can greatly benefit from automated verification done on continuous integration (CI) servers, but it generates a large number of performance test data with noise. To identify the change points in test data, statistical models have been developed in research. However, a considerable amount of detected change points are marked as the changes actually never need to be fixed (false positive). This work aims at giving a detailed understanding of the features of true positive change points and an automatic approach in change point triage, in order to alleviate project members’ burdens. To achieve this goal, we begin by characterizing the change points using 31 features from three dimensions, namely time series, execution result, and file history. Then, we extract the proposed features for true positive and false positive change points, and train machine learning models to triage these change points. The results demonstrate that features can be efficiently employed to characterize change points. Our model achieves an AUC of 0.985 on a median basis.

CCS CONCEPTS

• **Software and its engineering** → **Software performance**.

KEYWORDS

Machine Learning, Performance, Software Process

ACM Reference Format:
Jie Chen, Haiyang Hu, and Dongjin Yu. 2022. Characterizing and Triaging Change Points. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE ’22 Companion), April 9–13, 2022, Bejing, China.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3491204.3527487

1 INTRODUCTION

Performance testing in conjunction with continuous integration is a growing trend in the software engineering community and industry [6, 8, 10]. Performance testing is a time-consuming process that must be repeated in order to obtain valid results [11]. Based on the large amount of historical performance test data, many statistical models have been developed for change point detection in the literatures [5].

The main challenge in using the change detection technique is that there are a large number of change points that are ignored by developers. One major reason is the high rate of false positive change points reported [5]. Since the benchmarks under test is unstable, the test results do not accurately reflect the “true” performance [11]. The performance change assessment process invariably introduces bias, resulting in spurious change points that do not correspond to true ones. Additionally, even if the change point reveals to be true, they can be ignored. There are several reasons for this, including "trivial" changes that have no impact on the use, and real changes requiring significant effort to fix with little perceived benefit [20]. If the change point detection approach identified the changes that were not true, we refer to those as false positive change points (short for *FCPs*). False positives will waste developers’ time by requiring them to double-check the code. MongoDB perform performance tests with Evergreen and detects change points automatically [6]. Among the detected change points, only 44% of the change points are triaged as true. The software team must manually inspect these change points to determine whether they actually reveal changes - true positive change points (short for *TCPs*). As a result, triaging the large number of change point detection technique outcomes is a time-consuming and low-efficient process.

To make the change point detection approach more practical, it would be beneficial to have an automated mechanism that can determine whether or not a given change point is a false alarm. Numerous studies have been conducted on classifying bug reports [26] and actionable static warnings [25]. Nonetheless, in comparison to bug reports, change points lack detailed textual descriptions, and in comparison to static warnings, change points is identified with more noise. To help the understanding of the change points, we experimentally characterize *TCPs* using the 31 features and demonstrate their feasibility of distinguishing *TCPs* from *FCPs.* Then, using the proposed features, we develop machine learning models to triage change points, with promising results.

2 CHARACTERIZING CHANGE POINTS

We begin by defining the entities and properties that will be studied. To determine whether or not there has been any change in performance, a series of performance tests with various configurations is run. A time series $t \in T$ is the performance tests that have been configured uniquely using the project, variant, task, test, measurement and args options. Assuming that Vs_t is a list of versions doing the performance test in t, $Vs_t = [V_1, V_2, ..., V_n]$. For one version V_i, a set of source code files are changed, $F_{V_i} = \{f_1, f_2, ...f_m\}$. Change points is a subset of versions that are automatically identified when statistical changes occur in t, $CPs_t \subseteq Vs_t$. Beginning with a sample of triaged change points, we derive a set of features for each change point that may help distinguish false positives from true positives.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE ’22 Companion, April 9–13, 2022, Bejing, China
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9159-7/22/04…$15.00
https://doi.org/10.1145/3491204.3527487

The selection of features is motivated by previous research that investigated the root causes of performance failures and the change proneness of the code. We classified these features into three categories: features about the configuration of the performance test (Time Series), features about the execution result of the version (Execution Result) and features about the development history of the changed files (File History).

Then, we apply statistical hypothesis testing intending to characterize the *TCPs* and *FCPs*. Specially, we perform a non-parmetric MannWhitney U test [3] with the null hypothesis being that *there is NO difference between the feature values of TCPs and FCPs*. If a significant difference is observed, we believe that the corresponding features can be used to characterize a change point, and vice versa. Additionally, Cliff's Delta Effect Size is used to characterize the magnitude of such a difference [7]. Table 1 shows the studied features and quantitative analysis results. We emphasize the cell with a significant difference, and use a darker color to indicate a larger effect size. The following paragraphs analyze these findings in terms of each feature category.

Table 1: Studied Features

Dim	Feature	P-value & Effect Size		
Time Series	project	***S		
	variant	***S		
	task	***N		
	test	***N		
	measurement	***S		
	mongod	***N		
	thread_level	***M		
Execution Result	percent_change	**N		
	z_score_change	#N		
	num_build_failures	***L		
	last_change_age	***S		
	last_status	***N		
	change_rate	***S		
	max\min\avg_value	***N	***S	***N
File History	max\min\avg_cor_scores	***N	***M	***M
	max\min\avg_ver_cnts	***S	***N	***S
	max\min\avg_dev_cnts	***S	***N	***S
	max\min\avg_alines	***S	***N	***N
	max\min\avg_dlines	***N	***S	***N

*** p<0.01, ** p<0.01, * p<0.05, # p>=0.05
N: Negligible, S:Small, M:Medium, L: Large

Time Series Related Features: The data collected over time for each test measurement, with each time series being uniquely identified by the combination of configuration options. In this sense, we assume that a change point with different configuration should be treated differently. These features are extracted directly from the information of the time series for each change point. For quantitative analysis, the non-numerical features are transformed into categorical data types and assigned code that corresponds to the category's actual value. The majority of features differ significantly between *TCPs* and *FCPs*. Specially, the number of threads (*thread_level*) used to run the performance test can contribute to distinguishing the change points. In detail, the number of threads of *TCPs* is larger than *FCPs*. This may imply that additional attention should be paid to the performance tests runned with more threads. In contrast, *task*, *test* and *mongod* seems do not have substantial difference. The main causes of this phenomenon is an imbalance of data between them.

Execution Result Related Features: The execution results for change points should also be considered, and an ideal true positive change point should have large potential changes. These features are determined by the measurement of the change points. However, the features that quantify the magnitude of the change point (such as *percent_change* and *z_score_change*), which are frequently used for change point detection, do not exhibit significant differences between *TCPs* and *FCPs*, As a result, the result indicates that the additional scenarios should be considered. Unsurprisingly, the number of the linked build failures (*build_failures*) is significantly larger for *TCPs* than *FCPs*. Additionally, we design features to track the changes made to previous versions of the same time series. We further find that *TCPs* tend to be recently changed (*last_change_age* is defined as number of versions from the last change), with higher change rate (*change_rate* is defined as the percent of changed points among the previous versions in the time series) and higher minimum reported value from the test among the previous versions (*min_value*).

File History Related Features: Ideally, the development history of related entities is an important factor in determining performance regression prone tests [1]. We employ features to describe the evolution history of the changed files in a changed point. This features are extracted by analyzing the information on git versions. Obviously, most version involve changes in multiple files. A correlation score is defined refers to the confidence of an association rule between a time series and a source file. To be more precise, the correlation score for a time series t and source file f is defined as:

$$cor_score(f,t) = \frac{|Vs_{t,f} \cap CPs_t|}{|Vs_{t,f} \cap Vs_t|}$$

where $Vs_{t,f}$ is the subset of Vs_t, in which f has been changed.

For each file in one changed point, we extract four features: added and deleted lines within this file in this version (*alines* and *dlines*), and calculate the number of versions where this file changed and the cumulative number of distinct developers who contributed to this file up to this version in one time series (*ver_cnts* and *dev_cnts*). Each feature refers to the weighted sum (weighted by correlation scores) of the feature of the changed files. For example, the weighted summation of the version counts of changed files in time series t is calculated as follows, assuming that function $ver_cnt(f,t,V_i)$ computes the number of versions where f has been changed up to V_i in t.

$$ver_cnts = \sum_{f \in F_{V_i}} ver_cnt(f,t,V_i) * cor_score(f,t)$$

The results indicate that these investigated features can be utilized for characterizing *TCPs*. Specifically, features related to the *cor_scores* play a larger contribution in *TCPs* characterization. It reveals that if the changed files has higher correlation to the time series, the change tends to be a real one. Also, *TCPs* tends to be

with short history, less developers, and less added and deleted lines. This may implies that a changed should be confirmed if it related to a clear/simple code change.

3 TRIAGING CHANGE POINTS

This section will investigate to what extent the features proposed in Table 1 can be used to triage the change points. We use the features listed above to build machine learning models that predict whether a change point is true positive or false positive The following questions will be addressed.

•RQ1: Are the proposed features good at triaging change points?

•RQ2: What is the contribution of each category or each individual of features in triaging change points?

3.1 Answering RQ1

To address RQ1, we compare the prediction performance of 4 widely used classification algorithms: i.e., Logistic Regression (LR) [16], Decision Tree (DT) [15], Support Vector Machine (SVM) [4], and Random Forest (RF) [18]. Cross-validated grid-search over a parameter grid is used to optimize the estimator parameters used to apply these methods [19]. For evaluation, we conduct 10-fold cross validation and repeat it 10 times, collecting a total of 100 evaluations for each model. We evaluate the prediction performance using two widely used metrics:

- Area Under the Curve (AUC): refers to the area under the receiver operating characteristic curve, which measures the overall discriminationability of the model.
- PofG: borrowed from defect prediction studies [24] for effort-aware evaluation. PofG@K is defined as the percentage of true positive change points identified through examination of the model's topK percent change points.

Using *AUC* as a measure of the overall effectiveness, Fig 1 shows that all the four classifiers are considered applicable [21]. Even the worst one, DT-based model, achieved minimum *AUC* values of 0.79. Specially, RF-based model outperforms the others, with the *AUC* values ranging from 0.97 to 0.99 (with median of 0.985).

Fig 2 shows the average value for PofG@K (K is set between 0.1 and 0.6 with an interval 0.05) in evaluations. The average *PofG@15* of RF-based model is 0.5, and the average *PofG@35* reached 0.8. The result indicated that by inspecting a mere of 15% change points predicted by our model, 50% of the total true positive change points can be retrieved on average, and by inspecting a mere of 35% change points predicted by our model, about 80% of the total true positive change points can be retrieved. Although, DT-based model perform better than RF-based model with *PofG@K* when K ranges from 0.2 to 0.35, but its *AUC* is much lower (range from 0.79 to 0.95 with median of 0.84).

In all, machine learning models can be used to triage the change points based on our proposed features. In particular, the results indicate that RF-based model are good at triagement.

3.2 Answering RQ2

To answer RQ2, we use the same experimental setup for RQ1 and considers the best performing model from RQ1 (i.e, RF-based model) as the model under investigation in RQ2. This section builds the prediction model with a subset of the features and introduces the statistical tests performed on the feature importances and describes the results.

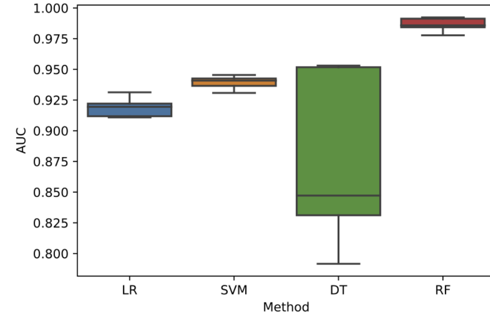

Figure 1: AUC under difference machine learning models

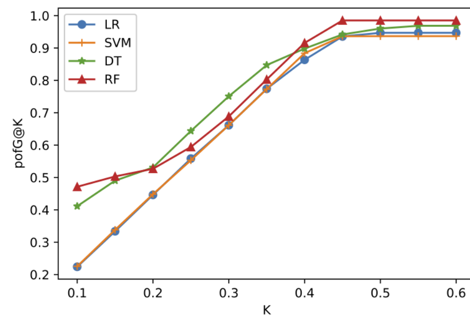

Figure 2: PofG@K under difference machine learning models

Feature categories: We investigate whether different feature categories may be important collectively for good prediction performance. We conduct experiments include a single category of features first, and then exclude each category of features separately. For each experiment, we calculate the *AUC* of the triagement and compare it to the *AUC* of the original classifier with the full feature set. The more the *AUC* values decrease, the more important the group of features are. Statistical tests are used to determine the significance and magnitude of the performance change.

Table 2: Importance of Feature Category

Feature	Include		Exclude	
	AUC	P-value & Effect Size	AUC	P-value & Effect Size
Time Series	0.88	***L	0.985	*N
Execution Result	0.949	***L	0.968	***L
File History	0.928	***L	0.963	***L

Table 2 shows the median *AUC* of models built include and exclude each category as well as the significance and magnitude of

the difference. The result demonstrates that in most case, the model suffers a sharply drop in *AUC* which is significantly lower than the performance using the whole set of features. The only exception is that the performance remain unchanged when removing the time series related features. This indicates that features related to the change and process are significantly more important for good predictions than the features related to the time series, when considering their collective importance.

Individual Feature: To investigates the importance of each individual feature, we adopt the Mean Decrease Impurity (MDI) [17]. Use the same experimental setup for RQ1, we can compute 100 importance scores for each feature. We then perform the non-parametric version of the Scott-Knott Effect Size Difference (ESD) test [22] on the importance scores to assign ranks for them.

The left part Table 3 shows the importance scores for 15 most important features. We can find that the cumulative importance of the top 15 features reached 80%. Specially, 4 features are individually important for good predictions (cumulative importance of them is larger than 50%). The number of build failures is the most important feature with a median importance of 0.38, which is consistent with the result of characterization. The second important features are the type of task and test of the time series, with a median importance of 0.042 and 0.041, respectively. The third most important features are related to the max number of deleted lines (median of 0.0397). These result suggest that there is a collective importance of different individual features from different categories for machine learning models with high prediction performance.

We also conduct the experiment to study the performance of using a subset of important features. The right part of Table 3 shows the prediction results by using the top n important features, and whether the results are significantly different from using the full feature set. The result shows that *AUC* could reach 0.8981 with the most important feature, i.e. *build_failures*. Although, *task* and *test* are important but including them has reduced the prediction performance. It happened because they can not be used to distinguish *TCPs* and *FCPs* significantly as discussed in Section 2. When we use more than the top 7 features, *AUC* is statically the same as using the full feature set (where p-value is larger than 0.001). Specially, when we use more then top 10 features, *AUC* reached 0.988 which is higher than using all features. It implies that less important features may have adverse side effects in prediction models.

Table 3: Importance of Features

Feature	Importance	Cumulative Importance	AUC	P-value & Effect Size
build_failures	0.3835	0.3835	0.8981	*** L
task	0.0418	0.4254	0.8290	*** L
test	0.0414	0.4668	0.8798	*** L
max_dlines	0.0397	0.5065	0.9745	*** L
avg_alines	0.0366	0.5431	0.9777	*** L
variant	0.0325	0.5756	0.9811	*** L
max_alines	0.0323	0.6078	0.9847	** M
avg_cor_scores	0.0297	0.6375	0.9845	* M
avg_dlines	0.0294	0.6670	0.9857	# S
thread_level	0.0274	0.6944	0.9883	# N
min_alines	0.0268	0.7211	0.9878	# N
avg_ver_cnts	0.0263	0.7474	0.9868	# N
project	0.0239	0.7714	0.9870	# N
min_cor_scores	0.0235	0.7948	0.9875	# N
avg_dev_cnts	0.0212	0.8161	0.9886	# N

4 THREATS TO VALIDITY

Internal threats stem from the acquisition of available data. Some performance test configurations are only applicable to a few versions, resulting in fewer execution records for time series that affect the quality of the datasets negatively. External validity is primarily threatened by the machine learning models and metrics used in our experiment. However, since we focus on the evaluation of features rather than machine learning techniques, the results of our study do hold for all RQs and the threat is relatively small.

5 RELATED WORK

Testing software continuously can greatly benefit from automated verification performed on continuous integration (CI) servers. Performance perspective is recently studied during this continuous development process [6, 8, 10]. Performance observations depend on the running of performance test which involved high overhead and variability [11, 12]. Identifying performance changes in the test results is a well-known challenge in performance engineering. Previous work has focused on applying change point detection algorithms on performance observations. To deal with the variability, researchers applied repetitive measurement [2, 11, 13] and statistical approaches [6, 9, 14]. Despite the performance test results, researchers also studied the performance change from the other perspectives. For example, statically-computed source code features [1, 11], history related features [1]. To make the change point detection approach more practical, it would be beneficial to automatically triage true positive changes points from the large amounts of detected change points. There have been various existing researches focused on classifying reports with kinds of features. Bug reports have been classified with rich textual descriptions [26] and even the visual features[23]. Static warnings have been classified with features related to code stucutures, file stucutures, and process [25]. But little is done for triaging change points.

6 CONCLUSIONS AND FUTURE WORK

The goal of this work is to identify the differences between *TCPs* and *FCPs* and recommend candidate true positive change points for developers to reduce their burden. Quantitative analysis shows that *TCPs* and *FCPs* have distinct features. Then we use these features to build a machine learning model to triage change points. As a result, these features can suggest *TCPs* to developers. In the future, we can extract more features by combining the relationship of the code to enhance our current approach.

7 ACKNOWLEDGMENTS

This work was supported by the National Science Foundation of China (No. 61702144), the Zhejiang Provincial National Science Foundation of China (No. LQ17F020003).

REFERENCES

[1] Jinfu Chen, Weiyi Shang, and Emad Shihab. 2020. PerfJIT: Test-level Just-in-time Prediction for Performance Regression Introducing Commits. *IEEE Transactions on Software Engineering* PP, 99 (2020), 1–1. https://doi.org/10.1109/tse.2020.3023955

[2] Jie Chen, Dongjin Yu, Haiyang Hu, Zhongjin Li, and Hua Hu. 2019. Analyzing performance-aware code changes in software development process. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 300–310.

[3] Corder, Gregory W. Foreman, and I Dale. 2009. Nonparametric Statistics for Non-Statisticians. *Wiley & Sons* (2009).

[4] Cvv Cortes, C. Cortes, V. Vapnik, C. Llorens, V. N. Vapnik, C. Cortes, and MVCB Côrtes. 1995. Support-vector networks[J]. (1995).

[5] David Daly. 2021. Creating a Virtuous Cycle in Performance Testing at MongoDB. (2021).

[6] David Daly, William Brown, Henrik Ingo, Jim O'Leary, and David Bradford. 2020. The use of change point detection to identify software performance regressions in a continuous integration system. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 67–75.

[7] Melinda R Hess and Jeffrey D Kromrey. 2004. Robust confidence intervals for effect sizes: A comparative study of Cohen'sd and Cliff's delta under non-normality and heterogeneous variances. In *annual meeting of the American Educational Research Association*. Citeseer, 1–30.

[8] Omar Javed, Joshua Heneage Dawes, Marta Han, Giovanni Franzoni, Andreas Pfeiffer, Giles Reger, and Walter Binder. 2020. *PerfCI: A Toolchain for Automated Performance Testing during Continuous Integration of Python Projects*. Association for Computing Machinery, New York, NY, USA, 1344–1348. https://doi.org/10.1145/3324884.3415288

[9] T. Kalibera and R. Jones. 2020. Quantifying Performance Changes with Effect Size Confidence Intervals. (2020).

[10] Christoph Laaber. 2019. Continuous Software Performance Assessment: Detecting Performance Problems of Software Libraries on Every Build. In *the 28th ACM SIGSOFT International Symposium*.

[11] Christoph Laaber, Mikael Basmaci, and Pasquale Salza. 2021. Predicting unstable software benchmarks using static source code features. *Empirical Software Engineering* 26, 6 (2021), 114. https://doi.org/10.1007/s10664-021-09996-y

[12] Christoph Laaber, Joel Scheuner, and Philipp Leitner. 2019. Software microbenchmarking in the cloud. How bad is it really? *Empirical Software Engineering* 24, 4 (2019), 2469–2508.

[13] Thanh HD Nguyen, Bram Adams, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. 2012. Automated detection of performance regressions using statistical process control techniques. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. 299–310.

[14] Thanh HD Nguyen, Bram Adams, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. 2012. Automated detection of performance regressions using statistical process control techniques. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. 299–310.

[15] J. R. Quinlan. 1986. Induction of Decision Trees. *Machine Learning* 1, 1 (1986), 81–106.

[16] L. J. Reed and J. Berkson. 1928. The Application of the Logistic Function to Experimental Data. *Journal of Physical Chemistry* 33, 5 (1928), 760–779.

[17] Gregg Rothermel, Doo-Hwan Bae, Zishuo Ding, Jinfu Chen, and Weiyi Shang. 2020. Towards the use of the readily available tests from the release pipeline as performance tests. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* 00 (2020), 1435–1446. https://doi.org/10.1145/3377811.3380351

[18] Claude Sammut and Geoffrey I. Webb (Eds.). 2017. *Random Decision Forests*. Springer US, Boston, MA, 1054–1054. https://doi.org/10.1007/978-1-4899-7687-1_100391

[19] M. Schonlau. [n.d.]. GRIDSEARCH: Stata module to optimize tuning parameter levels with a grid search. *Statistical Software Components* ([n. d.]).

[20] Marija Selakovic and Michael Pradel. 2016. Performance Issues and Optimizations in JavaScript: An Empirical Study. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 61–72. https://doi.org/10.1145/2884781.2884829

[21] C. Tantithamthavorn, S. Mcintosh, A. E. Hassan, and K. Matsumoto. 2016. Automated parameter optimization of classification techniques for defect prediction models. In *the 38th International Conference*.

[22] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 2019. The Impact of Automated Parameter Optimization on Defect Prediction Models. *IEEE Transactions on Software Engineering* 45, 7 (2019), 683–711. https://doi.org/10.1109/TSE.2018.2794977

[23] Junjie Wang, Mingyang Li, Song Wang, Tim Menzies, and Qing Wang. 2019. Images don't lie: Duplicate crowdtesting reports detection with screenshot information. *Information and Software Technology* 110 (2019), 139–155.

[24] Meng Yan, Xin Xia, David Lo, Ahmed E. Hassan, and Shanping Li. 2019. Characterizing and identifying reverted commits. *Empirical Software Engineering* 24, 7 (2019).

[25] Ulas Yüksel and Hasan Sözer. 2013. Automated classification of static code analysis alerts: A case study. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 532–535.

[26] Yu Zhou, Yanxiang Tong, Ruihang Gu, and Harald Gall. 2016. Combining text mining and data mining for bug report classification. *Journal of Software: Evolution and Process* 28, 3 (2016), 150–176.