Towards a usable defect prediction tool: crossbreeding machine learning and heuristics

Vladimir Kovalenko
IntelliJ Labs / JetBrains
St. Petersburg, Russia
Vladimir.Kovalenko@jetbrains.com

Galina Alperovich
IntelliJ Labs / JetBrains
St. Petersburg, Russia
Galina.Shchekotova@jetbrains.com

ABSTRACT

A traditional supervised learning approach to defect prediction has been evaluated in numerous research papers. The biggest drawback of plain machine learning prediction models is their demand for labeled data. It sets a high restriction on applicability of defect prediction tools based on such models, and requires considerable time and resources to train the algorithms.

Building a defect prediction tool that can be used in real-world applications requires a simpler model that does not require as much human effort to train. The issue is confronted by heuristic approaches, such as the Google Bug Prediction Score, however studies show that the accuracy of heuristic models is often too low to justify their implementation.

We introduce a combined approach: by using a set of simple heuristics to identify bug fixes in version control and generating training instances based on bugfix data and language independent features, our combined algorithm performs significantly better than Google's heuristic, while still remaining fully automated and language independent. As traditional classification quality metrics are not applicable in our case due to lack of precise labeling, we use an empirical method to compare relative prediction quality based on issue tracker activity.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

defect prediction, machine learning, heuristics, bug prediction tool, VCS data mining

1. INTRODUCTION

A typical scenario of modern software development process includes a geographically distributed team working with a huge – up to hundreds of thousands VCS commits and millions of lines of code – code repository. As the development processes speed up, the techniques for efficient quality assurance and software defect management become more and more important. Even though in general it is harder to maintain a larger project, some common patterns about the way the developers work, and, in particular, about the way they introduce and fix defects in code, get clearer as the codebase grows.

Over the last few years the amount of research concerning making use of data mined from VCS repositories has grown significantly. One of the most popular research topics, which addresses the code quality problem, is development of various defect prediction models. [1]

Despite a number of successful research experiments, which show that in most cases it is possible to build a model that is capable of predicting some of the hotspots in code with quite a high precision [2], no tools based on prediction models are known to be used in industry software development environments. The reason is such tools, easy to set up and use, probably don't exist.

According to research papers, the most efficient defect prediction models are based on machine learning algorithms. The obvious drawback of these models is their demand for labeled data about past defects and fixes. We believe that a usable defect prediction tool should not require a lot of human effort to train.

A well-known example of an experiment with usage of a bug prediction tool in a real development environment is the Google case study [3]. The algorithm used in Google was based on a purely heuristic approach, so the hotspot prediction process was fully automated. In this study we have used the heuristic score described in [3] as one of comparison points for the prediction quality.

2. PROBLEM STATEMENT

There are few challenges that we came across while planning the bug prediction tool prototype development. The very first one was how to define what the "suspicious" code is. After some discussion, we have decided that suspicious, i.e. buggy, code is the code that is likely to be affected by a bugfix in the future.

There are several techniques known for finding the suspicious code. They can be divided in two groups: techniques based on the heuristics and (more advanced) techniques based on machine learning.

The former techniques' disadvantage is the fact that they can not always be generalized for multiple projects and in general require building separate models for different projects. The advantages are simplicity of implementation and relatively low computational complexity.

The latter approaches' common drawback is the fact that they require human markup for machine learning process for each project and also complexity of implementation. Among advantages are higher precision due to ability to implicitly find potentially more sophisticated dependencies between features and dependent variable than any pre-defined heuristic is capable of making use of.

In the final implementation we combine these two approaches with the goal to get the best out of each and develop a new technique that is flexible and does not require human training.

Also, opposing our work in a way to many research papers, where the applicability scope of the algorithm becomes clear after the final quality and performance results are evaluated, we decided to start from the other end and keep the usability of the future tool in mind from the very beginning.

After discission on implementation restrictions we think that a *usable* tool for prediction of the code affected by bugfixes should satisfy the following criteria:

- It should be able to process data from any version control system repository
- It should be programming language independent (therefore the lowest granularity unit of the suspicious code will be file level)
- The tool algorithm should not use human interaction like it is often the case in the learning stage of ML-based algorithms.
- The code analysis and prediction of suspicious spots should be near realtime.
- As a result the tool should produce a list of the most suspicious files that are most likely to be affected by a bugfix in the future.

In the following sections we briefly describe various aspects of implementation of the prototype of a tool designed to meet the criteria mentioned above.

3. IMPLEMENTATION

A simple prediction model based on Weka library was implemented. Also a plugin for TeamCity continuous integration server[1], that was responsible for collecting and processing all the data the model needs, was built. The TeamCity API gave us the ability to write VCS-agnostic code with no bias to exact VCS implementation used, may it be Git, Subversion, Mercurial, Perforce or other.

[1]http://jetbrains.com/teamcity

3.1 Training data

As mentioned above, human training should not be heavily used in a tool. Also in this study we could not afford to collect enough of manually labeled data for prediction quality evaluation. Instead, an attempt to find a way to automatically detect bugfixes among VCS changes was undertaken.

To make it possible, we have manually labeled pieces of VCS history for several projects, marking if a change was a critical change for a bug fix or not. After that, for every change a set of simple per-commit and per-file metrics related to relative and absolute change size, number of files affected by commit, the presence of certain keywords like "fix" in the commit message, and a few others, was calculated. Then, using one of Weka clustering algorithms, we have composed a set of weighted rules based on these metrics, which a bugfix change is likely to comply to. After the optimization of the rules and the weights, the ruleset was capable of telling if a given set of metrics represents a bugfix change or not. Experimenting with the threshold value, we were able to settle the precision and recall of the ruleset to values close to 0.7.

Finally, a minimalistic tool that finds bugfix-candidate changes in a given code repository using the heuristic ruleset was built. *This heuristically generated data is later used for classifier learning.*

3.2 Learning process

During the learning stage, the input of the model are VCS commit data and the bugfix markup generated by the tool described above.

For every file in commit, the following metrics are calculated:

- Change frequency over a certain time period
- Number of authors
- Number of previous modifications
- Time since last change
- File age
- Google heuristic score

In addition, an indicator metric, showing if the commit is related to a bug tracker issue of type "Bug" or "Exception", is calculated.

The model keeps track of modifications for every source file in the project. If the commit is marked as a bugfix, the model looks up for the sets of metrics representing the states of every file in this commit right after their previous changes. These states are believed to be "buggy", as the files are fixed later, so these states are used as training instances for Weka classifier.

We used two different Weka classifier implementaions: NaiveBayesUpdateable and HoeffdingTree. As we found later, the exact choice of classifier implementation does not change the prediction quality metric dramatically.

During the prediction phase, we calculate the same metrics for every file in commit and classify the instances using the trained Weka classifiers.

The output of the classifiers we used is a floating-point number representing a class probability. Due to imperfection of the training data, the absolute value of classifier output does not make much sense and should not be shown to the user. Instead, the model marks as "suspicious" a given number of files with the highest output value among all files recently changed in the project. After the quality evaluation method described in the next section was established, we found that the optimal number of files in the model output was between 5 and 20. We used the value of 5 in the final quality evaluation.

4. EVALUATION OF QUALITY

After the model was implemented, we needed to evaluate the prediction quality. Even though we are lucky to have quite a few developers among our colleagues, we decided not to implement a process of manual prediction quality evaluation. The main reason for such decision is that the process of integration of a tool into the company's development environment could not be quick and seamless. Moreover, we were not sure if we could collect a significant amount of feedback in reasonable time at all. Instead we have automated the process of evaluation.

4.1 Method

A simple tool, which accesses the internal issue tracker and figures out if the file that was marked as suspicious by the model was affected by a bugfix change in the future, was developed. In such case we considered a single prediction for a file "successful", and vice-versa. By calculating the fraction of successful predictions among all of them, we get a value that can be interpreted as *bugfix probability* for files the model called buggy, which we consider the main prediction quality metric.

Though it is a synthetic metric and has very little in common with standard classification quality metrics like precision and recall, it does not seem to be a problem in our case. First reason is we don't have a precisely labeled train set to evaluate the model prediction precision and recall against. The other reason is, even if we had the labeled data, using precision and recall metrics could only let us evaluate the performance of machine learning algorithms used, and not quality of the tool in general. Also it is important to note that in the projects we used for quality evaluation the issue tracker is used very consistently, so we believe that the bugfixes retrieved from issue tracker do represent real bugfixes very well, so the future bugfix probability calculated in the way described at least does correlate with human-evaluated precision.

Using a synthetic quality metric, we cannot make any conclusions out of its absolute value. To see if predictions our model makes are informative at all, we have implemented two more simple models to compare the quality metric against. First of these two models is a *random model*, which for every revision marks as "dangerous" a given number of randomly chosen files from all files modified in previous revisions. The other model is *Google Score top model*. Its output consists of a given number of files with the highest values of Google heuristic score among all files modified previously.

Table 1: Bugfix probability for various models, project A, 12 months

Random	Google score top	Naive Bayes	Decision Tree
0.49	0.56	0.71	0.70

Table 2: Bugfix probability for various models, project B, 24 months

Random	Google score top	Naive Bayes	Decision Tree
0.50	0.63	0.86	0.87

4.2 Results

We have evaluated the model quality using the repository history and issue tracker data for two JetBrains' products. The sample quality metric values are shown in the Tables 1 and 2.

Project A is a very mature project, with the development history of over 5 years and over 50000 commits in the source repository.

Project B is smaller, yet still large. It has been developed for more than 3 years so far, and its repository contains over 20000 commits.

Both projects are written mostly in Java, though both projects' codebases contain a significant amount (over 10%) of code in other languages.

As the quality evaluation method described above is imperfect also in a way that some of predictions are made by the model that is learned with the dataset that includes data for the time period *after* a prediction was made, it was important to make it safe to assume that the repository and code effort structure does not change significantly over the period studied. It was the reason why we did not use the whole projects' change history for prediction quality measurements. Instead, we used smaller chunks no longer than two years each. The data we show for Project A in the Table 1 is for the 12 months long history chunk. The data in Table 2 is calculated using the 24 months long chunk of Project B's change and issue tracker history.

The prediction quality evaluation results shown in Tables 1 and 2 show that our approach performs significantly better than Google's heuristic, while still not requiring human training thanks to automated heuristic training data generation method. It is important to note that the Google score is one of the features used for classification, so the quality improvement actually comes from the information implicitly contained in other features' values.

5. CONCLUSION

During the study described, we built a prototype of a defect prediction tool implementing a novel approach to defect prediction with strong accent on applicability. The automated quality evaluation method shows that the tool performs better than Google's heuristic approach while possessing the same advantage of being automated.

One of the authors of this paper is currently working on integration of the algorithm described into the company's development environment to collect the feedback from developers and make the final conclusion on applicability of the approach

We would like to thank our colleagues, and personally Evgeniy Koshkin, for productive discussions and help with numerous technical issues we faced during the study.

6. REFERENCES

[1] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31–41. IEEE, 2010.

[2] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *Software Engineering, IEEE Transactions on*, 34(4):485–496, 2008.

[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. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 372–381. IEEE, 2013.