On the "Naturalness" of Buggy Code

Baishakhi Ray[†*] Vincent Hellendoorn[‡*] Zhaopeng Tu[♮] Connie Nguyen[†]
Saheel Godhane[†] Alberto Bacchelli[‡] Premkumar Devanbu[†]

University of California, Davis
Davis, CA 95616, USA
{bairay,ptdevanbu,srgodhane,cmnguyen}@ucdavis.edu

Huawei Technologies Co. Ltd.
Sha Tin, Hong Kong, China
tuzhaopeng@gmail.com

Delft University of Technology
Delft, Netherlands
{V.J.Hellendoorn,A.Bacchelli}@tudelft.nl

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. 8,296), from 10 different Java projects, and we 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. 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 language-independent and 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

Communication is ordinary, everyday human behavior, something we do *naturally*. This "natural" linguistic behavior is characterized by efficiency and fluency, rather than creativity. Most natural language (NL) is both repetitive and predictable, thus enabling humans to communicate reliably & efficiently in potentially noisy and dangerous situations. This repetitive property, *i.e. naturalness*, of spoken and written NL has been exploited in the field of NLP: Statistical language models (from hereon: *language models*) have been employed to capture it, and then use to good effect in speech recognition, translation, spelling correction, *etc.*

As it turns out, so it is with code! People also write code using repetitive, predictable, forms: Recent work [18] showed that code is amenable to the same kinds of language modeling as NL, and language models have been used to good effect in code suggestion [18, 32, 36, 14], cross-language porting [24, 23, 25, 19], coding standards [2], idiom mining [3], and code de-obfuscation [31]. 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 [6] 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; in this paper, we explore this.

To this end, we consider a large corpus of 8,296 bug fix commits from 10 different projects, and we 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 here on line-level defect analysis, giving far finer granularity of prediction than traditional 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 [8] and FindBugs [12]).

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. 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.
4. 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.

*Baishakhi Ray and Vincent Hellendoorn are both first authors, and contributed equally to the work.

2. BACKGROUND

Our main goal is evaluating the degree to which defective code appears "unnatural" to language models, and to what extent language models can actually enable programmers to zero-in on bugs during inspections. Furthermore, if language models can actually help direct programmers towards buggy lines, we are interested to know how they compare against static bug-finding tools. In this section, we present relevant technical background and the main research questions. We begin with a brief technical background on language modeling.

2.1 Language Modeling

Basics. Language models are statistical models that assign a probability to every sequence of *words*. Given a code sequence $S = t_1 t_2 \dots t_N$, a language model estimates the probability of this sequence occurring as a product of a series of conditional probabilities for each token:

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

Each probability $P(t_i|t_1, \dots, t_{i-1})$ denotes the chance that the token t_i follows the previous tokens, the *prefix*, $h = t_1, \dots, t_{i-1}$. In practice, however, the probabilities are impossible to estimate, as there is an astronomically large number of possible prefixes. The most widely used approach to combat this problem is to use the $n$gram language model, which makes a *Markov assumption* that the conditional probability of a token is dependent only on the $n-1$ most recent tokens. The $n$gram model places all prefixes that have the same $n-1$ tokens in the same equivalence class:

$$P_{n\text{gram}}(t_i|h) = P(t_i | t_{i-n+1}, \dots, t_{i-1}) \quad (2)$$

The latter is estimated from the training corpus as the fraction of times that the prefix $t_{i-n+1}, \dots, t_{i-1}$ was followed by the token t_i. Note that, given a complete sentence, we can also compute each token given its epilog (the subsequent tokens), essentially computing the probability of the sentence in reverse. We make use of this approach to better identify buggy lines, as described in Section 3.3.

The $n$gram language models have been shown to successfully capture the highly repetitive regularities in source code, and were applied to code suggestion tasks [18]. However, the $n$gram models fail to deal with a special property of software: source code is *also very localized*. Due to module specialization and focus, code tends to take special repetitive forms in local contexts. The $n$gram approach, rooted as it is in $\mathcal{NL}$, focuses on capturing the global regularities over the whole corpus, and neglects local regularities, thus ignoring the *localness of software*. To overcome this, Tu *et al.* [36] introduced a cache language model to capture the localness of code.

Cache language models. These models (for short: *$gram*) extend the traditional language models by deploying an additional *cache* to capture the regularities in the locality. It combines the global ($n$gram) model with the local (*cache*) model as

$$P(t_i|h, cache) = \lambda \cdot P_{n\text{gram}}(t_i|h) + (1-\lambda) \cdot P_{cache}(t_i|h) \quad (3)$$

$cache$ is the list of $n$grams extracted from the local context, and $P_{cache}(t_i|h)$ is estimated from the frequency with which t_i followed the prefix h in the *cache*. To avoid hand-tuned parameters, Tu *et al.* [36] replaced the interpolation weight λ with $\gamma/(\gamma + H)$, where H counts the times the prefix h has been observed in the *cache*, and γ is a concentration parameter between 0 and infinity.

$$P(t_i|h, cache) = \frac{\gamma}{\gamma + H} \cdot P_{n\text{gram}}(t_i|h) + \frac{H}{\gamma + H} \cdot P_{cache}(t_i|h) \quad (4)$$

If the prefix occurs few times in the cache (H is small), then the $n$gram model probability will be preferred; vice versa. This setting makes the interpolation weight self-adaptive for different $n$grams.

The $n$gram and cache components capture different regularities: the $n$gram component captures the corpus linguistic structure, and offers a good estimate of the mean probability of a specific linguistic event in the corpus; around this mean, the local probability fluctuates, as code patterns change in different localities. The cache component models these local changes, and provides variance around the corpus mean for different local contexts.

We use a *$gram* here to judge the "improbability" (measured as cross-entropy) of lines of code; the core research questions being, can cross-entropy provide a useful indication of the likely bugginess of a line of code, and how does this approach performs against/with comparable approaches, such as static bug finders.

2.2 Static Bug-finders ($\mathcal{SBF}$)

The goal of $\mathcal{SBF}$ is to use syntactic and semantic properties of source code to indicate locations of common errors, such as undefined variables and buffer overflows. They rely on methods ranging from informal heuristic pattern-matching to formal algorithms with proven properties. These tools typically report warnings at build time; programmers can choose to fix them. Pattern-matching tools (*e.g.,* PMD and FindBugs [8, 12]) are unsound, but fast and widely used; more formal approaches are sound, but slower. In practice all tools have false positives and/or false negatives, thus inspecting and fixing all the warnings is not always cost-effective.

Suffice for our purposes to note here that both $\mathcal{SBF}$ and *$gram* both (fairly imperfectly) indicate likely locations of defects; so our goal here is to compare these rather different approaches, and see if they synergize. It should be noted that *$gram* is quite easy to implement, since it requires only lexical information; however, as we see below it can be improved with some syntactic information.

2.3 Evaluating Defect Predictions

In our setting, we view $\mathcal{SBF}$ and *$gram* as two commensurate approaches to selecting lines of code to which drawing the programmers' attention as locations worthy of inspection, since they just might contain real bugs. To emphasize this similarity, from here on we refer to language model based bug prediction as $\mathcal{NBF}$ ("**N**aturalness **B**ug **F**inder"). With either $\mathcal{SBF}$ or $\mathcal{NBF}$, programmers will spend effort on reviewing the code and hopefully find some defects. Comparing the two approaches requires a performance measure. We adopt a cost-based measure that has become standard [4]: AUCEC (Area Under the Cost-Effectiveness Curve). AUCEC (like ROC) is a non-parametric measure, which does not depend on the defects' distribution. AUCEC assumes that the cost is the inspection effort and the payoff is the count of bugs found.

We normalize both to 100%, measure the payoff increase as we inspect more and more lines and draw a 'lift-chart' or Lorenz curve. AUCEC is the area under this curve. Suppose we inspect x% code at random; in expectation, we would find x% of the bugs, thus yielding a diagonal line on the lift chart; so the expected AUCEC if inspecting 5% lines at random would be 0.00125.[1] Typically, inspecting 100% code is very expensive; one could reasonably assume that 5% or even just 1% of the code, in a large system, could realistically be inspected; therefore, we compare AUCECs for $\mathcal{NBF}$ and $\mathcal{SBF}$ for this much smaller proportion.

Additionally, we investigate defect prediction performance under several *credit criteria*. A prediction model is awarded credit,

[1]Calculated as 0.5 * 0.05 * 0.05. This could be normalized differently; but we consistently use this measurement, so our comparisons work.

Ecosystem	Project	Study Period	Snapshots	#Files	NCSL	# of Changes	# of Bugs
Github	Atmosphere	May-10 to Jan-14	17	17,206	6,329,400	2,481	1,130
	Elasticsearch	Feb-10 to Jan-14	17	103,727	22,156,904	4,922	1,077
	Facebook-android-sdk (fdk)	May-10 to Dec-13	16	3,981	1,431,787	320	143
	Netty	Aug-08 to Jan-14	24	57,922	12,969,858	3,906	1,485
	Presto	Aug-12 to Jan-14	7	23,086	6,496,149	1,635	330
Apache	Derby	Sep-04 to Jul-14	41	143,906	61,192,709	5,275	1,453
	Lucene	Sep-01 to Mar-10	36	47,270	11,744,856	2,563	469
	OpenJPA	May-06 to Jun-14	34	131,441	27,709,778	2,956	558
	Qpid	Sep-06 to Jun-14	33	94,790	24,031,170	3,362	657
	Wicket	Sep-04 to Jun-14	41	159,332	28,544,601	10,583	994
Overall		Sep-01 to Jul-14	266	782,661	202,607,212	38,003	8,296

Table 1: **Summary data of projects that are analyzed for finding all the defects including development time bugs**

ranging from 0 to 1, for each line marked as defective. Previous work by Rahman *et al.* has compared $\mathcal{SBF}$ and $\mathcal{DP}$ (a file level statistical defect predictor) models using two types of credit: full (or optimistic) and partial (or scaled) credit [28], 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 awarded to the model is the fraction of the bug's defective lines that the model marked. Hence, partial credit assumes that the probability of a developer finding a bug is proportional to the fraction of the bug that is marked by the defect prediction model.

2.4 Research Questions

At the core of our research is the question whether "*un*naturalness" (measured as entropy, or improbability) is indicative of poor code quality. The abundant history of changes (including bug fixes) in OSS projects allows the use of standard methods [33] to find code that was implicated in bug fixes ("buggy code").

RQ1. Are buggy lines less "natural" than non-buggy lines?

In project histories, we can find numerous samples of bug fixes, where buggy code is replaced by bug-fix code to correct defects. Do language models rate bug-fix code as more natural than the buggy code they replaced? This would essentially mean that the bug fix code is assigned a higher probability than the buggy code. Such a finding would also have implications for automatic, search-based bug repair: If fixes tend to have higher probability, then a good language model might provide an effective organizing principle for the search, or perhaps (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 fingered as unnatural by language models, it is likely to be an unreliable indication; thus one can expect many false positives (correct lines indicated as unnatural) and false negatives (buggy lines indicated as natural). It would be interesting to know, however, how well naturalness (*i.e.* entropy) is a good ordering 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 and bugs); however, defectiveness is predicted at the granularity of files (or methods), thus, it is reasonable to compare naturalness as an ordering principle with $\mathcal{SBF}$, which provide warnings at the line level.

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

It is reasonable to expect that, if $\mathcal{SBF}$ provides a warning on a line *and* it appears unnatural to a language model, then it is even more likely a mistake. We therefore investigate whether naturalness 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 $\mathcal{SBF}$?

3. METHODOLOGY

In this section, we describe the projects that we studied and our approaches to data gathering and analysis.

3.1 Study Subject

We studied 10 OSS java projects, as shown in Table 1: Among them Atmosphere (an asynchronous web socket framework), FDK (an Android SDK for building Facebook application), Elasticsearch (a distributed search engine for cloud), Netty (an asynchronous network application framework), and Presto (a distributed SQL query engine) are Github projects, while Derby (a relational database), Lucene (a text search engine library), OpenJPA (a Java Persistence API), Qpid (a messaging system), and Wicket (a lightweight web application framework) are taken from Apache Software Foundation. We deliberately chose the projects from different application domains to measure $\mathcal{NBF}$'s performance in various types of systems. The Apache projects are relatively older; Lucene, the oldest one, started in 2001. The earliest Github project in our dataset (Netty) started in 2008. All projects are under active development.

We analyzed $\mathcal{NBF}$'s performance on this data set in two settings. In the first setting (see *Phase-I* in Section 3.2) we consider all the bugs—both development time and post release—that have appeared in the project's evolution. The performance is analyzed at different stages of each project's evolutionary history. We extracted snapshots of individual projects at an interval of 3 months from the version history. Such snapshots represent the current states of the projects at that time period (see Section 3.2 for details). In total, we analyzed 266 snapshots across 10 projects that include 782,661 distinct file versions, and 202.6 Million total non-commented source

Project	NCSL #K	#Warnings FindBug	PMD	#Issues
Derby (7)	420–630	1527–1688	140–192K	89–147
Lucene (7)	68–178	137–300	12–31K	24–83
OpenJPA (7)	152–454	51–340	62–171K	36–104
Qpid (5)	212–342	32–66	69–80K	74–127
Wicket (4)	138–178	45–86	23–30K	47–194

Table 2: **Summary data of projects that are analyzed for locating bugs reported in issue database. The dataset is taken from Rahman *et al.***

code lines (NCSL). These snapshots contain 38,003 distinct commits, of which 8,296 were marked as bug fixing changes using the procedures outlined in Section 3.2. The corresponding bugs include both development-time bugs as well as post-release bugs.

In the second setting, we only focus on post-release bugs that are reported in an issue tracking system. We used the data set prepared by Rahman *et al.* [28], in which snapshots of the five Apache projects were taken at selected project releases. At each snapshot, the project size varies between 68 and 630K NCSL. The bugs were extracted from Apache's JIRA issue tracking system and the total number of bugs reported against each release across all the projects varies from 24-194. Table 2 summarizes this dataset.

At each release version, Rahman *et al.* further collected warnings produced by two static bug finding tools, namely FINDBUGS [5] and PMD [8]. PMD operates on source code and produces line-level warnings; FINDBUGS operates on Java bytecode [5] and reports warning at line, method, and class level. For this reason FINDBUGS produces warnings covering significantly more lines, though the number of unique warnings is smaller than that of PMD (see Table 2). To make the comparisons between $\mathcal{NBF}$ and $\mathcal{SBF}$ fair, we further filtered out warnings for commented lines, since $\mathcal{NBF}$'s entropy calculation does not consider the commented lines. In fact, we have noticed that a majority of FINDBUGS line-level warnings are actually commented lines. Thus after removing comments, we are left with primarily method and class level FINDBUGS warnings.

3.2 Data Collection

As mentioned earlier, our experiment has two distinct phases. First, we describe the process of collecting data for Phase-I, which tries to locate all the bugs that developers fix during an ongoing development process. Next, we briefly summarize data collection of Phase-II, which locates bugs at project release time; this data set is taken from Rahman *et al.* [28].

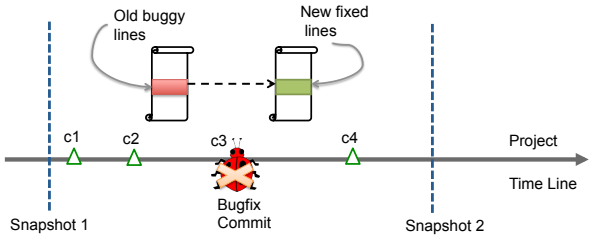


Figure 1: **Phase I Data Collection: note Project Time Line, showing snapshots (vertical lines) and commits (triangles) at c1. . .c4. For every bugfix file commit (c3) we collect the buggy version and the fixed version, and use `diff` to identify buggy & fixed lines.**

Phase-I. All our projects used GIT; we downloaded a snapshot of each project at 3-month intervals, beginning with the project's inception. A snapshot represents the project's state at that point of time, as shown by the dashed vertical lines in Figure 1. Then we retrieve all the commits (c_is in the Figure) made between each pair of consecutive snapshots. Each commit involves an old version of a file and its new version. Using `git diff` we identify the lines that are changed between the old and new versions. We also collected the number of deleted and added lines in every commit. We then removed the commits comprising more than 30 deleted lines (30 lines was at the 3^{rd} quartile of the sample of deleted lines per commit in our data set). We further removed the commits with no deleted lines, because we are only interested in locating buggy lines present in the old versions. Figure 2 shows a histogram of number of deleted lines per file commit; it ranges from at least one line to at max 30 lines deletion per commit with a median at 5.[2]

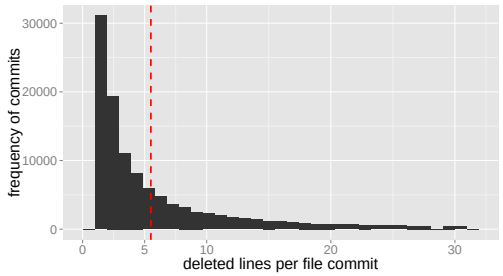


Figure 2: **Histogram of the number of lines deleted per file commit. The mean is 5, marked by the dashed line**

Each commit has an associated commit log. We mark a commit as *bugfix*, if the corresponding commit log contains at least one of the error related keywords: 'error', 'bug', 'fix', 'issue', 'mistake', 'incorrect', 'fault', 'defect' and 'flaw', as proposed by Mockus and Votta [22]. In this step, we first convert each commit message to a bag-of-words; we then remove words that appear only once among all of the bug fix messages to reduce project specific keywords; finally, we stem the bag-of-words using standard natural language processing (NLP) techniques. This method was taken from our previous work [30]. The deleted lines corresponding to the old version of a bugfix commit are marked as *buggy* lines. The added lines associated with new corrected version are marked as *fixed* lines.

Thus, from three sets of files (the files that did not change between two snapshots and the old and new versions of the changed files), we retrieve three sets of lines: (1) *unchanged lines*: all lines of the unchanged files and unchanged lines of the changed files. (2) *buggy lines*: lines that were corrected in the old version of the bugfix commits, and (3) *fixed lines*: lines that were fixed in the new version of the bugfix commits. In total, we compared 58,374,475 unchanged lines, 88,058 buggy lines, and 204,242 fixed lines across all the snapshots of all the projects.

Phase-II. To begin with, certain release versions of each Apache project were selected. Then, from the JIRA issue tracking system of the Apache projects, the post-release bugfix commits (corresponding to the selected release) were identified. Next, by blaming the old buggy file version associated with a bugfix commit using `git blame`, the corresponding buggy lines were detected. Since in this phase we are interested in locating post-release bugs, the identified buggy lines were further mapped to the release time file version using an adopted version of SZZ algorithm [33]. For each project release version, final outcome of Phase-II is two sets of lines: (1) *buggy lines*: lines that were marked as buggy lines based on post release fix and (2) *non-buggy lines*: all the other lines across all the Java files present at the release version.

[2]In the unfiltered data set the median was at 2

3.3 Measuring entropy using cache language model

Entropy of code snippets. We measure the *naturalness* of a code snippet using statistical language model with a widely-used metric – *cross-entropy* (*entropy* in short) [18, 2]. The key intuition is that snippets that are more like the training corpus (*i.e.* more natural) would be assigned higher probabilities or lower entropy from an LM trained on the same corpus. Given a snippet $S = t_i \dots t_N$, of length N, with a probability $P_M(S)$ estimated by a language model M. The entropy of the snippet is calculated as:

$$H_M(S) = -\frac{1}{N}\log_2 P_M(S) = -\frac{1}{N}\sum_{1}^{N}\log_2 P(t_i|h) \qquad (5)$$

$P(t_i|h)$ is calculated by the cache language model via Equation 4.

Building a Cache Language Model. For each project and each pair of snapshots, we are interested in the entropy of lines that were marked as buggy in some commit between these snapshots. We would like to contrast these entropy scores with those of lines that were not changed in any bug-fix commit in this same period. To compute these entropy scores, for each file, we first train a language model on the 'old' version of all other files (the version at the time of the previous snapshot), counting the sequences of tokens of various lengths; we then run the language model on the the current file, computing the entropy of each token based on both the prolog (the preceding tokens in the current file) and epilog (the succeeding tokens); finally, we compute the entropy of each line as the average of the entropy of each token on that line.

As an optimization step, we divided all 'old' versions of files into ten bins. Then, whenever testing on a file, we use the pre-counted training set on the nine bins that the old version of the current file is not in. This removes the need to compute a training set for each file separately. Since we use the cache-based language model, the entropy scores within each file are calculated using both the training set on the other nine bins and a locally estimated cache, built on only the current file, since Tu et al. [36] reported that building cache on both the prolog and epilog achieves best performance.

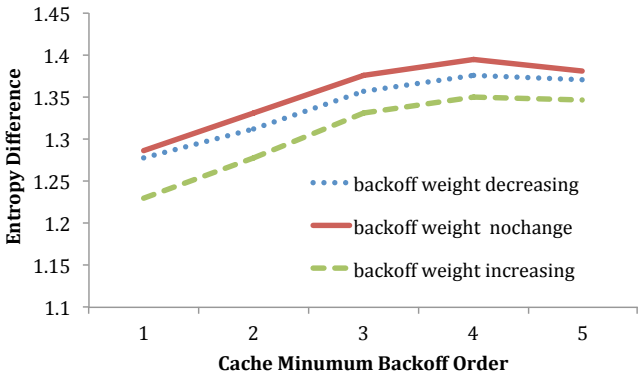


Figure 3: **Determining parameters of cache model. The experiments were conducted on Elasticsearch and Netty projects for one-line bugfix changes. Y axis represents difference of entropy of a buggy line *w.r.t.* non-buggy lines in the same file.**

Determining parameters for cache language model. Several factors of the locality would affect the performance of cache language model [36]: *cache context*, *cache scope*, *cache size*, and *cache order*. For the fault localization task, we build the cache on all the existing code in the current file. In this light, we only need to tune the cache order (*i.e.* the *maximum* and *minimum* order of $n$grams stored in the cache). In general, longer $n$grams are more reliable but quite rare, thus we back-off to shorter matching prefixes [20] when needed. We follow Tu et al. [36] to set the maximum order of cache $n$grams to 10. To determine the minimum back-off order, we performed experiments on the Elasticsearch and Netty projects to find the optimal performance, measured in terms of difference in entropy between buggy and non-buggy lines (Figure 3). The figure shows entropy difference with varying minimum backoff order and three different backoff weights (increasing, decreasing, no-change). We observed maximum difference in entropy between buggy and non-buggy lines at minimum backoff order of 4 with no change in the backoff weight. Thus, we set the minimum backoff order be 4 and the backoff weight be 1.0.

3.4 Adjusting the entropy scores

An important assumption underlying the applicability of language models to defect *prediction* is that higher entropy is associated with bug prone-ness. In practice, buggy lines are quite rare, thus a few non-buggy lines with high entropy scores could substantially increase false negatives and worsen performance. We undertook some tuning efforts to sharpen $\mathcal{NBF}$'s prediction ability.

We manually examined entropy scores of sample lines and found strong associations with lexical and syntactic properties. In particular, lines with many and/or previously unseen identifiers, such as package, class and method declarations, had substantially higher entropy scores than average. Lines such as the first line of for-statements and catch clauses had much lower entropy scores, being often repetitive and making use of earlier declared variables. We use these variations in entropy scores by introducing the notion of *line types*, based on the code's syntactic structure, *i.e.* the abstract syntax tree (AST), and computed a syntax-sensitive entropy score.

First, with each line, we associated syntax-type, corresponding to the grammatic entity that is the lowest AST node that includes the full line. These are typically AST node types such as statements (*e.g.,* if, for, while), declarations (*e.g.,* variable, structure, method) or nodes that typically span one line, such as switch cases and annotations. We then compute a normalized Z-score for the entropy of the line, *over all lines with that node type*.

$$z_{\text{line, type}} = \frac{entropy_{\text{line}} - \mu_{\text{type}}}{SD_{\text{type}}}$$

The above normalization essentially uses the extent to which a line is "unnatural" with respect to other lines of the same type, to predict how likely it is to be buggy. In addition, we don't expect all line types to be equally buggy; package declarations (`import...`) are probably usually correct, when compared to error handling (`try...catch`). The previously computed line-types come in handy here too: we can compute the *relative bug-proneness* of a type based on the fraction of bugs and total lines it had in all previous snapshots. Hence, we use the first snapshot as a 'training set' for this model and compute the bug-weight of a statement as:

$$w_{\text{type}} = \frac{bugs_{\text{type}}/lines_{\text{type}}}{\sum_{\text{t}\in\text{types}} bugs_{\text{t}}/lines_{\text{t}}}$$

where the bugs and lines of each type are counted over all previous snapshots. We then scale the z-score of each line by it's weight w to achieve our final model, which we name \$gram+wType.

project	max delete = 2		max delete = 5		max delete = 10		max delete = 20		max delete = 30	
	difference	effect	difference	effect	difference	effect	difference	effect	difference	effect
atmosphere	1.17 to 1.69	0.50	0.94 to 1.23	0.38	0.74 to 0.94	0.30	0.36 to 0.53	0.16	0.38 to 0.53	0.16
derby	1.56 to 1.96	0.56	1.63 to 1.85	0.55	1.32 to 1.47	0.44	1.01 to 1.13	0.34	0.84 to 0.94	0.28
elasticsearch	1.58 to 1.90	0.57	1.37 to 1.53	0.48	1.01 to 1.14	0.35	0.84 to 0.95	0.29	0.73 to 0.82	0.25
fdk	0.74 to 1.64	0.40	1.33 to 1.83	0.53	1.03 to 1.41	0.41	1.04 to 1.33	0.40	0.92 to 1.17	0.35
lucene	1.27 to 1.80	0.48	0.97 to 1.28	0.35	0.83 to 1.06	0.30	0.97 to 1.14	0.33	0.79 to 0.96	0.28
netty	1.97 to 2.24	0.68	1.58 to 1.74	0.54	1.32 to 1.44	0.45	1.12 to 1.22	0.38	0.98 to 1.07	0.33
openjpa	1.61 to 2.12	0.59	1.15 to 1.42	0.41	0.89 to 1.10	0.32	0.68 to 0.84	0.24	0.60 to 0.75	0.21
presto	1.12 to 1.73	0.47	0.95 to 1.30	0.37	0.88 to 1.14	0.33	0.76 to 0.96	0.28	0.72 to 0.90	0.27
qpid	1.35 to 1.75	0.51	1.19 to 1.40	0.42	0.98 to 1.13	0.35	0.65 to 0.77	0.23	0.58 to 0.68	0.21
wicket	1.51 to 1.88	0.56	1.44 to 1.64	0.51	1.18 to 1.33	0.41	0.95 to 1.08	0.33	0.92 to 1.03	0.32
overall	1.67 to 1.80	0.56	1.41 to 1.48	0.47	1.13 to 1.18	0.37	0.91 to 0.95	0.30	0.80 to 0.84	0.26

Table 3: **Buggy lines, in general, have higher entropy than non-buggy lines. Difference is measured with t-test for 95% confidence interval, and effect is Cohen's D. Wilcox non-parametric test also confirmed buggy lines have higher entropy with statistical significance. 'max delete' represents maximum number of buggy lines that are fixed in a file commit.**

4. EVALUATION

We begin with the question at the core of this paper:

RQ1. Are buggy lines different from non-buggy lines?

For each project, we compare line entropies of buggy and non-buggy lines. For a given snapshot, non-buggy lines consist of all the unchanged lines and the deleted lines that are not part of a bug-fix commit. The buggy lines include all the deleted lines in all bug-fix commits. Figure 4 shows the result, averaged over all the studied projects. Buggy lines are associated with higher entropies. Table 3 further details the average entropy difference between the buggy and non-buggy lines (buggy $>$ non-buggy) and the effect sizes (Cohen's D) between the two. Wilcox non-parametric test confirms the difference with statistical significance (p-value $< 2.2{*}10^{-16}$).

Note that both entropy difference and effect size decrease as we increase the threshold for the maximum number of deleted lines (max_delete) in a file commit. For example, for a max_delete size of 2, the entropies of buggy lines are on average 1.67 to 1.80 bits higher, with a high effect size of 0.56. However, when we consider all the studied commits (max_delete = 30), the entropies of buggy lines are, on average, less than a bit (0.80 to 0.84 bits) higher, with a small-to-moderate effect size of 0.26. One possible explanation: particularly in larger bug-fix commits, some of the deleted (or modified) lines might only be indirectly associated with the erroneous lines that most improbable ("unnatural"). These indirectly associated lines might actually be common, and thus have lower entropy; this would diminish overall entropy differences between the buggy and non-buggy lines. However, with statistical significance, we have the overall result:

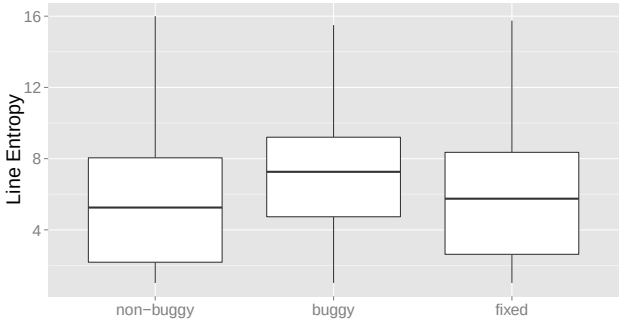


Figure 4: **Entropy difference between non-buggy, buggy, and fixed lines. File commits upto 5 deleted lines are considered, since five is the average number of deleted lines per file commit (see Figure 2).**

Result 1: *Buggy lines, on average, have higher entropies than non-buggy lines.*

RQ2. Does entropy of a buggy line drop after the bug is fixed?

In a bug-fix commit, the lines deleted from the original versions are considered *buggy lines* and lines added in fixed versions are considered *fixed lines*. To answer RQ2, we collected all the buggy and the fixed lines across all the projects and compared their average entropies. It is hard to establish a one-to-one correspondence between a buggy and a fixed line, because often buggy lines are fixed by a different number of new lines. Hence, we compare the mean entropies between buggy and non-buggy hunks. Figure 4 shows the result. On average, entropy of the buggy lines drop after the bug-fixes, with a drop of 1.19 to 1.26 bit, with 95% confidence.

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 (see the commit log). 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 4.6257 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.

The table below shows the average drop of entropy and the Cohen's D effect size of buggy vs. fixed lines, with varying thresholds for the maximum bug size in terms of deleted lines.

Max Delete	2	5	10	20	30
Entropy drop after bugfix	1.52 to 1.62	1.19 to 1.26	0.88 to 0.94	0.65 to 0.70	0.53 to 0.58
Effect Size	0.51	0.40	0.30	0.22	0.18

Similar to the result of RQ1, both the entropy difference and effect size vary with maximum delete threshold: when the delete threshold increases, the other two decrease. For example, at maximum delete size 2, mean entropy drops from 1.52 to 1.62 bit (with 95%

Example 1 : Wrong Initialization Value
Facebook-Android-SDK (2012-11-20)
File: `Session.java`
Entropy dropped after bugfix : **4.12028**

```
  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**

```
  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**

```
  if (!directory.exists())
    // Before (entropy = 9.213675):
-   directory.mkdir();
    // After (entropy = 5.33941):
+   if (!directory.mkdir())
+     throw new IOException
          ("Cannot create directory:  " +
               directory);

...
```

Table 4: **Examples of bug fix commits that $\mathcal{NBF}$ 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 red, and the fixes are highlighted in green.**

Example 4 : Wrong Argument ($\mathcal{NBF}$ could not detect)
Netty (2010-08-26)
File: `HttpMessageDecoder.java`
Entropy increased after bugfix : **5.75103**

```
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: "
                          + maxHeaderSize);
}
```

Example 5 : ($\mathcal{NBF}$ detected incorrectly)
Facebook-Android-SDK (multiple snapshots)
File: `Request.java`

```
  // Entropy = 9.892635
  Logger logger = new Logger(LoggingBehaviors.
      REQUESTS, "Request");
   ...
```

Table 5: **Examples of bug fix commits where $\mathcal{NBF}$ did not perform well. In Example 4, $\mathcal{NBF}$ could not detect the bug successfully (marked in red) and after bugfix the entropy has increased. In Example 5, $\mathcal{NBF}$ incorrectly detected the line as buggy due to its high entropy value.**

confidence) with statistical significance, with a large effect size ($>$ 0.50). However, with delete threshold at 30, mean entropy difference between the buggy and non-buggy lines are only half a bit with a small effect size of 0.18. For all the studied ranges, the Wilcox non-parametric test confirms with statistical significance that the entropy of buggy lines is higher than the entropy of the fixed lines.

However, in certain cases these observations do not hold. For instance, in the example 4 of Table 5, entropy increased after the bug fix by 5.75 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 [29]. 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 of Table 5, the statement related to `logger` was newly introduced in the corpus. Hence, its entropy was higher although it was not a bug.

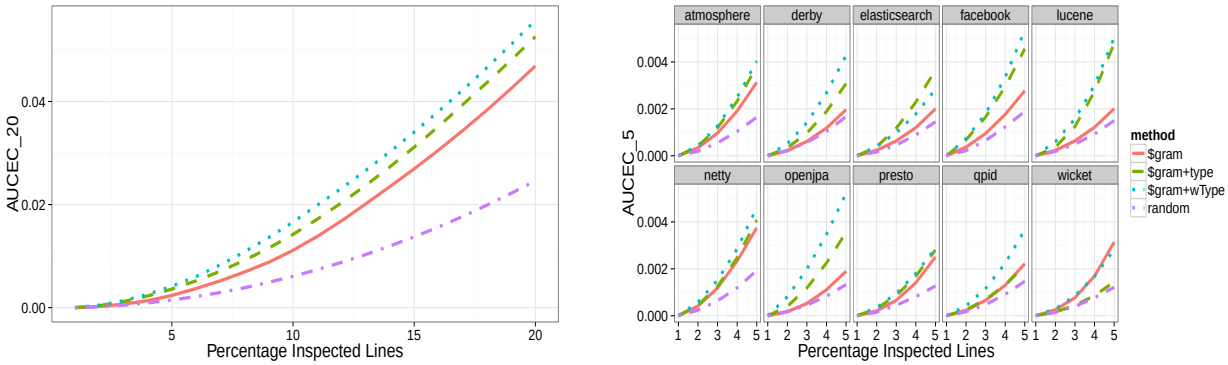
> **Result 2:** *Entropy of the buggy lines drops after bug-fixes, with statistical significance.*

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

Having established that buggy lines are significantly less natural than non-buggy lines, we investigate whether entropy of a line can be used to direct inspection effort towards buggy code. In particular, we start by asking whether ordering lines by entropy will better guide inspection effort that ordering lines at random. For the reasons outlined in Section 2.3, we evaluate the performance of entropy-ordering, with the AUCEC scores at 5% of inspected lines ($AUCEC_5$ in short). Furthermore, as outlined in section 2.3, we evaluate performance according to two types of *credit*: partial and full (in decreasing order of strictness). Finally, we disregarded all bugs that were part of a bug-fix which removed 15 or more lines, which we found this to be the 95th percentile of bug-fix sizes. As shown in Table 3, entropy plays a substantially smaller role in lines belonging to larger bug-fixes, hence we leave the identification of these lines to future research. We remind the reader that $AUCEC_5$ is a non-parametric, cost-sensitive measure, and the comparison to random choice is done on an equal credit basis.

Figure 5(a) shows the AUCEC scores for partial credit, averaged over all projects, up to 20% of the inspected lines. Figure 5(b) offers a closer look at the performance on the 10 studied projects, up to 5% of the inspected lines. We see that, under partial credit, the default \$gram model (without the syntax weighting described in §3.4) performs significantly better than random, particularly at more than 10% of inspected lines. However, at 5% of inspected lines its performance varies, consistently performing better than random but often just slightly. Indeed, average performance of \$gram was significantly better than random at 20% (nearly twice as good) but only marginally so at 5% (17% better than random).

This picture changes substantially with the introduction of linetypes. Scaling the entropy scores by line type improves $AUCEC_5$ performance in all but one case (Wicket) and significantly improves performance in all cases where \$gram performed no better than random. Including the bugginess history of linetypes (\$gram+wType) furthermore improves prediction performance in all but one system (Elasticsearch). The latter model consistently outperforms random and \$gram (except on Wicket), achieving $AUCEC_5$ scores of more than twice that of random. These results were quite the same under full credit. Since \$gram+wType is the best-performing “naturalness" approach, we hereafter refer to it as $\mathcal{NBF}$.



(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

Figure 5: **Performance Evaluation of $\mathcal{NBF}$ with Partial Credit.**

> **Result 3:** *Entropy, is a better way to choose lines for inspection than random*

In previous work, Rahman *et al.* compared static bug-finders with statistical defect prediction approaches [28]. To this end, they created a dataset consisting of 32 releases of 5 popular Apache projects and annotated the lines in each release with both bug information and $\mathcal{SBF}$ information, as described in §3.1. Among others, they found that ordering $\mathcal{SBF}$ warnings based on statistical defect prediction methods can improve the native $\mathcal{SBF}$ ordering. This provides an interesting challenge for the $\mathcal{NBF}$ algorithm: how does the $\mathcal{NBF}$ algorithm compare to $\mathcal{SBF}$, and can we improve the default ordering of the $\mathcal{SBF}$ by using the techniques presented before? We investigate this in the next two research questions.

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

To compare $\mathcal{NBF}$ with $\mathcal{SBF}$, we computed entropy scores for each line in Rahman's dataset using the \$gram+wType model. Here we again use the threshold of 14 lines for bugs, which roughly corresponded to the fourth quartile of bug-fix sizes on this dataset. Indeed, we found that Rahman's dataset had substantially more 'large' bugs compared to our earlier experiments, hence we also report results without imposing this threshold.

Rahman *et al.* developed a measure named AUCECL to compare $\mathcal{SBF}$ and $\mathcal{DP}$ methods on an equal footing. In this method, the $\mathcal{SBF}$ under investigation sets the line budget based on the number of warnings it returns and the $\mathcal{DP}$ 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 repeat to compare $\mathcal{SBF}$ with $\mathcal{NBF}$.

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 $\mathcal{SBF}$, we simulate them as follows: First assign each line the value zero if it was not marked by the $\mathcal{SBF}$ and the value of the $\mathcal{SBF}$ 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 returned by $\mathcal{SBF}$: first from those marked by the $\mathcal{SBF}$ in descending (native, $\mathcal{SBF}$ tool-based) priority, and within each priority level at random. We repeat the simulation multiple times and average the performance.

Figure 6(a) and 6(b) show the AUCEC_5 and AUCECL scores for PMD on the dataset by Rahman *et al.* [28] using partial credit. 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). $\mathcal{SBF}$ and $\mathcal{NBF}$ performed comparably; $\mathcal{NBF}$ performed slightly better when using both partial credit and the specified threshold for bug-sizes, but when dropping the threshold, and/or with full credit, no significant difference remains between $\mathcal{NBF}$ and $\mathcal{SBF}$. 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.

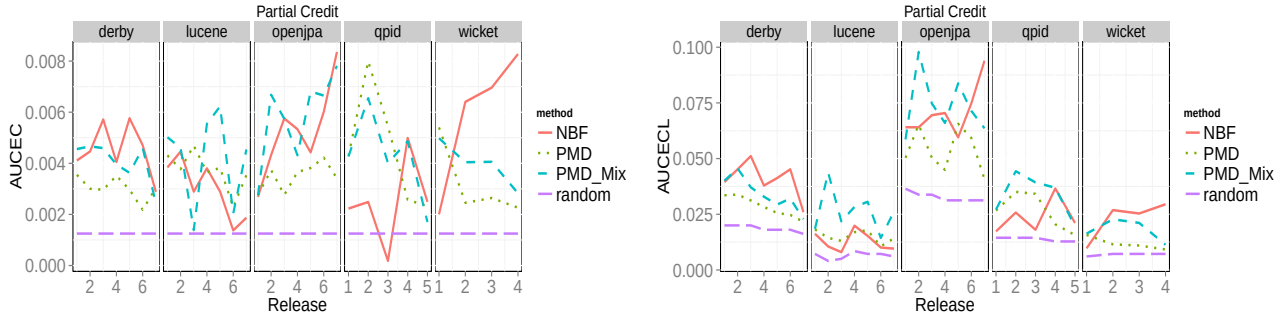
> **Result 4:** *Entropy achieves comparable performance to commonly used $\mathcal{SBF}$ in defect prediction.*

Notably, $\mathcal{NBF}$ had both the highest mean and standard deviation of the tested models, whereas PMD's performance was most robust. This suggest a combination of the models: We can order the warnings of the $\mathcal{SBF}$ using the \$gram+wType model. In particular, we found that the standard priority ordering of the $\mathcal{SBF}$ 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 $\mathcal{SBF}$?

Given the comparable performance of the $\mathcal{SBF}$ and $\mathcal{NBF}$ models and the robustness of the $\mathcal{SBF}$ algorithms, we may expect a combination of the models to yield superior performance. To this end, we again assigned values to each line based on the $\mathcal{SBF}$ priority as in RQ4. However, rather than add random tie-breakers, we rank the lines within each priority bin by the (deterministic) \$gram+wType score. The results for PMD are shown in Figure 6, first using the AUCEC_5 measure (6(a)) and then using the AUCECL measure (6(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 $\mathcal{SBF}$s in all cases ($p < 0.01$) and performed similarly to the $\mathcal{NBF}$ model (significantly better on Lucene and QPid, significantly worse



(a) $AUCEC_5$ performance of $gram+wType vs. PMD and the combination model

(b) AUCECL performance of $gram+wType vs. PMD and the combination model

Figure 6: **Partial Credit: Performance Evaluation of $gram+wType *w.r.t.* $\mathcal{SBF}$ and a mix model which ranks the $\mathcal{SBF}$ lines by entropy.**

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 $\mathcal{SBF}$ by entropy (*ignoring* the $\mathcal{SBF}$ priorities) and found comparable but slightly weaker results. These results suggests that both $\mathcal{NBF}$ and $\mathcal{SBF}$ contribute valuable information to the ordering of bug-prone lines and that their combination yields superior results.

Result 5: *Ordering $\mathcal{SBF}$ warnings by priority and entropy significantly improves $\mathcal{SBF}$ performance.*

5. THREATS TO VALIDITY

A number of threats to the internal validity of the study arise from the experimental setup.

Identifying Buggy Lines. The identification of buggy lines is a possible source of error. We used the procedure as proposed by Mockus and Votta to identify bugfix commits [22], which may have lead to both false negatives and false positives in the identification of buggy lines.

Programmers may fail to indicate bug fixes in log messages, leading to false negatives (missing bugs). There is no reason to suspect that these missing buggy lines have a significantly different entropy-profile. Second, as yet unfixed bugs may linger in the code, constituting a right-censorship of our data; these bugs might have a different entropy-profile (although, again, no reason to suspect that this is so). Finally, it has been noted that developers may combine multiple unrelated changes in one commit [17, 11]. This work (*op. cit.*) observed that, when multiple changes were combined, bug-fix commits were mostly combined with refactorings and code formatting efforts. This may have lead to the deletion of non-buggy high-entropy lines in a bugfix commit, although we expect these lines to form the minority of studied lines.

The threats identified above may certainly have lead to the misidentification of some lines; however, given the high significance of the difference in entropy between buggy and non-buggy lines, we consider it unlikely that these threats could invalidate our overall results. Furthermore, the performance in defect prediction of a model using entropies on the (higher quality, JIRA-based) bug dataset by Rahman *et al.* confirms our expectations regarding the validity of these results. A final threat regarding RQ2 is the identification of 'fixed' lines, lines that were added in the place of 'buggy' lines during a bugfix commit. It is possible that the comparisons between these categories is skewed, *e.g.,* because bugfix commits typically replace buggy lines with a larger number of fixed lines. We found no evidence of such a phenomenon but acknowledge the threat nonetheless. Future research may apply entropy to defect *correction* and further study this relation of buggy lines to fixed lines in terms of entropy.

In RQ4 and RQ5 we investigated the performance of the proposed $\mathcal{NBF}$ in comparison to (and in combination with) static bug finders. We note that here too the identification of buggy lines may be a cause for systematic error, for which we point both to the above discussion and to Section 5 of Rahman *et al.*, in which they identify a number of threats to the validity of their study [28].

Our comparison of $\mathcal{SBF}$ and $\mathcal{NBF}$ assumes that indicated lines are equally informative to the inspector, which is not entirely fair; $\mathcal{NBF}$ just marks a line as "surprising", whereas $\mathcal{SBF}$ provides specific warnings. On the other hand, we award credit to $\mathcal{SBF}$ whether or not the bug has anything to do with the warning on the same lines; indeed, earlier work [35] 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 [37]; the development of better, widely-accepted measures remains a topic of future research.

Generalizability. The selection of systems constitutes a potential threat to the external validity of this research. We attempted to minimize this threat by using systems from both Github and Apache, having a substantial variation in age, size and ratio of bugs to overall lines (see table 1).

Finally, does this approach generalize to other languages? There's nothing language-specific about the *implementation* of n-gram and *$gram* models (the $gram+wType model, however, does require parsing, which depends on language grammar). Our own prior research [18, 36] showed that these models work well to capture regularities languages such as Java, C, and Python, yielding low cross-entropies when well trained on a corpus of code. The question remains then whether language models can identify buggy code in other languages as well, and whether the entropy drops upon repair. As a sanity check, using the same methods described in Section 3, we gathered data from 3 C/C++ projects (`Libuv`, `Bitcoin` and `Libgit`). These projects together constituted just over 10M LOC. We gathered snapshots spanning the period of November 2008 - January 2014. The data comprised a total of 8298 commits, including 2518 bug-fix commits (identified as described in Section 3).

We lexicalized, and parsed these projects and computed entropy scores over buggy lines, fixed lines, and non-buggy lines as described earlier. The results were fully consistent with those presented in Table 3 and Figure 4; we found that buggy lines were between 0.87 and 1.16 bits more entropic than non-buggy llines when using a threshold of 15 lines (slightly smaller than among the Java projects). For a threshold of 2 lines, this difference was between 1.61 and 2.23 bits (slightly larger than in Table 3). Furthermore, the entropy of buggy lines (when using a threshold of 15 lines) dropped by nearly one bit on this dataset as well. These findings mitigate the external validity threat of our work, and strongly suggest that our results generalize to C/C++; we are investigating the applicability to other languages.

6. RELATED WORK

In the following we analyze work related to our investigation.

6.1 Statistical Defect Prediction

Software development is an incremental process. This incremental progress is successfully logged in version control systems like git, svn and issue databases. Learning from such historical data of reported (and fixed) bugs, Statistical Defect Prediction (DP) aims to predict location of the defects that are yet to be detected. This is a very active area (see [7] for a survey of the area), even having the dedicated PROMISE series of conferences (See [1] for recent proceedings). The state of the art DP not only leverages bug history, it also takes into account several other product (file size, code complexity, code churn etc.) and process metrics [27] (developer count, code ownership, developer experience, change frequency *etc.*.) to improve the prediction model. Thus, using different supervised learning techniques like logistic regression, Support Vector Machine (SVM) etc., DP associates various software entities (e.g., methods, files and packages) with their respective defect proneness.

Given a fixed budget of SLOC that needs to be inspected to effectively find most bugs, DP ranks files that one should inspect to detect most of the errors. DP doesn't necessarily have to work at the level of files; one could certainly use prediction models at the level of modules, or even at the level of methods. To our knowledge, no one has done purely statistical models to predict defects at a line-level, and this constitutes a novel aspect of our work. 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.

6.2 Static Bug Finders

The core idea of static bug finding is to develop an algorithm that automatically finds likely locations of known categories of defects in code. Some use heuristic pattern-matching; others are sophisticated algorithms that compute well-defined semantic properties over abstractions of programs carefully designed to accomplished specific speed-*vs*-accuracy tradeoffs in detecting certain categories of bugs. The former tools include FindBugs and PMD, which we studied; the latter includes tools like ESC-Java [13]. The former category can have both false positives and negatives. The overriding imperative in the latter approach is to never falsely certify a program (that actually has *e.g.,* memory leak bugs) to be bug-free; typically however, false positives can be expected.

The field has advanced rapidly, with many developments; researchers identify new categories of defects, and seek to invent clever methods to find these defects efficiently, either heuristically or though well-defined algorithms and abstractions. Since neither method is perfect, the actual effectiveness in practice is an empirical question. Since our goal here is just to compare $\mathcal{SBF}$ and $\mathcal{NBF}$, we refer the reader for a more complete discussion of related work regarding $\mathcal{SBF}$ and their evaluation to Rahman *et al.* [28].

6.3 Grammatical Error Correction in NLP

Grammatical error correction is an important problem in natural language processing (NLP), which is to identify grammatical errors and provide possible corrections for them. The pioneering work on grammatical error correction was done by Knight and Chander [21] on article errors. Along the same direction, researchers haver proposed different classifiers with better features for correcting article and preposition errors [16, 34, 15, 9]. However, the classifier approaches mainly focus on identifying and correcting specific types of errors (e.g. preposition misuse). To approach this problem, some researchers have begun to apply the statistical machine translation approach to error correction. For example, Park and Levy [26] model various types of human errors using a noisy channel model, while Dahlmeier and Ng [10] describe a discriminative decoder to allow the use of discriminative expert classifiers.

There is one fundamental difference between grammatical error correction in natural languages and defect localization in programming languages. Natural languages are close-vocabulary (i.e. have limited number of vocabulary), thus lead to limited types of grammatical errors (e.g. articles, prepositions, noun number) with enumerable corrections (e.g. possible article choices are *a/an*, *the*, and the empty article ϵ). In contrast, programming languages are open-vocabulary (e.g. programmers could arbitrarily construct new identifiers). Therefore, the defects in programming languages are more flexible and thus harder to localize. Based on the observation that software corpora are highly repetitive [18] and localized [36], we exploit a cache language model [36] to locate the defects that are not *natural* in the sense that the sequences of code are not observed frequently either in the training code repository or in the local file.

7. CONCLUSION

The repetitive, 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 as well as syntactic variance in defect occurrence, it is about as cost-effective as the commonly used static bug-finders PMD and FindBugs.

Finally, 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.

8. REFERENCES

[1] *PROMISE '14: Proceedings of the 10th International Conference on Predictive Models in Software Engineering*, New York, NY, USA, 2014. ACM.

[2] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *SIGSOFT FSE*, pages 281–293, 2014.

[3] M. Allamanis and C. Sutton. Mining idioms from source code. In *SIGSOFT FSE*, pages 472–483, 2014.

[4] E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *JSS*, 83(1):2–17, 2010.

[5] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[6] J. C. Campbell, A. Hindle, and J. N. Amaral. Syntax errors just aren't natural: improving error reporting with language models. In *MSR*, pages 252–261, 2014.

[7] C. Catal and B. Diri. A systematic review of software fault prediction studies. *Expert systems with applications*, 36(4):7346–7354, 2009.

[8] T. Copeland. *PMD applied*. Centennial Books San Francisco, 2005.

[9] D. Dahlmeier and H. T. Ng. Grammatical error correction with alternating structure optimization. In *ACL*, pages 915–923, 2011.

[10] D. Dahlmeier and H. T. Ng. A beam-search decoder for grammatical error correction. In *EMNLP*, pages 568–578, 2012.

[11] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse. Untangling fine-grained code changes. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering*, 2015.

[12] FindBugs. `http://findbugs.sourceforge.net/`. Accessed 2015/03/10.

[13] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *ACM Sigplan Notices*, volume 37, pages 234–245, 2002.

[14] C. Franks, Z. Tu, P. Devanbu, and V. Hellendoorn. Cacheca: A cache language model based code suggestion tool. In *ICSE Demonstration Track*, 2015.

[15] M. Gamon. Using mostly native data to correct errors in learners' writing: a meta-classifier approach. In *NAACL*, pages 163–171, 2010.

[16] N.-R. Han, M. Chodorow, and C. Leacock. Detecting errors in english article usage by non-native speakers. *Natural Language Engineering*, 12(02):115–129, 2006.

[17] K. Herzig and A. Zeller. Untangling changes. *Unpublished manuscript, September*, 2011.

[18] A. Hindle, E. Barr, M. Gabel, Z. Su, and P. Devanbu. On the naturalness of software. In *ICSE*, pages 837–847, 2012.

[19] S. Karaivanov, V. Raychev, and M. Vechev. Phrase-based statistical translation of programming languages. In *SPLASH, Onward!*, pages 173–184, 2014.

[20] S. Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 35:400–401, 1987.

[21] K. Knight and I. Chander. Automated postediting of documents. In *AAAI*, pages 779–784, 1994.

[22] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *ICSM*, pages 120–130, 2000.

[23] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Statistical learning of api mappings for language migration. In *ICSE Companion*, pages 618–619, 2014.

[24] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. Lexical statistical machine translation for language migration. In *SIGSOFT FSE*, pages 651–654, 2013.

[25] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen. Migrating code with statistical machine translation. In *ICSE Companion*, pages 544–547, 2014.

[26] Y. A. Park and R. Levy. Automated whole sentence grammar correction using a noisy channel model. In *ACL*, pages 934–944, 2011.

[27] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *ICSE*, pages 432–441, 2013.

[28] F. Rahman, S. Khatri, E. T. Barr, and P. T. Devanbu. Comparing static bug finders and statistical prediction. In *ICSE*, pages 424–434, 2014.

[29] B. Ray, M. Kim, S. Person, and N. Rungta. Detecting and characterizing semantic inconsistencies in ported code. In *ASE*, pages 367–377, 2013.

[30] B. Ray, D. Posnett, V. Filkov, and P. Devanbu. A large scale study of programming languages and code quality in github. In *SIGSOFT FSE*, 2014.

[31] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from "big code". In *POPL*, pages 111–124, 2015.

[32] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *PLDI*, pages 419–428, 2014.

[33] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR*, pages 1–5, 2005.

[34] J. R. Tetreault and M. Chodorow. The ups and downs of preposition error detection in esl writing. In *ICCL*, pages 865–872, 2008.

[35] F. Thung, D. Lo, L. Jiang, F. Rahman, P. T. Devanbu, et al. To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 50–59. ACM, 2012.

[36] Z. Tu, Z. Su, and P. Devanbu. On the localness of software. In *SIGSOFT FSE*, pages 269–280, 2014.

[37] H. Zhang and S. Cheung. A cost-effectiveness criterion for applying software defect prediction models. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 643–646. ACM, 2013.