Towards Automating Code Review at Scale

Vincent J. Hellendoorn
vhellendoorn@cmu.edu
Carnegie Mellon University
USA

Jason Tsay
jason.tsay@ibm.com
IBM Research
USA

Manisha Mukherjee
mmukherj@andrew.cmu.edu
Carnegie Mellon University
USA

Martin Hirzel
hirzel@us.ibm.com
IBM Research
USA

ABSTRACT

As neural methods are increasingly used to support and automate software development tasks, code review is a natural next target. Yet, training models to imitate developers based on past code reviews is far from straightforward: reviews found in open-source projects vary greatly in quality, phrasing, and depth depending on the reviewer. In addition, changesets are often large, stretching the capacity of current neural models. Recent work reported modest success at predicting review resolutions, but largely side-stepped the above issues by focusing on small inputs where comments were already known to occur. This work examines the vision and challenges of automating code review at realistic scale. We collect hundreds of thousands of changesets across hundreds of projects that routinely conduct code review, many of which change thousands of tokens. We focus on predicting just the locations of comments, which are quite rare. By analyzing model performance and dataset statistics, we show that even this task is already challenging, in no small part because of tremendous variation and (apparent) randomness in code reviews. Our findings give rise to a research agenda for realistically and impactfully automating code review.

CCS CONCEPTS

• **Software and its engineering** → Open source model; **Software verification and validation**; *Software maintenance tools.*

KEYWORDS

code review, neural networks

ACM Reference Format:
Vincent J. Hellendoorn, Jason Tsay, Manisha Mukherjee, and Martin Hirzel. 2021. Towards Automating Code Review at Scale. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3468264.3473134

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3473134

1 INTRODUCTION

Reviewing code contributions is vitally important in both open-source and enterprise software projects [**?**]. Besides certifying the code's technical quality, review discussions are an important socialization tool for project managers to convey project norms or styles to contributors [4]. Yet, reviewing also innately requires much time and effort from project maintainers. Contributions vary tremendously in size: in our dataset of reviews on GitHub, many changes span dozens of files. Much of this code may not need detailed review from core members, but reviewers still need to be vigilant because missing small mistakes can have major consequences.

A natural solution is automation: repeating patterns across code reviews may be harnessed by intelligent methods to automatically prioritize, comment on, and improve contributions. This is a major challenge: across projects, code reviews and contributions are incredibly diverse in purpose, size, and scope. We lay the groundwork for doing so at a **realistic scale**, specifically presenting early findings on review prioritization (*where* to comment). Concretely, given an incoming code contribution (a *pull request*, or PR), we recommend which part of this PR's "diff chunks" a reviewer should comment on. We collect PRs from 245 popular, well-reviewed projects, that change up to 16K tokens of code (~1KLOC+) across up to 16 diff chunks. Importantly, only a few diff chunks per PR received comments during code review – many received none. We thus target an imbalanced binary classification problem where the goal is to recommend whether to comment on a particular diff chunk. We implement this through a deep learning model that takes into account reviews from pull requests in the given project's past, and contrast this with a strictly "inter-project" setting.

ICSE'21 recently published a related exploration of an orthogonal goal (how to revise code given comments) [15], reporting a modest ability to imitate real developer comments. This work approached automating code review from the strong assumption that the area to comment on was already identified and limited to ~100 tokens of code. These constraints, coupled with other filters, led to using just 17K samples, and the models used were correspondingly small. While such constrained explorations help understand the **potential** of models on a new task, they provide little insight into its **challenges** in realistic settings. This is a common theme in "AI4SE" research: most work uses artificially balanced or synthetic datasets to tackle tasks with imbalanced, complex distributions in practice, such as bug detection [8, 16] or code generation [1]; yet, doing so risks a major loss in efficacy in practice [7]. We therefore approach this goal from the other direction: predicting reviews on

Table 1: Summary of our filtered & aligned dataset.

	Projects	PRs	Changes (LOC)	Comment threads
Java	119	98,801	2,892,840	257,553
Python	136	79,326	4,616,561	330,032
Total	245	178,127	7,509,401	587,585

entire code contributions with minimal filtering. We hope that this sets a blueprint for other explorations in this space in general.

We identify two major challenges for moving this area forward. The first is **data collection:** we describe the challenge of creating an aligned dataset of (old code, comment, new code) triples from any repository on GitHub and involving arbitrary code changes, imposing as few artificial "data cleanliness" limitations as possible. The second challenge is **scale:** the PRs we study frequently involve thousands of changed tokens across many diff chunks, of which few receive comments. We show that this leads to a challenging modeling task, where even large models struggle to reach 50% precision, and reflect on the high degree of *ambiguity* present in this data that complicates the task – maintainers apparently comment quite inconsistently and unpredictably. We conclude with a roadmap for further research to improve this state, which emphasizes the need to understand this data better and relax the strict "imitation" criterion that requires tools to precisely copy developer actions.

2 RELATED WORK

Our work relates to both the study of technical and social mechanisms of code contributions in GitHub, and the fast-growing body of work on machine learning for code. Free open-source software projects have a long history of reviewing open contributions with benefits such as identifying and reducing defects [11] and newcomer socialization [5]. In GitHub's code contribution mechanism of pull requests, discussions are often opportunities for core members to mentor and convey project norms [4] or for community members to negotiate design decisions and project direction with core members [14]. The recent GitHub feature of reviewers explicitly suggesting code changes also increases the discussion around changes and decreases the turnaround time to implement changes [3].

Machine learning and language models have been applied to study code review acceptance rates [6, 12], generate review comments [13], and predict specific review changes [15]. The latter uses a deep-learning model to recommend code revisions that a reviewer should suggest to the contributor. They consider two settings, both of which aim to predict the corrected code after review, the first just using the faulty/submitted code, while the second uses both the faulty code and the reviewer comments. Thus the task is complementary to the one we explore, which is predicting *where* to comment. Our data collection approach is also substantially more comprehensive and may benefit their models in turn.

3 APPROACH

To automate code reviews at a realistic scale requires collecting such reviews from many projects with as few constraints as possible. We used the GitHub Archive[1] to extract all pull request (PR) related events from January 2017 to November 2020. We sort projects by their total count of PR comments since 2017 and subset the top 1,000 projects. From these, we tentatively focused on Java and Python projects, which include many popular and well-reviewed repositories,[2] resulting in 117 Java and 128 Python projects and 2,9M total comments.[3] We clone these 245 projects and identify all commits associated with each merged pull request. Each commit/PR contains one or more diff chunks. We chose to analyze reviews in terms of arbitrary chunks rather than methods (as in [15]) or files because it is a realistic representation of contributed code, which ranges from single-line changes to ones spanning multiple files.

The key challenge from a data collection perspective is reconstructing review *triples*, of originally submitted code, aligned natural language comments, and changed code. Each comment has an associated diff chunk of its own, but this may only partially capture/overlap with diffs in the final commit, and line numbers may change due to unrelated changes. Creating an aligned dataset thus involves a best-effort match of this diff chunk to the corresponding final commit. We implemented a script that reconstructs these intermediate states by applying diff chunks in reverse, based on a window around the affected line numbers, finding the minimal changeset across both the indicated lines and line numbers with shifts of up to two lines. We are able to align the majority (75%) of in-line comments with the code this way. We also identified a surprisingly high rate of comments that led to no changes (23%), including more than a few where the submitter responded stating they would make the required change, but never did, as well as changes in entirely different locations. Such patterns indicate a high degree of ambiguity and diversity of purposes in these reviews that we see echoed in model performance as well; this corpus is worth studying in more depth to understand such human factors.

Overall, we analyzed ca. one TeraByte of compressed data to create a corpus with the statistics shown in Table 1. We only consider comments where we are able to identify aligned code changes.

3.1 Model and Training Setup

Figure 1 shows the high-level modeling approach used in this work. We model PRs in terms of their diff chunks, each of which contains a patch for a segment of source code. Each chunk is initially encoded separately by a Transformer-style architecture [17] to yield a per-diff embedding, comprised of a hidden state per token. We then combine those states into a single joined embedding by averaging over the token dimension. This allows us to attend once more, now between diff chunks to exchange information and coordinate which one(s) are likely to receive a comment based on mutual information. Once completed, we project to a simple binary decision for each chunk, independently.

To implement this, we first subtokenize inputs using byte-pair-encoding [10] with a vocabulary size of 25K. Each token is then embedded and encoded by the initial, intra-diff encoder, for which we considered a "tiny" (4 layers × 256 dimensions), small (8 × 512), and medium (12 × 768)-sized Transformer. We consider PRs with up to 16 diff chunks and up to 1,024 tokens per diff chunk, padding

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

[2] Based on the primary language. We did not filter PRs based by file types changed to preserve realism; our dataset also includes PRs (partly) involving READMEs, configuration files, and some code in other languages such as JavaScript.

[3] Many in longer "threads"; we focus just on the presence of a comment, e.g. in Table 1

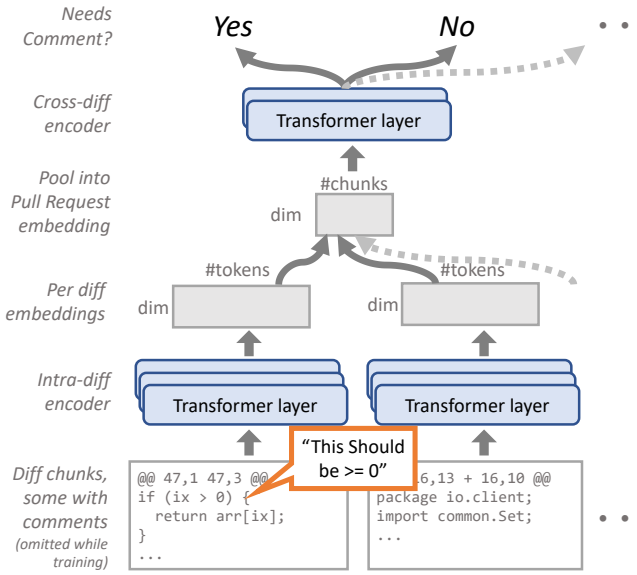

Figure 1: Overview of the modeling approach in our work. From bottom to top: we treat a PR as a series of diffs, some of which are known to receive comments; we first encode each part of a PR separately, then pool their states and attend across diffs, and finally predict a decision per chunk.

smaller samples to this size (ensuring proper treatment of masked tokens in pooling and attention operations). To manage such large inputs with attention-based methods, which connect every token to every other and are thus quadratic in complexity, we used the Nyström approximation, which pools token states into landmarks to decompose the attention product into more tractable terms [18]. This allows us to move beyond the typical upper bound of 512 tokens [8, 9]. Finally, we mean-pool (average) across (non-padding) token embeddings within each diff chunk to obtain 16 embeddings, one per diff chunk. We apply a second Transformer with 4 layers of (regular) attention on these, and project the resulting state to a single sigmoid-activated probability prediction per diff chunk.[4] These are trained with binary cross entropy loss to encourage the model to retrieve the original, binary ("has-comment") labels.

The average PR contains 1,172 tokens across 5.5 diff chunks. This distribution is skewed: half of PRs contain less than 4 chunks and 10% contain 12 or more. PRs with comments are converted into two samples: the before-review state and the submitted (correct) state. This forces our model to detect the (very) small changes during review. We process these in batches at a rate of 4 PRs per device across 4 NVIDIA RTX8000 GPUs in parallel. As such, each batch contains 256 diff chunks and up to ca. a quarter million tokens.

4 RESULTS

We train each model until held-out accuracy (at a per-diff level) stagnates. This typically happened early in training, within about one epoch on average, even with a low learning rate. This signals

[4]We also experimented with using the aforementioned diff-chunk embeddings directly, without another attention block, but that model performed far worse.

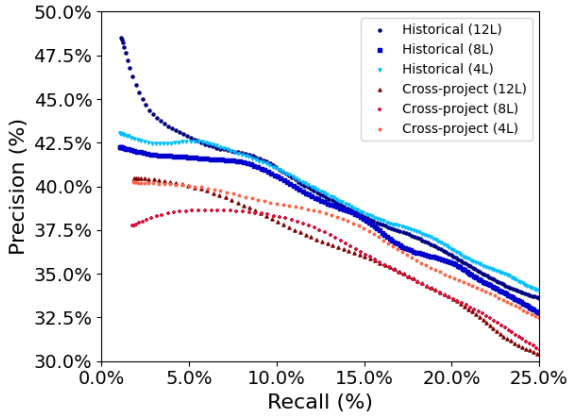

Figure 2: Precision-Recall trade-off of each model when ranking all diffs in the test set, focusing on the high-precision domain.

that the model derives more benefit from memorization than from finding generalizable patterns, perhaps due to a significant degree of randomness or ambiguity in the labeling [2].

The primary use of our models is for review prioritization: given a PR with a series of changes, identify which (if any) should be inspected first. Our model outputs a probability for each diff-chunk of needing a comment; this naturally lends itself to such ranking. We consider two settings for our models: history-based and cross-project. The former learns from reviews in a project's history and reserves the final 50 (or 10%, if fewer) to test on, thus allowing the models to learn from historical decisions. The second trains and evaluates on strictly non-overlapping sets of projects.

First, Figure 2 shows the precision/recall response when ranking diff-chunks across the full test set, which reflects the entire range of the models' probability assignments. Here, we focus just on the high-precision/low-recall domain, where predictions are most likely to be useful; all curves converge to the ~10% "random" base precision at 100% recall (as ~10% of diff chunks have comments). The history-based models consistently perform better than those operating only across project boundaries. Furthermore, some deeper models attain a slight edge in terms of top precision. At the same time, all models fail to break 50% precision. While that is five times better than random guessing,[5] it is far from practically useful. In addition, model capacity translates poorly into performance gain: in the cross-project setting, the smallest model (about the size of the largest model used in related work [15]) performs best!

We are especially interested in the ranking quality within a given PR. Figure 3 provides a breakdown of this behavior for both *All* pull requests, many of which have no comments, and only those with *Comments*, ordered by the number of diff chunks per PR. Here, we focus on perfect ranking accuracy, wherein all diffs with comments should be ranked strictly higher than ones without (by probability); we naturally omit single-chunk PRs, where the notion of ranking is moot. Both the historical and cross-project models clearly outperform a random baseline,[6] especially on PRs with comments. This task grows substantially more challenging with larger PRs, which

[5]Based on an average of 1 comment per 10 diff chunks in our dataset.
[6]Which does not track 1/#bins due to variations in number of comments by size.

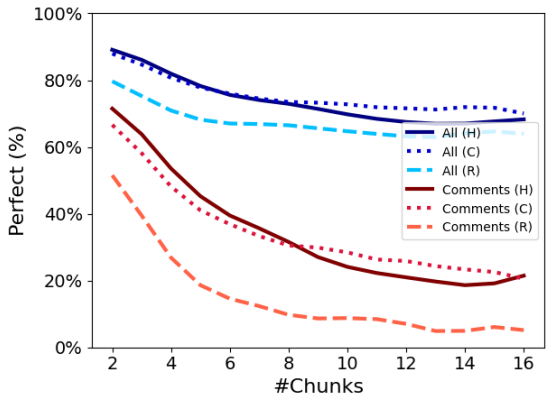

Figure 3: Per-PR ranking quality of the 12-layer historical (H) and cross-project (C) models and a random (R) baseline, grouped by PR size. "Comments" (bottom 3 red lines) consider only PRs with at least one comment.

tend to contain more comments. We also considered the accuracy of predicting just a single diff chunk that should receive a comment as the top prediction (for the with-comments setting); there, the model achieved ca. 40% accuracy among larger PRs.

5 IMPLICATIONS AND RESEARCH AGENDA

As AI4SE research has begun to turn to code review [15], we argue for doing so in a way that closely mimics its use in practice. We present a reality check on the challenge of this domain: even large models struggle to merely rank which parts of a changeset to review, achieving little to no gain over smaller architectures, nor with more training time, or with project-specific history. In a field that often trains and assesses its models on artificially balanced data, our work stresses the importance of focusing on realistic data distributions. That does not imply sacrificing the ability to learn – in this work, our models were able to prioritize reviews 5-10× better than a random base-rate; yet, since comments occur rarely and with little regularity, there is still a significant challenge ahead.

Our work makes two contributions towards this goal of realism: we provide a data collection method that is not reliant on any other tool than a GitHub event archive and is able to align a large proportion of comments with the eventually submitted code to reconstruct a contribution timeline. We also train models to capture the bulk of the range of changeset sizes seen in practice, going up to 16K tokens. This required adopting novel attention mechanisms not yet used in SE research. Our dataset and results provide a realistic baseline for research towards automating code review: evidently, achieving even modest precision in few cases is highly challenging.

Based on our findings, we propose the following research steps: **Quantify ambiguity:** both an anecdotal analysis of our data and empirical results of the models' inability to achieve high precision signal a high degree of uncertainty in practical code reviews. To automate this process, it is essential to identify how much of this uncertainty is *contextual* (e.g. could be resolved by constructing a detailed profile of the reviewer, including information from other conversations, etc.) vs. *inherent* (due to random inspection orders, day-to-day priorities, concentration levels, etc.). This may require human subject studies. Models can also help provide a signal by ablating architectures with and without such information sources. **Modeling Priorities:** deep learning is often heavily supervised, forcing it to imitate developers exactly. This need for a fine-grained signal leads to studies using synthetic, highly limited, or artificially balanced data for tasks where this is not realistic (e.g., bug detection: bugs are very rare in practice). Automating code review should avoid falling into these same traps of "tailoring the task to the model". Concretely, given the apparent ambiguity and complexity of this task, research should focus on developers' needs, rather than actions. For instance, imitating the exact wording or even placement for a particular comment may rarely be necessary, but a more useful task may be to predict the ultimate effect of such a comment (e.g., correcting a mistake). Identifying such needs may require human studies; once properly identified, training signals can be designed to respond to these needs.

REFERENCES

[1] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2019. Structural Language Models for Any-Code Generation. *arXiv preprint arXiv:1910.00577* (2019).
[2] Devansh Arpit, Stanisław Jastrzębski, Nicolas Ballas, David Krueger, Emmanuel Bengio, et al. 2017. A closer look at memorization in deep networks. *arXiv preprint arXiv:1706.05394* (2017).
[3] Chris Brown and Chris Parnin. 2020. Understanding the Impact of GitHub Suggested Changes on Recommendations between Developers. In *FSE*. 1065–1076.
[4] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social coding in GitHub: transparency and collaboration in an open software repository. In *CSCW*. 1277–1286.
[5] Nicolas Ducheneaut. 2005. Socialization in an Open Source Software Community: A Socio-Technical Analysis. *CSCW* 14, 4 (2005), 323–368.
[6] Vincent J Hellendoorn, Premkumar T Devanbu, and Alberto Bacchelli. 2015. Will they like this? evaluating code contributions with language models. In *MSR*. 157–167.
[7] Vincent J Hellendoorn, Sebastian Proksch, Harald C Gall, and Alberto Bacchelli. 2019. When Code Completion Fails: a Case Study on Real-World Completions. In *International Conference on Software Engineering (ICSE)*. 960–970.
[8] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global Relational Models of Source Code. In *ICLR*.
[9] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2019. Pre-trained Contextual Embedding of Source Code. *arXiv preprint arXiv:2001.00059* (2019).
[10] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. *arXiv preprint arXiv:2003.07914* (2020).
[11] Chris F Kemerer and Mark C Paulk. 2009. The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data. *IEEE Transactions on Software Engineering* 35, 4 (jul 2009), 534–550.
[12] Hengyi Li, Shuting Shi, Ferdian Thung, Xuan Huo, Bowen Xu, Ming Li, and David Lo. 2019. DeepReview : Automatic code review using deep multi-instance learning. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. 318–330.
[13] Jing Kai Siow, Cuiyun Gao, Lingling Fan, Sen Chen, and Yang Liu. 2020. CORE: Automating Review Recommendation for Code Changes. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 284–295.
[14] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Let's Talk About It: Evaluating Contributions through Discussion in GitHub. In *FSE*. 144–154.
[15] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. 2021. Towards Automating Code Review Activities. In *ICSE*.
[16] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2019. Neural program repair by jointly learning to localize and repair. *arXiv preprint arXiv:1904.01720* (2019).
[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762* (2017).
[18] Yunyang Xiong, Zhanpeng Zeng, Rudrasis Chakraborty, Mingxing Tan, Glenn Fung, Yin Li, and Vikas Singh. 2021. Nyströmformer: A Nyström-based Algorithm for Approximating Self-Attention. (2021).