# Code Mining

Gerard J. Holzmann

Machine learning techniques are starting to achieve some impressive feats. In part the successes are due to the availability of astonishingly large amounts of data that can be used as training materials. Show a suitably equipped system a billion images of cats, and a billion images of land mowers, and see if it can try to figure out what the common patterns are. If this succeeds we can then show the same system a picture of our own cat or lawn mower, and hopefully it can tell which is which. Of course, if you show the same system an image of a house, it wouldn't have much of a clue beyond saying that it's neither a cat nor a land mower.  Basically, what we're leveraging is statistics, not intelligence, but nonetheless the results can be impressive.

Could we somehow use the same type of approach in software analysis, to find likely bugs? Just superficially you can think of the experiment where we show some "suitably equipped" system a billion examples of good programs, and ask it to see if it can deduce common patterns.  Then we show it a billion examples of programs known to be buggy, and again spot some common characteristics among these. Next, we show the now learned system our own code and ask it to classify it as either good or bad.  Finding the input sets would not be very difficult. There are large enough databases available of reasonably "good" programs. I'll conveniently sidestep the landmine here of trying to give examples, but we can likely all think of a few that could fit the bill. We also have substantial repositories of examples of bad code, if only the fragments of code that are collected in the ever-growing data base of common weakness enumerations, or CWEs, that you can mine at https://cwe.mitre.org.

I think you'll agree that this approach is unlikely to work. I'll try to explain why I think so, although I would be delighted if someone could prove me wrong.

*Figure 1*, Picture of a smiling cat that with some luck a machine learning system could be trained to recognize as such. This one was drawn by my daughter Tessa when she was 10, with some deviously added orange splotches that she must have known could complicate recognition.

## Mining Cats

How do we know, or how could our learned system know, that we're looking at a picture of a cat like that one in Figure 1? Clearly there are clues in the picture. The outline of the cat is one such clue. Perhaps also the fact that most cats have two eyes and pointy ears and so on: both we and our systems can learn to recognize those patterns. The patterns in the image can of course be deduced with relatively simple image processing techniques, until we get what is basically a fingerprint or signature of each image. I'm not a machine learning expert, but I can imagine that some clever hashing techniques could be used to group similar images, which can then be used to get a confidence rating of the closeness of a match. Figure 1 should then score a lot better against our fingerprints of cat images than against those of lawn-mowers.

But, how would that work if we wanted to distinguish good from bad software? Program code is rife with patterns that are obvious to a human reader, but are they also easy to discover by a learning algorithm? A few examples can illustrate the degree of difficulty.

A human may realize, for instance, that we don't see many appearances of the word "goto" in well-written code, but also that whenever we do see it, it is in most cases followed by a name, and that same name appears somewhere else in the same code followed by a colon. That name, finally will almost never be followed by an equals sign in the same fragment of code.  Of course, the name in question is quite arbitrary, which means that although the pattern is real enough, it might be very hard for an algorithm to discover it. And we should add that even if our algorithm would discover it and could detect deviations from the pattern in bad code, it's not too useful because every half-way decent compiler can do the same.

## Code as Text

A little more useful could be if our algorithm could deduce that a name followed by an open round brace that is preceded by a word different from "void" in "good" programs (e.g., in the definition of a function) when that same name appears somewhere else in the code , again followed by an open round brace, it will almost never be preceded by either a colon or a semi-colon. That is: the value that is returned by the function is effectively used by the caller in an expression or an assignment.

As you can see, trying to describe these code patterns as just textual coincidences leads to some tortured language, but the statistics should definitely bear out these types of correlations. Whether they are also discoverable by a machine learning algorithm without further guidance is another matter. I have my doubts. Unfortunately, it is generally easier to record patterns that are present, but much harder to record things that are rare or absent. An example of a well-known pattern that is absent from well-written English prose, for instance, is that a preposition is rarely followed by a period. Our teachers made sure that we all follow that rule, at least most of the time.

## Code Forensics

Although the patterns I mentioned so far are somewhat dubious, it is not too hard to think of correlations that could be worth mining.  For example, consider a for-statement in the C language. It doesn't take much imagination to note that if the condition part of the for-statement contains a < or <= operator, then the increment part of the statement will most likely contain an increment operator, such as ++ or +=. Similarly, if the condition part contains a > or >= operator, then the increment part will most likely contain a – or -= operator. Intuitively this is fairly obvious. We should be on our guard now, because sometimes things that seem obvious are actually not really true. So, let's make sure.

I don't have a machine learning algorithm handy that can give me statistics for large code bases, but it is easy enough to run some tests, say over a couple of million lines of source code from one of the Linux distributions. I'll use the 14.9 Million lines of code from the Linux 4.3 distribution for this check.

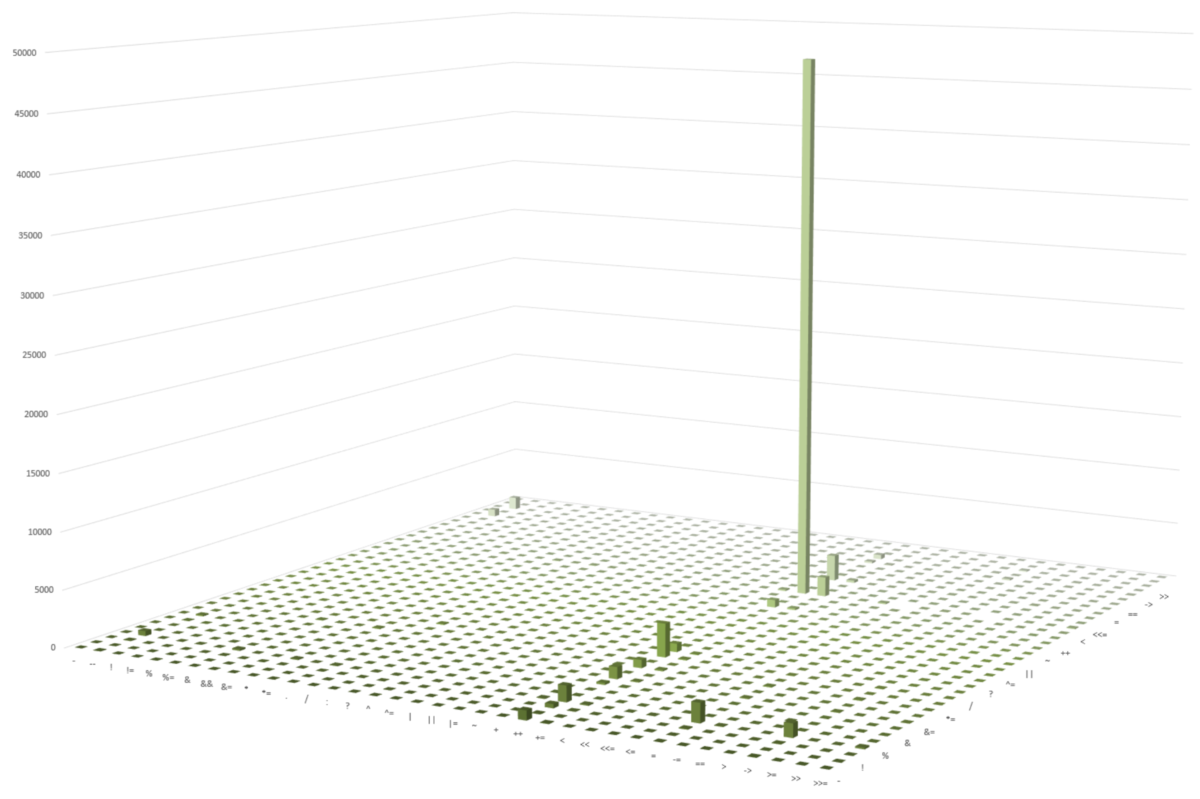

*Figure 2*— Correlation between operators used in the condition part of a for-loop and the operators used in the increment part of the same for-loop. The peak of 48,138 cases is for the combination of operator < in the condition part and operator ++ in the increment part of the same for-loop. The statistics are for 14.9 million lines of the Linux 4.3 distribution.

Figure 2 shows the correlation of operators used in the condition part of for-loops with operators used in the increment part of the same loop. So here we're ignoring everything else that might appear within the control part and just look at the operators. It's fair to say that there is indeed a correlation of the type we suspected, and a pretty strong one at that. The most common way to write a for-loop is to use a less-than operator in the condition and a ++ operator for the loop increment.

Table I – Some of the most frequent operator combinations used in for-loops

| Condition \ Increment | ++ | += | -- | -= |
|---|---:|---:|---:|---:|
| < | 48,138 | 1,726 | 26 | 7 |
| <= | 2,301 | 209 | 5 | 0 |
| > | 154 | 48 | 666 | 123 |
| >= | 35 | 20 | 1,157 | 98 |

Table I shows the most interesting combinations in a little more detail.

The more unexpected combinations, for instance the use of a < or <= operator and a decrement operator is quite rare, but curiously non-zero. Those cases, being the statistical outliers, could well warrant a little more scrutiny and may contain either mistakes or the use of obscure code that may be prone to misunderstanding. A good example of the former could be this line of code in file ./drivers/net/ethernet/qlogic/netxen/netxen_nic_hw.c, line 2335:

```
for (k = 0; k < read_cnt; k--) {
```

and an example of the latter could be this line of code in file mm/memtest.c, line 108:

```
for (i = memtest_pattern - 1; i < UINT_MAX; --i) {
```

The good thing is that once we know the statistics, the number of outliers is usually sufficiently small that we can inspect each one in detail and form an opinion about the quality of that code. A static analyzer could take advantage of these types of patterns, but the question is still how we could systematically discover them, with machine learning techniques or otherwise.

## Code Explorations

I haven't said yet how I generated the data that is shown in Figure 2 and Table I.  You can of course try to use simple text processing tools to extract the fragments of code that are of interest. For example, we can try to use the tool *grep* to find all occurrences of the keyword "for" and then filter out the operators in the control part that follows in round braces. Often, the complete control part of the for-loop appears on the same line as the initial keyword, but not always. There are enough exceptions to the one-line rule here that we may want to try another way.

A code browsing tool comes in handy here, especially if it allows us to specify a pattern of interest as a regular expression over lexical tokens, rather than plain characters. The tool I've used for my check is called Cobra (which you can find on http://spinroot.com/cobra), and it supports such a pattern matching language. So when we say, for instance:

```
$ cobra -pat "for ( .* )" *.c
```

we're not restricted to matches that appear on a single line, since the match is on the token sequence which is insensitive to white space.

Unfortunately, this still doesn't give us the statistics we need. We will have to post-process the code to extract the two groups of operators. And even though the input to the pattern matcher above isn't sensitive to white space, the output (the matching lines) will still contain all those line breaks, so in post-processing we must still deal with that.

I found it easier to do the entire process with a sequence of code querying commands, issued interactively in a code querying session over all the 14.9 million lines of code. I used those commands to home in on the various parts of each for-loop, and write it out in two groups of operators per line, one for the operators used in the condition part and the other for operators used in the increment part. A few small post-processing steps then gave me the data for Figure 2. But we can do more.

For example, the following is a sequence of Cobra commands that I used to find a relatively small set of deviations from the general rule that we can find in the code base. The pound sign # is a comment delimiter of the query command language:

```
mark for (           # mark all keywords "for" followed by a round brace
next                 # move the mark forward to the token following "for"
contains /^<=?$      # keep only those marks where the range of tokens that is
                     # enclosed in the round braces contains an < or <= operator
contains /^-[-=]$    # keep only those marks where the range of tokens also
                     # contains at least one decrement operator
contains no ++       # and no increment operators
= "nr matches:"      # report the number of matches
display              # and display them
```

The first command in this set marks all places in the code where we find the keyword "for" followed by a open round brace. Remember that white space doesn't matter since we're matching on lexical tokens here. The matches will point at the locations in the code where the "for" itself appears. With the next command we now move those match points forward to the open round brace. That open round brace is part of a pair that the tool knows about, and the range of tokens in between the matching open and closing round brace can be interrogated as a set. So, the third command checks that set and narrows down the selections to only those sets that contain tokens matching the regular expression that is shown. The regular expression matches only on token texts that start with < and may contain a subsequent = character, and nothing else. Now we have all for-loops that contain either a < or a <= operator. Next, we check if those same for-loops contain a decrement operator, which is either a – or a -= token, again with a regular expression. We refine the set of matches a little more by next also saying that the same clause does not also contain the increment operator ++, and we then display the small set of remaining matches.

The whole sequence can be abbreviated by using single-letter short-hands for each query command, and we can pass the entire command sequence in a command-line invocation of the tool, rather than typing them in one at a time in an interactive session, but you get the idea. Of course this whole process of querying the code base would be of little use if it took more than a few seconds, so speed is important here.

## Fast Approximations

We can get the tool to work fast by leveraging the fact that the matching algorithm is simple and therefore trivial to parallelize. It can in principle be executed independently on each separate file, so given enough CPUs, the 21,987 .c files in the Linux distribution we looked at could be scanned in a fraction of a second. On my own 32-core system the processing takes about 20 seconds, which includes about 9 seconds for preparing the lexical token stream itself. I should also add that this last code mining method that I used here is not precise. For example, I did not bother distinguishing between the three different parts of the for-loop. The reason I could do so is that the final set of matches was small enough, even for a code base of this size, that I could afford to overshoot the final set by a small amount, and then peruse the matches for the little gems that I was really looking for.

This brings to mind a quote from the late John Tukey, a Bell Labs mathematician who in the late forties and fifties worked with both John von Neumann and Claude Shannon. Tukey wrote in1962:

> "The most important maxim for data analysis to heed, and one which many statisticians seem to have shunned, is this: 'Far better an approximate answer to the right question, which is often vague, than an exact answer to the wrong question, which can always be made precise.'" [T62, p.13]

The observation was meant as a critique of the cookie-cutter approach that some of Tukey's colleagues were using to perform data analysis at the time: spending little or no time in carefully checking if the underlying assumptions for those methods were actually valid in each case at hand. Perhaps the same can be said about our use of static source code analysis techniques today. Although most tools do allow the user to customize the properties checked for to a given code base, these capabilities are rarely used.

There are several obstacles to the routine use of custom properties in the most commonly used static source code analyzers. A first is the difficulty of specifying new types of queries. One has to be very determined to learn the internal details of the tool, which are of course different for each one. Then the average query resolution time for new types of static analysis can be substantial if the code base grows to a few million lines. The long waits discourage experimentation and targeted approximation to home in on cases of interest.

## Casting Doubt

Query approximations that can be executed fast can have real value. Consider how we may try to find out how often different types of pointer arithmetic is used in a large code base. If you're brave enough to try to express this in one of the customization languages of the leading static analysis tools you are a better person than me. Here's how I would do it with a Cobra query that can both be written and executed in a few seconds:

```
$ cobra -pat "\* ) ( .* + .* )" *.c
```

The pattern I used tries to match type casts, enclosed in round braces, that end with a star operator, indicating that whatever follows is cast to a pointer. Next, the pattern checks that the expression in round braces that follows contains a plus operator, indicating that an addition is being performed. The syntax for the pattern expression that we used here is a simplified version of a regular expression. The initial star is escaped, to avoid that it is interpreted as a meta-symbol. The dot matches any token text, and the combination .* is used to indicate an arbitrary sequence of tokens. Cobra makes sure that the round braces in the pattern match, which forces the text in between the braces to refer to a single expression. In an interactive session with the Linux source code, matches to this query pattern are found in about a second on a multi-core system. If you're curious, there are about 6,000 such matches in the 14.9 million lines of Linux code. Here's the first one, found in file mm/slub.c, on line 247:

```
return *(void **)(object + s->offset);
```

It is of course not much harder to now home in on more egregious types of pointer arithmetic that use a sequence of additions and subtractions, and perhaps even multiplications. Once we have the initial set of matches, those refinements are simple to add interactively by querying the range of tokens contained in the braced expression, similar to what we did before when querying for-loops. If we do so we are rewarded with about 70 matches. The following cast, which contains an addition, a subtraction, and a multiplication, is found in file drivers/video/console/fbcon.c on lines 2697 to 2698 (note that the pattern spans multiple lines):

```
return (u16 *) (vc->vc_origin + offset –
     softback_lines * vc->vc_size_row);
```

Both the ease with which we can phrase these queries, and the speed with which they can be resolved makes a difference. So far though, it's most helpful to those old-fashioned types of learning systems: us.

## References

[T62] John W. Tukey, The Future of Data Analysis, *The Annals of Mathematical Statistics*, Vol. 33, No. 1 (Mar., 1962), pp. 1-67.