Will they like this? Evaluating Code Contributions With Language Models

Vincent J. Hellendoorn,[1] Premkumar T. Devanbu,[2] Alberto Bacchelli[1]
1: SORCERERS @ Software Engineering Research Group, Delft University of Technology, The Netherlands
2: Department of Computer Science, University of California, Davis, CA. USA

***Abstract*—Popular open-source software projects receive and review contributions from a diverse array of developers, many of whom have little to no prior involvement with the project. A recent survey reported that reviewers consider conformance to the project's code style to be one of the top priorities when evaluating code contributions on Github. We propose to quantitatively evaluate the existence and effects of this phenomenon. To this aim we use language models, which were shown to accurately capture stylistic aspects of code. We find that rejected changesets do contain code significantly less similar to the project than accepted ones; furthermore, the less similar changesets are more likely to be subject to thorough review. Armed with these results we further investigate whether new contributors learn to conform to the project style and find that experience is positively correlated with conformance to the project's code style.**

I. Introduction

Code review is the manual assessment of source code by human reviewers. Most open source software (OSS) projects, which often heavily rely on contributions of disparate developers, consider code review a best practice both to foster a productive development community [13] and to ensure high code quality [6]. Nowadays code reviews are often mediated by tools (*e.g.*, [14], [15], [26], [20]), which record information that can be mined to better understand the factors influencing the code review process and the acceptability of contributions.

Prior research (in both commercial and OSS settings) has investigated meta-properties stored in code review data, such as size, number of commits, and time to review, and properties of the files that were changed (*e.g.*, [16], [22], [5]). Other research investigated review information to explore the influence of social aspects on software development in OSS communities [5], [11], in particular on the actions of a reviewer when faced with a new contribution [21]. Rigby *et al.* used code review recorded data to triangulate an investigation on the influence of personal aspects of the reviewer, such as experience and efficiency on the code review process [29]. This research has led to valuable insights on how the reviewer's attitude towards the (potentially unknown) contributor affects the process and outcome of code review.

Little is known, yet, of what properties of the *submitted code* influence the review of a changeset and its acceptability. The earliest and most significant insights in this area are those by Gousios *et al.*, who conducted a qualitative study with reviewers of proposed code changes in GitHub [17]. Gousios *et al.* reported that integrators consider style conformance the top factor when evaluating quality of submitted code. Code quality and code style were reported as the top two factors influencing the decision to accept a contribution.

Our goal is to extend these qualitative insights by quantitatively evaluating the influence of stylistic properties of submitted code on both the process and the outcome of code review. To achieve this, we make use of *language models* constructed on source code [18], which were proven to be well-suited to capture stylistic properties of code in the context of a project [1]. Hence, we use this tool to measure how well submitted code *fits in* with the project's code as a whole and analyze how this influences code review.

We conduct our evaluation on projects that use GitHub, the popular social coding platform at the basis of the study by Gousios *et al.* GitHub offers built-in capability for code review by implementing the pull-based development model [16]. In this model, contributors do not have access to the main repository, rather they fork it, make their changes independently, and create a pull request with their proposed changes to be merged in the main repository. The project's core team is then responsible for reviewing and (if found acceptable) eventually merging the changes on the main development line. We consider 22 popular and independent projects on GitHub and analyze a total of 1.4M lines of submitted code, across 6,000 pull requests.

The results of our evaluation show that accepted changesets are significantly more similar to the project at the time of submission than rejected pull requests, supporting that conformance to the code style is a factor that influences code review. We further show that contributions that were subject to more extensive reviews, such as debate regarding a changeset, were substantially less similar to the project's code style. Further investigation supports our finding that highly dissimilar contributions are isolated by project maintainers and receive substantially different treatment during code review. Finally, we show that contributions by novel contributors show a substantial increase in similarity to the project's code as the contributor gains experience.

Structure of the paper. In Section II, we describe the technical background, particularly in the field of natural language processing (NLP) related to language models applied to source code. In Section III, we introduce our research questions and detail the research method we follow. We present our findings in Section IV, and discuss them in Section V. In Section VI we identify a number of threats to the validity of our study. We conclude in Section VII.

II. Background

This work builds on literature on code reviews, both in industrial settings and, more recently, in OSS projects (*e.g.*, [4], [6], [29]), and on recent discoveries in the application of language models to source code [18], [35], [2].

A. Code Reviews

Modern Code Review, as described by Bacchelli & Bird, has gained popularity in several large software development companies as well as in OSS projects and is of increasing interest in recent research [4]. In OSS projects in particular, code review is primarily done by a core group of project maintainers who receive independent contributions from a wide variety of contributors [23]. Bacchelli & Bird find that understanding of the code and the reason for a change is the most important factor in the quality of code reviews [4].

Dabbish *et al.* also study code review in OSS settings, and find that reviewers make a rich set of inferences about contributors based on their activity [11]. Marlow *et al.* find that some of these inferences influence both the nature of the review, and the likelihood of the code being accepted [21]. These works study familiarity and social issues in code review, and are complementary to ours. Tsay *et al.* further expand on this line of work by studying Pull Requests on Github in particular and studying both social and technical aspects of code review from the perspective of project maintainers [34].

Successive papers by Rigby *et al.* study the use of code review in OSS settings, as well as the influence of a number of meta-properties on the efficacy of a code review [28], [30], [29]. They statistically model the relationship of meta-properties on code review outcomes (*e.g.,* interval, efficiency). *Inter alia*, they study how reviewer's expertise *in re* submitted code influences the promptness of the review. Our focus on the properties of the submitted code *per se* complements theirs.

In this paper we focus on code-reviews using the pull-based software development model, which is offered by widely used platforms, such as Github and BitBucket. The pull-based software development model is gaining huge popularity [16], with more than half a million projects adopting it on GitHub alone. Our work uses *language models* to evaluate the *stylistic content* of the code in pull requests and to compare it with the code already existing in the project.

B. Language Models

Hindle *et al.* show that source code has *regularities* that can be captured with statistical models developed for natural language [18]. They find that the local regularity arises primarily from repetitiveness within a project rather than patterns belonging to the programming language. This intra-project regularity suggests that language models could quantify the extent to which new code fits in an existing code base. They evaluate the practical potential of the high regularity that language models found in source code by building a code completion tool and showing that it can significantly improve the default suggestions given by the Eclipse IDE.[1]

[1]https://www.eclipse.org/

Subsequent work investigates the potential of these models in a number of settings, such as their applicability to mining repositories at massive scale [2], and the influence factors such as semantic information [24] and local context [35] on code completion performance. In particular, Allamanis *et al.* investigate whether coding conventions can be extracted automatically from OSS projects using language models [1]. In this sense their work is closely related to ours, as our results suggest the presence of implicit conventions and coding standards in OSS projects. In similar work, Allamanis *et al.* investigate the possibility of mining code 'idioms'; generalized, non-trivial patterns of code that occur frequently across a project [3]. They show a number of examples of such idioms that can be mined using language models, which reveal promise for future work that investigates how new contributors can be supported in writing acceptable code.

1) Computing Language Models: To judge the similarity of a sequence of tokens with respect to a corpus, a language model assigns it a probability by counting the relative frequency of the sequence of tokens in the corpus [32]. In the natural language setting, these models are used for tasks such as speech recognition, machine translation, and spelling correction [10], [7], [8].

We can write the probability of a sequential language fragment s of N tokens $w_1 \dots w_N$ as:

$$\begin{aligned} p(s) &= p(w_1) \cdot p(w_2|w_1) \cdots p(w_N|w_1 \cdots w_{N-1}) \\ &= \prod_{i=1}^{N} p(w_i|w_1 \cdots w_{i-1}) \end{aligned}$$

Each $p(w_i|w_1 \cdots w_{i-1})$ can then be estimated as:

$$p(w_i|w_1 \cdots w_{i-1}) = \frac{c(w_1 \cdots w_i)}{c(w_1 \cdots w_{i-1})} \tag{1}$$

where c means 'count'.

However, as the context (or history) of a token lengthens, it becomes increasingly less likely that the sequence has been observed in the training data, which is detrimental to the performance of the model. N-gram models approach this problem by approximating the probability of each token based on a context of the last n tokens only:

$$p(s) = \prod_{i=1}^{N} p(w_i|w_1 \cdots w_{i-1}) \approx \prod_{i=1}^{N} p(w_i|w_{i-n+1} \cdots w_{i-1})$$

Intuitively, this approximation states that a token is only influenced by the preceding n tokens (formally assuming a Markovian property of language). Estimating the probabilities in an n-gram model is analogous to Equation (1), with the counts only considering the last n words. The choice of n is important: Short contexts are observed often but hold little information, whereas longer contexts hold more information but are rarely seen. Good *smoothing methods* make use of these qualities by combining models of different length [10]. Smoothing methods are used to guarantee that unseen token

sequences are not assigned zero probability, which would be destructive in applications such as speech recognition.

Although research in the Natural Language Processing (NLP) community has shown that this approximation discards a significant amount of information [31], [9], [12], n-gram models are widely used as they can be easily generated from the training data while providing powerful models [25].

2) Models on Source Code: Hindle *et al.* were the first to show that these models capture regularity in source code, and showed that source code is even more predictable than typical natural language corpora [18]. They define a 'sentence' as a program, composed of allowable program tokens.

3) Measuring Performance: The geometric mean of the probabilities assigned to each token in a sentence is at the core of the most common metrics used in NLP. It is well suited to deal with the widely varying range of probabilities that are typically assigned by a language model. Given a sentence s of length N and the probabilities for each word in s: $p(w_i|h)$ (where h denotes a context of some length), the geometric mean is computed as follows:[2]

$$\sqrt[N]{\prod_{i=1}^{N} p(w_i|h)} = 2^{\frac{1}{N}\sum_{i=1}^{N}\log_2(p(w_i|h))} \quad (2)$$

In NLP, the most commonly used metrics to measure the performance of a language models p are cross-entropy (H_p, measured in bits/token) and perplexity (PP_p):

$$\begin{aligned} H_p(s) &= -\frac{1}{N}\sum_{i=1}^{N}\log_2(p(w_i|h)) \\ PP_p(s) &= 2^{H_p(s)} \end{aligned}$$

The perplexity of a string is the inverse of the geometric mean of the probabilities of its tokens. The cross-entropy (also *entropy*) is the binary log of the perplexity; it was found to have a strong, negative correlation with predictive quality in NLP applications (*e.g.*, in speech recognition) [10]. Previous work in the application of NLP techniques to source code has consistently reported cross-entropy (lower results imply higher predictive quality) and we follow this example.

III. Methodology

The main goal of this research is to quantitatively evaluate the influence of stylistic similarity of submitted code (to existing code) on both the code review process and outcome. In prior work, we have shown that language models capture similarity within projects and differences across projects [17]; we have also found coarser differences (across/within application domains) and finer ones (between files within the same project) [33]. In this paper, we study the value of language models to gauge the relation between the acceptability of submitted code and its naturalness: its similarity to a project's code. Therefore, we structure our goal around the following research questions, which we iteratively refined while analyzing our results:

- RQ1: Are rejected PRs less natural than accepted ones?
- RQ2: Are more debated PRs less natural?
- RQ3: Does reviewing affect the entropy of debated PRs?
- RQ4: Does contributions' naturalness grow with experience?

In the first place, we seek to answer whether the outcome of code review is correlated with statistical properties of the submitted source code. Then, we study correlations with the process of the code review by looking at submissions that were subject to debate before being decided. In particular, we contrast these with submissions that were accepted with little feedback. Finally, we use the found correlations to investigate other properties that influence code reviews, such as author experience and goal of the contributions.

A. Experimental Setup

We focus our research on pull requests (PRs) submitted to Github projects. This has several advantages: Github data is readily accessible, both via an API[3] and through the `git` command-line tool.[4] In particular, this allows us to revert a copy of a project to its exact state at the time of any code review, which is necessary for our model to work. Future work may study the existence of similar patterns in industrial code bases and other OSS projects.

Our approach works in three steps. (i) For each project we extract the lines added by each pull request, as well as the lines added by each commit contained within that pull request. These lines constitute the *test set*. (ii) For each PR, we extract the *training set* from a copy of the project that is reverted to the project's state when the PR was submitted. (iii) We train a language model on all `.java` files in the training set and test it on the lines of `java` code in the test set. The output of the language model is an entropy score, reflecting the similarity of the pull request to the code base at the time of submission.

Additionally, we used the Github API to extract a number of properties for each pull request, such as author, commit message and number of (review) comments. With this information we analyze factors that influence the results and discover less obvious correlations with entropy. In the following, we detail the approach used in selecting and collecting the data, the procedure used to extract our training corpus and test set, and finally the criteria by which we processed the data.

B. Project selection

We use two criteria for selecting the projects: (1) The project must primarily contain `java` code, and (2) the project must make active use of pull requests. The first requirement is a matter of choice: the language models described in

[2]The right-hand (equivalent) form in Equation (2) avoids rounding problems that typically arise with a product over many probabilities.

[3]http://api.github.com/

[4]http://git-scm.com/

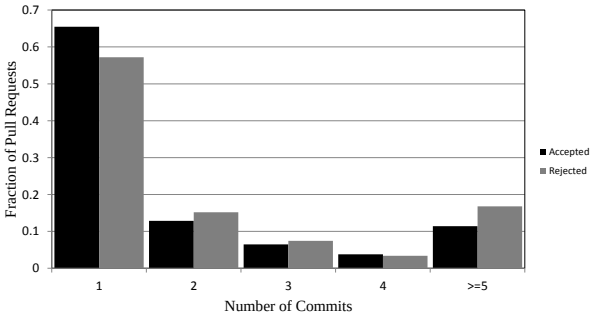

Figure 1. Distribution of commits over pull requests (accepted and rejected) in the selected projects.

section II-B are fairly agnostic w.r.t. the language and we discuss extensions to other languages in Section V. The second requirement relates to the findings by Gousios *et al.* [16], who reported that only a small fraction of projects on Github make active use of pull requests. Among these projects, most have used no more than 20 pull requests in total (25 among projects with more than one contributor [19]). They also found that a few projects account for a very large number of pull requests.

In light of the diverse usage of the pull-based software development model across Github, we define 'making active use of pull request' according to three criteria. Primarily, we looked for projects that had 26 or more closed pull requests. This guarantees that we consider only projects belonging to the 5% most active users of the pull-based model. Secondly, we omitted projects in which the majority of contributions were made outside the pull-based model. In most of the resulting projects, 80% or more of all commits were part of a pull request. Finally, we omitted projects in which no rejected PRs were found that qualified as *debated*, as described in section III-E.

The projects were selected based on these three criteria from among the `Java` projects that are currently popular, as reported by Github's 'Trending Repositories' feature. Following these criteria, we totaled 22 projects, containing approximately 10,000 pull requests and over 20,000 commits. We then filtered out the PRs that did not include `Java` code and PRs that contained an inordinate number of additions (a fair number of PRs submitted several thousands of lines, typically either by mistake or by merging two branches), which left a little over 7,500 PRs and 13,500 commits. Of these, 1,634 PRs received no comments of any kind before being closed, typically by the original author. As we are concerned with code that is *reviewed*, we omit these from our final dataset in all results except in the evaluation of RQ1, to avoid a potential confound. Figure 1 shows the distribution of the number of commits in accepted and rejected PRs. The majority of PRs, accepted or rejected, consist of a single commit, but the distribution is skewed in that a small number of PRs contain a large number of commits. The largest PRs contain over 40% of the total commits.

The selected PRs contain a total of 1.4 MLOC submitted to 22 popular Github projects over a period of 2-40 months. Table I reports the selected projects with the number of PRs linked to the projects in ascending order. The project names are reported with the corresponding user account, as many forks exist. The considered projects were aged between two months and eight years[5] and contained between 56 and 1009 PRs (after filtering) from between six (Physical-web) and 234 (Jenkins) distinct contributors (median: 49). This corpus represents a diverse array of size, age and diversity of contributors and provides a large enough base to report statistically significant results on.

C. Data extraction

For each project, we used the Github API to create a list of indices, corresponding to all PRs that were closed at that time. These indices were collected between Sep 9, 2014 and Nov 3, 2014. Additionally, we used the Github API on each PR to extract properties such as author, commit message, (review)comments, and size.

In the following we describe the process we used to generate entropy scores for each pull request (illustrated in Figure 2). Step 1: we use the information on the lines that were added and deleted in each PR (provided by GitHub in text format[6] through a URL) to create a test set for each PR, which consists of the lines added in the submission. As can also be seen in Figure 2, modified lines in this `diff` are represented by a removal of the old line and addition of the new line. Hence, by constructing our test set of all added lines, we include both newly added code and lines that were modified by the pull

[5]We found no pull requests from before 2010, when Github improved the pull request mechanism and UI: https://github.com/blog/712-pull-requests-2-0

[6]*e.g.*, https://github.com/ReactiveX/RxJava/pull/1094.diff

Table I
CONSIDERED PULL REQUESTS AND COMMITS BY PROJECT. THESE NUMBERS INCLUDE PRS THAT WERE ISOLATED BECAUSE THEY RECEIVED NO COMMENTS.

Project (user/projectname)	PRs	Commits	Category
google/physical-web	56	102	
thinkaurelius/titan	68	185	
excilys/androidannotations	81	240	
JakeWharton/ActionBarSherlock	126	350	
Elasticsearch/Elasticsearch	159	209	
loopj/android-async-http	186	358	
square/picasso	209	303	
netty/netty	215	720	small
realm/realm-java	232	1,575	
WhisperSystems/TextSecure	261	445	
eclipse/vert.x	263	971	
square/retrofit	269	384	
dropwizard/metrics	301	752	
junit-team/junit	319	1,040	
nathanmarz/storm	331	548	
jbosstm/narayana	384	519	
Catrobat/Catroid	459	2,096	
square/okhttp	594	680	
reactivex/rxjava	756	2,148	large
jenkinsci/jenkins	815	1,933	
spring-projects/spring-integration	909	973	
facebook/presto	1009	2,453	

Figure 2. An overview of the setup used to acquire entropy scores for pull requests.

request. Furthermore, we derive the test set for each commit part of the PR. Here, the test set contains the added and modified lines that the changeset contained up to that point in time. Step 2: we extract the training set (used to estimate the language model) from a local clone of the project. Using the `git reset` command, we revert the local copy of the project to its state when the PR is submitted. Step 3: we use the extracted testing set to build a language model out of the Java code in the resulting project, using the techniques outlined in Section II-B. We use the test set as input to the language model. The output is an entropy score reflecting how similar the submitted code was to the project at submission time.

D. Analysis

To answer our research questions, we need to know whether the pull request is eventually accepted. Gousios *et al.* find that a significant fraction of PRs in GitHub are (partially) merged in ways that are different from the official merge tool [16]. As a consequence, the Github API shows a significant fraction of pull requests as PRs as rejected, even in projects that rely heavily on pull requests. To better identify whether a pull request was merged, Gousios *et al.* develop four heuristics to find if the submitted code was merged into the code base via different means. In this study, we use the same heuristics (we found an acceptance ratio of 79% across the 26 projects, comparable to the results of Gousios *et al.*).

To study the effect of entropy on the *process* of a code review, we further divide the PRs into the categories *debated* and *undebated*. We define a PR as debated if it received at least three comments during review (review + discussion comments), following Gousios *et al.* who find that a PR on average receives 2.77 comments (review + discussion comments) and who suggest that a small fraction of pull requests receives the majority of comments [16]. For the sake of comparison, we furthermore define a PR as *undebated* if it received at least one comment but was not *debated*. We find that approximately 21% of all pull requests received no comments before being decided, 38% received feedback but not more than two comments, and 41% was debated.

We furthermore intend to investigate the influence of authorship on the entropy of contributions. In order to derive statistically significant results related to authorship, we restrict our focus to the ten largest projects, as these have sufficiently many distinct authors and contributions per author. Furthermore, in each project's history there was a small number of contributors (typically three or less) who were responsible for the vast majority of contributions. These authors may therefore have contributed so much that statistical similarities which we find between the project and their PRs may be due to them having shaped the project in the past. Hence, we restrict ourselves to PRs from authors who have submitted less than 20 PRs in the entire project history, which guarantees that all remaining authors have contributed no more than 5% of the project's code. Setting this cut-off to 10 PRs produced similar results but substantially reduced the number of data points.

Observing that the median number of contributions per author among the remaining PRs was 3, we set the threshold for *experienced* at having contributed at least three times before. Here too, similar thresholds yielded comparable results.

E. Evaluation

We employ language models to evaluate aspects of submitted code that impact the code review process. To this end, we typically divide the studied pull requests into two categories (*e.g.*, accepted and rejected) and compute the mean entropy for both categories in every project. We then pair the results per project and compare the outcome between projects using both a paired t-test and a, non-parametric, Wilcoxon signed rank test [33]. In general, we found that the tests yielded very comparable significance levels, suggesting that the data

is predominantly normally distributed. Where one test yielded a lower level of significance, we make mention of this and report the lesser value. Furthermore, to quantify the effect of the difference in means between two categories, we compute Cohen's D [33] on the difference scores of the paired data.

The *ecological fallacy* states that findings at an aggregated level (here: projects) may not apply to a disaggregated level (here: individual PRs); this was recently found to apply to empirical software engineering research as well [27]. Where applicable, we therefore validate our findings at the level of individual PRs by reporting results of a Wilcoxon rank sum test across the complete corpus of PRs (disregarding the projects). We generally aggregate to the project level because typical entropy scores of PRs differ substantially *between* projects (up to 1.5 bits).

In general, we found that results that held overall also consistently held on the largest projects, but not necessarily on the smaller projects (see Section VI for a discussion on this). We furthermore found that the effect size of significant effects differed substantially between these two categories (in favor of the larger projects). To avoid a potential size confound, we divide the selected projects into two categories: *large* and *small*, where the large projects contain approximately 1/3rd of the projects and 2/3rd of the pull requests. The category is shown in the last column of Table I. Aside from reporting results for the average across all projects, when appropriate we report results for the average within these categories.

IV. Findings

In this section, we report the results of our analysis.

RQ1: Are rejected PRs less natural than accepted ones?

For each of the 22 projects studied, we averaged the entropies over all accepted and rejected pull requests per project. The mean entropies of accepted and rejected PRs are 4.18 and 4.35 bits/token respectively (note that this measure, is log scaled; corresponding perplexities are 18.12 and 20.4, respectively). A paired t-test confirms that the average entropy of the rejected PRs is significantly higher than that of accepted PRs ($p < 0.01$) and computing the effect size (Cohen's D on the paired data) reveals a moderate effect (0.61). This result is consistent across the corpus of individual PRs (disregarding the projects), where the mean entropies of accepted and rejected PRs were 4.14 and 4.44 respectively ($p \ll 10^{-5}$).

Next, we divide the projects into the groups *large* and *small* as explained in Section III-E. The resulting statistical difference between accepted and rejected PRs can be seen on the first row of Table II. A paired t-test confirms that rejected PRs are significantly less natural in the group of larger projects ($p < 0.05$) but cannot confirm this difference among smaller projects ($p > 0.1$). Furthermore, the effect size between accepted and rejected pull requests is substantially larger when considering just the bigger projects. A similar result was found on the corpus of individual PRs (large projects: $p \ll 10^{-5}$, small projects: $p < 10^{-3}$). These results suggest that, particularly among more established projects, the relative similarity of submitted code to a project as a whole has a measurable influence on the outcome of a code review.

Additionally, we investigated the influence of the PRs that were removed because they did not receive any comments before being decided (1,634 PRs, 93% accepted) and found that the inclusion of these PRs did not harm the previous results (in fact, the significance level on *small* projects came within the $p < 0.1$ range). Additionally, we noted that these unreviewed PRs have significantly lower entropies than reviewed ones ($p < 0.05$, moderate effect). Given these results, we may expect to see a difference in entropy with more extensively discussed PRs as well.

RQ2: Are more debated PRs less natural?

We separate *debated* (3,201 PRs, 80% accepted) and *undebated* PRs (2,958 PRs, 85.7% accepted), as defined in section section III-E. If less natural code is more likely to be rejected, we may also expect that less natural contributions generate more discussion during review, so that debated PRs would have significantly higher entropies (*i.e.*, be significantly less natural) than undebated PRs.

We particularly expect (eventually) accepted PRs that triggered debate to be less natural than PRs that were accepted quickly. We do not necessarily expect a comparable result for rejected PRs, however: code that is rejected with little feedback might be the least natural contributions of all.

We first investigate the previous result within the groups of debated and undebated PRs; rows 2 and 3 of Table II report the results. We first repeat the previous investigation on undebated PRs and find a slightly stronger version of the results for RQ1, particularly among smaller projects (paired t-test: $p < 0.05$, signed rank test: $p < 0.1$). We conduct the same analysis among debated PRs and find that, although no significant difference was found overall, there is evidence of the previous result among debated PRs in large projects. Results on the corpus of individual PRs are comparable: For undebated PRs the results from Section IV hold; for debated PRs, a significant difference was found both overall and on the large projects ($p < 10^{-4}$) but none on the small projects ($p > 0.1$).

Comparisons 4 and 5 in Table II compare the results *between* debated and undebated PRs, first for (eventually) rejected PRs and then for accepted PRs. Here we find the largest difference between more established and smaller projects. Both rejected and accepted PRs were significantly more entropic when subject to debate in large projects, but on small projects only the latter result held and only with moderate effect.

The above results show several distinctions between undebated and debated pull requests: a) among the former, low entropy PRs were substantially more likely to be accepted, whereas among the latter we only found some evidence of this phenomenon on large projects, b) debated pull requests as a whole had substantially higher entropies than undebated ones, particularly those that were eventually accepted. The correlation between debate and entropy appears to be at least

Table II
SIGNIFICANCE AND EFFECT SIZE (COHEN'S D) OF ENTROPY DIFFERENCE BETWEEN DIFFERENT CATEGORIES OF PULL REQUESTS. $D < 0.5$ IS CONSIDERED *"small"* EFFECT; $D < 0.8$ *"medium"*; OTHERWISE *"large"* . BLACK p-VALUES REFLECT SIGNIFICANCE AT LEAST AT THE $p < 0.05$ LEVEL, ORANGE VALUES: ONLY EVIDENCE OF SIGNIFICANCE FOUND ($0.05 < p < 0.1$) AND RED VALUES: NO SIGNIFICANT CORRELATION FOUND.

Comparison			All projects		Large projects		Small projects	
id	PR type	Test on entropy difference	D	p	D	p	D	p
1	Overall	Accepted < Rejected	0.61	<0.01	1.05	<0.05	0.42	>0.1
2	Undebated	Accepted < Rejected	0.64	<0.01	1.04	<0.05	0.57	<0.1
3	Debated	Accepted < Rejected	0.11	>0.1	0.83	<0.1	-0.22	>0.1
4	Rejected	Undebated < Debated	0.06	>0.1	1.11	<0.05	-0.1	>0.1
5	Accepted	Undebated < Debated	0.65	<0.01	1.18	<0.05	0.63	<0.05

as large as that between acceptability and entropy. This raises a question: does entropy still play a role *among controversial* PRs? Or do our results primarily reflect that PRs with low entropy are likely to be accepted and receive little review? We study this in the following question.

RQ3: Does reviewing affect the entropy of debated PRs?

We start by analyzing the entropy of the commits that compose PRs. In particular, we investigate PRs that were revised under debate. To this end, we collected 11,600 commits that were part of PRs that were both debated and revised with subsequent commits (these PRs had an average of approximately 10 commits per PR). For each commit, we calculated the entropy of the PR after that commit with respect to the core project; then, we computed the average change in entropy for accepted and rejected PRs between the first and last commit. The results are shown in Figure 3.

A t-test confirms that both accepted and rejected PRs increase slightly in entropy during revision ($p < 0.05$). By manual inspection, we observed that contributors of debated PRs were asked to add novel code to their PRs (*e.g.*, test cases). As new code is in general less predictable, this may have worked as a confound. No significant difference in entropy increase was found between accepted and rejected PRs.

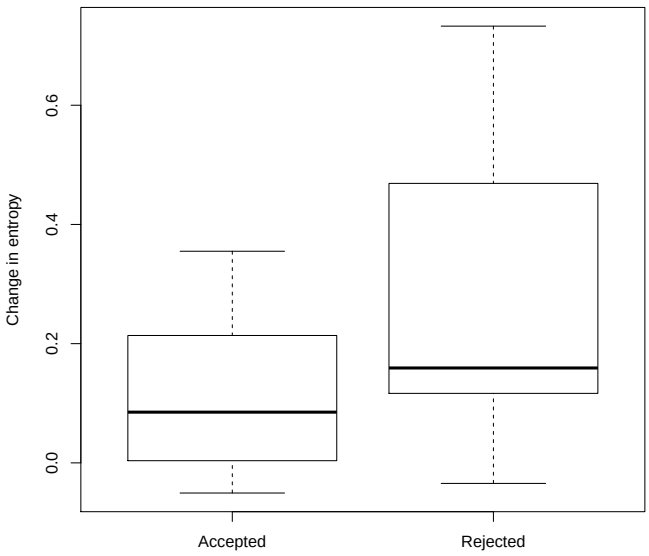

Figure 3. Change of entropy (in bits/token) of accepted and rejected pull requests following debate during code review.

To rule out confounds related to novel code, we then investigated changes at file level between successive commits. For each review comment (*i.e.*, a comment on a part of the submitted code), we compared the entropy of the file under discussion before and after revision. We found no significant evidence that the discussed files decrease in entropy during code review (neither among accepted nor rejected PRs), even when restricting ourselves to revisions in which the number of changed lines remained the same. We intend to refine this criterion in future work.

In the following question we study whether new contributors learn the project's style conventions as their experience grows.

RQ4: Does contributions' naturalness grow with experience?

Given the experimental setup, it can be argued that language models effectively measure the similarity in implicit coding style between a pull request and the existing code base. Hence, we expect the coding style of the author to have a significant impact. In particular, we expect novel contributors to write code that is more "fluently similar" (with respect to the project) than experienced contributors.

We first evaluate whether a difference in naturalness exists between PRs from more and from less experienced authors, where we use the definition of *experience* as in Section III-D. We only include authors who contributed four or more times in the project's history and therefore have contributions both before and after they qualified as *experienced*. We computed the difference in entropy of their contributions before and after they pass the 'experience' threshold. Applying the above restrictions left us with 1,432 PRs from 127 distinct authors.

The resulting distributions of entropies are shown in Figure 4. A t-test confirms that the entropy of contributions is significantly lower among PRs from contributors with prior experience than among PRs from contributors with little to no prior experience ($p < 0.01$). We furthermore found a large effect ($D = 1.11$). Further investigation revealed that the largest difference exists between contributors with one or two prior contributions and contributors with three or more prior contributions. Contributors with no prior experience showed a large degree of variation in entropy scores, which may be caused by this group often submitting relatively simple changesets upon their first contact with a project.

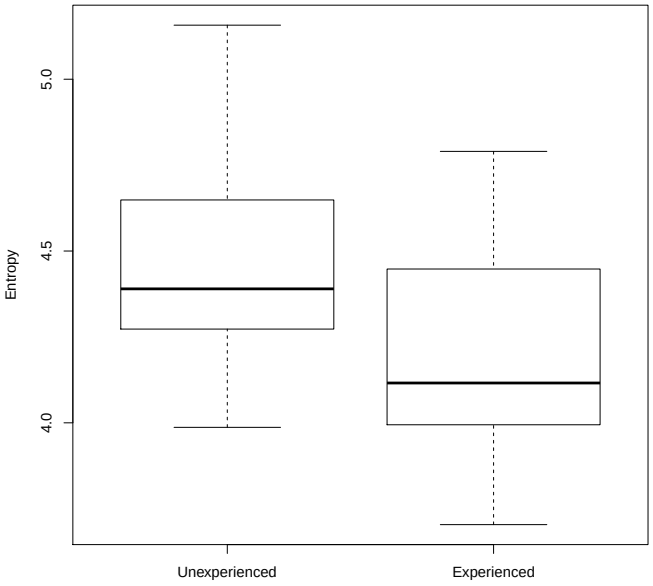

Figure 4. Distribution of contribution entropy among contributors with and without experience.

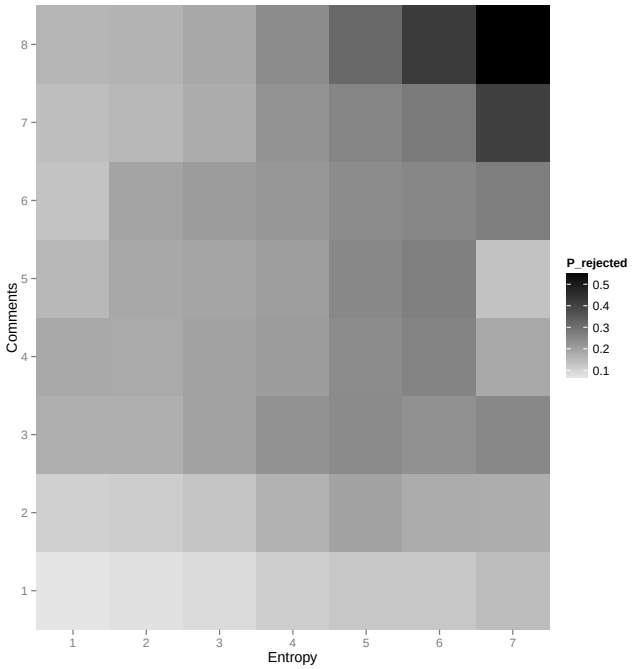

Figure 5. Probability of a PR to be rejected, by comments and entropy

V. Discussion

A variety of previous research has studied the question why certain contributions to OSS projects are rejected or not accepted until after revision. Among others, this has produced anecdotal evidence that conformance to the project's code style is considered a strong indicator of quality. The aim of this research was to quantitatively evaluate this aspect of code review using language models.

In the first research question we asked whether rejected pull requests have a higher entropy with respect to project's source code than accepted ones. We found that the statistical similarity between submitted code and the core project is both measurable by language models and reveals statistically significant differences between accepted and rejected pull requests. Code in accepted pull requests has significantly lower entropy to project code than code in rejected ones, across all projects. This result corroborates the qualitative findings by Gousios *et al.* [17] on the importance of style conformance when reviewers evaluate contributions.

We also found that the effect size is larger and more significant in larger projects than smaller ones. This finding is in line with earlier work by Tsay *et al.*, who found that well-established projects are more conservative in accepting pull requests [34]. Nevertheless, projects such as the Mozilla project[7] and the Linux Kernel[8] are both several times older and larger than most projects studied in this work, making them very interesting candidates to determine if and how our results generalize to other cases.

In the second research question we investigated whether contributions that are more debated are less similar to the project source code. We found that contributions that are directly accepted without discussion are significantly less entropic than debated ones, regardless of their eventual acceptability. Furthermore, the most entropic PRs were those that were both debated and eventually rejected, suggesting further that highly unconventional code is more likely to attract the attention (and face the disapproval of) project maintainers.

Figure 5 shows the probability of a pull request being rejected considering the number of comments it receives and its entropy. As can be seen, an increase in entropy almost consistently increases the likelihood of rejection, whereas when considering comments alone, we primarily see a distinction between PRs that received no more than two comments and those that received three or more comments. A larger number of comments is only correlated with a greater probability of rejection among PRs with a very high entropy. Furthermore, whereas comments can be seen as an artifact of the code review itself, the entropy of a changeset can be computed on submission. A ripe opportunity for future research would be to study whether entropy can be used to direct reviewers' attention to code that is more likely to generate debate or to be rejected.

The third research question dives deeper into the context of debated pull requests and investigates whether the debated contributions decrease their entropy due to the review. We found no significant evidence of such a decrease. The results suggest that, while reviewers are quick to recognize and approve low entropy PRs, the naturalness of submitted code plays less of a role during more extensive code review. By manual inspection, we found that extensive code review was

[7]https://www.mozilla.org/
[8]https://www.kernel.org

often concentrated around (changes of) functionality of the submitted code.

The aforementioned results do not preclude that code under review is asked to better conform to the project's coding conventions. Code review comes in many shapes and we found many instances where novel contributors were asked to better conform to coding standards (often besides requests for functionality improvement) during code review. In our fourth research question we investigated a possible effect of these comments: whether novel contributors tend to adhere more to project's style as their experience with the project grows. Our results indeed confirmed that contributors write code that is more similar over time. Nevertheless, we did not analyze further whether the project conformance comes naturally from a better knowledge of the project rather than comments received in code reviews. Studies can be designed and carried out to investigate this phenomenon.

Although we considered projects coming from a variety of communities we limited ourselves to OSS systems and the pull-based development model. We did not verify our results with industrial projects or on systems that make use of different code review models: these settings deserve additional study and may provide new insights on style conformance.

Throughout RQ2, 3 and 4, we have investigated a number of factors that influence both acceptability and entropy of PRs. Previous work provided evidence that many other factors influence the eventual acceptability of a contribution (*e.g.*, whether the contribution adds novel code or modifies existing code, personal relationship with project maintainers). Although a full investigation of the relationship with other factors is beyond the scope of this work, it is of particular importance to investigate the type of PRs that are likely to be rejected. For instance, it might be that the observed correlation is caused by one type of PRs that has both high entropy and a high rate of rejection. If this is the case, the language models may be mirroring an underlying effect: the type of the PR. As an initial step in this direction, we investigated the influence of the type (or *intent*) of PRs on entropy and acceptability: We classified all PRs into five categories ('Bug fix', 'Performance', 'Styling', 'Quality assurance', and 'Other'), by matching a set of manually derived keywords against the submission message. We found that the gap between accepted and rejected PRs was significant in all categories except among bug-fixes (only evidence of significance), although the mean entropies varied substantially between categories. The rate of acceptance was similarly consistent across the categories. PRs related to performance improvement (*e.g.*, memory usage, speed) had substantially higher entropies than other PRs, with bug-fixes being the second highest. This meets our expectations, as these PRs (particularly the first category) generally contribute novel code to a project. Styling PRs (*e.g.*, refactorings, enforcing coding conventions), on the other hand, showed the lowest entropy scores, as well as the largest gap between accepted and rejected PRs. This matches results from previous work, which found that low-entropy refactorings likely constitute acceptable contributions to OSS projects [1]. These results suggest that (a) the observed difference in entropy between accepted and rejected PRs is robust with respect to different types of PRs, and (b) language models may be most powerful when evaluating contributions that modify existing code. These results, and the combination of entropy with other factors that influence code review, warrant more extensive evaluation in future work.

Our study also effectively used language models as a tool to automatically compare new code to an existing code base. This opens a number of possible future applications: language models may be able to derive coding conventions corresponding to code that a new contributor is writing (*e.g.*, code idioms [3]), to save effort on both the developer's and the reviewer's end. Moreover, as previously mentioned, language models can be used to direct reviewers' attention to code that is more likely to generate debate. Finally, we hope to inspire new research by showing the effects that authorship and intent have on entropy scores, and the practical ramifications of these effects.

VI. Threats to Validity

a) Internal Validity: Assumptions we made in the selection of the projects may harm the internal validity of this study. The manual selection of the projects may have led to an unrepresentative sample of Github projects. To avoid this confound, we included every project listed on Github's "Trending Repositories" page and satisfied the criteria. The projects selected were "trending" in the last month and collected in three turns between September and November 2014.

The three criteria we used to filter the potential projects are another potential threat to validity. The first criterion concerns the use of 26 or more pull requests. This number is selected based on previous work by Kalliamvakou *et al.*, which found that 95% of the projects that have used pull requests for actual collaboration used no more than 25 pull requests in the project history [19]. In our eventual test set, (after filtering out unreviewed changes) the smallest project had 37 PRs and all others had more than 50.

The second criterion concerns the use of PRs as the primary source of contributions. On Github, the alternative to PRs is to commit directly to the main repository, something that is permitted to the maintainers of the project. Hence, we chose to identify projects as 'actively using the pull-based software development model' by taking the ratio of the count of the number of commits that were part of a PR to the total number of commits. Here, we took special care to count only commits made after the project adopted the pull request model, as some projects started before this feature was added. We excluded projects for which this ratio is less than the 0.5. Although most projects in our study scored 0.8 or higher, some project owners used direct commits more frequently than PRs. This may affect the results on author experience, although we found no substantial difference in these projects' results when compared with the remaining test set.

In a study using language models, it is critical that the training set and test set do not overlap. In the context of this study,

that means that the code base must be fully reverted to a point in time before a PR has been accepted or rejected, otherwise the model would be biased towards accepted PRs. Using the combination of the 'base SHA' from the Github API and the `git reset --hard` command should guarantee this, and we manually verified the correctness of this combination for a small number of cases. We furthermore ran a series of tests under more strict conditions. We reverted the repository to its state before the first pull request and then for each PR from oldest to newest, updated the repository only to the moment the project was forked. By effectively testing on the PRs in reverse order, we found that our earlier results still held.

Finally, it is possible that the observed correlation of the difference in entropy with PR acceptability and debate is spurious, mirroring another factor of influence. We discussed and minimized an important threat of this category in Section V, namely that certain types of PRs may both have high entropy and high rates of rejection. We found that all types of PRs display the same phenomenon (although to different extents), despite having different overall entropy scores, suggesting entropy is a general factor of influence in PR review.

b) External Validity: The current study focuses only on popular Java projects that use the Github OSS platform. The aspect of popularity can be seen as a necessity for this approach to yield significant results. However, a number of potential threats to the external validity of the study follow. (1) The results may not be representative for other languages on Github (see also Section V). (2) The results may not be representative for other OSS platforms. In particular, none of the projects in this study made use of pull requests for more than 40 months (when the feature was improved on Github). Other OSS projects, such as Eclipse, Mozilla, or many Apache projects, are both older and larger. (3) The results may not hold for industrial projects that make use of (modern) code review. Due to the financial concerns that play a role in such settings, investigating the potential of language models in identifying code style violations may lead to different results.

In general, our results held both on the entire set of projects and on the largest projects. For instance, we evaluated whether a significant difference in entropy exists between accepted and rejected PRs *within the individual projects* using a Benjamini-Hochberg test. We found a significant difference on 5 out of the 7 large projects ($p < 0.05$), but only on 1 out of the 15 small projects. This may in part be explained by the smaller projects having insufficient data points to achieve significant results. However, a number of the results presented in Table II held with significance and strong effect on the group of large projects but did not hold at all on the collection of small projects (as defined in Section III-E). This suggests a size confound, where the larger (and generally also older) projects have a stronger sense of coding conventions, which is also in line with the results of a recent study by Tsay *et al.* [34]. We have dealt with this threat to the validity of our study by dividing the projects into the categories *large* and *small* and reporting results for both of these categories when applicable.

VII. Conclusion

Source code contributions to OSS projects are evaluated through code reviews before being accepted and merged into the main development line. Surveying core members of popular projects on GitHub, Gousios *et al.* found that reviewers perceive conformance, in terms of style and quality, as the top factor when evaluating a code contribution [17]. In this paper, we extend this qualitative insights and use language models to quantitatively evaluating the influence of stylistic properties of code contributions on the code review process and outcome.

By analyzing 22 popular OSS projects on GitHub, totaling more than 6,000 code reviews, we found that (1) accepted code is significantly more similar to the project from a language model perspective than that rejected, while (2) highly dissimilar contributions receive a different treatment in code review, and (3) more debated ones are significantly less similar. Finally, (4) the contributed code shows a substantial increase in similarity as the contributor gains experience.

In this paper we make the following main contributions:

1) The first application of language models for an automated evaluation of properties of the *submitted code* on the code review outcome and process.
2) An extensive quantitative analysis on 22 OSS projects of the effect of similarity, from a language model perspective, between contributed code and project code on code review outcome and code review process.
3) A discussion of the implications of our findings with recommendations for future opportunities of research.

References

[1] M. Allamanis, E. T. Barr, and C. Sutton. Learning natural coding conventions. *arXiv preprint arXiv:1402.4182*, 2014.

[2] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 207–216. IEEE, 2013.

[3] M. Allamanis and C. A. Sutton. Mining idioms from source code. *CoRR*, abs/1404.0417, 2014.

[4] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of ICSE 2013 (35th ACM/IEEE International Conference on Software Engineering)*, pages 712–721, 2013.

[5] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. The influence of non-technical factors on code review. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 122–131. IEEE, 2013.

[6] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of MSR 2014 (11th Working Conference on Mining Software Repositories)*, pages 202–211, 2014.

[7] P. F. Brown, V. J. D. Pietra, S. A. D. Pietra, and R. L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational linguistics*, 19(2):263–311, 1993.

[8] A. Carlson and I. Fette. Memory-based context-sensitive spelling correction at web scale. In *Machine Learning and Applications, 2007. ICMLA 2007. Sixth International Conference on*, pages 166–171. IEEE, 2007.

[9] C. Chelba and F. Jelinek. Exploiting syntactic structure for language modeling. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics-Volume 1*, pages 225–231. Association for Computational Linguistics, 1998.

[10] S. F. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics*, ACL '96, pages 310–318, Stroudsburg, PA, USA, 1996. Association for Computational Linguistics.

[11] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1277–1286. ACM, 2012.

[12] D. Filimonov and M. Harper. A joint language model with fine-grain syntactic tags. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 3-Volume 3*, pages 1114–1123. Association for Computational Linguistics, 2009.

[13] K. Fogel. *Producing Open Source Software*. O'Reilly Media, first edition, 2005.

[14] Gerrit. https://code.google.com/p/gerrit/. Accessed 2015/02/12.

[15] GitHub. https://github.com/. Accessed 2015/02/12.

[16] G. Gousios, M. Pinzger, and A. van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of ICSE 2014 (36th ACM/IEEE International Conference on Software Engineering)*, pages 345–355. ACM, 2014.

[17] G. Gousios, A. Zaidman, M.-A. Storey, and A. v. Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering*, ICSE 2015, 2015. To appear.

[18] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu. On the naturalness of software. In *Proceedings of ICSE 2012 (34th International Conference on Software Engineering)*, pages 837–847, 2012.

[19] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 92–101. ACM, 2014.

[20] N. Kennedy. Google Mondrian: web-based code review and storage. http://www.niallkennedy.com/blog/2006/11/google-mondrian.html, 2006. Accessed 2015/02/12.

[21] J. Marlow, L. Dabbish, and J. Herbsleb. Impression formation in online peer production: activity traces and personal profiles in github. In *Proceedings of the 2013 conference on Computer supported cooperative work*, pages 117–128. ACM, 2013.

[22] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 192–201, New York, NY, USA, 2014. ACM.

[23] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(3):309–346, 2002.

[24] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. In *Proceedings of ESEC/FSE 2013 (9th Joint Meeting on Foundations of Software Engineering)*, pages 532–542, 2013.

[25] A. Pauls and D. Klein. Faster and smaller n-gram language models. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pages 258–267. Association for Computational Linguistics, 2011.

[26] Phabricator. http://phabricator.org/. Accessed 2015/02/12.

[27] D. Posnett, V. Filkov, and P. Devanbu. Ecological inference in empirical software engineering. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 362–371. IEEE Computer Society, 2011.

[28] P. C. Rigby and D. M. German. A preliminary examination of code review processes in open source projects. *University of Victoria, Canada, Tech. Rep. DCS-305-IR*, 2006.

[29] P. C. Rigby, D. M. German, L. Cowen, and M.-A. Storey. Peer review on open-source software projects: Parameters, statistical models, and theory. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):35:1–35:33, Sept. 2014.

[30] P. C. Rigby, D. M. German, and M.-A. Storey. Open source software peer review practices: a case study of the apache server. In *Proceedings of the 30th international conference on Software engineering*, pages 541–550. ACM, 2008.

[31] R. Rosenfeld. A maximum entropy approach to adaptive statistical language modelling. *Computer Speech & Language*, 10(3):187–228, 1996.

[32] Y. Shi, P. Wiggers, and C. M. Jonker. Towards recurrent neural networks language models with linguistic and contextual features. In *INTERSPEECH*, 2012.

[33] M. Triola. *Elementary Statistics*. Addison-Wesley, 10th edition, 2006.

[34] J. Tsay, L. Dabbish, and J. Herbsleb. Influence of social and technical factors for evaluating contribution in github. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 356–366, New York, NY, USA, 2014. ACM.

[35] Z. Tu, Z. Su, and P. Devanbu. On the localness of software. In *Proceedings of FSE 2012 (20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering)*, pages 269–280. ACM, 2014.