10-2022

Taming multi-output recommenders for software engineering

Christoph TREUDE

Singapore Management University, ctreude@smu.edu.sg

Taming Multi-Output Recommenders for Software Engineering

Christoph Treude
christoph.treude@unimelb.edu.au
The University of Melbourne
Melbourne, Victoria, Australia

ABSTRACT

Recommender systems are a valuable tool for software engineers. For example, they can provide developers with a ranked list of files likely to contain a bug, or multiple auto-complete suggestions for a given method stub. However, the way these recommender systems interact with developers is often rudimentary—a long list of recommendations only ranked by the model's confidence. In this vision paper, we lay out our research agenda for re-imagining how recommender systems for software engineering communicate their insights to developers. When issuing recommendations, our aim is to recommend diverse rather than redundant solutions and present them in ways that highlight their differences. We also want to allow for seamless and interactive navigation of suggestions while striving for holistic end-to-end evaluations. By doing so, we believe that recommender systems can play an even more important role in helping developers write better software.

CCS CONCEPTS

• **Information systems** → **Recommender systems**; *Information retrieval diversity*; • **Human-centered computing** → **Interactive systems and tools**; **Interaction paradigms**; • **Software and its engineering** → *Source code generation*.

KEYWORDS

Recommender systems, software engineering, user interaction, information retrieval diversity, information representation

1 INTRODUCTION

A recommender system for software engineering is defined as "a software application that provides information items estimated to be valuable for a software engineering task in a given context" [31]. These systems aim to assist developers in various activities from reusing code to writing effective bug reports. Recent years have witnessed the use of machine learning and deep learning in many recommender systems, making them more accurate and efficient [28].

In terms of output, recommender systems for software engineering are meant provide value by "exposing users to the most interesting items, and by offering novelty, surprise, and relevance" [31]. What this description implicitly conveys is true in practice for many recommender systems for software engineering: their benefits are realised through their ability to recommend multiple items for a given task in a given context. For example:

- Bug localisers provide a list of potentially buggy files ranked according to the system's confidence in the recommendation (e.g., [35]).
- Source code recommenders or synthesizers suggest multiple completions for the same source code context and task (e.g., [6]).
- Query reformulators provide a list of suggestions for better queries to allow more effective information retrieval (e.g., [7]).
- API recommenders provide a list of relevant API elements, such as methods, for a given scenario (e.g., [8]).
- Change location recommenders suggest multiple relevant spots for supplementary bug fixes (e.g., [36]).
- Tag recommenders for Stack Overflow and similar sites recommend multiple tags per post (e.g., [22]).
- Recommenders for 'who should fix this bug' provide multiple answers to this question [2].

The multi-output nature of recommender systems for software engineering is also reflected in the evaluation criteria that have been used in the research literature, such as the ratio of inputs for which at least one relevant result is returned within the top-k results (Hits@K), the ranks of relevant results within a ranked list (mean average precision), or the multiplicative inverse of the rank of the first relevant result (mean reciprocal rank) [30].

In the field of software engineering, there has been a plethora of work on improving the performance of recommender systems. However, surprisingly little research has been conducted on how these systems should communicate their insights to developers, especially in cases where the recommender system provides multiple recommendations for a given task in a given context. Many approaches in the literature are not accompanied by corresponding user interfaces, and in the few cases where a user interface is included, this is often limited to a list of recommendations ordered by the recommender's confidence in them. We argue that this does not do justice to the complexity of many software engineering tasks where multiple parallel recommendations need to be carefully compared and considered from multiple perspectives in order to understand complex solution spaces. Instead, many user interfaces ascribe too much importance to their top recommendation, often making navigation to further recommendations unnecessarily cumbersome. We show examples of this in our next section.

Re-imagining how multi-output recommenders for software engineering communicate their insights to developers can unlock the potential of recommender systems that have already been developed, as well as pave the way for new systems that explicitly cater to the multi-solution nature of software engineering tasks. The goals of our research agenda are:

Diversification Novelty and diversity have long been used as metrics for the evaluation of information retrieval systems in other fields [10]. We argue that diversity also plays a crucial role when exploring the solution space for software engineering tasks—providing diverse recommendations promotes serendipity and creativity [38] and can help resolve ambiguity and avoid redundancy.

Representation In many software engineering tasks, such as the synthesis of source code, small differences between recommendations completely change the semantics of the solution, as evidenced, for example, by small syntactic changes (such as reversing a logical operator) leading to large semantic changes [18], or the large number of single-statement bugs in source code [20]. We argue that recommender systems should visually support developers in comparing and contrasting multiple recommendations.

Navigation Recommender systems are often implemented as a separate window or tab from the main application, with multiple recommendations displayed at once. We argue for an integrated approach that enables requests for further recommendations that combine specific aspects of previous ones. This would allow developers to easily navigate between recommendations, similar to NLP2Code's 'cycle-through' functionality [6]. This kind of navigation support is essential to allow developers to adequately explore the solution space before deciding which recommendation to follow.

Evaluation Like other machine-learning-enabled systems, we argue that the evaluation of recommender systems for software engineering needs to include "evaluating components individually (including the model) as well as their integration and the entire system, often including evaluating and monitoring the system online (in production)" [25]. In particular, we call for more research on how developers do or do not make use of multiple recommendations issued for a given task in a given context.

The remainder of this vision paper is structured as follows. We provide three motivating examples in Section 2 before we describe our research agenda in Section 3. Section 4 concludes this work.
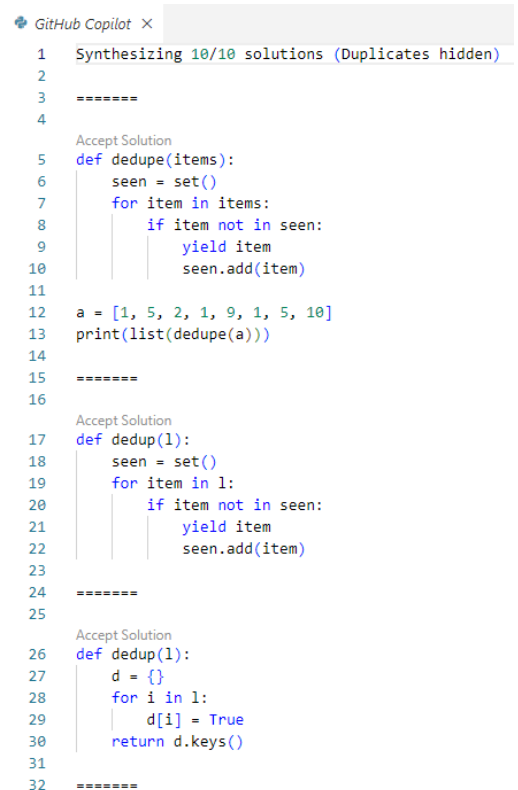
2 MOTIVATING EXAMPLES

In this section, we describe three motivating examples from GitHub Copilot[1], BugLocalizer [35], and SEQUER [7].

2.1 GitHub Copilot

GitHub Copilot is an example of a source code recommender system built by training a deep learning model on millions of open source repositories: The source code of these repositories acts as training data, allowing the model to learn "how to program" [9]. GitHub

```
GitHub Copilot  ×
1    Synthesizing 10/10 solutions (Duplicates hidden)
2
3    =======
4
     Accept Solution
5    def dedupe(items):
6        seen = set()
7        for item in items:
8            if item not in seen:
9                yield item
10               seen.add(item)
11
12   a = [1, 5, 2, 1, 9, 1, 5, 10]
13   print(list(dedupe(a)))
14
15   =======
16
     Accept Solution
17   def dedup(l):
18       seen = set()
19       for item in l:
20           if item not in seen:
21               yield item
22               seen.add(item)
23
24   =======
25
     Accept Solution
26   def dedup(l):
27       d = {}
28       for i in l:
29           d[i] = True
30       return d.keys()
31
32   =======
```

Figure 1: Three suggestions by GitHub Copilot for
 # deduplicate list
def

Copilot was released in October 2021 as a technical preview and can be installed as a Visual Studio Code extension.[2] Once installed, GitHub Copilot automatically suggests the code that it thinks the developer might want as the developer is typing.

While GitHub Copilot will always show its "best recommendation" in the editor,[3] developers can press `Ctrl + Enter` to view up to ten suggestions in a separate pane.[4] Figure 1 shows a screenshot of this pane, containing three of the ten suggestions that GitHub Copilot issued for the following Python code:

```
# deduplicate list
def
```

The first of these suggestions was automatically inserted into the source code as an inline suggestion and could be accepted by pressing the Tab key.

Considering the quality of the recommendations—all recommendations appear to produce the desired behaviour—the way in which the additional suggestions are communicated to a developer is surprisingly rudimentary. In terms of *diversity*, the methods suggested
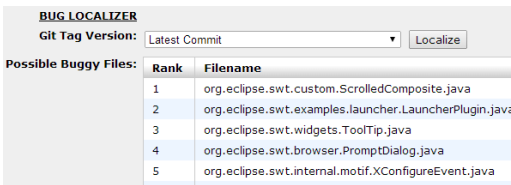
Figure 2: BugLocalizer Main User Interface, from [35]

Figure 3: SEQUER Browser Extension, from [7]

in the first two recommendations are identical apart from the identifier names, but as the third recommendation shows, this is not the only way to complete the task. In terms of *representation*, the differences between the first two recommendations are not immediately obvious: upon closer inspection, the first recommendation includes a method invocation with an example list, which is missing in the second recommendation. In terms of *navigation*, interactions between different recommendations are not taken into account, e.g., it is not easily possible to choose the third recommendation but with the example and `print` statement from the first recommendation.

2.2 BugLocalizer

Bug localisation is another popular research area in the context of recommender systems for software engineering, defined as the process of identifying where to make changes in response to a bug report [24]. Research efforts in this area are often evaluated in terms of performance metrics that take the multi-output nature of the problem into account (e.g., Hits@K [30]), while research on how to communicate the results of bug localisation to developers is rare. A notable exception is BugLocalizer [35], an extension of the Bugzilla issue tracking system aimed at disseminating research in bug localisation to practitioners. Figure 2 shows its interface after processing a bug report, indicating the "top-*k* files that are deemed the most likely to be buggy among files in the latest commit of the project's git repository" [35].

Similarly to the previous example, we argue that the way in which bug localisation results are currently communicated to developers does not unlock the full potential of the underlying models. Although the paths and names of the files in Figure 2 suggest *diversity* at least within the `org.eclipse.swt` package, the *representation* does not help to understand how these recommendations differ from each other, e.g., to what extent the changes in these files were similar and/or related. In terms of *navigation*, it is not possible to navigate the recommendations along axes such as the call graph or package hierarchy, making the evaluation of each recommendation a tedious and manual task.

2.3 SEQUER

Query reformulation is a popular research area in software engineering [17], covering a wide range of developer tasks, from Stack Overflow search [7] and search for software projects [21] to concept location [29] and bug localisation [14]. In most of these scenarios, the research tools show a list of recommendations for reformulations to developers. Figure 3 shows an example of SEQUER [7], a browser plugin that automatically analyses the query content on Stack Overflow and recommends the top 10 query reformulation
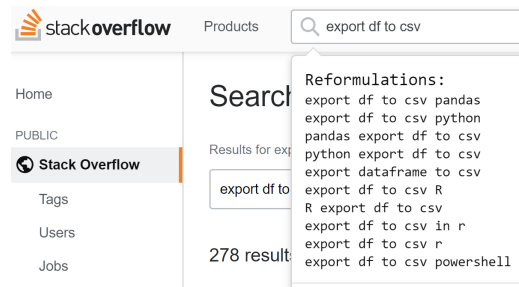
candidates to developers.[5] SEQUER was trained on the Stack Overflow activity logs and has been shown to be able to automatically correct misspelled terms, add language restrictions, remove overly specific query terms, replace symbols with their corresponding text, enclose domain-specific terms in double quotes, and simplify and refine Stack Overflow queries.

SEQUER outperformed its baselines in terms of metrics, such as the multiplicative inverse of the rank of the first relevant result (mean reciprocal rank), but we argue that the interface between the underlying model and developers has room for improvement. As Figure 3 shows, reformulation recommendations can be *diverse* in terms of characteristics such as programming languages, but this diversity is not well *represented*. In the example, the first five recommendations are specific to Python, the next four are specific to the R programming language, and the last one concerns PowerShell. In addition, several of the recommendations are redundant since they only change the order of tokens, e.g., the first and third recommendations. We believe that SEQUER could make better use of the limited number of spots in its list of recommendations by favouring diverse recommendations over redundant ones. In terms of *navigation*, the interface does not offer navigation along axes such as programming language, library, or API element. Like many similar tools, the *evaluation* of SEQUER focused on the performance of the underlying model, without an end-to-end evaluation to investigate how developers make use of the tool's multiple outputs.

3 RESEARCH AGENDA

In this section, we outline our envisioned methodology for addressing our research goals. These goals follow directly from our motivating examples, but we acknowledge that there are likely other goals worthy of investigation in this context.

3.1 Diversification

To address our goal of diversifying the output of multi-output recommenders for software engineering, we argue for the adoption and customisation of work in the area of diversity optimisation for other application domains, such as the evolution of diverse sets of images [27]. Diversity optimisation is "a new family of optimisation algorithms that, instead of searching for a single optimal solution to solving a task, searches for a large collection of solutions that all solve the task in a different way" [11]. Such optimisations require an underlying numeric representation of the problem domain,

---

[5]https://github.com/kbcao/sequer

similar to other software engineering tasks, such as vulnerability prediction, where artefacts must be represented as numeric vectors to be used as input for deep learning models [1]. Such representations are commonly based on syntax or semantics, e.g., learning representations of source code at the level of tokens (e.g., abstract syntax trees) or statements (e.g, control flow graphs), with recent approaches combining low-level syntactic information from the local context and high-level semantic information from the global context into a single representation [19]. Representation of text-based artefacts, such as file names or auto-complete suggestions, can be achieved through domain-specific word embeddings [13]. We believe that the use of such techniques can increase the diversity of recommendations and enable developers to explore the entire solution space for their tasks. Similar approaches have been successful in the diversification of reply suggestions for instant messaging systems [12] and emails [5] where more suggestions have been shown to be particularly useful for non-native speakers and in fostering creativity [34].

3.2 Representation

To improve the representation of multiple recommendations, we argue for integrating concepts from change visualisation in the context of software engineering into user interfaces of recommender systems. For example, representing source code commits by mixing text-based diff information with visual representation and metrics characterising the changes has been well received by developers [16]. A key difference between diff representation and representing multiple recommendation is cardinality—a diff concerns two versions (or three in the case of a three-way merge [15]) whereas recommender systems for software engineering tend to return dozens of results for a given task in a given context. To overcome this limitation, the adoption of variant graphs [32] from the text processing community can allow the representation of commonality between recommendations and differences between them. We envision the use of boldface to indicate commonality between recommendations, as well as pop-over comments for explanations of differences, similar to Casdoc [26].

3.3 Navigation

To enable effective navigation of multiple recommendations, we argue for borrowing mechanisms from poker games. In any given poker game, players must make the decision of which cards to keep and which to discard [3]. This analogy can be used to help developers navigate recommendations. For example, when given recommendations about files that might be buggy, we envision that developers are able to highlight parts of the file paths that they would like to 'keep' before asking the recommender system to generate further recommendations. Similarly, in the scenario of source code synthesis, we envision developers able to highlight parts of the code that they would like to see again while discarding parts that are irrelevant. In the scenario shown in Figure 1, a developer might highlight the example list and `print` statement from the first recommendation but ask the system for a different algorithm to process the list. We also believe that an integrated 'cycle-through' functionality that shows recommendations in their context rather than in a separate window or pane, similar to NLP2Code [6] and NL2Code [37], will improve the navigation of multiple recommendations.

3.4 Evaluation

To confirm that addressing the research goals presented above does indeed improve how multi-output recommender systems for software engineering communicate their insights to developers, we argue for system and usability testing, similar to related work on the evaluation of machine-learning-enabled systems [25]. Although the software engineering research community has so far focused on evaluating model performance [23], we lack behind other fields in terms of real-world integration of these models [33]. Performance metrics such as the ranks of relevant results within a list (mean average precision) are insufficient to determine whether a single recommendation or the interaction of multiple recommendations led developers onto the right path for solving their tasks. There is 'no silver bullet' for many software engineering tasks [4] and it is naïve to assume that a single recommendation will be sufficient to solve a non-trivial software engineering task. We need to embrace the multi-output nature of recommender systems for software engineering and enable these systems to effectively communicate their insights. Participant observations or interviews would be suitable methods to assess whether we were able to achieve these goals, for example [33].

4 CONCLUSION

The models that power automated recommender systems for software engineering have seen impressive performance increases over the last few years. However, these advances are not always accompanied by adequate ways in which the systems communicate their insights to developers. This can lead to misunderstandings and frustration on the part of developers who may not be able to easily interpret or use the recommendations generated by these systems. In this vision paper, we lay out our research agenda for re-imagining how systems such as bug localisers, source code synthesisers, and API recommenders can enable developers to navigate the diverse solution spaces inherent in many software engineering tasks. Our research agenda calls for recommending diverse rather than redundant solutions, aligned with the 'no silver bullet' nature of many software engineering tasks. We envision a representation of recommendations that enables developers to effortlessly spot similarities and differences, as well as interactions between multiple recommendations, and navigation mechanisms that allow developers to ask for further recommendations that contain aspects of items already recommended. To evaluate whether we are making progress towards these goals, we will require holistic end-to-end system and usability evaluations of recommender systems. We believe that this work will not only improve the effectiveness of recommender systems for software engineering but also help to build a community of developers who are confident in their ability to use automated tools for software engineering tasks.

ACKNOWLEDGMENTS

REFERENCES

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2vec: Learning Distributed Representations of Code. *Proceedings of the ACM on Programming Languages* 3, Popl (2019), 1–29.

[2] John Anvik, Lyndon Hiew, and Gail C Murphy. 2006. Who Should Fix This Bug?. In *Proceedings of the 28th International Conference on Software Engineering*. 361–370.

[3] Catalin Barboianu. 2007. *Draw Poker Odds: The Mathematics of Classical Poker*. Infarom Publishing.

[4] Fred Brooks. 1987. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer* 20, 04 (1987), 10–19.

[5] Daniel Buschek, Martin Zürn, and Malin Eiband. 2021. The Impact of Multiple Parallel Phrase Suggestions on Email Input and Composition Behaviour of Native and Non-native English Writers. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–13.

[6] Brock Angus Campbell and Christoph Treude. 2017. Nlp2code: Code Snippet Content Assist Via Natural Language Tasks. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. 628–632.

[7] Kaibo Cao, Chunyang Chen, Sebastian Baltes, Christoph Treude, and Xiang Chen. 2021. Automated Query Reformulation for Efficient Search Based on Query Logs from Stack Overflow. In *Proceedings of the International Conference on Software Engineering*. 1273–1285.

[8] Chi Chen, Xin Peng, Zhenchang Xing, Jun Sun, Xin Wang, Yifan Zhao, and Wenyun Zhao. 2021. Holistic Combination of Structural and Textual Code Information for Context Based API Recommendation. *IEEE Transactions on Software Engineering* (2021). To Appear.

[9] Matteo Ciniselli, Luca Pascarella, and Gabriele Bavota. 2022. To What Extent Do Deep Learning-based Code Recommenders Generate Predictions by Cloning Code from the Training Set?. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 167–178.

[10] Charles La Clarke, Maheedhar Kolla, Gordon V Cormack, Olga Vechtomova, Azin Ashkan, Stefan Büttcher, and Ian Mackinnon. 2008. Novelty and Diversity in Information Retrieval Evaluation. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. 659–666.

[11] Antoine Cully. 2019. Autonomous Skill Discovery with Quality-diversity and Unsupervised Descriptors. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 81–89.

[12] Budhaditya Deb, Peter Bailey, and Milad Shokouhi. 2019. Diversifying Reply Suggestions Using a Matching-conditional Variational Autoencoder. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Industry Papers)*. 40–47.

[13] Vasiliki Efstathiou, Christos Chatzilenas, and Diomidis Spinellis. 2018. Word Embeddings for the Software Engineering Domain. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 38–41.

[14] Juan Manuel Florez, Oscar Chaparro, Christoph Treude, and Andrian Marcus. 2021. Combining Query Reduction and Expansion for Text-retrieval-based Bug Localization. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*. 166–176.

[15] Gleiph Ghiotto, Leonardo Murta, Márcio Barros, and Andre Van Der Hoek. 2018. On the Nature of Merge Conflicts: A Study of 2,731 Open Source Java Projects Hosted by GitHub. *IEEE Transactions on Software Engineering* 46, 8 (2018), 892–915.

[16] Verónica Uquillas Gómez, Stéphane Ducasse, and Theo D'hondt. 2015. Visually Characterizing Source Code Changes. *Science of Computer Programming* 98 (2015), 376–393.

[17] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. 2013. Automatic Query Reformulations for Text Retrieval in Software Engineering. In *Proceedings of the International Conference on Software Engineering*. 842–851.

[18] Jane Huffman Hayes and Jeff Offutt. 2002. Applying a Semantic Fault Model to the Empirical Study of Corrective Maintenance. In *Proceedings of the IEEE Workshop on Empirical Studies of Software Maintenance*.

[19] Yuan Jiang, Xiaohong Su, Christoph Treude, and Tiantian Wang. 2022. Hierarchical Semantic-aware Neural Code Representation. *Journal of Systems and Software* (2022), 111355.

[20] Rafael-michael Karampatsis and Charles Sutton. 2020. How Often Do Single-statement Bugs Occur? The ManySStuBs4J Dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 573–577.

[21] Zhixing Li, Tao Wang, Yang Zhang, Yun Zhan, and Gang Yin. 2016. Query Reformulation by Leveraging Crowd Wisdom for Scenario-based Software Search. In *Proceedings of the 8th Asia-pacific Symposium on Internetware*. 36–44.

[22] Jin Liu, Pingyi Zhou, Zijiang Yang, Xiao Liu, and John Grundy. 2018. Fasttagrec: Fast Tag Recommendation for Software Information Sites. *Automated Software Engineering* 25, 4 (2018), 675–701.

[23] Qinghua Lu, Liming Zhu, Xiwei Xu, Jon Whittle, and Zhenchang Xing. 2022. Towards a Roadmap on Software Engineering for Responsible AI. In *Proceedings of the 1st International Conference on AI Engineering – Software Engineering for AI*. 101–112.

[24] Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. 2014. On the Use of Stack Traces to Improve Text Retrieval-based Bug Localization. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. 151–160.

[25] Nadia Nahar, Shurui Zhou, Grace Lewis, and Christian Kästner. 2022. Collaboration Challenges in Building ML-enabled Systems: Communication, Documentation, Engineering, and Process. In *Proceedings of the International Conference on Software Engineering*. 413–425.

[26] Mathieu Nassif, Zara Horlacher, and Martin Robillard. 2022. Casdoc: Unobtrusive Explanations in Code Examples. In *Proceedings of the International Conference on Program Comprehension*. 631–635.

[27] Aneta Neumann, Wanru Gao, Carola Doerr, Frank Neumann, and Markus Wagner. 2018. Discrepancy-based Evolutionary Diversity Optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 991–998.

[28] Ivens Portugal, Paulo Alencar, and Donald Cowan. 2018. The Use of Machine Learning Algorithms in Recommender Systems: A Systematic Review. *Expert Systems with Applications* 97 (2018), 205–227.

[29] Mohammad Masudur Rahman and Chanchal K Roy. 2016. Quickar: Automatic Query Reformulation for Concept Location Using Crowdsourced Knowledge. In *Proceedings of the International Conference on Automated Software Engineering*. 220–225.

[30] Mohammad Masudur Rahman and Chanchal K Roy. 2018. Improving IR-based Bug Localization with Context-aware Query Reformulation. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 621–632.

[31] Martin Robillard, Robert Walker, and Thomas Zimmermann. 2009. Recommendation Systems for Software Engineering. *IEEE Software* 27, 4 (2009), 80–86.

[32] Desmond Schmidt and Robert Colomb. 2009. A Data Structure for Representing Multi-version Texts Online. *International Journal of Human-computer Studies* 67, 6 (2009), 497–514.

[33] Mark P Sendak, William Ratliff, Dina Sarro, Elizabeth Alderton, Joseph Futoma, Michael Gao, Marshall Nichols, Mike Revoir, Faraz Yashar, Corinne Miller, and Others. 2020. Real-world Integration of a Sepsis Deep Learning Technology into Routine Clinical Care: Implementation Study. *JMIR Medical Informatics* 8, 7 (2020), E15182.

[34] Nikhil Singh, Guillermo Bernal, Daria Savchenko, and Elena L Glassman. 2022. Where to Hide a Stolen Elephant: Leaps in Creative Writing with Multimodal Machine Intelligence. *ACM Transactions on Computer-Human Interaction* (2022). To Appear.

[35] Ferdian Thung, Tien-duy B Le, Pavneet Singh Kochhar, and David Lo. 2014. Buglocalizer: Integrated Tool Support for Bug Localization. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 767–770.

[36] Xin Xia and David Lo. 2017. An Effective Change Recommendation Approach for Supplementary Bug Fixes. *Automated Software Engineering* 24, 2 (2017), 455–498.

[37] Frank F Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-IDE Code Generation from Natural Language: Promise and Challenges. *ACM Transactions on Software Engineering and Methodology* 31, 2 (2022), 1–47.

[38] Reza Jafari Ziarani and Reza Ravanmehr. 2021. Serendipity in Recommender Systems: A Systematic Literature Review. *Journal of Computer Science and Technology* 36, 2 (2021), 375–396.