Repository-Level Prompt Generation for Large Language Models of Code

Anonymous Authors[1]

Abstract

With the success of large language models (LLMs) of code and their use as code assistants (e.g. Codex (Chen et al., 2021) used in GitHub Copilot[1]), development of techniques where we can have the capability to introduce domain-specific knowledge in the prompt design process becomes important. In this work, we propose a framework called Repo-Level Prompt Generator that learns to generate example-specific prompts using a set of rules. These rules allow us to take context from the entire repository, thereby incorporating both the structure of the repository and the context from other relevant files (e.g. imports, parent class files). Our technique doesn't require any access to the weights of the LLM, making it applicable in cases where we only have a black-box access to the LLM. We conduct experiments on the task of single-line code-autocompletion using code repositories taken from Google Code archives. We demonstrate that an oracle constructed from our proposed rules gives up to 36% relative improvement over Codex, showing the quality of our proposed rules. Further, we show that when we train a model to select the best rule, we can achieve significant performance gains over Codex.

1. Introduction

Large Language Models (LLMs) have shown promising results in the area of natural language processing, where models such as GPT-3 (Brown et al., 2020) and PaLM (Chowdhery et al., 2022) demonstrated remarkable performance. Recently, LLMs have been used for modeling source code as well (Austin et al., 2021; Fried et al., 2022; Xu et al., 2022a; Feng et al., 2020). Codex (Chen et al., 2021) is one of the earliest LLM that contains about 12 billion parameters and was trained on publicly available code from GitHub. In particular, Codex has been deployed as part of GitHub Copilot[1], which is an in-IDE code assistant.

Despite the success of LLM for code, as illustrated in Figure 1 of Xu et al. (2022a) and mentioned in their paper "the strongest models are not publicly available". For instance, OpenAI provides access *only* to the generated output from Codex via an API [2]. The model weights and data are unavailable. Under this setting, most of the existing prompt generation techniques (Li & Liang, 2021; Shin et al., 2020; Lester et al., 2021) can't be used because they require access to some or all of the weights of the LLM.

Automatically generating an example-specific prompt is challenging and remains an active area of research (Liu et al., 2021a; Reynolds & McDonell, 2021). With increasing use of LLMs as assistive agents to humans, it is important to generate prompts that are easier for humans to understand and intuitive enough for them to change if desired. In this work, we propose a framework called *Repo-Level Prompt Generator (RLPG)* that enables the capability to incorporate domain-specific knowledge in the prompt-designing process via a set of *rules*. One of the distinguishing features of RLPG is that it considers the code context not just from the current file, but from the entire repository, thereby allowing us to incorporate both the structure of the repository as well as the context in the other files in the repository. This makes particular sense for source code where context coming from outside the current file, such as imports and parent classes, can be very useful for predicting the hole. The choice of *where* from and *what* to take from the repository is specified by each rule. For example, we could have a rule that returns all the identifiers used in the first import file. The prompts generated by our rules are discrete, similar to suggested in some previous works in NLP (Shin et al., 2020; Schick & Schütze, 2021). However, rather than fixing one particular rule for each example, we propose a *Rule Classifier* neural module that *learns* to come up with the best rule conditioned on that example, such that the resulting prompt is likely to produce the desired output.

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the ICML 2022 workshop on Knowledge Retrieval and Language Models. Do not distribute.

[1]https://copilot.github.com/

[2]https://openai.com/blog/openai-codex/

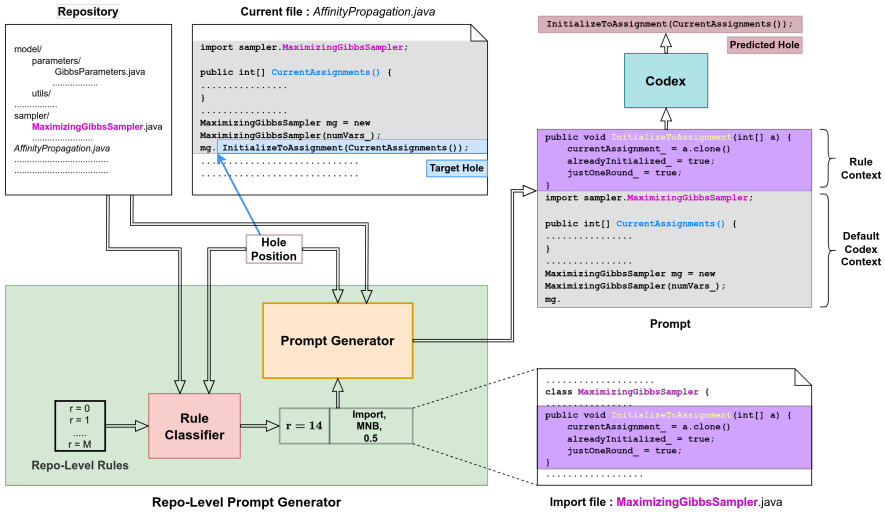

Figure 1: **Repo-Level Prompt Generator:** The prompt is generated by combining the rule context corresponding to $r = 14$ (method names and bodies from the imported file, `MaximizingGibbsSampler.java`) with the default Codex context. Codex then uses this prompt to output its predicted hole.

In this work, we focus on the task of single-line code-autocompletion in a line-level maintenance setting (Shrivastava et al., 2020) in an IDE, where the objective is to predict the blanked-out portion (or *target hole*) starting from the position of an imagined cursor to the end of line. There might also be code that follows below this line. This reflects the setting where a user is editing an existing file. Figure 1 provides an illustration of our system in progress. The rule classifier takes in the hole position (position of the cursor) in the current file, the repository to which the current file is a part of and a list of rules as input, and chooses a particular rule to be used for that hole prediction. In the figure, the chosen rule corresponds to taking the method names and bodies from the import file (i.e., *rule context*). The *Prompt Generator* uses this rule context and concatenates it with 50% of the default Codex context to get the prompt. The default Codex context consists of code prior to the position of the hole in the current file. The generated prompt can then be given as input to Codex so it can make a prediction for the hole (shown in the brown box on the top) as output. Our key contributions can be listed as follows:

- We propose a framework called the *Repo-Level Prompt Generator (RLPG)* that *learns* to generate prompts conditioned on the example, without requiring access to the weights of the LLM.
- To incorporate domain-knowledge in the prompt design process, RLPG uses a set of repository-level rules. These rules are designed to incorporate both the structure of the repository as well as the relevant context from other files in the repository.
- On the task of single-line code-autocompletion using code repositories in Java, we show that an oracle constructed from our proposed rules gives up to 36% relative improvement over Codex. Further, we show that when we use our rule classifier to select the best rule, we can achieve up to 17% relative improvement over Codex.

2. Repo-Level Prompt Generator (RLPG)

In this section, we provide details of our framework. We start by describing our proposed rules and then follow with a discussion about our rule classifier. Finally, we briefly describe the Prompt Generator module.

2.1. Repo-Level Rules

The core idea of RLPG consists of substituting part of the default Codex context with context coming from somewhere else in the repository. The decision of what to take and from where in the repository to take from is governed by a set of rules. These rules were decided based on manual inspection of our training data and intend to capture common coding patterns. A rule can be thought of as a function that takes in a target hole's position and the repository that the hole is a part of as input, and returns the *rule context* (string). A rule is specified by three quantities: (a) rule context location, (b) rule context type and (c) rule context fraction. We mention each of these along with the motivation for them below.

Rule Context Location: For a target hole position, a rule's context location determines from *where* should we take code that will be part of the rule context. We propose ten different rule context locations: (a) Current, (b) Parent Class, (c) Import, (d) Sibling, (e) Similar Name, (f) Child Class, (g) Import of Parent Class, (h) Import of Sibling, (i) Import

of Similar Name, and (j) Import of Child Class. For each rule context location, we use a heuristic to get either a single file or a ranked list of files. In the latter case, we will take context from these files until we exhaust the maximum context length allocated to the rule. See Appendix A.1 for details on the rule context locations and Appendix A.4 for details of the ranking of files based on their rule context location.

Rule Context Type: The rule context type determines *what* code to take from the rule context location. We propose seven different rule context types: (a) Post Lines (PL), (b) Identifiers (I), (c) Type Identifiers (TI), (d) Field Declarations (FD), (e) String Literals (SL), (f) Method Names (MN), and (g) Method Names and Bodies (MNB). See Appendix A.2 for more details.

Rule Context Fraction: The rule context fraction determines the fraction of the final prompt that will be made of the code coming from the rule context. We use rule context fraction values of 0.25, 0.5 and 0.75. Appendix A.3 for a list of our proposed rules. Note that depending on the target hole, not all rules will be applicable (e.g. if there are no parent classes in the current file, rules 8-13 and 50-55 won't be applicable.

2.2. Rule Classifier

Given a hole position, the purpose of the rule classifier is to select the best rule r that is likely to produce a prompt that will lead to success, where success happens when the target hole h exactly matches the predicted hole $\hat{h}$. This task is formulated as a multi-label binary classification problem because for a given target hole, more than one rule can lead to success. In this formulation, we treat using the default Codex context as complete prompt, as one of the rules. Next, we describes the training procedure for the rule classifier.

Training: For each target hole h, we generate a ground-truth vector $Y^h = \{y_r^h\}_{r=1}^{M}$ which is a multi-hot vector of size M, where M is the total number of rules. This vector is obtained by feeding the prompt generated by using the context from rule r into Codex and then seeing whether $\hat{h} = h$. If there is a match, we say that the rule r is successful. For hole h if a rule r is applicable and leads to success, $y_r^h = 1$ or will be zero otherwise. For each hole h, we obtain a mask T^h where $T_r^h = 1$ when r is applicable or $T_r^h = 0$ otherwise. The overall training loss, $\mathcal{L}$ can be expressed as follows:

$$\mathcal{L} = \frac{1}{N} \sum_{h=1}^{N} \frac{1}{M^h} \sum_{r=1}^{M^h} BCE(\hat{y}_r^h, y_r^h) * T_r^h \qquad (1)$$

In the above equation, N is the total number of holes encountered while training, M^h denotes the total number of applicable rules for h and BCE corresponds to the binary cross entropy loss. Masking ensures that we consider only the rules that are applicable. Next, we describe two variants of our rule classifier that can be used to obtain the prediction $\hat{y}_r^h$.

RLPG-H: Let H^h be the *hole window* that includes code present around the hole h excluding the hole itself. In our work, we take two lines before the hole position, the code up to the hole position and two lines after the hole position. We use a pretrained model F_ϕ to obtain a context representation vector of size Z where Z is the dimension of the hidden state of the model F_ϕ. Specifically, we take the hidden state at the first position, i.e. the representation of the `[CLS]` token. To make training of the rule classifier computationally efficient, the parameters ϕ are frozen during training. The RLPG-H model takes the context representation of the hole window and projects it to the rule space of size M via two dense layers with a non-linearity in between. Taking sigmoid of this output gives the prediction of the rule. See Appendix D.1 for more details.

RLPG-R: The motivation behind this variant is to use the similarity in the hole window and the rule context to determine which rule can be useful. Given a particular hole h, let C_r^h denote the rule context from rule r . Intuitively, if the hole window contains variables, say identifiers that are similar to the variables in the rule context, then there are chances that h might occur somewhere in C_r^h. The similarity is modeled using an attention mechanism, by treating the projected hole window representation Q^h as a query and the projected rule context representation, K_r^h (Equation 3) as a key. The value V_r^h is same as the key. See Appendix D.2 for more details.

2.3. Prompt Generator

The Prompt Generator takes the rule r chosen by the rule classifier, the repository and the hole position as input and generates the prompt as output. The rule context is always added before the default Codex context.

3. Experiments and Results

Dataset Creation: We scraped Google Code [3] for repositories in Java. We divided the repositories into train, validation and test splits where each repository in its entirety is part of a split. In each file within a repository, we remove lines that are blanks and comments, and set the hole position to be the middle character in the line. See Appendix C for complete details.

Methods: The details of the methods used in our experiments is given below. Foe each method, to measure success, we used exact match between the predicted hole string gener-

[3] https://code.google.com/archive/

Data Split	SR Codex(%)	SR Oracle(%)	Rel. ↑ over Codex(%)
Train	59.78	80.29	34.31
Val	62.10	79.05	27.28
Test	58.73	79.63	35.58

Method	Success Rate(%) (hole-wise)	Rel. ↑(%) (hole-wise)	Success Rate(%) (repo-wise)	Rel. ↑(%) (repo-wise)
Codex (Chen et al., 2021)	58.73	-	60.64	-
Oracle	79.63	35.58	80.24	32.31
Fixed Rule ($k = 1$)	65.78	12.00	68.01	12.15
RLPG-H ($k = 1$)	**68.51**	**16.65**	69.26	14.21
RLPG-R ($k = 1$)	67.80	15.44	**69.28**	**14.26**

Figure 2: ***(Left)*** Performance of the oracle relative to Codex; ***(Right)*** Success Rate (SR) of different methods on the test data when averaged across all holes (hole-wise) and across individual repositories (repo-wise)

ated by Codex and the target hole string. In our experiments, we report this fraction, called *success rate* (SR).

1. **Codex:** Using the default Codex context as prompt.
2. **Oracle:** Using the ground-truth vector Y^h (mentioned in Section 2.2). The prompt generated corresponds to using any of the successful rules with $y_r^h = 1$. Since this information is not available at inference, the oracle performance represents an upper bound.
3. **Fixed Rule (k):** Using k prompts generated with top-k rules. The k rules are fixed for all target holes. The top-k rules were decided based on decreasing order of success rate of the individual rules on the validation dataset. When $k = 1$, this corresponds to generating a single prompt using a single rule. In our case, it was $r = 7$, which corresponds to taking 0.75 fraction of post lines from the current file.
4. **RLPG-H (k) and RLPG-R (k):** Using k prompts generated by top-k rules. The top-k rules correspond to the top-k rule predictions given by the RLPG-H and RLPG-R variants of our rule classifier.

3.1. Results

Performance of Proposed Rules: The left part of Table 2 shows the performance of an oracle constructed from our proposed rules. We see that across all data splits, the proposed rules contribute to large relative improvements when compared to Codex. This result was pleasantly surprising to us as Codex has not been trained on prompts that consist of context other than the default Codex context. These results display that incorporating repo-level knowledge in the form of rules can be quite effective strategy for generating useful prompts.

Performance of the Rule Classifier: Having seen promise in our proposed rules, next we present the results of RLPG, which for each target hole, predicts a single best rule that can lead to success. The right part of Table 2 presents the success rates along with the percentage of relative improvements for the test data. The second and third columns correspond to the averages across all holes in the test data. The last two columns correspond to the average success rate of individual repositories. The latter metric doesn't account for the size of the repository. As can be seen from the table, both the RLPG variants as well as fixed rule improve the performance significantly over Codex, again highlighting the value of the proposed rules. In addition, both the RLPG variants show improved performance when compared to fixed rules highlighting the importance of learning a rule classifier. See Appendix B.3 for performance of all methods on individual repositories and Appendix B.1, Appendix B.2 and Appendix B.4 for experiments with scaling behaviour, rule-wise performance and analysis of sample cases.

4. Related Work

There are two broad categories of prompt generation techniques. The first category corresponds to producing continuous/soft prompts where the prompt is described in the latent space of a language model (Li & Liang, 2021; Qin & Eisner, 2021; Bragg et al., 2021; Lester et al., 2021; Liu et al., 2021b). The second category produces discrete prompts where the prompt is a text string that can be interpreted by a human (Shin et al., 2020; Gao et al., 2021; Schick & Schütze, 2021). Our work falls in the category of discrete prompt generation techniques as we produce a prompt consisting of code tokens that can be easily interpreted by a human. However, in contrast to prior works that use a set of fixed templates for all examples, we learn to generate prompts conditioned on each example. Another important distinction is that we do not require access to the weights of the LM. To the best of our knowledge, our work is the first to explore automatic prompt generation in the domain of source code. We provide additional related works in Appendix G.

5. Conclusions and Future Directions

We present RLPG, a framework that learns to automatically generate prompts conditioned on the example without requiring access to the weights of the LLM. RLPG utilizes the structure of the repository as well as the context from other files in the repository using a set of easy to understand rules. For future work, we want to explore the composition of contexts coming from two or more rules to generate a prompt and incorporating user's feedback in RLPG.

References

Allamanis, M. The adverse effects of code duplication in machine learning models of code, 2018. URL https://arxiv.org/abs/1812.06469.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer normalization, 2016.

Black, S., Biderman, S., Andonian, A., Anthony, Q., Gali, P., Gao, L., Hallahan, E., Levy-Kramer, J., Leahy, C., Nestler, L., Parker, K., Phang, J., Pieler, M., Purohit, S., Songz, T., Wang, P., and Weinbach, S. GPT-NeoX: Large scale autoregressive language modeling in pytorch, 2021a. URL http://github.com/eleutherai/gpt-neox.

Black, S., Gao, L., Wang, P., Leahy, C., and Biderman, S. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, March 2021b. URL https://doi.org/10.5281/zenodo.5297715. If you use this software, please cite it using these metadata.

Bragg, J., Cohan, A., Lo, K., and Beltagy, I. FLEX: Unifying evaluation for few-shot NLP. In Beygelzimer, A., Dauphin, Y., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=_WnGcwXLYOE.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W.-t., Zettlemoyer, L., and Lewis, M. Incoder: A generative model for code infilling and synthesis, 2022. URL https://arxiv.org/abs/2204.05999.

Gao, T., Fisch, A., and Chen, D. Making pre-trained language models better few-shot learners. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 3816–3830, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.295. URL https://aclanthology.org/2021.acl-long.295.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Kanade, A., Maniatis, P., Balakrishnan, G., and Shi, K. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 12-18 July 2020*, Proceedings of Machine Learning Research. PMLR, 2020.

Khandelwal, U., Levy, O., Jurafsky, D., Zettlemoyer, L., and Lewis, M. Generalization through memorization: Nearest neighbor language models. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=HklBjCEKvH.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In Bengio, Y. and LeCun, Y. (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

Lester, B., Al-Rfou, R., and Constant, N. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 3045–3059, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. URL https://aclanthology.org/2021.emnlp-main.243.

Li, X. L. and Liang, P. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 4582–4597, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.353. URL https://aclanthology.org/2021.acl-long.353.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., Hubert, T., Choy, P., d'Autume, C. d. M., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D. J., Robson, E. S., Kohli, P., de Freitas, N., Kavukcuoglu, K., and Vinyals, O. Competition-level code generation with alphacode, 2022. URL https://arxiv.org/abs/2203.07814.

Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., and Neubig, G. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing, 2021a. URL https://arxiv.org/abs/2107.13586.

Liu, X., Zheng, Y., Du, Z., Ding, M., Qian, Y., Yang, Z., and Tang, J. Gpt understands, too. *arXiv:2103.10385*, 2021b.

Qin, G. and Eisner, J. Learning how to ask: Querying LMs with mixtures of soft prompts. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 5203–5212, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.410. URL https://aclanthology.org/2021.naacl-main.410.

Reynolds, L. and McDonell, K. Prompt programming for large language models: Beyond the few-shot paradigm, 2021. URL https://arxiv.org/abs/2102.07350.

Schick, T. and Schütze, H. Exploiting cloze-questions for few-shot text classification and natural language inference. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pp. 255–269, Online, April 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.eacl-main.20. URL https://aclanthology.org/2021.eacl-main.20.

Shin, T., Razeghi, Y., IV, R. L. L., Wallace, E., and Singh, S. AutoPrompt: Eliciting knowledge from language models with automatically generated prompts. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2020.

Shrivastava, D., Larochelle, H., and Tarlow, D. On-the-fly adaptation of source code models. In *NeurIPS 2020 Workshop on Computer-Assisted Programming*, 2020. URL https://openreview.net/forum?id=FeVaSthrFst.

Tunstall, L., von Werra, L., and Wolf, T. *Natural Language Processing with Transformers*. " O'Reilly Media, Inc.", 2022.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, u., and Polosukhin, I. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pp. 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

Wang, B. and Komatsuzaki, A. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax, May 2021.

Wu, Y., Rabe, M. N., Hutchins, D., and Szegedy, C. Memorizing transformers. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=TrjbxzRcnf-.

Xu, F. F., Alon, U., Neubig, G., and Hellendoorn, V. J. A systematic evaluation of large language models of code. *arXiv preprint arXiv:2202.13169*, 2022a.

Xu, F. F., He, J., Neubig, G., and Hellendoorn, V. J. Capturing structural locality in non-parametric language models. In *International Conference on Learning Representations*, 2022b. URL https://openreview.net/forum?id=nnU3IUMJmN.

Yue Wang, Weishi Wang, S. J. S. C. H. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*, 2021.

Zhang, J., Panthaplackel, S., Nie, P., Mooney, R. J., Li, J. J., and Gligoric, M. Learning to generate code comments from class hierarchies, 2021. URL https://arxiv.org/abs/2103.13426.

A. Repo-Level Rules

A.1. Rule Context Location

1. **Current:** take code from the current file. The current file is the file that contains the target hole. The code in the current file (e.g. the lines after the hole position) can be very useful in predicting the target hole.
2. **Parent Class:** take code from the file that contains the parent class of the target hole. The intuition behind this is to account for cases where a method present in the parent class is invoked in the current file (i.e. the child class).
3. **Import:** take code from the import files used in the current file. The intuition here is same as parent class.
4. **Sibling:** take code from the files that are in the same directory as the current file. Files in the same directory tend to share code variables (e.g. identifiers).
5. **Similar Name:** take code from files that have a similar name as the current file. Similar names are determined by doing a splitting of the file name based on underscore or camel-case formatting and then matching parts of the filename. If one or more parts matches, two files are considered to have similar names. The intuition behind this is that developers tend to name files based on the functionality of the code written in that file. Therefore, a similar name file might contain some portion of the code that is common with the current file and hence might be useful for predicting the target hole.
6. **Child Class:** take code from files that have the current file as their parent class file.
7. **Import of Parent Class:** take code from the import files used in the parent class files.
8. **Import of Sibling:** take code from the import files used in the sibling files.
9. **Import of Similar Name:** take code from the import files used in the similar name files.
10. **Import of Child Class:** take code from the import files used in the child class files.

The last four rule context locations are useful when the target hole occurs at the very beginning of the current file where there might be less rule context coming from the other rule context locations.

A.2. Rule Context Type

1. **Post Lines (PL):** Take all the lines after the target hole line. This context type is applicable only when the rule context location is the current file. For the example shown in Figure 1 of the main paper, post lines will take all the lines after the line `mg.InitializeToAssignment(CurrentAssignments())` till we reach the end of the file (`AffinityPropagation.java`).
2. **Identifiers (I):** Take all the identifiers used in the rule context location. Identifiers are the names of variables used in the code. For example, for the rule context taken from the imported file shown in Figure 1 in the main paper (highlighted in violet), identifiers are `InitializeToAssignment` (line 1), `a` (line 1), `currentAssignment_` (line 2), `a` (line 2), `clone` (line 2), `alreadyInitialized_` (line 3), `justOneRound_` (line 4).
3. **Type Identifiers (TI):** Take all the type identifiers used in the rule context location. Type Identifiers define the type of an identifier. For example, in the code snippet `class DPAffinityPropagation extends AffinityPropagation , [` `AffinityPropagation` is labeled as a type identifier. Similarly in the snippet `DPAPParameters parameters_;`, `DPAPParameters` is a type identifier.
4. **Field Declarations (FD):** Take all the field declarations used in the rule context location. The variables of a class type are introduced by field declarations. For example, `double[][] mHijMujT_;` and `MessageValuePair[][] sortedMHijMujTs_;` are examples of field declarations.
5. **String Literals (SL):** Take all the string literals used in the rule context location. A string literal is the sequence of characters enclosed in double-quotes. For example, in the code snippet, `System.err.println("DPAP load Warning: unknown parameter " + entries[0] + ", value = " + entries[1]);`, we have two string literals: (a) `"DPAP load Warning: unknown parameter "` ; (b) `", value = "` .
6. **Method Names (MN):** Take all the method names along with their signature used in the rule context location. For the example shown in Figure 1 of the main paper, `public void InitializeToAssignment(int[] a)` is the method name rule context type.
7. **Method Names and Bodies (MNB):** Take all the method names along with their signatures and corresponding bodies used in the rule context location. For the example shown in Figure 1 of the main paper, the part highlighted in violet represents the method names and bodies.

A.3. List of Repo-Level Rules

Table 1: **List of our proposed repo-level rules**

Rule ID	Rule Context Location	Rule Context Type	Rule Context Fraction
0, 1, 2, 3, 4	Current	MN, I, TI, SL, FD	0.5
5, 6, 7	Current	PL	0.5, 0.25, 0.75
8, 9, 10, 11, 12, 13	Parent Class	MNB, MN, I, TI, SL, FD	0.5
14, 15, 16, 17, 18, 19	Import	MNB, MN, I, TI, SL, FD	0.5
20, 21, 22, 23, 24, 25	Sibling	MNB, MN, I, TI, SL, FD	0.5
26, 27, 28, 29, 30, 31	Similar Name	MNB, MN, I, TI, SL, FD	0.5
32, 33, 34, 35, 36, 37	Child Class	MNB, MN, I, TI, SL, FD	0.5
38, 39, 40, 41, 42, 43	Import of Sibling	MNB, MN, I, TI, SL, FD	0.5
44, 45, 46, 47, 48, 49	Import of Similar Name	MNB, MN, I, TI, SL, FD	0.5
50, 51, 52, 53, 54, 55	Import of Parent Class	MNB, MN, I, TI, SL, FD	0.5
56, 57, 58, 59, 60, 61	Import of Child Class	MNB, MN, I, TI, SL, FD	0.5
62	Codex	-	-

A.4. Ranking of files based on rule context location

In Table 2, we provide details of the heuristics we used for selecting files for a given rule context location. Depending on the rule, we get either a single file or a list of files ranked based on some criteria. For example, if the rule context location is Import, we take all the import statements used in the current file and identify the location in the current file where the corresponding imports have been used. According to our heuristic, the closer is the import usage to the hole position, the more likely it is for the rule context coming from the corresponding import file to be more relevant (to predict the target hole). We get a ranked list of import files sorted based on increasing order of distance (i.e., number of lines) between the import usage and the hole position. We start by taking all of the rule context from the first file in the ranked list and then keep iterating the ranked list until either the total context length allocated to the rule gets exhausted or we reach the end of the ranked list.

Table 2: **Heuristics for selecting files for a rule context location**

Rule Context Location	File Ranking Heuristic
Current	file with the target hole. Returns a single file.
Parent Class	file that contains the parent class that occurs closest to the target hole. Returns a single file.
Import	files with the corresponding import usage ranked based on the proximity to the hole. Returns a ranked list of files.
Sibling	files with import usage common to the current file and the sibling file, ranked based on the proximity to the hole. The total number of common imports between the current and the sibling file is used as a tie-breaker. Returns a ranked list of files.
Similar Name	files with import usage common to the current file and the similar name file, ranked based on the proximity to the hole. The total number of common imports between the current and the similar name file is used as a tie-breaker. Returns a ranked list of files.
Child Class	files with import usage common to the current file and the child file, ranked based on the proximity to the hole. The total number of common imports between the current and the child class file is used as a tie-breaker. Returns a ranked list of files.
Import of Sibling	import files ranked based on the frequency of usage in all the sibling files. Returns a ranked list of files.
Import of Similar Name	import file ranked on the basis of frequency of usage in all the similar name files. Returns a ranked list of files.
Import of Parent Class	import file ranked on the basis of frequency of usage in all the parent class files. Returns a ranked list of files.
Import of Child Class	import file ranked on the basis of frequency of usage in all the child class files. Returns a ranked list of files.

A.5. Truncation Strategies for Rule Context

If the rule context is greater than the context length allocated to it, i.e., rule context fraction multiplied by total context length, then we need to truncate the rule context. We followed the below two schemes for truncating context:

- **front:** We truncate the context from the front. This is used for all rule context locations except Parent Class and when we take PL from Current.
- **back:** We truncate the context from the back. This is used when the rule context location is Parent Class and when we take rule context types other than PL from Current.

The truncation strategies for each case were selected based on results on a small validation set. For Current context location, except when the rule context type is PL, we always start by taking code of rule context type from after the hole position. This makes sense as the default Codex context will anyways contain code before the hole. Only if this turns out to be blank, we will use the code of context type from before the hole.

A.6. Other Rule Variations

We experimented with other variations that include: (a) appending class names at the beginning of the rule context, (b) using newline or space to join the rule context and the default Codex context, (c) taking all or top-k of the rule context types, (d) ordering of top-k. See Appendix for more details on these. Table 1 lists our proposed rules. In Figure 1, rule $r = 14$ is predicted and corresponds to taking rule context location of Import, Rule Context Type of MNB and Rule Context Fraction of 0.5.

- Context Separator: This defines how we join the rule context string to the default Codex context string. We experimented with space and newline as context separators.
- Rule Context Formatting: We can format the rule context before giving it to the prompt generator. We experimented with the following options:
  1. class_name: append [class name of the file] at the beginning of the rule context taken from each file that is part of the rule context location. For example, if we are taking rule context from two import files $f1$ and $f2$, the rule context will be formatted as: [class name of $f1$] rule context from $f1$ + space + [class name of $f2$] rule context from $f2$. We use this when the rule context types are MN, I, TI, FD and SL.
  2. class_method_name: we apply this only when the rule context type is MNB. We append method names at the beginning of each of the corresponding method bodies. We also append the rule context from a file with the name of the class as described in the previous item.
  3. comment: Adding in the rule context as a comment, i.e., formatting it as: /** rule context */. This wasn't found to be much useful.
  4. none: passing the rule context as it is. We use this when the rule context type is PL.
- Top-k Type: For each of the rule context types, except PL, we experimented with taking the (a) first (b) last and (c) all of the rule context types, i.e., we can take first-10 identifiers. We found 'all' to be the best among all.
- Top-k: We experiment with k values of (a) 10 (b) 20 and (c) all. We found 'all' to work best for all rule context types.

B. Additional Results

B.1. Variation with k

Figure 3 shows the variation of the success rate over the validation data with the value of k. The purpose of this study is to see how does the performance scale with the number of prompts (or rules). One of the potential real-world usage scenario of this study can be when there is a human-in-the-loop confirming that the hole-prediction is good or not, with the assumption that then human is allowed to use multiple prompts for getting prediction for the same target hole. For these experiments, if any of the k selected rules lead to success, success ratio will be set to one. From the figure, we notice that as we increase the value of k, the performance increases gradually at first and then saturates towards the oracle performance (79.05% for val data). This behaviour is observed for both Fixed Rule as well as RLPG. However, we see that for the same value of k, the success rate for RLPG is higher both when taking the repo-wise as well as the hole-wise averages. This indicates that sampling k rules using RLPG is better than taking top-k most frequent rules, emphasizing the contribution of our rule classifier. Moreover, this gap in performance scales with increasing the value of k. This offers an interesting analogy to

some recent works (Li et al., 2022; Chen et al., 2021) that have shown such scaling behaviour but for samples taken from the output of the LLM as opposed to more samples (i.e. prompts) at the input of the LLM.

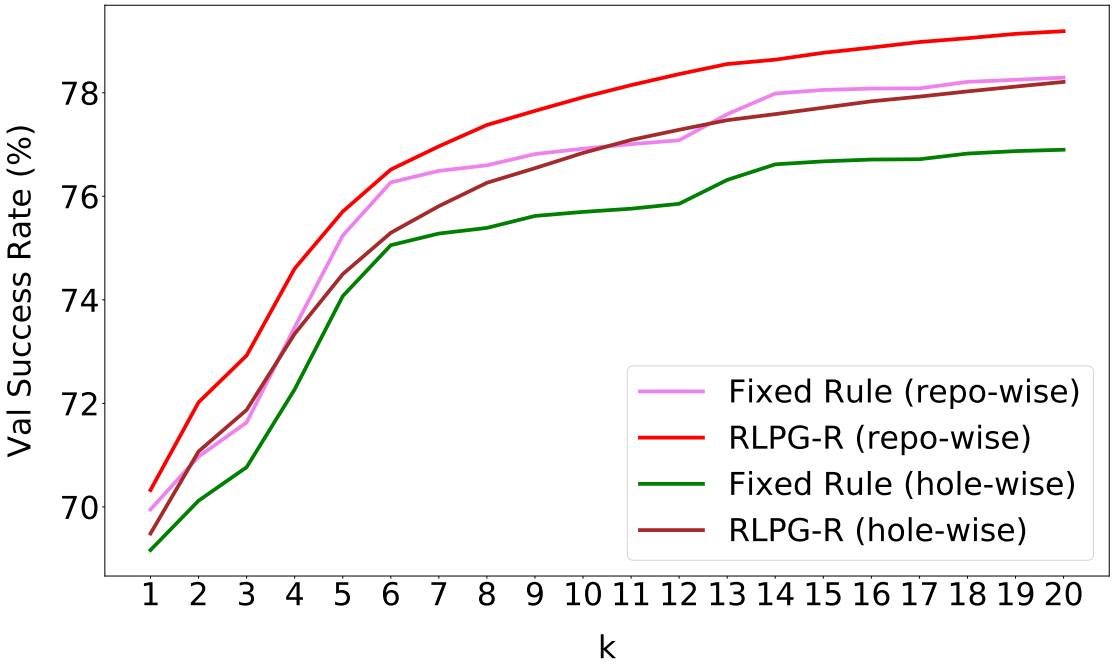

Figure 3: Variation of RLPG and Fixed Rule with k

B.2. Rule-Wise Performance

Figure 4 shows the plot of the top-20 successful rules. The orange bar indicates the fraction of times (in percentage) when a particular rule was applicable and the blue bar indicates the fraction of times when that rule was successful. We can see that the first four rules (i.e. 5, 7, 6, 1) are using context coming from the same file. In particular, using PL can be quite useful. The next line of successful rules correspond to the sibling files. Note that even though these rules have a overall low success rate as compared to the current file rules, it is more likely for them to be successful whenever they are applicable. This is in contrast to $r = 1$ and $r = 62$ (Codex) where the rule being applicable doesn't always lead to success. Note that it is possible to extend our framework by adding more rules and then retraining the rule classifier.

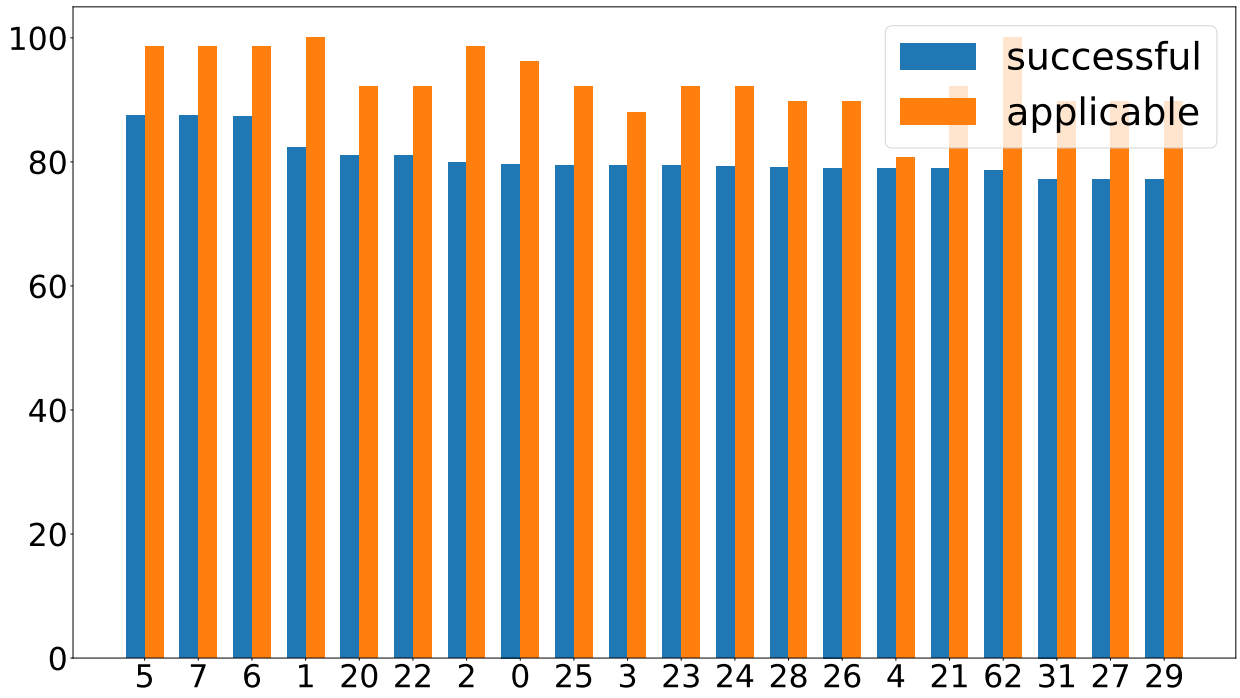

Figure 4: Plot of top-20 successful rules showing the fraction of their success rate vs the fraction of times they were applicable.

B.3. Performance on individual repositories

Table 3: Success Rate of different methods on training data

Repo name	#Total Holes	Oracle	Codex	Fixed Rule (k=1)	RLPG-H (k=1)	RLPG-R (k=1)
largemail	1653	75.38	55.11	62.73	63.94	63.28
ftpserverremoteadmin	7323	86.44	66.11	76.09	76.21	76.76
myt5lib	838	91.65	53.58	61.34	73.51	74.46
seamlets	4890	92.74	62.25	62.72	71.55	74.27
gloodb	10000	91.07	57.50	57.50	70.32	72.31
jjskit	9043	80.36	65.61	72.18	72.00	72.44
mobileexpensetracker	2298	75.94	57.88	67.28	66.84	66.97
gfsfa	10000	80.55	57.33	57.33	59.28	65.24
swe574-group3	2029	76.79	54.46	66.19	65.16	64.91
strudem-sicsa	6131	77.83	64.96	72.55	73.25	73.32
soap-dtc	1370	81.24	64.82	70.73	71.61	72.70
openprocesslogger	7191	81.06	62.19	71.77	72.22	72.62
tapestry-sesame	397	72.54	45.84	61.21	60.71	63.98
exogdx	735	84.76	63.81	75.51	75.92	76.60
designpatternjavapedro	1069	78.30	54.82	64.36	63.99	68.57
quidsee	3020	81.66	60.79	69.50	70.36	70.26
healpix-rangeset	4734	63.54	48.71	54.67	54.94	55.07
sol-agent-platform	10000	73.76	58.22	65.72	65.65	65.94
rsbotownversion	10000	75.23	57.89	65.58	66.22	66.31

Table 4: Success Rate of different methods on validation data

Repo name	#Total Holes	Oracle	Codex	Fixed Rule (k=1)	RLPG-H (k=1)	RLPG-R (k=1)
tyrond	721	83.91	60.33	71.15	71.57	72.68
math-mech-eshop	2225	83.46	62.20	72.76	73.53	73.17
infinispan-storage-service	373	82.31	71.85	78.55	76.94	77.75
teammates-shakthi	7665	82.02	63.74	72.38	72.47	72.46
javasummerframework	10000	79.27	55.92	65.30	65.74	65.55
tinwiki	10000	73.67	69.27	69.27	69.12	69.58
jloogle	3145	84.55	73.16	77.87	77.17	77.36
jcontenedor	5464	81.26	58.99	67.77	67.95	68.32
sohocms	772	76.68	57.90	67.10	67.49	67.62
affinity_propagation_java	1466	79.54	59.14	70.33	70.26	70.26
jata4test	1921	71.06	44.09	54.92	55.91	57.47
swinagile	2595	79.69	63.01	72.29	72.49	72.68
navigablep2p	1322	75.72	59.76	65.43	65.13	65.28
springlime	879	83.50	62.34	74.18	74.86	74.40

Table 5: Success Rate of different methods on test data

Repo name	#Total Holes	Oracle	Codex	Fixed Rule (k=1)	RLPG-H (k=1)	RLPG-R (k=1)
dovetaildb	10000	76.89	57.12	66.45	66.06	66.25
project-pt-diaoc	10000	82.01	52.67	52.81	65.08	61.25
realtimegc	2513	77.64	57.58	67.01	67.85	68.48
fswuniceubtemplates	2070	77.44	55.70	58.89	66.81	65.80
qwikioffice-java	1138	76.45	70.21	70.21	69.86	70.56
glperaudsimon	1766	78.65	53.57	62.51	62.40	61.66
xiaonei-java-api	839	73.42	57.57	62.10	62.69	63.29
ircrpgbot	6591	83.67	69.67	77.24	76.71	76.65
robotsimulator2009w	7514	75.63	56.28	67.55	67.53	67.55
gwt-plugindetect	73	84.93	60.27	68.49	65.75	68.49
apiitfriends	1385	85.05	65.05	74.80	75.67	75.31
wicketbits	754	83.02	59.81	72.94	72.81	73.08
hucourses	590	84.41	70.68	77.46	77.63	77.97
xfuze	3055	84.09	62.82	73.62	72.73	73.62

Table 3, Table 4 and Table 5 present the success rates of different methods over individual repositories in the training, validation and test splits, respectively. The repo-wise averages in Table 2 in the main paper were calculated by taking the average of numbers corresponding to each column. The hole-wise averages correspond to multiplying the repo-wise numbers of each method by the total holes in the repo to get the total number of successful holes by that method for that repo. We then add the total number of successful holes across repos and divide it by the total number of holes in the entire data split to get the hole-wise averages.

B.4. Analysis of Sample Cases

B.4.1. Positive Cases

In Figure 1, RLPG predicts the rule that corresponds to taking method names and bodies from the imported file (i.e. `MaximizingGibbsSampler.java`). Note that `mg.` before the hole position indicates that a method used in the imported file is likely to be invoked. In this case, the rule context (highlighted in violet) contains the method name `InitializeToAssignment` (part of target hole). This in conjunction with the default Codex context which contains the method `CurrentAssignments()` (part of target hole) leads to generation of a successful prompt. On the other hand,

the prompt created from the default Codex context fails to predict the target hole in this case. We provide some examples of cases where RLPG led to the correct prediction and Codex failed.

1. Cases where part of the target hole is found exactly in the rule context.
   - RLPG = `Propagation(int numVars)` vs Codex = `Propagation()`
   - RLPG = `tersFromFile(String filename) {` vs Codex = `ters(String filename) {`
   - RLPG = `als("dampingFactor")) {` vs Codex = `als("numVars")) {`
   - RLPG = `] + ", value = " + entries[1]);` vs Codex = `]);`
   - RLPG = `stem.exit(1);` vs Codex = `stem.err.println("DPAP load error: " + ex.get`
2. Cases where Codex takes strong hint from the preceding natural language comment, e.g. naming the method based on the comment. thereby producing incorrect predictions.
   - RLPG = `d PassMessages()` vs Codex = `d DoOneRoundOfMessagePassing()`
   - RLPG = `teger> CurrentExemplars() {` vs Codex = `teger> ChooseExemplars() {`
   - RLPG = `ring FileName() {` vs Codex = `ring GetAlgorithmFilename() {`

B.4.2. Negative Cases

In certain cases, extra information from rule-context might lead to confusion and produce incorrect predictions.

- RLPG = `an hasConverged_;` vs Codex = `an converged_;`
- RLPG = `_[i][j] = -Double.MAX_VALUE;` vs Codex = `_[i][j] = 0;`

C. Details of Dataset Creation

C.1. Creation of Hole Completion Data

To mitigate the effects caused by potential memorization of the code present in the dataset used for training Codex, we tried to use repositories that Codex might not have been trained on and avoided code repositories from GitHub (Chen et al., 2021)). Instead, we scraped Google Code [4] for repositories tagged with the language "Java". Then we deduplicated repositories by searching for a matching repository with the same name on GitHub. For those repositories with zero matching names on GitHub, we downloaded the archive and extracted the source code (preserving the directory structure). Next, we tried to determine the licenses of all repositories by either looking for a LICENSE file or matching with keywords "license", "copyright", "mit", etc. For repos for which our process was able to come up with a known license, we selected the ones having a permissive license, i.e., MIT, ApacheV2 and BSD. Since code duplication has been shown to have adverse effects (Allamanis, 2018), we removed files that are exact duplicates of each other within a repo. One of the reasons we found this inter-repository duplication may be because sometimes developers adopt lousy practises where instead of declaring a package and importing functions, they simply copy-paste the desired file in the current folder. The target holes coming from any of the duplicate files do not form part of the hole completion dataset. However, these files might be used to contribute to rule context for completing a target hole in a non-duplicate file (e.g. in a sibling file). For the remaining files, we took each line that is not a blanked line or a comment, and chose the middle character as the hole position, i.e., all the characters from the middle of the line to the end of the line form target hole. To avoid large repos having strong bias on our rule classifier, we capped the contribution from each repo to be a maximum of 10000 holes. If the number of holes in the repo exceeds 10000, we randomly select 10000 holes. Please see Table 6 for statistics of our dataset. The #Holes represent the holes after deduplication and capping.

C.2. Creation of Data for Repo-Level Rules

We used the tree-sitter API for Java [5] to get the parse-tree of an individual file in a repo. To get information at a repo-level, for each file in the repo, we stored the following information:

[4] https://code.google.com/archive/
[5] https://github.com/tree-sitter/tree-sitter-java

Table 6: Corpus Statistics

Feature	Train	Val	Test	Total
# Repositories	19	14	14	47
# Files	2655	1060	1308	4757
# Holes	92721	48548	48288	189557

1. list of all class names in the file. This helped us to get the parent or child class file corresponding to a given parent or child class.
2. the file corresponding to each import statement.
3. for each import statement in the file, the position in the file where the import is used. This is used for ranking the files based on the heuristics mentioned in Table 2.
4. list of sibling files
5. list of similar name files. This was done by splitting the filenames based on either camel-case or underscore. If the sub-parts of two files match, then they are said to have similar name.

The above meta-data was calculated only once for each repo. The subsequent hole completions can use the same cached information. In practise, we can use a hash to store and retrieve this info efficiently. For a rule, given the rule context location, we first obtain a single file or ranked list of files (see Table 2) using the info in the parse tree in conjugation with the above repo-level meta-data. All the rule context type information (MN, MNB, SL, I, TI, FD) can then be obtained by querying the parse tree of the selected file.

D. Details of the Rule Classifier

D.1. RLPG-H

Equation 2 shows the prediction of the rule label conditioned on the hole window representation.

$$\hat{y}_r^h = P(y_r^h = 1|H^h) = \text{sigmoid}(W^2(\text{relu}(W^1(F_\phi(H^h)) + b^1)) + b^2) \tag{2}$$

We used Adam (Kingma & Ba, 2015) optimizer with a learning rate of 3e-4 and batch size of 64. We used CodeBERT (Feng et al., 2020) as our pretrained model F_ϕ to obtain the representation of hole window. The size of the representation (corresponding to the hidden dimension of the `[CLS]` token) is 768. $W^1 \in \mathbb{R}^{512\times768}, b^1 = 512, W^2 \in \mathbb{R}^{63\times512}, b^2 = 63$.

D.2. RLPG-R

We have a multiheaded attention mechanism between our keys, values and query as described in Equation 5. For each head, we perform a scaled dot-product attention (Vaswani et al., 2017)(Equation 4).

$$Q^h = F_\phi(H^h), \quad K_r^h = F_\phi(C_r^h), \quad V_r^h = F_\phi(C_r^h) \tag{3}$$

$$Att(Q^h, K_r^h, V_r^h) = V_r^h \text{softmax}\Big(\frac{{Q^h}^\top K_r^h}{\sqrt{d_k}}\Big) \tag{4}$$

$$MultiHead(Q^h, K_r^h, V_r^h) = W^O \text{concat}(head_i, head_2, \ldots head_\tau) \tag{5}$$

$$\text{where} \quad head_i = Att(W_i^Q Q^h, W_i^K K_r^h, W_i^V V_r^h)$$

$$\hat{y}_r^h = P(y_r^h = 1|H^h, C_r^h) = \text{sigmoid}\Big(W_r G(MultiHead(Q^h, K_r^h, V_r^h)) + b_r\Big) \tag{6}$$

In the equations above, d_k is the dimension of the key, W_i^Q, W_i^K, W_i^V are the query, key and value projection matrices, τ is the number of heads and W^O is the linear projection that combines the heads. The output from Equation 5 is fed to a positionwise fully-connected feedforward network (Vaswani et al., 2017). We employ a residual connection (He et al., 2016) followed by layer normalization (Ba et al., 2016) before and after the feedforward network. In the above equation these transformations are represented by the module G. The resulting output is then linearly projected and sigmoid is applied to get the prediction of the rule.

We used Adam (Kingma & Ba, 2015) optimizer with a learning rate of 3e-4 and batch size of 64. We used CodeBERT (Feng et al., 2020) as our pretrained model F_ϕ to obtain the representation of hole window and rule context. The size of the representation (corresponding to the hidden dimension of the `[CLS]` token) is 768. In equations 1, 2 and 3 in Section 3.2, the projection matrices $W_i^Q \in \mathbb{R}^{d_q \times d_{model}}$, $W_i^K \in \mathbb{R}^{d_k \times d_{model}}$, $W_i^V \in \mathbb{R}^{d_v \times d_{model}}$, $W^O \in \mathbb{R}^{d_{model} \times \tau d_v}$. For the multihead attention, we used $d_k = d_q = d_v = 32$, $\tau = 4$ and $d_{model} = 768$, $W_r \in \mathbb{R}^{63 \times 768}$ and $b_r = 63$. G module consists of a dropout layer, a residual connection, a layernorm, followed by a positionwise-feedforward network, i.e., A sequence of (a) dense layer of weights=2048×768, bias=768, (b) relu activation, (c) dense layer of weights=768×2048, bias=2048, (d) dropout layer, (e) residual connection, (f) layernorm. A dropout value of 0.25 was used while training.

E. Experimental Details on Prompt Generation

We used the OpenAI Codex Completions API for generating the predicted hole from the Codex model. In particular, we used the code-davinci-001 engine with temperature set to 1.0 and stop criteria as newline. The completion length was kept to be 24 and the maximum prompt length was 4072. Tokenization was done using the suggested tokenizer [6]. To allow for fast computation, we used simple models like CodeBERT (Feng et al., 2020) and GraphCodeBERT (Guo et al., 2020) as our pretrained models. One of the limitations of these pretrained models is that the maximum context length that can be taken as input by these models is much smaller than the maximum context length allowed by Codex. Therefore, we need to truncate the rule context which means that some important parts of the rule context might be omitted while getting the rule context representation. Using pretrained models that allow larger context length or models that augment the context (Wu et al., 2022) offer avenues for future work.

F. Computational Complexity of RLPG

To collect the ground-truth data for training our rule classifier we queried the Codex API for each applicable rule per hole (maximum rate limit of 400 holes per minute). The computational complexity of training our larger RLPG-R variant (parameters=3.6M, data=141269 holes and compute=9.19 min per epoch on a single Tesla V100 GPU) is much smaller than finetuning all or some part of Codex, thereby ensuring low latency when deployed in an IDE. During inference, we need to calculate the repo-level rules statistics just once and all the subsequent hole completions in the repo can utilize this cached information. So, there is no additional computational complexity in creating the prompt during inference.

G. Related Work

LLMs for Code: Recently, there has been a lot of work around large language models of code. One class of models are the decoder-only models that correspond to generating code from left-to-right. Codex (Chen et al., 2021) is an example of a decoder-only model. Other models include Google's model (Austin et al., 2021), GPT-J-6B (Wang & Komatsuzaki, 2021), GPT-Neo (Black et al., 2021b), GPT-Neo-X (Black et al., 2021a), CodeParrot (Tunstall et al., 2022) and PolyCoder (Xu et al., 2022a). We also have some encoder-only models that use a masked language modelling objective. CodeBERT (Feng et al., 2020), GraphcodeBERT (Guo et al., 2020) and CuBERT (Kanade et al., 2020) are example of such models. Lastly, we have the class of encoder-decoder models that generally use a bidirectional encoding of a context to decode a series of masked tokens. Code-T5 (Yue Wang, 2021), AlphaCode (Li et al., 2022) and recently released InCoder (Fried et al., 2022) are examples of such models. We refer the reader to Xu et al. (2022a) for details about the size of each model and the data that was used for training these models.

Repo-Level Info: Few works use information from outside the current file. Zhang et al. (2021) uses the parent class to generate the comments for the child class. Xu et al. (2022b) incorporate three types of structural locality features while training the kNN-LM (Khandelwal et al., 2020). These features are binary variables that correspond to the presence or absence of similar hierarchy. The three levels of hierarchy are (a) sibling file, (b) file in the same repo (c) no hierarchy. In contrast we have a much richer set of rules incorporating the semantics and structure of the repository. Also, we assume black-box access to the actual LM and restrict ourselves to generating prompt(input) for the LLM.

Prompt Generation: There have been promising works around prompt generation techniques in the area of natural language processing. Broadly, there are two categories of automatic prompt generation techniques. The first category corresponds to producing continuous/soft prompts where the prompt is described in the latent space of a language model (Li & Liang,

[6] https://huggingface.co/docs/transformers/model_doc/gpt2#transformers.GPT2TokenizerFast

2021; Qin & Eisner, 2021; Bragg et al., 2021; Lester et al., 2021; Liu et al., 2021b). For example, Prefix-Tuning (Li & Liang, 2021) adds a prefix to the LM that can be learned by finetuning on examples from the downstream task. The second category produces discrete prompts where the prompt is a text string that can be interpreted by a human (Shin et al., 2020; Gao et al., 2021; Schick & Schütze, 2021). For example, Autoprompt (Shin et al., 2020) generates prompt using a fixed template consisting of trigger tokens. The trigger tokens are shared across all inputs and determined by a gradient-guided search involving the LM. Our work falls in the category of discrete prompt generation techniques as we produce a prompt consisting of code tokens that can be easily interpreted by a human. However, in contrast to prior works that use a set of fixed templates for all examples, we learn to generate prompts conditioned on each example. Another important distinction is that we do not require access to the weights of the LM. To the best of our knowledge, our work is the first to explore automatic prompt generation in the domain of source code.