e-Informatica Software Engineering Journal, Volume 12, Issue 1, 2018, pages: 199–216, DOI 10.5277/e-Inf180108

Applying Machine Learning to Software Fault Prediction

Bartłomiej Wójcicki*, Robert Dąbrowski*
*Institute of Informatics, University of Warsaw
bartwojcicki@gmail.com, r.dabrowski@mimuw.edu.pl

Abstract

Introduction: Software engineering continuously suffers from inadequate software testing. The automated prediction of possibly faulty fragments of source code allows developers to focus development efforts on fault-prone fragments first. Fault prediction has been a topic of many studies concentrating on C/C++ and Java programs, with little focus on such programming languages as Python.

Objectives: In this study the authors want to verify whether the type of approach used in former fault prediction studies can be applied to Python. More precisely, the primary objective is conducting preliminary research using simple methods that would support (or contradict) the expectation that predicting faults in Python programs is also feasible. The secondary objective is establishing grounds for more thorough future research and publications, provided promising results are obtained during the preliminary research.

Methods: It has been demonstrated [1] that using machine learning techniques, it is possible to predict faults for C/C++ and Java projects with recall 0.71 and false positive rate 0.25. A similar approach was applied in order to find out if promising results can be obtained for Python projects. The working hypothesis is that choosing Python as a programming language does not significantly alter those results. A preliminary study is conducted and a basic machine learning technique is applied to a few sample Python projects. If these efforts succeed, it will indicate that the selected approach is worth pursuing as it is possible to obtain for Python results similar to the ones obtained for C/C++ and Java. However, if these efforts fail, it will indicate that the selected approach was not appropriate for the selected group of Python projects.

Results: The research demonstrates experimental evidence that fault-prediction methods similar to those developed for C/C++ and Java programs can be successfully applied to Python programs, achieving recall up to 0.64 with false positive rate 0.23 (mean recall 0.53 with false positive rate 0.24). This indicates that more thorough research in this area is worth conducting.

Conclusion: Having obtained promising results using this simple approach, the authors conclude that the research on predicting faults in Python programs using machine learning techniques is worth conducting, natural ways to enhance the future research being: using more sophisticated machine learning techniques, using additional Python-specific features and extended data sets.

Keywords: classifier, fault prediction, machine learning, metric, Naïve Bayes, Python, quality, software intelligence

1. Introduction

Software engineering is concerned with the development and maintenance of software systems. Properly engineered systems are reliable and they satisfy user requirements while at the same time their development and maintenance is affordable.

In the past half-century computer scientists and software engineers have come up with numerous ideas for how to improve the discipline of software engineering. Structural programming [2]

restricted the imperative control flow to hierarchical structures instead of *ad-hoc* jumps. Computer programs written in this style were more readable, easier to understand and reason about. Another improvement was the introduction of an object-oriented paradigm [3] as a formal programming concept.

In the early days software engineers perceived significant similarities between software and civil engineering processes. The waterfall model [4], which resembles engineering practices, was widely adopted as such regardless of its original description actually suggesting a more agile approach.

It has soon turned out that building software differs from building skyscrapers and bridges, and the idea of extreme programming emerged [5], its key points being: keeping the code simple, reviewing it frequently and early and frequent testing. Among numerous techniques, a test-driven development was promoted which eventually resulted in the increased quality of produced software and the stability of the development process [6]. Contemporary development teams started to lean towards short iterations (sprints) rather than fragile upfront designs, and short feedback loops, thus allowing customers' opinions to provide timely influence on software development. This meant creating even more complex software systems.

The growing complexity of software resulted in the need to describe it at different levels of abstraction, and, in addition to this, the notion of software architecture has developed. The emergence of patterns and frameworks had a similar influence on the architecture as design patterns and idioms had on programming. Software started to be developed by assembling reusable software components which interact using well-defined interfaces, while component-oriented frameworks and models provided tools and languages making them suitable for formal architecture design.

However, a discrepancy between the architecture level of abstraction and the programming level of abstraction prevailed. While the programming phase remained focused on generating a code within a preselected (typically object-oriented) programming language, the architecture phase took place in the disconnected component world. The discrepancies typically deepened as the software kept gaining features without being properly refactored, development teams kept changing over time working under time pressure with incomplete documentation and requirements that were subject to frequent changes. Multiple development technologies, programming languages and coding standards made this situation even more severe. The unification of modelling languages failed to become a silver bullet.

The discrepancy accelerated research on software architecture and the automation of software engineering. This includes the vision for the automated engineering of software based on architecture warehouse and software intelligence [7] ideas. The architecture warehouse denotes a repository of the whole software system and software process artefacts. Such a repository uniformly captures and regards as architectural all information which was previously stored separately in design documents, version-control systems or simply in the minds of software developers. Software intelligence denotes a set of tools for the automated analysis, optimization and visualization of the warehouse content [8, 9].

An example of this approach is combining information on source code artefacts, such as functions, with the information on software process artefacts, such as version control comments indicating the developers' intents behind changes in given functions. Such an integration of source code artefacts and software process artefacts allows to aim for more sophisticated automated learning and reasoning in the area of software engineering, for example obtaining an ability to automatically predict where faults are likely to occur in the source code during the software process.

The automated prediction of possibly faulty fragments of the source code, which allows developers to focus development efforts on the bug prone modules first, is the topic of this research. This is an appealing idea since, according to a U.S. National Institute of Standards and Technology's study [10], inadequate software testing infrastructure costs the U.S. economy an esti-

mated $60 billion annually. One of the factors that could yield savings is identifying faults at earlier development stages.

For this reason, fault prediction was the subject of many previous studies. As yet, software researchers have concluded that defect predictors based on machine learning methods are practical [11] and useful [12]. Such studies were usually focused on C/C++ and Java projects [13] omitting other programming languages, such as Python.

This study demonstrates experimentally that techniques used in the former fault prediction studies can be successfully applied to the software developed in Python. The paper is organized as follows: in section 2 the related works are recalled; in section 3 the theoretic foundations and implementation details of the approach being subject of this study are highlighted; the main results are presented in section 4, with conclusions to follow in section 5. The implementation of the method used in this study for predictor evaluation is outlined in the Appendix, it can be used to reproduce the results of the experiments. The last section contains bibliography.

2. Related work

Software engineering is a sub-field of applied computer science that covers the principles and practice of architecting, developing and maintaining software. Fault prediction is a software engineering problem. Artificial intelligence studies software systems that are capable of intelligent reasoning. Machine learning is a part of artificial intelligence dedicated to one of its central problems - automated learning. In this research machine learning methods are applied to a fault prediction problem.

For a given Python software project, the architectural information warehoused in the project repository is used to build tools capable of automated reasoning about possible faults in a given source code. More specifically: (1) a tool able to predict which parts of the source code are fault-prone is developed; and (2) its operation is demonstrated on five open-source projects.

Prior works in this field [1] demonstrated that it is possible to predict faults for C/C++ and Java projects with a recall rate of 71% and a false positive rate of 25%. The tool demonstrated in this paper demonstrates that it is possible to predict faults in Python achieving recall rates up to 64% with a false positive rate of 23% for some projects; for all tested projects the achieved mean recall was 53% with a false positive rate of 24%.

Fault prediction spans multiple aspects of software engineering. On the one hand, it is a software verification problem. In 1989 Boehm [14] defined the goal of verification as an answer to the question *Are we building the product right*? Contrary to formal verification methods (e.g. model checking), fault predictors cannot be used to prove that a program is correct; they can, however, indicate the parts of the software that are suspected of containing defects.

On the other hand, fault prediction is related to software quality management. In 2003 Khoshgoftaar et al. [15] observed that it can be particularly helpful in prioritizing quality assurance efforts. They studied high-assurance and mission-critical software systems heavily dependent on the reliability of software applications. They evaluated the predictive performance of six commonly used fault prediction techniques. Their case studies consisted of software metrics collected over large telecommunication system releases. During their tests it was observed that prediction models based on software metrics could actually predict the number of faults in software modules; additionally, they compared the performance of the assessed prediction models.

Static code attributes have been used for the identification of potentially problematic parts of a source code for a long time. In 1990 Porter et al. [16] addressed the issue of the early identification of high-risk components in the software life cycle. They proposed an approach that derived the models of problematic components based on their measurable attributes and the attributes of their development processes. The models allowed to forecast which components were likely to share the same high-risk properties, such as like being error-prone or having a high development cost.

Table 1. Prior results of fault predictors using NASA data sets [17]

Data set	Language	Recall	False positive rate
PC1	C	0.24	0.25
JM1	C	0.25	0.18
CM1	C	0.35	0.10
KC2	C++	0.45	0.15
KC1	C++	0.50	0.15
	In total:	0.36	0.17

Table 2. Prior results of fault predictors using NASA data sets [1] (logarithmic filter applied)

Data set	Language	Recall	False positive rate
PC1	C	0.48	0.17
MW1	C	0.52	0.15
KC3	Java	0.69	0.28
CM1	C	0.71	0.27
PC2	C	0.72	0.14
KC4	Java	0.79	0.32
PC3	C	0.80	0.35
PC4	C	0.98	0.29
	in total:	0.71	0.25

In 2002, the NASA Metrics Data Program Data sets were published [18]. Each data set contained complexity metrics defined by Halstead and McCabe, the lines of code metrics and defect rates for the modules of a different subsystem of NASA projects. These data sets included projects in C, C++ and Java. Multiple studies that followed used these data sets and significant progress in this area was made.

In 2003 Menzies et al. examined decision trees and rule-based learners [19–21]. They researched a situation when it is impractical to rigorously assess all parts of complex systems and test engineers must use some kind of defect detectors to focus their limited resources. They defined the properties of good defect detectors and assessed different methods of their generation. They based their assessments on static code measures and found that (1) such defect detectors yield results that are stable across many applications, and (2) the detectors are inexpensive to use and can be tuned to the specifics of current business situations. They considered practical situations in which software costs are assessed and additionally assumed that better assessment allowed to earn exponentially more money. They pointed out that given finite budgets, assessment resources are typically skewed towards areas that are believed to be mission critical; hence, the portions of the system that may actually contain defects may be missed. They indicated that by using proper metrics and machine learning algorithms, quality indicators can be found early in the software development process.

In 2004 Menzies et al. [17] assessed other predictors of software defects and demonstrated that these predictors are outperformed by Naïve Bayes classifiers, reporting a mean recall of 0.36 with a false positive rate of 0.17 (see Table 1). More precisely they demonstrated that when learning defect detectors from static code measures, Naïve Bayes learners are better than entropy-based decision-tree learners, and that accuracy is not a useful way to assess these detectors. They also argued that such learners need no more than 200–300 examples to learn adequate detectors, especially when the data has been heavily stratified; i.e. divided into sub-sub-sub systems.

In 2007 Menzies et al. [1] proposed applying a logarithmic filter to features. The value of using static code attributes to learn defect predictors was widely debated. Prior work explored issues such as the merits of McCabes versus Halstead versus the lines of code counts for generating defect predictors. They showed that such debates are irrelevant since *how* the attributes are used to build predictors is much more important than *which* particular attributes are actually used. They demonstrated that adding a logarithmic filter resulted in improving recall to 0.71, keeping a false positive rate reasonably low at 0.25 (see Table 2).

In 2012 Hall et al. [13] identified and analysed 208 defect prediction studies published from January 2000 to December 2010. By a systematic review, they drew the following conclusions: (1) there are multiple types of features that can be used for defect prediction, including static code metrics, change metrics and previous fault

metrics; (2) there are no clear best bug-proneness indicators; (3) models reporting a categorical predicted variable (e.g. fault prone or not fault prone) are more prevalent than models reporting a continuous predicted variable; (4) various statistical and machine learning methods can be employed to build fault predictors; (5) industrial data can be reliably used, especially data publicly available in the NASA Metrics Data Program data sets; (6) fault predictors are usually developed for C/C++ and Java projects.

In 2016 Lanza et al. [22] criticized the evaluation methods of defect prediction approaches; they claimed that in order to achieve substantial progress in the field of defect prediction (also other types of predictions), researchers should put predictors out into the real world and have them assessed by developers who work on a live code base, as defect prediction only makes sense if it is used *in vivo*.

The main purpose of this research is to extend the range of analysed programming languages to include Python. In the remaining part of the paper it is experimentally demonstrated that it is possible to predict defects for Python projects using static code features with an approach similar to (though not directly replicating) the one taken by Menzies et al. [1] for C/C++ and Java.

3. Problem definition

For the remaining part of this paper let *fault* denote any flaw in the source code that can cause the software to fail to perform its required function. Let *repository* denote the storage location from which the source code may be retrieved with version control capabilities that allow to analyse *revisions* denoting the precisely specified incarnations of the source code at a given point in time. For a given revision K let $K{\sim}1$ denote its parent revision, $K{\sim}2$ denote its grandparent revision, etc. Let *software metric* denote the measure of a degree to which a unit of software possesses some property. Static metrics can be collected for software without executing it, in contrast to the dynamic ones. Let *supervised learning* denote a type of machine learning task where an algorithm learns from a set of training examples with assigned expected outputs [23].

The authors follow with the definition central to the problem researched in this paper.

Definition 3.1. Let a *classification problem* denote an instance of a machine learning problem, where the expected output is *categorical*, that is where: a *classifier* is the algorithm that implements the classification; a *training set* is a set of instances supplied for the classifier to learn from; a *testing set* is a set of instances used for assessing classifier performance; an *instance* is a single object from which the classifier will learn or on which it will be used, usually represented by a *feature vector* with *features* being individual measurable properties of the phenomenon being observed, and a *class* being the predicted variable, that is the output of the classifier for the given instance.

In short: in classification problems classifiers assign classes to instances based on their features.

Fault prediction is a process of predicting where faults are likely to occur in the source code. In this case machine learning algorithms operate on instances being units of code (e.g. functions, classes, packages). Instances are represented by their features being the properties of the source code that indicate the source code unit's fault-proneness (e.g. number of lines of code, number of previous bugs, number of comments). The features are sometimes additionally preprocessed; an example of a feature preprocessor, called a *logarithmic filter*, substitutes the values of features with their logarithms. For the instances in the training set the predicted variable must be provided; e.g. the instances can be reviewed by experts and marked as *fault-prone* or *not fault-prone*. After the fault predictor learns from the training set of code units, it can be used to predict the fault-proneness of the new units of the code. The process is conceptually depicted in Figure 1.

A *confusion matrix* is a matrix containing the counts of instances grouped by the actual and predicted class. For the classification problem it is a 2×2 matrix (as depicted in Table 3). The confusion matrix and derived metrics can be used to evaluate classifier performance, where the typical indicators are as follows:

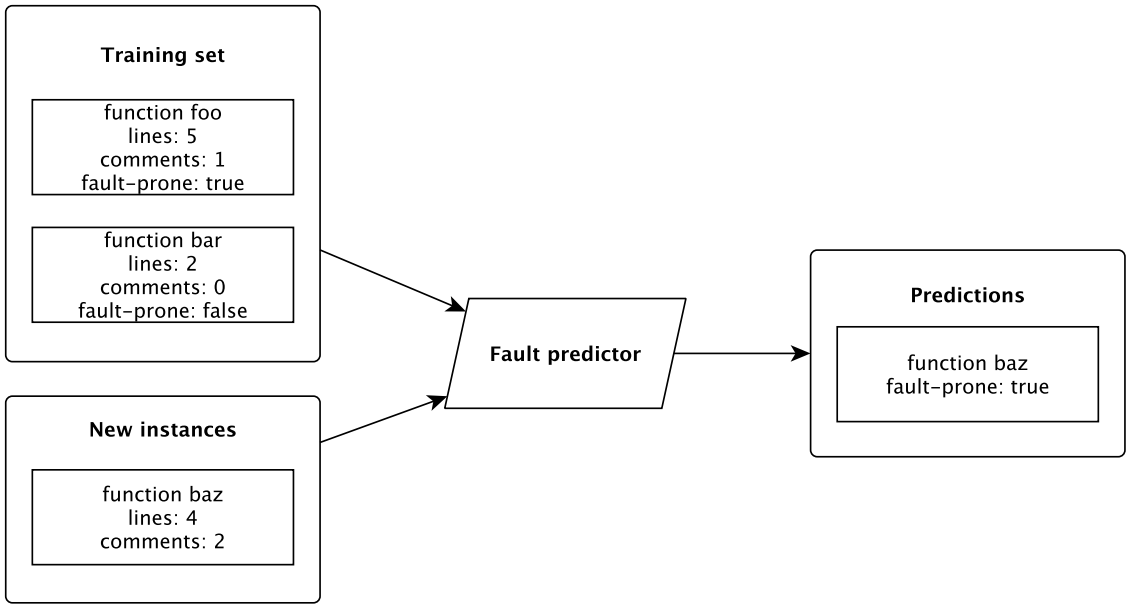

Figure 1. Fault prediction problem (sample)

Table 3. Confusion matrix for classification problems

Actual/predicted	Negative	Positive
negative	true negative (tn)	false positive (fp)
positive	false negative (fn)	true postive (tp)

Definition 3.2. Let *recall* denote a fraction of actual positive class instances that are correctly assigned to positive class:

$$\frac{tp}{tp + fn}$$

Let *precision* denote a fraction of predicted positive class instances that actually are in the positive class:

$$\frac{tp}{tp + fp}$$

Let a *false positive rate* denote a fraction of actual negative class instances that are incorrectly assigned to the positive class:

$$\frac{fp}{fp + tn}$$

Let *accuracy* denote a fraction of instances assigned to correct classes:

$$\frac{tp + tn}{tp + fp + tn + fn}$$

The remaining part of this section contains two subsections. In 3.1 the classification problem analysed in this study is stated in terms typical to machine learning, that is instances: what kinds of objects are classified; classes: into what classes are they are divided; features: what features are used to describe them; classifier: which learning method is used. Section 3.2 focuses on the practical aspects of fault prediction and describes the operational phases of the implementation: identification of instances, feature extraction, generation of a training set, training and predicting.

3.1. Classification problem definition

3.1.1. Instances

The defect predictor described in this study operates at the function level, which is a *de facto* standard in this field [13]. As *the first rule of functions is that they should be small* [24], it

was assumed that it should be relatively easy for developers to find and fix a bug in a function reported as fault-prone by a function-level fault predictor. Hence, in this research functions being instances of problem definition were selected.

3.1.2. Classes

For simplicity of reasoning, in this research the severity of bugs is not predicted. Hence, problem definition instances are labelled as either *fault-prone* or *not fault-prone*.

3.1.3. Features

To establish defect predictors the code complexity measures as defined by McCabe [25] and Halstead [26] were used.

The following Halstead's complexity measures were applied in this study as code metrics for estimating programming effort. They estimate complexity using operator and operand counts and are widely used in fault prediction studies [1].

Definition 3.3. Let n_1 denote the count of distinct operators, n_2 denote the count of distinct operands, N_1 denote the total count of operators, N_2 denote the total count of operands. Then Halstead metrics are defined as follows: program vocabulary $n = n_1 + n_2$; program length $N = N_1 + N_2$; calculated program length $\widehat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$; volume $V = N \times \log_2 n$; difficulty $D = n_1/2 \times N_2/n_2$; effort $E = D \times V$; time required to program $T = E/18$ seconds; number of delivered bugs $B = V/3000$.

In this research all the metrics defined above, including the counters of operators and operands, are used as features; in particular preliminary research indicated that limiting the set of features leads to results with lower recall.

In the study also The McCabe's cyclomatic complexity measure, being quantitative measure of the number of linearly independent paths through a program's source code, was applied. In terms of the software's architecture graph, cyclomatic complexity is defined as follows.

Definition 3.4. Let G be the flow graph being a subgraph of the software architecture graph, where e denotes the number of edges in G and n denotes the number of nodes in G. Then cyclomatic complexity CC is defined as $CC(G) = e - n + 2$.

It is worth noting that some researchers oppose using cyclomatic complexity for fault prediction. Fenton and Pfleeger argue that it is highly correlated with the lines of code, thus it carries little information [27]. However, other researchers used McCabe's complexity to build successful fault predictors [1]. Also industry keeps recognizing cyclomatic complexity measure as useful and uses it extensively, as it is straightforward and can be communicated across the different levels of development stakeholders [28]. In this research the latter opinions are followed.

3.1.4. Classifier

In this study, the authors opted for using a Naïve Bayes classifier. Naïve Bayes classifiers are a family of supervised learning algorithms based on applying Bayes' theorem with naïve independence assumption between the features. In preliminary experiments, this classifier achieved significantly higher recall than other classifiers that were preliminary considered. Also, as mentioned in section 2, it achieved best results in previous fault prediction studies [1].

It should be noted that for a class variable y and features $x_1, \ldots, x_n$, Bayes' theorem states the following relationship:

$$P(y|x_1, \ldots, x_n) = \frac{P(y)P(x_1, \ldots, x_n|y)}{P(x_1, \ldots, x_n)}.$$

This relationship can be simplified using the naïve independence assumption:

$$P(y|x_1, \ldots, x_n) = \frac{P(y)\prod_{i=1}^{n} P(x_i|y)}{P(x_1, \ldots, x_n)}.$$

Since $P(x_1, \ldots, x_n)$ does not depend on y, then the following classification rule can be used:

$$\widehat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i|y),$$

where $P(y)$ and $P(x_i|y)$ can be estimated using the training set. There are multiple variants of

the Naïve Bayes classifier; in this paper a Gaussian Naïve Bayes classifier is used which assumes that the likelihood of features is Gaussian.

3.2. Classification problem implementation

3.2.1. Identification of instances

A fault predicting tool must be able to encode a project as a set of examples. The identification of instances is the first step of this process. This tool implements it as follows: (1) it retrieves a list of files in a project from a repository (Git); (2) it limits results to a source code (Python) files; (3) for each file it builds an Abstract Syntax Tree (AST) and walks the tree to find the nodes representing source code units (functions).

3.2.2. Feature extraction

A fault predictor expects instances to be represented by the vectors of features. This tool extracts those features in the following way. Halstead metrics are derived from the counts of operators and operands. To calculate them for a given instance, this tool performs the following steps: (1) it extracts a line range for a function from AST; (2) it uses a lexical scanner to tokenize function's source; (3) for each token it decides whether the token is an operator or an operand, or neither. First of all the token type is used to decide if it is an operator or operand, see Table 4.

If the token type is not enough to distinguish between an operator and an operand; then if tokenize.NAME indicates tokens are Python keywords, they are considered operators; otherwise they are considered operands. McCabe's complexity for functions is calculated directly from AST. Table 5 presents effects of Python statements on cyclomatic complexity score.

3.2.3. Training set generation

Creating a fault predicting tool applicable to many projects can be achieved either by training a universal model, or by training predictors individually for each project [29]. This research adopts the latter approach: for each project it generates a training set using data extracted from the given project repository. Instances in the training set have to be assigned to classes; in this case software functions have to be labelled as either *fault-prone* or *not fault-prone*. In previous studies, such labels were typically assigned by human experts, which is a tedious and expensive process. In order to avoid this step, this tool relies on the following general definition of fault-proneness:

Definition 3.5. For a given revision, function is *fault-prone* if it was fixed in one of K next commits, where the choice of K should depend on the frequency of commits.

The definition of fault proneness can be extended due to the fact that relying on a project architecture warehouse enables mining information in commit logs. For identification of commits as bug-fixing in this research a simple heuristic, frequently used in previous studies, was followed [30, 31].

Definition 3.6. Commit is *bug-fixing* if its log contains any of the following words: *bug*, *fix*, *issue*.

Obviously such a method of generating training data is based on the assumption that bug fixing commits are properly marked and contain only fixes, which is consistent with the best practices for Git [32]. It is worth noting that since this might not be the general case for all projects, the tool in its current format is not recommended for predicting faults in projects that do not follow these practices.

3.2.4. Training and predicting

Training a classifier and making predictions for new instances are the key parts of a fault predictor. For these phases, the tool relies on *GaussianNB* from the *Scikit-learn* (scikit-learn.org) implementation of the Naïve Bayes classifier.

4. Main result

The tool's performance was experimentally assessed on five arbitrarily selected open-source

Table 4. Operator and operand types

OPERATOR_TYPES = [tokenize.OP, tokenize.NEWLINE, tokenize.INDENT, tokenize.DEDENT]

OPERAND_TYPES = [tokenize.STRING, tokenize.NUMBER]

Table 5. Contribution of Python constructs to cyclomatic complexity

Construct	Effect	Reasoning
if	+1	An **if** statement is a single decision
elif	+1	The **elif** statement adds another decision
else	0	Does not cause a new decision - the decision is at the **if**
for	+1	There is a decision at the start of the loop
while	+1	There is a decision at the **while** statement
except	+1	Each **except** branch adds a new conditional path of execution
finally	0	The **finally** block is unconditionally executed
with	+1	The **with** statement roughly corresponds to a **try**/**except** block
assert	+1	The **assert** statement internally roughly equals a conditional statement
comprehension	+1	A *list/set/dict comprehension* of generator expression is equivalent to a **for** loop
lambda	+1	A *lambda function* is a regular function
boolean	+1	Every boolean operator (**and**, **or**) adds a decision point

Table 6. Projects used for evaluation

Project	Location at github.com
Flask	/mitsuhiko/flask
Odoo	/odoo/odoo
GitPython	/gitpython-developers/GitPython
Ansible	/ansible/ansible
Grab	/lorien/grab

Table 7. Summary of projects used for evaluation: projects' revisions (Rv) with corresponding number of commits (Co), branches (Br), releases (Rl) and contributors (Cn)

Project	Rv	Cm	Br	Rl	Cn
Flask	7f38674	2319	16	16	277
Odoo	898cae5	94106	12	79	379
GitPython	7f8d9ca	1258	7	20	67
Ansible	718812d	15935	34	76	1154
Grab	e6477fa	1569	2	0	32

projects of different characteristics: Flask – a web development micro-framework; Odoo – a collection of business apps; GitPython – a library to interact with Git repositories; Ansible – an IT automation system; Grab – a web scraping framework. Analyzed software varies in scope and complexity: from a library with narrow scope, through frameworks, to a powerful IT automation platform and a fully-featured ERP system. All projects are publicly available on GitHub (see Table 6) and are under active development.

Data sets for evaluation were generated from projects using method described in section 3, namely: features were calculated for revision HEAD $\sim$ 100, where HEAD is a revision specified in Table 7; functions were labeled as fault-prone if they were modified in bug-fixing commit between revisions HEAD $\sim$100 and HEAD; data set was truncated to files modified in any commit between revisions HEAD $\sim$100 and HEAD. Table 8 presents total count and incidence of fault-prone functions for each data set.

As defined in section 3, *recall* and *false positive rates* were used to assess the performance of fault predictors. In terms of these metrics, a good fault predictor should achieve: *high recall* – a fault predictor should identify as many faults in the project as possible; if two predictors obtain the same false positive rate, the one with higher recall is preferred, as it will yield more fault-prone functions; *low false positive rate* – code units identified as bug prone require developer action;

Table 8. Data sets used for evaluation

Project	Functions	Fault-prone	% fault-prone
Flask	786	30	3.8
Odoo	1192	50	4.2
GitPython	548	63	11.5
Ansible	752	69	9.2
Grab	417	31	7.4

Table 9. Results for the best predictor

Project	Recall		False positive rate	
	mean	SD	mean	SD
Flask	0.617	0.022	0.336	0.005
Odoo	0.640	< 0.001	0.234	0.003
GitPython	0.467	0.019	0.226	0.003
Ansible	0.522	< 0.001	0.191	0.002
Grab	0.416	0.010	0.175	0.004
In total:	0.531	< 0.03	0.240	< 0.03

the predictor with fewer false alarms requires less human effort, as it returns less functions that are actually not fault-prone.

It is worth noting that Zhang and Zhang [33] argue that a good prediction model should actually achieve both high recall and high precision. However, Menzies et al. [34] advise against using precision for assessing fault predictors, as it is less stable across different data sets than the false positive rate. This study follows this advice.

For this research a stratified 10-fold cross validation was used as a base method for evaluating predicting performance. K-fold cross validation divides instances from the training set into K equal sized buckets, and each bucket is then used as a test set for a classifier trained on the remaining $K-1$ buckets. This method ensures that the classifier is not evaluated on instances it used for learning and that all instances are used for validation.

As bug prone functions were rare in the training sets, folds were stratified, i.e. each fold contained roughly the same proportions of samples for each label.

This procedure was additionally repeated 10 times, each time randomizing the order of examples. This step was added to check whether predicting performance depends on the order of the training set. A similar process was used by other researchers (e.g. [1, 35]).

Main result 1. The fault predictor presented in this research achieved recall up to 0.64 with false positive rate 0.23 (mean recall 0.53 with false positive rate 0.24, see Table 9 for details).

It is worth noting that: the highest recall was achieved for project *Odoo*: 0.640; the lowest recall was achieved for project *Grab*: 0.416; the lowest false positive rate was achieved for project *Grab*: 0.175; the highest false positive rate was achieved for project *Flask*: 0.336. For all data sets recall was significantly higher than the false positive rate. The results were stable over consecutive runs; the standard deviation did not exceed 0.03, neither for recall nor for the false positive rate.

Main result 2. This research additionally supports the significance of applying the logarithmic filter, since the fault predictor implemented for this research without using this filter achieved significantly lower mean recall 0.328 with false positive rate 0.108 (see Table 10 for details).

Table 10. Results for the best predictor without the logarithmic filter

Project	Recall		False positive rate	
	mean	SD	mean	SD
Flask	0.290	0.037	0.119	0.004
Odoo	0.426	0.009	0.132	< 0.001
GitPython	0.273	0.016	0.129	0.006
Ansible	0.371	0.012	0.068	< 0.001
Grab	0.219	0.028	0.064	0.005
In total:	0.328		0.108	

It should be emphasised that similar significance was indicated in the case of the detectors for C/C++ and Java projects in [1].

5. Conclusions

In this study, machine learning methods were applied to a software engineering problem of fault prediction. Fault predictors can be useful for directing quality assurance efforts. Prior studies showed that static code features can be used for building practical fault predictors for C/C++ and Java projects. This research demonstrates

that these techniques also work for Python, a popular programming language that was omitted in previous research. The tool resulting from this research is a function-level fault prediction tool for Python projects. Its performance was experimentally assessed on five open-source projects. On selected projects the tool achieved recall up to 0.64 with false positive rate 0.23, mean recall 0.53 with false positive rate 0.24. Leading fault predictors trained on NASA data sets achieved higher mean recall 0.71 with similar false positive rate 0.25 [1]. Labour intensive, manual code inspections can find about 60% of defects [36]. This research is close to reaching a similar level of recall. The performance of this tool can be perceived as satisfactory, certainly proving the hypothesis that predicting faults for Python programs has a similar potential to that of C/C++ and Java programs, and that more thorough future research in this area is worth conducting.

5.1. Threats to validity

Internal There are no significant threats to internal validity. The goal was to take an approach inspired by the experiments conducted by Menzies et al. [1] The experimental results for Python demonstrated to be consistent with the ones reported for C/C++ and Java, claiming that: static code features are useful for the identification of faults, fault predictors using the Naïve Bayes classifier perform well, however, using a logarithmic filter is encouraged, as it improves predicting performance. Using other methods of extracting features used for machine learning (i.e. Python features which are absent in C/C++ or Java), could potentially lead to a better performance of the tool.

External There are threats to external validity. The results obtained in this research are not valid for generalization from the context in which this experiment was conducted to a wider context. More precisely, the range of five arbitrarily selected software projects provides experimental evidence that this direction of research is worth pursuing; however, by itself it does not provide enough evidence for general conclusions and more thorough future research is required. Also the tool performance was assessed only in terms of recall and false positive rates, it has not been actually verified in practice. It is thus possible that the tool current predicting ability might prove not good enough for practical purposes and its further development will be required. Therefore, the conclusion of the universal practical applicability of such an approach cannot be drawn yet.

Construct There are no significant threats to construct validity. In this approach the authors were not interested in deciding whether it is a well selected machine learning technique, project attributes used for learning or the completeness of fault proneness definition for the training-set that were mainly contributing to the tool performance. The important conclusion was that the results obtained do not exclude but support the hypothesis, that automated fault prediction in Python allows to obtain accuracy comparable to the results obtained for other languages and to human-performed fault prediction, hence they encourage more research in this area. Thus, the results provided in this paper serve as an example and the rough estimation of predicting performance expected nowadays from fault predictors using static code features. There are few additional construct conditions worth mentioning. As discussed in section 3, the tool training set generation method relies on project change logs being part of the project architecture warehouse. If bug-fixing commits are not properly labelled, or contain not only fixes, then the generated data sets might be skewed. Clearly, the performance of the tool can be further improved, as it is not yet as good as the performance of fault predictors for C/C++ and Java; the current result is a good start for this improvement. Comparing the performance of classifiers using different data sets is not recommended, as predictors performing well on one set of data might fail on another.

Conclusion There are no significant threats to conclusion validity. Fault recall (detection rate) alone is not enough to properly assess the performance of a fault predictor (i.e. a trivial fault predictor that labels all functions as fault-prone achieves total recall), hence the focus on both re-

call (detection) and false positives (false alarms). Obviously the false positive rate of a fault predictor should be lower than its recall, as a predictor randomly labelling p of functions as fault-prone on average achieves a recall and false positive rate of p. This has been achieved in this study, similarly to [1]. From the practical perspective, in this research the goal recognizing automatically as many relevant (erroneous) functions as possible, which later should be revised manually by programmers; that is the authors were interested in achieving high recall and trading precision for recall if needed. From the perspective of this research goals, evaluating classifiers by measures other than those used in [1] (i.e. using other elements in the confusion matrix) was not directly relevant for the conclusions presented in this paper.

5.2. Future research

Additional features As mentioned in section 3, static code metrics are only a subset of features that can be used for training fault predictors. In particular, methods utilizing previous defect data, such as, [37] can also be useful for focusing code inspection efforts [38, 39]. Change data, such as code churn or fine-grained code changes were also reported to be significant bug indicators [40–42]. Adding support for these features might augment their fault predicting capabilities. Moreover, further static code features, such as object oriented metrics defined by Chidamber and Kemerer [43] can be used for bug prediction [32, 44]. With more attributes, adding a feature selection step to the tool might also be beneficial. Feature selection can also improve training times, simplify the model and reduce overfitting.

Additional algorithms The tool uses a Naïve Bayes classifier for predicting software defects. In preliminary experiments different learning algorithms were assessed, but they performed significantly worse. It is possible that with more features supplied and fine-tuned parameters these algorithms could eventually outperform the Naïve Bayes classifier. Prediction efficiency could also be improved by including some strategies for eliminating class imbalance [45] in the data sets. Researchers also keep proposing more sophisticated methods for identifying bug-fixing commits than the simple heuristic used in this research, in particular high-recall automatic algorithms for recovering links between bugs and commits have been developed. Integrating algorithms, such as [46] into a training set generation process could improve the quality of the data and, presumably, tool predicting performance.

Additional projects In preliminary experiments, a very limited number of Python projects were used for training and testing. Extending the set of Python projects contributing to the training and testing sets is needed to generalize the conclusions. The selection of additional projects should be conducted in a systematic manner. A live code could be used for predictor evaluation [22], which means introducing predictors into the development toolsets used by software developers in live software projects. The next research steps should involve a more in-depth discussion about the findings on the Python projects, in particular identification why in some projects the proposed techniques have a better performance than in other projects.

References

[1] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, Vol. 33, No. 1, 2007, pp. 2–13.

[2] E.W. Dijkstra, "Letters to the editor: go to statement considered harmful," *Communications of the ACM*, Vol. 11, No. 3, 1968, pp. 147–148.

[3] J. McCarthy, P.W. Abrams, D.J. Edwards, T.P. Hart, and M.I. Levin, *Lisp 1.5 programmer's manual*. The MIT Press, 1962.

[4] W. Royce, "Managing the development of large software systems: Concepts and techniques," in *Technical Papers of Western Electronic Show and Convention (WesCon)*, 1970, pp. 328–338.

[5] K. Beck, "Embracing change with extreme programming," *IEEE Computer*, Vol. 32, No. 10, 1999, pp. 70–77.

[6] R. Kaufmann and D. Janzen, "Implications of test-driven development: A pilot study," in *Companion of the 18th annual ACM SIGPLAN Conference on Object-oriented Programming,*

Systems, Languages, and Applications, ser. OOPSLA '03. New York, NY, USA: ACM, 2003, pp. 298–299.

[7] R. Dąbrowski, "On architecture warehouses and software intelligence," in *Future Generation Information Technology*, ser. Lecture Notes in Computer Science, T.H. Kim, Y.H. Lee, and W.C. Fang, Eds., Vol. 7709. Springer, 2012, pp. 251–262.

[8] R. Dąbrowski, K. Stencel, and G. Timoszuk, "Software is a directed multigraph," in *5th European Conference on Software Architecture ECSA*, ser. Lecture Notes in Computer Science, I. Crnkovic, V. Gruhn, and M. Book, Eds., Vol. 6903. Essen, Germany: Springer, 2011, pp. 360–369.

[9] R. Dąbrowski, G. Timoszuk, and K. Stencel, "One graph to rule them all (software measurement and management)," *Fundamenta Informaticae*, Vol. 128, No. 1-2, 2013, pp. 47–63.

[10] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," National Institute of Standards and Technology, Tech. Rep., 2002. [Online]. https://pdfs.semanticscholar.org/9b68/5f84da00514397d9af7f27cc0b7db7df05c3.pdf

[11] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A.B. Bener, "Defect prediction from static code features: Current results, limitations, new approaches," *Automated Software Engineering*, Vol. 17, No. 4, 2010, pp. 375–407.

[12] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Transactions on Software Engineering*, Vol. 34, No. 4, 2008, pp. 485–496.

[13] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Transactions on Software Engineering*, Vol. 38, No. 6, 2012, pp. 1276–1304.

[14] B.W. Boehm, "Software risk management," in *ESEC '89, 2nd European Software Engineering Conference*, ser. Lecture Notes in Computer Science, C. Ghezzi and J.A. McDermid, Eds., Vol. 387. Springer, 1989, pp. 1–19.

[15] T.M. Khoshgoftaar and N. Seliya, "Fault prediction modeling for software quality estimation: Comparing commonly used techniques," *Empirical Software Engineering*, Vol. 8, No. 3, 2003, pp. 255–283.

[16] A.A. Porter and R.W. Selby, "Empirically guided software development using metric-based classification trees," *IEEE Software*, Vol. 7, No. 2, 1990, pp. 46–54.

[17] T. Menzies, J. DiStefano, A. Orrego, and R.M. Chapman, "Assessing predictors of software defects," in *Proceedings of Workshop Predictive Software Models*, 2004.

[18] T. Menzies, R. Krishna, and D. Pryor, *The Promise Repository of Empirical Software Engineering Data*, North Carolina State University, Department of Computer Science, (2015). [Online]. http://openscience.us/repo

[19] T. Menzies, J.S.D. Stefano, K. Ammar, K. McGill, P. Callis, R.M. Chapman, and J. Davis, "When can we test less?" in *9th IEEE International Software Metrics Symposium (METRICS)*, Sydney, Australia, 2003, p. 98.

[20] T. Menzies, J.S.D. Stefano, and M. Chapman, "Learning early lifecycle IV&V quality indicators," in *9th IEEE International Software Metrics Symposium (METRICS)*, Sydney, Australia, 2003, pp. 88–97.

[21] T. Menzies and J.S.D. Stefano, "How good is your blind spot sampling policy?" in *8th IEEE International Symposium on High-Assurance Systems Engineering (HASE)*. Tampa, FL, USA: IEEE Computer Society, 2004, pp. 129–138.

[22] M. Lanza, A. Mocci, and L. Ponzanelli, "The tragedy of defect prediction, prince of empirical software engineering research," *IEEE Software*, Vol. 33, No. 6, 2016, pp. 102–105.

[23] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning*, ser. Adaptive computation and machine learning. The MIT Press, 2012.

[24] R.C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[25] T.J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, 1976, pp. 308–320.

[26] M.H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Ltd, 1977.

[27] N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed. Boston, MA, USA: Course Technology, 1998.

[28] C. Ebert and J. Cain, "Cyclomatic complexity," *IEEE Software*, Vol. 33, No. 6, 2016, pp. 27–29.

[29] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, "Towards building a universal defect prediction model," in *11th Working Conference on Mining Software Repositories, MSR*, P.T. Devanbu, S. Kim, and M. Pinzger, Eds. Hyderabad, India: ACM, 2014, pp. 182–191.

[30] S. Kim, "Adaptive bug prediction by analyzing project history," Ph.D. dissertation, University of California at Santa Cruz, Santa Cruz, CA, USA, 2006, aAI3229992.

[31] S. Kim, T. Zimmermann, K. Pan, and E.J. Whitehead, Jr., "Automatic identification of bug-introducing changes," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Tokyo, Japan: IEEE Computer Society, 2006, pp. 81–90.

[32] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering*, Vol. 31, No. 10, 2005, pp. 897–910.

[33] H. Zhang and X. Zhang, "Comments on 'Data mining static code attributes to learn defect predictors'," *IEEE Transactions on Software Engineering*, Vol. 33, No. 9, 2007, pp. 635–637.

[34] T. Menzies, A. Dekhtyar, J.S.D. Stefano, and J. Greenwald, "Problems with precision: A response to 'Comments on data mining static code attributes to learn defect predictors'," *IEEE Transactions on Software Engineering*, Vol. 33, No. 9, 2007, pp. 637–640.

[35] M.A. Hall and G. Holmes, "Benchmarking attribute selection techniques for discrete class data mining," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 15, No. 6, 2003, pp. 1437–1447.

[36] F. Shull, V.R. Basili, B.W. Boehm, A.W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M.V. Zelkowitz, "What we have learned about fighting defects," in *8th IEEE International Software Metrics Symposium (METRICS)*. Ottawa, Canada: IEEE Computer Society, 2002, p. 249.

[37] S. Kim, T. Zimmermann, E.J. Whitehead, Jr., and A. Zeller, "Predicting faults from cached history," in *29th International Conference on Software Engineering (ICSE 2007)*. Minneapolis, MN, USA: IEEE, 2007, pp. 489–498.

[38] F. Rahman, D. Posnett, A. Hindle, E.T. Barr, and P.T. Devanbu, "BugCache for inspections: hit or miss?" in *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13)*, T. Gyimóthy and A. Zeller, Eds. Szeged, Hungary: ACM, 2011, pp. 322–331.

[39] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E.J.W. Jr., "Does bug prediction support human developers? Findings from a Google case study," in *35th International Conference on Software Engineering, ICSE*, D. Notkin, B.H.C. Cheng, and K. Pohl, Eds. San Francisco, CA, USA: IEEE / ACM, 2013, pp. 372–381.

[40] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *27th International Conference on Software Engineering (ICSE)*, G. Roman, W.G. Griswold, and B. Nuseibeh, Eds. ACM, 2005, pp. 284–292.

[41] A.E. Hassan, "Predicting faults using the complexity of code changes," in *31st International Conference on Software Engineering, ICSE*. Vancouver, Canada: IEEE, 2009, pp. 78–88.

[42] E. Giger, M. Pinzger, and H.C. Gall, "Comparing fine-grained source code changes and code churn for bug prediction," in *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR*, A. van Deursen, T. Xie, and T. Zimmermann, Eds. ACM, 2011, pp. 83–92.

[43] S.R. Chidamber and C.F. Kemerer, "Towards a metrics suite for object oriented design," in *Sixth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91)*, A. Paepcke, Ed. Phoenix, Arizona, USA: ACM, 1991, pp. 197–211.

[44] R. Shatnawi, "Empirical study of fault prediction for open-source systems using the Chidamber and Kemerer metrics," *IET Software*, Vol. 8, No. 3, 2014, pp. 113–119.

[45] N.V. Chawla, N. Japkowicz, and A. Kotcz, "Editorial: Special issue on learning from imbalanced data sets," *ACM SIGKDD Explorations Newsletter*, Vol. 6, No. 1, Jun. 2004, pp. 1–6.

[46] R. Wu, H. Zhang, S. Kim, and S. Cheung, "ReLink: recovering links between bugs and changes," in *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13)*, T. Gyimóthy and A. Zeller, Eds. Szeged, Hungary: ACM, 2011, pp. 15–25.

[47] T. Gyimóthy and A. Zeller, Eds., *SIGSOFT-/FSE 11 Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering*. Szeged, Hungary: ACM, 2011.

Appendix

The implementation of the method used in this study for predictor evaluation is outlined below, it can be used to reproduce results of the experiments.

```
# imports available on github.com
import git
import numpy as np
from sklearn import cross_validation
from sklearn import metrics
from sklearn import naive_bayes
from sklearn import utils
from scary import dataset
from scary import evaluation

def run():
        projects = [
                "path/to/flask",
                "path/to/odoo",
                "path/to/GitPython",
                "path/to/ansible",
                "path/to/grab",
        ]
        classifier = naive_bayes.GaussianNB()
        EvaluationRunner(projects, classifier).evaluate()

class EvaluationRunner:
        def __init__(self, projects, classifier, from_revision="HEAD~100", to_revision="HEAD",
                shuffle_times=10, folds=10):
                self.projects = projects
                self.classifier = classifier
                self.from_revision = from_revision
                self.to_revision = to_revision
                self.shuffle_times = shuffle_times
                self.folds = folds

        def evaluate(self):
                total_score_manager = self.total_score_manager()
                for project in self.projects:
                        project_score_manager = self.project_score_manager()
                        training_set = self.build_training_set(project)
                        for data, target in self.shuffled_training_sets(training_set):
                                predictions = self.cross_predict(data, target)
                                confusion_matrix = self.confusion_matrix(predictions, target)
                                total_score_manager.update(confusion_matrix)
                                project_score_manager.update(confusion_matrix)
                        self.report_score(project, project_score_manager)
                self.report_score("TOTAL", total_score_manager)

        def project_score_manager(self):
                return ScoreManager.project_score_manager()

        def total_score_manager(self):
                return ScoreManager.total_score_manager()
```

```
50
51          def build_training_set(self, project):
52                  repository = git.Repo(project)
53                  return dataset.TrainingSetBuilder.build_training_set(repository,
54                          self.from_revision, self.to_revision)
55
56          def shuffled_training_sets(self, training_set):
57                  for _ in range (self.shuffle_times):
58                          yield utils.shuffle(training_set.features, training_set.classes)
59
60          def cross_predict(self, data, target):
61                  return cross_validation.cross_val_predict(self.classifier, data, target,
62                          cv=self.folds)
63
64          def confusion_matrix(self, predictions, target):
65                  confusion_matrix = metrics.confusion_matrix(target, predictions)
66                  return evaluation.ConfusionMatrix(confusion_matrix)
67
68          def report_score(self, description, score_manager):
69                  print(description)
70                  score_manager.report()
71
72  class ScoreManager:
73          def __init__(self, counters):
74                  self.counters = counters
75
76          def update (self, confusion_matrix):
77                  for counter in self.counters:
78                          counter.update(confusion_matrix)
79
80          def report(self):
81                  for counter in self.counters:
82                          print(counter.description, counter.score)
83
84          @classmethod
85          def project_score_manager(cls):
86                  counters = [MeanScoreCounter(RecallCounter),
87                          MeanScoreCounter(FalsePositiveRateCounter),]
88                  return cls(counters)
89
90          @classmethod
91          def total_score_manager(cls):
92                  counters = [RecallCounter(),
93                          FalsePositiveRateCounter(),]
94                  return cls(counters)
95
96  class BaseScoreCounter:
97          def update (self, confusion_matrix):
98                  raise NotImplementedError
99
100         @property
101         def score (self):
102                 raise NotImplementedError
103
```

```
104         @property
105         def decription(self):
106                 raise NotImplementedError
107
108 class MeanScoreCounter(BaseScoreCounter):
109         def __init__(self, partial_counter_class):
110                 self.partial_counter_class= partial_counter_class
111                 self.partial_scores = []
112
113         def update(self, confusion_matrix):
114                 partial_score = self.partial_score(confusion_matrix)
115                 self.partial_scores.append(partial_score)
116
117         def partial_score(self, confusion_matrix):
118                 partial_counter = self.partial_counter_class()
119                 partial_counter.update(confusion_matrix)
120                 return partial_counter.score
121
122         @property
123         def score (self):
124                 return np.mean(self.partial_scores), np.std(self.partial_scores)
125
126         @property
127         def description(self):
128                 return "mean {}".format(self.partial_counter_class().description)
129
130 class RecallCounter(BaseScoreCounter):
131         def __init__(self):
132                 self.true_positives = 0
133                 self.false_negatives = 0
134
135         def update(self, confusion_matrix):
136                 self.true_positives += confusion_matrix.true_positives
137                 self.false_negatives += confusion_matrix.false_negatives
138
139         @property
140         def score(self):
141                 return self.true_positives/(self.true_positives+self.false_negatives)
142
143         @property
144         def description(self):
145                 return "recall"
146
147 class FalsePositiveRateCounter(BaseScoreCounter):
148         def __init__(self):
149                 self.false_positives = 0
150                 self.true_negatives = 0
151
152         def update (self, confusion_matrix):
153                 self.false_positives += confusion_matrix.false_positives
154                 self.true_negatives += confusion_matrix.true_negatives
155
156         @property
157         def score (self):
```

```
158                 return self.false_positives/(self.false_positives+self.true_negatives)
159
160         @property
161         def description(self):
162                 return "false positive rate"
163
164 if __name__ == "__main__":
165         run ()
```