2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Predicting Bad Commits

Finding bugs by learning their socio-organizational patterns

Christian Graber, Verifyter, San Jose CA, USA (christian.graber@verifyter.com)

Daniel Hansson, Verifyter, Lund, Sweden (daniel.hansson@verifyter.com)

Adam Tornhill, Empear, Malmo, Sweden (adam.tornhill@empear.com)

***Abstract*—This paper explores the feasibility of predicting bad commits before they happen and how this capability can be used in the context of CI and regression testing. Using standard machine learning techniques we demonstrate that it is possible to achieve 34% precision in bug prediction on a commercial ASIC IP project. That means 1 in 3 predictions were correct. Key to achieving this outcome is the combination of PinDown, Code Maat and feature engineering. PinDown is an automatic debugger of regression test failures. Code Maat is a free tool for mining and analyzing data from version control systems.**

Keywords—debugging; bug prediction; machine learning; continuous integration.

I. INTRODUCTION

Continuous Integration (CI) systems are now commonplace in the hardware design flow. CI systems make sure code updates compile and pass a small smoke test suite before they are committed to revision control. Frequent updates are encouraged to reduce the risk of integration failures. For larger design teams this can lead to significant use of farm resources. Furthermore, the coverage of the smoke test suite is typically low to average, increasing the chance for failures post-integration. This paper proposes a method to lessen both, burden on farm resources and the probability of post-integration failures, by flagging bad commits even before they enter the CI stage and by running regression test suites customized to risk profiles.

II. FEATURE SELECTION

In this paper we are using supervised machine learning techniques [1]. The anonymous training data stems from a commercial large team ASIC project that used PinDown [2] to automatically find regression bugs. A regression bug is a commit that breaks one or more tests. Raw training data was extracted from the Perforce revision control system for a period of 8 months. Labels are the bugs found by PinDown. If a commit is a bug detected by PinDown the label is True, otherwise it is False. The false positive rate in this project is below 1%, which makes for a highly unbalanced set of data. Labels generated by PinDown are very accurate. Accuracy of PinDown labels is typically exceeding 99% in most real life projects. Altogether this data set has 36793 commits containing 93 bugs. We have opted to try traditional machine learning techniques first on this set because of its small size. In this context the performance of machine learning does largely depend on the quality of the features. So feature engineering played a central role in achieving our results. We have partnered with Empear who have open-sourced their feature extraction tool Code Maat [3]. Code Maat looks at the evolution of the code base by blending technical, social and organizational information to find patterns. In addition to features produced by Code Maat Verifyter have added a set of proprietary features.

In this paper we are focusing on Code Maat features while Verifyter features are kept proprietary. By default, Code Maat runs an analysis on the number of authors per module. The authors analysis is based on the idea that the more developers working on a module, the larger the communication challenges. Logical coupling refers to modules that tend to change together. Modules that are logically coupled have a hidden, implicit dependency between them such that a change to one of them leads to a predictable change in the coupled module. Code churn is related to post-release defects. Modules with higher churn tend to have more defects. There are several different aspects of code churn supported by Code Maat: Absolute churn, churn by author and churn by entity.

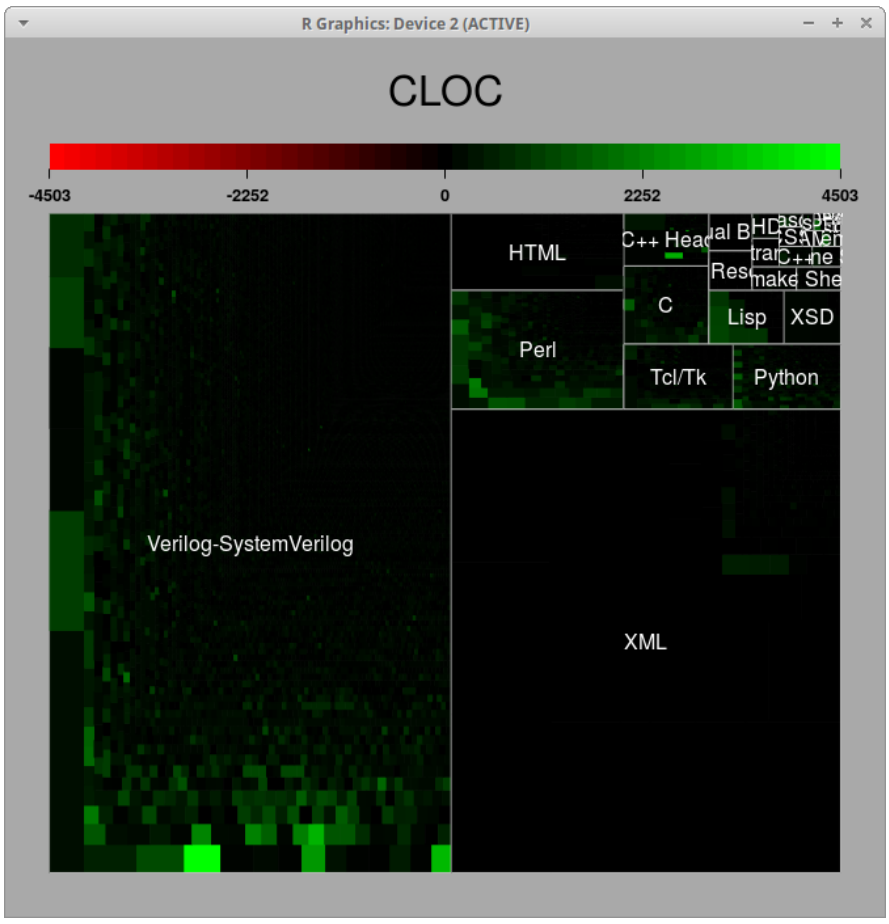

Figure 1: Treemap of Complexity

Code complexity is another strong feature used in our training. The tool used to analyze code complexity is CLOC [8]. See Figure 1. The illustration itself was done with R.

Here is how to interpret Figure 1:

- Every rectangle represents one file
- The size of the rectangle corresponds to number of source code lines
- The color of the rectangle corresponds to number of comment lines
- Rectangles are grouped by language

That means dark rectangles have very few comments while bright green ones are very well commented.

Since Pindown was configured to find bugs at commit level and features were collected at file level, they had to be 'rolled-up' to commit level by calculating their averages (_avg), maximums (_max) and sums (_tot). Here is the list of total 51 features that were used. Code Maat features are mentioned by name, other features are Pindown proprietary:

Feature	Description
author_revs_avg	Number of commits per author per file
author_revs_max	Number of commits per author per file
author_revs_tot	Number of commits per author per file
n_authors_avg	Total number of authors per file
n_authors_max	Total number of authors per file

n_authors_tot	Total number of authors per file
n_revs_avg	Total number of commits per file
n_revs_max	Total number of commits per file
n_revs_tot	Total number of commits per file
f1-f4	proprietary
soc_avg	Sum of coupling
soc_max	Sum of coupling
soc_tot	Sum of coupling
f5-f18	proprietary
cpx_total_avg	Total complexity per file
cpx_total_max	Total complexity per file
cpx_total_tot	Total complexity per file
cpx_mean_avg	Mean complexity per file
cpx_mean_max	Mean complexity per file
cpx_mean_tot	Mean complexity per file
cpx_sd_avg	Complexity standard deviation per file
cpx_sd_max	Complexity standard deviation per file
cpx_sd_tot	Complexity standard deviation per file
cpx_max_avg	Maximum complexity per file
cpx_max_max	Maximum complexity per file
cpx_max_tot	Maximum complexity per file
f19-f27	proprietary

Table 1: List of Features

III. BALANCING THE DATA SET

In this dataset only about 0.25% of commits are faulty. This makes the classes in the data set highly imbalanced. For training a machine learning classifier we need a fairly balanced set. Here we have opted for the Synthetic Minority Over-sampling Technique (SMOTE) to balance the set. SMOTE is a common oversampling technique in data analysis [4].

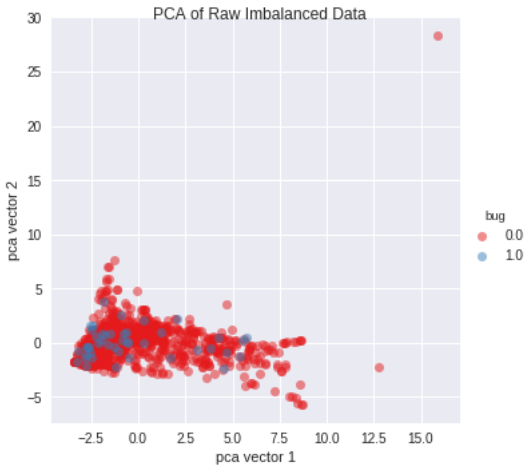

Figure 3: Imbalanced Data

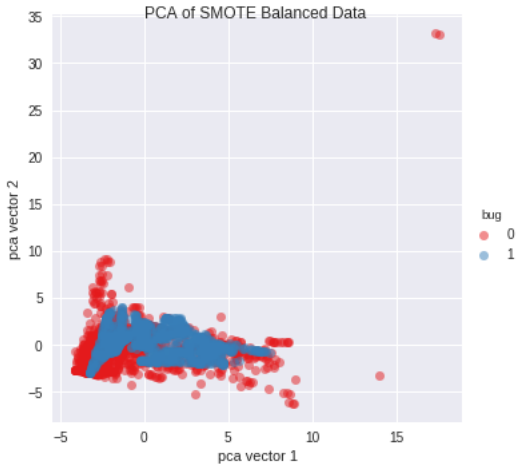

Figure 4: Balanced Data

Dimensionality reduction with Principal Component Analysis (PCA) was used to reduce the data set to two dimensions for visualization [5]. With that the effect of balancing the data set can be visualized as in figures 2 and 3. Figure 2 shows the imbalanced set where the majority of commits are good (red). Figure 3 visualizes how the number of bad commits (blue) was significantly increased. In fact, the number of bad commits increased to produce a 50% split between good and bad commits in the SMOTE data set. This set was then used for training.

IV. TRAINING THE CLASSIFIER

After some exploration of machine learning techniques we settled on XGBoost. XGBoost stands for eXtreme Gradient Boosting and is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable [6]. Every classifier has its own set of parameters, called hyper-parameters, and they need to be tuned to achieve good results. For hyper-parameter exploration we used the popular SciKit-Learn data science library [7].

Here is the set of best hyper parameters found, listing only parameters that deviate from XGBoost defaults:

Hyper-parameter	Value
max_depth	3
learning_rate	0.1
n_estimators	600

Table 2: Hyper-Parameters

V. VALIDATING THE CLASSIFIER

Finally it was time to evaluate the classifier on the validation set. The validation set was randomly selected from the original data. Since this selection and the generation of SMOTE data are random and can generate quite different outcomes we had to run validation many times to generate summary statistics. Table 3 shows summary statistics for default threshold of 0.5 (threshold defines at what probability a commit is considered a bug):

Metric	Value
Precision Mean	0.344
Precision Standard Error	0.0035
Recall Mean	0.198
Recall Standard Error	0.0024

Table 3: Classifier Metrics Summary Statistics

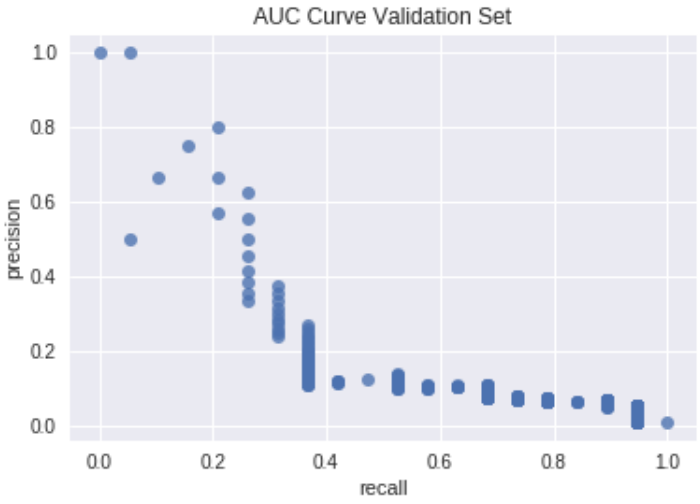

Figure 5: AUC Curve of Sample Validation Set

The AUC curve in figure 3 shows how precision relates to recall for different thresholds. Varying the threshold allows to trade off precision vs. recall. Our focus was to prioritize precision over recall.

VI. OUTCOMES

The classifier trained in the previous section can then be used to produce a risk for each commit. That is the risk for this commit to be faulty. In the context of regression testing this can be used to find faulty commits in a range of commits. We have tested this idea in live projects where the range of commits are all the commits between two subsequent regression runs. Sorting this range in chronological is illustrated in Figure 5. This would allow e.g. to run a test suite with higher coverage when the aggregate risk level is high and vice versa.

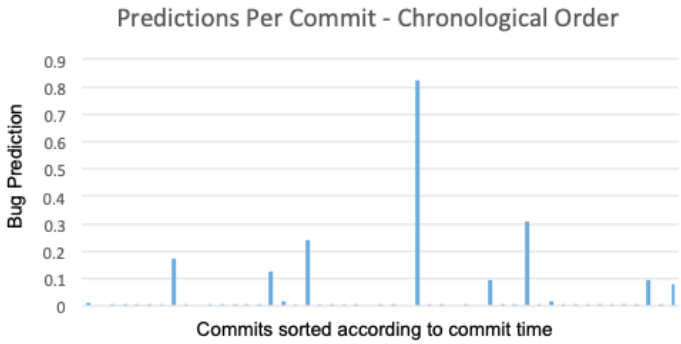

Figure 6

Sorting all the commits in range by risk level allows to give immediate feedback to the committer of high risk commits before any simulations are run. This is illustrated in Figure 6. We have verified the accuracy of these risk levels by running regression tests on the range and having PinDown find faulty commits when the regression failed.

Figure 7

The outcome of this for a real live commercial project is shown in Figure 7. We have gradually increased the classifier precision to determine at what precision the results are useful. We found empirically that a precision of 30% or above starts being very useful.

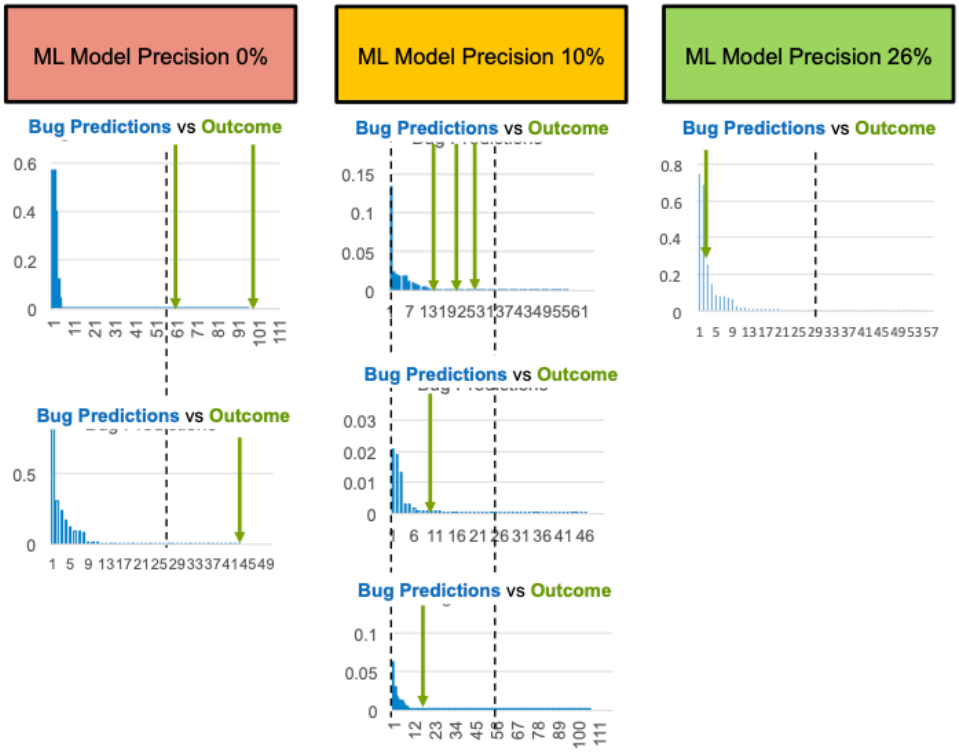

Figure 8: Sample Outcomes

VII. APPLICATIONS

Risk profiles are very useful in the context of CI and regression testing. Figure 8 illustrates four different application scenarios. The CI staging and debug scenarios are already supported by PinDown today.

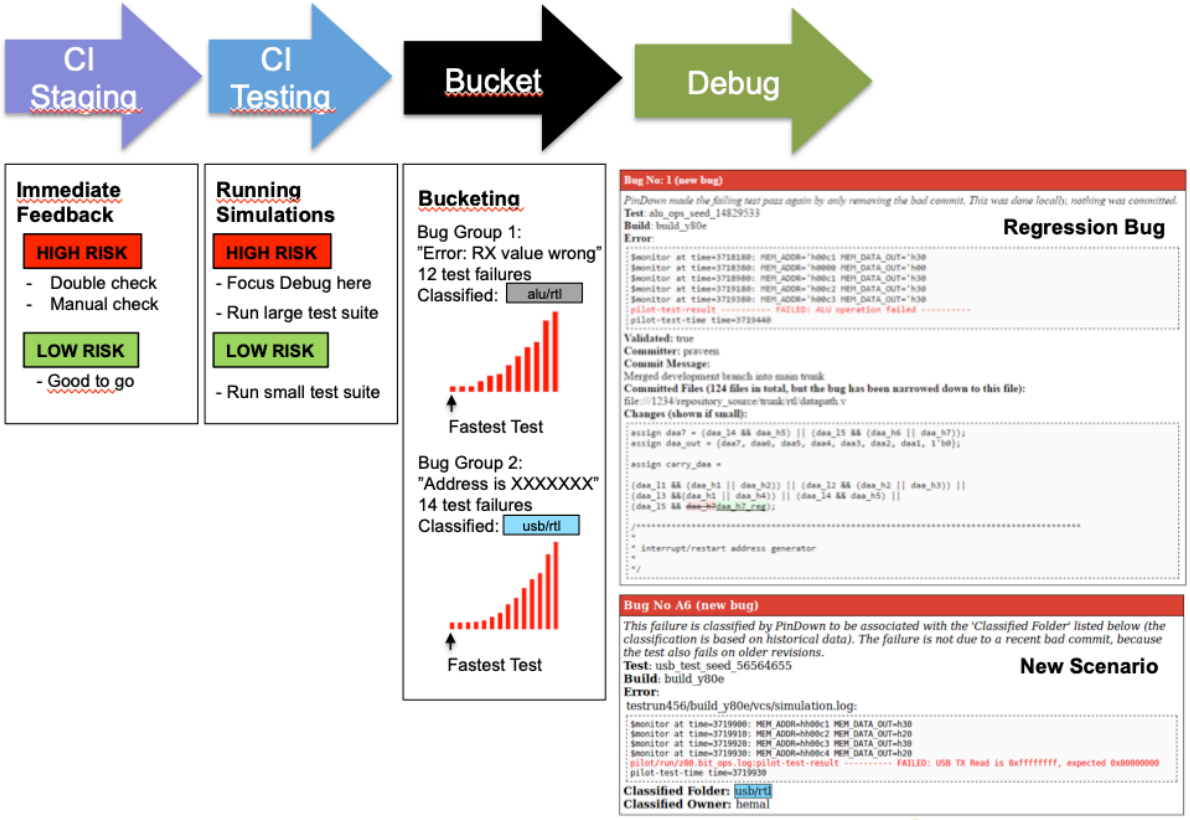

Figure 9: Applications of Machine Learning

- CI staging: A commit is first staged for CI testing. Before tests are run immediate feedback is provided to the committer.
- CI testing: Depending on the risk profile of all staged commits a larger or smaller test suite can be launched allowing to optimize farm resources.
- Bucketing: This is an application that does not use risk profiles. Instead the error signatures are used to find the best matching bucket with ML.
- Debug: Risk profiles allow optimizing the search for faulty commits by prioritizing search in high risk areas.

The user interface for immediate feedback in the CI staging phase is simply a list of staged commits ordered by risk profile. In Figure 9 six commits are predicted faulty which translates to a 92% chance for a faulty commit.

92% chance the bug is in top 6 commits

Bug Predictions

Prediction	Revision	Date	Committer	Commit message
0.999403	...stbed0.git:7c643333a5	Feb 12 2011 5:19 AM CET	carlos	repo1@7 change c2t3: c2t6: c2t5:
0.996919	...stbed0.git:f36f5c8981	Feb 12 2011 5:16 AM CET	prashant	repo1@6 change c2t3: result (bit 0) to F c2t6: result (bit 0) to F c2t5: result (bit 0) to F
0.759582	...stbed0.git:26c2a86713	Feb 12 2011 5:25 AM CET	nageshwar	repo1@9 change config_2: result (bit 0) to P
0.759582	...stbed0.git:18e57d6808	Feb 12 2011 5:13 AM CET	sharon	repo1@5 change config_2: result (bit 0) to F
0.620521	...stbed1.git:816246c63f	Feb 12 2011 5:17 AM CET	praveen	repo2@6 change c2t3: c2t6: c2t5:
0.587064	...stbed2.git:54abe25cc2	Feb 12 2011 5:21 AM CET	prashant	repo3@7 change c2t2: result (bit 0) to F c2t1: result (bit 0) to F c2t4: result (bit 0) to F
0.043611	...stbed2.git:5800e5fee3	Feb 12 2011 5:18 AM CET	carlos	repo3@6 change c2t3: c2t6: c2t5:
0.000000	...stbed1.git:f0679c2296	Feb 12 2011 5:14 AM CET	hemal	repo2@5 change empty update
0.000000	...stbed1.git:c48c888f86	Feb 12 2011 5:05 AM CET	nageshwar	repo2@2 change empty update
0.000000	...stbed1.git:463089ee6b	Feb 12 2011 5:26 AM CET	prashant	repo2@9 change empty update
0.000000	...stbed1.git:a8f0c9ff4d	Feb 12 2011 5:11 AM CET	prashant	repo2@4 change empty update
0.000000	...stbed2.git:2255f0208e	Feb 12 2011 5:06 AM CET	prashant	repo3@2 change empty update
0.000000	...stbed2.git:5da6eb734b	Feb 12 2011 5:33 AM CET	sharon	repo3@11 change empty update
0.000000	...stbed2.git:16e699db5d	Feb 12 2011 5:30 AM CET	carlos	repo3@10 change empty update
0.000000	...stbed2.git:2286b531e8	Feb 12 2011 5:12 AM CET	sharon	repo3@4 change empty update
0.000000	...stbed2.git:a936a95dbc	Feb 12 2011 5:09 AM CET	carlos	repo3@3 change empty update
0.000000	...stbed1.git:1e01c3e2d0	Feb 12 2011 5:32 AM CET	sharon	repo2@11 change empty update
0.000000	...stbed1.git:f3816f9638	Feb 12 2011 5:29 AM CET	carlos	repo2@10 change empty update
0.000000	...stbed1.git:e58f2c0b20	Feb 12 2011 5:23 AM CET	sharon	repo2@8 change empty update
0.000000	...stbed1.git:9edc996e9e	Feb 12 2011 5:20 AM CET	carlos	repo2@7 change empty update
0.000000	...stbed1.git:7559f5647d	Feb 12 2011 5:08 AM CET	carlos	repo2@3 change empty update
0.000000	...stbed2.git:70cd69b001	Feb 12 2011 5:27 AM CET	praveen	repo3@9 change empty update
0.000000	...stbed2.git:6ff1c0be08	Feb 12 2011 5:24 AM CET	nageshwar	repo3@8 change empty update
0.000000	...stbed2.git:c9537c0d63	Feb 12 2011 5:15 AM CET	hemal	repo3@5 change empty update
0.000000	...stbed0.git:e46bf2274d	Feb 12 2011 5:07 AM CET	praveen	repo1@3 change empty update
0.000000	...stbed0.git:97aeededc1	Feb 12 2011 5:22 AM CET	sharon	repo1@8 change empty update
0.000000	...stbed0.git:28f0f5ed39	Feb 12 2011 5:04 AM CET	nageshwar	repo1@2 change empty update
0.000000	...stbed0.git:9d0fb28415	Feb 12 2011 5:31 AM CET	prashant	repo1@11 change empty update
0.000000	...stbed0.git:55bf132a5c	Feb 12 2011 5:28 AM CET	carlos	repo1@10 change empty update
0.000000	...stbed0.git:3a5b0dc8fc	Feb 12 2011 5:10 AM CET	carlos	repo1@4 change empty update

Generated by PinDown v4.2.5acf331+ Jul 9 2018 2:46 PM CEST

Figure 10: Staged Commits Risk Profile

VIII. CONCLUSION

The ML techniques demonstrated in this paper produced a classifier with a precision of about 34% on average on a real life commercial project. We found this level of precision to have very useful applications in the context of CI and regression testing.

Going forward we expect to collect significantly more data to improve classifier metrics. Another direction for further exploration would be semantic analysis of source code and extraction of design patterns. In contrast the features described in this paper are language agnostic. Semantic analysis and pattern detection would allow for better representation of e.g. code complexity. However, such techniques are significantly more complex to implement while the methods laid out in this paper are very straightforward to use.

REFERENCES

[1] Supervised Learning on Wikipedia https://en.wikipedia.org/wiki/Supervised_learning

[2] Verifyter PinDown automatic debugger http://verifyter.com/technology/debug

[3] Code Maat on github https://github.com/adamtornhill/code-maat

[4] SMOTE on Wikipedia https://en.wikipedia.org/wiki/Oversampling_and_undersampling_in_data_analysis

[5] PCA on Wikipedia https://en.wikipedia.org/wiki/Principal_component_analysis

[6] XGBoost https://xgboost.readthedocs.io/en/latest/

[7] SciKit-Learn data science library http://scikit-learn.org/

[8] CLOC http://cloc.sourceforge.net/

[9] OpenCores TV80 https://opencores.org/projects/tv80