Detecting Defects with an Interactive Code Review Tool Based on Visualisation and Machine Learning

Stefan Axelsson  
Blekinge Institute of Technology  
stefan.axelsson@bth.se

Dejan Baca  
dejan.baca@bth.se

Robert Feldt  
robert.feldt@bth.se

Darius Sidlauskas  
dsidlauskas@gmail.com

Denis Kacan  
denis.kacan@gmail.com

Abstract

Code review is often suggested as a means of improving code quality. Since humans are poor at repetitive tasks, some form of tool support is valuable. To that end we developed a prototype tool to illustrate the novel idea of applying machine learning (based on Normalised Compression Distance) to the problem of static analysis of source code. Since this tool learns by example, it is trivially programmer adaptable. As machine learning algorithms are notoriously difficult to understand operationally (they are opaque) we applied information visualisation to the results of the learner. In order to validate the approach we applied the prototype to source code from the open-source project Samba and from an industrial, telecom software system. Our results showed that the tool did indeed correctly find and classify problematic sections of code based on training examples.

1. Introduction

An important part of ensuring code quality is code review [11]. Code review, which in effect is a form of manual static analysis of the code, is important especially when it comes to finding code that hides problems that are more difficult to find using some form of dynamic analysis, for example relating to non-functional requirements such as security. However, code review is problematic in that human operators are especially fallible when faced with a repetitive monotonous task, such as going through reams and reams of source code [12]. Some form of automatic static code analysis is then suggested. Static analysis have many inherent 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 runnable and test cases do not have to be developed.

To support manual code review we have developed a prototype tool—called The Code Distance Visualiser—to help the operator find problematic sections of code. The prototype is based on the novel idea of applying a machine learning technique, using Normalised Compression Distance (NCD) calculations, to provide the operator with an interactive, supervised self learning static analysis tool. This also makes the tool trivially programmer adaptable, that is, the tool will adapt to the task at hand as a consequence of the applied training. This is important in that traditional tools that are programmer adaptable (such as Coverity [8]) seldom are applied in that capacity [7].

Another advantage of machine learning is its capacity for generalising from a set of examples. Correctly applied machine learning has the capacity to surprise the operator by producing results that were previously unanticipated, while still being relevant and correct. However, machine learners are notoriously difficult to understand operationally, that is, they are opaque; it is difficult to understand when they are operating optimally and why they produce the results they do. To combat this problem we have applied information visualisation to the learner.

In the remainder, section 2 present related work, while sections 3 and 4 present Normalized Compression Distance and how we parse and represent the source code in order to apply it. In section 5 our tool and its visualisation capability are presented briefly. Section 6 present the experiments we have conducted with the tool and the results we have obtained. In section 7 we discuss the results. Finally, section 8 points to future work and concludes.

2. Related work

The previous work that is closest overall to the work presented here is probably that of Brun and Ernst [3]. They implemented a tool that is trained using machine learning techniques to identify program properties which indicate
errors. There are two primary differences between our approach and their work. First, they use dynamic analysis to extract semantic properties of the program’s computation, whereas we use static analysis. Second, their tool uses a classical batch-learning approach, in which a fixed quantity of manually labelled training data is collected at the start of the learning process. In contrast, the focus of our work is on incremental learning by (potentially a series of) manual user interactions.

Statistical machine learning techniques (Markov modelling, bootstrapping) were successfully applied on classifying the program’s behaviour by Bowring, et. al. [2]. There the classifier was trained previously to map execution statistics such as branch profiles to a label of program behaviour such as pass or fail.

So far we have not found any static code analyser proper similar to ours. The one most related is probably the one by Kremenek, et. al. [14], where a feedback rank scheme is used. Correlation among reports (errors reported by analyser) is represented in a probabilistic model Bayesian network. Then network is trained during interactive inspection of reports and probabilities for uninspected reports are recalculated. This approach primarily attacks the false positives problem in static code analysis and learning techniques are applied just to perform error ranking. In comparison, we use machine learning explicitly for finding errors and apply it directly to static analysis.

The successful applications of normalised information distance to various problem domains are too numerous to detail here, but it has been previously proposed for software quality-related tasks [9]. However, probably the closest application of normalised information distance to the one proposed here is in plagiarism detection within student programming assignments. The Software Integrity Diagnosis (SID) system [5] uses a variant of the normalized information distance (NID) to measure the similarity between two source code files. The plagiarism detector parses the source code much as our approach does. However, the SID system does not provide any interactivity for the user. The process also acts as a black box. This is not surprising, as the aims of their technique and ours are completely different. Indeed, keeping the analysis opaque might be done on purpose to avoid exposing inner system’s state to the cheaters.

3. Normalised Compression Distance (NCD)

The problem is one of supervised machine learning, i.e. the operator will select sections of code to train the machine learner with. Of the several available algorithms we have chosen to base our machine learner on a fairly recent algorithm that computes distances between arbitrary data vectors: Normalised Compression Distance (NCD) [6] as it is generally applicable [10], parameter free [13], noise resistant [4] and demonstrated theoretically optimal [17]. The NCD is an approximation to the uncomputable Normalised Information Distance, that is based on the notion of Kolmogorov Complexity.

NCD is based on the idea that by using a compression algorithm on data vectors (in whatever shape or form these may be) both individually and together, we will receive a measure of how distant they are. The better the combination of the two vectors compress, compared to how the individual vectors compress on their own (normalised to remove differences in length between the set of all vectors), the more similar they are. More formally, NCD is a metric:

\[
NCD(x, y) = \frac{C(x, y) - \min(C(x), C(y))}{\max(C(x), C(y))}
\]

where \(C(x)\) is the compressed length of \(x\) and \(C(x, y)\) the compressed length of \(x\) concatenated with \(y\).

In order to apply this metric as a supervised learner one selects features of the input data to train on and then calculates the distances from instances in the two (or more) sets of the selected features. Our tool contains two possible training sets; one bad containing undesirable features, and one good containing desirable features. The tool then calculates distance from all bad and all good features and presents the results to the operator. The classification is by the closest example in either set, e.g. a code feature that is closest in distance to one particular instance in the bad set is classified as bad, and vice versa, i.e. the machine learning algorithm proper is \(k\)-nearest-neighbour with \(k = 1\).

4. Parsing

The machine learning algorithm could not be successfully applied to the source code as is, as usual, feature vectors had to be selected [15]. We run the code under study through a parser to produce our features. In the prototype we have chosen the C-language, as parsers (and code with which to validate our approach) are readily available. However, besides the particulars of parsing and feature selection we see no reason that our approach would not be applicable to other languages, even those dissimilar to C. For the prototype we have chosen the freely available parser Sparse.¹

The task of how to do feature selection of source code is not at all well studied. Based on our—admittedly limited—experience we have chosen to first parse the code and then emit an adapted textual representation of the source code that is fed back to the machine learning step. Based on our experiences we’ve chosen the following strategies for parsing and presenting the source code.

In order to make the source code amenable to machine learning, all whitespace and a few other “uninteresting” reserved words (such as semi colons, angle

¹http://www.kernel.org/pub/software/devel/sparse/
brackets etc.) are removed. A stickier problem with a textual representation, stemming from dealing with a machine learner that can’t differentiate between different data types, is that longer strings will fool the learner into thinking they carry more information compared to shorter strings. That means that for example the_very_long_identifier_that_never_ends, e.g. used as a variable in the source code, will carry relatively more weight than a shorter identifier, say foo. The same is true of reserved words. In order to alleviate this problem the parser can be set to exchange all variable names with a two character abbreviation (the same two character substitution is maintained for the same variable as far as is possible).

A third problem, opposite of the previous, is that certain C operators are too short and too similar to each other to be distinct enough in a pure textual representation. Examples are == and =, which are known from experience to be difficult to tell apart. In order to remedy this problem these operators are exchanged with unique textual representations that are longer and more different from each other to give the machine learner more to latch on to.

Since our chosen algorithm does not naturally handle sequences of varying length we have to address the unit-of-analysis problem. One approach is to slide a window of a certain length over the features under study, but this tends towards a combinatorial explosion that we can ill afford, and it also performed poorly in preliminary tests with short but reasonable window sizes. Instead we have chosen to implement a varying level of detail that the user can choose; the statement (basically a line of source code) and basic block level (code between curly braces). The basic block can contain a while/if etc. statement introducing the block.

5. The Code Distance Visualizer

We will now describe how the preceding pieces were put together to form a tool that enables a user to select sections of source code to train the detector on and view the results of the training on source code (classifying). As described in the previous section, the naive NCD classification classifies code fragments into faulty or correct. However, when a training process is being presented as a black box, it is difficult to ascertain how the classifier is being trained. In order to be able to judge the quality of the output, the training process has to be transparent. Once the user has visual access to the internal state of the classifier, she can more precisely understand what the learner is actually learning and then interactively guide it by marking additional code fragments on screen as faulty or correct. Afterwards, all code fragments must be classified and visually marked. Unfortunately a lack of space precludes us from going into more detail on how this is done more exactly, save to say that the selected code fragments are heat-mapped [16] whereby the code fragment is coloured from red via yellow to green. The spectrum of colour depends on the NCD value of a particular code fragment and whether it is more likely to be faulty (red colour) or correct (green colour). This approach is inspired by that of Axelsson [1].

In order to present the ideas implemented in the prototype we present the user interface in figure 1. It can be divided into a few major parts: the original source view, the adapted source view, the list of training instances, the ranking view and the code fragment information view.

Figure 1. The main window of the prototype, no training.

A code fragment can be composed of structures such as a code block, individual statements, expressions, and mixture of the above. The user has a wide range of choices available for selecting code fragments of various lengths. For instance, in figure 1 one can see how the corresponding individual statements are selected in both original (if (integer = 5)) and adapted views (IF(qe:::1)).

After having trained the analyser on a particular code fragment the user may call up a list of the top ranked faulty or correct code fragments as classified by the machine learner. The user has a choice of which types of code fragments (individual statements, code blocks and expression blocks) that will be ranked.

The status of each code fragment depends on the distance to the closest training instance. Thus, in order to understand why a particular code fragment was classified as being faulty or correct, the user has to see how it is related to all the training instances.

The tool is available under the GPL on sourceforge.
6. Experimental Validation

A serious problem for researchers wishing to experiment with software validation techniques is the problem of locating suitable experimental samples. We even need multiple versions of the same software (erroneous and error-free). Obtaining such samples is nontrivial. To empirically evaluate the proposed technique, we applied our prototype to experimental samples from two real-world sources. Our goal for the experiments with our prototype was to obtain information about the effectiveness of the proposed technique for fault detection. The question of how effective the visualisation approach was in conveying information to the users that they would otherwise not have, will be left for further study.

We analysed a number of open source projects and chose the open source project Samba, at the time stable release Samba 3.0.28a, as that had the most accessible defect database. In addition a closed source commercial telecommunications software was selected. Both are server software that are written in the C language and each is in the million lines of source code range. For each project a subset of the source code was selected and analysed. Approxi-mately 20000 lines of code was used during the analysis, 13000 from Samba and 7000 from the commercial product. The commercial product handles large quantities of network data and performs extensive computation on that data. Due to corporate policy the product must unfortunately remain unnamed, as must the company. The decision of which freely available project to include was not rigorous in the sense that we made our choice depending on first impressions, thus other software may also be suitable. The telecom software was chosen because we already had access to and are very familiar with it. By using software from two different domains we aim to improve the general applicability of the results of the experiments.

The experiment was set up as a proof of principle; we applied the Code Distance Visualiser on code fragments that were analysed in advance, fragments where we knew what the answer already should be. We thus trawled the Samba bug database for defect data. When good experimental data was available, the prototype was trained by feeding it some known faulty code fragments and their corrected versions. We then checked whether the prototype correctly identified the remaining faults. Since an interactive tool with feedback was tested, several possible strategies for (re)training presented themselves. From our experience, a good strategy was to start training the prototype with one faulty code fragment and its corrected version, as it then learned the precise differences between faulty and correct code. We call this two-instance training initial training as it was the first training performed in the experiments. Also the ranking feature was used to improve the analysis process. For example, in Samba’s code (tagged as SAMBA-3-0-RELEASE) a memory leak bug was removed in nine places (revision 21755 of file net_rpc.c). Thus, we retrieved this and previous revisions, trained the prototype with one of the faulty code blocks and its corrected version (with the memory leak removed). Then we used the ranking feature to see the top nine (as we knew there were nine) faulty code blocks and checked whether all the remaining memory leaks were present among them. If some of them were missing, because of some false positives being present in the top ranking, we continued the training. Additional training involved marking the false positive at the highest position in the ranked list as correct. Usually this made the top ranked and similar false positives disappear from the ranking, freeing up their positions for other code fragments. From the two products that were examined, four distinctly different types of code defects were identified and used during the investigation. They were:

String overflows Defects where during various string operations a target char array overflowed. These defects very often lead to a security vulnerability. The most common string overflow is created by using insecure library functions such as strcpy(). An often used solution is to perform input validation or using a more secure function (such as strncpy()).

Null pointer references These are defects were a pointer that might be NULL were used without prior checking. These defects result in segmentation faults or other un-
predictable behaviours. The main cause is lack of error checking or using already freed memory pointers.

**Memory leaks** These result from dynamically allocated memory that is not freed. Dynamically allocated memory needs to be de-allocated before its pointers goes out of scope. These defects are often introduced by carelessness but can also arise from special cases when the program returns unexpectedly from a function.

**Incorrect API usage** A fault that stems from libraries and functions that are used correctly on the face of it but do not perform the task the programmer intended. These coding defects often occur many times in the code and a simple textual search could detect them, but instances of correct usage would drown the true positives in the search results.

Since we chose to investigate the same types of flaws in both products we will present the aggregated results. Where the discussion refers to only one code base that will be noted. Table 1 summarise the data from the investigation. The first column states what fault type is examined. The total column shows how many known faults of that type exist in the data. The CI are correctly identified faults while FN are false negatives; faults that were not detected by the tool. Correctly identified and false negative faults are shown in two separate columns, after initial training and after additional training as described in the previous section. The improvement column lists the relative improvement in detection rate the tool displayed after additional training compared to the initial training.

<table>
<thead>
<tr>
<th>Name</th>
<th>Total</th>
<th>Init Train</th>
<th>Addtl</th>
<th>Imprvmt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CI</td>
<td>FN</td>
<td>CI</td>
<td>FN</td>
</tr>
<tr>
<td>Str ovfl</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Null p ref</td>
<td>44</td>
<td>22</td>
<td>22</td>
<td>15</td>
</tr>
<tr>
<td>Mem-leak</td>
<td>12</td>
<td>7</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>Inc API</td>
<td>14</td>
<td>9</td>
<td>5</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 1. Experimental results

The string overflows are string copy operations that in some cases also create a security vulnerability. The known string overflow bugs had been reported by both testers and the static code analyser, Coverity Prevent.2

All the Null pointer reference bugs were originally reported from a static code analyser, either KlocWork3 or Coverity. While they had a bug report none was from a tester that had experienced a crash (this relates to both tested products). These defects often propagate and turn up later during execution as a segmentations fault and are difficult for the tester to ascribe a particular section of code.

There were two types of memory leaks of different complexity in the projects. The simpler memory leaks were instances the programmer forgot to free or in this case call shutdown to free allocated memory.

The incorrect API usage involved four commits with fixes. It was a correction to the caller unistr2_to_ascii, as the maxlen parameter should be set to the size of the destination, not to the size of the source string. The entire procedure took less than 10 minutes.

7. Discussion

Our results show that the tool can be used to effectively detect security defects in real-world source code. Even the initial training on a single faulty and correct example lead to the correct identification of 53.8% of the 78 identified defects. More extensive training, utilising the ranking feature of our tool, lead to the correct identification of 76.9% of the total number of defects. Detected defects are not simple in the way that a traditional find search in the source code would easily identify the patterns in the defects. For example, in the incorrect API case, 216 different instances of the API call was present but only in 14 of them is the API used incorrectly. With our tool all 14 of these defects were detected in only 10 minutes and there were no false negatives.

We focused on following a consistent training strategy for our results to be comparable across faults and projects studied. We found that keeping a balance between the examples of faulty and correct instances worked best. This relates to the machine learning concept of overfitting where the learner might pick up on some very specific feature and become over trained in one category.

The choice of unit-of-analysis (how large a code fragment we train on) affects the size and scope of the defects we can detect. Defects that depend on an interplay of dependent parts of the code might be hard to detect unless its individual parts are unique enough that they can be detected in isolation. But at the same time, isolated defects without the surrounding code do not present enough unique data to be detected, as was shown by the NULL pointer references defects. We could possibly find alternative ways to extract features that could extend the reach of our proposed method.

While we have not formally verified the usability of the prototype, it has had two users who were not part of the development of the tool. They both spontaneously reported that the visualisation and interactivity were both worthwhile additions and made working with the automatic classifier more pleasant and less opaque, even though there is still room for improvement in that respect.

---

2http://www.coverity.com
3http://www.klocwork.com/
8. Conclusion and Future work

We have only really scratched the surface of the possibilities of this technique. Obvious future work includes: study of how to extract features useful for machine learning from a static representation of source code, as the present approach is rather naive and no real extraction of semantic features takes place, investigating the applicability of different machine learning algorithms to static analysis, investigating the effect the visualisation has on the usability of the tool, to name just a few.

Moving further from the area of static analysis; by modifying the approach we have presented here, the approach could possibly be applied at even earlier stages of software development projects. Potentially it could be used on design, requirements or specification documents to highlight decisions that might lead to security or other software quality challenges.

In conclusion: We have developed a prototype tool that applies the novel idea of machine learning to aid in code review. The tool uses information visualisation techniques to try to alleviate the problem of machine learning techniques being opaque to the user, i.e. it is difficult to ascertain why they perform a particular classification in the manner they do. While much work still remains to be done in this domain, initial results were promising. When we applied the prototype to two different code bases, the tool successfully managed to identify problematic sections based on the examples given, without producing too many false alarms. Furthermore, during the experiment the tool managed to generalise from the example of the faulty strlen operation so that it detected that a similar strcat fault was also erroneous. The ability to generalise from given examples are a prime reason to apply machine learning to a problem. The tool also managed to identify a situation where features were missing from the code, which is an important class of faults.

References


