Automatically Selecting Follow-up Questions for Deficient Bug Reports

Mia Mohammad Imran
Virginia Commonwealth University
Richmond, Virginia, U.S.A.
imranm3@vcu.edu

Agnieszka Ciborowska
Virginia Commonwealth University
Richmond, Virginia, U.S.A.
ciborowskaa@vcu.edu

Kostadin Damevski
Virginia Commonwealth University
Richmond, Virginia, U.S.A.
kdamevski@vcu.edu

***Abstract*—The availability of quality information in bug reports that are created daily by software users is key to rapidly fixing software faults. Improving incomplete or deficient bug reports, which are numerous in many popular and actively developed open source software projects, can make software maintenance more effective and improve software quality. In this paper, we propose a system that addresses the problem of bug report incompleteness by automatically posing follow-up questions, intended to elicit answers that add value and provide missing information to a bug report. Our system is based on selecting follow-up questions from a large corpus of already posted follow-up questions on GitHub. To estimate the best follow-up question for a specific deficient bug report we combine two metrics based on: 1) the compatibility of a follow-up question to a specific bug report; and 2) the utility the expected answer to the follow-up question would provide to the deficient bug report. Evaluation of our system, based on a manually annotated held-out data set, indicates improved performance over a set of simple and ablation baselines. A survey of software developers confirms the held-out set evaluation result that about half of the selected follow-up questions are considered valid. The survey also indicates that the valid follow-up questions are useful and can provide new information to a bug report most of the time, and are specific to a bug report some of the time.**

***Index Terms*—follow-up questions, bug reporting, bug triage**

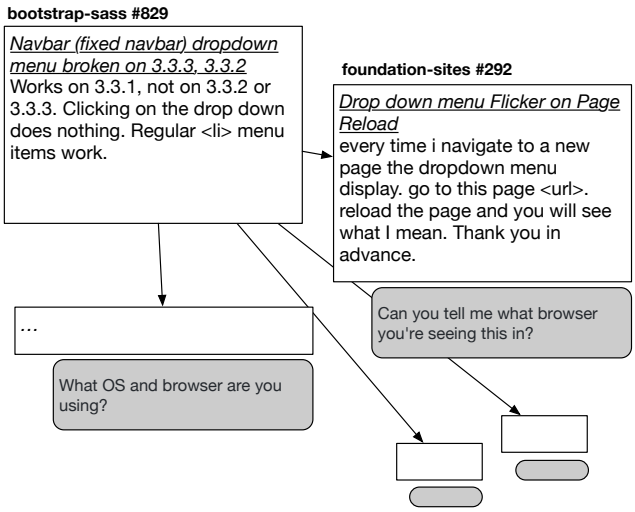

Fig. 1: The text for bug report #829 of the *bootstrap-sass* project is similar to other bug reports with already posed follow-up questions. The similarity between a pair of bug reports is illustrated as the length of the line connecting them.

I. Introduction

In many popular software projects, bug reports arrive with frequency and in bursts that can overwhelm even well-resourced and well-organized bug triage. At the same time, a significant proportion of the arriving bug reports lack sufficient actionable information for bug triagers to reproduce the bug. Researchers have observed this problem of bug report deficiency (or incompleteness), e.g., reporting that over 60% of bug reports lack any steps to reproduce and over 40% lack any description of the expected behavior [1]. Missing information in bug reports was also a key concern in the first open letter to GitHub from the maintainers of open source projects [2] [3], which was partially addressed via a bug report template mechanism. While nowadays some of the software projects on GitHub rely on specific templates or publish bug reporting guidelines that bug reports must follow, there are many cases where templates are ignored and guidelines are poorly followed by reporters. Posting a quick follow-up questions in order to obtain additional information from bug reporters is one method bug triagers use to augment the bug reports with necessary information. However follow-up questions are only effective if they are posed quickly, before the user reporting the bug loses focus on the specifics. In this paper, we examine how posing such follow-up questions for bug reports can be performed automatically, designing and describing a system to reduce bug triage effort, and improving overall bug report quality by automatically posing follow-up questions for deficient bug reports.

We base our automatic follow-up question posing system on the following assumptions and ideas: 1) relevant follow-up question are common, not overly specific, and have already been posed in other prior bug reports in the current project or in others; 2) similar bug reports necessitate similar follow-up questions; and 3) the utility of the answer provided to a prior similar follow-up question is indicative of its value to the current bug report. Based on this, our system performs an information retrieval task, locating the most relevant and useful follow-up question for a specific deficient bug report, given a large corpus of previous bug reports, follow-up questions, and their answers. For instance, consider the example shown in

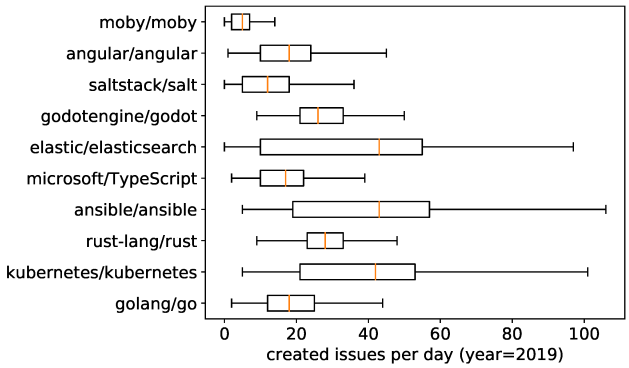

Fig. 2: Daily issue creation activity in 2019 for a selection of ten highly active (by number of commits) repositories on GitHub.

Figure 1, where the text of the bug report *boostrap-sass* #829 is similar to several other bug reports on GitHub with existing follow-up questions. In our system, choosing the right follow-up questions is a combination of the similarity between bug reports and the utility of the expected answer to the follow-up question. A requirement for a system that locates candidate follow-up questions and then ranks them in order of their perceived utility is a large-scale corpus.

To curate a corpus of prior bug reports, follow-up questions, and their answers we leverage GitHub, where we focus on popular repositories that have a high level of activity and therefore are likely to have numerous relevant follow-up questions, encoded as GitHub issue comments. We gather the answers to these follow-up questions that occur either as additional GitHub issue comments or as edits to the original bug report text. We base our estimate of the utility of an answer on the quantity of Observable Behavior (OB), Expected Behavior (EB) and Steps to Reproduce (S2R) it contributes, which we identify using the linguistic patterns published by Chaparro et al [1]. We evaluate our prototype in two ways, based on its ability to predict valid follow-up questions on a manually annotated held-out dataset, and based on a developer survey that aims to gauge the usefulness of the follow-up questions we recommend. The results indicate that the technique is viable, with 0.677 MRR on the held-out dataset, 53% of respondents indicating that the follow-up questions are valid, and 92% of the valid questions asking for new information for each bug report. To summarize, the primary contributions of this paper are:

1) mechanism for automatically posing follow-up questions to aid with incomplete bug reports;
2) metrics to select follow-up questions for a specific bug report from a corpus of bug reports, follow-up questions and their answers extracted from bug triage histories;
3) process for curating a large and high-quality corpus of bug reports, follow-up questions and answers from online software development platforms like GitHub.

Relative to the prior efforts by the software engineering research community towards improving the quality of bug

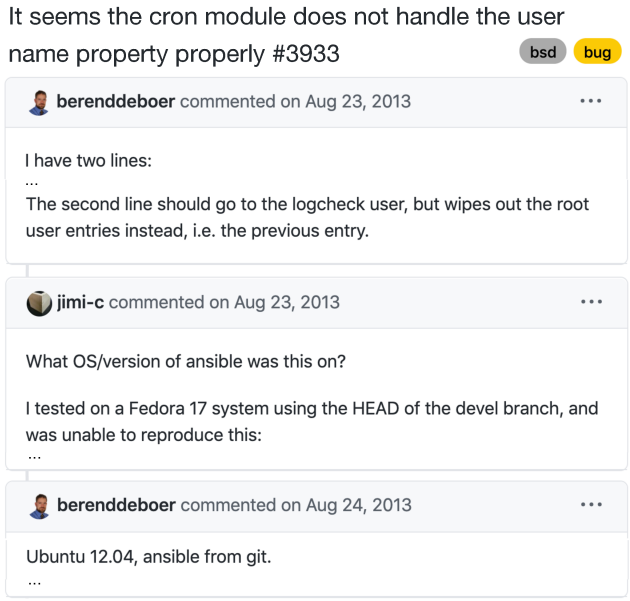

Fig. 3: Bug report with follow-up question and OB in answer.

reports, this paper is the first to propose follow-up questions. Automatically posing follow-up questions has been proposed in several other domains, e.g., for improving the quality of Web forum posts [4], product reviews in online retail [5], and improving query quality in Web search [6].

II. Bug Reporting in Open Source Projects

The overall trend in software development in recent years is towards increased speed of development and delivery. There are nowadays numerous projects on popular public software collaboration platforms like GitHub that have large development teams and user communities. Many of these projects experience significant bug reporting traffic [7]. Figure 2 shows the issue creation frequency for a selection of ten GitHub repositories that are currently active with a high numbers of commits and developers. Three of these repositories have a median of over 40 issues created daily, where most of them are bug reports reported by GitHub identities that have not contributed to the project (i.e., users). In addition, the same three projects exhibit high variance in daily issue creation, likely indicative of bursty and hard to predict bug reporting activity. This is a considerable burden for bug triagers and it motivates the need for the type of work as described in this paper, which intends to make bug triage more efficient and less of a burden for project maintainers.

Posing follow-up questions is an already practiced mitigation strategy for deficient bug reports. To quantify how widespread is the use of follow-up questions, we performed a small scale study of the prevalence of follow-up questions on GitHub, focusing on the 10 active projects used in Figure 2. From each of the 10 repositories, we randomly sampled 50 closed bug reports (500 in total) and manually examined them (by two of the paper's authors) for follow-up questions. We

were also interested whether the answers to those questions (if they are present) provide any of the three key parts of a bug report: Observable Behavior (OB), Expected Behavior (EB) or Steps to Reproduce (S2R). We found that follow-up questions were present in 23.6% (118/500) of bug reports and about 73% (86/118) of them were answered with 57% (49/86) of the answers containing Observable Behavior, Expected Behavior or Steps to Reproduce. Our analysis of this small randomly-sampled dataset indicates that, since follow-up questions tend to be answered by bug reporters at a relatively high rate (73% in our study) with answers that seem to add value to the bug report, an automated technique to pose such follow-up question should be of value.

We highlight one of the bug reports we examined (**ansible/ansible #3933**) in Figure 3. The follow-up question *What OS/version of Ansible was this on?* elicited an answer that provided key OB to this bug report, leading to its quick subsequent fix. Developer surveys have confirmed the importance of OB, EB and S2R in bug reports, observing that S2R is among the most valuable aspects of a bug report with OB and EB closely behind [8], [9]. The availability of existing follow-up questions on social coding platforms like GitHub provides the preconditions for the approach described in this paper, which leverages such existing follow-up questions to automatically rank and select the most appropriate one to be asked for a newly written, incomplete bug report. In the remainder of this paper, we describe the design of this system, which we entitle Bug-AutoQ – Bug Automated Questioner[1].

III. System Description

As input, our system for retrieving follow-up questions, Bug-AutoQ, requires: 1) a bug report of interest; 2) a corpus of already posed follow-up questions extracted from GitHub issues, including their corresponding bug reports and answers. When the bug report is deficient in OB, EB, or S2R (computed based on the language patterns described in Section III-C2), Bug-AutoQ poses a single follow-up question (or a small set) appropriate to the deficient bug report. The overall vision of Bug-AutoQ is shown in Figure 4. The left side of the figure shows the online part of Bug-AutoQ that determines the optimal follow-up questions for a deficient bug report, while the right side shows the steps necessary for the offline generation of a corpus of follow-up questions for reuse. In this section, we describe our system, including how we create a large corpus of follow-up questions to recommend, how we select candidate follow-up questions from this corpus for a specific incomplete bug report, and how we rank these questions in descending order of their potential utility to the bug report.

A. Selecting a Corpus of Bug Reports

Our goal in curating a corpus of bug report-related follow-up questions and their answers is to find a large, representative and high-quality corpus. Manually curated corpora are of high quality but they are difficult to scale-up. Automatic curation can easily scale but it can be affected by significant noise, leading to low data quality, unless care is taken to filter and sample follow-up questions in a way that noise is mitigated. As corpus size is an important factor in our system, we opt for an automated approach with numerous filters to ensure the data is of highest possible quality. With the number of active repositories available on GitHub providing a very large input domain, we can afford to err on the side of being overly restrictive in our filtering. To automatically curate our corpus, we: 1) select GitHub repositories that have high bug reporting activity, as measured by the number of issues created by non-contributors over some fixed period of time; 2) select issues in those bug repositories that contain rapidly asked and succinct follow-up questions contained in GitHub issue comments; 3) locate answers to the follow-up questions encoded as either comments or as edits to the original bug report.

In more detail, we used the following sequence of steps to curate the corpus. The highlights of the corpus curation process are also illustrated on the right part of Figure 4.

1) Using the public GitHub APIs, we scraped a set of public GitHub repositories with a high rate of non-contributor created issues, where a non-contributor is a GitHub user that has never committed any code in the specific repository. GitHub repositories with these characteristics form our target population, i.e., projects that are more likely to be in need our technique. In order to somewhat constrain the number of GitHub repositories, we focused on longer-running projects, specifically, with repositories created between 2008-2014, and recently active with new issues created after Jan 1, 2019.
2) For each of these repositories, in descending order of their number of non-contributor created issues per day, we selected all issues from each repository's GitHub issue tracker that are labeled as "bug", "crash", "fix", "defect" or unlabeled. As an example, the bug report in Figure 3 is labeled as "bsd" and "bug". Our goal for this step was to avoid feature requests and focus on bug reports. We observed that issue labels were not used consistently enough in projects on GitHub, which is why we opted to include unlabeled issues. Since we are interested in deficient bug reports, we selected bug reports that do not contain any Observable Behavior, Expected Behavior, or Steps to Reproduce.
3) We further selected only issues that contain follow-up questions in one of the issue comments. We identified follow-up questions as comments containing only questions, identified by both starting with an interrogative word and ending with a question mark. In order to ensure we selected follow-up questions and not just any questions, we constrained our selection based on time and comment sequence. That is, the comment containing the follow-up question must have been posted within 60 days of the issue creation date and must have occurred as the comment immediately following the post. We also

[1]Replication package available at: https://tinyurl.com/y4k43fll

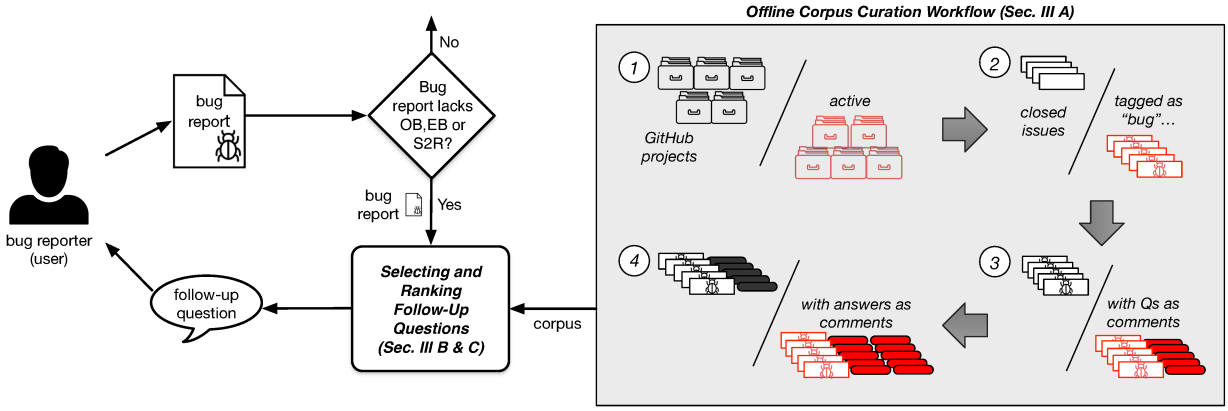

Fig. 4: Overall workflow for how the Bug-AutoQ system is used to improve bug report quality.

avoided follow-up posts by the issue author by ensuring that the comment was authored by a different user from the author of the issue.

4) The set of issues and candidate follow-up questions from the previous step were further filtered to retain issues and follow-up questions where an answer was provided. A key heuristic we used for recognizing an answer was that it was authored by the original issue creator and occurred as the the next sequential comment to the follow-up question. We also searched for answers that were encoded as edits to the original issue text by the author, which occurred after the follow-up question was posted and were limited to add at least 4 additional words to the original text in order to avoid minor spelling or grammar modifications.

We stopped the collection process when we gathered a dataset of 25K GitHub issues, which we deemed to be sufficient for our purpose. With 25K issues, the dataset provided a distribution of OB/EB/S2R in the answers, specifically 10930 answers containing OB, 3100 answers containing EB, and 266 containing S2R. Each data point in our dataset is a triple of bug report, follow-up question, and answer. Together, the 25K triples form the primary corpus we used to retrieve and rank follow-up questions for a given incomplete bug report.

B. Selecting Candidate Follow-Up Questions

Selecting a set of most appropriate candidate follow-up questions for a specific incomplete bug report of interest from the corpus of 25K triples (bug reports, follow-up questions and their answers) can be formulated as an information retrieval problem. That is, as a query we use the text of the incomplete bug report. We represent the corpus as an inverted index of the bug report text (i.e., using Lucene), and use *tf*idf* as the ranking mechanism. In this way, we retrieve a set of 10 candidate follow-up questions for each incomplete bug report of interest, where these 10 candidates have the most similar bug report text to the bug report of index. Later, through manual annotation of a set of bug reports, as shown in Figure 6, we confirmed that 10 is a good upper-bound for the number of valid follow-up questions per bug report.

More specifically, we create a Lucene index of the corpus of bug reports by following this set of steps:

- *Filtering.* Removing code or stack traces from bug reports in our index allows for more accurate matching. GitHub issues use markdown so we remove all text surrounded by triple-quotes as this is typically how source code and stack traces are encoded. We also remove quoted text (i.e., lines that start with the greater than symbol) as these typically refer to some external information, which, again, is often stack traces, code, or project documentation.
- *Tokenization.* We perform standard tokenization used in software engineering applications of information retrieval [10], [11]. We tokenize on white space, remove punctuation (except for horizontal dashes) and split on camel case and dashes, while also keeping the original unsplit tokens.
- *Indexing.* We use Lucene's standard configuration to index the title and the body in separate fields. In order to provide more context, we add to the index the labels on GitHub that the bug report is associated with (e.g., *fix-later, critical*) and the labels the GitHub repository is associated with (e.g., *java, linux, web-server*). The former provides specifics on the issue while the latter usually denotes technologies the project uses or broad categories it belongs to.

To create a query out of the incomplete bug report, we tokenize its title and body using the equivalent process to the one we performed on the bug reports in the corpus. While Lucene returns a ranked list of follow-up questions according to *tf*idf*, our use of this mechanism is only in retrieving a set of 10 candidates. In the following section, we describe an improved, customized ranking, which takes into account both the question's compatibility and its utility in order to select which follow-up question to pose to the deficient bug report.

C. Ranking the Candidate Follow-Up Questions

To rank the set of candidate follow-up questions we reformulate and apply towards bug reporting the notion of the Expected Value of Perfect Information (EVPI), initially proposed as a means to rank follow-up quesitons by Rao et

al. [4]. To evaluate a follow-up question, EVPI suggests using the (expected) value of its answer, i.e., for an incomplete bug report, EVPI estimates the value of the information provided by the answer to the follow-up question. The higher the EVPI the higher we rank a follow-up question from the candidate set.

Given an incomplete bug report of interest, br, we express the EVPI of a specific follow-up question q_i from our candidate set as the product of: 1) the *compatibility* of a specific follow-up question to the bug report, i.e., the probability of a specific question and answer pair occurring for that bug report, $P(q_i + a_i|br)$, and 2) the *utility* of that question, $U(q_i)$.

$$EVPI(q_i|br) = P(q_i + a_i|br) * U(q_i)$$

The utility of a question is further expressed in terms of the average quality of the answers it has received, i.e.,

$$U(q_i) = \frac{1}{|A|} * \sum_{\hat{a} \in A} \frac{|OB(\hat{a})| + |EB(\hat{a})| + |S2R(\hat{a})|}{|\hat{a}|}$$

,where A is the set of answers the question q_i has received across all of its one or more occurrences in the corpus, $\hat{a}$ is one of those answers, and $OB(\hat{a})$, $EB(\hat{a})$ and $S2R(\hat{a})$ are the sentences describing Observable Behavior (OB), Expected Behavior (EB), or Steps to Reproduce (S2R) found in the answer. We use the $|\cdot|$ operator to denote the cardinality of a set. The idea of this metric is that questions whose set of answers have a high proportion of OB, EB, or S2R have higher utility than questions whose answers do not contribute much in terms of this type of information. Bug-AutoQ uses this formulation of EVPI, the product of compatibility and utility, which we estimate using a set of steps that we describe next. Rao et al. [4] initially defined EVPI as the product of the probability distribution (which we call compatibility) and the utility, but their definition of these two constituent terms and how they are computed differs in significant ways from ours.

1) Estimating Compatibility: To compute the probability of a follow-up question occurring for a given bug report, $P(q_i + a_i|br)$, we use a neural network approximation, using a deep NN consisting of 3 layers (as shown in Figure 5). As the first layer, in order to introduce semantics, we encode the bug report text with GloVe word embeddings that we pre-trained on the entirety of Stack Overflow (using the most recent Stack Overflow data dump as of June, 2020) with default parameters ($vector\_size = 200; window\_size = 15$). As the second layer, in order to capture the word sequence, we train a LSTM and compute an average across the hidden states. As the third and final layer we use a dense neural network. The output of the entire network is a vector that is intended to be as similar as possible (according to cosine similarity) to the concatenated average word embeddings of q_i and a_i (using the same GloVe vectors as above). We train the neural architecture using the actually posed follow-up questions and answers as positive labeled examples for q_i and a_i and the remaining 9 question-answer pairs in the candidate set as negatively labeled. We use cosine embedding loss and weight balancing to manage the resulting class imbalance as the negative examples significantly outnumber the positive ones, 9 to 1.

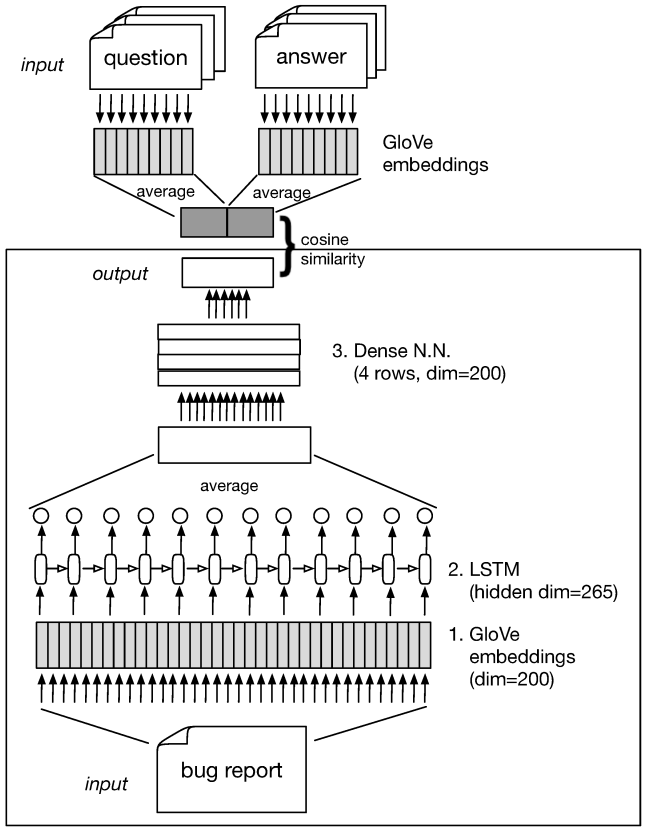

Fig. 5: The neural architecture for estimating Compatibility. The neural network outputs a vector similar to the "ideal" question and answer for the input bug report.

2) Estimating Utility: To compute the utility of the follow-up question, $U(q_i)$, we leverage the pattern based identification of the constituent pieces of a bug report (OB, EB and S2R) created by Chaparro et al. [1] We implement 5 of the most common and most productive patterns for each of OB, EB and S2R, as described by the authors. Each of the total 15 patterns rely on predefined sets of keywords and part-of-speech tagging to ensure high precision[2]. We value high precision over recall since we believe it is more important for Bug-AutoQ to produce fewer false positives rather than fewer false negatives, i.e, it is better to pose a good follow-up question rather than ensure a follow-up question for all incomplete bug reports.

In order to compute the average utility of the answers for a particular question in the corpus, we again rely on Lucene to provide matching between questions and we reserve 10 of the most similar question variants for each queried follow-

[2]Depending on the choice of patterns, Chaparro et al. report average EB precision of 95.1%-96.7%, recall of 46.1%-76.6% and average S2R precision of 81.6%-84.5% and recall of 31.0%-38.5%; OB results were not reported [1].

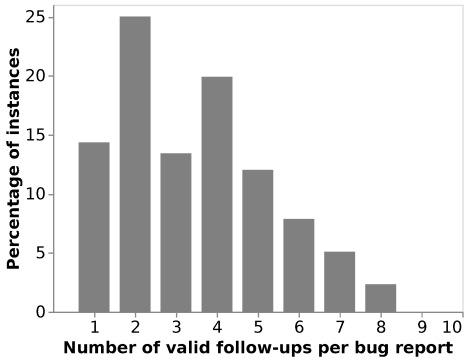

Fig. 6: Distribution of valid follow-up questions in the held-out set (average=3.45).

up question. The intuition is that it is beneficial to estimate the utility of a follow-up question not by what is in a single answer, which can be noisy, but by the average amount of OB/EB/S2R across a number of answers of this question (or its close variants). In other words, individual answers on GitHub, even to well posed follow-up questions, can sometimes be inadequate, but, when averaging over several instances of the similar question we can more accurately estimate its utility. Note that Lucene is used at two different stages of Bug-AutoQ; we used Lucene to compare the incomplete bug reports as well as here within the Utility computation.

IV. Evaluation

We implemented a prototype of the follow-up question selection of the Bug-AutoQ system (i.e., Offline Corpus Curation and Selecting Follow-Up Questions, as shown in Figure 4) with the aim of evaluating its combined efficacy along a few different dimensions. First, we use metrics and a held-out data set to evaluate the quality of the recommendation, i.e., how well the system selects valid follow-up questions for incomplete bug reports. We define *valid* follow-up questions as those that are a match to the topic and content of a specific bug report. Second, we use a survey of software developers to evaluate the follow-up questions on their usefulness, novelty and specificity. We define *usefulness* as the perceived ability of the follow-up question to elicit additional valuable information for diagnosing the bug; *novelty* as the perceived ability to elicit new, previously unreported information, and *specificity* as how much the follow-up question is tailored for the target bug report vs. applicable to a broad range of other bug reports.

A. Quality of Follow-Up Question Ranking

One way of evaluating the ranking system, based on a held-out dataset (of bug reports and candidate follow-up questions), is by using the posed questions as the ground truth. However, this simple setup has a serious deficiency in that the actually posed question may not always be the most optimal among the set of candidate follow-up questions. As evidence for this, we sometimes observed cases when bug reporters responded negatively to the posed follow-up questions, e.g., answering "why is that relevant" (https://github.com/mautic/mautic/issues/1550). More importantly, several of the remaining candidate questions may be valid and (more) relevant to the bug report and therefore should not be considered as negatively labeled instances for evaluation. Therefore, in order to provide an evaluation set that identifies all of the valid questions in the candidate set, we perform manual annotation that clearly identifies all of the valid follow-up questions for a specific bug report.

1) Annotation: We annotated 400 randomly chosen bug reports that were held-out from training in our original corpus of 25K (curated as described in Section III-A). The annotation was performed by two of the authors following an agreed-upon predefined procedure. We focused our annotation on the 10 candidate follow-up question retrieved by Lucene. For each bug report, each annotator 1) read the bug report carefully, spending a few minutes to understand its context, e.g., by looking at the purpose of the overall GitHub project and the types of technologies it relies on; 2) marked all of the follow-up questions for the candidate set of 10 that were valid. Both of the annotators processed the same set of 400 bug reports, marking an average of 3.45/10 of the follow-up questions as valid with an inter-annotator agreement (Cohen's kappa) of 0.60. The distribution of valid follow-up questions is shown in Figure 6. We use the set of follow-up questions that both annotators agreed were valid, i.e., the intersection between their annotations.

2) Baselines: The baselines we identified are meant to convey both straightforward approaches to ranking (e.g., directly using the Lucene output) and ablation, i.e., using one part of our ranking function but not the other (e.g., ranking based only on the question utility, $U(q_i)$). We did not find directly related prior techniques to compare against, since the research direction is novel and models from other domains with a similar purpose are too different in form. Below is an enumerated list of all of the ranking baselines we used.

- *Random* – A random permutation of the candidate follow-up question list. We present metrics averaged over 10 runs.
- *Lucene* – Lucene uses the vector space model (i.e., *tf*idf*) to rank follow-up questions based on the similarity between the bug reports. This baseline just transfers Lucene's ranking, which we use to generate our candidate set of 10 follow-up questions, as the system's output.
- *Rao et al. [4]* – The technique proposed by Rao et al. targeting Web forum posts.
- *Utility only* – $U(q_i)$ – The utility function, described in detail in Section III-C, computes the average amount of OB, EB or S2R found in the answers to the specific follow-up question.
- *Compatibility only* – $P(q_i + a_i|br)$ – The compatibility function computes the probability a bug report can be combined with a specific follow-up question and answer pair. The implementation uses a deep NN architecture to compute this value.

TABLE I: Evaluation results contrasting our system (Bug-AutoQ) relative to several baselines.

	MRR	Wilcoxon p-value	Effect size	**P@1**	Wilcoxon p-value	Effect size	**P@3**	Wilcoxon p-value	Effect size	**P@5**	Wilcoxon p-value	Effect size
Bug-AutoQ	**0.677**	-	-	**0.486**	-	-	**0.492**	-	-	**0.446**	-	-
BASELINES:												
Random	0.542	$p < 0.01$	0.229	0.319	$p < 0.01$	0.167	0.368	$p < 0.01$	0.216	0.355	$p < 0.01$	0.214
Lucene	0.534	$p < 0.01$	0.252	0.347	$p < 0.01$	0.139	0.318	$p < 0.01$	0.308	0.317	$p < 0.01$	0.294
Rao et al. [4]	0.551	$p < 0.01$	0.218	0.342	$p < 0.01$	0.144	0.336	$p < 0.01$	0.279	0.342	$p < 0.01$	0.245
Utility only	0.646	$p = 0.11$	0.059	0.468	$p = 0.60$	0.019	0.443	$p = 0.01$	0.087	0.412	$p = 0.01$	0.077
Compatibility only	0.612	$p = 0.01$	0.115	0.426	$p = 0.11$	0.060	0.383	$p < 0.01$	0.196	0.377	$p < 0.01$	0.152

3) Metrics: We use two popular information retrieval evaluation metrics: Mean Reciprocal Rank (MRR) and Precision@n (P@n).

The goal of MRR is to evaluate how effective is our technique, or a baseline, in locating the first valid follow-up question, as, presumably, this is a proxy for the ease with which an end-user would locate a follow-up question in the ranking. It is computed as:

$$MRR = \frac{1}{|B|}\sum_{i=1}^{|B|}\frac{1}{rank_i}$$

,where B is the set of bug reports in the test set and $rank_i$ is the ranked position of the first valid follow-up question for the i^{th} bug report.

The goal of Precision@n is to measure the number of valid results when considering the top n positions in the ranking. Unlike MRR, it consider all, not only the topmost ranked, results. It is computed as:

$$P@n = \frac{1}{|B|}\sum_{i=1}^{|B|}\frac{|v|}{n}$$

,where, as before, B is the set of bug reports in the test set and v is the set of valid follow-up questions ranked in the top n positions. We use values of 1, 3 and 5 for n.

We compute Wilcoxon's signed rank test for each of the above metrics to estimate the statistical significance of the difference between our technique Bug-AutoQ and the baselines. The effect size of the comparison is calculated using Cliff's delta (δ) [12], which ranges from -1 (all values in the first group are larger than the second group) to +1 (all values in the second group are larger than the first group). A value of zero indicates that the two groups are identical. The criteria for interpreting δ is that $|\delta| > 0.147 \rightarrow$ small effect, $|\delta| > 0.33 \rightarrow$ medium effect, and $|\delta| > 0.474 \rightarrow$ large effect [13].

4) Results: We summarize the results of our technique (Bug-AutoQ) versus the identified baselines in Table I. Our results indicate that Bug-AutoQ outperforms all of the baselines, with the ablation-type baselines performing better than the simple baselines. The Lucene ranking does surprisingly poor, basically in line with the Random baseline. The Utility only baseline is the ones that comes closest to the performance of the full system. Perhaps the most intuitive result is $P@1$, where Bug-AutoQ scores 0.49, indicating that just about half of all of the top selected follow-up questions by our system were valid. The Wilcoxon's signed rank test and the Cliff's delta confirm the observations from the raw metric values, i.e., that Utility only has a strong similarity to Bug-AutoQ and could be strong contributing factor to the approach's effectiveness. They also confirm that Bug-AutoQ has a strong advantage over the simple baselines.

We interpret the results to mean that our formulations of Utility and Compatibility, which are designed to be more resilent to noisy data than Rao et al. [4] are indeed effective. We also observe that within the top 10 candidate bug report, as retrieved by Lucene, there is usually a reasonable number of good candidates so that a random choice within them (i.e., the Random baseline in Table I) is fairly effective with a reasonable MRR and $P@1$ of almost one third. However, past the initial retrieval of the top 10, the Lucene ranking within them (i.e., the Lucene baseline in Table I) does not seem to provide much quality as is equivalent to the Random baseline.

B. Developer Survey

While a recommended follow-up question may be valid, it may not possess other properties that would encourage its use in practice, i.e., in a system that automatically poses follow-up questions for incomplete bug reports. For instance, a follow-up question may be overly generic, lacking detail or context specific to the bug report (e.g., *Can you provide additional information?*). To investigate how Bug-AutoQ performs across several such dimensions of interest we conducted a survey with software developers.

Through personal contacts, we e-mailed 10 software developers about the study, providing the basic context of our project, brief definitions and examples of the characteristics, and a link to a Web form containing the survey. None of the developers were aware about the details of our technique. The developers were half (5) from academia (graduate students at institutions in the U.S. and Europe) and half professional developers from industry. All had programming experience of 4 or more years with popular languages like Java and Python and all indicated one of their primary responsibilities was developing software.

We randomly assigned the developers into two groups of 5 and each group was assigned 12 instances of bug report and follow-up question pairs, where all of the follow-up questions were the top-1 selected by Bug-AutoQ. Each group was presented with bug reports from our corpus that belong to GitHub projects where Java or Python are the primary technologies. For each of the assigned bug reports, a developer was presented with a screenshot from GitHub containing the

Fig. 7: Responses to developer survey grouped per category (top) and per follow-up question (bottom). FQn = Follow-Up Question n.

TABLE II: Comparison of the Bug-AutoQ follow-up question and the actually posed (original) follow-up question for three of the highest rated instances by the survey respondents.

Instance	Bug report	Bug-AutoQ follow-up question	Original follow-up question
FQ22	*Title* = "Plugin is not working on latest android" *Body* = "On app start: [...] On Button Click: [...] I am running the example code on Arch Linux with Java 8, latest Android SDK with target API 22 and default nexus 5 ADM Config. Any ideas why the SpeechRecognition object is not being initialized?"	[...] Are you waiting for the device ready to fire before calling the SpeechRecognition?	[...] Are you waiting for the device ready to fire before calling the SpeechRecognition?
FQ14	*Title* = "Multiple time tests discovering" *Body* = "I have strange problem with running xunit tests as part of my build process. Since yesterday after running all tests my tfs build starts discovering tests again and after that it runs tests again. And that's repeated 15 times. Below I have log from TFS: [...]"	Can you describe your scenario a little more? One way of doing this is described in [...]	Can you check if you have multiple copies of the xUnit adapter under [...]? Thanks!
FQ8	*Title* = "32-bit Intel tests for python2 for test_fitpack fail with segfault in fpbisp #8122" *Body* = "Environment: Archlinux 32, python-scipy 1.0.0, python-numpy 1.13.3, python 2.7.14 [...]"	Do you have a more verbose log? It looks like it's not picking up the python library at all judging by the error.	How did you install or build SciPy? Compiler versions, build log?

title and text of the bug report, the follow-up question, and a link to the project the bug report came from for context. Prior to beginning the survey, we gave instructions to the developers to read both the bug report and the follow-up question before answering the provided set of survey questions. A preliminary survey question, which was posed on an initial screen, asked *Is the follow-up question valid?*. A negative response indicated that the follow-up question was invalid and unusable, therefore we asked no additional questions for that specific bug report - follow-up question pair. The remaining survey questions only appeared for instances deemed valid by a developer.

For a valid follow-up question, we posed a yes/no survey question asking *Does the follow-up question ask for new information currently not included in the description?*, followed by two Likert score survey questions (5 point; Strongly Disagree, Disagree, Neutral, Agree, Strongly Agree) interrogating whether *The follow-up question is specific to the bug report* and *The follow-up question is useful to the bug report*. In the following, we refer to the first (yes/no) survey question as measuring New Information, the second measuring Specificity, and the third measuring Usefulness.

The survey results showed that out of the 24 different bug report - follow-up question pairs, a majority of participants (at least 3 out of 5) considered 13 follow-up questions as valid, and 11 as invalid. This ratio is analogous to the results we observed for Precision@1 in the held-out set evaluation

presented above, confirming our expectations.

The results of the survey, along each dimension, are presented both at the granularity of an individual response and at the granularity of a bug report (and follow-up question pair) in Figure 7. The survey results indicate that among the categories of Usefulness, Specificity, and New Information, the follow-up questions selected by Bug-AutoQ provide the most of New Information, followed by Usefulness and Specificity. However, all three categories were generally positive with over 50% of all responses on each either agreeing or strongly agreeing with the statements on Specificity and Usefulness or affirming that the follow-up question was asking for New Information. Considering the data per follow-up question, in the bottom part of Figure 7, we observe most follow-up questions, which were deemed as valid, were rated positively. Some follow-up questions have strongly positive ratings, e.g., follow-up question 14 (FQ14) was rated by all 5 respondents as valid, 4/5 agreeing or strongly agreeing that the question was specific, 4/5 agreeing or strongly agreeing that the question was useful, and 5/5 agreeing that the question aimed to provide new information for the bug report.

To further illustrate Bug-AutoQ's performance, we contrast the Bug-AutoQ follow-up questions to the ones posed in the original bug report for three of the highest rated survey instances in Table II. While for one of the instances, FQ22, the recommended follow-up question matches the posed follow-up question, which is possible since the original follow-up question is among our candidate set, the follow-up question in FQ15 is different, asking the question of *Can you describe your scenario a little more?* and providing some additional context on how to do so in the succeeding sentence. In FQ8 both the Bug-AutoQ and original follow-up questions inquire about a log in order to better investigate a Python library dependency issue.

C. Threats to Validity

The presented approach is affected by several limitations that may negatively impact the validity of our findings.

Construct validity. One threat to construct validity is our use of a manually annotated dataset for the held-out dataset evaluation. We limited this threat by following an annotation process that required the annotators to get familiar with each project. We also observed reasonable Cohen's Kappa values between the two annotators. Another threat to construct validity is using Chaparro et al.'s [1] definitions to find OB/EB/S2R in bug reports. We did not perform an explicit validation of the pattern accuracy as that would have required us to curate a non-trivial dataset specifically for this purpose. Instead, we aimed to control noise introduced in recognizing OB/EB/S2R by our formulation of EVPI. A threat to construct validity is also in the preprocessing of bug reports, since we focus only on the textual content of an issue and ignore other enclosed types of information, such as images or links. As a result, for some deficient bug reports, our technique may miss potentially relevant information to locate the most useful follow-up questions. This threat is partially mitigated by the size of the corpus, containing 25K GitHub issues with various characteristics and content. The notion of utility of a follow-up question poses another threat to validity, since it is based only on a subset of information that can be enclosed in a bug report. To mitigate this threat and select the most useful type of information, we followed prior studies that reported S2R, OB and EB among the most helpful categories of information according to software developers [14].

Internal validity. The limited number of follow-up questions associated with each bug report poses a threat to internal validity. Each bug report is assigned 10 candidate question to rank, instead of processing the whole corpus of available questions. This may lead to omitting follow-up questions that are valid and specific in the context of a particular bug. We partially mitigate this threat by selecting the most similar bug reports, while also extending the content of each bug report with the bug's labels and repository tags in order to provide Lucene with contextual information that can be leveraged when locating similar issues.

External validity. In this study, we leveraged a dataset of 25K GitHub issues, however, during evaluation we use a subset of 400 manually annotated bug reports. The limited size of the test set may impact our observations as it covers only a small subset of population. To mitigate that threat, we include bug reports from 357 open-source software projects leveraging different frameworks and technologies. Moreover, to ensure the quality of the proposed approach, we conducted a user study with 10 software developers. We observed that the number of valid follow-up questions in the user study nearly matched the result obtained in the held-out evaluation, emphasizing the overall quality of the system and supporting validity of the results. The scope of the developer survey poses another threat to external validity due to the low number of enclosed bug reports and selection of issues only from Python or Java-related software projects. We partially mitigate bias caused by limiting the bug reports to only two technologies by sampling issues with different content characteristics (e.g., with or without stack traces) from multiple projects. To ensure the quality and generalizability of the survey results, each pair of bug report and follow-up question was assessed by half of the respondents.

V. Related Work

To our knowledge, the proposed approach is the first effort towards improving the quality of bug reports by asking follow-up questions. Prior research related to this area can be broadly grouped into three main categories including evaluation of bug reports quality, approaches for improving deficient bug reports, and techniques automatically posing follow-up questions in domains external to software engineering field.

Analyzing the quality of bug reports. The quality of user written bug reports is a topic that several researchers have been interested in. Linstead et al. applied Latent Dirichlet Analysis to a large corpus of bug reports to study their semantic coherence [15], while Huo et al. investigated how the content of a bug report changes depending on the level

of expertise of its author [16]. Di Sorbo et al. observed that issues marked as "won't fix" often contains numerous errors in their reports [17], while insufficient amount of information supplied within a bug report can lead to developers not being able to reproduce as bug, as noted by Joorabchi et al. [18]. Researchers have been extensively investigating the content of bug reports to determine the most useful information leading to locating and fixing buggy code efficiently. To this end, Davies et al. manually analyzed a corpus of bug reports from four popular open-source projects finding that observable behavior and expected behavior are among the most consistently encountered parts of a bug report [19]. Survey of software developers conducted by Sasso et al. reveled that steps to reproduce, test cases and stack traces are the most helpful types of information, however they were also the hardest for users to supply [20]. These finding were confirmed and further expanded in the study of Laukkanen et al. who indicated the importance of application's configuration [9]. Chaparro et al. developed a technique leveraging language patterns to automatically extract observable behavior, expected behavior, and steps to reproduce from a bug report [1]. Liu et al. proposed to improve Chaparro's technique by eschewing predefined patterns, instead relying on pre-trained classifier to identify steps to reproduce [21]. Recently, Yu et al. developed a tool, S2RMiner, that extracts steps to reproduce from a bug report with high accuracy [22].

Improving inadequate bug reports. Researchers have approached the problem of improving the quality of bug reports from a few different angels. One line of work, with numerous proposed techniques, is to detect duplicate bug reports [23]–[25]. Another research avenue is to classify bug reports into valid vs. invalid or easy vs. difficult bug reports [26]–[28]. Researchers have also attempted to automatically improve specific parts of bug reports. Moran et al. provided auto-completion for the steps to reproduce portion of bug reports by leveraging image processing of screenshots taken from the application's UI [29]. Chaparro et al. explored how bug report quality can be improved based on unexpected vocabularies in the steps to reproduce [30]. Recently proposed BEE tool, implemented as a GitHub plugin, extracts observable behavior, expected behavior, and steps to reproduce from a bug report in order to alert bug reporters when this information is not provided [31].

Automatically posing follow-up questions. Research on automatic question generation has been applied within a few different domains and applications. One topic of extensive prior research is on generating questions from an existing document, i.e., questions whose answers can be found within the given text [32]–[37]. For instance, such generated questions can be used for educational assessment and automation. More recently, researchers have envisioned a future where a user's information need will be satisfied via dialog with a virtual assistant, i.e., follow-up questions that are automatically posed to clarify the user's intent. To this end, Braslavski et al. analyzed clarification question patterns on question-answering (QA) websites in order to understand users behavior, and the types of clarification questions asked [38]. Trienes et al. focused on detecting when the original questions in community QA sites are unclear and clarification questions are needed [39]. Qu et al. curated and published a large dataset of question and answers intended to help develop conversational search systems [40]. In Web search, follow-up questions have been used for improving document retrieval for low-quality queries [6], [41], [42]. Targeting information that is missing from a document, Rao et al. used generative adversarial neural networks to automatically generate questions that seek to augment Amazon product reviews [5]. Asking follow-up questions has been explored in several other contexts such as chatbots [43], open domain question answering systems [44], [45], search engines [46], search within a Q&A forum [47], and image content [48].

VI. Conclusions and Future Work

This paper describes a technique for posing follow-up questions for incomplete bug reports that lack important information for triage, e.g., the bug's observable behavior. Our technique automatically selects follow-up questions from a corpus of such questions mined from the development histories of open source projects on GitHub. We first identify by using *tf*idf* a set of candidate follow-up questions whose original bug reports have high similarity to the deficient bug report of interest. Next, we use neural estimates of two metrics, compatibility and utility, to rank and select the optimal follow-up question to recommend. To evaluate our technique we curated a dataset of 25K bug reports from 6452 unique repositories and implemented four baselines. Our technique outperformed the baselines across the board, with a reasonable *Precision@1* score for our model of 0.49, i.e., nearly half of the top most recommended follow-up questions were considered valid. We also performed a survey of software developers which showed a follow-up question validity rate that aligned to the held-out dataset evaluation and also indicated that developers, at a high rate, considered the selected follow-up questions as: useful, specific, and asking for new information not contained in the bug report.

There are several avenues of future work. First, following Rao et al. [5], we can attempt to automatically generate follow-up questions using sequence-to-sequence neutral network models [49]–[51]. Second, following Braslavski et al. [38], we can work on generating frequently asked question patterns in bug reports. Third, we can develop a tool and integrate the Bug-AutoQ model with real world platforms like GitHub or JIRA in order to assist developers in the field and gather more developer feedback. Lastly, another line of future work can be on broadening the evaluation, since it is a vital challenge to determine the most relevant follow-up question for different software development contexts and requirements.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 1812968.

REFERENCES

[1] O. Chaparro, J. Lu, F. Zampetti, L. Moreno, M. Di Penta, A. Marcus, G. Bavota, and V. Ng, "Detecting missing information in bug descriptions," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 396–407.

[2] dear-github/dear-github, "https://github.com/dear-github/dear-github," 2020.

[3] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann, "Information needs in bug reports: improving cooperation between developers and users," in *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*, 2010, pp. 301–310.

[4] S. Rao and H. Daumé III, "Learning to ask good questions: Ranking clarification questions using neural expected value of perfect information," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, jul 2018, pp. 2737–2746.

[5] ——, "Answer-based Adversarial Training for Generating Clarification Questions," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 143–155. [Online]. Available: https://www.aclweb.org/anthology/N19-1013

[6] H. Zamani, S. Dumais, N. Craswell, P. Bennett, and G. Lueck, "Generating clarifying questions for information retrieval," in *Proceedings of The Web Conference 2020*, ser. WWW '20, 2020, p. 418–428.

[7] J. Zhang, X. Wang, D. Hao, B. Xie, L. Zhang, and H. Mei, "A survey on bug-report analysis," *Science China Information Sciences*, vol. 58, pp. 1–24, 2014.

[8] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss, "What makes a good bug report?" *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 618–643, 2010.

[9] E. I. Laukkanen and M. V. Mantyla, "Survey reproduction of defect reporting in industrial software development," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, 2011, pp. 197–206.

[10] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *Proceedings of the 11th Working Conference on Reverse Engineering*. IEEE, 2004, pp. 214–223.

[11] D. Shepherd, K. Damevski, B. Ropski, and T. Fritz, "Sando: an extensible local code search framework," in *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–2.

[12] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological bulletin*, vol. 114, no. 3, p. 494, 1993.

[13] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys," in *annual meeting of the Florida Association of Institutional Research*, 2006, pp. 1–33.

[14] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter, and C. Weiss, "What makes a good bug report?" *IEEE Transactions on Software Engineering*, vol. 36, pp. 618–643, 2010.

[15] E. Linstead and P. Baldi, "Mining the coherence of gnome bug reports with statistical topic models," in *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, 2009, pp. 99–102.

[16] D. Huo, T. Ding, C. McMillan, and M. Gethers, "An empirical study of the effects of expert knowledge on bug reports," *Proceedings of the International Conference on Software Maintenance and Evolution*, pp. 1–10, 2014.

[17] A. D. Sorbo, J. Spillner, G. Canfora, and S. Panichella, ""Won't we fix this issue?" qualitative characterization and automated identification of wontfix issues on github," *ArXiv*, vol. abs/1904.02414, 2019.

[18] M. Erfani Joorabchi, M. Mirzaaghaei, and A. Mesbah, "Works for me! characterizing non-reproducible bug reports," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 62–71.

[19] S. Davies and M. Roper, "What's in a bug report?" in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '14, 2014.

[20] T. Dal Sasso, A. Mocci, and M. Lanza, "What makes a satisficing bug report?" in *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2016, pp. 164–174.

[21] H. Liu, M. Shen, J. Jin, and Y. Jiang, "Automated classification of actions in bug reports of mobile apps," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 128–140.

[22] Y. Zhao, K. Miller, T. Yu, W. Zheng, and M. Pu, "Automatically extracting bug reproducing steps from android bug reports," in *Reuse in the Big Data Era*, X. Peng, A. Ampatzoglou, and T. Bhowmik, Eds. Springer International Publishing, 2019, pp. 100–111.

[23] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 253–262.

[24] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2012, pp. 70–79.

[25] O. Chaparro, J. M. Florez, U. Singh, and A. Marcus, "Reformulating queries for duplicate bug report detection," in *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 218–229.

[26] Y. Fan, X. Xia, D. Lo, and A. E. Hassan, "Chaff from the wheat: Characterizing and determining valid bug reports," *IEEE Transactions on Software Engineering*, vol. 46, no. 5, pp. 495–525, 2020.

[27] Y. Zhou, Y. Tong, R. Gu, and H. Gall, "Combining text mining and data mining for bug report classification," *Journal of Software: Evolution and Process*, vol. 28, no. 3, pp. 150–176, 2016.

[28] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 34–43.

[29] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk, "Auto-completing bug reports for android applications," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 673–686.

[30] O. R. Chaparro, C. Bernal-Cárdenas, J. Lu, K. Moran, A. Marcus, M. D. Penta, D. Poshyvanyk, and V. Ng, "Assessing the quality of the steps to reproduce in bug reports," *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[31] Y. Song and O. Chaparro, "Bee: A tool for structuring and analyzing bug reports," in *Proceedings of the 28th ACM Joint Meeting on the Foundations of Software Engineering (ESEC/FSE'20), Tool Demo Track*, 2020, to appear.

[32] L. Vanderwende, "The importance of being important: Question generation," in *Proceedings of the 1st Workshop on the Question Generation Shared Task Evaluation Challenge, Arlington, VA*, 2008.

[33] V. Rus, B. Wyse, P. Piwek, M. Lintean, S. Stoyanchev, and C. Moldovan, "Question generation shared task and evaluation challenge–status report," in *Proceedings of the 13th European Workshop on Natural Language Generation*, 2011, pp. 318–320.

[34] Q. Zhou, N. Yang, F. Wei, C. Tan, H. Bao, and M. Zhou, "Neural question generation from text: A preliminary study," in *National CCF Conference on Natural Language Processing and Chinese Computing*. Springer, 2017, pp. 662–671.

[35] M. Heilman and N. A. Smith, "Good question! statistical ranking for question generation," in *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, 2010, pp. 609–617.

[36] N. Duan, D. Tang, P. Chen, and M. Zhou, "Question generation for question answering," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 2017, pp. 866–874.

[37] X. Du, J. Shao, and C. Cardie, "Learning to ask: Neural question generation for reading comprehension," *arXiv preprint arXiv:1705.00106*, 2017.

[38] P. Braslavski, D. Savenkov, E. Agichtein, and A. Dubatovka, "What do you mean exactly? analyzing clarification questions in CQA," in *Proceedings of the 2017 Conference on Conference Human Information Interaction and Retrieval*, ser. CHIIR '17. New York, NY, USA:

Association for Computing Machinery, 2017, p. 345–348. [Online]. Available: https://doi.org/10.1145/3020165.3022149

[39] J. Trienes and K. Balog, "Identifying unclear questions in community question answering websites," in *Proceedings of the European Conference on Information Retrieval*. Springer, 2019, pp. 276–289.

[40] C. Qu, L. Yang, W. B. Croft, J. R. Trippas, Y. Zhang, and M. Qiu, "Analyzing and characterizing user intent in information-seeking conversations," in *Proceedings of the 41st International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 989–992. [Online]. Available: https://doi.org/10.1145/3209978.3210124

[41] M. Aliannejadi, H. Zamani, F. Crestani, and W. B. Croft, "Asking clarifying questions in open-domain information-seeking conversations," in *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR'19, 2019, p. 475–484.

[42] S. Stoyanchev, A. Liu, and J. Hirschberg, "Towards natural clarification questions in dialogue systems," in *AISB symposium on questions, discourse and dialogue*, vol. 20, 2014.

[43] B. Hancock, A. Bordes, P.-E. Mazaré, and J. Weston, "Learning from dialogue after deployment: Feed yourself, chatbot!" in *ACL*, 2019.

[44] M. De Boni and S. Manandhar, "Implementing clarification dialogues in open domain question answering," *Natural Language Engineering*, vol. 11, no. 4, pp. 343–362, 2005.

[45] ——, "An analysis of clarification dialogue for question answering," in *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, 2003, pp. 48–55.

[46] P. Ren, Z. Chen, Z. Ren, E. Kanoulas, C. Monz, and M. de Rijke, "Conversations with search engines," *ArXiv*, vol. abs/2004.14162, 2020.

[47] N. Zhang, Q. Huang, X. Xia, Y. Zou, D. Lo, and Z. Xing, "Chatbot4qr: Interactive query refinement for technical question retrieval," *IEEE Transactions on Software Engineering*, 2020.

[48] N. Mostafazadeh, I. Misra, J. Devlin, M. Mitchell, X. He, and L. Vanderwende, "Generating natural questions about an image," *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016. [Online]. Available: http://dx.doi.org/10.18653/v1/P16-1170

[49] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[50] J. Yin, X. Jiang, Z. Lu, L. Shang, H. Li, and X. Li, "Neural generative question answering," *arXiv preprint arXiv:1512.01337*, 2015.

[51] I. V. Serban, A. Sordoni, Y. Bengio, A. Courville, and J. Pineau, "Building end-to-end dialogue systems using generative hierarchical neural network models," *arXiv preprint arXiv:1507.04808*, 2015.