U N I V E R S I T Y O F T A R T U
Faculty of Mathematics and Computer Science
Institute of Computer Science

Anton Litvinenko

Automatic Prediction of Source Code Contribution Type

Master Thesis

Supervisors: Jaak Vilo, PhD
Ulrich Norbisrath, Dipl.-Inform.

TARTU 2007

Contents

Acknowledgements

There are a lot of people whose help and contribution I would like to recognise.

First of all, I am lucky enough to have two advisors: Dr. Jaak Vilo and Ulrich Norbisrath. The existence of this thesis wouldn't be possible without either of them. Jaak has guided me throughout the whole period of my master studies. His energy and devotion moves mountains. It is impossible to resist them, they are infectious. Ulrich's fresh ideas and constructive criticism have led me during the most difficult time of the studies — the last month before the thesis submission.

Not a single achievement is possible without the support of the family and friends. Darja has always been there for me with her support, patience and empathy. Mark has been open to discussions of any ideas, troubles and results. I can't imagine overcoming all obstacles without their help.

Nothing motivates more than the success of the friends. Darja, Liina, and Dan, I want to thank you for the demonstration of how devotion and hard working can lead to tremendous results. I wish you all a happy graduation!

I would like to mention Jevgeni Kabanov and Karel Kravik. Jevgeni has provided me with the Changelogic database dumps characterising the development of Aranea Framework. Karel has helped to figure out how to examine these dumps and what kind of information should I look for.

Finally, I would like to thank members of the BIIT group for their willingness to help. Meelis, Pavlos, Hedi, Ilja and especially Konstantin thank you for the help and criticism!

This Master's Thesis has been partially supported by the ETF grant number 5722 and SourceKibitzer OÜ.

Introduction

The phenomenon of open source software is conquering the world. The number of open source software developers increases every month. Contributing to open source projects is attractive for developers. It is a chance to learn from their smart and talented colleagues. Users love open source software for its community and chance to see what actually they are using. Commercial companies start to be more favourable to open source solutions even when it comes to performing their mission-critical activities and developing custom software using open source libraries.

Unfortunately, it is rare for an open source software project to have a legal entity behind it: self-selected volunteers perform crucial tasks like customer support and defect fixing. Often this causes a problem for companies that deploy open source software in their production environments: even though the source code and infrastructure is open, it is difficult to tell if software is mature enough or its support and development won't be abandoned in the near future. Today, the problem of evaluation of the open source project's state is an important topic.

Let's consider a typical scenario of implementing a new feature in custom software that will help us to surface the problem. For instance, a requirement of implementing the data import from the Microsoft Excel spreadsheet in the Java language. While one option is to implement the functionality all by ourselves, the quicker way would be to use already existing libraries or products that we could easily integrate into our software. Let's assume that we would really like to use a library developed under the open source model and we have found two libraries equally well satisfying our functional requirements: Java Excel API[1] and Jakarta POI[2]. Which one should we choose?

In order to choose, one has to review and compare both libraries from

[1]`http://jexcelapi.sourceforge.net/`

[2]`http://jakarta.apache.org/poi/`

different perspectives. Lets consider some of them in detail:

maturity: Most open source software projects struggle to produce the first stable release. If such release is not produced during the first 3–4 years of the development then probability of the stable release reduces dramatically [SFT07]. Thus, it is really important to check how long a library has been developed, what its current state is and if the community has managed to perform a stable release. Otherwise usage of the library might compromise stability or availability of the application.

community: The community built around the library should be active and responsive. This way it is more likely that the library will continue to evolve and the developer won't be left alone with her troubles. To analyse how active the community is, one has to check:

- how quickly and how often help requests are answered on the mailing lists and forums of the library
- how often answering party doesn't belong to the team of core contributors
- what is the ratio of closed defects to the total number of defects in the issue tracking system of the project.

documentation: Library usage should be extensively documented. FAQ-s, references, guides and examples — all these should be easily accessible. Lack of the diverse documentation will decrease development speed and complicate software maintenance.

source code quality: Source code of the library should be covered with commentaries, shouldn't contain unnecessarily complex components and be easily extendable and maintainable.

The openness is the key to such data: issue tracking systems, source code repositories, mailing lists, forums and wiki pages — all these are freely accessible thanks to the nature of open source development. On the other hand manual execution of such analysis would not only compromise the speed of the development, but also require a lot of routine and uninteresting work to be performed by the developer.

A much more rational and convenient approach would be to have all this information already put together and automatically renewed. The developer would only have to login to some environment and there she would be able to compare various solutions from different angles. There is a number of initiatives that are already trying to provide similar services: SourceKibitzer [Soub], Ohloh [Ohl], Business Readiness Rating [Busa] and others.

This thesis tackles the problem of predicting the open source software state and concentrates on a smaller part of this problem: automation of prediction of the source code contribution type. By the word *contribution* we refer to a collection of source code modifications grouped by the common goal. For instance, a contribution might be a bug fix or the implementation of some new functionality. Usually, the growth of the number of bug fixes is a sign of an upcoming release. On the other hand the large number of new functionality contributions indicates that software is being actively developed and no stable release is planned in the near future. Thus, if one is able to tell the type of each contribution then she could also tell the state of the software.

The thesis is structured as follows: we start with an overview of software development including open source software development, versioning systems and software metrics. We then continue with an introduction to data mining along with a description of the prediction algorithm used in the case study. We propose a method to be used for prediction of the contribution type. Afterwards we test our method on the historical data of Aranea Framework and evaluate its accuracy. We continue with an overview of the studies and initiatives related to our work. As the proposed method is far from being complete, we conclude this work with the summary of results and directions for further method improvements.

Chapter 1

Software Development

This chapter starts with a brief analysis of open source software development. Then we continue with an introduction of the software development aspects relevant to the method and case study described in the following chapters.

1.1 Open Source Software

It may seem that the title “Open Source” speaks for itself: it is something for which source is widely available: for software — the source code, for dishes — the recipe, or for architecture — the blueprints. However, the availability of the source is not the only criterion. Many other aspects like redistribution, derived work, and licences are also involved in the definition of open source [Ini].

Let’s consider the case of open source software. According to the definition [Ini], the licence of the open source software shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The licence shall not require a royalty or other fee for such.

In order to maintain the definition of the open source software, the special corporation named Open Source Initiative (OSI) has been founded [Ope]. According to OSI, “open source” is a software development method that harnesses the power of distributed peer review and process transparency. The promise of open source is better quality, higher reliability, more flexibility, lower cost, and an end to predatory vendor lock-in.

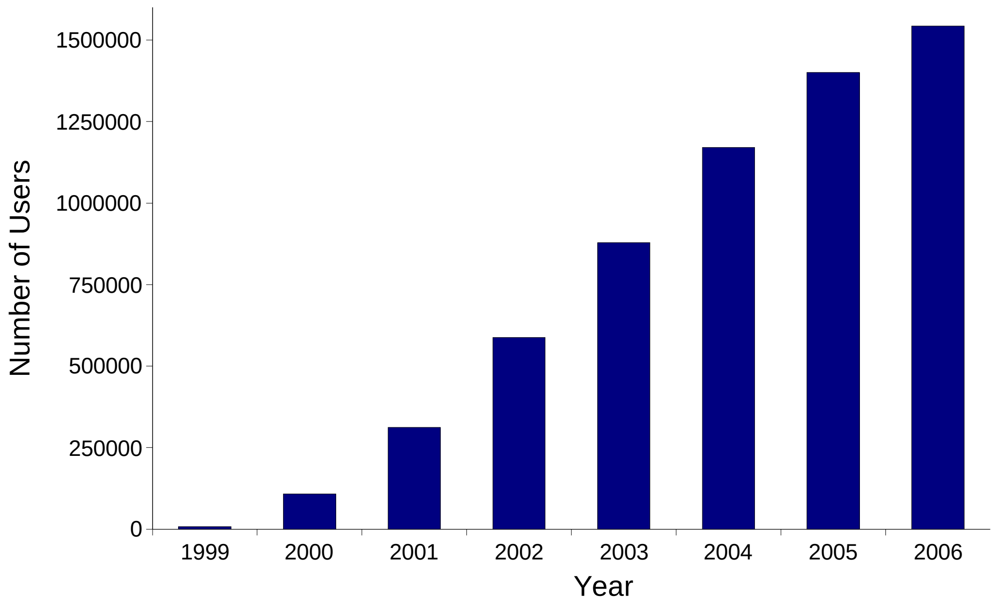


Figure 1.1: Growth of the number of registered users at the popular open source software hosting service — SourceForge.

The open source software development model is very popular among software developers. For instance, the number of registered users at SourceForge [Soua], the most popular open source software hosting service, has grown over 1.5M (Figure 1.1) [FLOa]. Developers are motivated to join and actively participate in the development of open source software. According to the study performed by Ghosh *et al.* the main drivers for both joining and participation are similar: possibility to learn and develop new skills along with sharing already existing knowledge and expertise [GGKR02].

1.2 Open Source Software Research

The success of open source software has attracted many researches to study it. Initially, the main investigation topic was the popularity of open source software development. Nowadays more and more studies are devoted to see how open source projects evolve [RAGBH05, Wu06, RGBMA06], how communities are organised [LFRGB04, XCM06, CWL$^+$07], and why some projects are more successful than the others [SFT07].

The number of researches couldn't be that large without support by governments and corporations. We have found around 20 different scientific groups, projects and programmes studying open source software. To name a few:

- FLOSSWorld [FLOb]
- QUALOSS [QUAb]
- QualiPSo [Quaa]
- GSyC/LibreSoft Research Group [GSy]
- SQO-OSS [SQO]

On the other hand all these groups and studies are stimulated by the large number of open source software development projects. Throughout the development a lot code and other accompanying information, like messages in mailing lists, posts in forums, feature requests, and defect descriptions in issue tracking systems, is produced. If we consider the fact that for a typical software engineering researcher it is extremely difficult to get access to the development data of closed commercial applications, then such a diverse data produced by open source projects seems like a natural choice for the empirical research.

1.3 Versioning Systems

Implementation of a particular feature or fix of a defect involves inspection and modification of several files at a time. At the same time other developers should be able to work on their tasks, perhaps modifying the same files. And a tester would want to get the latest version of the software to run her tests on it. In order to overcome these and many other difficulties, caused by the distributed software development, special tools are used. Often they are called *versioning systems* or *version control systems*. At the moment the most well-known versioning systems are Concurrent Versions System [CVS], Subversion [SVN], Git [Git], Perforce [Per] and BitKeeper [Bit].

Firstly, the usage of versioning systems enables developers to work on different versions of the same document at the same time. Secondly, all files are located at one centralised place where any member of the development team

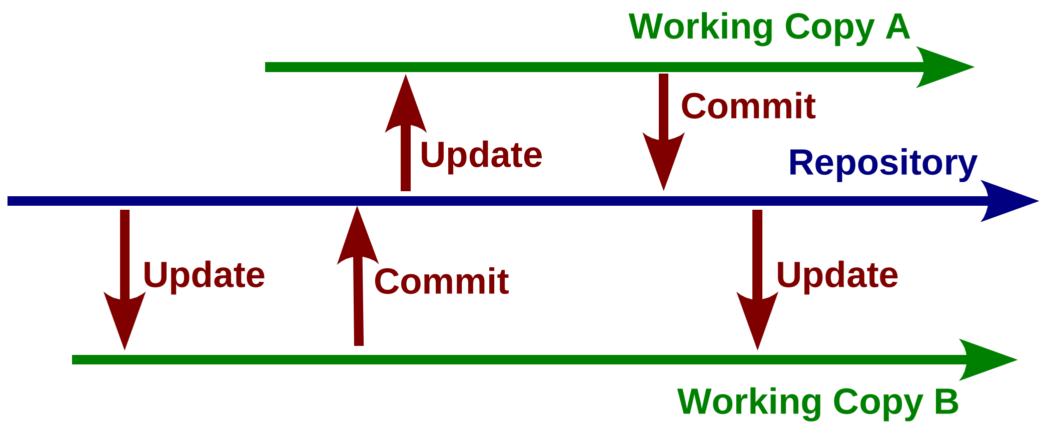


Figure 1.2: Illustration of the versioning system's typical operations.

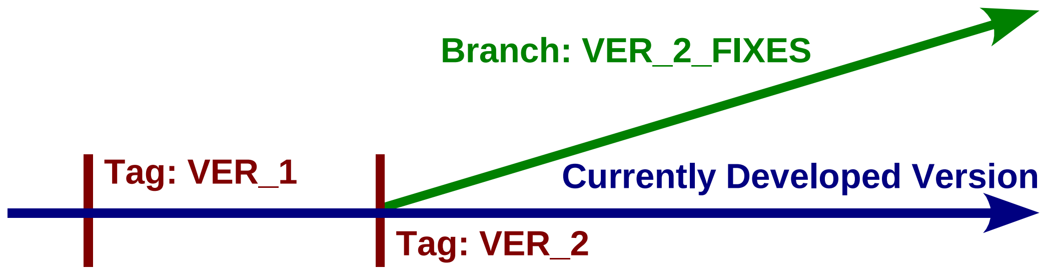


Figure 1.3: Illustration of the possible usage of the tags and branches. Version 1 of the software was tagged with a label *VER_1*, version 2 — *VER_2*. Defects, that were found in the version 2 of the software, were fixed in the branch *VER_2_FIXES*.

may find them without disturbing colleagues. Finally, versioning systems enable exploration of the software evolution in time.

A place where different versions of files are stored is called *repository*. Throughout the development developers submit to the repository modifications to the documents and download modifications made by other developers. The action of putting own modifications to the repository is called *commit*. The download of modifications made by others — *update*. The downloaded version of files that is used by the developer to perform her tasks is called *working copy* (Figure 1.2). Actual semantics of these operations depend on the concrete versioning system.

Two other important concepts of the versioning system usage are *tag* and *branch.* A tag refers to some important snapshot in time, which is consistent across many files. These files at that point may be all tagged with some name. Branches enable developers to develop several versions of the same file (set of files) at the same time, at different speeds, independently of others. A particularly useful application of tags and branches together would be during the release of the software new version: with a tag developers mark the version of files that constitute the particular software release. Later, if defects are found in this version, developers will be able to locate the exact files included in the release and fix the defects in the corresponding branch without disturbing the development of new features (Figure 1.3).

It is rare for a developer to modify a single file while performing a task. Usually, she works on a number of files and after finishing the task commits them to the repository. Throughout the thesis such sets of modifications are called *contributions.* It is worth noticing that a contribution and a commit do not represent the same concept. In fact a contribution constitutes of one or more commits and may span many minutes due to network latency or developer's style of performing commits.

1.4 Software Metrics

Software metrics are quantified expressions of software characteristics. Based on the object of measurement software metrics are divided into 3 categories: product metrics (in our case product is a result of the development — software), process metrics and project metrics [Kan02]. Product metrics describe the characteristics of the product, such as size, complexity, design features and performance. Process metrics are used to improve software development and maintenance. Examples of process metrics are effectiveness of defect removal during development and response time of the fix process. Project metrics describe project characteristics and execution: number of software developers, cost, schedule, and productivity [Kan02].

Probably the most famous product metric is the number of lines of code — "LOC". This metric expresses the size of the software. The value of LOC is convention dependant: should we count number of physical lines or only non-blank lines, should we include lines containing commentaries? The size of the code in Figure 1.4 could be either equal to 11 — number of physical lines, or 7 — number of lines containing source code, or 9 — number on non-blank lines.

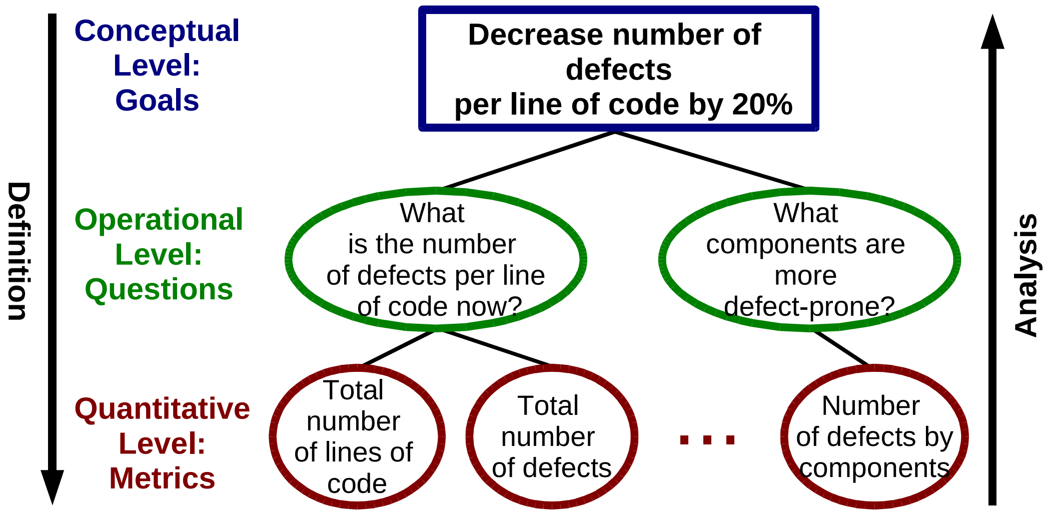
```
// setting type attribute to be the class
trainingSet
        .setClassIndex(reference.numAttributes() - 1);
testSet.setClassIndex(reference.numAttributes() - 1);

RandomForest classifier = new RandomForest();
classifier.setNumFeatures(featureNum);
classifier.setNumTrees(treeNum);

// building the classifier using training set
classifier.buildClassifier(trainingSet);
```

Figure 1.4: Example of LOC metric.

Usage of software metrics is motivated by the famous saying stating that it is not possible to control something that cannot be measured [DeM82]. Managers strive to have a control over the software project in order to mitigate risks threatening the project success.

Unfortunately it is easier to misuse software metrics rather than apply them fruitfully. First of all, it is tempting to judge developer's performance based on the set of metrics. But, this is a dangerous move because a lot of aspects should be kept in mind. For instance, the most talented developers often get the most difficult and time consuming tasks. However, based purely on numbers we might conclude that the developer is underperforming. Secondly, as soon as the team members find out that they are being evaluated using particular metric or combination of metrics, they start searching for possibilities of maximising the management perception of their performance. Thirdly, no known metric is considered to be accurate and meaningful at the same time. Finally, in many cases managers just start collecting metrics and then try to find goals that fit with them. Such approach is not working because of the large number of observable characteristics in software. Without an appropriate model and concrete goals it is not clear which metrics should be used and how they should be interpreted.

A better approach is the opposite: we should start on the conceptual



Conceptual Level: Goals
Operational Level: Questions
Quantitative Level: Metrics
Definition
Analysis
Decrease number of defects per line of code by 20%
What is the number of defects per line of code now?
What components are more defect-prone?
Total number of lines of code
Total number of defects
...
Number of defects by components

Figure 1.5: Example of Goal-Question-Metric approach.

level — define goals to be achieved by the development team. Then carefully select such questions, that their answers would indicate if the goal has been reached. And finally, we should specify metrics, which would help in finding the answers to the previously defined questions. During the analysis, we should start with collection of metrics values, continue with answering the questions and finish with evaluation of our achievements.

Let's consider a simple example: our goal is to decrease number of defects in software per line of code by 20% (Figure 1.5). In order to monitor how this goal is being achieved we need to know what is the ratio of defects to the number of code lines at the moment. Collection of values for two metrics is required for answering this question: total number of defects found during the last month and total number of lines of code written during the last month. In the similar manner we specify metrics for the second question: what are the software components where ratio of defects to the number of lines is the highest. List of such components would indicate where we need to direct our energy.

The demonstrated approach is known by the name of Goal-Question-Metric (GQM) framework and was introduced by Basili *et al.* in "The Goal Question Metric Approach" [BCR94].

Chapter 2

Data Mining

In this chapter we are giving an overview of the data mining methods used in the remaining chapters of the thesis. We start with a short introduction to data mining. Then continue with the data mining notation. Finally, we describe the technique used for prediction of the contribution type.

2.1 Introduction

The steady progress in the information technology has lead to the situation when we have automated tools for data collection. Existence of such tools has caused the availability of tremendous amounts of data stored in databases and data warehouses. But the data itself is not the knowledge: "We are drowning in data, but starving for knowledge!" [HK00]. Necessity of turning all these amounts of data into useful knowledge and information has attracted a lot of attention to data mining.

There are quite many definitions of what data mining is [WF05, HK00, HMS01]. Most agree on these key points: it is an automatic or semi-automatic process of discovering patterns in data. Discovered patterns should be previously unknown, non-trivial and meaningful. Non-trivial means that it is difficult or not possible to spot such patterns by the naked eye. Meaningful means that found patterns lead to some advantage, usually an economic. Finally, the data should be present in substantial quantities.

2.2 Notation

A data mining technique or algorithm is usually applied to data sets. Data sets consist of records — *instances*. Every instance is characterised by its values on the fixed set of *attributes*. If we take a database or a spreadsheet table as an analogy then its rows are instances and columns are attributes. For example, Table 2.1 contains 7 instances and 4 attributes: filename, time, author, and type.

Filename	**Time**	**Author**	**Type**
AjaxRequestErrorWidget.java	20:10 26.04.07	alar	BUG
BaseServiceRouterService.java	20:10 26.04.07	-	BUG
TemplateMenuWidget.java	15:24 28.04.07	taimo	DEVEL.
StandardWizardWidget.java	15:24 28.04.07	taimo	DEVEL.
RootWidget.java	15:24 28.04.07	taimo	DEVEL.
EventButtonHtmlTag.java	*16:38 01.01.67*	taimo	BUG
SqlFunctionFilter.java	16:38 28.04.07	taimo	BUG

Table 2.1: Example of a data set. Each row is a separate instance and each column — an attribute.

Sometimes instances in the data set contain a special *class attribute*. Values of this attribute illustrate how all instances can be divided into different groups. For instance, for the data set depicted in Table 2.1, the type attribute may be considered as a class, because it tells us that all instances can be divided into two separate categories: BUG and DEVEL (DEVELOPMENT).

It may happen that some instances don't have values for some attributes. This may be caused by an error in the data collecting software or people not filling in the data completely. Such, not existing values are usually called *missing values*. Another problem appears when a small number of instances have extreme values of some attributes. Such values are called *outliers*. In our example (Table 2.1), the second instance has a missing value for the attribute author and in the penultimate instance the value for attribute time is an outlier. Both missing values and outliers compromise the accuracy of the results produced by data mining techniques.

2.3 Prediction and Classification

The following tools are usually considered to play a central role in the data mining process: data description, association rule mining, clustering and prediction [HK00]. This thesis focuses only on prediction, more specifically on classification.

Prediction techniques are applied when some instances in the data set contain values of the class attribute and researcher's task is to predict class for the remaining instances. If values of the class attribute are discrete or nominal then it is a case for the *classification*. In case of continuous or ordered values — *regression* [HK00]. Examples of classification algorithms are Decision Tree Induction, Naive Bayes and k-Nearest Neighbor [HK00].

The process of prediction starts with a training: an algorithm gets set of instances with specified value of the class attribute. Such data set is called *training set*. During this step the algorithm tries to learn possible dependencies between the values of the class and other attributes. After that we ask the trained algorithm to predict class for other instances.

The *accuracy* of the classification is usually evaluated as the ratio of correctly classified instances to the number of all instances. This is easy if we have the whole data set at hand. But how should we estimate the accuracy if we have only a subset of the data?

Usage of the training set for both training and estimation of the accuracy would produce too optimistic estimate [WF05]. A popular alternative method is called *cross-validation*. The whole training set is divided into n subsets — *folds*. Then for each fold the algorithm is trained with the remaining folds and after that the current accuracy is evaluated using the current fold. Hence, the algorithm is trained and tested for n times. The final accuracy estimation is equal to the ratio of number of correctly predicted classes for all folds to the total number of all instances in folds [WF05]. n — the number of folds is specified by the researcher. Commonly, researchers take the value of n to be 10 and then the technique is called — 10-*fold cross-validation*. When the number of folds is taken to be equal to the number of instances in the data set, then such validation is called *leave-one-out*. The name comes from the fact, that at each step we leave one instance out, train the algorithm using all remaining instances and check if the class of the left out instance was predicted correctly [WF05].

2.4 Random Forest Classifier

During the experiments we have used a classification algorithm called *Random Forest Classifier*. The idea behind it is quite simple: generate many decision trees and output the most frequent class that appeared in the classifications produced by all these trees [Bre01]. Under the hood random forest puts together 3 different techniques: decision tree, bagging and randomisation.

Decision tree is a natural implementation of the "divide-and-conquer" approach to the classification of the set of the independent instances. In such tree each node represent a test of a particular attribute. The result of the test dictates which path down the tree should be taken. Leaf nodes of the tree represent classes. Hence, to classify an unknown instance one has to traverse the tree according to the test results in the nodes. The classification result is the class assigned to the leaf where the traversal ends [WF05]. A simple example of the decision tree can be seen in Figure 2.1.

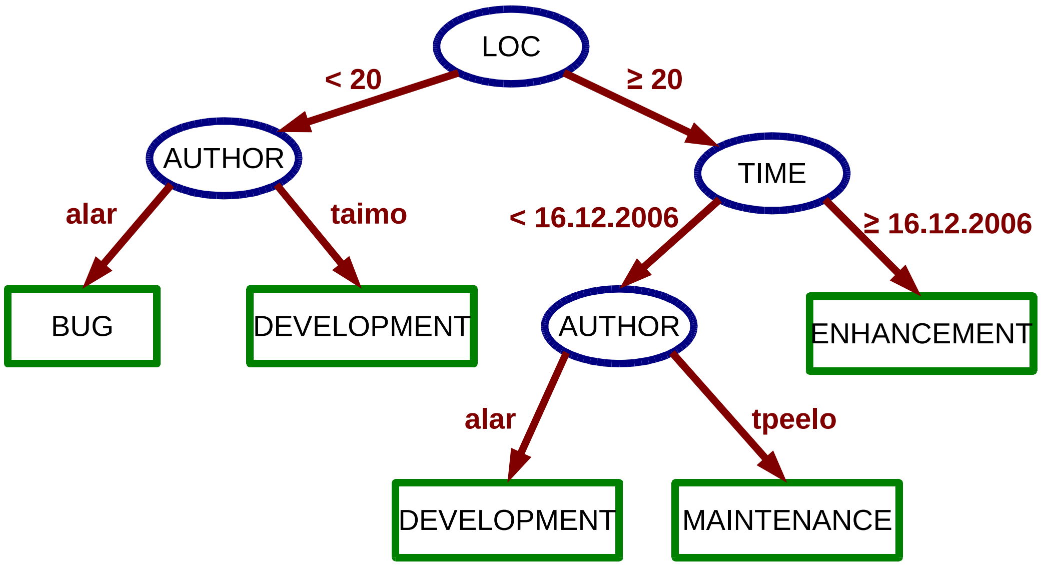


Figure 2.1: Example of the decision tree.

In the real life when we have to make an important decision we consult with our friends and experts lowering the risk of taking an incorrect decision. The same idea is behind the *bagging* technique: instead of using only one classifier instance for prediction we use many instances of the same classifier.

Each classifier is trained using instances randomly sampled, with replacement, from the initial training set. During the classification each classifier performs prediction on its own and the final result is the most frequent class among predictions of all classifiers. [WF05].

Randomisation is another technique that is used to improve performance of the classification. The idea is to introduce some sort of randomness into the classifier. Bagging and randomisation are quite similar in sense that bagging introduces randomness into the training set of the classification algorithm. An example of randomisation, applied to the decision tree classifier, is the following. Typically, during the tree construction, at each node we consider all available attributes and for the test select the attribute, which permits the best splitting of the data set instances into subsets. With the randomisation applied we take into account only some number of randomly chosen attributes and select the best one out of them. In this case randomisation ensures that every time even the same data set produces a different decision tree. The previous example illustrates exactly how random forest classifier employes randomisation [Bre01].

Random forest classifier takes 2 input parameters:

- number of decision trees to be constructed — value of this attribute influences the classification stability. If the number of trees is too small, then in some cases we might get a classifier with a high accuracy, in others — with low. The higher number of trees ensures more stable results.

- number of randomly selected attributes to be considered during the tree construction — this attribute affects overall accuracy of the classification. The value of this attribute must be much less than the number of attributes in the data set. It is advised to tune the value of this parameter.

To summarise, random forest classifier uses bagging to generate many decision trees and during the construction of each tree uses randomly chosen subset of attributes to create a test at each node of the tree.

Chapter 3

Methodology

This chapter starts with an overview of the method we propose to predict the type of source code contributions. The chapter continues with a step-by-step description of the method. Each step is illustrated with a simple example.

3.1 Method Overview

We propose the novel method for the prediction of the source code contribution type. The novelty of our method lays in the decision to use software metrics to perform the classification. Amor *et al.* have used commentaries, submitted by the developer, in order to discriminate between different types of contributions [ARGBN] (see Chapter 5 for details).

Usually, during the commit of the files to the versioning system, the developer is asked to provide a commentary describing the modifications. It is quite easy to make such commentaries mandatory, but it is much more complicated to ensure their meaningfulness without an additional burden. Contrarily, the usage of software metrics is much less invasive — a developer doesn't have to submit extra information or perform additional actions.

The proposed method consists of 10 steps depicted in Algorithm 1. The following section describes every step in detail.

Algorithm 1 Overview of the proposed method.

```
Query historical data from the versioning system
Group source code modifications into contributions
for all contributions do
    Checkout the version of code valid before the contribution
    Calculate metrics for the checked out code
    Checkout the version of code valid right after the contribution
    Calculate metrics for the checked out code
    Calculate delta values of metrics
end for
Ask expert to provide classes for the small number of contributions
Train Random Forest classifier
Evaluate the classification accuracy
```

3.2 Method Detailed Description

The first step is to obtain historical data of the source code from the versioning system. Each record in such data set represents the commit of a single modification of one file by a developer (Table 3.1).

Filename	**Time**	**Author**
AjaxRequestErrorWidget.java	20:10 26.04.07	alar
BaseServiceRouterService.java	20:10 26.04.07	alar
TemplateMenuWidget.java	15:24 28.04.07	taimo
StandardWizardWidget.java	15:24 28.04.07	taimo
RootWidget.java	15:24 28.04.07	taimo
EventButtonHtmlTag.java	16:38 28.04.07	taimo
SqlFunctionFilter.java	16:38 28.04.07	taimo
...	...	...

Table 3.1: Historical data obtained from the versioning system.

Usually one contribution consists of modifications of several files. So in order to obtain code contributions we group commit statements together using information about author, commentary and time of the commit (Table 3.2).

Filename	Time	Author
AjaxRequestErrorWidget.java	20:10 26.04.07	alar
BaseServiceRouterService.java	20:10 26.04.07	alar
TemplateMenuWidget.java	15:24 28.04.07	taimo
StandardWizardWidget.java	15:24 28.04.07	taimo
RootWidget.java	15:24 28.04.07	taimo
EventButtonHtmlTag.java	16:38 28.04.07	taimo
SqlFunctionFilter.java	16:38 28.04.07	taimo

Table 3.2: Modifications grouped into contributions.

After that we calculate the software metrics for each contribution. We start with checking out the version of code valid just before the contribution and calculate metrics values on this code. Then we checkout the version of the code valid right after the contribution and calculate metrics values. As a result for each modified file we have obtained metrics values valid at the beginning and at the end of the contribution (Table 3.3). If a file was deleted during the contribution, then there are no metrics values for this file at the end of the contribution. Similarly, if a file was created during the contribution, then its start metrics are missing.

Filename	$\mathbf{LOC_{start}}$	$\mathbf{LOC_{start}}$	...
AjaxRequestErrorWidget.java	134	168	...
BaseServiceRouterService.java	276	154	...
TemplateMenuWidget.java	342	342	...
StandardWizardWidget.java	94	231	...
RootWidget.java	379	-	...
EventButtonHtmlTag.java	143	143	...
SqlFunctionFilter.java	-	164	...

Table 3.3: Start and end metrics values.

During the next step we calculate delta values of all metrics for each file. Delta values are calculated as the value of the metric at the end of the contribution minus the value of the metric at the start of the contribution.

If a file has delta values of all metrics equal to 0, then this file is considered to be unchanged during the corresponding contribution (Table 3.4). If a file was deleted or created during the contribution then we substitute missing values with zeros and calculate delta values similarly to the general case.

Filename	**LOC$_{\text{delta}}$**	**...**
AjaxRequestErrorWidget.java	34	...
BaseServiceRouterService.java	-122	...
TemplateMenuWidget.java	***0***	***0***
StandardWizardWidget.java	137	...
RootWidget.java	-379	...
EventButtonHtmlTag.java	***0***	***0***
SqlFunctionFilter.java	164	...

Table 3.4: Delta values of metrics.

Then we ask an expert to manually classify the type of the small number of contributions. These values will be used to train the classifier and evaluate its accuracy (Table 3.5).

Filename	**LOC$_{\text{delta}}$**	**Type**
AjaxRequestErrorWidget.java	34	BUG
BaseServiceRouterService.java	-122	
TemplateMenuWidget.java	***0***	DEVELOPMENT
StandardWizardWidget.java	137	
RootWidget.java	-379	
EventButtonHtmlTag.java	***0***	MAINTENANCE
SqlFunctionFilter.java	164	

Table 3.5: Contrubutions manually classified by the expert.

Next we train random forest classifier and calculate its accuracy. Here we propose two possibilities for classification[1]. The first option is to aggregate

[1]These are not the only possible approaches. For instance, we may consider each contribution as an instance with a special (non-vectorial) structure. Then we will be able to apply kernel- or distance-based prediction methods.

metrics delta values over the contributions and classify the contributions themselves. The second option is to classify modified files and aggregate their classes into the contribution class. In the end we would like to test which approach performs better. Let's consider both of them in detail.

Approach 1. As we have already mentioned, each contribution consists of several file modifications. We start by aggregating metrics delta values over all files modified during the particular contribution. As the result we obtain delta values for contributions and may use contributions as separate instances in the training set (Table 3.6). After that we train the random forest classifier using this data set. Classification algorithm starts with construction of a large number of decision trees (see Section 2.4). A real part of such tree can be seen in Figure 3.1. Each tree is used to make its own prediction and the overall result is the most frequent class among all predictions. The accuracy of the classifier is evaluated using 10-fold cross-validation (see Section 2.3).

$\mathbf{LOC_{sum}}$	$\mathbf{LOC_{avg}}$	$\mathbf{LOC_{median}}$	**Type**
-88	-44	-44	BUG
-242	-80.67	0	DEVELOPMENT
164	82	82	MAINTENANCE

Table 3.6: Approach 1. Training set.

Approach 2. The second approach is to classify each modified file separately. We train the random forest classifier using data set containing file modifications as instances (Table 3.7). Figure 3.2 illustrates a top level part of the real decision tree constructed by the classifier during experiments.

The accuracy of the classifier is evaluated using an algorithm (Agorithm 2) similar to the "leave-one-out" validation (see Section 2). For each contribution, we use only modifications that belong to the remaining contributions as the training set. Then the classifier is asked to predict types of all modifications of the given contribution. We take the most frequent class to be the contribution's class. The overall accuracy of the classifier is the ratio of correctly predicted classes of contributions to the total number of contributions.

We have described the proposed method and illustrated it with simple examples. The next chapter contains the overview to the application of this method on the real open source software project.

Filename	$\mathbf{LOC_{delta}}$	**Type**
AjaxRequestErrorWidget.java	34	BUG
BaseServiceRouterService.java	-122	BUG
StandardWizardWidget.java	137	DEVELOPMENT
RootWidget.java	-379	DEVELOPMENT
SqlFunctionFilter.java	164	MAINTENANCE

Table 3.7: Appoach 2. Training set.

Algorithm 2 Approach 2. Accuracy estimation.

```
Require: $C-$ collection of contributions
Ensure: $ACCURACY-$ accuracy of the classification
  for all contribution $c \in C$ do
    $M^{\{c\}} \Leftarrow$ all modifications of the contribution $c$
    $M^{C-\{c\}} \Leftarrow$ modifications of all remaining contributions
    $CLASSIFIER \Leftarrow$ instance of the classifier
    train $CLASSIFIER$ using $M^{C-\{c\}}$
    classify $M^{\{c\}}$
    $CLASSES^{\{c\}} \Leftarrow$ classes of all modifications of the contribution $c$
    $RESULT \Leftarrow$ most frequent class in $CLASSES^{\{c\}}$
    if $RESULT$ is correct then
      $NO\_OF\_CORRECT \Leftarrow NO\_OF\_CORRECT + 1$
    end if
  end for
  $ACCURACY \Leftarrow \frac{NO\_OF\_CORRECT}{NO\_OF\_CONTRIBUTIONS}$
```

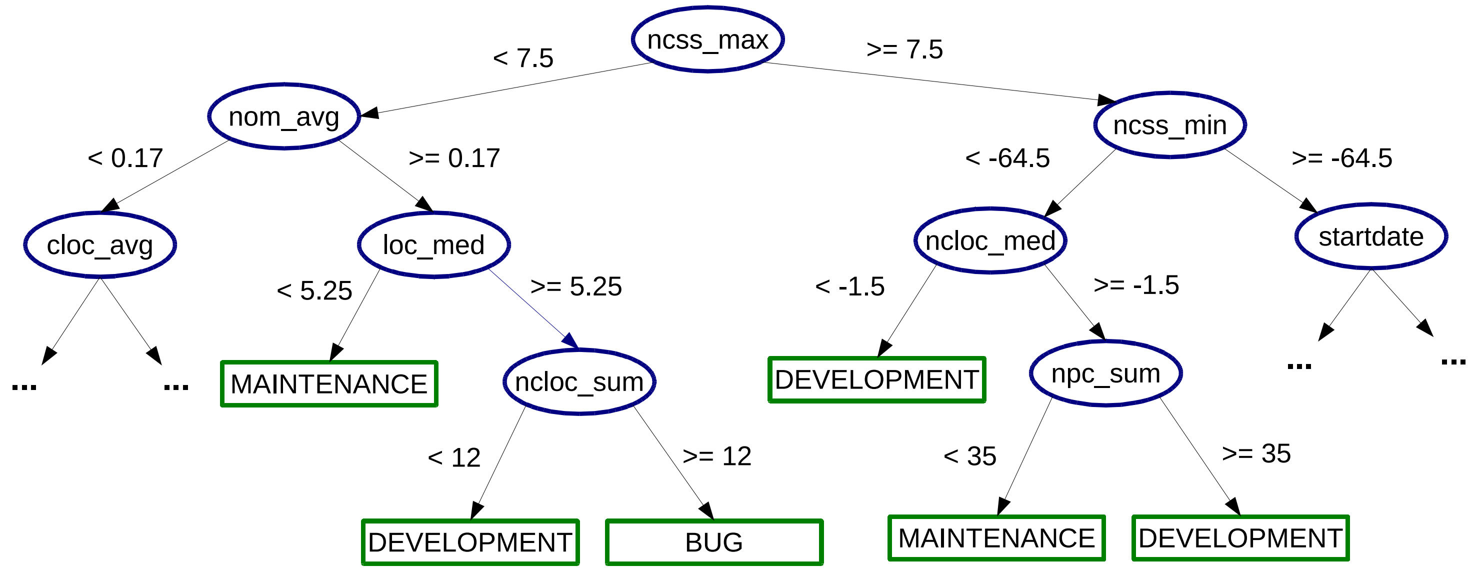


Figure 3.1: Example of the decision tree constructed by random forest classifier during the approach 1 experiments. Figure illustrates only a small part of the tree including its root and top level nodes.

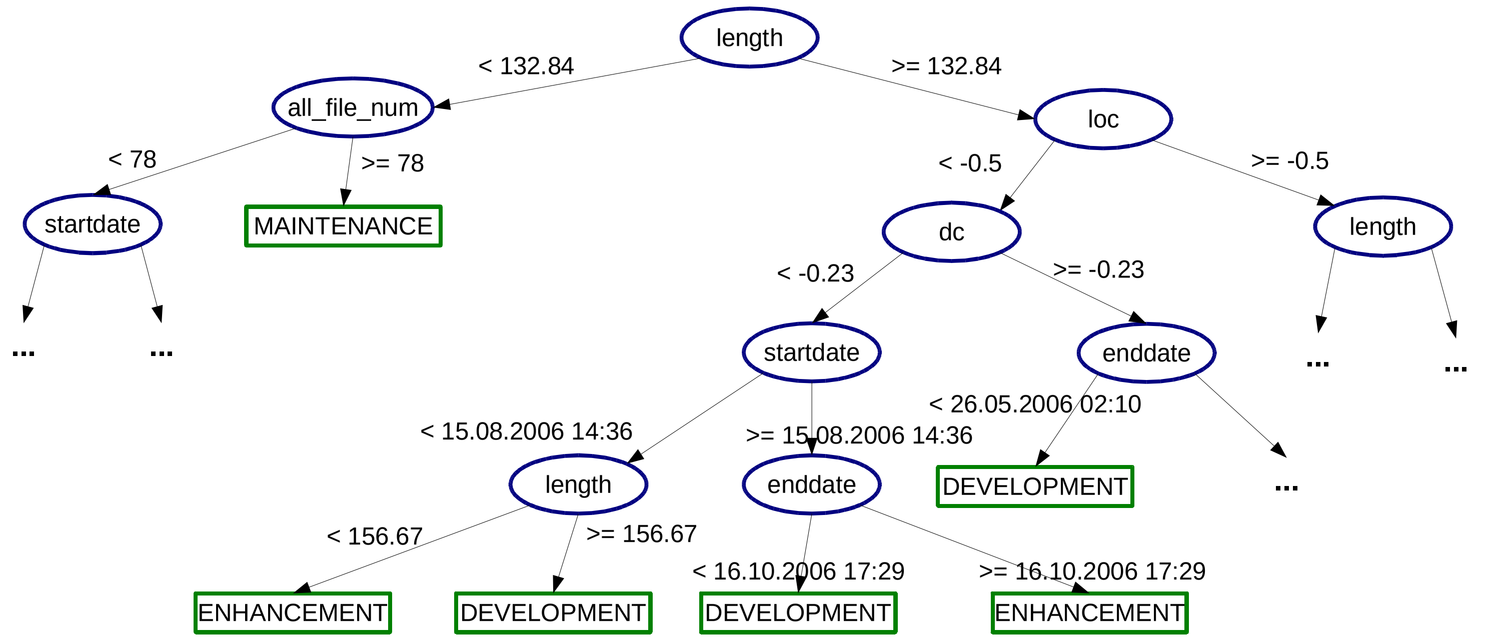


Figure 3.2: Example of the decision tree constructed by random forest classifier during the approach 2 experiments. Figure illustrates only a small part of the tree including its root and top level nodes.

Chapter 4

A Case Study: Aranea Framework

The previous chapter contained the description of the proposed method. Here we introduce the results of its application on the real open source project. The chapter starts with a description of the open source project and toolkit used during the case study. We continue with a presentation of the data sets used in experiments. The chapter finishes with an overview of results.

4.1 Aranea Framework

Aranea is an open source Java MVC[1] web framework that provides a common Object-Oriented approach for building web applications, reusing GUI logic and extending the framework. Additionally it serves as an integration platform, allowing free intermingling of arbitrary frameworks, components and applications [MK06].

The choice of Aranea was dictated by the fact that its development style ensures that each source code contribution has a manually defined type associated with it. This provides a natural data set for validation of the method described earlier.

[1] *Model-View-Controller* (MVC) — is an architectural pattern for building user interfaces. The core idea is to make clear division between domain objects that model application's perception of the real world, and presentation objects that are the user interface elements [Fow]

Throughout the development history the directory tree of Aranea Framework contained 9 different directories with Java source files. Two of them contain source code of the tests and the framework itself.

- `src/` — the source code of the framework
- `tests/src/` — the source code of the unit/integration tests

Seven remaining contain source code of the examples demonstrating the framework usage:

- `template/src/`
- `examples/main/src/`
- `examples/blank/src/`
- `examples/common/src/`
- `examples/widgetHelloName/src/`
- `examples/serviceHelloWorld/src/`
- `examples/serviceHelloName/src/`

During experiments we have used modifications made to all files in these directories.

4.2 Toolkit

4.2.1 Changelogic

In order to manage defect issues, feature requests, and code contributions the Aranea team uses Changelogic system. Changelogic is a configuration management tool based on CVS and is meant for small and medium size IT development and management enterprises as a support tool for software development processes [Cha]. Recently, SVN support has been added to Changelogic.

Changelogic uses the concept of a *change* to express source code contributions. The change binds together tasks and the real modifications of the source code: “while task is a description why and how to implement something, change is the realisation of that on code base” [Cha]. In terms of the versioning system every change is a separate branch. Hence, it is possible to get versions of code valid just before the change and right after finishing it.

Changelogic defines 4 types of changes:

bug — defect fixing, as a result the software becomes more stable

development — implementation of new requirements, as a result new defects are introduced and software is less stable

enhancement — stands in between of bug and development — the trigger for the task is similar to the one of the bug, but as a result functionality is altered and software doesn't become more stable

maintenance — modifications that do not alternate the functionality of the software, but may affect some of its aspects and introduce new defects.

In the case study we have used changes instead of contributions. The main difference between the change and the typical source code contribution to the open source project is granularity. While contributions usually contains smaller number of modifications, changes consist of several contributions grouped by the same goal. It is hard to foretell whether the usage of changes would improve or worsen the overall accuracy of the prediction. Although we haven't tested, we tend to think that the latter holds, because contributions represent smaller tasks. Hence, they are more focused and should represent the goal more accurately.

The proposed method was tested on all types of changes available from the Changelogic instance used in the Aranea development.

4.2.2 SourceKibitzer

SourceKibitzer is an online service that collects and measures programming metrics from open source java projects all over the web in order to get an idea about the quality of the code, member activity, development process and project size [Soub].

We have made several modifications to the SourceKibitzer platform so that it would serve the needs of the thesis. The modified version was used to collect metrics for the source code of Aranea Framework.

The usage of SourceKibitzer has helped us to calculate the list of 11 source code metrics for Aranea Framework:

Lines of Code (LOC) — counts the number of physical lines in the source file including blank lines, executable lines and comments.

Comment Lines of Code (CLOC) — counts number of lines in the source file that contain commentaries.

Non-Comment Lines of Code (NCLOC) — counts the number of lines in the source file that both don't contain commentaries and are not blank.

Density of Comments (DC) — the ratio of the number of commented lines of code (CLOC) to the number of all lines of code (LOC) in the source file. Illustrates how much of the source code is commented.

Non Commenting Source Statements (NCSS) — counts the number of source code statements excluding blank lines and commentaries [Jav]. This is less sensitive to the code style: one statement may occupy several lines, and both LOC and NCLOC would count these lines separately. NCSS would count them as a single statement.

Number of Methods (NOM) — counts the number of methods in the source file.

Weighted Method Count (WMC) — counts the number of methods using their McCabe's Cyclomatic Complexity as a weight. For a method McCabe's Cyclomatic Complexity counts number of linearly independent paths that may be taken during program execution [Van00].

Boolean Expression Complexity (BEC) — counts the total number of boolean operations in the source file.

Class Data Abstraction Coupling (CDAC) — measures the number of instantiations of classes within the given class. The higher the value of the metric the more complex data structure of the software [Che].

Class Fan Out Complexity (CFOC) — measures the number of files a given class relies on. Square of this value has been shown to indicate the amount of maintenance required in functional programs [Che].

NPath Complexity (NPC) — measures the number of possible execution paths through a function. Calculation takes into account the nesting conditional statements and multi-part boolean expressions.

All these metrics can be divided into three subgroups:

- metrics expressing the *size* of the source code — LOC, NCSS, NCLOC, and NOM
- metrics expressing the *complexity* of the source code — WMC, BEC, CDAC, CFOC, and NPC
- metrics expressing the various *commentary* properties of the source code — CLOC and DC.

Such categorisation demonstrates another limitation of this case study. We have been able to test how well the classifier is able to distinguish different types of contributions using quantitative representations of size, complexity and commentaries of the source code. At this time representations of software structure and design were left out due to difficulties in automation of their calculations.

4.2.3 Weka

Weka is a collection of machine learning algorithms for data mining tasks. The algorithms can either be applied directly to a data set or called from your own Java code. Weka contains tools for data pre-processing, classification, regression, clustering, association rules, and visualisation. It is also well-suited for developing new machine learning schemes. Weka was developed at the University of Waikato in New Zealand and its name stands for Waikato Environment for Knowledge Analysis [WF05].

We have used Weka implementation of the random forest classifier in order to predict types of changes.

4.3 Method Application

At the time of experiments, the Aranea's Changelogic instance contained 148 changes. The period of the development covered with these changes ranges from 28.03.2006 to 19.01.2007. The historical data was obtained from the

SVN versioning system used by the Aranea development team. The repository contained data of 147 changes (one change less than the Changelogic instance). Among them there were 60 BUG-s, 48 DEVELOPMENT-s, 17 ENHANCEMENT-s and 22 MAINTENANCE-s.

Metrics calculation. We have started with calculation of source code metrics. For each change we have measured all source files at the beginning of the change and at the end. Thus, for each file of the change we have calculated two collections of metrics values: change start values and change end values.

Totally, we have measured 1400 unique source files written in the Java language. This resulted in 181060 measurements, where each measurement contained values of 11 metrics valid for the particular source file of the particular version.

The calculation of metrics values involved parsing of the source code. If a file wasn't syntactically correct then it was impossible to calculate its metrics. This happened to be a problem with 2 files:

- `src/org/araneaframework/jsp/tests/StringUtilTest.java`
- `tests/src/org/araneaframework/tests/jsp/StringUtilTest.java`

For these files we weren't able to calculate values for LOC, CLOC, NCLOC, NOM and DC metrics. During the experiments all measurements of these two files were removed from data sets.

Deltas calculation. We have continued with a calculation of the metrics delta values. Values valid at the start of the change were subtracted from the values valid at the end of the change. The total number of obtained instances was 110308. Each instance contained all delta values for the particular file of the particular change.

Unfortunately, there was a large number of instances which delta values contained only zeros. This means that corresponding files either weren't modified during the change or used set of metrics couldn't express the modifications. An example of such modification is rename of the method with the length of the new name being close to the length of the old name. In such case all, the data structure, the number of lines and the number of statements remains unaltered.

Instances containing only zero values were removed from the data set. They do not help in classification, since all of them express the same pattern (all values are equal to 0) independently of the change type. The number of remaining instances was 3040. This also has caused the removal of 16 changes — all of them contained only instances with all delta values equal to zero. Among them there were 11 BUG, 3 DEVELOPMENT, 1 MAINTENANCE and 1 ENHANCEMENT changes. As a result 131 changes remained.

Further, the data set contained a clearly visible outlier. While average value for the NPC metric was 5.379 and standard deviation 162.595, there was one instance with delta value of the NPC equal to 691206. This instance corresponded to an addition of the file `src/org/araneaframework/http/CustomProxyHandler.java` during the change number 31. The file contained large number of sequential control statements causing such a high value. We have removed this instance from the data set.

So the full data set contained 3039 instances, 131 changes with 49 BUG-s, 45 DEVELOPMENT-s, 16 ENHANCEMENT-s and 21 MAINTENANCE changes among them.

Data sets. The proposed method was tested on 12 different data sets. We have used 3 conditions to derive these data sets from the full data set:

- exclusion of some source code directories
- exclusion of instances with small delta values
- inclusion of metrics that characterise changes, like length of the change, number of files modified during the change and others.

The summary of the all constructed data sets is available in Table 4.1.

Let's cover all data set construction conditions in detail. The first one, exclusion of some source code directories, was influenced by the fact that modifications made to the source code of examples and tests might not express the goal of the change. In order to test this assumption we have formed 3 data sets out of the full one:

a) modifications made to files in all 9 source code directories (data set number 4 in Table 4.1)

b) modifications made to — `src/` and `tests/src/` directories (data set number 8)

c) modifications made only to the files in `src/` directory (data set number 12).

data set number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
directories used												
src/	✓				✓				✓			
tests/src/	✓				✓							
template/src/	✓											
examples/blank/src/	✓											
examples/common/src/	✓											
examples/main/src/	✓											
examples/serviceHelloName/src/	✓											
examples/serviceHelloWorld/src/	✓											
examples/widgetHelloName/src/	✓											
LOC, NCSS or NPC > 9	✓				✓				✓			
source code metrics	✓		✓		✓		✓		✓		✓	
change metrics	✓		✓		✓		✓		✓		✓	
number of modifications	1533	1533	3039	3039	1238	1238	2431	2431	1169	1169	2306	2306
number of changes	109	109	131	131	106	106	128	128	105	105	128	128

Table 4.1: Description of the datasets used in the experiments

Behind the second condition, exclusion of instances with small delta values, there is an intuition that files that were modified the most express the goal of the change more accurately. In other words, if a file was only slightly altered then it is a carrier of peripheral modification that doesn't provide interesting information to the classifier. To express this intuition we have used the following condition:

$$\text{delta value of at least one of the LOC, NCSS or NPC is larger than 9} \tag{4.1}$$

Each existing data set was used to built one additional by removing all instances not satisfying Condition 4.1. As a result 3 additional data sets were created: 2, 6, and 10 (Table 4.1).

Finally, we have used the third condition to create 6 more data sets: to each already existing data set we added number of metrics that characterise changes:

- start date of the change
- end date of the change
- length of the change in hours
- number of modified files during the change
- number of files satisfying Condition 4.1
- ratio of the number of files not satisfying Condition 4.1 to the number of modified files during the change

These metrics were added with a goal of testing if change specific information influences the accuracy of the prediction. Numbers of new data sets in Table 4.1 are 1, 3, 5, 7, 9, and 11.

The next step was the classification itself. We have applied the random forest classifier to all these data sets. In the methodology chapter we have introduced two approaches for the classification. Let's cover their application separately and after that evaluate the overall result.

4.4 Classification Using Aggregated Modifications

The central idea of the first approach is to classify changes themselves. All data set instances were grouped by changes and their delta values were aggregated. For each source code metric we have calculated 5 aggregated values: minimum, maximum, sum, average and median. Values of the change specific metrics were taken as-is because their values are equal for all instances belonging to the same change.

Every instance in data sets containing both source code metrics and change metrics (data sets 1, 3, 5, 7, 9, 11) had 62 attributes: 55 source code metrics, 6 change metrics, and 1 class attribute. Correspondingly, instances from data sets with only source code metrics (data sets 2, 4, 6, 8, 10, 12) had 56 attributes.

We had to specify values for two input parameters of the random forest classifier:

- number of trees to be generated and afterwards used in the prediction — due to the fact that number of instances in each data set was quite small (ranging from 109 to 131) we had to pay special attention to the stability of the prediction. Hence, we have decided to generate 1000 trees.

- number of random attributes used during construction of the tree — according to the specification of random forest classifier value of this parameter should be much less than the total number of attributes in the data set. We have decided to check how the value of this parameter affects the prediction accuracy. Values of the parameter were taken from the interval 1, 2, ..., 10. For each setting (data set, number of attributes, number of trees) we have performed 10 experiments.

Totally, 1200 experiments were performed to test accuracy of the first approach — 100 experiments per every data set.

The peak accuracy achieved during experiments was correct classification of 55.47% of changes. We have observed such result for 4 times with the following settings:

- data set number 7, 3 attributes, 1000 trees

- data set number 7, 8 attributes, 1000 trees — 2 times
- data set number 11, 2 attributes, 1000 trees.

In the average the best accuracy was produced with the data set number 7 using 8 random attributes for the tree construction — 52.27% of changes were classified correctly.

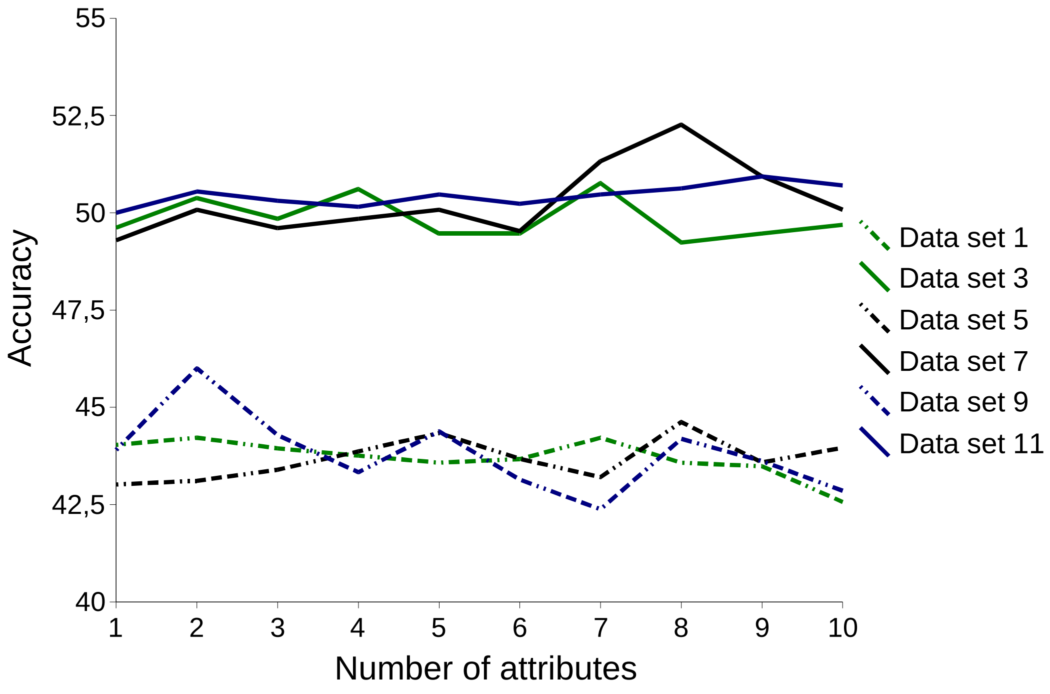


Figure 4.1: Approach 1. A chart illustrating the performance of the random forest classifier on the data sets containing both source code and change metrics (data sets number 1, 3, 5, 7, 9, 11).

Results of all experiments are available in Appendix A. Here, we only present charts that depict the average classifier performance depending on the setting (see Figures 4.1 and 4.2).

Interestingly, data sets containing all instances (data sets number 3, 4, 7, 8, 11, 12) have performed better than data sets containing only instances satisfying Condition 4.1 (data sets number 1, 2, 5, 6, 9, 10). The average difference of accuracy between them was 6.45%. Such a high margin have

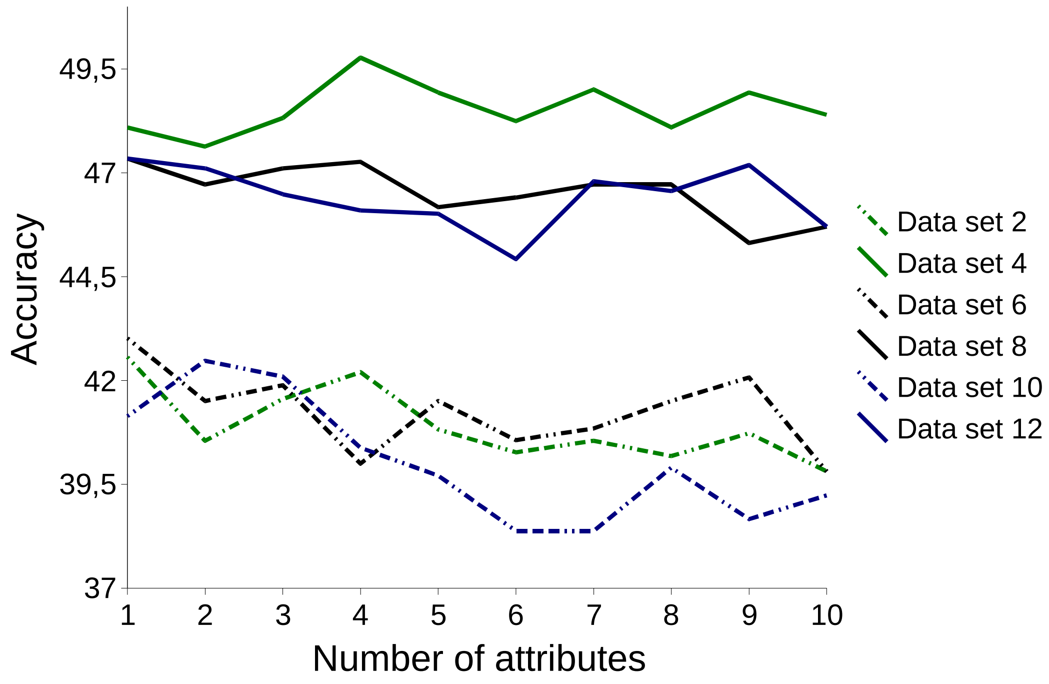


Figure 4.2: Approach 1. A chart illustrating the performance of the random forest classifier on the data sets containing only source code metrics (data sets number 2, 4, 6, 8, 10, 12).

led us to the conclusion, that even small delta values carry the information useful for making predictions.

It is worth noticing that usage of change metrics (start date, end date, number of files etc) in the average improved accuracy by 3.01%. Based on this observation we conclude that not only source code metrics should be used for classification but also quantitative information concerning the change itself.

4.5 Classification Using Separate Modifications

The idea of the second approach was to predict type for all data set instances separately and then aggregate predicted types over the changes. The total number of attributes for data sets containing both source code and change metrics (data sets number 1, 3, 5, 7, 9, 11) was 16: 11 source code metrics, 4 change metrics, and a class attribute. This time set of change specific metrics included:

- start date of the change
- end date of the change
- length of the change in hours
- number of modified files during the change.

Instances from the data sets that contained only source code metrics (data sets number 2, 4, 6, 8, 10, 12) had 12 attributes: 11 source code metrics and a class attribute.

During the test of the second approach we have used random forest classifier with slightly different settings:

- number of generated trees — the size of the data set comparing to the previous experiments was a magnitude larger: the smallest data set contained 1169 instances and the largest — 3039 instances. Hence, we were able to decrease the number of trees to be 500.

- number of random attributes — data sets, used for experimenting with the second approach, contained at most 15 attributes eligible for usage during the tree construction. We have decided to try different values for this parameter from the interval 1, 2, . . . , 5.

This time the number of experiments was much smaller — each data set was used for prediction for 5 times: one time for each possible value of the "number of random attributes" parameter. We explain this by the fact that large size of the data sets should ensure stable results. Another argument is usage of a more sophisticated and time-consuming algorithm for the evaluation of the classifier accuracy (Alogrithm 2). Basically, for each change we have built a random forest classifier using modifications from the remaining changes and then checked its accuracy on the modifications of the given change.

The table containing all results can be found in Appendix B. Here we outline only the most interesting facts.

The best classifier has correctly classified 44.53% of changes. Top four results were produced by the following settings:

- data set number 7, 1 attribute, 500 trees — 44.53%

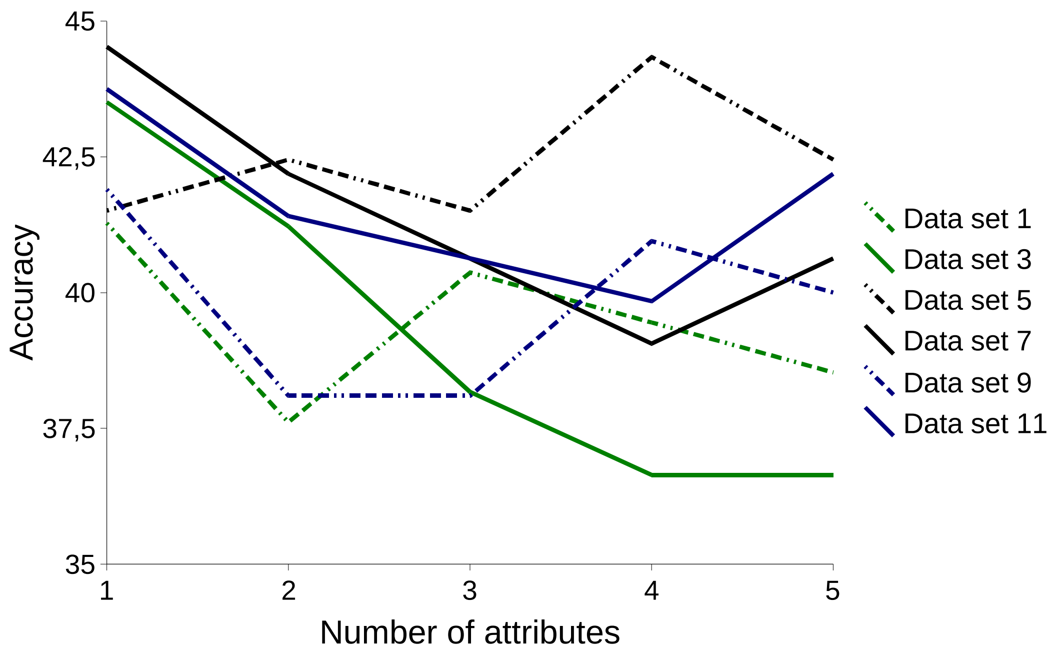


Figure 4.3: Approach 2. Chart that illustrates performance of the random forest classifier on the data sets that contain both code and change metrics (1, 3, 5, 7, 9, 11).

- data set number 5, 4 attributes, 500 trees — 44.34%
- data set number 11, 1 attribute, 500 trees — 43.75%
- data set number 3, 1 attribute, 500 trees — 43.51%.

All of these settings included data sets containing both source code and change metrics. While data sets number 3, 7 and 11 contained all instances, the data set number 5 consisted of the instances satisfying the condition 4.1.

From Figure 4.3 we can see that classifier accuracy achieved with data sets containing all instances and data sets constructed with Condition 4.1 was almost equal: in some case the former (data sets number 3, 7, 11) performed better, in other cases — the latter (data sets number 1, 5, 9). The average difference was −0.17% (negative) and the median — 0.26%

Interestingly, the classification of data sets containing all modifications with only source code metrics (data sets number 4, 8, 12) didn't produce any results. The random forest classifier wasn't able to construct any tree

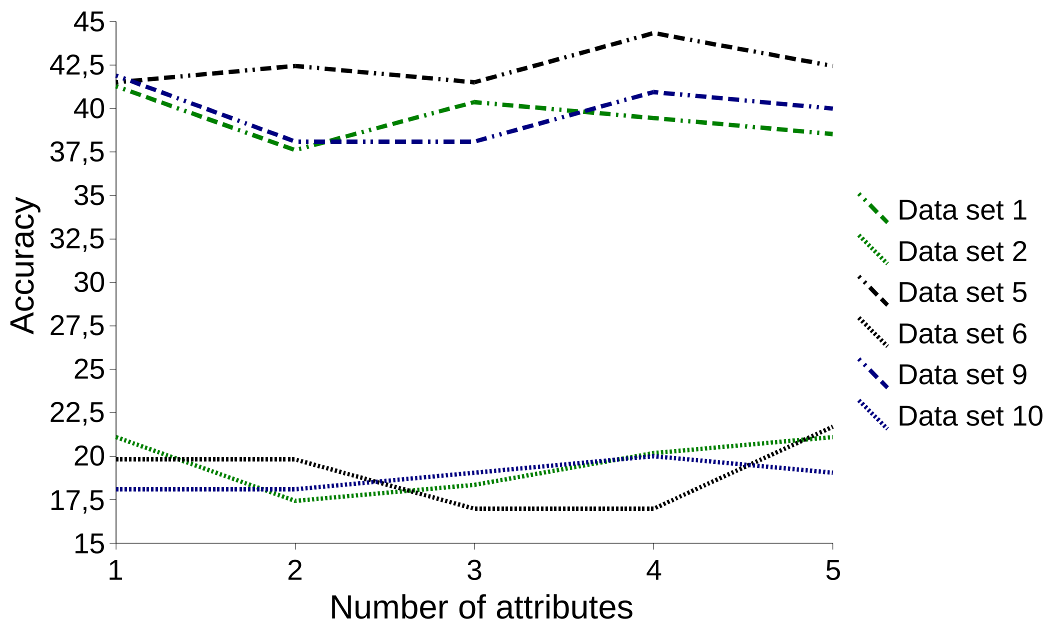


Figure 4.4: Approach 2. Chart that compares performance of the random forest classifier on the data sets that contain both code and change metrics against the data sets containing only code metrics (1, 2, 5, 6, 9, 10).

— it went into very deep recursion and caused “out of memory” and “stack overflow” errors. We have concluded, that this was caused by the controversy in the data which didn’t permit the splitting of the instances into to the different tree branches according to the class.

Fortunately, we were able to see how the existence of change metrics affected the accuracy of the classifier on the data sets satisfying Condition 4.1 (Figure 4.4). The average difference in accuracy was very large — 21.39%. Such increase could be explained by the fact that quantitative data about the change helps classifier to relate all modifications belonging to the same change and provide them with a more consistent prediction of the type.

4.6 Results Overview

We have had expectations of the prediction accuracy to be around 70% — the accuracy achieved in the similar research conducted by Amor *et al.* [ARGBN].

But, during the experiments we weren't able to predict the type of changes with accuracy higher than 55.47%. We don't consider this to be a poor result, because of the several facts.

First of all, we are able to conclude that approach of aggregating metric values over the changes and then classifying them (approach 1) performed better than classification of modifications and subsequent aggregation of modification types into the change type (approach 2). While the highest accuracy achieved by the second approach was 44.53%, the average accuracy of the first approach with the best setting was 52.27%. In comparison, the predictor, that would classify all changes as BUG-s, would have been correct in 37.40% of the cases.

Secondly, we have demonstrated how choices made during the data set construction affect the performance of the classifier:

- Prediction of the data set containing all instances is more accurate than the one of the data sets with only instances satisfying the condition of "delta value of at least one of the LOC, NCSS or NPC being larger than 9" (Condition 4.1). In the tests of the first approach the former performed better than the latter by 6.45% in the average. Tests of the second approach demonstrated equal accuracy.

- Addition of the quantitative information concerning the change improves the accuracy. The average increase for the first approach was 3.01%, for the second — 21.39%.

- Throughout the experiments there was no clearly visible difference in the prediction accuracy for the data sets containing file modifications from all resource directories and data sets with modifications from some subset of directories.

Thirdly, our later quick experiments have demonstrated that random forest classifier was able to distinguish BUG changes from other changes with the accuracy equal to 70%. Our intuition tells us that BUG changes are usually smaller in size and values of source code metrics change only a little. The classifier was able to use this information for prediction. Other types of changes (DEVELOPMENT, ENHANCEMENT and MAINTENANCE) are not so easily distinguishable and classifier needs an additional information. We think that the needed information may be provided by the more diverse set of metrics. Such set would include not only size and complexity metrics, but also quantitative expressions of the structural and design alternations of the software.

Chapter 5

Related Work

In this chapter we review articles and tools that solve similar problems in the same area.

First of all, we would like to emphasise the article by Amor *et al.* titled "Discriminating Development Activities in Versioning Systems: A Case Study" [ARGBN]. Although the purpose of their work is different from ours — automation of effort estimation, the goal is similar: classification of the developer contributions using the historical data available from the versioning system. Authors have defined 4 types of contributions:

corrective — defect fixing

adaptive — implementation of new functionality

perfective — code refactorings and modifications in source code documentation

administrative tasks — changes in copyright, version number.

Authors have started with grouping of code modifications into contributions, just like we did. After that the expert was asked to classify manually small number of contributions. This information along with the contribution commentaries was used to train the naive Bayes classifier. Finally, accuracy of the classifier was evaluated on the remaining data. The approach was tested using the historical data of the FreeBSD operating system [Fre]. The number of contribution was 33335 and expert has provided classes for 500 of them. The classifier predicted type correctly in more than 70% of the cases verified by the expert.

The main difference between the method proposed by Amor *et al.* and our method is the source of the information used for classification: we have used values of software metrics and Amor *et al.* have used commentaries submitted by developers. Unfortunately, commentaries are side-effects of the commit: one can ensure existence of the commentary, but it is difficult to ensure its meaningfulness. On the other hand if a source code was modified, then there is also a change in the values of metrics. Hence, no commentaries are needed to apply our method — to obtain the data we use the core result (source code) of the developer's work.

The second article that inspired our study is "An empirical study of fine-grained software modifications" by German *et al.* [Ger06]. Authors have tried to find answers for several questions. For instance:

- Are defect fixing contributions look differently from contributions intended to add new functionality?
- Do contributions made during the different stages of development look differently?
- Can we create metrics that describe contributions?

During their experiments authors have used historical data of the Evolution mail client [Evo]. Among other results described in the article, two were particularly interesting for us. First of all, authors have found that the contributions intended to fix defects typically contain smaller number of files than the ones used to improve the code documentation: 2.95 files in average versus 13.48. We have applied this observation in our experiments by adding the "number of modified files" metric to the data sets.

The second interesting observation is that during the software maintenance period the number of contributions is smaller than during the period when new functionality is being implemented. This observation provided us with an intuition of how project's state may be predicted: one can calculate the number of contributions during the last month and if it is smaller than some threshold n then no new features are being developed and project is being only maintained.

Finally, a very interesting article by Singh *et al.* titled "An Empirical Investigation of Code Contribution, Communication Participation, and Release Strategy in Open Source Software Development: A Conditional Hazard

Model Approach" provides results of investigation how development characteristics, like practices in code contribution, participation in communication, and release strategy, influence the progress of the open source software project [SFT07].

The study includes data from more than 200 open source projects hosted at the SourceForge [Soua] — the largest hosting services provider for open source projects. Authors have defined a hazard function that specifies the probability density (over time) of releasing a stable version of the software given that the team doesn't have a stable release up to the given time. The function's definition included the following metrics:

GDEV — Gini index of developer code contributions [Dam]

GCOMM — Gini index of developer communication contributions

NTHREAD — Total number of threads in the mailing list

THDEPTH — Average thread depth of the mailing list for a project

NRLEASE — Total number of beta packages released by a project prior to the stable release

NDLOAD — Total number of downloads prior to the stable release

NDVLPR — The number of unique developers that have contributed coding

SIZE — Total number of files of the source code in the stable release.

Using this function the authors have found out that chances of releasing first stable version raise during first 4 years of the development, but then they drop significantly. Secondly, 20% of the most active developers contribute 80% of the source code. Almost the same group of developers produce 81% of the messages posted to the project mailing list. Thirdly, authors have made a conclusion that probability of being successful increases with the existence of the core developer group. At the same time domination of a small group of people in communication leads to unhealthy group dynamics and lowers the level of innovation and knowledge sharing in the community.

Not only researchers have been attracted by the success of open source software — there is a number of initiatives that try to make open source software even more open and provide the data to make process of its evaluation more simple. The most noticeable are:

SourceKibitzer — an online service started in 2006 [Soub]. The team behind SourceKibitzer plans to create a knowledge base that would add transparency to open source Java projects through analysis, benchmarking and criticism. At the moment (April 2007) it hosts the data of nearly 600 open source projects written in the Java language. The focus on one programming language enables SourceKibitzer to collect a deeper set of software metrics.

Ohloh — another online service founded in 2004 as a way to provide more visibility into software development [Ohl]. Founders characterise it as a resource for open source intelligence of thousands of open source projects. Ohloh uses the source code of the project and infrastructure used by the team to collect its metrics. According to various resources by the April 2007 Ohloh has indexed from 3000 to 4200 projects [Ohl, Wik07].

Business Readiness Index — a framework for rating open source software [Busa]. Rating is calculated as a weighted sum of scores in several categories like functionality, usability, security, scalability and so forth [Busb]. Authors propose the framework as a trusted and open standard for open source software evaluation. Currently (April 2007), the download section of the site lists evaluations of 17 projects.

Conclusions

Open source software is gaining acceptance and support from the larger number of commercial, public and governmental organisations. But, before making the decision to start using the particular solution organisations carefully evaluate all possibilities. Otherwise, an insufficiently thought-out selection may compromise the development speed, the stability, maintainability and availability of the application. Thus, the ability to objectively evaluate open source solutions is becoming crucial to success.

We have introduced the novel approach for the automatic prediction of the source code contribution type. Such prediction provides an objective insight into the progress of the open source project and helps to evaluate its current state. The novelty of our method lays in the application of software metrics to perform the prediction. Usage of software metrics doesn't impose additional constraints on the development.

Our method was tested on the real open source project: Aranea Framework [MK06]. We have calculated values of 17 software metrics for each contribution to Aranea, aggregated and combined this data in various manners (Table 4.1) and performed several experiments on it.

The overall accuracy of the classification was poorer than we had expected. We were looking for the accuracy around 70%. In reality, we were able to achieve a peak accuracy of 55.47%. Nevertheless, we consider results to be positive.

First of all, we have found that classification of the contributions with aggregated metrics values is more accurate than classification of the source code modifications and subsequent aggregation of the results into the contribution type. The former approach produced the correct type for $45 - 50\%$ of contributions, depending on the particular data set, with the maximum equal to 55.47% (Section 4.4). The best result of the latter approach enabled correct prediction of the contribution type in 44.53% of cases (Section 4.5).

In comparison, the classifier, that would predict the types of all contributions to be equal to the most frequent type in the data set, would have been correct in 37.40% of the cases.

Secondly, the classifier makes heavy usage of the metrics that characterise the contribution itself, like length in hours, number of files and others. Addition of such metrics to the data sets has improved the classification accuracy. For the data sets with metrics values aggregated over the contributions average improvement was 6.45%. Classification using the source file modifications separately was improved by 21.39% in average.

Thirdly, we have shown that assumption that files modified the most express the type of the contribution more informatively doesn't hold. In fact, the inclusion of all modifications independently of their metrics values has generally increased the accuracy by 3.01%.

All these facts have led us to a conclusion that in order to ensure the better accuracy of the prediction the data set should include all modifications, every modification should be characterised by both source code and contribution metrics and modifications should be aggregated over contributions prior to classification.

Finally, we have demonstrated the potential of usage of software metrics for the prediction of the contribution type and we hope that our work would serve as a solid base for further experiments. For instance, it would be interesting to test the accuracy of the method using more diverse set of metrics. Inclusion of metrics that express the alternation of the code structure, hierarchy of classes and modularisation would surely provide a classifier with an additional information to be used for discrimination of contribution types.

It is very important to test the proposed method on the larger number of open source projects. Only then we will be able to objectively evaluate its performance.

Throughout the experiments we have noticed interesting patterns in the data. For instance, sometimes delta metrics of two files in the same contribution had exactly opposite values — this is a clear sign that one file was moved or renamed during the contribution. Such reorganisation might be specific to the MAINTENANCE contributions. We were able to see other patterns as well: (un)commenting of a source code block, extraction of a functions from the file, and so on. It would be interesting to annotate all such patterns and then analyse associations between them and types of contributions.

We expect that the implementation of the proposed method will be used by a larger tool in order to evaluate open source software projects. The tool will combine the types of source code contributions with the similar knowledge obtained from the mailing list and issue tracking system of the same project, and perform automatic analysis of the whole information. As the result the state of the open source software project will be evaluated. The classification of the source code contribution type will play a significant role in such analysis.

Lähtekoodi muudatuste tüüpi automaatne ennustamine

Magistritöö (40AP)

Anton Litvinenko

Sisukokkuvõte

Äriühingud ja riigiasutused kasutavad ja toetavad üha enam avatud lähtekoodiga tarkvaralahendusi. Enne konkreetse lahenduse kasutuselevõttu tuleb aga alternatiive hoolikalt hinnata. Läbimõtlemata valik seab asutuse tegevuse ohtu — missiooni- ja ärikriitilised lahendused võivad muutuda ebastabiilseteks või kättesaamatuteks, arenduse kiirus võib kannatada ning hooldustööde maksumus kasvada kordades. Objektiivne hinnang avatud lähtekoodiga tarkvara küpsusele muutub otsustavaks edukuse faktoriks.

Avatud lähtekoodiga tarkvara on tuntud oma avalikuse tõttu. Selle lähtekood on vabalt kättesaadav ning arendusprojekti seltskond ja infrastruktuur on läbipaistvad. Kõik see võimaldab koguda tarkvara oleku hindamiseks vajalikku infot. Sellise analüüsi teostamine käsitsi on aga ajakulukas ning rutiinne.

Käesolevas töös tehakse esimesed sammud avatud lähtekoodiga tarkvara hindamise automatiseerimise suunas. Lähenemise põhieelduseks on seose olemasolu tarkvara oleku ja koodimuudatuse iseloomu vahel. Näiteks, juhul kui viimaste muudatuste hulgas on parandatud peamiselt vigu, siis tarkvara stabiliseerub ning lähiajal on oodata uut väljalaset. Samas, kui viimaste muudatuste eesmärk on olnud peamiselt uute funktsionaalsuste lisamine, on tarkvara alles aktiivses arendusprotsessis.

Käesoleva töö eesmärk on töötada välja meetod, mis võimaldab koodimuudatuste tüüpe automaatselt määratleda. Eesmärgi saavutamiseks on kavas analüüsida tarkvarameetrikate väärtusi, sest iga koodimuudatus toob endaga kaasa ka ka teatud meetrikate väärtuste languse või kasvu. Näiteks suurendab iga uue funktsiooni lisamine meetrikate "koodiridade arv" ja "funktsioonide arv" väärtusi.

Magistritöö esimeses kahes peatükis antakse ülevaade avatud lähtekoodiga tarkvaraarendusest, versioonihaldussüsteemidest, tarkvarameetrikatest ja andmekaevanduse tehnikatest. Versioonihaldussüsteemid aitavad jälgida tarkvara muudatusi ning kätte saada lähtekoodi varasemad versioonid. Andmekaevanduse tehnikaid kasutatakse muudatuste tüüpide ennustamiseks.

Kolmandas peatükis kirjeldatakse pakutud meetodit ja illustreeritakse seda lihtsate näidetega. Neljandas peatükis kirjeldatakse meetodi rakendamist reaalse avatud lähtekoodiga projekti Aranea Framework peal. Tähelepanu pööratakse erinevatele ennustuse täpsust mõjutavatele aspektidele: pisimuudatuste väljafiltreerimine, lähtekoodi failide valikkasutus ja meetrikate komplekti valik. Peatüki lõpus demonstreeritakse meetodi täpsust erineval viisil konstrueeritud andmehulkade peal. Põgusalt kirjeldatakse ka potentsiaalseid viise meetodi täpsuse parandamiseks.

Töö tulemused näitavad, et tarkvarameetrikad on rakendatavad koodimuudatuste iseloomu ennustamiseks. Paremaid tulemusi oodati meetodi rakendamisel ennustuse täpsuse osas, samas võib öelda, et tulemusi mõjutas kindlasti kasutatud meetrikate hulga ühekülgsus. Meetodile täpsema hinnangu andmiseks vajab see testimist mitmekülgsema tarkvarameetrikate hulgaga suurema arvu projektide peal.

Bibliography

[ARGBN] J.J. Amor, G. Robles, J.M. Gonzalez-Barahona, and A. Navarro. Discriminating Development Activities in Versioning Systems: A Case Study. http://gsyc.es/~jjamor/research/papers/2006-promise-jjamor-robles-barahona-anavarro.pdf. [Online; accessed 10-May-2007].

[BCR94] V.R. Basili, G. Caldiera, and H.D. Rombach. The Goal Question Metric Approach. *Encyclopedia of Software Engineering*, 1:528–532, 1994.

[Bit] BitKeeper. http://www.bitkeeper.com/. [Online; accessed 14-May-2007].

[Bre01] L. Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[Busa] Business Readiness Rating. http://www.openbrr.org/. [Online; accessed 04-May-2007].

[Busb] Business Readiness Rating for Open Source — White paper. http://www.openbrr.org/wiki/images/d/da/BRR_whitepaper_2005RFC1.pdf. [Online; accessed 04-May-2007].

[Cha] Changelogic. http://www.changelogic.com/GettingStarted/CreateChange. [Online; accessed 27-April-2007].

[Che] Checkstyle. Checkstyle Metrics Definition. http://checkstyle.sourceforge.net/config_metrics.html. [Online; accessed 27-April-2007].

[CVS] Concurrent Versions System (CVS). http://www.nongnu.org/cvs/. [Online; accessed 14-May-2007].

[CWL+07] K. Crowston, K. Wei, Q. Li, U. Y. Eseryel, and J. Howison. Self-organization of teams in free/libre open source software development. *Information and Software Technology Journal*, 49:564–575, 2007.

[Dam] C. Damgaard. Gini Coefficient. From MathWorld — A Wolfram Web Resource, created by Eric W. Weisstein. http://mathworld.wolfram.com/GiniCoefficient.html. [Online; accessed 04-May-2007].

[DeM82] T. DeMarco. *Controlling Software Projects: Management, Measurement & Estimation. Yourdon Computing Series.* Prentice-Hall, Inc., Englewood Cliffs, NJ, USA, 1982.

[Evo] Evolution Mail Client. http://www.gnome.org/projects/evolution/. [Online; accessed 14-May-2007].

[FLOa] FLOSSmole – collaborative collection and analysis of free/libre/open source project data. http://ossmole.sourceforge.net/. [Online; accessed 04-May-2007].

[FLOb] FLOSSWorld. http://www.flossworld.org/. [Online; accessed 14-May-2007].

[Fow] M. Fowler. GUI Architectures. http://www.martinfowler.com/eaaDev/uiArchs.html. [Online, accessed 08-May-2007].

[Fre] FreeBSD Operating System. http://www.freebsd.org/. [Online; accessed 14-May-2007].

[Ger06] D.M. German. An empirical study of fine-grained software modifications. *Empirical Software Engineering*, 11(3):369–393, 2006.

[GGKR02] R.A. Ghosh, R. Glott, B. Krieger, and G. Robles. Free/Libre and Open Source Software: Survey and Study, Part 4: Survey of Developers. *International Institute of Infonomics, University of Maastricht*, 2002.

[Git] Git. http://git.or.cz/. [Online; accessed 14-May-2007].

[GSy] GSyC/LibreSoft Research Group. http://libresoft.urjc.es/. [Online; accessed 14-May-2007].

[HK00] J. Han and M. Kamber. *Data Mining: Concepts and Techniques.* Morgan Kaufmann, 2000.

[HMS01] D.J. Hand, H. Mannila, and P. Smyth. *Principles of Data Mining.* Bradford Book, 2001.

[Ini] Open Source Initiative. Open Source Definition. http://www.opensource.org/docs/osd. [Online; accessed 27-April-2007].

[Jav] JavaNCSS. JavaNCSS Metrics Definition. http://www.kclee.de/clemens/java/javancss/. [Online; accessed 27-April-2007].

[Kan02] S.H. Kan. *Metrics and Models in Software Quality Engineering.* Addison-Wesley, 2002.

[LFRGB04] L. Lopez-Fernandez, G. Robles, and J.M. Gonzalez-Barahona. Applying Social Network Analysis to the Information in CVS Repositories. *International Workshop on Mining Software Repositories*, 2004.

[MK06] Oleg Mürk and Jevgeni Kabanov. Aranea: web framework construction and integration kit. In *PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 163–172, New York, NY, USA, 2006. ACM Press.

[Ohl] Ohloh. http://www.ohloh.net/. [Online; accessed 04-May-2007].

[Ope] Open Source Initiative. http://www.opensource.org/. [Online; accessed 27-April-2007].

[Per] Perforce. http://www.perforce.com/. [Online; accessed 14-May-2007].

[Quaa] QualiPSo. http://www.qualipso.org/. [Online; accessed 14-May-2007].

[QUAb] QUALOSS. http://www.qualoss.org/. [Online; accessed 14-May-2007].

[RAGBH05] G. Robles, JJ Amor, JM Gonzalez-Barahona, and I. Herraiz. Evolution and growth in large libre software projects. *Principles of Software Evolution, Eighth International Workshop on*, pages 165–174, 2005.

[RGBMA06] G. Robles, J.M. Gonzalez-Barahona, M. Michlmayr, and J.J. Amor. Mining large software compilations over time: another perspective of software evolution. *Proceedings of the 2006 international workshop on Mining software repositories*, pages 3–9, 2006.

[SFT07] P.V. Singh, M. Fan, and Y. Tan. An Empirical Investigation of Code Contribution, Communication Participation, and Release Strategy in Open Source Software Development: A Conditional Hazard Model Approach. http://opensource.mit.edu/papers/singh_fan_tan.pdf, 2007. [Online; accessed 27-April-2007].

[Soua] SourceForge. http://sourceforge.net/docs/about. [Online; accessed 04-May-2007].

[Soub] SourceKibitzer. http://www.sourcekibitzer.org/. [Online; accessed 04-May-2007].

[SQO] SQO-OSS. http://www.sqo-oss.eu/. [Online; accessed 14-May-2007].

[SVN] Subversion (SVN). http://subversion.tigris.org/. [Online; accessed 14-May-2007].

[Van00] E. VanDoren. Cyclomatic Complexity. http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html, 2000. [Online; accessed 27-April-2007].

[WF05] I.H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2 edition, 2005.

[Wik07] Wikipedia. Ohloh — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=Ohloh&oldid=125097327`, 2007. [Online; accessed 04-May-2007].

[Wu06] J. Wu. *Open Source Software Evolution and Its Dynamics*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 2006.

[XCM06] J. Xu, S. Christley, and G. Madey. Application of Social Network Analysis to the Study of Open Source Software. `http://www.nd.edu/~oss/Papers/Ch11-N52769.pdf`, 2006. [Online; accessed 04-May-2007].

Appendix A

Prediction Results for the Approach 1

Here we present the results of the prediction described in Section 4.4. Metrics delta values for source file modifications were aggregated over contributions prior to prediction of the contribution type.

Data Set 1	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
1 att.	44.04	43.12	44.04	44.95	45.87	43.12	43.12	41.28	44.95	45.87	44.04
2 att.-s	45.87	45.87	43.12	45.87	44.04	47.71	41.28	43.12	41.28	44.04	44.22
3 att.-s	41.28	44.04	42.2	45.87	43.12	44.95	43.12	44.04	44.04	46.79	43.95
4 att.-s	44.04	41.28	45.87	45.87	44.04	39.45	44.95	45.87	46.79	39.45	43.76
5 att.-s	44.04	41.28	39.45	42.2	44.04	47.71	44.95	44.95	42.2	44.95	43.58
6 att.-s	44.04	44.04	45.87	44.04	40.37	43.12	44.04	44.95	42.2	44.04	43.67
7 att.-s	44.95	43.12	44.04	42.2	41.28	44.95	45.87	46.79	44.04	44.95	44.22
8 att.-s	***48.62***	42.2	43.12	43.12	42.2	41.28	43.12	46.79	43.12	42.2	43.58
9 att.-s	47.71	42.2	39.45	44.95	42.2	41.28	45.87	44.04	44.95	42.2	43.49
10 att.-s	44.95	44.04	45.87	40.37	43.12	39.45	40.37	41.28	41.28	44.95	42.57

Table A.1: Percentage of correctly predicted contribution types for the data set number 1. Each row represents a different value for the "number of randomly selected attributes" parameter of the random forest classifier. For each value of the parameter 10 experiments were performed — each column is a result of one such experiment. The peak accuracy was 48.62%.

Data Set 2	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
1 att.	42.2	39.45	39.45	41.28	***44.95***	44.04	***44.95***	42.2	***44.95***	42.2	42.57
2 att.-s	41.28	39.45	34.86	37.61	43.12	***44.95***	41.28	44.04	38.53	40.37	40.55
3 att.-s	41.28	39.45	42.2	42.2	42.2	40.37	41.28	43.12	41.28	42.2	41.56
4 att.-s	44.04	44.04	44.04	40.37	***44.95***	38.53	43.12	40.37	39.45	43.12	42.2
5 att.-s	40.37	40.37	39.45	43.12	39.45	41.28	39.45	42.2	41.28	41.28	40.83
6 att.-s	42.2	***44.95***	38.53	39.45	38.53	43.12	37.61	36.7	41.28	40.37	40.27
7 att.-s	44.04	40.37	39.45	39.45	35.78	43.12	40.37	39.45	41.28	42.2	40.55
8 att.-s	36.7	43.12	40.37	41.28	38.53	43.12	42.2	37.61	36.7	42.2	40.18
9 att.-s	39.45	44.04	38.53	37.61	38.53	40.37	42.2	42.2	42.2	42.2	40.73
10 att.-s	35.78	38.53	40.37	41.28	43.12	39.45	39.45	40.37	40.37	39.45	39.82

Table A.2: Percentage of correctly predicted contribution types for the data set number 2. Each row represents a different value for the "number of randomly selected attributes" parameter of the random forest classifier. For each value of the parameter 10 experiments were performed — each column is a result of one such experiment. The peak accuracy was 44.95%.

Data Set 3	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
1 att.	49.62	53.44	48.85	52.67	48.09	49.62	48.09	48.85	48.85	48.09	49.62
2 att.-s	48.85	53.44	48.85	52.67	49.62	50.38	49.62	51.15	48.85	50.38	50.38
3 att.-s	51.91	52.67	47.33	49.62	49.62	50.38	48.09	49.62	48.85	50.38	49.85
4 att.-s	51.91	46.56	50.38	51.91	50.38	53.44	50.38	50.38	50.38	50.38	50.61
5 att.-s	48.09	48.85	50.38	***54.96***	48.85	48.09	51.91	47.33	47.33	48.85	49.47
6 att.-s	50.38	49.62	48.09	48.85	51.15	48.85	50.38	49.62	50.38	47.33	49.47
7 att.-s	48.85	47.33	53.44	48.09	51.91	50.38	51.91	51.15	49.62	***54.96***	50.76
8 att.-s	48.85	51.15	47.33	49.62	49.62	49.62	50.38	47.33	47.33	51.15	49.24
9 att.-s	49.62	50.38	50.38	48.09	48.85	48.09	50.38	48.09	49.62	51.15	49.47
10 att.-s	51.15	48.85	50.38	47.33	48.85	51.91	49.62	51.91	51.15	45.8	49.69

Table A.3: Percentage of correctly predicted contribution types for the data set number 3. Each row represents a different value for the "number of randomly selected attributes" parameter of the random forest classifier. For each value of the parameter 10 experiments were performed — each column is a result of one such experiment. The peak accuracy was 54.96%.

Data Set 4	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
1 att.	49.62	51.15	48.85	47.33	46.56	45.8	47.33	47.33	47.33	49.62	48.09
2 att.-s	46.56	45.8	47.33	47.33	48.09	47.33	48.85	48.09	48.09	48.85	47.63
3 att.-s	47.33	48.85	49.62	48.09	51.91	45.8	49.62	47.33	48.09	46.56	48.32
4 att.-s	51.91	50.38	50.38	51.15	49.62	50.38	46.56	48.09	49.62	49.62	49.77
5 att.-s	48.85	48.85	48.85	50.38	47.33	49.62	48.09	46.56	***52.67***	48.09	48.93
6 att.-s	48.85	48.09	51.15	49.62	44.27	45.8	45.8	47.33	48.85	***52.67***	48.24
7 att.-s	45.8	48.09	47.33	48.85	48.85	49.62	51.91	50.38	49.62	49.62	49.01
8 att.-s	49.62	48.85	48.09	47.33	46.56	49.62	47.33	47.33	47.33	48.85	48.09
9 att.-s	47.33	45.04	48.85	48.09	***52.67***	48.85	49.62	50.38	49.62	48.85	48.93
10 att.-s	49.62	50.38	45.8	45.04	50.38	48.09	50.38	49.62	46.56	48.09	48.4

Table A.4: Percentage of correctly predicted contribution types for the data set number 4. Each row represents a different value for the "number of randomly selected attributes" parameter of the random forest classifier. For each value of the parameter 10 experiments were performed — each column is a result of one such experiment. The peak accuracy was 52.67%.

Data Set 5	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
1 att.	41.51	41.51	44.34	42.45	44.34	44.34	44.34	41.51	44.34	41.51	43.02
2 att.-s	45.28	44.34	45.28	45.28	41.51	39.62	42.45	44.34	41.51	41.51	43.11
3 att.-s	43.4	44.34	42.45	44.34	40.57	44.34	41.51	46.23	42.45	44.34	43.4
4 att.-s	45.28	43.4	45.28	47.17	43.4	42.45	43.4	46.23	40.57	41.51	43.87
5 att.-s	42.45	45.28	39.62	46.23	45.28	47.17	42.45	45.28	44.34	45.28	44.34
6 att.-s	44.34	44.34	44.34	45.28	42.45	46.23	43.4	45.28	43.4	37.74	43.68
7 att.-s	43.4	38.68	44.34	46.23	41.51	40.57	43.4	43.4	45.28	45.28	43.21
8 att.-s	43.4	41.51	46.23	43.4	44.34	43.4	42.45	***48.11***	45.28	***48.11***	44.62
9 att.-s	42.45	45.28	40.57	41.51	41.51	44.34	42.45	46.23	47.17	44.34	43.58
10 att.-s	44.34	41.51	45.28	46.23	44.34	43.4	***48.11***	44.34	42.45	39.62	43.96

Table A.5: Percentage of correctly predicted contribution types for the data set number 5. Each row represents a different value for the "number of randomly selected attributes" parameter of the random forest classifier. For each value of the parameter 10 experiments were performed — each column is a result of one such experiment. The peak accuracy was 48.11%.

Data Set 6	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
1 att.	40.57	39.62	44.34	42.45	43.4	46.23	45.28	45.28	39.62	43.4	43.02
2 att.-s	43.4	40.57	40.57	39.62	37.74	37.74	***47.17***	44.34	43.4	40.57	41.51
3 att.-s	40.57	45.28	42.45	41.51	43.4	35.85	36.79	44.34	43.4	45.28	41.89
4 att.-s	41.51	38.68	35.85	41.51	43.4	39.62	39.62	42.45	36.79	40.57	40
5 att.-s	***47.17***	43.4	39.62	40.57	42.45	41.51	38.68	42.45	40.57	38.68	41.51
6 att.-s	40.57	38.68	41.51	38.68	42.45	35.85	44.34	42.45	39.62	41.51	40.57
7 att.-s	43.4	40.57	44.34	37.74	38.68	34.91	40.57	42.45	44.34	41.51	40.85
8 att.-s	39.62	43.4	43.4	43.4	41.51	43.4	40.57	43.4	36.79	39.62	41.51
9 att.-s	41.51	41.51	41.51	38.68	43.4	45.28	43.4	40.57	43.4	41.51	42.08
10 att.-s	39.62	40.57	38.68	34.91	36.79	45.28	41.51	40.57	40.57	39.62	39.81

Table A.6: Percentage of correctly predicted contribution types for the data set number 6. Each row represents a different value for the "number of randomly selected attributes" parameter of the random forest classifier. For each value of the parameter 10 experiments were performed — each column is a result of one such experiment. The peak accuracy was 47.17%.

Data Set 7	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
1 att.	50	50	48.44	46.09	49.22	48.44	49.22	51.56	52.34	47.66	49.3
2 att.-s	53.13	52.34	53.13	50.78	48.44	47.66	48.44	51.56	48.44	46.88	50.08
3 att.-s	***55.47***	49.22	50	50.78	45.31	49.22	52.34	48.44	48.44	46.88	49.61
4 att.-s	53.13	47.66	50	52.34	47.66	44.53	53.91	50	47.66	51.56	49.84
5 att.-s	50.78	50	45.31	53.13	46.88	52.34	52.34	46.88	50.78	52.34	50.08
6 att.-s	50.78	49.22	50.78	53.91	52.34	48.44	42.97	52.34	48.44	46.09	49.53
7 att.-s	48.44	50	53.13	50.78	51.56	53.91	53.13	49.22	50	53.13	51.33
8 att.-s	51.56	48.44	50	53.91	52.34	***55.47***	49.22	53.13	***55.47***	53.13	52.27
9 att.-s	53.13	50	50	52.34	49.22	52.34	52.34	48.44	54.69	46.88	50.94
10 att.-s	53.13	48.44	50	48.44	51.56	50.78	48.44	50.78	46.88	52.34	50.08

Table A.7: Percentage of correctly predicted contribution types for the data set number 7. Each row represents a different value for the "number of randomly selected attributes" parameter of the random forest classifier. For each value of the parameter 10 experiments were performed — each column is a result of one such experiment. The peak accuracy was 55.47%.

Data Set 8	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
1 att.	48.44	46.09	47.66	46.09	45.31	46.09	46.88	46.88	47.66	***52.34***	47.34
2 att.-s	49.22	43.75	45.31	46.09	44.53	46.09	46.88	49.22	46.88	49.22	46.72
3 att.-s	46.88	50.78	47.66	45.31	46.88	47.66	50.78	46.88	43.75	44.53	47.11
4 att.-s	46.88	47.66	49.22	46.88	49.22	43.75	44.53	48.44	48.44	47.66	47.27
5 att.-s	47.66	44.53	46.09	47.66	46.09	46.88	45.31	46.88	42.97	47.66	46.17
6 att.-s	45.31	47.66	46.09	42.19	47.66	46.09	45.31	50	45.31	48.44	46.41
7 att.-s	47.66	46.88	45.31	49.22	45.31	45.31	48.44	46.09	42.97	50	46.72
8 att.-s	46.88	50.78	48.44	42.19	43.75	45.31	44.53	***52.34***	46.88	46.09	46.72
9 att.-s	47.66	46.09	38.28	46.88	46.09	46.88	42.97	44.53	45.31	48.44	45.31
10 att.-s	46.88	46.09	49.22	46.88	44.53	42.97	46.09	47.66	42.19	44.53	45.7

Table A.8: Percentage of correctly predicted contribution types for the data set number 8. Each row represents a different value for the “number of randomly selected attributes” parameter of the random forest classifier. For each value of the parameter 10 experiments were performed — each column is a result of one such experiment. The peak accuracy was 52.34%.

Data Set 9	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
1 att.	44.76	42.86	42.86	46.67	46.67	45.71	45.71	41.9	41.9	40	43.9
2 att.-s	48.57	46.67	47.62	42.86	45.71	44.76	49.52	48.57	43.81	41.9	46
3 att.-s	41.9	40.95	41.9	44.76	46.67	43.81	46.67	43.81	47.62	44.76	44.29
4 att.-s	44.76	43.81	44.76	44.76	39.05	42.86	43.81	40.95	43.81	44.76	43.33
5 att.-s	44.76	43.81	40.95	42.86	47.62	44.76	42.86	45.71	43.81	46.67	44.38
6 att.-s	43.81	43.81	46.67	40	40.95	42.86	42.86	45.71	40	44.76	43.14
7 att.-s	42.86	40	41.9	46.67	40	44.76	44.76	40.95	37.14	44.76	42.38
8 att.-s	47.62	38.1	44.76	42.86	42.86	41.9	43.81	44.76	47.62	47.62	44.19
9 att.-s	40.95	42.86	45.71	43.81	47.62	37.14	43.81	44.76	44.76	44.76	43.62
10 att.-s	40	41.9	***50.48***	43.81	38.1	47.62	40	43.81	40	42.86	42.86

Table A.9: Percentage of correctly predicted contribution types for the data set number 9. Each row represents a different value for the "number of randomly selected attributes" parameter of the random forest classifier. For each value of the parameter 10 experiments were performed — each column is a result of one such experiment. The peak accuracy was 50.48%.

Data Set 10	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
1 att.	41.9	41.9	41.9	39.05	42.86	41.9	40.95	41.9	40	39.05	41.14
2 att.-s	40	40.95	42.86	44.76	40.95	40.95	43.81	42.86	43.81	43.81	42.48
3 att.-s	***45.71***	43.81	43.81	42.86	40	41.9	40	40.95	43.81	38.1	42.1
4 att.-s	34.29	40.95	41.9	***45.71***	40	38.1	40	37.14	41.9	43.81	40.38
5 att.-s	42.86	43.81	39.05	40	34.29	41.9	38.1	39.05	39.05	39.05	39.71
6 att.-s	41.9	39.05	37.14	40.95	37.14	38.1	40	32.38	39.05	38.1	38.38
7 att.-s	40	***45.71***	37.14	35.24	38.1	41.9	36.19	36.19	36.19	37.14	38.38
8 att.-s	36.19	40.95	41.9	38.1	40.95	40.95	43.81	40	43.81	32.38	39.9
9 att.-s	37.14	40	40.95	39.05	41.9	37.14	35.24	38.1	40	37.14	38.67
10 att.-s	39.05	40	37.14	40	36.19	40	41.9	37.14	38.1	42.86	39.24

Table A.10: Percentage of correctly predicted contribution types for the data set number 10. Each row represents a different value for the "number of randomly selected attributes" parameter of the random forest classifier. For each value of the parameter 10 experiments were performed — each column is a result of one such experiment. The peak accuracy was 45.71%.

Data Set 11	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
1 att.	48.44	48.44	49.22	50	48.44	53.13	51.56	50	51.56	49.22	50
2 att.-s	50.78	48.44	47.66	50.78	46.88	50.78	53.91	50.78	***55.47***	50	50.55
3 att.-s	50	53.13	50.78	46.88	46.09	51.56	50.78	49.22	52.34	52.34	50.31
4 att.-s	49.22	50	50.78	52.34	44.53	50	53.13	50.78	50.78	50	50.16
5 att.-s	52.34	50	50	46.88	50	50.78	50	53.91	50.78	50	50.47
6 att.-s	50.78	50.78	50.78	50.78	48.44	49.22	50.78	49.22	50.78	50.78	50.23
7 att.-s	52.34	48.44	48.44	50	50.78	52.34	48.44	50	49.22	54.69	50.47
8 att.-s	51.56	49.22	52.34	51.56	51.56	50	53.13	48.44	47.66	50.78	50.63
9 att.-s	50.78	51.56	50.78	51.56	50.78	53.91	49.22	51.56	50.78	48.44	50.94
10 att.-s	50.78	53.13	46.88	53.13	50.78	49.22	49.22	50	53.13	50.78	50.7

Table A.11: Percentage of correctly predicted contribution types for the data set number 11. Each row represents a different value for the "number of randomly selected attributes" parameter of the random forest classifier. For each value of the parameter 10 experiments were performed — each column is a result of one such experiment. The peak accuracy was 55.47%.

Data Set 12	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Average**
1 att.	45.31	44.53	48.44	49.22	46.88	45.31	48.44	49.22	50	46.09	47.34
2 att.-s	50	46.88	46.09	47.66	49.22	46.09	***50.78***	44.53	46.09	43.75	47.11
3 att.-s	42.19	47.66	44.53	48.44	46.88	45.31	46.09	50	47.66	46.09	46.48
4 att.-s	40.63	48.44	47.66	47.66	44.53	46.09	50	46.09	44.53	45.31	46.09
5 att.-s	50	42.19	49.22	46.09	45.31	46.09	41.41	46.88	46.88	46.09	46.02
6 att.-s	44.53	48.44	40.63	45.31	44.53	40.63	46.88	45.31	44.53	48.44	44.92
7 att.-s	48.44	46.88	46.88	42.97	43.75	47.66	43.75	48.44	48.44	***50.78***	46.8
8 att.-s	44.53	46.88	43.75	45.31	48.44	***50.78***	48.44	44.53	50	42.97	46.56
9 att.-s	46.09	49.22	47.66	46.88	47.66	***50.78***	42.19	46.88	49.22	45.31	47.19
10 att.-s	45.31	46.09	48.44	46.88	46.09	43.75	46.09	49.22	42.97	42.19	45.7

Table A.12: Percentage of correctly predicted contribution types for the data set number 12. Each row represents a different value for the "number of randomly selected attributes" parameter of the random forest classifier. For each value of the parameter 10 experiments were performed — each column is a result of one such experiment. The peak accuracy was 50.78%.

Attributes	**1 vs 2**	**3 vs 4**	**5 vs 6**	**7 vs 8**	**9 vs 10**	**11 vs 12**
1	1.47	1.53	0	1.95	2.76	2.66
2	3.67	2.75	1.6	3.36	3.52	3.44
3	2.39	1.53	1.51	2.5	2.19	3.83
4	1.56	0.84	3.87	2.58	2.95	4.06
5	2.75	0.53	2.83	3.91	4.67	4.45
6	3.4	1.22	3.11	3.13	4.76	5.31
7	3.67	1.76	2.36	4.61	4	3.67
8	3.39	1.15	3.11	5.55	4.29	4.06
9	2.75	0.53	1.51	5.63	4.95	3.75
10	2.75	1.3	4.15	4.38	3.62	5
Average	**2.78**	**1.31**	**2.41**	**3.76**	**3.77**	**4.02**
Median	**2.75**	**1.26**	**2.59**	**3.63**	**3.81**	**3.95**

Table A.13: Comparison of the prediction accuracy between the data sets containing both source code and contribution metrics (data sets $1, 3, 5, 7, 9, 11$) and the data sets containing only source code metrics (data sets $2, 4, 6, 8, 10, 12$). Each row represents a different value for the "number of randomly selected attributes" parameter of the random forest classifier. Each column — a difference between two data sets.

Attributes	**1 vs 3**	**2 vs 4**	**5 vs 7**	**6 vs 8**	**9 vs 11**	**10 vs 12**
1	5.58	5.52	6.28	4.32	6.1	6.2
2	6.16	7.08	6.96	5.21	4.55	4.63
3	5.9	6.76	6.21	5.22	6.03	4.39
4	6.85	7.57	5.98	7.27	6.82	5.71
5	5.89	8.11	5.74	4.66	6.09	6.3
6	5.79	7.97	5.85	5.84	7.09	6.54
7	6.54	8.46	8.12	5.87	8.09	8.42
8	5.66	7.91	7.64	5.21	6.43	6.66
9	5.98	8.2	7.35	3.24	7.32	8.52
10	7.13	8.58	6.12	5.89	7.85	6.47
Average	**6.15**	**7.62**	**6.63**	**5.27**	**6.64**	**6.38**
Median	**5.94**	**7.94**	**6.25**	**5.22**	**6.63**	**6.38**

Table A.14: Comparison of the prediction accuracy between the data sets that take into account only modifications with "delta values of LOC, NCSS or NPC > 9" (Condition 4.1, data sets $1, 2, 5, 6, 9, 10$) and the data sets containing all modifications (data sets $3, 4, 7, 8, 11, 12$). Each row represents a different value for the "number of randomly selected attributes" parameter of the random forest classifier. Each column — a difference between two data sets.

Appendix B

Prediction Results for the Approach 2

This appendix contains the results of prediction described in Section 4.5. Modifications were classified separately. The most frequent type of the modifications belonging to the same contribution was taken as the contribution type.

Attributes \ Data Sets	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
1	41.28	21.1	*43.51*	-	41.51	19.81	***44.53***	-	41.9	18.1	*43.75*	-
2	37.61	17.43	41.22	-	42.45	19.81	42.19	-	38.1	18.1	41.41	-
3	40.37	18.35	38.17	-	41.51	16.98	40.63	-	38.1	19.05	40.63	-
4	39.45	20.18	36.64	-	*44.34*	16.98	39.06	-	40.95	20	39.84	-
5	38.53	21.1	36.64	-	42.45	21.7	40.63	-	40	19.05	42.19	-

Table B.1: Percentage of correctly predicted contribution types for all data sets. Each row represents a different value for the "number of randomly selected attributes" parameter of the random forest classifier. Each column — a different data set. The peak accuracy was 44.53%. The random forest classifier wasn't able to built a type prediction model for the data sets $4, 8, 12$

Attributes	**1 vs 2**	**5 vs 6**	**9 vs 10**
1	20.18	21.7	23.8
2	20.18	22.64	20
3	22.02	24.53	19.05
4	19.27	27.36	20.95
5	17.43	20.75	20.95
Average	**19.82**	**23.4**	**20.95**
Median	**20.18**	**22.64**	**20.95**

Table B.2: Comparison of the prediction accuracy between the data sets containing both source code and contribution metrics (data sets $1, 5, 9$) and the data sets containing only source code metrics (data sets $2, 6, 10$). Each row represents a different value for the "number of randomly selected attributes" parameter of the random forest classifier. Each column — a difference between two data sets. Comparisons that include data sets $4, 8, 12$ are missing because random forest classifier wasn't able to built a type prediction for them.

Attributes	**1 vs 3**	**5 vs 7**	**9 vs 11**
1	-2.23	-3.02	-1.85
2	-3.61	0.26	-3.31
3	2.2	0.88	-2.53
4	2.81	5.28	1.11
5	1.89	1.82	-2.19
Average	**0.21**	**1.04**	**-1.75**
Median	**1.89**	**0.88**	**-2.19**

Table B.3: Comparison of the prediction accuracy between the data sets that take into account only modifications with "delta values of LOC, NCSS or NPC > 9" (Condition 4.1, data sets $1, 5, 9$) and the data sets containing all modifications (data sets $3, 7, 11$). Each row represents a different value for the "number of randomly selected attributes" parameter of the random forest classifier. Each column — a difference between two data sets. Comparisons that include data sets $4, 8, 12$ are missing because random forest classifier wasn't able to built a type prediction for them.