Feature Defect Prediction

Stefan Strüder
University of Koblenz-Landau, Germany

Daniel Strüber
Radboud University Nijmegen, Netherlands

Thorsten Berger
Chalmers | University of Gothenburg, Sweden

Mukelabai Mukelabai
Chalmers | University of Gothenburg, Sweden

ABSTRACT

Software errors are a major nuisance in software development and can lead not only to damage of reputation but also to considerable financial losses for companies. For this reason, numerous techniques for detecting and predicting errors have been developed over the past decade, which are largely based on machine learning methods. The usual approach of these techniques is to predict errors at file level. For some years now, however, the popularity of feature-based software development has been increasing - a paradigm that relies on function increments of a software system (features) and thus ensures a wide variability of the software product. A common implementation technique for features is based on annotations with preprocessor instructions, such as `#IFDEF` and `#IFNDEF`, whose code is spread over several files of the software's source code ("code scattering"). A bug in such a feature code can have far-reaching consequences for the functionality of the entire software. If a part of the feature code contains errors, the entire function of the feature becomes faulty and may lead to the failure of the entire functionality of the software (features are "cross-cutting"). This problem is the subject of this thesis. A prediction technique for faulty features is developed, which is based on methods of machine learning. The evaluation of eight classifiers, each based on an individual classification algorithm, shows that the feature-based data set created for this thesis allows an accuracy of up to 84% for the prediction of faulty or error-free features. It is also shown how the feature orientation aspect was incorporated into the creation of the dataset and what results were achieved compared to the traditional file-based methodology. This comparison showed that the additional inclusion of the feature aspect in the file-based defect prediction has no significant influence on the prediction results.

ACM Reference Format:
Stefan Strüder, Daniel Strüber, Thorsten Berger, and Mukelabai Mukelabai. 2020. Feature Defect Prediction. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 5 pages.

1 INTRODUCTION

Software errors are a significant trigger for financial damage and damage to the reputation of companies. Such errors range from minor bugs to serious security vulnerabilities. For this reason, there is a great deal of interest in warning a developer when they release updated software code that may contain one or more bugs. To this end, researchers and software developers have developed various techniques for error detection and error prediction over the past decade, which are largely based on methods and techniques of Machine Learning [3]. These techniques usually use historical data of defective and defect-free changes to software systems in combination with a carefully compiled set of attributes (usually called features[1]) to train a given classifier [1, 5]. This can then be used to make an accurate prediction of whether a new change to a piece of software is defective or clean. The choice of algorithms for classification is large. Studies show that, out of the pool of available algorithms, both tree-based (e.g. J48, CART or Random Forest) and Bayesian algorithms (e.g. Naïve Bayes (NB), Bernoulli-NB or multinomial NB) are the most widely used [13]. Alternative algorithms include regression, k-nearest-neighbors or artificial neural networks [3]. It should be noted, however, that there is no consensus on the best available algorithm, since each algorithm has different strengths and weaknesses for specific use cases. The goal of this work is to develop such a prediction technique for software errors based on software features. These features describe increments of the functionality of a software system. The software systems developed in this way are called software product lines and consist of a set of similar software products. They are characterized by having a common set of features and a common code base [15]. By having different features along the software products, a wide variability within a product line can be achieved. In the development of the prediction technique, the implementation of features is performed by means of preprocessor instructions, such as `#IFDEF` and `#IFNDEF` (also called preprocessor directives). This approach [10], which has so far only been considered once in a case study, is promising for several reasons:

(1) Since a given feature might be historically more or less error-prone, a change that updates the feature may be more or less eror-prone as well.
(2) Features more or less likely to be error-prone might have certain characteristics that can be harnessed for defect predictoin.
(3) Code that contains a lot of feature-specific code (specifically, the so-called feature interactions) might be more error-prone than others.

The above-mentioned goal of the work consists of several subgoals. Among them is the creation of a dataset including the feature aspect. This dataset in turn serves to train a representative selection of classifiers with a subsequent comparative evaluation of these. In addition, the feature-based data set is compared with a classical file-based data set, the development of which was taken from a scientific paper [8].

Conference'17, July 2017, Washington, DC, USA
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

[1]To avoid ambiguous and misleading use of the feature term, the term "attribute" will be used throughout this paper to describe the characteristics of data.

Table 1: Used software projects

Project	Purpose	Data source
Blender	3D modelling tool	GitHub mirror
Busybox	UNIX toolkit	Git repository
Emacs	text editor	GitHub mirror
GIMP	graphics editor	GitLab repository
Gnumeric	spreadsheet	GitLab repository
gnuplot	plotting tool	GitHub mirror
Irssi	IRC client	GitHub repository
libxml2	XML parser	GitLab repository
lighttpd	web server	Git repository
MPSolve	polynom solver	GitHub repository
Parrot	virtual machine	GitHub repository
Vim	text editor	GitHub repository

2 BACKGROUND / RELATED WORK

3 METHODOLOGY

3.1 Creation of Dataset

The data set forms the basis for the training of the machine learning classifiers and is created specifically for this work based on commits from 12 feature-based software projects. The software projects are selected on the basis of previous use in scientific literature [6, 7, 10, 11]. The software projects used for this thesis are listed in Table 1 together with their purpose and data sources.

To get the commit data of the software projects the Python library PyDriller[2] was used [14]. This allows easy data extraction from Git repositories to obtain commits, commit messages, developers, diffs, and more (called "metadata" in the following). The URLs to the Git repositories of the software projects were used as input for the specially created Python scripts for receiving the commit metadata. Furthermore, the metadata was divided into commits per release. This was made possible by specifying release tags in the PyDriller code, based on the tag structure of Git repositories. For each modified file within a commit and a release, the following metadata was retrieved using PyDriller:

- commit hash (unique identifier of a commit)
- commit author
- commit message
- filename
- lines of code
- cyclomatic complexity
- number of added lines
- number of removed lines
- diff (changeset)

The data obtained in this way was stored in a MySQL database after retrieval. For each software project, a separate table was created in which, in addition to the metadata above, the name of the software project and the release numbers associated with the commits were stored. Each modified file of a commit receives one row of the database tables. The further construction of the data set is divided into several phases of data processing and optimization.

The first phase consists of extracting the features involved in a modified file. This was done by using a Python script to identify the preprocessor statements `#IFDEF` and `#IFNDEF` in the diffs of the modified files, and then saving the string following the directives as a feature until the end of the line of code. The identification was done using regular expressions. The features identified per file are stored in an additional column in the respective MySQL tables. In case a feature is identified after the `#IFNDEF` directive, the feature is stored with a preceding "not". It will be saved as a separate feature, along with its non-negated form. Combinations of features are stored in their identified form. If no feature could be identified, "`none`" is saved accordingly.

This way of identification has some obstacles. In some C programming paradigms, it is common to include header files in the source code using preprocessor directives, so that they appear to be features. However, these "header features", as they will be referred to later, should be ignored as they do not create variability throughout the source code. In general, these header features are identifiable by their naming in the form of an appended `_h_` to the feature name, such as `featurename_h_`. This appended part allows the header features to be identified and filtered out using regular expressions. It is also possible that "wrong" features can be identified. Examples of this can come from `#IFDEFs` used in comments. Such false features were removed in a manual review of the identified features and replaced with "`none`".

The next phase of processing consists of identifying corrective commits. A common method used for this, and one used in this paper, is to analyze commit messages for the presence of certain keywords [16]. The keywords are "bug," "bugs," "bugfix," "error," "fail," "fix," "fixed," and "fixes. The analysis was carried out by means of a Python script using simple methods of Natural Language Processing (NLP). Identification was limited to the first line of each commit message. The results were stored in another column of the MySQL tables (true = corrective, false = not corrective).

The search for corrective commits is followed by an analysis for commits that introduced bugs. A PyDriller implementation of the SZZ algorithm according to Sliwerski, Zimmermann and Zeller was used [12, 14]. This algorithm allows to find commits that introduce bugs in locally stored software repositories [2]. It requires that the corrective commits have already been identified, as they serve as the algorithm's input [2].

An overview of the number of corrective and bug-introducing commits and the number of features identified per software project is given in Table 2.

Here!

3.2 Selection of Metrics

3.3 Selection of Classifiers

By choosing the programming language Python, the decision to select a specific machine learning tool was already foreseeable. The Python library scikit-learn[3], which was developed in 2007 by Pedregosa et. al [9], is used. The tool offers a wide range of machine

[2]https://github.com/ishepard/pydriller

[3]https://scikit-learn.org/

Table 2: Number of corrective and bug-introducing commits and number of identified features

Project	#corrective	#bug-introducing	#features
Blender	7.760	3.776	1.400
Busybox	1.236	802	628
Emacs	4.269	2.532	718
GIMP	1.380	854	204
Gnumeric	1.498	1.191	637
gnuplot	854	1.215	558
Irssi	52	22	9
libxml2	324	88	200
lighttpd	1.078	929	230
MPSolve	151	211	54
Parrot	3.109	3.072	397
Vim	371	696	1.158

learning algorithms for supervised and unsupervised learning and also allows easy use and easy integration of other Python libraries, such as the Matplotlib for creating mathematical representations [9].

The WEKA Workbench[4] is also used as an additional machine learning tool. The tool (WEKA as acronym for **W**aikato **E**nvironment for **K**nowledge **A**nalysis) was developed at the University of Waikato in New Zealand and offers a large collection of machine learning algorithms and preprocessing tools for use within a graphical user interface [4]. There are also interfaces for the Java programming language [4].

The use of two machine learning tools allows a comparison of the respective implementations of the classification algorithms used in the subsequent evaluation. An overview of the selected classification algorithms for each tool can be found in **??**.

4 EVALUATION

5 DISCUSSION

6 CONCLUSION

REFERENCES

[1] Abdullah Alsaeedi and Mohammad Zubair Khan. 2019. Software Defect Prediction Using Supervised Machine Learning and Ensemble Techniques: A Comparative Study. *Journal of Software Engineering and Applications* 12, 05 (2019), 85–100. https://doi.org/10.4236/jsea.2019.125007

[2] Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. 2019. SZZ unleashed: an open implementation of the SZZ algorithm - featuring example usage in a study of just-in-time bug prediction for the Jenkins project. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation - MaLTeSQuE 2019*. ACM Press. https://doi.org/10.1145/3340482.3342742

[3] Venkata Udaya B. Challagulla, Farokh B. Bastani, I. Ling Yen, and Raymond A. Paul. 2008. Empirical assessment of machine learning based software defect prediction techniques. *International Journal on Artificial Intelligence Tools* 17, 2 (2008), 389–400. https://doi.org/10.1142/S0218213008003947

[4] Eibe Frank, Mark A. Hall, and Ian H. Witten. 2016. *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"* (fourth edition ed.). Morgan Kaufmann.

[5] Awni Hammouri, Mustafa Hammad, Mohammad Alnabhan, and Fatima Alsarayrah. 2018. Software Bug Prediction using Machine Learning Approach. *International Journal of Advanced Computer Science and Applications* 9, 2 (2018). https://doi.org/10.14569/ijacsa.2018.090212

[6] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. 2015. Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Empirical Software Engineering* 21, 2 (apr 2015), 449–482. https://doi.org/10.1007/s10664-015-9360-1

[7] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*. ACM Press. https://doi.org/10.1145/1806799.1806819

[8] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 2008. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 13th international conference on Software engineering - ICSE '08*. ACM Press. https://doi.org/10.1145/1368088.1368114

[9] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[10] Rodrigo Queiroz, Thorsten Berger, and Krzysztof Czarnecki. 2016. Towards predicting feature defects in software product lines. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development - FOSD 2016*. ACM Press. https://doi.org/10.1145/3001867.3001874

[11] Rodrigo Queiroz, Leonardo Passos, Marco Tulio Valente, Claus Hunsen, Sven Apel, and Krzysztof Czarnecki. 2015. The shape of feature code: an analysis of twenty C-preprocessor-based systems. *Software & Systems Modeling* 16, 1 (jul 2015), 77–96. https://doi.org/10.1007/s10270-015-0483-z

[12] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes* 30, 4 (jul 2005), 1. https://doi.org/10.1145/1082983.1083147

[13] Le Son, Nakul Pritam, Manju Khari, Raghvendra Kumar, Pham Phuong, and Pham Thong. 2019. Empirical Study of Software Defect Prediction: A Systematic Mapping. *Symmetry* 11, 2 (feb 2019), 212. https://doi.org/10.3390/sym11020212

[14] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. ACM Press. https://doi.org/10.1145/3236024.3264598

[15] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surveys* 47, 1 (jul 2014), 1–45. https://doi.org/10.1145/2580950

[16] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting Defects for Eclipse. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*. IEEE. https://doi.org/10.1109/promise.2007.10

[4]https://www.cs.waikato.ac.nz/ml/weka/

Table 3: Overview of used metrics

	Metric	Description	Source
Process metrics	Number of commits (COMM)	Number of commits associated with the feature/file in a release.	[10]
	Number of active developers (ADEV)	Number of developers who have edited (changed, deleted or added) the feature / file within a release	[10]
	Number of distinct developers (DDEV)	Cumultative number of developers who have edited (changed, deleted or added) the feature / file within a release	[10]
	Experience of all develepoers (EXP)	geometric mean of the experience of all developers who have edited (changed, deleted or added) the feature / file within a release.~ Experience is defined as the sum of the changed, deleted or added lines in the commits associated with the feature / file.	[10]
	Experience of the most involved developers (OEXP)	Experience of the developer who has edited (changed, deleted or added) the feature / file most often within a release. Experience is defined as the sum of changed, deleted, or added lines in the commits associated with the feature/file.	[10]
	Degree of modifications (MODD)	Number of edits (change, removal, extension) of the feature / file within a release.	*
	Scope of modifications (MODS)	Number of edited features / files within a release (feature or file overlapping value). Idea: The more features / files have been edited in a release, the more error-prone they seem to be.	*
Code metrics	Lines of code (NLOC)	Average number of lines of code of the files associated with the feature / ~file within a release.	*
	Cyclomatic Complexity (CYCO)	Average cyclomatic complexity of the files associated with the feature / file within a release.	*
	Number of added lines (ADDL)	Average number of lines of code added to the files associated with the feature / file within a release.	*
	Number of removed lines (REML)	Average number of lines of code deleted from the files associated with the feature / from the file within a release	*
** These values were calculated based on the metadata obtained with PyDriller. Feature-level metrics were calculated based on the metadata of the underlying files.*			