On the “Naturalness” of Buggy Code

Ray, Baishakhi; Hellendoorn, Vincent; Godhane, Saheel; Tu, Zhaopeng; Bacchelli, Alberto; Devanbu, Premkumar

DOI
10.1145/2884781.2884848

Publication date
2016

Document Version
Accepted author manuscript

Published in

Citation (APA)

Important note
To cite this publication, please use the final published version (if applicable). Please check the document version above.
On the “Naturalness” of Buggy Code

Baishakhi Ray‡, Zhaopeng Tu†, Vincent Hellendoorn†*, Alberto Bacchelli†, Saheel Godhane‡, Premkumar Devanbu‡

ABSTRACT

Real software, the kind working programmers produce by the kLOC to solve real-world problems, tends to be “natural”, like speech or natural language; it tends to be highly repetitive and predictable. Researchers have captured this naturalness of software through statistical models and used them to good effect in suggestion engines, porting tools, coding standards checkers, and idiom miners. This suggests that code that appears improbable, or surprising, to a good statistical language model is “unnatural” in some sense, and thus possibly suspicious. In this paper, we investigate this hypothesis. We consider a large corpus of bug fix commits (ca. 7,139), from 10 different Java projects, and focus on its language statistics, evaluating the naturalness of buggy code and the corresponding fixes. We find that code with bugs tends to be more entropic (i.e. unnatural), becoming less so as bugs are fixed. Ordering files for inspection by their average entropy yields cost-effectiveness scores comparable to popular defect prediction methods. At a finer granularity, focusing on highly entropic lines is similar in cost-effectiveness to some well-known static bug finders (PMD, FindBugs) and ordering warnings from these bug finders using an entropy measure improves the cost-effectiveness of inspecting code implicated in warnings. This suggests that entropy may be a valid, simple way to complement the effectiveness of PMD or FindBugs, and that search-based bug-fixing methods may benefit from using entropy both for fault-localization and searching for fixes.

1. INTRODUCTION

Our work begins with the observation by Hindle et al [22], that “natural” code in repositories is highly repetitive, and that this repetition can be usefully captured by language models originally developed in the field of statistical natural language processing (NLP). Following this work, language models have been used to good effect in code suggestion [22, 48, 53, 15], cross-language porting [38, 37, 39, 24], coding standards [2], idiom mining [3], and code de-obfuscation [47]. Since language models are useful in these tasks, they are capturing some property of how code is supposed to be. This raises an interesting question: What does it mean when a code fragment is considered improbable by these models?

Language models assign higher naturalness to code (tokens, syntactic forms, etc.) frequently encountered during training, and lower naturalness to code rarely or never seen. In fact, prior work [7] showed that syntactically incorrect code is flagged as improbable by language models. However, by restricting ourselves to code that occurs in repositories, we still encounter unnatural, yet syntactically correct code; why? We hypothesize that unnatural code is more likely to be wrong, thus, language models actually help zero-in on potentially defective code.

This notion appears plausible; highly experienced programmers can often intuitively zero-in on “funny-looking” code, when trying to diagnose a failure. If statistical language models could capture this capability, then they could be a useful adjunct in a variety of settings: they could improve defect prediction; help provide an improved priority ordering for static analysis warnings; improve the performance of fault-localization algorithms; or even recommend “more natural” code to replace buggy code.

To investigate this phenomenon, we consider a large corpus of 7,139 bug fix commits from 10 different projects and focus on its language statistics, evaluating the naturalness of defective code and whether fixes increase naturalness. Language models can rate probabilities of linguistic events at any granularity, even at the level of characters. We focus on line-level defect analysis, giving far finer granularity of prediction than typical statistical defect prediction methods, which most often operate at the granularity of files or modules. In fact, this approach is more commensurate with static analysis or static bug-finding tools, which also indicate potential bugs at line-level. For this reason, we also investigate our language model approach in contrast and in conjunction with two well-known static bug finders (namely, PMD [10] and FindBugs [14]).

Overall, our results corroborate our initial hypothesis that code with bugs tends to be more unnatural. In particular, the main findings of this paper are:

1. Buggy code is rated as significantly more “unnatural” (improbable) by language models.
2. This unnaturalness drops significantly when buggy code is replaced by fix code.
3. Furthermore, we find that above effects are substantially stronger when:
   • the buggy code fragment is shorter (fewer lines), and
   • the bug is “short-lived”, viz. more quickly fixed.
4. Using cost-sensitive measures, inspecting “unnatural” code indicated by language models works quite well: Performance is comparable to that of static bug finders FindBugs and PMD.
5. Ordering warnings produced by the FindBugs and PMD tools, using the “unnaturalness” of associated code, significantly improves the performance of these tools.

Our experiments are mostly done with Java projects, but we have strong empirical evidence indicating that the first two findings above generalize to C as well; we hope to confirm the rest in future work.

2. BACKGROUND

Our main goal is evaluating the degree to which defective code appears “unnatural” to language models and the extent to which this can enable programmers to zero-in on bugs during inspections. Furthermore, if language models can help pinpoint buggy lines, we want to identify how their performance and applicability relate to commonly used fault-detection methods. To this end, we explore the application of language models, first to file-level defect prediction (comparing with statistical defect prediction methods) and then to line-level defect prediction, comparing their performance with popular Static Bug Finders (SBF). In this section, we present the relevant technical background and the main research questions.

2.1 Language Modeling

Language models assign a probability to every sequence of words. Given a code token sequence $S = t_1 t_2 \ldots t_N$, a language model estimates the probability of this sequence occurring as a product of a series of conditional probabilities for each token’s occurrence:

$$ P(S) = P(t_1) \cdot \prod_{i=2}^{N} P(t_i|t_1, \ldots, t_{i−1}) $$

$P(t_i|t_1, \ldots, t_{i−1})$ denotes the chance that the token $t_i$ follows the previous tokens, the prefix, $h = t_1, \ldots, t_{i−1}$. The probabilities are impractical to estimate, due to the huge number of possible prefixes. A common fix is the n-gram language model, using the Markov assumption to condition just on the preceding $n−1$ tokens.

$$ P_{\text{n-gram}}(t_i|h) = P(t_i|t_{i−n+1}, \ldots, t_{i−1}) $$

This we estimate from the training corpus as the fraction of times that $t_i$ follows the prefix $t_{i−n+1}, \ldots, t_{i−1}$. This is reversible: we can also compute each token given its suffix (the subsequent tokens). We compute entropies based on both prefix and suffix token sequences to better identify the buggy lines (Section 3.3).

The n-gram language models can effectively capture the regularities in source code and have been applied to code suggestion tasks [22, 2]. Tu et al. [53] improved such language models by considering that software tends to be repetitive in a local context. They introduced a cache language model ($S\text{gram}$) that deploys an additional cache—list of ngrams extracted from the local context, to capture the local regularities. The ngrams extracted from each file under test form its local context in the cache model. We use the state of the art $S\text{gram}$ to judge the “improbability” (measured as cross-entropy) of lines of code.

2.2 Line-level defect detection: SBF

Static Bug-Finders (SBF) use syntactic and semantic properties of source code to locate common errors, such as null pointer dereferencing and buffer overflows. They rely on methods ranging from informal heuristic pattern-matching to formal algorithms with proven properties; they typically report warnings at build time. Most of the pattern-matching tools [10, 14, 13, 9] require users to specify the buggy templates. Others [51, 32] can automatically infer rules by mining existing software; they raise warnings if violations of the rules occur. Most (e.g., PMD and FindBugs) are unsound, yet fast and widely used, compared to more formal approaches. Generally, SBF produce false positives and false negatives, which reduce their cost-effectiveness [23, 26].

Both SBF and our model (fairly imperfectly) indicate potential defect locations; our goal is to compare these approaches and see whether they can be combined.

2.3 Evaluating Defect Predictions

We take the simplified view that SBF and $S\text{gram}$ are comparable, that in they both select suspicious lines of code for manual review. We therefore refer to language model based bug prediction as NBF (“Natural Bug Finder”). With either SBF or NBF, human code review effort, spent on lines identified as bug-prone, will hopefully find some defects. Comparing the two approaches requires a performance measure. We adopt a cost-based measure that has become standard: AUCEC (Area Under the Cost-Effectiveness Curve) [4]. Like ROC, AUCEC is a non-parametric measure that does not depend on the defects’ distribution. AUCEC assumes that cost is inspection effort and payoff is the number of bugs found.

Given a model (SBF or NBF) that predicts buggy lines, we rank all the lines in decreasing order of their defect-proneness score. Thus, the best possible model would place the buggiest lines at the top of the list. This approach helps reviewers to inspect a smaller portion of code (i.e., cost), while finding a disproportionately larger fraction of defects (i.e., payoff). We normalize both cost and payoff to 100% and visualize the improvement that a prediction model provides when compared against a random guess using a ‘lift-chart’ [55]. In this chart, cost (on the x-axis) refers to the percentage of the code-base inspected at prediction time, and payoff (on the y-axis) indicates the portion of the known bugs (discovered by data gathering) already in the code, that are covered by warned lines. AUCEC is the area under this curve. Under uniform bug distribution across SLOC, inspecting $x\%$ of lines of code at random will, in expectation, also yield $x\%$ of the bugs, i.e., random selection produces a diagonal line on the lift chart. The corresponding AUCEC when inspecting 5% of lines at random is 0.00125.

Inspecting 100% of SLOC in a project is probably unrealistic. Prior research has assumed that 5% (sometimes 20%) of the code could realistically be inspected under deadline [44]. Additionally, Rahman et al. compare SBF with DP (a file level statistical defect predictor) by allowing the number warnings from SBF to set the inspection budget (denoted AUCECL) [43]. They assign the DP the same budget and compare the resulting AUCEC scores. We extend this approach to our comparison of SBF and NBF.

To understand how NBF’s payoff varies with cost, we first measure its performance for both 5% and 20% inspection budget. We then compare AUCECs of NBF and SBF at both the 5% budget and under AUCECL budget.

Finally, we investigate defect prediction performance under several credit criteria. A prediction model is awarded credit, ranging from 0 to 1, for each (ipso facto eventually) buggy line flagged as suspicious. Previous work by Rahman et al. has compared SBF and DP models using two types of credit: full (or optimistic) and partial (or scaled) credit [43], which we adapt to line level defect prediction. The former metric awards a model one credit point for each bug iff at least one line of the bug was marked buggy by the model. Thus, it assumes that a programmer will spot a bug as soon as one of its lines is identified as such. Partial credit is more conservative: for each bug, the credit is awarded in proportion to the fraction of the bug’s defective lines that the model marked. Hence, $1$ Calculated as $0.5 \times 0.05 \times 0.05$. This could be normalized differently, but we consistently use this measurement, so our comparisons work.
partial credit assumes that the probability of a developer finding a bug is proportional to the portion of the bug that is marked by the model. It should be noted the AUCEC is non-parametric under partial credit, but not under full credit, as it depends on the defect distribution; however we get the same overall result under both regimes.

2.4 Research Questions

Our central question is whether "unnaturalness" (measured as entropy, or improbability) indicates poor code quality. The abundant history of changes (including bug fixes) in OSS projects allows the use of standard methods [50] to find code that was implicated in bug fixes ("buggy code").

**RQ1. Are buggy lines less “natural” than non-buggy lines?**

Project histories contain many bug fixes, where buggy code is modified to correct defects. Do language models rate bug-fix code as more natural than the buggy code that was replaced (i.e., was bug fix code rated more probable than the buggy code)? Such a finding would also have implications for automatic, search-based bug repair: if fixes tend to be more probable, then a good language model might provide an effective organizing principle for the search, or (if the model is generative) even generate possible candidate repairs.

**RQ2. Are buggy lines less “natural” than bug-fix lines?**

Even if defective lines are indeed more often rated improbable by language models, this may be an unreliable indicator; there may many be false positives (correct lines marked improbable) and false negatives (buggy lines indicated as natural). Therefore, we investigate whether unnaturalness (i.e. entropy) can provide a good organizing principle for directing inspection.

**RQ3. Is “naturalness” a good way to direct inspection effort?**

One can view ordering lines of code for inspection by ‘naturalness’ as a sort of defect-prediction technique; we are inspecting lines in a certain order, because prior experience suggests that certain code is very improbable and thus possibly defective. Traditional defect-prediction techniques typically rely on historical process data (e.g., number of authors, previous changes or bugs); however, defectiveness is predicted at the granularity of files (or methods). Thus, we may reasonably compare unnaturalness as an ordering principle with SBF, which provide warnings at the line level.

**RQ4. How do SBF and NBF compare in terms of ability to direct inspection effort?**

Finally, if SBF provides a warning on a line and it appears unnatural to a language model, we may expect that this line is even more likely a mistake. Therefore we investigate whether unnaturalness is a good ordering for warnings provided by static bug-finders.

**RQ5. Is “naturalness” a useful way to focus the inspection effort on warnings produced by SBF?**

### 3. Methodology

We now describe the projects we studied, and how we gathered and analyzed the data.

#### 3.1 Study Subject

We studied 10 OSS Java projects, as shown in Table 1: among these are five projects from Github, while the others are from the Apache Software Foundation. We chose the projects from different domains to measure NBF’s performance in various types of systems. All projects are under active development.

We analyzed NBF’s performance in two settings: Phase-I considers NBF’s ability to find bugs, based on continuous usage during active development (see Section 3.2). We chose to analyze each project for the period of one-year which contained the most bug-fixes in that project’s history; here, we considered both development-time and post-release bugs. Then, for the chosen one-year duration, we extracted snapshots at 1-month intervals. A snapshot captures the state of the project at a given point in time. Thus, for each project we studied 12 snapshots, in total analyzing 120 snapshots across 10 projects, including 113,762 distinct file versions, and 35.3 Million total non-commented source code lines (NCSL). Overall, we studied 7,139 distinct bug-fix commits comprising of 2.2 Million total buggy lines. Subsequently, we confirmed our results across the entire history of each studied project, using snapshots at 6-month intervals (157 snapshots, 63,301 commits, 23,664 bug-fixes). Due to page limitations, we are presenting results from our study of 1-month snapshots only.

Next, for Phase-II (see Section 3.2), we focused only on post-release bugs to evaluate NBF’s performance as a release-time bug prediction tool. We used the data set from Rahman et al. [43], in which snapshots of the five Apache projects were taken at...
lected project releases. The project snapshot sizes vary between 68 and 630K NCSL. The bugs were extracted from Apache’s JIRA issue tracking system; the bug count per release, across all the projects, varies from 24-194 (see Table 2). We further used warnings produced by two static bug finding tools: FINDBUGS [5] and PMD [10], as collected by Rahman et al.. PMD operates on source code and produces line-level warnings; FINDBUGS operates on Java bytecode and reports warnings at line, method, and class level.

### 3.2 Data Collection

#### Phase-I

Here we describe how we identified the buggy lines in a snapshot corresponding to the bugs that developers fixed in an ongoing development process.

**Estimating bug-fixing commits.** Development time bug fixes are often not recorded in an issue database. Thus, to estimate bug fixing activities during an ongoing development process, we analyzed commit messages associated with each commit for the entire project evolution and looked for error related keywords. First, we converted each commit message to a bag-of-words and then stemmed the bag-of-words using standard natural language processing (NLP) techniques. Then similar to Mockus et al. [33], we marked a commit as a bug-fix, if the corresponding stemmed bag-of-words contains at least one of the error related keywords: ‘error’, ‘bug’, ‘fix’, ‘issue’, ‘mistake’, ‘incorrect’, ‘fault’, ‘defect’, ‘flaw’, and ‘type’. This method was adapted from our previous work [46]. For the Apache projects, as well as Atmosphere and Netty we further improved the classification with information available from the JIRA issue database.

To evaluate the accuracy of the above classification, we manually verified the result for 30 commits (30 from each project, chosen randomly). Here, we only evaluated whether the author of a presumed bug-fix commit really marked their commit as a bug-fix. Out of these 300 commits, 288 were classified correctly (96%), 3 commits (1%) were described as a potential “issue” (thus may have developed into a bug later) and 9 commits (3%) were classified incorrectly—5 false negatives and 4 false positives. Thus, our approach achieves 96% accuracy (95% conf.int.: 93.8% to 98.2%).

**Selecting snapshots.** To evaluate NBF in continuous active development, ideally we need to study all the commits made to a project for its full history. But using git blame to get line-level bug data at this scale is not feasible. Thus, we chose 1-year evaluation periods for each project. Since our focus is studying bugs, we considered the one-year period of each project that contained the most bug fixes. Within these periods, we looked at monthly snapshots thereby simulating near-continuous usage of our tool. For instance, snapshots were taken at 1-month intervals between 2006-07-06 and 2007-07-06 for project Derby (see Table 1).

**Identifying buggy lines in a snapshot.** This part consists of three steps: (1) identifying lines related to a bug, (2) identifying commits that introduced the bug, and (3) mapping the buggy lines to the snapshots of interest.

In step (1), we assumed that all the lines deleted (or changed) in a bug-fix commit were buggy lines. To find these lines, we looked at the versions of a file before and after a bug-fix. We used `git diff` to identify the deleted/changed lines in the old version and marked them as ‘buggy’; the added/changed lines in the new version are marked as ‘fixed’. Next, in step (2), we used `git blame` to locate the commits that had introduced these buggy lines in the system. The first two steps are analogous to the SZZ algorithm [50]. Once we know where the buggy lines originated, we used `git blame --reverse` option to locate these lines in the snapshots of interest. This step ‘maps’ the buggy lines to specific snapshots. Figure 1 explains the procedure where commit c4 is a bug-fix commit. The corresponding buggy lines (marked red in the old version) are found to originate from two earlier commits c1 and c2. We then map the buggy lines from c1 to both S1 and S2, whereas the buggy lines from c2 are mapped only to S2.

Note that we considered all the bugs that appeared at any time in the entire evolution and map them back to the snapshots of interest. However, we lose the buggy lines that were fixed before, or arose after, our study period as we cannot map these to any snapshots of interest. We also miss some transient bugs that appeared and were fixed within a snapshot interval (thus lasting less than a month). At the end of this step, we know exactly which lines in each of our snapshots are buggy (and were fixed in some future commit) and which ones are benign, modulo time-window censoring effects.

#### Phase-II

In Phase-II, we studied post-release bugs for the Apache projects, using Rahman et al. [43]’s dataset. Rahman et al. selected a number of release versions of each Apache project and, for each release, identified post-release bug-fix commits from the JIRA issue tracking system. They then identified buggy and non-buggy lines for each release version similar to steps 2 and 3 of the previous section.

### 3.3 Entropy Measurement

**Choice of language model.** We measured entropy using Tu et al.’s cache-based language model (Sgram) tool [53], as described in Section 2.1. We computed the entropy over each lexical token in all the Java files of all the snapshots. For a given file in a snapshot, the tool estimates a language model on all the other files of the same snapshot. It then builds a cache by running the language model on the given file, computing the entropy of each token based on both prolog (preceding tokens) and epilog (succeeding tokens). Finally, based on the training set and locally built cache, the tool computes the entropy of each token of the file; the line and file entropies are computed by averaging over all the tokens belong to a line and all lines corresponding to a file respectively.

To generate entropies of the fixed lines, we leveraged the data-set gathered for the entire evolution period with 6 months interval, as mentioned in Section 3.1. This was necessary because a bug may get fixed after our studied period. For each bug-fix commit, we trained the dataset on its immediate preceding snapshot and tested it on the new file version corresponding to the bug-fix.

**Determining parameters for cache language model.** Several factors of locality can affect the performance of the cache language model: cache context, cache scope, cache size, and cache order. In this work, we built the cache on the entire file under investigation.
In this light, we only needed to tune cache order (i.e. maximum and minimum order of ngrams stored in the cache). In general, longer ngrams are more reliable but quite rare, thus we backed off to shorter matching prefixes (or suffixes) [25] when needed. We followed Tu et al. [53] to set the maximum order of cache ngrams to 10. To determine the minimum back-off order, we performed experiments on the Elasticsearch and Netty projects looking for optimal performance measured in terms of entropy difference between buggy and non-buggy lines. The maximum difference was found at a minimum back-off order of 4 with no change in the backoff weight. Thus, we set the minimum back-off order to 4, and the backoff weight to 1.0.

Adjusting entropy scores. Language models could work for defect prediction at line granularity if bug-prone lines are more entropic. For instance, a non-buggy but high-entropy line would be a false positive and worsen the language model’s performance at the prediction task. For example, lines with previously unseen identifiers, such as package, class and method declarations, have substantially higher entropy scores on average. Vice versa, for-loop statements and catch clauses – being often repetitive – have much lower entropy scores. Such inter-type entropy differences do not necessarily reflect their true bug-proneness. In fact, for-statements, though less entropic, are often more bug-prone than the more entropic import-declarations.

This observation led us to using abstract-syntax-based line-types and computing a syntax-sensitive entropy score. First, we used Eclipse’s JDT\(^2\) to parse an Abstract Syntax Tree (AST) of all files under consideration. Any line in a Java file is always contained by either a single AST node (e.g., Compilation Unit root-node) or several AST nodes in hierarchical order, e.g., a nested line with if-statement, method-declaration, and class-declaration. For each line, its syntax-type is the grammatical entity associated with the lowest AST node encompassing the full line. Examples include statements (e.g., if, for, while, return), declarations (e.g., variable, structure, method, import) or other AST nodes that tend to span one line, such as switch cases and annotations. We then computed how much a line’s entropy deviated from the mean entropy of its line-type using normalized Z-score:

\[
\text{Line}_\text{type} = \frac{\text{Entropy}_\text{line} - \mu_{\text{type}}}{\sigma_{\text{type}}},
\]

where \(\mu_{\text{type}}\) denotes mean $gram entropy of all the lines of a given type, and \(\sigma_{\text{type}}\) denotes standard deviation. This gave us a syntax-sensitive entropy model $gram+type.

The above normalization essentially uses the extent to which a line is “unnatural” w.r.t. other lines of the same type. In addition, based on the fact that all line-types are not equally buggy, we computed relative bug-proneness of a type based on the fraction of bugs and total lines (LOC) it had in all previous snapshots. Here we used the previous snapshots as training set and computed bug-weight of a line-type as:

\[
w_{\text{type}} = \frac{\text{bugtype}/\text{LOCtype}}{\sum_{\text{type}} \text{bugtype}}/\text{LOC},
\]

where the bugs and LOCs per type were counted over all previous snapshots. We then scaled the z-score of each line by its weight \(w\) to achieve our final model, which we name $gram+wType.

Phase-I and Phase-II data set and the entropy generation tool are available at http://odd-code.github.io/.

4. EVALUATION

This section discusses the answers to Research Questions introduced in Section 2.4. The First two RQs are primarily based on the Phase-I data set. RQ3 uses Phase-II data to evaluate N$\beta$F’s capability as a File level defect predictor. Line level defect prediction is evaluated using both data sets (RQ3, RQ4, and RQ5). We begin with the question that is at the core of this paper:

\(^2\)Java development tools, http://www.eclipse.org/jdt/

<table>
<thead>
<tr>
<th>Bug-fix threshold</th>
<th>Entropy diff (bug &gt; non-bug) Cohen’s d effect</th>
<th>Entropy diff (bug &gt; fixed) Cohen’s d effect</th>
<th>% Unique bugs detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.95 to 2.00 0.61</td>
<td>1.58 to 1.67 0.60</td>
<td>14.92</td>
</tr>
<tr>
<td>2</td>
<td>1.68 to 1.72 0.53</td>
<td>1.43 to 1.51 0.52</td>
<td>23.93</td>
</tr>
<tr>
<td>4</td>
<td>1.37 to 1.40 0.43</td>
<td>1.16 to 1.23 0.41</td>
<td>34.65</td>
</tr>
<tr>
<td>7</td>
<td>1.15 to 1.17 0.36</td>
<td>0.98 to 1.04 0.34</td>
<td>43.94</td>
</tr>
<tr>
<td>15</td>
<td>0.91 to 0.93 0.29</td>
<td>0.75 to 0.81 0.26</td>
<td>55.91</td>
</tr>
<tr>
<td>20</td>
<td>0.81 to 0.83 0.25</td>
<td>0.62 to 0.68 0.21</td>
<td>60.16</td>
</tr>
<tr>
<td>50</td>
<td>0.58 to 0.59 0.18</td>
<td>0.39 to 0.44 0.13</td>
<td>71.68</td>
</tr>
<tr>
<td>100</td>
<td>0.55 to 0.57 0.12</td>
<td>0.36 to 0.41 0.12</td>
<td>80.96</td>
</tr>
<tr>
<td>Overall</td>
<td>0.86 to 0.87 0.26</td>
<td>0.69 to 0.74 0.19</td>
<td>100.00</td>
</tr>
</tbody>
</table>

Buggy lines have higher entropy than non-buggy lines. Buggy lines also have higher entropy than fixed lines. Entropy difference decreases as bug-fix threshold increases. Entropy differences are measured with t-test for 95% confidence interval and shows statistical significance (p-value \(\leq 0.05\)). Cohen’s d effect size: 0.2 = ‘small’, 0.5 = ‘medium’, and 0.8 = ‘large’.

RQ1. Are buggy lines less “natural” than non-buggy lines?

To evaluate this question, we compare entropies of buggy and non-buggy lines for all the studied projects. A Wilcoxon non-parametric test confirms that buggy lines are indeed more entropic than non-buggy lines with statistical significance (\(p < 2.2 \times 10^{-16}\)). Average entropy of buggy lines is 6.21 while that of non-buggy lines is 5.34. However, Cohen’s D effect size between the two is 0.26 (see last row in Table 3), which is considered small.

One explanation for the small effect size across bugs of all sizes is an impact of tangled bug fixes—lines that are changed in a bug-fix commit but are not directly related to the bug. Herzig et al. [21] showed that around 17% of all source files are incorrectly associated with bugs due to such tangled changes. The impact of tangled changes on bug entropy is more visible for larger bug-fix commits. Some of the lines changed in a larger bug fix may not be directly associated with the erroneous lines and thus may not be “unnatural”. In contrast, for smaller bug-fix commits, say for 1 or 2 lines of fix, the fixed lines (i.e. the lines deleted from the older version) are most likely to be buggy.

\(^7\)75% of all changes (both bug-fixes and feature implementations) in our data set contain no more than 7 lines of deletion

![Figure 2](image-url)

To understand the effect of tangled changes on the naturalness of buggy code, we compute entropy difference between buggy and non-buggy lines at various bug-fix size thresholds. We define bug-fix threshold as the number of lines in a file that are deleted from the older version of a bug-fix commit. Table 3 shows the result. Both entropy difference and effect size decrease as the bug-fix threshold increases. For example, for a threshold size of 1, entropies of buggy lines are on average 1.95 to 2.00 bits higher than non-buggy lines (95% confidence interval). Cohen’s D effect size between the two lies between medium and large (0.61). 14.92% of the bug-fixes lie below this threshold. For a threshold size of 7, buggy lines have 1.15 to 1.17 bits higher entropy than their non-buggy counterpart.

In this case, we see a small to medium effect (effect size = 0.36).

Table 3: Entropy Difference and Effect Size Between buggy vs. non-buggy and buggy vs. fixed lines
This is also shown in Figure 2(a). At bug-fix threshold 100, we see only 0.55 to 0.57 entropy difference with small effect (0.17). These results indicate that the lines that are indirectly associated with real buggy lines in tangled bug-fix commits may have lower entropy, and thus would diminish the overall entropy of buggy lines.

We further observe that bugs that stay longer in a repository tend to have lower entropy than the short-lived bugs. **Bug duration** of a buggy line is measured as the number of months until a buggy line is fixed starting from the day of its introduction (bug_duration = bug-fix date minus bug-introduction date). The following table shows the summary of bug duration in our data set (in months):

<table>
<thead>
<tr>
<th>Min.</th>
<th>1st Qu.</th>
<th>Median</th>
<th>Mean</th>
<th>3rd Qu.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.03</td>
<td>8.33</td>
<td>23.40</td>
<td>29.03</td>
<td>39.83</td>
<td>125.6</td>
</tr>
</tbody>
</table>

Based on bug duration, we divide all the buggy lines into three groups: low, medium, and high. The bugs in low group “survived” for less than 9 months (1st quartile); bugs in medium group survived from 9 to 24 months (1st quartile to median), and the remaining bugs are in the high group. Figure 2(b) shows their entropy variation. The low group has significantly higher entropy than the medium and high group, with Cohen’s d effect size of 0.62 and 0.75 respectively (medium to large effect size). The difference is also confirmed with Wilcoxon non-parametric test with statistical significance. The medium group is also slightly more entropic than the high group with statistical significance, although the effect size is very small (0.10). These results indicate that the bugs that are fixed more quickly are more “unnatural” than the longer-lived bugs. We hope to explore the reasons in future work: perhaps the highly entropic bugs are easier to locate, diagnose and fix, and thus get speedily resolved—or perhaps (more intriguingly) highly-entropic code is more strongly associated with failures that are more likely to be quickly encountered by users.

In summary, we have the overall result:

**Result 1:** Buggy lines, on average, have higher entropies, i.e. are “less natural”, than non-buggy lines.

A natural question is whether the entropy of the lines in a bug drops once the bug is fixed. This leads us to the following question:

**RQ2. Are buggy lines less “natural” than bug-fix lines?**

To answer RQ2, we collected bug-fix commit patches of all the bugs that exist in any snapshot under study. In a bug-fix commit, the lines deleted from the original version are considered buggy lines and lines added in the fixed versions are considered fixed lines. We collected all such buggy and fixed lines for all the projects, as described in Section 3.2. Establishing a one-to-one correspondence between a buggy and fixed line is hard because buggy lines are often fixed by a different number of new lines. Hence, we compare the mean entropies between buggy and fixed lines across all the patches. Wilcoxon non-parametric test confirms that entropy of buggy lines, in general, drops after the bug-fixes with statistical significance (see Figure 2(a)).

Similar to RQ1, tangled changes may also impact the entropies of bugs and their fixes. To measure the impact, we further compare the entropy differences between buggy and fixed lines at various bug-fix thresholds. Table 3 shows the result. Both the entropy difference and effect size decreases as bug-fix threshold increases. For example, at a bug-fix threshold of one line, average entropy drops, upon fixing, between 1.58 to 1.67 bits (95% confidence interval and with statistical significance). The Cohen d’s effect size is medium to large (0.60). However, with threshold size at 30, mean entropy difference between the two are slightly more than half a bit with a small effect size of 0.18. Such behavior suggests that tangled changes may be diluting the entropy of buggy code and their fixes.

Bug duration also impacts the drop of entropy after bug fix. For bugs with low duration, the entropy drop is significant: 2.68 to 2.75 bits on average (effect size: 0.50). For medium duration bugs, entropy drops from 0.09 to 0.18 bits (effect size: 0.04), while entropy does not necessarily drop for high duration bugs.

Table 4 shows three examples of code where entropy of buggy lines dropped significantly after bug-fixes. In the first example, a bug was introduced in Facebook-Android-SDK code due to a wrong initialization value—tokenInfo was incorrectly reset to null. This specific initialization rarely occurred elsewhere, so the buggy line had a rather high entropy of 6.07. Once the bug was fixed, the fixed line followed a repetitive pattern (indeed, with two prior instances in the same file). Hence, entropy of the fixed line dropped to 1.95, an overall 4.12 bit reduction. The second example shows an example of incorrect method call in the Netty project. Instead of calling the method trySuccess(used three times earlier in the same file), the code incorrectly called the method setSuccess, which was never called in a similar context. After the fix, entropy drops by 4.63 bits. Finally, example 3 shows an instance of missing conditional check in Lucene. The developer should check whether directory creation is successful by checking return value of directory.mkdir() call, following the usual code pattern. The absence of this check raised the entropy of the buggy line to 9.21. The entropy value drops to 5.34 after the fix.

In certain cases these observations do not hold. For instance, in example 4 of Table 5, entropy increased after the bug fix by 5.75

---

**Table 4:** Examples of bug fix commits that \textit{\&BF} detected successfully. These bugs evinced a large entropy drop after the fix. Bugs with only one defective line are shown for simplicity purpose. The errors are marked in \textit{red}, and the fixes are highlighted in \textit{green}.

**Example 1:** Wrong Initialization Value
Facebook-Android-SDK (2012-11-20)
File: Session.java
Entropy dropped after bugfix : 4.12028
```java
if (newState.isClosed()) {
    // Before (entropy = 6.07042):
    this.tokenInfo = null;
    // After (entropy = 1.95014):
    this.tokenInfo = AccessToken.createEmptyToken
                  (Collections.<String>emptyList());
```

**Example 2:** Wrong Method Call
Netty (2013-08-20)
File: ThreadPerChannelEventLoopGroup.java
Entropy dropped after bugfix : 4.6257
```java
if (isTerminated()) {
    // Before (entropy = 5.96485):
    terminationFuture.setSuccess(null);
    // After (entropy = 1.33915):
    terminationFuture.trySuccess(null);
```

**Example 3:** Unhandled Exception
Lucene (2002-03-15)
File: FSDirectory.java
Entropy dropped after bugfix : 3.87426
```java
if (!directory.exists())
    // Before (entropy = 9.213675):
    directory.mkdir();
    // After (entropy = 5.33941):
    if (!directory.mkdir())
        throw new IOException
            ("Cannot create directory: " +
             directory);
```

---
We rank each file in each release based on entropy and proneness score is assigned to every file under test. We repeat this for our model on one release and evaluate on the next release: a defect-better predictors of file level defects [42]. For each project, we train from [42, 11], such as #developers, #file-commit, code churn, and where the response is a binary variable indicating whether a file is buggy or not. The predictor variables are the process metrics from [42, 11], such as #developers, #file-commit, code churn, and previous bug history; prior research shows that process metrics are better predictors of file level defects [42]. For each project, we train our model on one release and evaluate on the next release: a defect-proneness score is assigned to every file under test. We repeat this procedure for all releases for all the projects under study.

A file’s entropy is measured as the average entropy of all the lines in that file. We rank each file in each release based on entropy and

### Example 4: Wrong Argument \(NBF\) could not detect
Netty (2010-08-26)
File: MessageDecoder.java
```
if (maxHeaderSize <= 0) {
    throw new IllegalArgumentException()
    // Before (entropy = 2.696275):
    - "maxHeaderSize must be a positive integer: "+ maxChunkSize);  
    // After (entropy = 8.447305):
    + "maxHeaderSize must be a positive integer: "+ maxChunkSize);
```

#### Example 5: \(NBF\) detected incorrectly
Facebook-Android-SDK (multiple snapshots)
File: Request.java
```
// Entropy = 9.892635
Logger logger = new Logger(LoggingBehaviors.
REQUESTS, "Request");
```

bits. In this case, developer copied maxChunkSize from a different context but forgot to update the variable name. This is a classic example of copy-paste error [45]. Since, the statement related to maxChunkSize was already present in the existing corpus, the line was not surprising. Hence, its entropy was low although it was a bug. When the new corrected statement with maxHeaderSize was introduced, it increased the entropy. Similarly, in Example 5, the statement related to logger was newly introduced in the corpus. Hence, its entropy was higher despite not being a bug.

However, for all bug-fix thresholds, Wilcoxon non-parametric test confirms with statistical significance that the entropy of buggy lines is higher than the entropy of fixed lines. Overall, we see 0.69 to 0.74 bit entropy drops after bug fixes with a small effect size of 0.19. Thus, in summary:

**Result 2:** Entropy of the buggy lines drops after bug-fixes, with statistical significance.

Having established that buggy lines are significantly less natural than non-buggy lines, we investigate whether entropy can be used to direct inspection effort towards buggy code. We start with the following research question:

**RQ3. Is “naturalness” a good way to direct inspection effort?**

**Baseline: detecting Buggy Files.** We first consider file-level defect prediction (DP), the de facto standard in the literature. Specifically, we evaluate whether ordering files by entropy will better guide us to identifying buggy files than traditional logistic regression and random forest based DP.

DP is typically used at release-time to predict post-release bugs [35, 57, 42, 36, 11]; so, for this comparison we use the post release bug data collected in Phase-II. DP is implemented using two classifiers: logistic regression (LR) [43, 42] and Random Forest (RF), where the response is a binary variable indicating whether a file is buggy or not. The predictor variables are the process metrics from [42, 11], such as #developers, #file-commit, code churn, and previous bug history; prior research shows that process metrics are better predictors of file level defects [42]. For each project, we train our model on one release and evaluate on the next release: a defect-proneness score is assigned to every file under test. We repeat this procedure for all releases for all the projects under study.

A file’s entropy is measured as the average entropy of all the lines in that file. We rank each file in each release based on entropy and

logistic regression-based prediction score. Figure 3 shows the normalized AUCEC performance of all the classifiers, similar to [4]. Here, the y-axis shows the AUCEC scores as a fraction of the “perfect” score (files ranked using an oracle) for each model. At the higher inspection budget of 20% SLOC, the logistic regression and Random Forest DP models perform 56.5% and 30.4% better than the entropy-based model respectively. However, at the stricter inspection budget of 5% of SLOC, the entropy-based predictor performs 30% better than LR, and only 4.2% worse than RF, all are measured w.r.t. entropy-based predictor.

**Detecting Buggy Lines.** Having shown that entropy can help detect bug-prone files, we now focus on a finer granularity: can the entropy of a line of code be used to direct inspection effort towards buggy lines? Specifically, will ordering lines by entropy will guide inspection effort better than ordering lines at random? In all our experiments, the random baseline chooses lines at random from Non-Commented Source Lines (NCSL), picking just as many as NBF and SBF (in RQs 4&5). For the reasons outlined in Section 2.3, we evaluate the performance of entropy-ordering, with the AUCEC scores at 5% and 20% of inspected lines (AUCEC\(_{5/20}\) in short) according to two types of credit: partial and full (in decreasing order of strictness). Since this is a line-level experiment, comparing AUCEC values here with file-level optimum, as we did earlier in Figure 3, risks confusion arising from ecological inference [41]; so we just present the raw AUCEC scores, without normalization.

We further calculate AUCEC for different bug-fix thresholds, as entropy is better at predicting smaller bug-fixes. When measuring AUCEC at a threshold of, say, n lines, we ignore bug-fixes spanning ≥ n lines. Performance is evaluated in terms of percentage gain of AUCEC over random AUCEC:

\[
gain = \frac{\sum_{project}(aucec - random\_aucec)}{\sum_{project} random\_aucec}
\]

Figure 4(a) shows AUCEC\(_{20}\) scores for partial credit, averaged over all projects for bug-fix threshold 7. Under partial credit, the default $\text{Sgram}$ model (without the syntax weighting described in §3.3) performs better than random, particularly at ≥ 10% of inspected lines. At 20% of inspected line, $\text{Sgram}$ performs 41.95% better than random. Figure 4(b) focuses on the performance on 10 studied projects, up to 5% of the inspected lines. At this level, $\text{Sgram}$’s performance varies. For projects Facebook, Netty, and Qpid $\text{Sgram}$ performs significantly better than random; but in other projects $\text{Sgram}$ either performs similar or worse than random.

On closer examination, we found that some program constructs are intrinsically more entropic than others. For example, method declarations are often more entropic, because they are less frequent. This observation led us to consider syntactic line-type in bug prediction, as discussed in Section 3.3. Scaling the entropy scores by line type improves AUCEC\(_{5}\) performance in all but Facebook

```
LL
```
and Atmosphere and significantly improves performance in all cases where $gram$ performed no better than random. Including the bugginess history of line-types ($gram + wType$) furthermore outperforms random and $gram$ in all but Lucene and Atmosphere and achieves an overall AUCEC$_5$ scores 92.53% higher than random at bug-fix threshold 7. These results are similar under full credit (see Table 6). Since $gram + wType$ is the best-performing “naturalness” approach so far, we hereafter refer to it as $\mathcal{NB}F$.

Table 6: Performance evaluation of $\mathcal{NB}F$ with random for different bug-fix threshold

<table>
<thead>
<tr>
<th>BugFix Threshold</th>
<th>Full Credit</th>
<th>Partial Credit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>auccecc</td>
<td>random</td>
</tr>
<tr>
<td>AUCEC$_5$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.0035</td>
<td>0.0016</td>
</tr>
<tr>
<td>4</td>
<td>0.0037</td>
<td>0.0019</td>
</tr>
<tr>
<td>7</td>
<td>0.0038</td>
<td>0.0023</td>
</tr>
<tr>
<td>14</td>
<td>0.0041</td>
<td>0.0028</td>
</tr>
<tr>
<td>all</td>
<td>0.0051</td>
<td>0.0051</td>
</tr>
<tr>
<td>AUCEC$_{20}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.0531</td>
<td>0.0252</td>
</tr>
<tr>
<td>4</td>
<td>0.0543</td>
<td>0.0300</td>
</tr>
<tr>
<td>7</td>
<td>0.0559</td>
<td>0.0345</td>
</tr>
<tr>
<td>14</td>
<td>0.0565</td>
<td>0.0401</td>
</tr>
<tr>
<td>all</td>
<td>0.0619</td>
<td>0.0567</td>
</tr>
</tbody>
</table>

Table 6 further shows that $\mathcal{NB}F$ performance worsens with larger bug-fix thresholds, for both AUCEC$_5$ and AUCEC$_{20}$. For example, for AUCEC$_5$, with bug-fix threshold 2, we see 122.74% and 113.46% performance gain over random AUCEC for full and partial credit respectively. These gains drop to 47.38% and 87.77% at a threshold of 14.

Notice that, in Table 6, under Partial Credit, the random selection approach yields constant AUCEC$_5$ and AUCEC$_{20}$ scores, independent of the bug fix threshold. Partial credit scoring assigns credit to each line based on the size of the bug-fix that it is part of (if any); thus, selecting 5% of lines at random should, in expectation, yield 5% of the overall credit that is available (see also Section 2.3). Full Credit, on the other hand, assigns the credit for detecting a bug as soon as a single line of the bug is found. Therefore, the AUCEC scores of a random selection method under full credit will depend on the underlying distribution of bugs: large bugs are detected with a high likelihood even when inspecting only a few lines at random, whereas small bugs are unlikely to be detected when inspecting 5% of lines without a good selection function. This is reflected in Table 6: as the bug-fix threshold increases, the random AUCEC scores increase as well. The $\mathcal{NB}F$ approach, on the other hand, exhibits a threshold of 14.

Figure 4: Performance Evaluation of $\mathcal{NB}F$ with Partial Credit.

(a) Overall AUCEC upto inspecting 20% lines for all the projects

(b) Closer look at low order AUCEC, upto inspecting 5% lines for individual project

RQ4. How do $\mathcal{SB}F$ and $\mathcal{NB}F$ compare in terms of ability to direct inspection effort?

To compare $\mathcal{SB}F$ with $\mathcal{NB}F$, we use $gram + wType$ model on Phase-II data set. To investigate the impact of tangled changes, we choose the overall data set and a bug-fix threshold of 14 (roughly corresponds to the fourth quartile of bug-fix sizes on this dataset). Further, we select PMD and FindBugs from a pool of available $\mathcal{SB}F$ tools, because they are popular and have been studied in previous research [43, 23, 26].

As discussed in Section 2.3, Rahman et al. developed a measure named AUCECL to compare $\mathcal{SB}F$ and DP methods on an equal footing [43]. In this method, the $\mathcal{SB}F$ under investigation sets the line budget based on the number of warnings it returns and the DP method may choose a (roughly) equal number of lines. The models’ performance can then be compared by computing the AUCEC scores both approaches achieve on the same budget. We follow this approach to compare $\mathcal{SB}F$ with $\mathcal{NB}F$.

Furthermore, we also compare the AUCEC$_5$ scores of the algorithms. For the $gram + wType$ model, this is analogous to the results in RQ3. To acquire AUCEC$_5$ scores for the $\mathcal{SB}F$, we simulate them as follows: First assign each line the value zero if it was not marked by the $\mathcal{SB}F$ and the value of the $\mathcal{SB}F$ priority otherwise ([1, 2] for FindBugs, [1 - 4] for PMD); then, add a small random amount (tie-breaker) from $U[0, 1]$ to all line-values and order the lines by descending value. This last step simulates the developer randomly choosing to investigate the lines returned by $\mathcal{SB}F$: first from those marked by the $\mathcal{SB}F$ in descending (native, $\mathcal{SB}F$ tool-based) priority, and within each priority level at random. We repeat the simulation multiple times and average the performance.

Figure 5(a) and 5(b) show the AUCEC$_5$ and AUCECL scores for PMD using partial credit and at bug-fix threshold 14. The results for FindBugs were comparable, as were the results using full credit. As can be seen, performance varied substantially between projects and between releases of the same project. Across all releases and under both AUCEC$_5$ and AUCECL scoring, all models performed significantly better than random (paired t-test: $p < 10^{-3}$), with
large effect (Cohen’s D > 1). SBF and NBF performed comparably; NBF performed slightly better when using both partial credit and the specified bug-fix threshold, but when dropping the threshold, and/or with full credit, no significant difference remains between NBF and SBF. No significant difference in performance was found between FINDBUGS and PMD either.

In all comparisons, all approaches retrieved relatively bug-prone lines by performing substantially better than random. In fact, at 5% inspection budget, both the line-level NBF and the two SBF performed substantially better than the earlier presented DP method and file-level NBF (compare Figure 5(a) and Figure 3).

**Result 4:** Entropy achieves comparable performance to commonly used Static Bug Finders in defect prediction.

Notably, NBF had both the highest mean and standard deviation of the tested models, whereas PMD’s performance was most robust. This suggests a combination of the models: we can order the warnings of the SBF using the Sgram+wType model. In particular, we found that the standard priority ordering of the SBF is already powerful, so we propose to re-order the lines within each priority category.

**RQ5.** Is “naturalness” a useful way to focus the inspection effort on warnings produced by SBF?

To answer this question, we again assigned values to each line based on the SBF priority as in RQ4. However, rather than add random tie-breakers, we rank the lines within each priority bin by the (deterministic) Sgram+wType score. The results for PMD are shown in Figure 5, first using the AUCEC$_5$ measure (S(a)) and then using the AUCECL measure (S(b)). PMD Mix refers to the combination model as proposed.

Overall, the combined model produced the highest mean performance in both categories. It significantly outperformed the two SBFs in all cases ($p < 0.01$) and performed similarly to the NBF model (significantly better on Lucene and QPID, significantly worse on Derby ($p < 0.05$), all with small effect). These results extended to the other evaluation methods, using full credit and/or removing the threshold for max bug-fix size. In all cases, the mix model was either significantly better or no worse than any of the other approaches when averaged over all the studied releases.

We further evaluated ranking all warnings produced by the SBF by entropy (ignoring the SBF priorities) and found comparable but slightly weaker results. These results suggest that both NBF and SBF contribute valuable information to the ordering of bug-prone lines and that their combination yields superior results.

**5. THREATS TO VALIDITY**

**Internal Validity.** A number of threats to the internal validity arise from the experimental setup. First, our identification of buggy lines could be wrong, as we used a simple key-word based search to identify buggy commits (see Section 3.2). To minimize this threat, we manually evaluated our bug classification tool and reported an overall accuracy of 96%.

Another source of false negatives is the presence of (yet) undetected bugs that linger in the code-base. Also, developers may “tangle” unrelated changes into one commit [20, 12]. However, given the high significance of entropy difference betweenbuggy and non-buggy lines, we consider it unlikely that these threats could invalidate our overall results. Furthermore, NBF’s positive performance on (higher quality, JIRA-based) Phase-II dataset confirms our expectations regarding the validity of these results.

A threat regarding RQ2 is the identification of ‘fixed’ lines, which replaced ‘buggy’ lines during a bugfix commit. The comparisons between these categories could be skewed, especially when the bugfix commits replace buggy lines with a larger number of fixed lines. In fact, small bug-fixes do indeed add more lines than they delete on average (the reverse holds for fixes spanning over 50 lines). However, a majority of bugs of any size were fixed with at most as many lines. In particular, more than two-third of one and two-line bugs, which demonstrated the greatest decrease in entropy were fixed with one and two lines respectively. Thus, these findings minimize the above threat. Other non-bugfixing changes (e.g., introduction of clone) may also show drop in entropy w.r.t. its previous version.

Our comparison of SBF and NBF assumes that indicated lines are equally informative to the inspector, which is not entirely fair; NBF just marks a line as “surprising”, whereas SBF provides specific warnings. On the other hand, we award credit to SBF whether or not the bug has anything to do with the warning on the same lines; indeed, earlier work [52] suggests that warnings are not often related to the buggy lines which they overlap. So this may not be a major threat to our RQ4 results.

Finally, the use of AUCEC to evaluate defect prediction has been criticized for ignoring the cost of false negatives [56]; the development of better, widely-accepted measures remains a topic of future research.

**External Validity.** External Validity concerns generalizability of our result. To minimize this threat, we use systems from different
domain from Github and Apache, having a substantial variation in age, size and ratio of bugs to overall lines (see table 1). We also confirmed our overall result by studying entire evolutionary history of all the projects, analyzed with 6 months snapshot interval.

Next, does this approach generalize to other languages? There is nothing language-specific about the implementation of n-gram and $gram$ models ($gram$+wType model, however, does require parsing, which depends on language grammar). Prior research showed that these models work well to capture regularities in Java, C, and Python [22, 53]. To investigate NBF’s performance for other languages, we performed a quick sanity check on 3 C/C++ projects: (Libuv, Bitcoin and Libgit), studying evolution from November 2008 - January 2014. The results are consistent with those presented in Table 3 and Figure 2—buggy lines are between 0.87 and 1.16 bits more entropic than non-buggy lines at bug-fix threshold 15 (slightly larger than in Table 3). Also, the entropy of buggy lines at this threshold drops by nearly one bit. These findings strongly suggest that our results generalize to C/C++; we are investigating the applicability to other languages.

Finally, even though we showed good empirical results with our approach, this does not assure that it actually helps developers in their bug finding efforts, as was shown in a similar scenario (with automated debugging techniques) by Parnin et al. [40]. To tackle this, the natural next step would be a controlled experiment with developers using our approach.

6. RELATED WORK

Statistical Defect Prediction. DP aims to predict defects yet to be detected by learning from historical data of reported bugs in issue databases (e.g., JIRA). This is a very active area (see [8] for a survey), with even a dedicated series of conferences (i.e., PROMISE [1]). D’Ambros et al. survey and provide a direct comparison of a number of representative DP approaches (including those using process metrics, such as previous changes [34, 17] and defects [27], and product metrics, such as code complexity metrics [6]). While earlier work evaluated models using IR measures such as precision, recall and F-score, more recently non-parametric methods such as AUC and AUCEC have gained in popularity. D’Ambros et al. follow this trend and conduct an evaluation similar to ours.

A DP may work beyond file granularity; Giger et al. presented a DP at the level of individual methods [16]. We are the first to predict defects at a line-level using only statistical models.

Static Bug Finders. SBF can work at fine granularity as opposed to DP, hence the comparison with our approach. The work closely related to ours is by Rahman et al. [43]; by comparing DP performance with SBF they reported that popular SBF tools like FindBugs and PMD do not necessarily perform better than DP. We also find that NBF can be used to rank SBF warnings, but we are not the first to tackle this challenge. Kremenek et al. use z-ranking and a cluster-based approach to prioritizing warnings based on the warnings’ previous success rate [29, 28]. Kim and Ernst mine information from code change history to estimate the importance of warnings [26]. Our approach ranks warnings based on properties of the source code rather than the output of the SBF or whether and how warnings have been fixed in history. Future research could evaluate how our approaches can complement the work above. Ruthruff et al. propose a filtering approach to detecting accurate and actionable SBF warnings [49]. They use priority of warnings, defined by the SBF type of error detected, and features of the affected file (e.g., size and warning depth) to do the filtering. Our approach ranks warnings on a different aspect of source code than those they consider and could be used to complement their model. Finally, Heckman et al. proposed Faultbench, a benchmark for comparison and evaluation of static analysis alert prioritization and classification techniques [18] and used it to validate the Aware [19] tool to prioritize static analysis tool warnings. Since results of our approach are promising, further research could investigate our approach against this additional benchmark.

The field of SBF has advanced rapidly, with many developments; researchers identify new categories of defects, and seek to invent methods to find these defects efficiently, either heuristically or through well-defined algorithms and abstractions. Since neither method is perfect, the actual effectiveness in practice is an empirical question. A comprehensive discussion of related work regarding SBF and their evaluation can be found in Rahman et al. [43].

Inferring rules and specifications. Statistical language models are employed to capture the repetitive properties of languages, in our case of programming languages, thus inferring the style or even some latent specification about how the language is supposed to be used in a specific project and context. As such, our work is related to previous research that tries to automatically infer specifications and use it to identify outliers as probable defects. Kermenek et al. present a framework based on a probabilistic model (namely, factor graph [31]) for automatically inferring specifications from programs and use it to find missing and incorrect properties in a specification used by a commercial static bug-finding tool [30]. In our case, we allow errors to be localized without language-specific tuning, without defining the set of annotations to infer, and without modeling domain-specific knowledge. Wasylykowski et al. mine usage models from code to detect anomalies and violations in methods invoked on objects and demonstrated that these can be used to detect software defects [54]. Similarly, Thummalapenta and Xie develop Alattin, an approach to mine patterns from APIs and detect violations [51]. In contrast, our model is less specialized and tries to highlight unexpected patterns at token level, without focusing on the specific case of method invocations.

7. CONCLUSION

The predictable nature (“naturalness”) of code suggests that code that is improbable (“unnatural”) might be wrong. We investigate this intuition by using entropy, as measured by statistical language models, as a way of measuring unnaturalness.

We find that unnatural code is more likely to be implicated in a bug-fix commit. We also find that buggy code tends to become more natural when repaired. We then turned to applying entropy scores to defect prediction and find that, when adjusted for syntactic variances in entropy and defect occurrence, our model is about as cost-effective as the commonly used static bug-finders PMD and FindBugs. Applying the (deterministic) ordering of entropy scores to the warnings produced by these static bug-finders produces the most cost-effective method. These findings suggest that entropy scores are a useful adjunct to defect prediction methods. The findings also suggest that certain kinds of automated search-based bug-repair methods might do well to have the search in some way influenced by language models.

In the near future, we plan to build extensions into PMD, FindBugs and other static bug finders that order warnings based on our $gram$+wType regime. Further ahead, we plan to study other applications of these approaches including dynamic fault-isolation methods and automated bug-patching tools.

Acknowledgment. This material is based upon work supported by the National Science Foundation under Grant No. 1414172.
8. REFERENCES


