Defect Prediction in Software Using Predictive Models Based on Historical Data

Daniel Czyczyn-Egird(✉) and Adam Slowik

Department of Electronics and Computer Science, Koszalin University of Technology, Sniadeckich 2 Street, 75-453 Koszalin, Poland
daniel.czyczyn-egird@cicomputer.pl,
aslowik@ie.tu.koszalin.pl

Abstract. Nowadays, there are many methods and good practices in software engineering that aim to provide high quality software. However, despite the efforts of software developers, there are often defects in projects, the removal of which is often associated with a large financial effort and time. The article presents an example approach to defect prediction in IT projects based on prediction models built on historical information and product metrics collected from various data repositories.

Keywords: Data mining in software · Defect prediction models
Software metrics

1 Introduction

In the era of the constant development of computers and software, there is great need for new and increasingly advanced IT systems, which require, in addition, certain functionalities, as well as the highest level of reliability. Unfortunately, the vast majority of software is burdened with defects causing unstable operation of certain functionalities or can cause a malfunction of the entire system. The defect appears in the software when the person creating the system makes a mistake that may occur at various stages of software development, such as requirements analysis, system documentation design (general/detailed design), test plan, inappropriate test scripts and source code, etc.

Therefore, one important aspect is the testing process, during which the tester performs specific test cases and can observe whether the results of these tests coincide with expectations. Any deviations from expectations are treated as incidents that need to be checked and explained. All defects and problems detected should be saved to issue tracking systems (ITS) and/or version control systems (VCS) for further analysis and finding an attempt to solve the problem.

Data repositories in which the above information is stored can be an interesting source of knowledge for researchers and developers, who deal with the issues of software improvement processes (SPI) [1] or quality assurance (QA).

The article aims to present a general approach to the problem of predicting software defects. It is based on models operating on historical data from repositories and analysing data based on predictive classifiers.

2 Related Works

There are many tools supporting the work of programmers and testers in their daily work on information systems. Among these tools are those that allow for convenient and quick testing of solutions created both at source code and post-compilation levels. Finding errors in the software is extremely important to ensure the final delivery of products without defects. However, continuous testing and debugging of systems involves spending on the use of human resources (programmers, testers) as well as financial resources [2]. Research on the prediction of defects has thus generated more and more interest on the part of both practitioners and researchers.

Ramler and Himmelbauer [3] have proposed prediction of defects using predictive models associated with software systems at the level of their modules. The modules can be files, classes and components, as well as subsystems of a given system. These modules are described by sets of attributes (e.g. by code metrics or number of changes in a given iteration), which are available by extracting them from various data sources such as databases or source code repositories.

There are also many studies in which the authors devote a lot of time to the issues of data acquisition from repositories, in addition to focusing only on models. Several tools have been developed for tasks related to data acquisition and supporting predictive modelling (e.g. defect prediction).

Jureczko and Magot [4] have prepared a *QualitySpy* [5] open-source framework (Apache 2.0 license [6]) whose task is to read and collect raw data from source code and event repositories, as well as user-defined metrics. Their project has focused on two modules for data acquisition and reporting. In the last released version, the framework allowed metrics to be read in classes for Java technology and reading JIRA [7] system events, as well as entries from the Subversion (SVN) [8] – version control system.

D'Ambros and Lanza [9] have suggested a tool to support collaborative software evolution analysis through a web interface going by the name of – *Churrasco*. It is an open source tool that retrieves and processes data from the Bugzilla and SVN systems, based on the FAMIX meta-model, which is independent of the programming technology used. In addition, the object relational mapping module (GLORP), the fact extraction module (MOOSE) and the SVG module for visualisation were used.

Madeyski and Majchrzak [10] have developed *Defect Prediction for software systems (DePress)*, a special framework that aims to extend the measurement of software and data integration for predictive purposes (defect prediction, cost/effort prediction). The *DePress* framework is based on the KNIME [11] project and extends it through a set of plugins, enabling models of data flow to be built in a graphical, simple and transparent way for the user. The main assumption of the project was prediction of software defects based on historical data. For this purpose, a set of plugins was prepared responsible for collecting, transforming and analysing data. The authors focused on data acquisition, data transformation and reporting operations, while at the same time leaving statistical operations and data mining [12] to a proven KNIME environment that has appropriate built-in mechanisms.

One important aspect upon which the authors placed emphasis is archiving and sharing data sets outside (e.g. for other researchers who wanted to test their own solutions). In this type of research, it is vital that repositories of historical data and prepared predictive models are as public as possible (after an earlier process of anonymization of content protected by commercial copyright).

3 Defect Prediction Using Predictive Models Based on Historical Data

3.1 Defect Prediction and Software Development Life Cycle

The defect can be introduced at any stage of the process called SLDC (Software Development Life Cycle) [13]. Therefore, it is very important that the testers are involved from the beginning of the life cycle of the software to detect and remove defects.

The sooner the defect is located and repaired, the cost of maintaining the quality will be lower. For example, if the defect is identified in the requirement analysis phase, then the cost of repair will be reduced to modify the requirements on the appropriate document. However, if the requirements are incorrectly described and implemented, and the defect is detected only during the testing phase, then the cost of the repair will be very high and will involve the improvement of the requirements and specifications as well as the change in the implementation.

It will also require a further testing process. In this article, the authors focus on the implementation and testing phase because, in relation to the level of these phases, it is possible to obtain relevant historical data from version control systems (VCS) and issue tracking systems (ITS), as long as these are stored and maintained.

3.2 General Assumptions

One of the main assumptions of the defect prediction operation is to determine the sources of historical data on the basis of which the entire prediction process will take place. The data in which we are interested can be obtained from software configuration management systems and error tracking systems.

There are currently many types of these systems on the market. Those proving to be most popular and most frequently used in professional programming teams are, respectively, the top five version control systems: Subversion, Git, CVS, Mercurial, Perforce, and the top five for bug tracking systems: JIRA, Github, Redmine, Bugzilla, BitBucket (study conducted by RebelLabs based on surveys in over 100 IT companies) [14].

Acquisition of data from the aforementioned systems can take place in two ways: directly or indirectly. The direct approach allows access to system repositories most often by connecting to their database or by means of appropriate mechanisms that allow reading information from these databases. In the case of the JIRA system, in addition to parametrized SQL queries directly executed on the appropriate tables in the database (the database engine supporting the JIRA system is MSSQL), there is also an API that returns the expected results using appropriate requests.

JIRA also offers its own micro-language JIRA Query Language (JQL). It's the most flexible way to search for issues in JIRA and can be used by everyone: developers, testers, project managers, and even non-technical business users. This method can be dedicated to those who have no experience with database queries, as well as those who want faster access to information in JIRA. The second way to access data is based on exchange files, which are exported manually from systems through appropriate interfaces, e.g. to CSV or XML format.

However, analysis of exported files may be the easiest and most frequently used way to access data if, for example, due to the lack of rights, we do not have direct access to the system and/or its database. In the case of version control systems, the same two options are available for selection. For example, for Git or Subversion, we can also try to connect to their (file) databases and search for artifacts that interest us or use data to export data to swap files.

The downloaded historical data should be unified in some way and cleared of unnecessary fields (e.g. from the information about the authors of a given entry) before being transformed into a common format, e.g. a tabular form, which will be explored in subsequent stages.

In addition to the historical data downloaded, source code metrics should also be used. These metrics should be appropriately linked to historical data to enable the construction of prediction classifiers. Without including the relevant metrics (e.g. number of lines of code in the file, the number of classes in the file and the degree of class nesting), it would not be possible to apply the predictive model to newly created systems where historical data does not exist.

3.3 An Exemplary Approach to Creating a Prediction Model

One of the methods of building a predictive model is the two-stage approach. In the first stage, we build and trial our model, while in the second stage it is used to predict the new input data. In each of the two stages, we can distinguish several of the following phases (Fig. 1):

1. Data acquisition phase – data collection processes.
2. Data transformation phase – processes of unification and matching of input data.
3. Data mining phase – processes related to the discovery of new knowledge.
4. Reporting phase – processes of presenting results and their archiving.

From the research point of view, a vital element for further research, if the most difficult one, is the selection of appropriate classifiers in the third phase. For this purpose, different techniques are used from areas related, for example, to evolutionary algorithms, swarm intelligence algorithms [15], artificial neural networks or fuzzy logic.

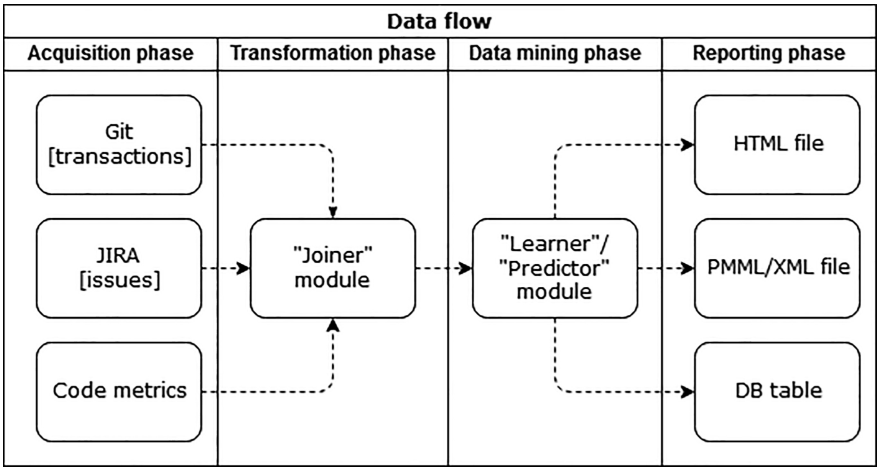

Fig. 1. Data flow in defect prediction process - diagram

4 Example of Defect Predictions

4.1 Selection of the Project for Analysis

For the following study, one of the main criteria for the selection of analysis software was free access to archival data contained in version control systems and error tracking systems. This criterion is met by open source projects; here, it was decided to choose the Apache Maven [16] system. It is a tool that automates software development for the Java platform. The various functions of Maven are realised through plug-ins, which are automatically downloaded at their first use. The file defining the method of building the application is called POM (Project Object Model). Just like other Apache products, Maven is distributed under the Apache License. This tool was first released in July 2004 and is constantly being developed; in fact, its latest release dates back to October 2017 (version 3.5.2 for Java 7). Maven has been developed throughout its history by a large group of programmers - open source software supporters (about 100 people involved in the project over a dozen years, recent releases supported by 10 constantly working programmers). The fact of such an active development of the chosen tool is undoubtedly caused by the lack of limitations in the development of this software, along with unqualified access to source code repositories, in accordance with the Apache Foundation license policy.

Source code repositories are stored in Subversion and Git [17] systems, while the defects detected are reported in the JIRA system.

4.2 Data Acquisition

Maven source codes are stored both in the Subversion system and the Git system. However, recent times have shown the growing popularity of software developers affiliated with the Apache foundation to migrate their projects to the Git system. Thus, the choice was made to download data from the Git system. For this purpose, appropriate tools were used, among others, GraphQL API v4 [18] for Github (data query language) and a set of bash scripts operating on the Git system. The range of dates for these activities is January - December 2017. In Table 1, the general data information from Git are presented:

Table 1. General data information from Git

Time period	01.01.2017–31.12.2017
Release	3.5.0–3.5.2
Commits (Transactions)	752
Files changed	5053
No. of active developers	10

The next step was to acquire data from the JIRA system for a given period of time. From the JIRA repository, all the events were described as Bug with status Closed/Resolved, and final result Fixed/Duplicated. These were taken into account, although the priority parameter for the need to repair the defect was omitted to obtain a wider range of information about events. In Table 2, the general data information from JIRA are presented:

Table 2. General data information from JIRA

Time period	01.01.2017–31.12.2017
Release	3.5.0–3.5.2
No. of issues	175
No. of issues (type: Bug)	111
Improvement tasks	40

In the next stage, an attempt is made to pair artifacts taken from the Git system with artifacts taken from the JIRA system. Such pairing is possible if, for example, in transaction descriptions (Git) there are descriptions of solved problems reported in the JIRA system. For example, you should search (e.g. with the use of regular expressions) string with the number of a given event, e.g. “Resolved issue MNG-6221”, where the JIRA side describes the event: “(MNG-6221) Maven cannot build with Java 1.8.0_131”. The last stage of data acquisition is downloading the source code metrics. This can be performed using tools dedicated to the given programming technology. In the case of Maven for Java, you can use the JavaNCSS library.

4.3 Data Unification Process

During the transformation and normalization process, the data (columns) unnecessary from the prediction view point are deleted or the assigned values are set to zero. Each artifact that was downloaded in the previous step and related to each other receives a description of the properties *HasDefects* and *NoOfDefects*, with the appropriate values describing whether for a given class/method from the source code, there are any defects that were reported in the JIRA system and repaired in the next transaction to the version control system (Git).

4.4 Prediction Process and Results

The prediction methodology should be based on a minimum of three stages: selection of attributes for prediction, model construction and its validation. In the first stage, we select the attributes on which the model will be based. The use of all attributes (historical data and metrics) may lead to the re-loading of the model; therefore, it is recommended that only those that stand out be selected (you can use, for example, backwards elimination).

The next stage is the construction of a model that can be based on one classifier or several linked together, creating the so-called hybrid classifier. One popular classifier used for all predictions are decision trees. For a given case, the prediction attribute is *HasDefects*, which tells you if a defect has been found in the given input set. The input variables selected in the first stage are independent variables in this case.

As the last stage, validation of the model may be based, for example, on cross-validation (K-fold cross validation), where the input data is divided into two segments (learning and validating) and the entire process is repeated several times with random division (different each time).

While the results of the model work will be satisfactory, it can be applied to subsequent project releases, especially during the code creation and testing phase. All operations related to modeling and testing can be performed, for example, in the KNIME research environment, which has a set of modules supporting the techniques of creating predictive models. It also allows you to save results to a database, or export to XML as a PMML model (useful for further research).

5 Summary and Future Works

The article aims to present a general approach to the problem of predicting defects in software, based on prediction models operating on historical attributes obtained from version control systems, issue tracking systems and metrics obtained from the source code level.

A general mechanism of the defect prediction process based on the analysis of selected Apache Maven data repositories has been presented. This tool is characterized by a long presence on the IT market (since about 2004), so the version control systems (Git) as well as issue tracking systems (JIRA) contain a lot of interesting data for exploration. Based on them, a predictive model can be built, which after proper validation will be able to predict defects in the future in a given project. What will translate into measurable benefits in the form of time savings (testing) and financial resources (corrections).

The subject on effective defect prediction is a very developmental subject and raises the real, modern problems of the IT market, therefore further work is foreseen on prediction of defects. Future work plans should focus on own capabilities, competitive implementation of tools supporting acquisition, standardization and predictive modeling. In addition, there is a need to build more effective predictive models, for this purpose further interest may be focused, for example, on hybrid prediction classifiers.

References

1. Petersen, K., Wohlin, C.: Software process improvement through the Lean Measurement (LEAM) method. J. Syst. Softw. **83**(7), 1275–1287 (2010)
2. Wojszczyk, R.: Quality assessment of implementation of strategy design pattern. In: Advances in Intelligent Systems and Computing, vol. 620, pp. 37–44. Springer (2018)
3. Ramler, R., Himmelbauer, J.: Building defect prediction models in practice. In: Handbook of Research on Emerging Advancements in Software Engineering, pp. 540–565 (2014)
4. Jureczko, M., Magott, J.: QualitySpy: a framework for monitoring software development processes. J. Theor. Appl. Comput. Sci. **6**(1), 35–45 (2012)
5. Jureczko, M. and Contributors: Quality Spy. http://java.net/projects/qualityspy
6. The Apache Software Foundation, Apache License, Version 2.0. http://www.apache.org/licenses/LICENSE-2.0.html
7. JIRA. Atlassian. https://www.atlassian.com/software/jira
8. SVN. Enterprise-class centralized version control. https://subversion.apache.org/
9. D'Ambros, M., Lanza, M.: Distributed and collaborative software evolution analysis with churrasco. Sci. Comput. Program. **75**, 276–287 (2010)
10. Madeyski, L., Majchrzak, M.: Software Measurement and defect prediction with depress extensible framework. Found. Comput. Decis. Sci. **39**(4), 249–270 (2014)
11. Berthold, M.R., Cebron, N., Dill, F., Gabriel, T.R., Meinl, T., Ohl, P., Sieb, C., Thiel, K., Wiswedel, B.: KNIME: the konstanz information miner. In: Studies in Classification, Data Analysis, and Knowledge Organization (GfKL 2007). Springer (2007)
12. Czyczyn-Egird, D., Wojszczyk, R.: The effectiveness of data mining techniques in the detection of DDoS attacks. In: 14th International Conference on Distributed Computing and Artificial Intelligence, vol. 620, pp. 53–60. Springer (2018)
13. Kazim, A.: A study of software development life cycle process models. Int. J. Adv. Res. Comput. Sci. **8**(1) (2017)
14. RebelLabs. Developer Productivity Report 2013 – How Engineering Tools & Practices Impact Software Quality & Delivery. http://bit.ly/2nQVSFh. Accessed Feb 2018
15. Slowik, A., Kwasnicka, H.: Nature inspired methods and their industry applications - swarm intelligence algorithms. IEEE Trans. Ind. Inform. **14**(3), 1004–1015 (2018)
16. Apache Maven. https://maven.apache.org/
17. GitHub Inc. http://www.github.com
18. GraphQL API v4. https://developer.github.com/v4/