2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)

Empirical study in using version histories for change risk classification

Max Kiehn
Advanced Micro Devices, Inc.
Markham, Canada
max.kiehn@amd.com

Xiangyi Pan
Advanced Micro Devices, Inc.
Markham, Canada
lucas.pan@amd.com

Fatih Camci
Amazon*
Austin, USA
camcfati@amazon.com

***Abstract*— Many techniques have been proposed for mining software repositories, predicting code quality and evaluating code changes. Prior work has established links between code ownership and churn metrics, and software quality at file and directory level based on changes that fix bugs. Other metrics have been used to evaluate individual code changes based on preceding changes that induce fixes. This paper combines the two approaches in an empirical study of assessing risk of code changes using established code ownership and churn metrics with fix inducing changes on a large proprietary code repository. We establish a machine learning model for change risk classification which achieves average precision of 0.76 using metrics from prior works and 0.90 using a wider array of metrics. Our results suggest that code ownership metrics can be applied in change risk classification models based on fix inducing changes.**

Keywords—Change risk, code ownership, file metrics, machine learning

I. INTRODUCTION

In recent years many studies explored mining of software repository data for software development analytics [1]. This data has also been used for predicting code areas with software quality issues. Prior works established multiple metrics based on code ownership [2] and code churn [3] for evaluating software quality at file and directory level. Software quality of a file or directory is usually based on the number of changes that fix defects in that file or directory [2]-[4]. Other studies [5]-[8], [13]-[14] used different sets of metrics to predict changes that introduce defects, a process called change classification [5]. In this paper we evaluate a machine learning model for change classification that utilizes code ownership and code churn metrics. We also compare performance of these metrics on change classification task to their performance on software quality evaluation in original studies.

The remainder of this paper is organized as follows: Section II describes relevant environment characteristics of the software repository used for the study. Section III discusses details of dataset generation, feature selection, and model tuning. Section IV presents and discusses the results of the empirical study. Finally, Section V provides the conclusion of this paper and future work.

II. ENVIRONMENT

A. Repository Structure

Source code in the proprietary software repository used for this paper is organized into multiple components each with a well-defined part of the source code tree. Software development is managed across multiple branches of three types: development, integration and release. Each release branch can have an arbitrary number of child sub-release branches. Due to the specific mechanics of the software repository each branch represents a unique file path.

B. Data Collection

We used a proprietary mechanism for mining historical data from the software repository into several data types, including change, branch and file, which align conceptually with artifacts described in [1]. Each change is defined by a set of features collected from three data sources as depicted in Fig 1. (1) The version control system contains direct features about changes including purpose of this change, date and time when this change is submitted, submitter and files modified by this change. (2) Every change is attached to at least one issue in the issue tracking system defined by features such as issue type, issue priority, and issue description. (3) Person related features, such as management chain and team of the submitter, are collected from internal organizational database.

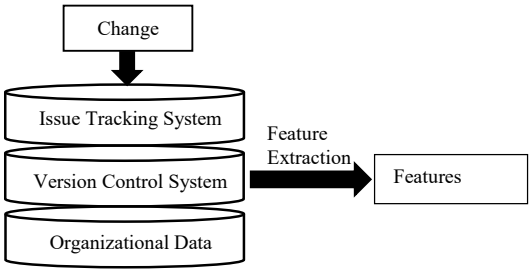

Fig. 1. Data collection process

C. Fix Inducing Change Identification

In this paper we define “fix inducing change” and “bug fixing change” as follows. If a source code file was modified, as an example, by change A, and that file was updated again by

* Dr. Camci was employed by Advanced Micro Devices during the study

2574-3864/19/$31.00 ©2019 IEEE
DOI 10.1109/MSR.2019.00018

a subsequent change B in order to fix an issue introduced by change A, we call change A a "fix inducing change" and change B a "bug fixing change".

We used two approaches to identify fix inducing changes in our historical dataset. First, fix inducing changes are recorded by developers in the issue tracking system for defects where regression or root cause analysis have been performed. This information is not mandatory however and is sparsely populated. Second, we scanned change descriptions in version control system where developers sometimes identify the cause of a bug fixing change. We look for specific keywords to flag changes and manually extract relevant fix inducing changes.

Approximately half of all identified fix inducing changes in the dataset were discovered by each approach above. These approaches are not able to identify all fix inducing changes therefore our dataset likely contains unidentified fix inducing changes. We also observe that fix inducing changes are not distributed equally across components and branches in our dataset.

III. Empirical Study

A. Dataset Description

The dataset used for this study consisted of over 50,000 changes made over the period of approximately three years. The changes were authored by more than 500 developers to selected components within proprietary source code repository primarily in C and C++ programming languages. Note that dataset characteristics above are kept vague and others, such as the number of defects and defect ratio, are omitted for confidentiality reasons.

One challenge with the dataset used for this study is that it is highly imbalanced with approximately 1% of the changes identifiable as fix inducing changes. This is a well-known issue in defect prediction field and several techniques exist to overcome this problem [4][5]. To address this, we use under-sampling to remove data points from the majority class (i.e., changes that are not labeled as fix inducing) to make a balanced dataset.

B. Model Evaluation by Cross-validation

We use dataset under-sampling along with cross-validation to evaluate our model. Results in this paper are averages of 30 evaluation runs - we repeat cross-validation 6 times with different sets of K randomly divided partitions which results in the 30 evaluation runs in total. Specifically, fix inducing changes and all other (good) changes are randomly divided into K partitions respectively (5 is the value we used). Then, the partitions of good changes are under-sampled so that the dataset is balanced. K-1 partitions are used to train the machine learning model and the remaining partition is used to evaluate the trained model.

Although a recent study [5] suggests that traditional cross-validation is not suitable for change classification and may result in false high precision as model may use future knowledge not available at the time of the change. However, in our dataset, we consider each change as an independent occurrence and its features only describe the history available up to the time the change is made. Therefore, we believe model results achieved with our cross-validation approach to be valid.

C. Feature Selection

Our goal in initial feature selection was to avoid person, branch or component specific features that may introduce model bias. We started with a set of over 70 candidate features and applied Wrapper Method [9] to reduce the set of features to the ones contributing the most to model performance. Specifically, we recursively remove every one of the current K features and compare the model with K-1 features to the model with K features. If the performance is better, then the feature is eliminated, and we repeat the previous step. Finally, the set of features is optimized to a certain combination and removing any of the remaining features will reduce performance. Fig 2. shows the results of the last Wrapper experiment. Baseline model with all features has the highest average precision.

D. List of Selected Features

Features are divided into four categories. Table I. includes features directly extracted from our issue tracking system, and version control system. Table II. includes engineered features derived from the raw data.

Another category of the features we evaluate is Time Weighted Risk (TWR) [3]. We use two measures of time for normalization: (1) mTWR uses actual time and it's accurate to minutes. (2) gTWR uses a unit time (e.g. 1) between each pair of consecutive commits.

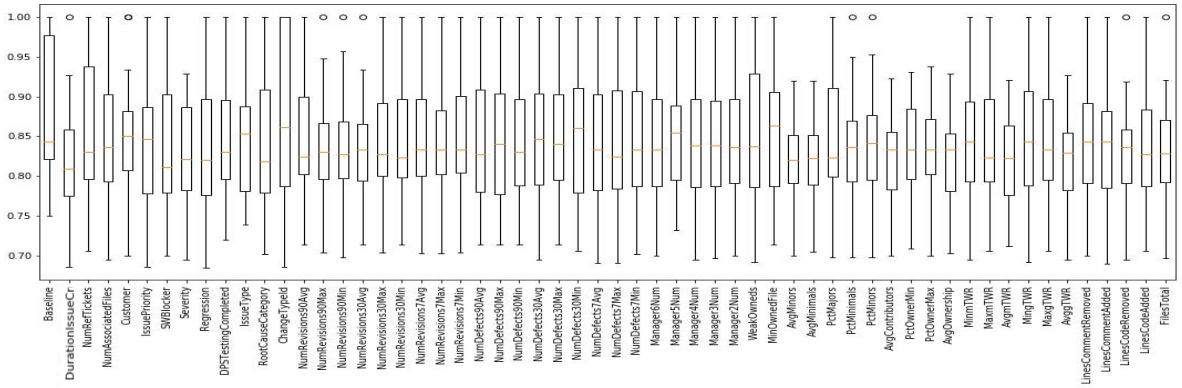

Fig. 2. Box-plots of model precision over 30 runs of the last Wrapper experiment. Baseline (leftmost) is model performance with all features listed in the figure. Individual feature boxplots are models with that feature removed. Baseline feature selection model achieves the highest average precision.

The last category includes ownership features which are based on a previous study [2]. The original study derived directory ownership metrics from metrics of all files in a directory. In our study, we use ownership metric definitions as in the original study but apply them to files in a change rather than in a directory. We call them "change ownership metrics" (i.e., aggregation of ownership metrics of all files changed in a change). Also, for calculating ownership metrics, we aggregate sub-releases to major releases. Namely, change ownership metrics are calculated based on files' histories within each major release. We also add Manager2 and Manager6 to the list of organizational features in addition to Manager3, Manager4, and Manager5 features.

TABLE I. RAW FEATURES

Feature	Description
FilesTotal	No. of files in a change
LinesCodeAdded	No.of code lines added
LinesCodeRemoved	No.of code lines removed
LinesCommentAdded	No.of comment lines added
LinesCommentRemoved	No.of comment lines removed
ChangeTypeId	Type of change (two categories)
RootCauseCategory	Issue root cause category (five categories)
Customer	Issue customer type (two categories)
IssueType	Issue type (two categories)
Regression	Is issue a regression? (two categories)
Severity	Issue severity (four categories)
Blocker	Is issue blocking? (two categories)
IssuePriority	Issue priority (four categories)

TABLE II. ENGINEERED FEATURES

Feature	Description
NumRefTickets	No. of issues related to a change
DurationIssueCreated	Duration between issue created and change submitted
NumIntegratedCLs	No. changes this change integrated
NumDefects7Min, NumDefects7Max, NumDefects7Avg, Etc.	Max/Min/Avg number of defects found in a file in the past 7/30/90 days. (9 features in total)
NumRevisions30Min, NumRevisions30Max, NumRevisions30Avg, Etc.	Max/Min/Avg number of revisions made to a file in the past 7/30/90 days. (9 features in total)

E. Classification Model

We use Neural Network [10] to train and evaluate classification models to predict fix inducing changes. As with feature selection process, we used repeated cross-validation on over 100 combinations of hyperparameters of the neural network model such as number of hidden layers, number of hidden neurons, cost functions. The hyperparameters that achieved the best results are as follows: (1) 3 hidden layers with 500, 250, and 100 hidden neurons respectively. (2) Weighted Cross-entropy [11] is used as loss function with a class weight of 4 which helps reduce false positive rate (i.e., tune to precision).

IV. RESULTS AND DISCUSSION

A. Classification Performance

Table III. shows precision, recall, and F1 values for two models: (1) The ownership model uses only ownership features, (2) The full model uses all 57 features. In addition, for each model the table shows results for model variants trained using a class weight to penalize false positives, tuned to precision, and trained with equal class weights, not tuned to precision. with and without tuning to precision. The values are calculated using a repeated cross-validation method described above. Our final precision tuned model achieves precision of 0.90 and recall of 0.20. The low recall value is partially due to our application of a large class weight to punish false positive predictions during the training process. This is confirmed by results for untuned model, trained with equal class weights, which achieves 0.70 precision and 0.72 recall. Another potential reason for low recall of the precision tuned models is the presence of unidentified fix inducing changes in our dataset.

In addition, we trained a model using only ownership features established in [2]. This model is shown in Table III as Ownership model and its precision and recall are 0.76 and 0.24 respectively for precision tuned model, and 0.66 and 0.62 for untuned model. The precision value of precision tuned ownership model is much lower than the full precision tuned model which suggests other features in this paper provide useful information for change classification.

TABLE III. OWNERSHIP MODEL AND FULL MODEL PERFORMANCE

	Precision	Recall	F1
Ownership Model (tuned)	0.76	0.24	0.36
Full Model (tuned)	0.90	0.20	0.33
Ownership model (not tuned)	0.66	0.62	0.64
Full model (not tuned)	0.70	0.72	0.71

B. Feature Analysis

In order to interpret results of newly added features and provide meaningful comparison to results in the original paper, we employ two measures as defined in the original study [2]. (1) A Variable Importance function (FilterVarImp) [12] of R's caret package is used to describe each feature's predictive importance. (2) And as computed in the original study, Spearman's rank correlation coefficient is used to describe the correlation between features and whether a change is a fix inducing change (i.e., whether a change contains 0 or 1 bug).

Table IV. shows the top 20 most important features and their importance scores. There are 16 ownership features in total, 13 of them (highlighted in Table IV.) appear in this table which confirms findings from the original study. Furthermore, we see that all features except LinesCodeAdded are engineered features. This indicates that engineered features carry more useful and important information about changes in this classification model.

Table V. shows the top 20 most correlated features and their Spearman's correlation coefficients. Spearman's

coefficient measures the direction and strength of monotonicity between two variables and is always between 1 and -1. For example, PctOwnerMax appears to be the most correlated feature. So, from its correlation value of -0.42, we see that the larger the percentage of changes of the strongest owner in a change, the lower the number of bugs in that change. Furthermore, 13 of 16 ownership features (highlighted in Table V.) appear in this table which again confirms the usefulness of ownership features. Also, it appears that engineered features are more correlated than raw features.

TABLE IV. FEATURE IMPORTANCE BASED ON FEATURES CLASSIFYING FIX INDUCING CHANGES (TOP 20 MOST IMPORTANT FEATURES)

Feature	Feature Importance	Feature	Feature Importance
PctOwnerMax	0.74	AvgContributor	0.68
MinOwnedFile	0.73	PctMajors	0.68
AvgMinors	0.73	AvggTWR	0.67
AvgMinimals	0.71	MaxgTWR	0.67
AvgOwnership	0.71	AvgmTWR	0.66
Manager6Num	0.70	MaxmTWR	0.66
PctMinimals	0.69	LinesCodeAdded	0.65
WeakOwneds	0.69	PctOwnerMin	0.64
PctMinors	0.68	NumDefects90Avg	0.64
Manager5Num	0.68	NumDefects90Max	0.64

TABLE V. SPEARMAN CORRELATION BETWEEN FEATURES AND CHANGE CLASSES (TOP 20 MOST CORRELATED FEATURES)

Feature	Spearman Correlation	Feature	Spearman Correlation
PctOwnerMax	-0.42	PctMajors	-0.31
MinOwnedFile	-0.40	AvgContributors	0.31
AvgMinors	0.39	AvggTWR	0.30
AvgMinimals	0.38	MaxgTWR	0.30
AvgOwnership	-0.36	AvgmTWR	0.28
Manager6Num	0.35	MaxmTWR	0.29
PctMinors	0.35	LinesCodeAdded	0.26
WeakOwneds	0.35	PctOwnerMin	-0.25
PctMinimals	0.34	NumDefects90Avg	0.24
Manager5Num	0.32	NumDefects90Max	0.24

C. Prior Work Comparison

The original study [2] focused on classification of files and directories in several proprietary projects therefore its results are not directly comparable to change classification results (Table III.) in this paper. It is notable that, except for full precision tuned model precision of 0.90, our results for both recall and precision are lower than recall values between 0.53 and 0.79, and precision values between 0.75 and 0.85 for classifying directories [2]. This may be due to the differences in the model, dataset or characteristics of the underlying proprietary source code. Also, as many changes span files in multiple directories, aggregating feature values across multiple files and directories may not be as useful for classification performance as those within a single directory.

Our change classification results on proprietary source code repository (Table III.) are also not directly comparable to those obtained on open source repositories in prior work on just-in-time defect prediction [13]-[14]. In addition, specific features and technical approaches used in [13]-[14] are significantly different from those used in this paper. It is worth noting however that, compared to Deeper model [14] which achieves recall of 0.6903 and precision of 0.3564 on average evaluated over six open source projects, our full untuned model achieves a similar recall of 0.72 but higher precision of 0.70. While it is beyond the scope of this paper to investigate the reasons for the differences, it may be a valuable subject for future work.

For metrics importance (Table IV.), ownership metrics seems to be the most important among the 57 features. In our model, the 3 most important metrics are PctOwnerMax, MinOwnedFile and AvgMinors while the top 3 metrics in the original study are WeakOwned, PctMinors and PctMajors. Moreover, organizational metrics have lower importance than other ownership metrics in the original study however Manager6Num ranks the 6th most important feature in our evaluation. This can be due to the fact that managers at Manager6 level are closer to developers, compared to managers at other levels, thus they are more likely to have direct impact on the changes and have more important influences. In general, the original study and our model have similar average feature importance - 0.68 and 0.68 (average of top 20) respectively.

The rank of correlation values (Table V.) is very similar to the rank of importance scores in our model - the top 12 features are the same with a slightly different order. In the original study, the most correlated features are PctMinors, PctMinimal, PctMajors and WeakOwned. But we have PctOwnerMax, MinOwnedFile and AvgMinor as the most correlated features. However, the average of absolute values of our top 20 correlations is 0.32 while the original study has an average of 0.45.

D. Limitations and Threats to validity

In production setting the low recall value of our precision tuned model may still be preferable over the higher recall but lower precision of the untuned model. A high false positive rate (i.e., low precision) could easily make users lose trust in the system [3] even if the model is improved in the future. In addition, predictions from our model were only one part of a broader system providing historical source code analytics to developers. Therefore, in order build trust with the users and to minimize potential wasted effort as a result of false alarms, we believe high precision to be the preferred approach at the introduction of such a system. As system adoption grows, however, it should be possible to use a model with lower precision if its higher recall helps flag additional issues.

We find that our model does not perform equally on all components and branches. Components with the largest number of identified fix inducing changes tend to have a similar proportion of predicted fix inducing changes. While this may reflect the actual distribution of fix inducing changes, it may also imply that the model overfits on features specific to components with higher number of fix inducing changes.

Also, due to the way file path data was captured in the dataset, the model treated the same file on development, integration and release branches as unique files (files on release and sub-release branches were treated as the same file). Since 1) the largest number of changes occur on development branch, and 2) the most fix inducing changes are identified on development branch, this meant that files on integration and release branches were treated as unique entries with their own values for each feature and did not benefit from features learned on development branch. As a result, the model makes almost no predictions for fix inducing changes on integration and release branches. More work is needed to confirm whether this is an artifact of the data collection or model bias towards development branch.

V. Conclusion and Future Work

In this paper, we focused on evaluating change classification model using a set of code ownership, churn and defect metrics on proprietary code base. We extended metrics from prior works to be usable for change level evaluation. Our model achieves average precision of 0.76 using metrics from prior works and 0.90 using a wider array of metrics. We also confirmed that changes involving files with low code ownership are riskier than those with high code ownership. Our results suggest that code ownership metrics can be applied in change risk classification models based on fix inducing changes.

The change classification model described in this paper has been adopted as part of a broader tool set providing historical source code analytics to developers. As future work, we intend to apply our findings for just-in-time change classification and evaluate the effects of model predictions on developer behavior. We also plan to update the features with consistent branch and path attribution, and evaluate a broader array of metrics highlighted in recent studies including personalized metrics [6].

Acknowledgment

The authors would like to thank many individual contributors for their help and feedback, and Advanced Micro Devices, Inc. for supporting the effort.

References

[1] J. Czerwonka, N. Nagappan, W. Schulte and B. Murphy, "CODEMINE: Building a Software Development Data Analytics Platform at Microsoft," in IEEE Software, vol. 30, no. 4, pp. 64-71, July-Aug. 2013.

[2] M. Greiler, K. Herzig and J. Czerwonka, "Code Ownership and Software Quality: A Replication Study," 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, Florence, 2015, pp. 2-12.

[3] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou and E. J. Whitehead, "Does bug prediction support human developers? Findings from a Google case study," 2013 35th International Conference on Software Engineering (ICSE), San Francisco, CA, 2013, pp. 372-381.

[4] L. Pelayo and S. Dick, "Applying Novel Resampling Strategies To Software Defect Prediction," NAFIPS 2007 - 2007 Annual Meeting of the North American Fuzzy Information Processing Society, San Diego, CA, 2007, pp. 69-72.

[5] M. Tan, L. Tan, S. Dara and C. Mayeux, "Online Defect Prediction for Imbalanced Data," 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, 2015, pp. 99-108.

[6] T. Jiang, L. Tan and S. Kim, "Personalized defect prediction," 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), Silicon Valley, CA, 2013, pp. 279-289.

[7] S. Suzuki, H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara, "An application of the pagerank algorithm to commit evaluation on git repository," in Proc. 43rd Euromicro Conf. Softw. Eng. & Advanced Applications, Aug. 2017, pp. 380–383.

[8] A. T. Misirli, E. Shihab, and Y. Kamei, "Studying high impact fix-inducing changes,"Empir. Softw. Eng., vol. 21, no. 2, pp. 605–641, June2016.

[9] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," Journal of Machine Learning Research, vol. 3, pp. 1157–1182, 2003.

[10] D. F. Specht, "A general regression neural network," in IEEE Transactions on Neural Networks, vol. 2, no. 6, pp. 568-576, Nov. 1991.

[11] S. Panchapagesan, M. Sun, A. Khare, S. Matsoukas, A. Mandal, B. Hoffmeister, S. Vitaladevuni, "Multi-task learning and weighted cross-entropy for DNN-based keyword spotting", Interspeech 2016, pp. 760-764, 2016.

[12] M. Kuhn, "caret: Classification and Regression Training," 2011.

[13] Y. Kamei et al., "A large-scale empirical study of just-in-time quality assurance," in IEEE Transactions on Software Engineering, vol. 39, no. 6, pp. 757-773, June 2013.

[14] X. Yang, D. Lo, X. Xia, Y. Zhang and J. Sun, "Deep Learning for Just-in-Time Defect Prediction," 2015 IEEE International Conference on Software Quality, Reliability and Security, Vancouver, BC, 2015, pp. 17-26.