Using Data Mining for Static Code Analysis of C

Conference Paper · December 2012

DOI: 10.1007/978-3-642-35527-1_50

3 authors, including:

Stefan Axelsson
Norwegian University of Science and Technology at Gjøvik, Norway
62 PUBLICATIONS **3,118** CITATIONS

SEE PROFILE

Using Data Mining for Static Code Analysis of C

Hannes Tribus, Irene Morrigl, and Stefan Axelsson

Blekinge Institute of Technology

Abstract. Static analysis of source code is one way to find bugs and problems in large software projects. Many approaches to static analysis have been proposed. We proposed a novel way of performing static analysis. Instead of methods based on semantic/logic analysis we apply machine learning directly to the problem. This has many benefits. Learning by example means trivial programmer adaptability (a problem with many other approaches), learning systems also has the advantage to be able to generalise and find problematic source code constructs that are not exactly as the programmer initially thought, to name a few. Due to the general interest in code quality and the availability of large open source code bases as test and development data, we believe this problem should be of interest to the larger data mining community. In this work we extend our previous approach and investigate a new way of doing feature selection and test the suitability of many different learning algorithms. This on a selection of problems we adapted from large publicly available open source projects. Many algorithms were *much* more successful than our previous proof-of-concept, and deliver practical levels of performance. This is clearly an interesting and minable problem.

Keywords: software engineering, static analysis, application

1 Introduction

The finding and fixing of bugs, and other problems, is important to the writing of quality software. It is an important part of software development. Several different approaches have been proposed. These range from the widely used; *coding rules*, which focus on avoiding the introduction of faults, via *manual inspection* (*code review*), and *testing*, to more sophisticated methods like tool supported *error prediction* or *detection*. Today the latter is becoming popular as they can help directly by pointing to the places in the source code where an actual fault is presumed to exist. Depending on how these tools operate, they are able to find more or less complicated faults. One class of approaches is *static analysis*, by which is meant that the source code of the program is analysed for flaws that can be found without having to execute the program. This approach is especially useful in finding code that hides flaws that are more difficult to find with some form of dynamic analysis, for example those relating to non-functional requirements such as security flaws etc. This is due to the problem of the *coverage* of testing tools. Unusual flows of execution, such as those that involve error conditions, may receive insufficient testing unless *special* care is

taken. Static analysis has many other advantages; it can be applied early in the development process to provide early fault detection, the code does not have to be fully functional, or even executable, and test cases do not have to have be developed.

Several tools of this nature already exist. Some are even meant to be programmer adaptable (such as *Coverity* [6]), but in actual practise this feature is seldom taken advantage of, due to its perceived difficulty [6]. In order to address esp. the problem of programmer adaptability, we have previously turned to supervised machine learning, in order to provide trivially *programmer extensible* static code analysis. In this approach the programmer first trains the analyser by providing it with examples of correct, and problematic, code. The trained analyser is then run on code with potential problems and (hopefully) points out problematic situations, without too many false alarms, or missed problems. Another advantage of machine learning, is its capacity to generalise from a set of given examples. Correctly applied, machine learning has the capacity to *surprise*, by producing results that were previously unanticipated, but still relevant, and correct.

In order to utilise a machine learning framework, example and evaluation data has to be prepared. Hence, the raw source code has to be converted into suitable input for a machine learning algorithm. Thus, one of the goals of this project was the development of a feature selection model for a representative procedural language (the C programming language in this case) by using an appropriate parser. We will used the data to evaluate possible machine learning algorithms, and then compared and evaluated them in terms of accuracy and false positive/negative rate. We used source code from various open-source projects for the experiment. We sought to answer: What are the relevant source code features needed to classify an instance of code as faulty or correct? How can those features be transformed and represented as input to the machine learner? How accurate is the resulting static analyser/machine learning algorithm used?

The rest of this paper is organised as follows: We start by describing related work in section 2. Then, in section 3, the types of software flaws we focused on are described. The actual approach to converting *C language* source code to machine learning feature vectors is described in section 4. The experiments and results, to validate the approach are explained in section 5, with discussion, conclusions and future work finishing the paper in section 6.

2 Related Work

Both machine learning and static code analysis have been widely studied. However, when it comes to the application of machine learning *to static* analysis we know of only our own previous work in the area: Sidlauskas et.al. [1]. The work demonstrated that this approach was possible and that the static analyser even managed to generalise from e.g. a faulty *strcpy* example to detecting a similarly incorrect application of *strcat*. However, the work also had several limitations that we try to address here: only one machine learning algorithm based on the

normalised compression distance (NCD) was tried, the experimental data was limited in scope, and the results in terms of accuracy etc. were several percentage points (even tens of percent) worse than what we achieve here.

While there are no direct analogs, there is work in the related fields; *fault prediction* and *fault detection*. Most of the work done so far in the area of *fault prediction* deals with the prediction of faults in source code. The basic idea here is to extract properties, or attributes, of the code which allows the drawing of conclusions about the likelihood of the presence of faults. Properties in this case could be metrics such as *lines-of-code* (LOC), or *cyclomatic complexity* (CC), or in the case of object oriented programming even *number-of-children* (NOC) etc. These are also known collectively as "CK-Metrics". A study by Fenton et.al. [7] criticised the models presented so far, called *single-issue models*, and suggested instead the use of machine learning techniques in order to arrive at a more general model for predicting faults in software. This approach was confirmed by the studies of Turhan and Kutlubay [16]. Most of the work in the prediction area deal with the code metrics above, but there are a few different approaches. Challagulla [4] showed that similar results using code metrics can be obtained by using design or change metrics, such as the number of times a file has been changed, the expertise of the person affecting the change etc. This was supported by Heckman and Williams [9] and Moser et.al. [13]. Another contribution to the area by Jiang et.al. [11] who compared the performance of design and code metrics in fault prediction models. They came to the conclusion that models using a combination of these metric outperform models that use either code or design metrics alone.

Despite the approaches above being more or less successful, they all try to draw conclusions about the presence of errors in the source code by studying meta data *about* the project instead of using the actual source to detect the faults. Burn et.al. showed in a case study [3] that support vector machines (SVM) and decision trees (DT), previously trained on faulty code execution vectors and corrected versions, can be used to identify errors during program execution. Despite their approach using dynamic analysis instead of static, it demonstrated that machine learning could be used for fault detection. A similar approach by Kreimer [12] is a tool that is able to detect design flaws during execution. Similarly, Song et al. [14] used the *association rule mining* (ARM) machine learner to find dynamic execution patterns that are similar or related to previously found errors. Finally Jiang et.al. [10] the authors successfully applied neural networks to the same problem.

Approaches to static analysis ranges from the very simple (searching for pure textual patterns of known problems) to the very complex. Most advanced methods depend on some form of formal method, based on e.g. denotational/axiomatic/operational semantics, or abstract interpretation. Practical implementations of these theories include:

Model-checking Given a model of the system (e.g. a FSM), and a specification (e.g. a temporal specification) determine if the model meets the specification.

Data-flow analysis At control points in the program information about the possible set of values for e.g. variables is tracked and this information propagated to later stages of the analysis.

Abstract interpretation The partial abstract execution of a program that preserves and gathers information about data-flow and control-flow such that important information about properties of the programme can be studied.

The area has seen extensive study in the past decades, and we can scarcely do it justice here.

3 Types of Software Faults

In order to develop a method of detecting faults in source code, we need to define what we mean by fault in this context. A multitude of different kinds of software flaws have been categorised. We have decided to limit our research to the ones described below, as they are important in that they have shown to be the cause of many problems, security and otherwise. Most of these flaws have been adapted from [5].

Buffer Overflow A buffer overflow or buffer overrun, results when a program inadvertently stores data outside an allocated buffer, most commonly by continuing to write data past the end of the allocated storage. If the buffer is allocated on the stack, this allows an attacker to overwrite active parts of the stack, including the return address of the current function. This leads trivially to a severe security flaw. Even the inadvertent triggering of a buffer overrun leads to data corruption, and most often a program crash. This flaw was popularised in the nineties and exploited heavily which lead to some calling it "the vulnerability of the nineties."

Memory Handling This second kind of error regards dynamic memory allocation and de-allocation. The two main errors in this group are the use of the memory after it has been freed, and the freeing of memory that has already been freed before (*double-free*). Both can lead to a buffer overflow attack. A third problem, which is usually not that serious, but a waste of memory is whenever memory is allocated, but not subsequently used.

Dereferencing a Null pointer A very common fault is the dereferencing of a null pointer. This fault occurs whenever a pointer to NULL is used, meaning that a pointer variable is used before e.g. it has been assigned a memory address. Dereferencing such a pointer will cause a program crash.

Control Flow By this we mean failing to check a return value and, failure to release a resource. Whenever a functions return status is not checked, for example, assigned variables could be left with undefined values. This often leads to a null pointer reference.

Signed to Unsigned Conversion The problem here is the automatic conversion between signed to unsigned values in the C-language. If a programmer checks a signed value and finds that it is smaller than a e.g. a buffer size (due to it being a negative value of large magnitude) this value could then be automatically converted to a large positive value when it is passed as a parameter to a

function that take an unsigned value. This often leads to a situation where much more data is allocated/read than there is room for, and hence a buffer overflow.

4 Static Analysis by Data Mining

We now arrive at the problems of how to transform static source code to a suitable format for data mining, which data to use to train the classifier, and which classifier/learning algorithm that bests fits the problem.[1]

In order to perform our experiments we have chosen to us the WEKA data mining framework [8]. The main reasons were its popularity, and suitability, for the task, including its offering a wide variety of different learners. Furthermore all these learners can be used with the same kind of input and the same configuration. WEKA also provides a good framework for performing evaluation of the experiments.

In order to address the problem of feature extraction we wrote a prototype tool (*CMore*), that analyses C source code by extracting information about basic functions and procedures. The basic idea is that this should enable us to follow the execution flow of a program and capture the state of all the variables involved. Having all the state, it is possible to detect several kind of possible memory access faults such as are null-pointer-references and misuse of memory (see above).

CMore first divides the input into function definitions (with parameters and types), definitions, variable declarations, types, structure types etc. Information about these are stored in a data base for future reference. When the general structural information about the program have been identified, CMore goes into a more detailed processing stage where each line of the source code is analysed to see if it contains a mathematical expression, an assignment, a declaration, function call, control flow instruction etc.

```
@RELATION cmore

@ATTRIBUTE operation  {Functioncall,Assignment,Usage}

@ATTRIBUTE left_type  {FILE, Function, header, char, int, Reference, Simple, void, ..
    .}
@ATTRIBUTE left_name  {open, alloc, close, free, gets, printf, seek, strdup, strcpy,
     strncpy, strcat}
@ATTRIBUTE left_isRef   {true,false}
@ATTRIBUTE left_isNull  {true,false}
@ATTRIBUTE left_isAss   {true,false}
@ATTRIBUTE left_isUse   {true,false}
@ATTRIBUTE left_isChk   {true,false}

[...]

@ATTRIBUTE orig_stmt    STRING
@ATTRIBUTE err_free     {true,false}

@DATA
Assignment,int,?,true,true,false,false,false,Reference,?,true,false,false,true,false
    ,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,"co_=_&cp",false
Assignment,int,?,true,true,false,false,false,Reference,?,true,false,false,false,true,false
    ,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,"com_=_&ptr1",false
...
```

Listing 1.1. Snapshot of the final ARFF

[1] The treatment here is necessarily short, due to space constraints. Much more details can be found in [15].

After this analysis, an attribute file is written that details the information we have collected about the program. Listing 1.1 shows an example of the output from this stage. The structure is quite simple and emulates the structure of a typical statement. The first attribute states what kind of operation the statement is; *assignment*, *usage* or *function call*. Depending on this attribute one or more of the groups (*left*, *right*, *par1–parX*) are emitted. A group contains seven attributes. The last two attributes contain the real statement needed to identify the faulty line in the code and the *error-free* flag.

The following examples should clarify the behaviour of the single groups and why they are emitted:

a = b; $--> $ In this case *a* is used for the *left* attribute and *b* to fill the *right* group

aProcedure(a); $--> $ In this case, *aProcedure* is the *right* attribute and *a* is used for *par1*

a = aFunction(b,c); $--> $ In this case *a* is *left*; *aFunction* is *right* and *b*, and *c* are the two parameters

Note that one statement in the source code can produce multiple instances in the attribute file.

In order to help in training it is useful to have help in determining the differences between two versions of a program. This is because we need labelled instances of faulty and correct versions. Our input data will most often be in the form of two different versions of a source code file. One with an error ("bug") and one with the error fixed. In order to identify these differences we developed a tool, *Holmes*, to be able to pick out and flag these differences when given two ARFF files as input.

5 Experiment

In order to test this approach we performed an experiment using a number of publicly available software projects. Suitable projects for such an experiment have to be available (obviously), have a bug data base (that enables us to identify interesting software problems), and a version control repository (that enables us to access different versions of the software, one with bugs fixed).[2] However, in some cases we were unable to identify the precise flaw, and in those cases we resorted to artificially injecting a bug in the previous version. As we were uncertain of our initial feature selection, we reduced the number of features as the experiment progressed. We display the results of both the full and the reduced set. As a last step the best learners were tested on the complete input from one of the open source projects previously used to generate the training set for the learners, to evaluate its performance on a real data set. This to show that the classifiers are working in a more real world scenario.

[2] The connection between bug database and source code control is important, as it enables us to connect a bug report, which gives us a clue about the type of fault, with the actual change in the source code that fixed that error.

After extensive study we selected six candidates for the experiment: *Algoview*; a small program to visualise a program as a graph (*Sourceforge*), *Boxc*; a vector graphics language to simplify their creation (*Sourceforge*), *ClamAV*; an anti-virus program for Linux (*Sourceforge*), *PocketBook*; an open source firmware for e-book readers (*Sourceforge*), *FalconHTTP*; a HTTP server (*Google Code*), and *Siemens Programs*; a collection of programs with several versions, where faults have been introduced for educational purposes (*Other*[3]).

From the programs selected above it was possible to extract correct and faulty instances using the tools described above. In total we developed 496 instances of faulty and correct versions, and also some correct instances reflecting the most important standard situations (to give contrasting examples of correct usage). The latter were added to reduce false alarms on correct standard situations in source code (important for the second experiment).

We used the most common measures of success; *accuracy*, *false positive* and *false negative* rate. All experiments were run using ten-fold cross validation.

5.1 Accuracy

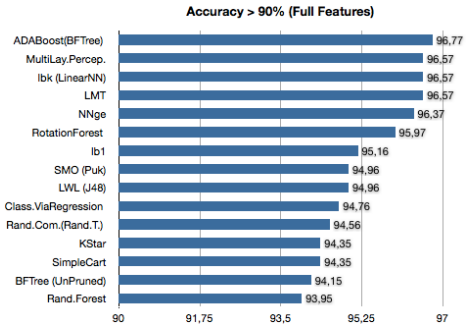

Fig. 1. Accuracy (Full)

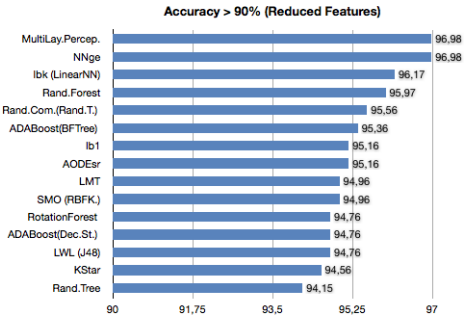

Fig. 2. Accuracy (Reduced)

The resulting ranking in terms of accuracy is shown in figure 1.[4] In the figure, of the seven classifiers performing better than 95%, three are based on nearest neighbour algorithms (*Ib1*, *Ibk*, *NNge*), three are tree based (*LMT*, *RotationForest* and *ADABoost* performed on a *BFTree*) and the the last based on artificial neural network; *MultiLayerPerceptron*.

Despite the fact that trees perform quite well on the given data set, study of the attributes used, showed that most of the time the reasoning was based on the real data type of the involved variables. To see what effect the removal of this data had, these features were removed in the second step of the experiment. The

[3] http://pleuma.cc.gatech.edu/aristotele/Tools/subject/index.html

[4] Due to space constraints, some of the graphs are truncated to focus on the interesting parts for the conclusions drawn, the complete data set is available on request.

Table 1. Average Accuracy per Category

	Accuracy		False positive		False negative	
Category	Full features	Reduced feature	Full features	Reduced feature	Full features	Reduced feature
Total average	82.0	82.4	0.24	0.23	0.14	0.14
Lazy	90.7	91.2	0.08	0.06	0.10	0.11
Tree	85.9	85.6	0.19	0.20	0.10	0.10
Functions	82.2	81.7	0.23	0.20	0.14	0.17
Bayes	81.4	84.0	0.17	0.13	0.20	0.18
Meta	80.4	80.8	0.30	0.30	0.11	0.11
Rule	79.4	79.7	0.22	0.17	0.14	0.10
Misc	61.8	65.0	0.14	0.09	0.56	0.55

results in terms of accuracy of this change is shown in figure 2. As expected the performance of the tree algorithms was reduced and *ADABoost* could not sufficiently improve matters. However, the performance of the *MultiLayerPerceptron* increased to be the highest overall, together with the *NNge* nearest neighbour learner.

This result is supported by the left table (*accuracy*) in 1 which shows the average accuracy per category (These categories are taken from the categories used in WEKA to describe the various classifiers). It shows that almost all classifiers improve by not taking the real name of the involved types under consideration, except the tree learners.

5.2 False Positive Rate

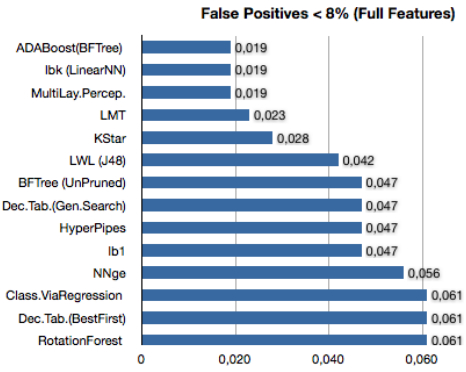

Fig. 3. False Positive (Full)

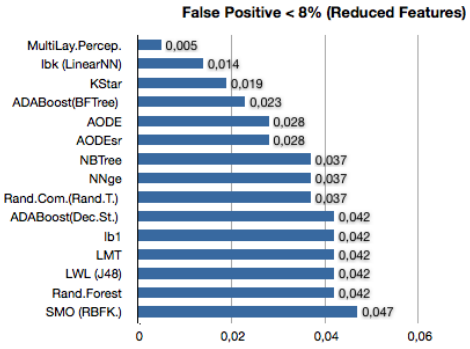

Fig. 4. False Positive (Reduced)

The false positive analysis paints a similar picture. Figure 3 shows the results for the full feature set and figure 4 for the reduced one. Among the best classifiers for the full data set are still the nearest neighbour algorithm and the *MultiLayerPerceptron*, together with the ADAboosted *BFTree* and some other trees like the *LMT* and a version of *J48*. As before when using the reduced

feature set, the trees fared worse, but the nearest neighbour algorithms and the *MultiLayerPerceptron* improved. The results are again supported by the averages per category which demonstrate that all algorithms except the trees gained from the reduced feature set (see table 1).

It is interesting to note that the *NNge* nearest neighbour algorithm was not as good when it came to false positive rate and also the feature reduction did not significantly increase its performance. This in combination with its good accuracy in fact meant that this classifier should have a low value when it comes to false negative rate

5.3 False Negative Rate

According to the results by Baca [2] it is very important for static analysis tools to have a low false positive rate. Otherwise, the users will quickly tend to distrust the results of the tool, even when they are correct, or even worse, introduces new faults at the location the tool indicates.

On the other hand, it is also important to have a low false negative rate, that is, that the tool finds most of the flaws present in the code. Thus, we need to find an acceptable trade off between false positives and false negatives.

The results in terms of false positive rate are shown in figure 5 for the complete set, and figure 6 for the reduced one.

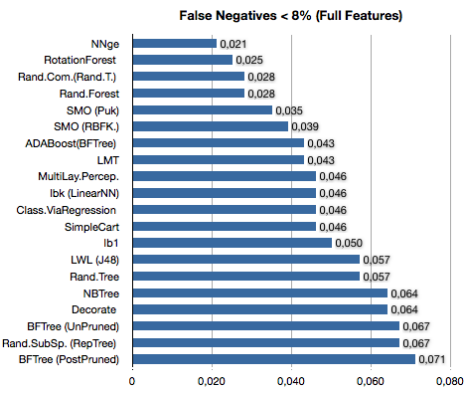

Fig. 5. False Negative (Full)

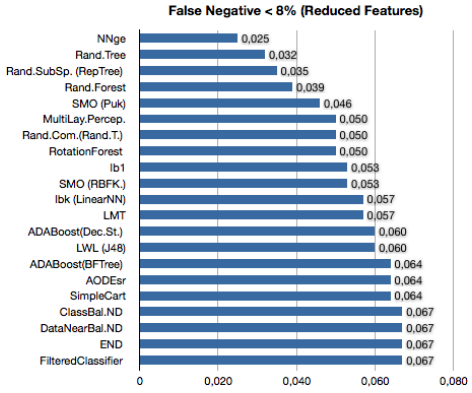

Fig. 6. False Negative (Reduced)

As expected the best performing algorithm according to this measure is *NNge* due to its bad performance in the false positive measure. However it is worth noticing that the tree algorithms perform quite well, and most of them are able to gain from the reduced feature set. Unfortunately as they are not as good in terms of accuracy. This means that their average false positive rate is also high. Generally this measure suffers from the reduction of the feature set, as the number of learners below 8% decreases. The average values of the various

categories do not change much due to the decrease shown in table 1. The only real notable difference is the rule based learners, but they were not among the top learners before the reduction, and they did not gain sufficiently to rank at the top afterwards.

5.4 Selected Project Evaluation Results

In the previous experiment there were only well sorted examples that had pairs of two or more instances showing the difference between correct and faulty source code. In order to test more real world performance of the method, we ran an experiment with the best performing machine learning algorithms against a complete data set obtained from *FalconHTTP*. *CMore* was able to extract over 4000 feature instances from this software. Using fault injection, 44 feature instances were marked as faulty, the remaining were considered fault free, even though we of course cannot prove this correct. For this experiment the whole set of instances used for the first experiment was used to train the six previously best performing classifiers. The new generated set of instances was used to test the performance. The experiment was performed for both the full and the reduced feature set. The results are listed in table 2, sorted by accuracy for the full feature set.

Table 2. Results from WEKA for the test experiment

		Full Features				Reduced Features			
		corr	incorr	FN	FP	corr	incorr	FN	FP
Ibk (LinearNN)	Lazy	98,434	1,566	0,064	0,015	96,311	3,689	0,064	0,036
Rand.Com.(RandT.)	Meta	97,979	2,021	0,083	0,019	94,795	5,205	0,083	0,052
LMT	Tree	97,423	2,577	0,043	0,025	97,423	2,577	0,043	0,025
NNge	Rule	96,059	3,941	0,064	0,039	96,059	3,941	0,064	0,039
RotationForest	Meta	94,543	5,457	0,083	0,054	94,543	5,457	0,083	0,054
MultiLay.Perc.	Functions	91,359	8,641	0,043	0,086	99,394	0,606	0,043	0,005
AdaBoost(BFTree)	Meta	77,160	22,840	0,083	0,230	73,421	26,579	0,083	0,268

The most important result of this experiment is that five out of the six tested classifiers achieved an accuracy of over 90%, which makes them applicable to real world problems.

Despite that we observed a gain in accuracy and false positive rate by reducing the feature set in the previous experiment, in this case only the *MultiLayerPerceptron* was able to profit. This learner increased its accuracy from ca. 91% to over 99%, reducing the false positive rate from over 8%, down to 0.5%.

Probably the most surprising result from this experiment, is the poor performance of the *ADABoost* classifier using both feature sets. The major difference between the data set used for this experiment and the one used in the previous (which is the training set for this experiment) is the relation between faulty and correct instances. Lines of correct code should (hopefully) greatly exceed faulty in a real world project. So the test set contained over 4000 feature instances of which only 44 were considered faulty, which is a realistic number, on the high side even.

6 Discussion, Conclusions and Future Work

Despite that our results show that this method could be a possible approach for future static analysis tools, there are some limitations of the current work. The most important limitation come from the feature extraction process and what ability it has to track sufficient information. Even though the proposed feature extraction method produces a good result, and is not completely *ad-hoc* like our previous approach, much research remains before this area has been sufficiently studied. The approach here should be seen as a first step.

Also, even though we have performed an experiment on several substantial open source projects, there is of course much to be done, to expand the data set, both in terms of identifying projects of different kinds (commercial etc.), and identifying important and relevant flaws and other software problems.

The results of our experiment show that the best performing machine learning algorithms perform well enough in terms of accuracy, false positive and false negative rate, to be useful in practise. It was possible to train seven classifiers to demonstrate an accuracy of over 95%, five to have a false positive rate below 4% and six to have a false negative rate below 4% for the full feature set and eight with accuracy over 95%, nine with false positive below 4% and four below 4% for false negative on the reduced feature set. To validate the results a second experiment was performed by running the best six machine learning algorithms on a more realistic problem. This showed that five out of six classifiers could meet the requirements, while the one using *ADABoost* on a *BFTree* suffered from reduced accuracy, from over 96.77% in the first experiment down to below 77.15% for the full feature set, and from 95.36% down to 73.43% for the reduced feature set (Similarly for false positive/negative rates).

In summary; the timely identification of software bugs is an industrially important problem. We feel we have demonstrated that this is clearly a data minable problem, with readily and publicly available data from which to build data sets. We would like to bring this data mining problem to the attention of the larger community.

References

1. Axelsson, S., Baca, D., Feldt, R., Sidlauskas, D., Kacan, D.: Detecting defects with an interactive code review tool based on visualisation and machine learning. the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE 2009) (2009)
2. Baca, D.: Automated static code analysis : a tool for early vulnerability detection. Department of Systems and Software Engineering, School of Engineering, Blekinge Institute of Technology, Karlskrona (2009), licentiatavhandling Ronneby : Blekinge tekniska högskola, 2009
3. Brun, Y., Ernst, M.D.: Finding latent code errors via machine learning over program executions. In: ICSE '04: Proceedings of the 26th International Conference on Software Engineering. pp. 480–490. IEEE Computer Society, Washington, DC, USA (2004)

4. Challagulla, V.U.B., Bastani, F.B., Yen, I.L., Paul, R.A.: Empirical assessment of machine learning based software defect prediction techniques. In: WORDS '05: Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems. pp. 263–270. IEEE Computer Society, Washington, DC, USA (2005)
5. Chess, B., West, J.: Secure Programming with Static Analysis. Addison Wesley Professional, Erewhon, 1.ed. edn. (2007)
6. Engler, D., Chelf, B., Chou, A., Hallem, S.: Checking system rules using system-specific, programmer-written compiler extensions. In: Proceedings of the 4th Symposium on Operating System Design and Implementation (OSDI 2000). USENIX, San Diego, California, USA (Oct 2000)
7. Fenton, N.E., Neil, M.: A critique of software defect prediction models. IEEE Trans. Softw. Eng. 25(5), 675–689 (1999)
8. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: An update. SIGKDD Explorations 11(1) (2009)
9. Heckman, S., Williams, L.: A model building process for identifying actionable static analysis alerts. In: ICST '09: Proceedings of the 2009 International Conference on Software Testing Verification and Validation. pp. 161–170. IEEE Computer Society, Washington, DC, USA (2009)
10. Jiang, M., Munawar, M.A., Reidemeister, T., Ward, P.A.S.: Detection and diagnosis of recurrent faults in software systems by invariant analysis. In: HASE '08: Proceedings of the 2008 11th IEEE High Assurance Systems Engineering Symposium. pp. 323–332. IEEE Computer Society, Washington, DC, USA (2008)
11. Jiang, Y., Cuki, B., Menzies, T., Bartlow, N.: Comparing design and code metrics for software quality prediction. In: PROMISE '08: Proceedings of the 4th international workshop on Predictor models in software engineering. pp. 11–18. ACM, New York, NY, USA (2008)
12. Kreimer, J.: Adaptive detection of design flaws. Electron. Notes Theor. Comput. Sci. 141(4), 117–136 (2005)
13. Moser, R., Pedrycz, W., Succi, G.: A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: ICSE '08: Proceedings of the 30th international conference on Software engineering. pp. 181–190. ACM, New York, NY, USA (2008)
14. Song, Q., Shepperd, M., Cartwright, M., Mair, C.: Software defect association mining and defect correction effort prediction. IEEE Trans. Softw. Eng. 32(2), 69–82 (2006)
15. Tribus, H.: Static Code Features for a Machine Learning based Inspection An approach for C. Master's thesis, School of Engineering, Blekinge Institute of Technology, SE371 79 Karlskrona, Sweden (Jun 2010), computer Science Thesis no: MSE-2010-16
16. Turhan, B., Kutlubay, O.: Mining software data. In: ICDEW '07: Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering Workshop. pp. 912–916. IEEE Computer Society, Washington, DC, USA (2007)