Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Adam Wierzbicki
Nr albumu: 306441

Using Code History for Defect Prediction

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
prof. dr. hab. Krzysztofa Stencla
Instytut Informatyki

Czerwiec 2015

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Streszczenie

Detection of software defects has become one of the major challenges in the field of automated software engineering. Numerous studies have revealed that mining data from repositories could provide a substantial basis for defect prediction. In this thesis I introduce my approach towards this problem relying on the analysis of source code history and machine learning algorithms. I describe in detail the proposed computational procedures and explain their underlying assumptions. Following the theoretical basis, I present the results of performed experiments which serve as an empirical assessment of the effectiveness of my methods.

Słowa kluczowe

code history, defect, bug-proneness, prediction, repository, metrics, machine learning

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

D. Software
D.2. Software Engineering
D.2.8 Metrics
D.2.9 Management

Tytuł pracy w języku polskim

Wykorzystanie historii kodu do predykcji błedów

Table of contents

List of tables

List of figures

List of code snippets

Chapter 1

Introduction

Ever since the legendary first bug in 1947[1], the detection of software faults has been a crucial part of the quality assurance process. Undiscovered bugs were the cause of many misfortunes with the crash of the $500-million space rocket Ariane 5 being the most spectacular example.[2] This work aims to contribute to the general improvement of software quality by investigating certain methods of error identification.

1.1. Topic choice rationale

Various procedures have been applied to track down software defects before they would cause any problems. Most widely used techniques are testing and code reviewing. Both of them are quite successful, but unfortunately also very arduous. In order to improve the performance of these methods and reduce the programmers' effort needed to conduct them, automatic debugging programs are developed.

Such programs include a broad range of approaches towards detecting bugs. Some of them are run-time debuggers which help programmers analyse the execution of a program (either standalone as GDB [29] or integrated with IDEs as Microsoft Visual Studio Debugger [21]). Others generate test cases using randomization and symbolic execution [8, 14]. Others support the reviewing process by highlighting potentially dangerous program parts.

Identification of such fault-prone elements could be performed using many different methods. There are tools which rely on hard-coded patterns of so-called "bad code smells" (*e.g.* FindBugs [16]). Other ones incorporate statistical analysis and machine learning, using various software metrics and properties (*e.g.* HATARI [71]). In this work I focus on the latter approach, because I find it more interesting and less frequently encountered. I decided to use historical metrics[3] which have proven to be well-suited for this task [12, 58, 64]. With my research I hope to extend the knowledge about methods of bug prediction, which is not only of theoretical but also of practical value. This thesis makes the following contributions:

- A proposal of a novel method for bug prediction,

[1] According to [38] on 9th of September, 1947 an investigation of malfunctioning Harvard University Mark II Aiken Relay Calculator revealed a moth trapped between the points of relay #70 in panel F. This event was reported in a log with the following statement: *"First actual case of bug being found."*

[2] On 4th of June, 1996 the rocket was launched by the European Space Agency from Kourou in French Guyana. It exploded only about 40 seconds after take-off due an error in the inertial reference system – conversion of a 64-bit float to a 16-bit signed integer failed, because the number was larger than the largest storable value. [6]

[3] Software metrics based on code change history. For details see Section 2.5.2.

- A prototype implementation of the proposed method,
- An experimental evaluation of the implemented tool.

1.2. Document structure

Chapter 1 includes general introductory information for the thesis, topic choice rationale, and a sketch of the document structure.

Chapter 2 contains a broad and comprehensive description of the problem of bug prediction. I try to define in a possibly precise manner all terms relevant to this topic. Then I present the general goals of the prediction process and its consecutive phases. I describe several kinds of sub-problems which have to be solved in order to successfully develop an error detection instrument.

Chapter 3 presents my approach towards the problem. The proposed method is based on extraction of historical information from a version control system and application of a machine learning algorithm. I describe the general model of a changing method[4], which is the principal element of the whole procedure, as well as successive phases such as feature extraction, identification of bug-fixes, assignment of bug-proneness scores and training a regressor. In the last section of this chapter, I list four different machine learning algorithms which are evaluated in Chapter 5.

Chapter 4 describes the ChangeAnalyzer tool – an implementation of the approach proposed in Chapter 3. I present all used technologies (languages and libraries), software architecture and the interface.

Chapter 5 contains a description of the experiments which were performed to evaluate the proposed approach and compare the performance of different algorithms listed in Section 3.4. I explain the methodology of these experiments and the software configuration used to conduct them. I give an overview of the experimental data and finally present the results with a short commentary.

Chapter 6 includes a general assessment of my approach towards the problem of bug prediction. Further, I describe several threats to the validity of my research, and in the last section, I present the possible ways of continuation of my work on this topic.

[4] In the object-oriented-programming sense

Chapter 2

Problem overview

Simply speaking, bug prediction is a procedure aimed at automatic detection of faults in an unreviewed and untested code, basing on reports about previously discovered bugs. However, this deceptively plain definition hides a very convoluted and problematical process. In this chapter I would like to explore its meanders and formulate a more precise description by answering questions such as: ***What is it*** *exactly that we are predicting?* (Section 2.2) *What is* ***the scope*** *of our prediction?* (Section 2.3) *How do we learn about* ***former defects****?* (Section 2.4) and *What* ***kinds of information*** *can we utilize in our prediction?* (Section 2.5)

2.1. Terminology

Before referring the topic, I would like to define some concepts of major importance, namely: *version control system*, *software repository*, *commit*, *error*, *bug-fix*, *bug-tracker*, *software metric*, and *error model*. Below are given brief explications of how I understand these terms.

Version control system
: "A system capable of recording the changes made to a file or a set of files over a time period in such a way that it allows us to get back in time from the future to recall a specific version of that file" [73, p. 8]. Usually, such system is used to manage the source code of a software project. Examples of VCSs are: Apache™ Subversion® [3], Git [32], or Mercurial [55].

Software repository
: A directory containing the source code of a software project, which is managed by a version control system. It could be either local or remote (accessed via HTTPS or SSH protocol).

Commit
: "An atomic collection of changes to files in a repository. It contains all recorded local modifications that lead to a new revision of the repository" [53]. Apart from file changes, a commit contains meta-information such as: committer's name and e-mail address, date and time, and a description of changes. Commit is also often called *changeset*.

Error
: "An incorrect step, process, or data definition. For example, an incorrect instruction in a computer program" [39, p. 31]. Error is also called *bug*, *fault*, or *defect*.

Bug-fix
A commit created in order to repair a software defect. It is usually distinguished by an appropriate commit message describing the fixed error.

Bug tracking system
A system that "stores and manages all bug status and information such as the module where a bug occurred, when a bug was found, the severity of a bug, comments describing a bug's effect on the software, instructions on replicating a bug, who reported the bug, and whether the bug has been fixed yet" [45]. Also called simply *bug-tracker*. Examples of bug-trackers are: Bugzilla [37] and Jira [43].

Software metric[1]
"A quantitative measure of the degree to which a system, component, or process possesses a given attribute" [39, p. 47-48].

Error model
"In software evaluation, a model used to estimate or predict the number of remaining faults, required test time, and similar characteristics of a system" [39, p. 31].

2.2. Dependent variable

Prediction in general may be described as discovering a relation between several observable variables and a dependent variable (which may also be called *target variable*). The discovered relation allows for estimating values of the target variable by analysing the observables. Trying to conform *defect prediction* to this description seems problematic, because "defect" does not name any well-defined variable. Therefore, the first fundamental concern of bug prediction is specifying a variable which will both suit our theoretical framework and embrace the common-sense intuitions.

The most obvious candidate (used *e.g.* by Janes *et al.* in [40]) is the number of defects per software unit[2] (for possible units see Section 2.3). However, as in many cases the numbers of bugs in software components are relatively small, a dichotomous boolean variable (indicating bug presence or absence) is used more often (*e.g.* by Moser *et al.* in [58] or Giger *et al.* in [30]). More complex examples include *fault density* – number of faults divided by a code size measure (Tomaszewski *et al.* [76]), or expected number of repairs in some period of time (Hassan [35]).

As Arisholm *et al.* reasonably state in [5, p. 5], the choice of dependent variable should be determined by the further usage of the developed model. For assigning a software component for a detailed examination by a reviewer, a boolean variable is probably sufficient. On the other hand, creating a ranking or visualisation of the software quality requires numeric values, so number of defects or fault density would suit these purposes much better.

2.3. Unit of analysis

All of the variables defined in the above section are relative to some code segment. Size of such segment can span from a single function up to a whole project – every possibility has its

[1] I prefer the term "measure" to be used in this context instead of "metric", because it is more consistent with the mathematical understanding of measure and metric. However, "software metric" has become a widely used phrase in software engineering, so I will incline to this tendency.

[2] [39, p. 31] endorses the choice of this kind of variables by defining error prediction as "a quantitative statement about **the expected number** or nature of faults in a system or component" (emphasis added).

advantages and drawbacks. As nine of ten most popular programming languages are object-oriented [75, 59], also the most commonly encountered units of analysis are related to OOP paradigm: module/package, class, or method/function.

Package-level prediction (used *e.g.* by Jin *et al.* in [42]) can make use of structural software properties which cannot be computed for smaller units. However, the typical size of a package in some languages would make the corresponding fault-proneness value of very little value for practical application. Therefore, module- or package-level prediction is usually restricted to languages to which the class level does not apply, *e.g.* Fortran, Pascal or Ada.

Class level (used *e.g.* by Tomaszewski *et al.* in [76]) is the most popular choice among defect prediction studies. Probably, the reason for this popularity is the fact that class is the most natural and accustomed part of object-oriented software. In case of non-object-oriented languages, the middle-sized unit of analysis is file, which is easily separable and trackable with a VCS machinery. Class shares the advantages of file, since numerous software projects obey, either by virtue of conventional guidelines or due to language constraints, the *one class per file* rule. The scope of a class or a file is also usually narrow enough to serve as an object of manual inspection in case of high bug-proneness score.

Using method or function as the unit of analysis is a more complicated task, but it has been also successfully accomplished (*e.g.* by Giger *et al.* in [30]). Contrary to higher-level units described above, which can be identified using solely lists of files, ascribing faults to certain methods requires examining detailed information about code changes. I consider the results of method-level prediction most useful for debugging due to their fine granularity.

The scope of prediction is limited not only *spatially* (by which I understand focusing on particular software components, as described above in this section), but also *temporally*. One may predict the number (or presence) of bugs in a software release, over some fixed period of time, or in a single commit. This choice should also be affected by the later utilisation of predicted values.

2.4. Identification of bugs

Prediction of new bugs has to be based on historical data about the heretofore detected ones. However, identification of these is not a trivial task. It has become one of the major challenges in the discipline of *mining software repositories*.

2.4.1. Bug-fixes

First step towards identification of bugs is the recognition of bug-fixing commits. Traditionally used heuristic (*e.g.* by Kim in [45]) involves searching for keywords such as “bug”, “issue” or “fix” in commit messages. A more refined approach (used in [25, 30, 45]), instead of relying on hard-coded patterns, utilizes information obtained from a bug-tracking system. Bug-trackers assign IDs to the tracked issues, which could be later referenced in change logs. As the vast majority of contemporary software projects uses bug-trackers, searching for these IDs in commit messages could therefore constitute a reliable method for detecting bug-fixes.

2.4.2. Noise in error data

Unfortunately, research has shown that the aforementioned methods are not quite successful. Herzig *et al.* [36], after analysing over 7,000 error reports, make the following statements:

- “More than 40% of issue reports are inaccurately classified”,

- "33.8% of all bug reports do not refer to corrective code maintenance",
- "Due to misclassifications, 39% of files marked as defective actually have never had a bug".

Bachmann *et. al* in [7] confirm the poor quality of error reporting (particularly in Apache project), by declaring that "only 47.6% of bug fix related commits are documented in the bug tracking database". Human failure and negligence makes the error data obtained from repositories highly biased an untrustworthy. Which in turn negatively affects bug localization and prediction [36, 46, 47].

In order to improve the efficiency of fault detection, Wu *et al.* have developed ReLink – an automatic link[3] recovery algorithm [80]. It uses machine learning approach with features such as:

- The interval between the bug-fixing time and the change-commit time,
- Mapping between bug owners and change committers,
- The similarity between bug reports and change logs.

Evaluated on five open-source projects, ReLink achieved up to 26.6% more recall than the traditional heuristics.

2.4.3. Bug-introducing changes

The next stage, after identifying bug-fixes, is linking them with *bug-introducing changes*. No earlier than this is achieved, one can give the precise temporal location of an error. Several algorithms have been developed for this purpose. Probably the most popular one (used *e.g.* by Kim [45] and Fukushima *et al.* [28]) is the SZZ algorithm developed by Śliwerski, Zimmermann & Zeller in [72]. It relies on annotation graphs produced by CVS[4] *annotate*[5] command. This approach was criticised by in [79] by Williams & Spacco, who propose a different technique using line-number maps and DiffJ – a Java syntax-aware diff tool. Application of such algorithms allows to conclude with high certainty, which parts of software at which point of time are to be considered fault-prone.

2.5. Observable variables

Prediction of the target variable has to be done through a measurement of some observable variables. A multitude of software metrics has been applied for this purpose. In this section I will present a condensed survey of the most common ones (and some less common, but interesting).

2.5.1. Static metrics

Static metrics are "measures of structural properties derived from the source code" [5, p. 5]. Their essential property (responsible for the name) is that they can be computed using only a snapshot (revision) of the code. Some of static metrics are specific to object-oriented software,

[3]The term "link" refers here to the links between bug reports and code changes.

[4]Concurrent Versions System is an open-source version control system [20].

[5]The *annotate* command allows to "print the head revision of the trunk, together with information on the last modification for each line" [19]. Its counterpart in Git and SVN is *blame*.

other are more universal (*e.g.* LOC or complexity). Examples of static metrics are listed in Table 1. Please note, that metrics presented as low-level could be also used on higher levels by aggregation operations (*e.g.* minimum, maximum or average).

Table 1: Static code metrics

Level	Variable	Used by
Method/ function	Number of parameters	[30]
	Number of local variables	[30]
	Maximum depth of nested control sequences	[30, 64]
	Number of possible paths in the method body	[30, 64]
	Number of methods referenced by a given method	[30, 64]
	Number of methods referencing a given method	[30, 64]
Class	Number of attributes	[5]
	Number of methods	[5, 50]
	Number of implemented interfaces	[5]
	Number of imported packages	[5]
	Number of subclasses	[5, 50, 64]
	Number of superclasses	[5, 64]
	Depth of inheritance tree	[50, 64]
	Lack of cohesion of methods[6]	[5, 50, 64]
	Message passing coupling[7]	[5]
Universal	Lines of code	[5, 50, 64]
	Number of statements	[5, 30, 50, 64]
	Ratio of comments to source code	[30, 64]
	McCabe cyclomatic complexity[8]	[5, 30, 64]
	Halstead effort[9]	[5]

[6] LCOM = $max\,(|P| - |Q|, 0)$, where P is the set of all pairs of methods which do not share any fields and Q is the set of all pairs of methods sharing least one field [15, p. 488].

[7] "The MPC measures the number of method calls defined in methods of a class to methods in other classes, and therefore the dependency of local methods to methods implemented by other classes" [56].

[8] CC = $E - N + P$, where E is the number of edges in a *program control graph*, N is the number of nodes and P is the number of connected components [54].

[9] It is "the *total number of elementary mental discriminations* required to generate a given program", which can be described by the following equation: $E = V^2/V^*$, where V is the program volume and V^* is the minimal possible program volume [34, p. 47].

2.5.2. Historical metrics

As the name suggests, historical metric are obtained through the analysis of software history, which is stored in a version control system. Some of them are computed from raw change data (that is a series of code revisions), while others rely on meta-information stored by VCS such as commit times, authors, and descriptions. Historical metrics are also called *process metrics* or *change metrics*[10]. Table 2 contains examples of such metrics.

Table 2: Historical code metrics

Type	Variable	Used by
Raw changes	Number of commits	[5, 12, 30, 50, 58, 64]
	Total/average/max number of added lines	[5, 12, 50, 58, 64]
	Total/average/max number of deleted lines	[5, 12, 50, 58, 64]
	Total/average/max number of added statements	[30]
	Total/average/max number of deleted statements	[30]
	Total/average/max *code churn*[11]	[30, 50, 58]
Meta--information	Number of bug-fixes	[5, 50, 58]
	Number of refactorings	[5, 50, 58]
	Number of distinct authors	[5, 12, 30, 50, 58, 64]
	Number of lines added by file owner	[64]
	File owner's experience	[64]
	File age	[50, 58]
	Weighted file age[12]	[50, 58]

2.5.3. Micro-interaction metrics

Micro-interaction metrics, proposed by Lee *et al.* in [50], are tracking behavioural interaction patterns of software developers. Usage of such metrics is motivated by research showing correlations between work habits and productivity [48]. They can be obtained from systems tracking developers' activities, such as Mylyn [60], a task management plug-in for Eclipse IDE. Experimental results presented in [50] show that micro-interaction metrics outperform both static and historical metrics. However, the scope of their usage is very limited, due to requirement for programmers' activities registration, which is not so common in software projects. Examples of micro-interaction metrics are presented in Table 3.

[10]However, in [28] this last name is used for metrics which can be attributed to a *change itself*, not to a software component.

[11]The difference between numbers of added and deleted lines [58] or statements [30].

[12]It is the weighted arithmetic mean of commit ages, where weights are proportional to the numbers of added lines in corresponding commits.

Table 3: Micro-interaction metrics

Level	Category	Variable
File	Effort	Number of selection events of the file
		Number of edit events of the file
	Interest	Average *degree of interest*[13] for the file
		Variance of *degree of interest* for the file
	Intervals	Average time interval between edit events of the file
		Average time interval between selection events of the file
Task	Effort	Total time spent on the task
		Number of unique selected files during the task
		Number of unique edited files during the task
	Distraction	Number of selection events with low *degree of interest*
		Number of edit events with low *degree of interest*
	Work portion	Ratio of time spent (out of total time) before the first edit
		Ratio of time spent (out of total time) after the last edit
		Number of unique selections before the first edit
		Number of unique selections after the last edit
	Repetition	Number of files selected more than once in the task
		Number of files edited more than once in the task
	Task load	Number of ongoing tasks at the same time
		Average depth of the file hierarchy

[13] *Degree of interest* (DOI) is a parameter increasing on programmer's interactions and decreasing with time, thus measuring the interest in a particular file or code element. It was introduced by Kersten & Murhpy in [44] together with an implementation named Mylar.

Chapter 3

Proposed approach

Having referred the general goals and issues connected with defect prediction, I would like to present my own approach towards this problem. I decided to perform the prediction on method level, which is the finest granularity reached by state-of-the-art algorithms. Similarly to [30], I chose to use historical metrics based on fine-grained source code changes. However, instead of relying on sparse software releases as time scope boundaries, I developed a model allowing for a more continuous prediction.

Underlying assumptions and a general outline of the **model** is presented in Section 3.1. Then, in Section 3.2, a detailed summary of the model's **observable variables** is given. Section 3.3, on the other hand, describes the **target variable**. Section 3.4 explains four different **machine learning algorithms** which can be applied to the model.

3.1. Model

3.1.1. General assumptions

Design of the further described model relies on the following fundamental assumptions:

(A1) Every method, in every moment can be assigned a certain ***bug-proneness*** level ranging between 0.0 and 1.0 (these values are arbitrary and do not denote any units).

(A2) **Immediately before a bug-fix**, the method to be fixed is **undoubtedly deficient**, therefore its bug-proneness equals 1.0.

(A3) **Immediately after a bug-fix**, the fixed method is **perfectly correct**, therefore its bug-proneness equals 0.0.

The above presumptions may be criticized in many ways as not accurately reflecting the reality. However, they were not intended to do so, but rather to form an idealization which would allow for a possibly simple modelling of the quality of software methods.

The bug-proneness parameter constitutes the target variable in my approach. As it is not precisely defined, one could use different methods of measuring this value. These are described later, in Section 3.3. An example of how bug-proneness score can change with succeeding commits is presented in Figure 1.

3.1.2. Instance construction

Since my aim is to follow changes affecting the analysed method, model instances are constructed from succeeding commits. However, as the mentioned changes are *cumulative*, each

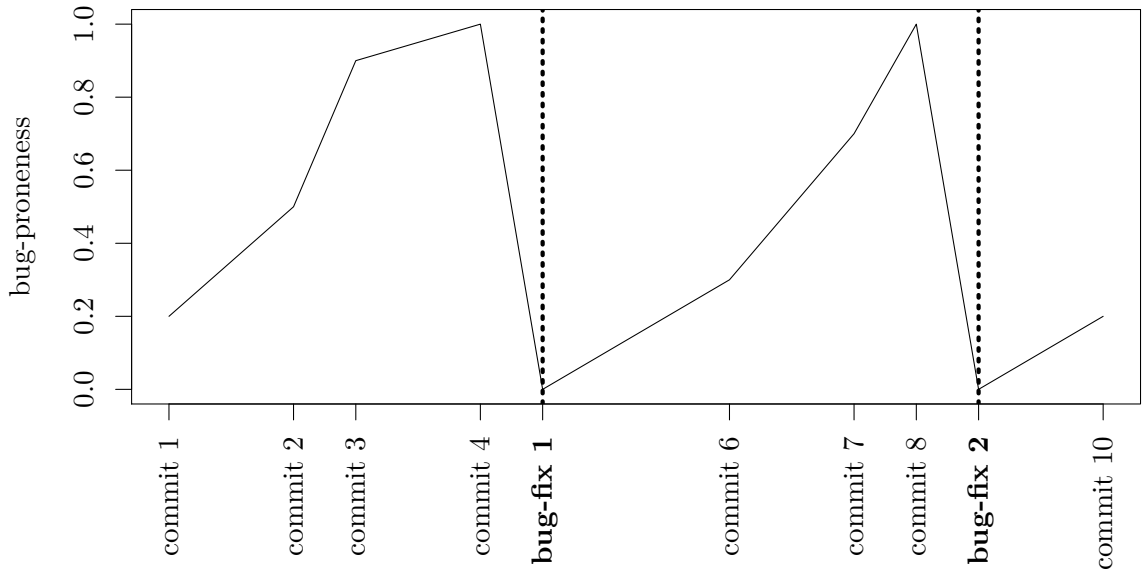

Figure 1: Bug-proneness alteration

instance corresponds to a whole *chunk* of commits, rather than to single one. In virtue of assumptions A2 and A3, the maximal time scope of an instance is bounded by two adjacent bug-fixes. This means that instances range in size from single commits to continuous groups delimited by bug-fixes. Example of instance construction is presented in Figure 2.

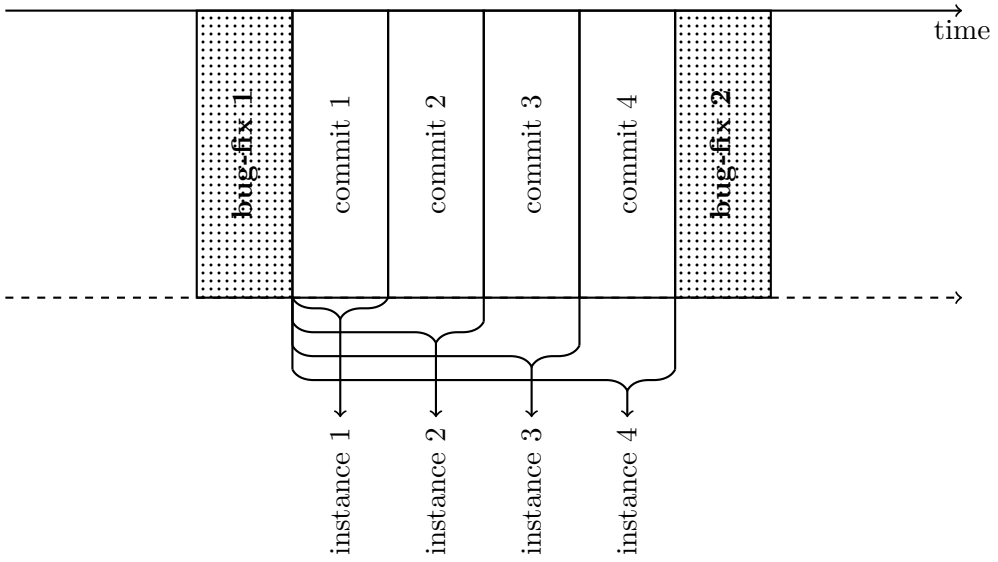

Figure 2: Instance construction

3.2. Features

In order to be useful for prediction of the target variable (bug-proneness), the model incorporates several observable variables. I decided to use historical metrics (described in Section

2.5.2), which are considered superior to static measures (Section 2.5.1) by many researchers [12, 58, 64] and at the same time are more widely applicable than micro-interaction metrics (Section 2.5.3).

Opposed to most works using historical metrics (*e.g.* [5, 12, 50, 58, 64]), which rely on standard source code differencing utilities[1], I employ a syntax-aware *change distilling* algorithm developed by Fluri *et al.* in [27], described in more detail in the next section. A similar approach was taken by Giger *et al.* in [30].

3.2.1. Change distilling

Change distilling algorithm, presented in [27] aims to provide a smart, syntax-aware method of differencing source code files. It extracts *fine-grained source code changes* (down to the level of single statements) by matching the *abstract syntax trees*[2] representing the compared files.

Leaves of the ASTs are matched using the *bi-gram Dice coefficient* – an adaptation of an ecological association measure (proposed by Dice in [22]) to the text distance measurement. The similarity of inner nodes is computed as a combination of node values similarities and subtrees similarities. After matching the appropriate nodes, the algorithm calculates the minimal number of *insert*, *delete*, *move*, and *update* operations required to transform the first version of the AST into the other. These elemental operations constitute fine-grained source code changes, which are later classified using a taxonomy presented by Fluri and Gall in [26]. A summary of change types is given in Table 4, while Snippets 1 to 6 contain examples of classified changes.

Table 4: Fine-grained source code changes

Location	Change type	Significance[3]
Body	Additional functionality	low
	Additional object state	low
	Condition expression change (Snippet 1)	medium
	Decreasing statement delete[4]	high
	Decreasing statement parent change[4]	high
	Else-part delete	medium
	Else-part insert (Snippet 2)	medium
	Increasing statement insert[4]	high

[1] Such as `diff` and `diff3` provided by GNU DIFFUTILS package [23].

[2] Abstract syntax tree (AST) is the "hierarchical syntactic structure" of a sentence belonging to some formal language (*e.g.* a programming language). "Every internal node of a parse tree is labeled with a nonterminal symbol; every leaf is labeled with a terminal symbol. Every subtree of a parse tree describes one instance of an abstraction in the sentence." [68, p. 121] An abstract syntax tree is also called a *parse tree*, for it is generated by an act of *parsing*.

[3] "The significance level of a change is defined as the impact of the change on other source code entities, i.e., how likely is it that other source code entities have to be changed, when a certain change is applied." [26]

[4] *Decreasing* and *increasing* refers here to the change of the *overall nested depth* of a method.

Location	Change type	Significance
Body	Increasing statement parent change[4]	high
	Removed functionality	crucial
	Removed object state	crucial
	Statement delete	medium
	Statement insert	medium
	Statement ordering change	low
	Statement parent change (Snippet 3)	medium
	Statement update	low
Declaration	Attribute renaming	high
	Attribute type change (Snippet 4)	crucial
	Class renaming	high
	Decreasing accessibility change	crucial
	Final modifier insert	crucial
	Final modifier delete	low
	Increasing accessibility change	medium
	Method renaming	high
	Parameter delete	crucial
	Parameter insert	crucial
	Parameter ordering change (Snippet 5)	crucial
	Parameter type change	crucial
	Parameter renaming	medium
	Parent class delete	crucial
	Parent class insert	crucial
	Parent class update	crucial
	Return type delete	crucial
	Return type insert (Snippet 6)	crucial
	Return type update	crucial

Snippet 1: Condition expression change

```
static int abs(int x) {
    if (x < 0) {
        return x;
    } else {
        return -x;
    }
}
```

```
static int abs(int x) {
    if (x >= 0) {
        return x;
    } else {
        return -x;
    }
}
```

Snippet 2: Else-part insert

```
void printSqrt(int n) {
    if (n < 0) {
        System.out.print("-");
    }
    int sqrt = sqrt(abs(n));
    System.out.print(sqrt);
}
```

```
void printSqrt(int n) {
    if (n < 0) {
        System.out.print("-");
    } else {
        System.out.print("+");
    }
    int sqrt = sqrt(abs(n));
    System.out.print(sqrt);
}
```

Snippet 3: Statement parent change

```
private int foo(int a) {
    if (this.isReady()) {
        this.startFoo(a);
        this.createX();
    }
    return this.getX();
}
```

```
private int foo(int a) {
    if (this.isReady()) {
        this.startFoo(a);
    }
    this.createX();
    return this.getX();
}
```

Snippet 4: Attribute type change

```
int pow(int x, int y) {
    ...
}
```

```
int pow(double x, int y) {
    ...
}
```

Snippet 5: Parameter ordering change

```
void foo(int[] a, Blob b) {
    ...
}
```

```
void foo(Blob b, int[] a) {
    ...
}
```

Snippet 6: Return type insert

```
void add(Object obj) {
    ...
}
```

```
boolean foo(Object obj) {
    ...
}
```

3.2.2. Feature engineering

Basing on fined-grained code changes (extracted by the algorithm mentioned in the previous section) as well as commit meta-information, each instance of my model is ascribed a number of features. As instances represent groups of commits, most features are computed by an aggregation of parameters of single commits (*e.g. via* averaging or summing). Some attributes, related to commits' authors, utilize not only local, but also global information. They require an evaluation of the authors' experience, which cannot be computed using only data related to a single method. All the features are presented in Table 5.

Table 5: The model's features

Type	Feature
Raw changes	Total number of condition expression changes
	Total number of else-part deletes
	Total number of else-part inserts
	Total number of statement deletes
	Total number of statement inserts
	Total number of statement ordering changes
	Total number of statement parent changes
	Total number of statement updates
Processed changes	Total number of header changes[5]
	Total number of parameter changes[6]

[5]Sum of the numbers of the following changes: decreasing accessibility change, increasing accessibility change, method renaming, return type change, return type delete, return type insert.

[6]Sum of the number of the following changes: parameter delete, parameter insert, parameter ordering change, parameter type change, parameter renaming.

Type	Feature
Processed changes	Average number of changes
	Average number of changed methods in a commit
	Average *change ratio*[7]
	Change *Gini index*[8]
Meta--information	Total number of commits
	Total number of authors
	Average number of commits made by an author
	Average number of changes made by an author
	Time in seconds since the last bug-fix to the latest commit

3.3. Bug-proneness estimation

For the sake of simplicity, I decided to use a plain textual match to identify the bug-fixing commits. Log messages are searched for one of the following strings: `"bug"`, `"fix"`, or `"issue"` (the matching is case-insensitive). The problem of identification of bug-fixes has been described in more detail in Section 2.4.1.

After identifying the bug-fixes, defect-proneness has to be estimated for the other commits. As the assumptions put forward in Section 3.1.1 do not impose any particular method of estimation, I propose three different strategies which could be used fo this purpose:

(S1) **Linear** – It is the simplest method, which assigns linearly ascending scores to successive commits. The common difference of this sequence is determined by the number of commits in a chunk (n).

$$\underset{1 \leqslant i \leqslant n}{\forall} lin(i) = \frac{i}{n}$$

(S2) **Geometric** – This strategy uses geometric progression of bug-proneness. As it is impossible for this kind of sequence to advance so "smoothly" from 0.0 to 1.0 as the arithmetic sequence, the common ratio needs to be specified by an additional parameter α.

$$\underset{1 \leqslant i \leqslant n}{\forall} geom(i) = \alpha^{n-i}$$

(S3) **Weighted** – This method, based on sizes of commits (expressed in numbers of fine-grained changes), assigns higher bug-proneness increases to big commits and lower increases to smaller ones.

$$\underset{1 \leqslant i \leqslant n}{\forall} wght(i) = \sum_{1 \leqslant j \leqslant i} size(j) \Big/ \sum_{1 \leqslant j \leqslant n} size(j)$$

[7] Ratio of the number of changes to the modelled method to the total number of changes in the commit.

[8] Gini index, developed by Gini in [31], is a measure of statistical dispersion, most commonly used as a scale of income inequality. It is also called *Gini coefficient* or *Gini ratio*. Here, it is applied to the sizes (in number of changes) of all commits composing the instance.

Table 6 and corresponding Figure 3 present examples of using all of the above strategies.

Table 6: Bug-proneness estimation strategies

Commit	**Size**	**Bug-proneness**		
		Linear	**Geometric$_{0.5}$**	**Weighted**
commit 1	35	0.25	0.13	0.29
commit 2	56	0.50	0.25	0.75
commit 3	13	0.75	0.50	0.83
commit 4	11	1.00	1.00	1.00
bug-fix 1	–	0.00	0.00	0.00
commit 6	34	0.33	0.25	0.11
commit 7	28	0.67	0.50	0.74
commit 8	16	1.00	1.00	1.00
bug-fix 2	–	0.00	0.00	0.00

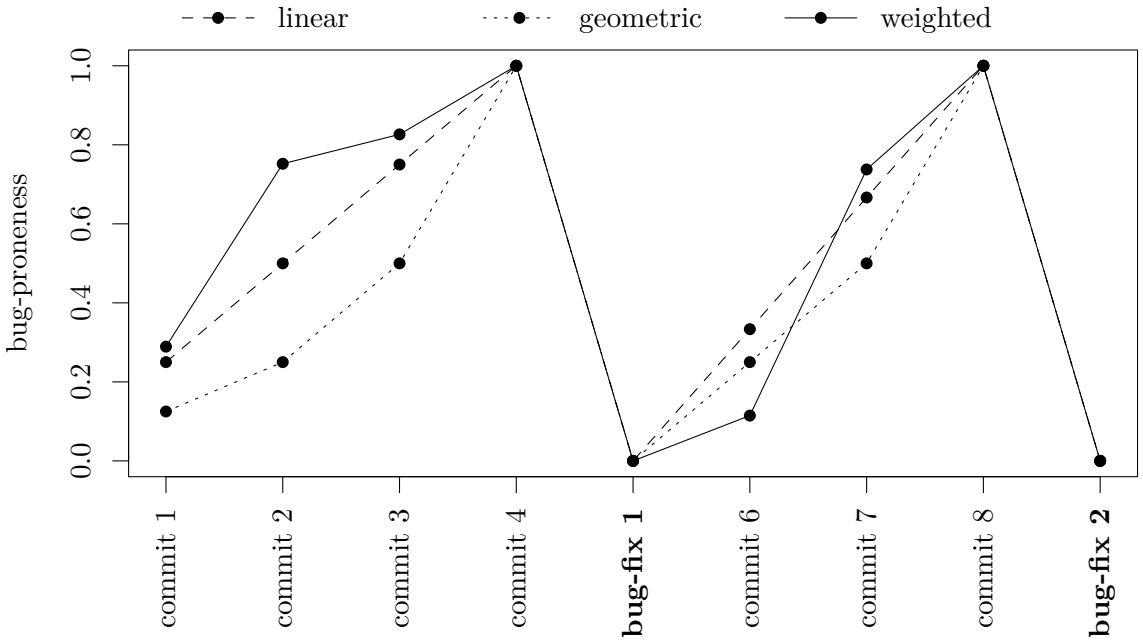

Figure 3: Bug-proneness estimation strategies

3.4. Machine learning

The final step of the development of a defect prediction tool is training a classifier (which in our case should be called a *regressor* for the continuous character of the bug-proneness parameter) that later would be used for the classification of new instances. My approach incorporates four machine learning algorithms, described in the following sections (and evaluated and compared in chapter 5).

3.4.1. Decision tree

According to [67, p. 263], decision tree is a simple, tree-structured classification model. Its inner nodes represent tests, edges correspond to possible test outputs, while leaves contain potential values of the target variable. For each test, the outcomes are both exhaustive and mutually exclusive. Classification of a new case proceeds in a top-down manner. Starting from the tree root, the classified instance is checked against successive conditions (progressing according to the results) until a leaf is reached, from which the result value is retrieved.

The learning algorithm for decision trees, called *recursive splitting* (described in [67, p. 264]) also takes a top-down approach. Beginning with a set of training instances, the best possible split of this set is chosen. Candidates for this choice are all values of attributes appearing in the training dataset. For a discrete attribute, the split has a trivial form – one subset corresponding to one value. For a numerical one, an inequality test or some kind of discretization is required. Quality of the possible splits (or, more precisely, its reciprocal called *impurity*) is evaluated with measures such as *information entropy* [70] or *Gini index* [31]. After choosing the best parameter, the training set is appropriately divided and the algorithm proceeds recursively for each subset. The procedure terminates when the processed subset can no longer be split (either because all instances belong to one class, or due to their inseparability), or when the predefined maximum tree depth is reached.

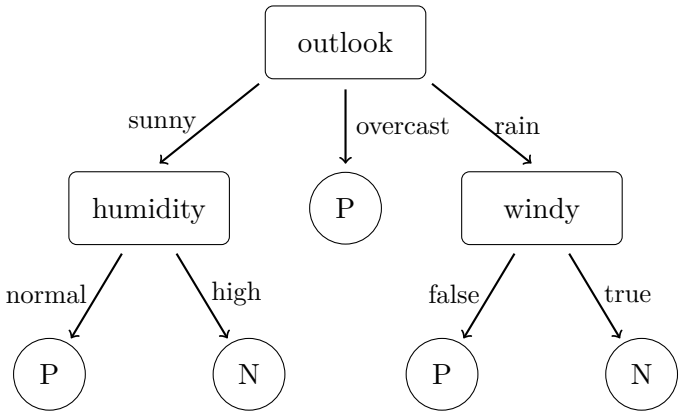

Figure 4: Decision tree example[9]

3.4.2. Random forest

Random forest, introduced by Breiman in [11], is an *ensemble learning algorithm*, using the decision tree (see Section 3.4.1) as the base classifier. It combines two important techniques:

[9]This tree describes the *Play Golf* data set. "P" labels indicate weather suitable for playing golf, while "N" means unsuitable weather. This figure was copied from [61].

bagging and *random subspace method*.

Basically, a random forest is a set of decision trees. Each one of them is trained using a separate random sample of an input training set. The output of such ensemble for test instances is obtained from individual responses by computing their mode (in case of classification) or average (regression). This method, called *bagging* or *bootstrap aggregating*, reduces the instability of the base classifier and improves its accuracy [10].

Random subspace method increases the diversity between members of the ensemble by choosing random subspaces of the features space to be used [67, p. 828]. In case of random forests, this means that each choice of a data split is restricted to a subset[10] of features (not all possibilities are taken into account) [11].

3.4.3. Support vector machine

Support vector machine is a classification model based on the construction of a hyperplane or a set of hyperplanes dividing the feature space of the data. The original form of the algorithm, proposed by Vapnik and Lerner in [77], assumes linear separability of classes. The hyperplane is constructed in a way which minimizes the distance between the hyperplane and the closest points from both classes.

Figure 5 shows two examples of separation of the same set of instances in a two-dimensional space. While both lines separate the classes properly, only **A** is optimal in the aforementioned sense. The *support vectors* (marked with larger points) are those which are placed closest to the separation line. By a formalization of the maximum-margin hyperplane problem and its transformation to the dual form, one can show that the result classifier relies only on these vectors [67, pp. 942–944].

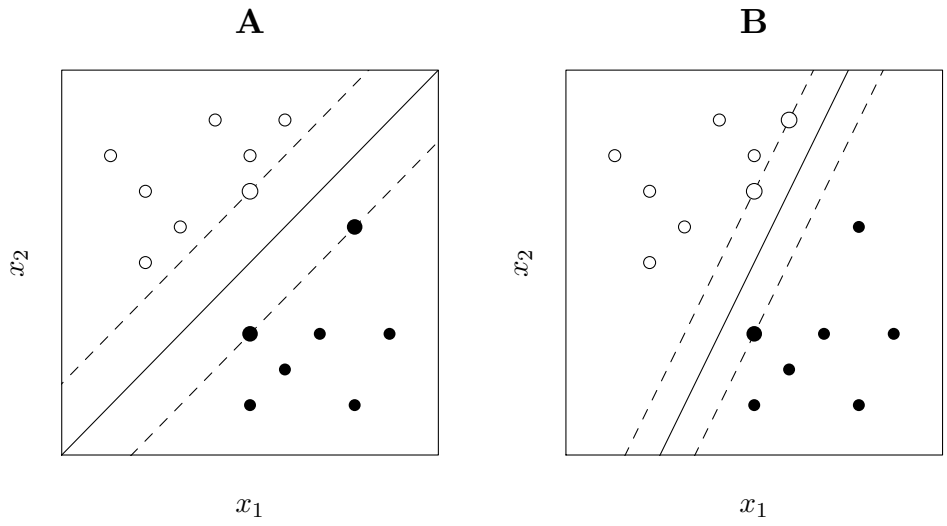

Figure 5: SVM example

The original version of SVM is a linear classifier, which is a significant disadvantage when dealing with non-linearly separable classes. This issue was addressed by Boser *et al.* in [9] through the incorporation of *kernel functions* to the model. This technique allows for a

[10] Typically of size $\sqrt{n}$ where n is the total number of features.

transformation of feature vectors into a high-dimensional space. However, by using the *kernel trick*, no explicit representation of the transformed vectors is created. Another improvement to the algorithm, which enables SVM to handle mislabeled cases, is the *soft margin* introduced by Cortes and Vapnik in [18].

3.4.4. Neural network

(Artificial) neural network is a learning algorithm based on the structure of neural connections in the human brain. A neural network is composed of simple nodes, known as *neurons*, connected by edges. A neuron is a "nonlinear, parameterized, bounded function" [24, p. 2]. It has several inputs (variables) and exactly one output (value). Typical neuron can be described with the following equation:

$$y(x) = f\left(w_0 + \sum_{i=1}^{n}(x_i \cdot w_i)\right)$$

where x_i are variables, w_i are appropriate weights, w_0 is an additional constant called *bias*, and f is an *activation function* (usually a *sigmoid* function).

A neural net consists of several layers of neurons. Attribute values of classified instances are fed to the first layer. Then, successively, outputs of lower layers are propagated along the edges to the higher ones, up to the classifier output. Learning of a network is performed via adjustment of weights associated with the edges by the *backpropagation* algorithm [24, p. 31].

Chapter 4

Implementation

The defect prediction method described in Chapter 3 has been implemented in form of an integrated tool – ChangeAnalyzer. It is an open source Java software, licensed under Apache License 2.0 [2]. The code repository can be found on GitHub: `https://github.com/Wiezzel/changeanalyzer`. The following sections describe the **languages and tools** used for the development (Section 4.1), **interface** (Section 4.2), and **architecture** (Section 4.3) of the program.

4.1. Languages and tools

4.1.1. Java

Java [49] is an object-oriented programming language, developed by James Gosling at Sun Microsystems, currently supported by Oracle Corporation. It has been chosen as the implementation language of ChangeAnalyzer due to its portability, wide availability of useful libraries (described further), and author's experience in Java programming.

JGit
: JGit [41] is an implementation of Git [32] version control system in Java. It allows for programmatically performing such operations as inspecting a repository, tracking commits affecting a given file, or retrieving particular file revisions. JGit is used at the very beginning of the feature extraction process to obtain consecutive revisions of a source file, which are later compared by ChangeDistiller.

ChangeDistiller
: The ChangeDistiller [27] is an Eclipse plugin (also available as a stand-alone library), developed in order to extract fine-grained source code changes by syntax-aware file differentiation (for details see Section 3.2.1). ChangeDistiller is incorporated in ChangeAnalyzer to compare abstract syntax trees of successive file versions provided by JGit and compute the numbers of changes of particular types (see Table 4).

Weka
: The Waikato Environment for Knowledge Analysis (WEKA) [33] is an open source data mining software suite written in Java. It contains a broad collection of state-of-the-art machine learning algorithms and can be used either as a stand-alone application, or as a library. ChangeAnalyzer utilizes Weka to train various predictive models (described in Section 3.4) and predict the defect-proneness of unclassified methods.

LIBSVM

LIBSVM [13] is a library for *support vector machines* (see Section 3.4.3) in C++ and Java. It supports classification (C-SVC, ν-SVC), regression (ε-SVR, ν-SVR) and distribution estimation (one-class SVM). LIBSVM bindings are provided for many languages and frameworks (*e.g.* Python, R, Ruby). ChangeAnalyzer uses a wrapper interface compatible with Weka [52].

4.1.2. R

R [63] is a free, cross-platform software environment for statistical computations. It is not used for the implementation of ChangeAnalyzer itself, but of auxiliary scripts (see Section 5.3) developed for the evaluation of the algorithms presented in Section 3.4.

foreign

foreign [62] is an R package containing functions for reading and writing data in formats specific to other software. It is used to import ARFF[1] files generated by Weka library.

cvTools

cvTools [1] is an R package containing tools for cross-validation (see Section 5.1) of classification and regression models. It used to evaluate te performance of machine learning algorithms described in Section 3.4.

rpart

rpart [74] is an R package containing an implementation of *decision tree* algorithm described in Section 3.4.1.

randomForest

randomForest [51] is an R package containing an implementation of *random forest* classifier described in Section 3.4.2.

e1071

e1071 [57] is an R package providing a biding for LIBSVM and thus allowing for the use of *support vector machines* in R.

nnet

nnet [78] is an R package containing an implementation of *neural network* classifier described in Section 3.4.4.

4.2. Interface

ChangeAnalyzer includes a command-line interface for users. Section 4.2.1 presents the main functionalities offered by ChangeAnalyzer as an integrated tool. Section 4.2.2 describes a test interface, which provides more limited capabilities and was employed during system development and testing.

[1] *Attribute-Relation File Format*. For details see [4].

4.2.1. Main interface

The main interface is provided by the main class of the ChangeAnalyzer runnable JAR. In order to use it, one must simply run the following command:

```
java -jar ChangeAnalyzer.jar <options>
```

From the options described below exactly one input (`extract` or `read`) and at least one output (`save` or `classify`) option has to be chosen.

Input options

`--extract <path>` Extract data from a Git repository under the given `<path>`. Extracted data can be saved to a file or used for method classification. When using this option, one of the below bug-estimation strategies has to be specified (for details see Section 3.3).

`--linear <number>` Use the *linear* strategy, where the given `<number>` is the initial bug-proneness in each chunk.

`--geometric <number>` Use the *geometric* strategy, where the given `<number>` is the bug-proneness increase ratio.

`--weighted` Use the *weighted* strategy.

`--read <path>` Read previously extracted data stored under the given `<path>`.

Output options

`--save <path>` Save the extracted data under the given `<path>` in ARFF[2] format.

`--classify <path>` Predict the bug-proneness of all changed[3] methods and save classification results under the given `<path>` in CSV[4] format.

`--decision-tree` Use the *decision tree* classifier (see Section 3.4.1).

`--random-forest` Use the *random forest* classifier (see Section 3.4.2).

`--svm` Use the *support vector machine* classifier (see Section 3.4.3).

`--neural-net` Use the *neural network* classifier (see Section 3.4.4).

Other options

`--help` Display a list of available options with descriptions.

4.2.2. Test interface

The test interface provides a very limited functionality of extracting data from repositories and saving it. The format of extracted data is not compatible with the main interface – it includes total numbers of fine-grained changes of all types (see Table 4) and three separate bug-proneness estimation columns: linear, geometric$_{0.5}$, and weighted (for details see Section 3.3).

In order to use the test interface, the following command has to be run:

[2] See note 1 on page 30.

[3] By *changed* I mean methods which have been affected by at least one code change since their last bug-fix.

[4] *Comma-Separated Values*. For details see [69]

```
java -cp ChangeAnalyzer.jar pl.edu.mimuw.changeanalyzer.ExtractAndSave \
    <path1> <path2> ...
```

It will result in extracting data from repositories under the provided paths and storing it in ARFF files in the current working directory. Each file is named after the last part of the corresponding repository path (if name collisions occur, files will be overwritten).

4.3. Architecture

This section gives a broad overview of the architecture of ChangeAnalyzer by describing briefly all system components. Subsequent sections contain information about individual packages, which dependencies are presented in Figure 6. The full class diagram is given in Figure 7 on page 39.

Please note, that the following description is not intended to serve as a complete technical documentation. For more details, visit the JavaDoc page: `http://wiezzel.github.io/changeanalyzer/`.

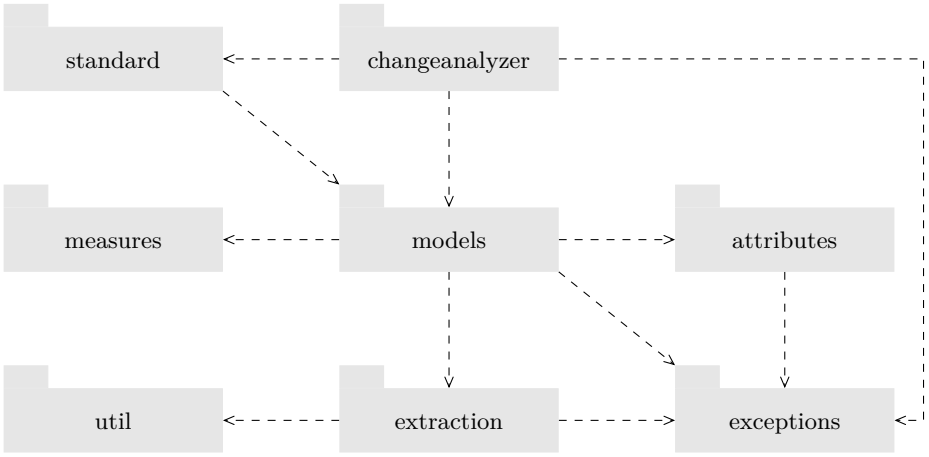

Figure 6: ChangeAnalyzer package diagram

4.3.1. Root package

Full package name: `pl.edu.mimuw.changeanalyzer`

This package contains classes responsible for the interface of ChangeAnalyzer (both main and test interface).

ChangeAnalyzer

ChangeAnalyzer is the main class of the project. It provides the `main` method and several others, which compose the interface described in Section 4.2.1:

- Extract data from a repository,
- Read extracted data from an ARFF file,
- Save extracted data to an ARFF file,
- Classify instances (methods) with missing class attribute.

ChangeAnalyzerOptionParser

This class is a parser of command-line options for `ChangeAnalyzer.main` method, based on Apache Commons™ CLI2 library [17].

ExtractAndSave

This class provides a test interface for performing classifier evaluation experiments (for details, see Section 4.2.2). It is capable of extracting data from repositories and saving unprocessed datasets.

4.3.2. Extraction package

Full package name: `pl.edu.mimuw.changeanalyzer.extraction`

This package contains classes responsible for the extraction of method histories from software repositories.

RepoHistoryExtractor

`RepoHistoryExtractor` is the master class of the extraction package. It contains a `ClassHistoryExtractor` and extracts all class histories and all commits from a repository.

ClassHistoryExtractor

This class is responsible for the extraction of class histories. It holds a reference to a local Git repository. In order to extract a class history (containing the histories of all its methods), one has to provide a relative (to the repository root directory) path to the .java file containing the class. ChangeDistiller library is employed for this task.

AuthorInfo

This class contains the following relevant information about a code author:

- Name,
- E-mail,
- Total number of commits,
- Total number of code changes.

AuthorInfoExtractor

This class is responsible for gathering information about authors extracted from `CommitInfo` objects. It updates the numbers of commits and code changes assigned to an author accordingly to the properties of commits.

CommitInfo

This class contains the following relevant information about a commit:

- Author,
- ID,
- Timestamp,
- Being or not being a bug-fix,
- Number of code changes,

- Number of changed methods.

CommitInfoExtractor

This is an extractor of relevant information from JGit's objects representing commits, which is stored as `CommitInfo` objects. It updates the numbers of code changes and changed methods by analysing method histories provided by ChangeDistiller.

ClassHistoryWrapper

This is a class suitable for wrapping a collection of class histories and iterating over histories of all their methods. It allows also for an iteration over histories of methods of inner classes.

MethodHistoryIterator

`MethodHistoryIterator` is a static nested class of `ClassHistoryWrapper`. This iterator walks through all class histories by a *depth-first search* on the composition graph and yields method histories. Next element to be returned by this iterator is always pre-loaded.

ExtractionUtils (interface)

This is a utility class for repository information extraction (providing one static method which allows to obtain the HEAD of a repository).

4.3.3. Models package

Full package name: `pl.edu.mimuw.changeanalyzer.models`

This package contains classes responsible for defining the data model and creation of its instances by processing of data extracted from software repositories.

DataSetBuilder (abstract)

This is the abstract base class for dataset builders – classes transforming method histories into model instances. Each builder class defines a set of attributes and provides methods to produce instances with this attributes. Created instances may represent commits, chunks, method snapshots, etc. depending on the model used by a particular builder.

ChunkDataSetBuilder (abstract)

extends `DataSetBuilder`

`ChunkDataSetBuilder` is a dataset builder that processes commits in chunks separated by bug-fixes. Defect-proneness scores are assigned by pluggable measures (strategies). Obtaining multiple scores for a single instance is possible by providing multiple bug-proneness measures to the builder.

DataSetProcessor (interface)

This is an interface for objects processing data sets (that means modifying and/or removing existing features as well as computing new ones). Typically, a dataset processor aggregates a few attribute processors (see description of attributes package on page 35).

DataSetProvider

`DataSetProvider` is a top-level class responsible for retrieving datasets either by extracting them from repositories or by reading files with previously extracted data. A provider wraps functionalities provided by a `DataSetBuilder` and a `DataSetProcessor`, and additionally separates training instances from test instances.

ReadOnlyDataSetProvider

extends `DataSetProvider`

`ReadOnlyDataSetProvider` is a simple dataset provider incapable of extracting or processing data. It expects processed data with the class (target) attribute on the last position. Any attempts to call methods responsible for the unsupported operations result in throwing an exception.

DefaultDataSetProcessor

implements `DataSetProcessor`

This is a static nested class of `ReadOnlyDataSetProvider`. The only operation performed by this processor is marking the last attribute of a dataset as the class attribute.

ChangeCounter

This is a simple class for counting code changes of different types. It holds a separate, internal counter for each change type.

4.3.4. Attributes package

Full package name: `pl.edu.mimuw.changeanalyzer.models.attributes`

This package contains classes representing attributes of data models and operations which could be performed on them.

Attributes

This is a class representing a set of model attributes.

AttributeProcessor (interface)

`AttributeProcessor` is an interface for classes manipulating attributes of dataset.

SumAttributes

implements `AttributeProcessor`

This is an attribute processor class which adds to the given dataset a new attribute being the sum of the defined source attributes.

DeleteAttributes

implements `AttributeProcessor`

This is an attribute processor class which removes a group of defined attributes from the given dataset.

ReorderClassAttribute

implements `AttributeProcessor`

This is an attribute processor class which moves the class attribute to the last position.

4.3.5. Measures package

Full package name: `pl.edu.mimuw.changeanalyzer.models.measures`

This package contains bug-proneness measure classes representing estimation strategies defined in Section 3.3.

BugPronenessMeasure (interface)

`BugPronenessMeasure` is an interface for classes assigning bug-proneness scores to model instances produced by a `ChunkDataSetBuilder`. A measure could be state-aware in a sense that each assigned score does not depend solely on properties of a single method version, but of a whole chunk.

LinearMeasure

implements `BugPronenessMeasure`

This measure assigns bug-proneness score of 1.0 to the last commit of a chunk and linearly decreasing scores to the previous ones.

GeometricMeasure

implements `BugPronenessMeasure`

This measure assigns bug-proneness score of 1.0 to the last commit of a chunk and geometrically decreasing scores to the previous ones.

WeightedMeasure

implements `BugPronenessMeasure`

This measure computes bug-proneness scores proportionally to numbers of fine-grained changes since the last bug-fix. It assigns each group of commits a score equal to the ratio of number of changes in this group to the total number of changes in the whole chunk.

4.3.6. Standard implementations package

Full package name: `pl.edu.mimuw.changeanalyzer.models.standard`

This package contains standard implementations of abstract classes and interfaces defined in the models package.

StandardDataSetBuilder

extends `DataSetBuilder`

This data set builder class uses commit group as a model. Each produced instance represents a group of subsequent commits starting with a commit directly succeeding a bug-fix. Groups contain from one commit up to a whole chunk.

StandardDataSetProcessor

implements `DataSetProcessor`

This is the standard dataset processor used by ChangeAnalyzer for processing raw data sets built by a `StandardDataSetBuilder`. It performs the following operations:

- Discard some unnecessary attributes,
- Sum values of parameter change-related attributes into a new attribute,
- Sums values of header change-related attributes a new attribute.

StandardDataSetProvider

extends `DataSetProvider`

A `StandardDataSetProvider` uses a `StandardDataSetBuilder` and a `StandardDataSetProcessor` to create and process datasets.

4.3.7. Exceptions package

Full package name: `pl.edu.mimuw.changeanalyzer.exceptions`

This package contains exceptions thrown by ChangeAnalyzer.

ChangeAnalyzerException

This is the base class for all ChangeAnalyzer exceptions.

ExtractionException

extends `ChangeAnalyzerException`

`ExtractionException` is an exception thrown when an error occurs during the extraction of information from a repository.

DataSetBuilderException

extends `ChangeAnalyzerException`

This is an exception thrown by the `DataSetBuilder` class. It indicates an error during the creation of a dataset.

ProcessingException

extends `ChangeAnalyzerException`

`ProcessingException` is an exception thrown when an error occurs during the processing of a dataset.

PredictionException

extends `ChangeAnalyzerException`

`PredictionException` is an exception thrown due to an error in performing the bug-proneness prediction.

4.3.8. Utilities package

Full package name: `pl.edu.mimuw.changeanalyzer.util`

This package contains some universal, reusable utility classes.

LazyList<T>

`LazyList` is a generic container class designed for storing sequences provided by disposable iterable objects. It wraps up an ordinary list which is extended on demand by advancing the source iterator.

LazyIterator

`LazyIterator` is an inner class of `LazyList` enabling the iteration through its elements.

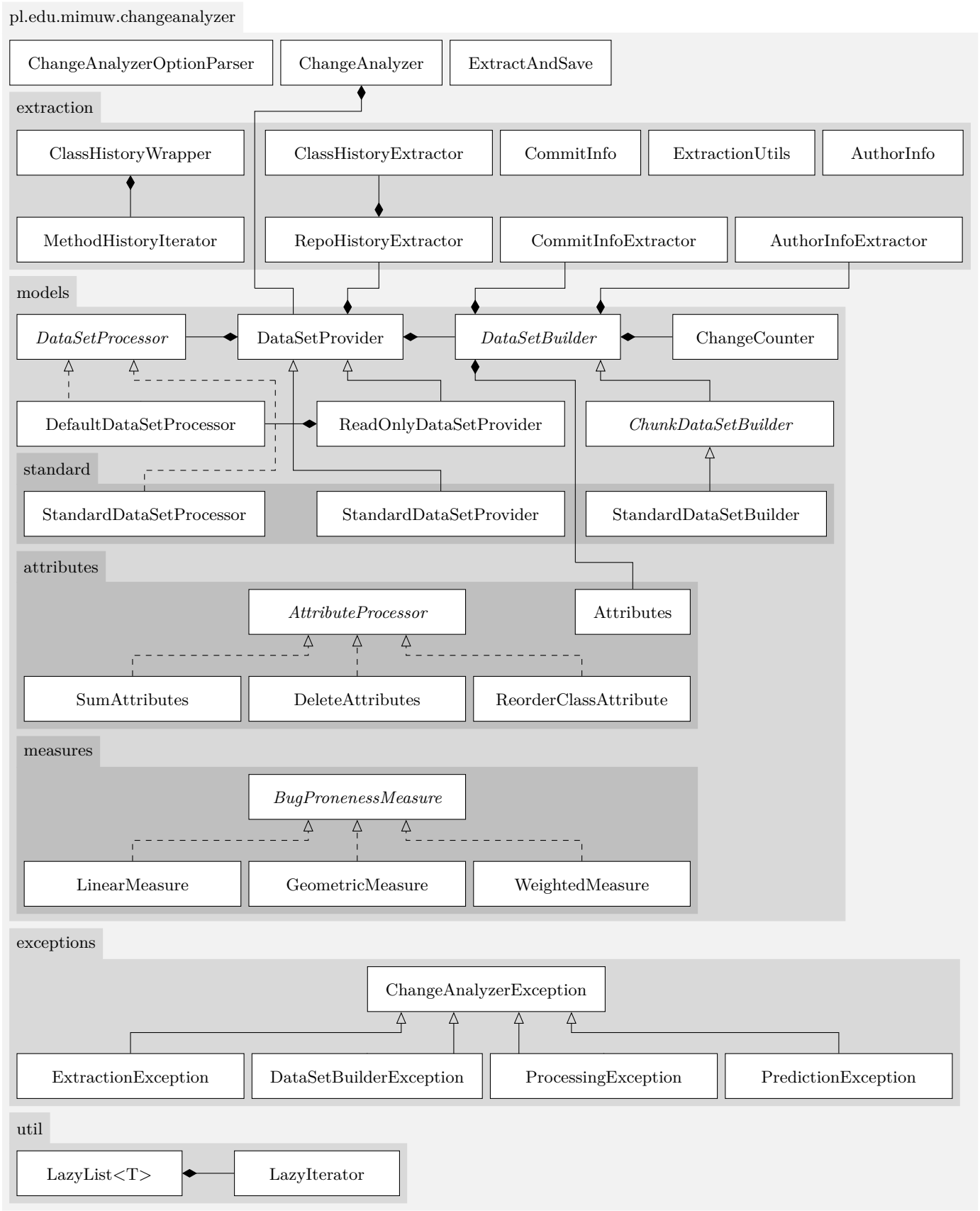

Figure 7: ChangeAnalyzer class diagram

Chapter 5

Experiments

In order to evaluate the accuracy of different machine learning algorithms (presented in Section 3.4) applied to the proposed data model (Section 3.1), as well as the efficiency of the implementation, several experiments were performed. The data for analysis was obtained from open source Java projects. Experiments were conducted using both the ChangeAnalyzer tool and a special purpose R script.

The following sections describe the general **methodology** (Section 5.1), **experimental data** (Section 5.2), **software set-up** (Section 5.3), and **results** (Section 5.4) followed by a short **commentary** (Section 5.5).

5.1. Methodology

In the intended ordinary use of ChangeAnalyzer, all code changes followed by a bug-fix compose the training set, while the unfixed ones (lacking bug-proneness scores) make up the test set. However, predictions of defect-proneness made in such situation are not verifiable until new fixes appear. Therefore, in order to evaluate the proposed predictive models, part of the training set has to be used as the test set.

A popular technique applied in such situations is called *cross-validation*. As described in [67, p. 249], cross-validation starts with splitting the dataset into several subsets (also called *folds*) $S_1 ... S_n$ of approximately equal size. Then the learning algorithm is applied n times, in every i-th step using fold S_i as the test set and the union of other folds as the training set. Evaluation scores from all steps are averaged. In order to reduce the variability, cross-validation is often applied multiple times with different dataset partitions, averaging the results. In my experiments I used 20-fold cross-validation repeated 10 times for each dataset.

Another important issue is the choice of an appropriate accuracy measure. I decided to utilize the widely applied *root mean squared error* (RMSE), which is defined in [66] with the following formula:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Where n is the number of test observations, y is the vector of *gold standard* values and $\hat{y}$ is the vector of predicted values.

5.2. Data

Ten big open source Java projects were used as experimental data. They were cloned from their GitHub repositories on 23 February 2015. The list of projects with several size-related statistics is presented in Table 7.

Table 7: Test datasets

Project	#classes	#commits	#instances	#changes
Commons Lang	279	4 113	7 426	30 362
Guava	1 738	2 695	10 222	38 207
Hibernate	7 037	5 693	19 172	116 256
Jetty	2 351	10 455	16 087	352 499
JGit	1 133	3 507	4 198	28 903
JUnit	393	2 020	2 384	13 493
Log4j	309	2 644	808	5 037
Maven	941	9 984	6 005	88 069
Mockito	917	2 545	3 680	14 581
Spring	5 943	9 831	9 566	18 181

5.3. Experimental set-up

Repositories containing the test data were cloned from GitHub into local directories. Further, the test interface of ChangeAnalyzer was used (see Section 4.2.2), to extract data from the repositories into ARFF files.

The data was further processed by an R script, which performed the following operations:

1. Read the data.
2. Pre-process the data.
   2.1 Remove unnecessary features.
   2.2 Sum header changes[1] into one attribute.
   2.3 Sum parameter changes[2] into one attribute.
3. Run 10 times a 20-fold cross-validation for all datasets, for all models, for all bug-proneness scores.
4. Average the cross-validation results by datasets.

Full code listing of the script can be found in Appendix A.

[1] See note 5 on page 22.
[2] See note 6 on page 22

All experiments were performed on a single machine with 4GB of RAM and Intel® Core™ 2 Duo T6570 CPU, running Windows XP 32-bit operating system. Time of the extraction was measured using the system clock.

5.4. Results

5.4.1. Accuracy

Table 8 and corresponding Figure 8 present results of the cross-validation, for all classifiers and bug-proneness estimation strategies, averaged by datasets. The results were rounded up to three decimal places.

Table 8: RMSE averaged by datasets

Classifier	**Bug-proneness**		
	Linear	**Geometric$_{0.5}$**	**Weighted**
Decision tree	0.253	0.325	0.272
Random forest	0.222	0.281	0.238
SVM	0.267	0.341	0.288
Neural network	0.271	0.351	0.292

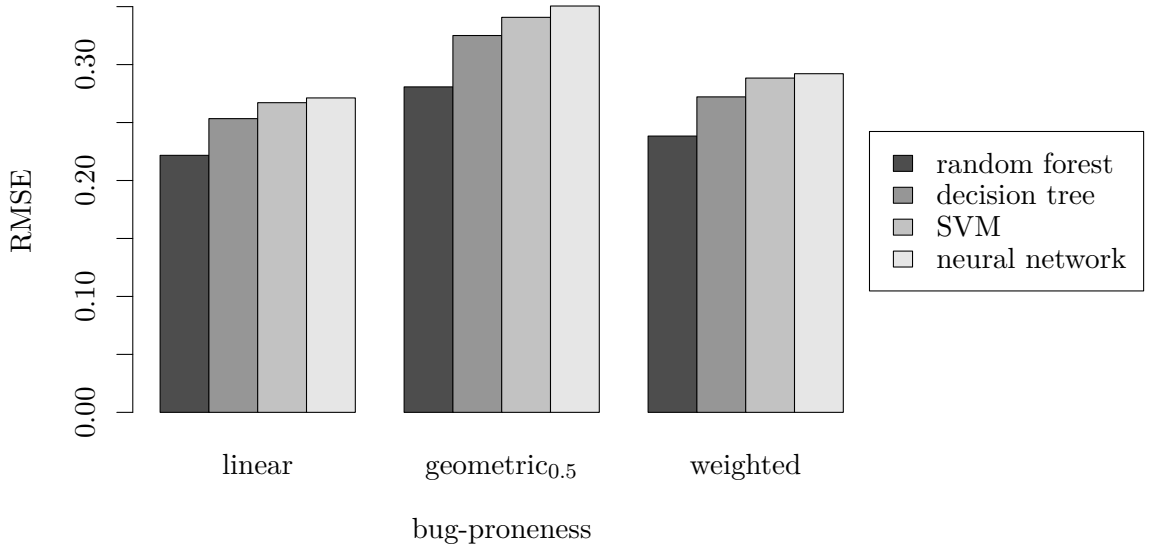

Figure 8: Classification accuracy

5.4.2. Performance

A comparison of project sizes (in numbers of classes) and corresponding extraction times (in seconds) is presented in Table 9 below. The extraction times were rounded up to two decimal places. Figure 9 additionally includes a plot of the above function fitted to the data using *linear regression*.

$$time(size) = 0.49 \cdot size + 24.95$$

Table 9: Data extraction performance

Project	**#classes**	**extraction time (s)**
Commons Lang	279	475.80
Guava	1 738	685.97
Hibernate	7 037	3 587.91
Jetty	2 351	1 611.86
JGit	1 133	502.42
JUnit	393	69.33
Log4j	309	60.34
Maven	941	663.08
Mockito	917	191.03
Spring	5 943	2731.47

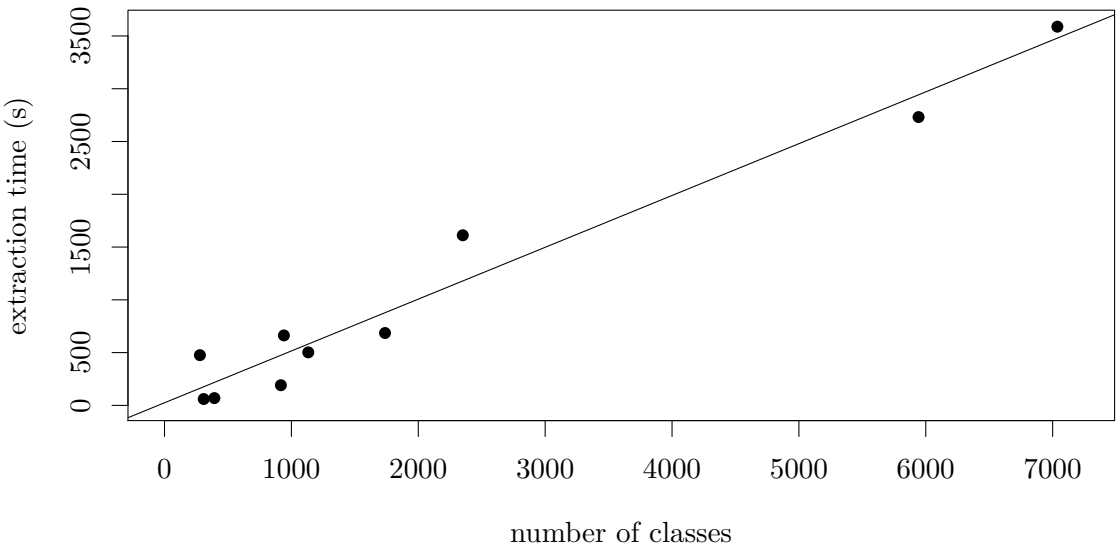

Figure 9: Data extraction performance

5.5. Commentary

The experimental results presented in the previous section show clearly that random forest classifier achieves the best results from all proposed machine learning algorithms. Regardless of the bug-proneness estimation strategy, RMSE value for random forest was the lowest.

For every strategy, the performance order of the classifiers was the same, which confirms that using different ways of defining the defect-proneness parameter does not establish multiple problems, but rather different aspects of a single one.

Regarding the strategies, it is noticeable that using the geometric one (with $\alpha = 0.5$) results in significantly higher error values than the two others. This behaviour may be caused by the large difference (0.5) between bug-proneness scores assigned by this strategy to the last two commits in each chunk.

Time measurements show that the extraction of data from repositories is a rather computationally heavy process, approximately linearly dependent on the size of the analysed project. The extraction time is ranging from about 1 minute for the smallest projects, up to nearly 1 hour for the biggest one. Such values exclude the possibility of integration of the proposed method into an interactive tool, leaving only the option of batch processing.

Chapter 6

Conclusions

In this thesis I have referred the topic of identification and prediction of defects in software. Further, I have presented my approach towards this problem and described an implementation of a software tool – ChangeAnalyzer – which realizes this approach. Finally, I have presented the results of experiments which were performed in order to provide a fair empirical basis for the assessment of the tool and its underlying methods. In this chapter I would like to conclude my work by issuing a judgement of the presented ideas, describing possible threats to the validity of my research and outlining the possibilities of its further extension.

6.1. Assessment of approach

Due to the unique character of the target variable, the outcomes of classifiers' evaluation are incomparable with results presented in related work. I my opinion, they should be described as promising, but still far from perfect. An error rate reaching over 30% of the target variable's range definitely calls for some more improvement.

6.2. Threats to validity

6.2.1. Abstract target variable

The *bug-proneness* parameter, defined in Section 3.1.1, contrary to most variables presented in Section 2.2, is not directly observable. In other words, such choice of dependent variable may be criticized as lacking clear empirical interpretation, which makes it scarcely employable for practical solutions.

However, this charge can addressed by pointing to an observation, that *bug-proneness* (by every computation strategy described in Section 3.3) can be interpreted as a measure of *temporal proximity of the next commit*, which is definitely an information of practical value.

6.2.2. Simple identification of bugs

ChangeAnalyzer uses the simplest possible defect identification method, which drawbacks have been already described in Section 2.4. Due to its inaccuracy, the intended golden standard, which serves as a basis for prediction, may significantly diverge from the *true golden standard*.

6.2.3. Usage of default parameters

Machine learning algorithms, evaluated in the experiments, were used mostly with default parameter values defined by the authors of appropriate libraries. These values may not be well-suited for the specific character of the data processed by ChangeAnalyzer. Therefore, the presented results may be an underestimation of the possible performance of the used classifiers.

6.2.4. Choice of test data

The test data used for the experimental evaluation of the proposed approach was obtained exclusively from publicly available, open source projects. Such projects may have some distinct characteristics arising from their unrestrained organizational structure. Obtaining more representative results would also require analysing of commercial software, developed using different methodologies.

Moreover, no cross-project prediction was performed. Numerous studies show that it is a significantly more challenging task than building a within-project model [65, 81, 82]. As Zimmmermann *et al.* point out in [82], universal models are of high value for new software projects, from which no sufficient amount of data to train a classifier could be extracted.

6.2.5. Cost-insensitive evaluation

The evaluation of models was performed using RMSE measure, which does not take into account the amount of effort which would be wasted in case of a misclassification. Estimation of the cost of potential corrective actions would provide a basis for a more robust assessment. However, the construction of an error measure which addresses this issue requires a deeper insight into the software development process. It also has to be mentioned, that the possible consequences of the two different kids of errors[1] may vary a lot. Arisholm *et al.* in [5] propose the *cost-effectiveness* measure, which tries to embrace all the mentioned restrictions.

6.3. Possible further work

6.3.1. New features

Expansion of the feature set used by ChangeAnalyzer may lead to the improvement of its performance. Possible new attributes, which could be introduced into the tool include:

- **Static code metrics**, such as number of method parameters or statements, cyclomatic complexity, or Halstead effort (for more examples, see Table 1),
- **Delta metrics**, used by Arisholm *et al.* in [5], which are representing the changes in static metrics over time,
- **More historical metrics**, including all listed in Table 2.

[1] Namely: *false positives* and *false negatives*. False positive (also called *type I error*) is an indication of the presence of the predicted condition (in our case, a software defect) in the absence of this condition. On the other hand, false negative (called *type II error*) is a situation where the condition has not been discovered despite its occurrence.

6.3.2. Advanced bug identification methods

In order to address the issue risen in Section 6.2.2, some improvements can be made in the defect identification procedure. It is possible to extend ChangeAnalyzer to utilize information from a bug-tracking system, which is a more reliable data source than commit messages. Another possibility is the integration with an external bug identification tool, such as ReLink [80].

Independently of the above enhancements, which concern the bug-fix identification, there is a possibility of improvement in the field of tracking bug-introducing changes. By the incorporation of a tailored version of the SZZ algorithm (described in [72]), a new defect-proneness computation method could be implemented. Basing on matches provided by SZZ, increases of bug-proneness could be ascribed exclusively to commits affecting these parts of code which are later modified by a bug-fix.

6.3.3. Parameter tuning

Accuracy of the machine learning algorithms used by ChangeAnalyzer can be augmented by *parameter tuning*. Cross-validation method, used for the evaluation (see Section 5.1), can be also employed for this purpose. In order for the tuning to be successful, it has to be performed for each analysed project individually. This however, would largely impair the performance of the tool, so the scope of the tuning has to be limited by a prior analysis of a possibly broad range of software projects.

6.3.4. On-line learning

In its current shape, ChangeAnalyzer does not support any form of incremental development of prediction models. After obtaining new training data (which emerges after every bug-fix), the extraction of method histories and training of a model needs to be reiterated. Such design was chosen for the simplicity and brevity of the implementation, but it could be abolished in the further development. The possibility of limiting the scope of computation to the latest code changes would provide a great performance boost for ChangeAnalyzer.

Bibliography

[1] Andreas Alfons. *cvTools: Cross-validation tools for regression models*. R package version 0.3.2. 2012. URL: http://CRAN.R-project.org/package=cvTools (visited on 05/03/2015).

[2] *Apache License, Version 2.0*. Apache Software Foundation. Jan. 2004. URL: http://www.apache.org/licenses/LICENSE-2.0 (visited on 05/02/2015).

[3] *Apache Subversion*. Apache Software Foundation. URL: http://subversion.apache.org/ (visited on 03/22/2015).

[4] *ARFF (stable version)*. University of Waikato. Mar. 2, 2015. URL: http://weka.wikispaces.com/ARFF+(stable+version) (visited on 05/07/2015).

[5] Erik Arisholm, Lionel C. Briand, and Eivind B. Johannessen. "A Systematic and Comprehensive Investigation of Methods to Build and Evaluate Fault Prediction Models". In: *J. Syst. Softw.* 83.1 (Jan. 2010), pp. 2–17. ISSN: 0164-1212.

[6] Douglas N. Arnold. *The Explosion of the Ariane 5*. Aug. 23, 2000. URL: http://www.ima.umn.edu/~arnold/disasters/ariane.html (visited on 03/20/2015).

[7] Adrian Bachmann et al. "The Missing Links: Bugs and Bug-fix Commits". In: *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE '10. Santa Fe, New Mexico, USA: ACM, 2010, pp. 97–106. ISBN: 978-1-60558-791-2.

[8] W. H. K. Bester, C. P. Inggs, and W. C. Visser. "Test-case Generation and Bug-finding Through Symbolic Execution". In: *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*. SAICSIT '12. Pretoria, South Africa: ACM, 2012, pp. 1–9. ISBN: 978-1-4503-1308-7.

[9] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. "A Training Algorithm for Optimal Margin Classifiers". In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*. COLT '92. Pittsburgh, Pennsylvania, USA: ACM, 1992, pp. 144–152. ISBN: 0-89791-497-X.

[10] Leo Breiman. "Bagging Predictors". In: *Machine Learning* 24.2 (1996), pp. 123–140. ISSN: 0885-6125.

[11] Leo Breiman. "Random Forests". In: *Machine Learning* 45.1 (2001), pp. 5–32. ISSN: 0885-6125.

[12] Bora Caglayan, Ayse Bener, and Stefan Koch. "Merits of Using Repository Metrics in Defect Prediction for Open Source Projects". In: *Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*. FLOSS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 31–36. ISBN: 978-1-4244-3720-7.

[13] Chih-Chung Chang and Chih-Jen Lin. “LIBSVM: A library for support vector machines”. In: *ACM Transactions on Intelligent Systems and Technology* 2 (3 2011). Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm, 27:1–27:27.

[14] Ning Chen and Sunghun Kim. “Puzzle-based Automatic Testing: Bringing Humans into the Loop by Solving Puzzles”. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2012. Essen, Germany: ACM, 2012, pp. 140–149. ISBN: 978-1-4503-1204-2.

[15] S. R. Chidamber and C. F. Kemerer. “A Metrics Suite for Object Oriented Design”. In: *IEEE Trans. Softw. Eng.* 20.6 (June 1994), pp. 476–493. ISSN: 0098-5589.

[16] Brian Cole et al. “Improving Your Software Using Static Analysis to Find Bugs”. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA ’06. Portland, Oregon, USA: ACM, 2006, pp. 673–674. ISBN: 1-59593-491-X.

[17] *Commons CLI – CLI2 – Overview*. Apache Software Foundation. Jan. 12, 2013. URL: http://commons.apache.org/sandbox/commons-cli2/manual/ (visited on 05/15/2015).

[18] Corinna Cortes and Vladimir Vapnik. “Support-Vector Networks”. In: *Mach. Learn.* 20.3 (Sept. 1995), pp. 273–297. ISSN: 0885-6125.

[19] *CVS – Concurrent Versions System – annotate*. URL: ftp://ftp.gnu.org/old-gnu/Manuals/cvs/html_node/cvs_74.html (visited on 04/12/2014).

[20] *CVS – Open Source Version Control*. Dec. 3, 2006. URL: http://cvs.nongnu.org/ (visited on 04/12/2014).

[21] *Debugging in Visual Studio*. Microsoft Corporation. URL: http://msdn.microsoft.com/en-us/library/sc65sadd.aspx (visited on 03/20/2015).

[22] Lee R. Dice. “Measures of the Amount of Ecologic Association Between Species”. In: *Ecology* 26.3 (July 1945), pp. 297–302. ISSN: 0012-9658.

[23] *Diffutils – GNU Project*. Free Software Foundation, Inc. May 21, 2010. URL: https://www.gnu.org/software/diffutils/ (visited on 04/24/2015).

[24] Gérard Dreyfus. *Neural Networks: Methodology and Applications*. 1st. Berlin, Heidelberg: Springer-Verlag, 2005. ISBN: 978-3-540-28847-3.

[25] Jayalath Ekanayake et al. “Time Variance and Defect Prediction in Software Projects”. In: *Empirical Softw. Engg.* 17.4-5 (Aug. 2012), pp. 348–389. ISSN: 1382-3256.

[26] Beat Fluri and Harald C. Gall. “Classifying Change Types for Qualifying Change Couplings”. In: *Proceedings of the 14th IEEE International Conference on Program Comprehension*. ICPC ’06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 35–45. ISBN: 0-7695-2601-2.

[27] Beat Fluri et al. “Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction”. In: *IEEE Trans. Softw. Eng.* 33.11 (Nov. 2007), pp. 725–743. ISSN: 0098-5589.

[28] Takafumi Fukushima et al. “An Empirical Study of Just-in-time Defect Prediction Using Cross-project Models”. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: ACM, 2014, pp. 172–181. ISBN: 978-1-4503-2863-0.

[29] *GDB: The GNU Project Debugger*. Free Software Foundation, Inc. URL: http://www.gnu.org/software/gdb/ (visited on 03/20/2015).

[30] Emanuel Giger et al. “Method-level Bug Prediction”. In: *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM ’12. Lund, Sweden: ACM, 2012, pp. 171–180. ISBN: 978-1-4503-1056-7.

[31] Corrado Gini. *Variabilità e Mutabilità: contributo allo Studio delle distribuzioni e delle relazioni statistiche*. Tipogr. di P. Cuppini, 1912.

[32] *Git*. URL: `http://git-scm.com/` (visited on 03/22/2015).

[33] Mark Hall et al. “The WEKA Data Mining Software: An Update”. In: *SIGKDD Explor. Newsl.* 11.1 (Nov. 2009), pp. 10–18. ISSN: 1931-0145.

[34] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977. ISBN: 0-444-00205-7.

[35] Ahmed E. Hassan. “Predicting Faults Using the Complexity of Code Changes”. In: *Proceedings of the 31st International Conference on Software Engineering*. ICSE ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–88. ISBN: 978-1-4244-3453-4.

[36] Kim Herzig, Sascha Just, and Andreas Zeller. “It's Not a Bug, It's a Feature: How Misclassification Impacts Bug Prediction”. In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE ’13. San Francisco, CA, USA: IEEE Press, 2013, pp. 392–401. ISBN: 978-1-4673-3076-3.

[37] *Home :: Bugzilla*. Feb. 27, 2015. URL: `https://www.bugzilla.org/` (visited on 04/11/2015).

[38] James S. Huggins. *First Computer Bug*. Aug. 16, 2000. URL: `http://www.jamesshuggins.com/h/tek1/first_computer_bug.htm` (visited on 03/20/2015).

[39] “IEEE Standard Glossary of Software Engineering Terminology”. In: *IEEE Std 610.12-1990* (Dec. 1990), pp. 1–84.

[40] Andrea Janes et al. “Identification of Defect-prone Classes in Telecommunication Software Systems Using Design Metrics”. In: *Inf. Sci.* 176.24 (Dec. 2006), pp. 3711–3734. ISSN: 0020-0255.

[41] *JGit – Documentation*. Eclipse Foundation. 2015. URL: `http://eclipse.org/jgit/` (visited on 05/02/2015).

[42] Xin Jin et al. “Support Vector Machines for Regression and Applications to Software Quality Prediction”. In: *Proceedings of the 6th International Conference on Computational Science - Volume Part IV*. ICCS’06. Reading, UK: Springer-Verlag, 2006, pp. 781–788. ISBN: 3-540-34385-7, 978-3-540-34385-1.

[43] *JIRA – Issue & Project Tracking Software*. URL: `https://www.atlassian.com/software/jira` (visited on 04/11/2015).

[44] Mik Kersten and Gail C. Murphy. “Mylar: A Degree-of-interest Model for IDEs”. In: *Proceedings of the 4th International Conference on Aspect-oriented Software Development*. AOSD ’05. Chicago, Illinois: ACM, 2005, pp. 159–168. ISBN: 1-59593-042-6.

[45] Sunghun Kim. “Adaptive Bug Prediction by Analyzing Project History”. AAI3229992. PhD thesis. Santa Cruz, CA, USA, 2006. ISBN: 978-0-542-83718-0.

[46] Sunghun Kim et al. “Dealing with Noise in Defect Prediction”. In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE ’11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 481–490. ISBN: 978-1-4503-0445-0.

[47] Pavneet Singh Kochhar, Tien-Duy B. Le, and David Lo. "It's Not a Bug, It's a Feature: Does Misclassification Affect Bug Localization?" In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: ACM, 2014, pp. 296–299. ISBN: 978-1-4503-2863-0.

[48] Thomas D. LaToza, Gina Venolia, and Robert DeLine. "Maintaining Mental Models: A Study of Developer Work Habits". In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. Shanghai, China: ACM, 2006, pp. 492–501. ISBN: 1-59593-375-1.

[49] *Learn about Java Technology*. Oracle Corporation. 2015. URL: https://www.java.com/en/about/ (visited on 05/02/2015).

[50] Taek Lee et al. "Micro Interaction Metrics for Defect Prediction". In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE '11. Szeged, Hungary: ACM, 2011, pp. 311–321. ISBN: 978-1-4503-0443-6.

[51] Andy Liaw and Matthew Wiener. "Classification and Regression by randomForest". In: *R News* 2.3 (2002), pp. 18–22. URL: http://CRAN.R-project.org/doc/Rnews/ (visited on 05/03/2015).

[52] *LibSVM*. University of Waikato. Mar. 26, 2014. URL: http://weka.wikispaces.com/LibSVM (visited on 05/08/2015).

[53] MaytagMetalark. *ChangeSet - Mercurial*. Nov. 6, 2013. URL: http://mercurial.selenic.com/wiki/ChangeSet (visited on 03/25/2015).

[54] Thomas J. McCabe. "A Complexity Measure". In: *IEEE Trans. Softw. Eng.* 2.4 (July 1976), pp. 308–320. ISSN: 0098-5589.

[55] *Mercurial SCM*. URL: http://mercurial.selenic.com/ (visited on 03/22/2015).

[56] *Message Passing Coupling*. ARiSA. 2014. URL: http://www.arisa.se/compendium/node113.html (visited on 04/15/2015).

[57] David Meyer et al. *e1071: Misc Functions of the Department of Statistics (e1071), TU Wien*. R package version 1.6-1. 2012. URL: http://CRAN.R-project.org/package=e1071 (visited on 05/03/2015).

[58] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. "A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction". In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. Leipzig, Germany: ACM, 2008, pp. 181–190. ISBN: 978-1-60558-079-1.

[59] *Most Popular Coding Languages of 2015*. Feb. 9, 2015. URL: http://blog.codeeval.com/codeevalblog/2015 (visited on 04/02/2015).

[60] *Mylyn*. The Eclipse Foundation. 2015. URL: http://projects.eclipse.org/projects/mylyn (visited on 04/17/2015).

[61] J.R. Quinlan. "Induction of decision trees". In: *Machine Learning* 1.1 (1986), pp. 81–106. ISSN: 0885-6125.

[62] R Core Team. *foreign: Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, dBase, ...* R package version 0.8-54. 2013. URL: http://CRAN.R-project.org/package=foreign (visited on 05/03/2015).

[63] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2014. URL: http://www.R-project.org (visited on 05/02/2015).

[64] Foyzur Rahman and Premkumar Devanbu. "How, and Why, Process Metrics Are Better". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 432–441. ISBN: 978-1-4673-3076-3.

[65] Foyzur Rahman, Daryl Posnett, and Premkumar Devanbu. "Recalling the "Imprecision" of Cross-project Defect Prediction". In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE '12. Cary, North Carolina: ACM, 2012, 61:1–61:11. ISBN: 978-1-4503-1614-9.

[66] *Root Mean Squared Error*. Kaggle Inc. Jan. 5, 2015. URL: http://www.kaggle.com/wiki/RootMeanSquaredError (visited on 05/19/2015).

[67] Claude Sammut and Geoffrey I. Webb. *Encyclopedia of Machine Learning*. 1st. Springer Publishing Company, Incorporated, 2011. ISBN: 0387307680, 9780387307688.

[68] Robert W. Sebesta. *Concepts of Programming Languages*. 10th. Pearson, 2012. ISBN: 0273769103, 9780273769101.

[69] Y. Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. RFC 4180. Oct. 2005. URL: http://www.rfc-editor.org/rfc/rfc4180.txt (visited on 05/10/2015).

[70] C.E. Shannon. "A mathematical theory of communication". In: *Bell System Technical Journal, The* 27.3 (July 1948), pp. 379–423. ISSN: 0005-8580.

[71] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. "HATARI: Raising Risk Awareness". In: *SIGSOFT Softw. Eng. Notes* 30.5 (Sept. 2005), pp. 107–110. ISSN: 0163-5948.

[72] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. "When Do Changes Induce Fixes?" In: *SIGSOFT Softw. Eng. Notes* 30.4 (May 2005), pp. 1–5. ISSN: 0163-5948.

[73] Ravishankar Somasundaram. *Git: Version Control for Everyone*. Birmingham - Mumbai: Packt Publishing, Jan. 2013. ISBN: 978-1-84951-752-2.

[74] Terry Therneau, Beth Atkinson, and Brian Ripley. *rpart: Recursive Partitioning*. R package version 4.1-0. 2012. URL: http://CRAN.R-project.org/package=rpart (visited on 05/03/2015).

[75] *TIOBE Index for March 2015*. URL: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (visited on 04/02/2015).

[76] Piotr Tomaszewski, Lars Lundberg, and Hakan Grahn. "Increasing the Efficiency of Fault Detection in Modified Code". In: *Proceedings of the 12th Asia-Pacific Software Engineering Conference*. APSEC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 421–430. ISBN: 0-7695-2465-6.

[77] Vladimir N. Vapnik and Aleksandr Lerner. "Pattern Recognition Using Generalized Portrait Method". In: *Automation and Remote Control* 24.6 (1963), pp. 774–780. ISSN: 0005-1179.

[78] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S*. Fourth. New York: Springer, 2002. ISBN: 0-387-95457-0.

[79] Chadd Williams and Jaime Spacco. "SZZ Revisited: Verifying when Changes Induce Fixes". In: *Proceedings of the 2008 Workshop on Defects in Large Software Systems*. DEFECTS '08. Seattle, Washington: ACM, 2008, pp. 32–36. ISBN: 978-1-60558-051-7.

[80] Rongxin Wu et al. "ReLink: Recovering Links Between Bugs and Changes". In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE '11. Szeged, Hungary: ACM, 2011, pp. 15–25. ISBN: 978-1-4503-0443-6.

[81] Feng Zhang et al. "Towards Building a Universal Defect Prediction Model". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: ACM, 2014, pp. 182–191. ISBN: 978-1-4503-2863-0.

[82] Thomas Zimmermann et al. "Cross-project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process". In: *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC/FSE '09. Amsterdam, The Netherlands: ACM, 2009, pp. 91–100. ISBN: 978-1-60558-001-2.

Appendix A

Experimental script

```
# Load libraries

library(foreign)
library(cvTools)

library(randomForest)
library(rpart)
library(e1071)
library(nnet)

# Informational columns

infoCols = c(
  "methodName",
  "commitId"
)

# Uninteresting columns to be discarded

discardCols = c(
  "ADDING_ATTRIBUTE_MODIFIABILITY",
  "ADDING_CLASS_DERIVABILITY",
  "ADDITIONAL_CLASS",
  "ADDITIONAL_FUNCTIONALITY",
  "ADDITIONAL_OBJECT_STATE",
  "ATTRIBUTE_RENAMING",
  "ATTRIBUTE_TYPE_CHANGE",
  "CLASS_RENAMING",
  "COMMENT_DELETE",
  "COMMENT_INSERT",
  "COMMENT_MOVE",
  "COMMENT_UPDATE",
  "DOC_DELETE",
  "DOC_INSERT",
  "DOC_UPDATE",
  "PARENT_CLASS_CHANGE",
  "PARENT_CLASS_DELETE",
  "PARENT_CLASS_INSERT",
  "PARENT_INTERFACE_CHANGE",
  "PARENT_INTERFACE_DELETE",
```

```
  "PARENT_INTERFACE_INSERT",
  "REMOVED_CLASS",
  "REMOVED_FUNCTIONALITY",
  "REMOVED_OBJECT_STATE",
  "REMOVING_ATTRIBUTE_MODIFIABILITY",
  "REMOVING_CLASS_DERIVABILITY",
  "UNCLASSIFIED_CHANGE"
)

# Change columns to be used as raw features

rawChangeCols = c(
  "ALTERNATIVE_PART_DELETE",
  "ALTERNATIVE_PART_INSERT",
  "CONDITION_EXPRESSION_CHANGE",
  "STATEMENT_DELETE",
  "STATEMENT_INSERT",
  "STATEMENT_ORDERING_CHANGE",
  "STATEMENT_PARENT_CHANGE",
  "STATEMENT_UPDATE"
)

# Parameter change columns

paramChangeCols = c(
  "PARAMETER_DELETE",
  "PARAMETER_INSERT",
  "PARAMETER_ORDERING_CHANGE",
  "PARAMETER_RENAMING",
  "PARAMETER_TYPE_CHANGE"
)

# Header change columns

headerChangeCols = c(
  "ADDING_METHOD_OVERRIDABILITY",
  "DECREASING_ACCESSIBILITY_CHANGE",
  "INCREASING_ACCESSIBILITY_CHANGE",
  "METHOD_RENAMING",
  "REMOVING_METHOD_OVERRIDABILITY",
  "RETURN_TYPE_CHANGE",
  "RETURN_TYPE_DELETE",
  "RETURN_TYPE_INSERT"
)

# Additional features

additionalFeatureCols = c(
  "numCommits",
  "numAuthors",
  "avgChanges",
  "avgEntities",
  "avgAuthorCommits",
  "avgAuthorChanges",
  "avgChangeRatio",
```

```
  "changeGini",
  "timeSinceLastFix"
)

# Bug-proneness columns

bugPronenessCols = c(
  "linBugProneness0.0",
  "geomBugProneness0.5",
  "weightBugProneness"
)

# A function for reading data, preprocessing features
# and filtering instances

readTrainData = function(filePath) {
  data = read.arff(filePath)

  bugProneness = data[bugPronenessCols]
  rawChanges = data[rawChangeCols]
  paramsChanges = apply(data[paramChangeCols], 1, sum)
  headerChanges = apply(data[headerChangeCols], 1, sum)
  additionalFeatures = data[additionalFeatureCols[
    additionalFeatureCols %in% colnames(data)]]

  trainData = cbind(
    rawChanges, additionalFeatures, bugProneness,
    paramsChange = paramsChanges, headerChange = headerChanges
  )
  trainData[complete.cases(bugProneness),]
}

# Functions for performing cross-validation of various models

performCV = function(trainData, classColName, ...) {
  x = trainData[, !names(trainData) %in% bugPronenessCols]
  y = trainData[, classColName]
  trainData = cbind(x, class=y)
  complete = complete.cases(trainData)

  forestCV = suppressWarnings(
    cvFit(randomForest, class~., trainData, ...))$cv
  treeCV = suppressWarnings(
    cvFit(rpart, class~., trainData, ...))$cv
  svmCV = suppressWarnings(
    cvFit(svm, class~., trainData, ...))$cv
  nnetCV = suppressWarnings(
    cvFit(nnet, x=x[complete,], y=y[complete],
          args=c(size=10, linout=T, trace=F), ...))$cv

  c(forest=forestCV, tree=treeCV, svm=svmCV, nnet=nnetCV)
}

performAllCV = function(trainData, K=20, R=10, seed=12345, ...) {
  sapply(bugPronenessCols, function(col) performCV(
```

```
    trainData, classColName = col, K=K, R=R, ...
  ))
}

# Read data extracted from repositories

dataSets = list(
  commonsLang = readTrainData("commons-lang.arff"),
  guava = readTrainData("guava.arff"),
  hibernate = readTrainData("hibernate.arff"),
  jetty = readTrainData("jetty.arff"),
  jgit = readTrainData("jgit.arff"),
  junit = readTrainData("junit.arff"),
  log4j = readTrainData("log4j.arff"),
  maven = readTrainData("maven.arff"),
  mockito = readTrainData("mockito.arff"),
  spring = readTrainData("spring.arff")
)
sizes = sapply(dataSets, nrow)

# Perform experiments and aggregate the results

results = lapply(dataSets, performAllCV)
avgResults = apply(simplify2array(results), c(1,2), mean)
```