An Empirical Analysis of InCoder on the Statement Prediction Task

Frank van der Heijden
Supervisors: Maliheh Izadi, Arie van Deursen
EEMCS, Delft University of Technology, The Netherlands

June 19, 2022

A Dissertation Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

s

An Empirical Analysis of InCoder on the Statement Prediction Task

Frank van der Heijden
Delft Univeristy of Technology
Delft, The Netherlands

***Abstract*—Automatic code completions are a widely used feature when programming code efficiently. These completions can be made by various code language models, and these can be differentiated in three categories: single token completion, statement (line) completion and block completions. These completions, and in particular statement predictions are usually created using only the left context, missing key information and context on the other side. InCoder, a novel state of the art model is capable of using both contexts. In this study we aim to show the impact of using both contexts in statement completions. The results show that on average, an improvement of 9.9% exact match and similar results for Edit Similarity, BLEU-4, ROUGE-L F1, and METEOR when using both contexts instead of only the left context.**

I. Introduction

Most code language models that are capable of statement prediction are unable to work with both the left and right context of the tokens they are predicting. Therefore, because they are only working with one context in a document, key information might be missing while predicting tokens. *InCoder* [1] is a generative model which is capable of infilling code in between the left and right context, combining strengths of both causal language models and masked language models. In the InCoder paper it is mentioned that the combined strengths are an improvement, however, the exact measurements of how well it performs better are missing. In this study, an empirical analysis is performed measuring the performance of the InCoder model on infilling statements in Python and JavaScript code, using a novel dataset: P1K-22 and JS1K-22, scraped from the top 1000 most starred Python and JavaScript projects on GitHub. From this dataset, statements will be masked and are predicted in two settings: using only the left context, and using both the left and right contexts. Results are assessed using the metrics Exact Match, Edit Similarity, smoothed 4-gram BLEU, ROUGE-L F1, and METEOR. On average, an improvement of 9.9% on the exact match was seen when using both contexts instead of only the left context. The code used in this study is publicly available on GitHub[1].

In the following section, a small background is introduced on code language models and how they operate. The related works section touches upon related papers in the field of Transformers, code completion improvements and problems in the field of code completion. After, the methodology for creating the dataset and configuring the InCoder-1B model is described. In the Experiment Design, the research questions are listed and their assessment is explained. After the research questions, the results are listed, and analysed in the discussion section. Finally, a conclusion is drawn and a possibility for future work is suggested.

[1]https://github.com/FrankHeijden/incoder-analysis

II. Background

Generative language models predict tokens based on a tokenized input. These tokens are character sequences, which a model has been trained on, and is considered the vocabulary of the model. Code (as text) is transformed into this vocabulary specific to that model, which can be fed into the model. For causal language models, new tokens are generated repeatedly until a stopping condition has been met. Masking is where a single token is masked in a list of tokens, and the model will predict this single token. InCoder combines masking and the causal language model by predicting infills, which can consist of multiple tokens. After predicting, the list of tokens can be converted back into regular text, containing the input code with the prediction.

III. Related Works

In this section we provide an overview of related work in the field of statement completion. We categorize these studies in three groups. In the first paragraph, since InCoder is based on the Transformer model, the main improvement of a transformer based model approach is described. In the second paragraph, it is described how code completion models have been improved in previous works. Finally, problems in the field of code completion are discussed.

A. Code generation using a Transformer

Code completion is one of the most widely used features in a modern integrated development environment (IDE), which can be achieved using a Transformer language model. IntelliCode Compose [2], a code completion framework, is capable of generating code sequences such as local variables, methods, APIs, arguments, and more. In the paper it is mentioned that previous code completion tools fail to take into account the surrounding context of the code. This means that the frameworks fail to create the correct code suggestions the user actually need, resulting in a low acceptance rate. IntelliCode Compose introduces a Generative Pretrained Transformer (GPT), based on the transformer network architecture [3], and applies a form of source code understanding: natural language understanding with the help of lexemes, an abstract syntax tree (AST), a concrete syntax tree (CST), and a dataflow graph. The best model yielded an average edit similarity of 86.7%, and a perplexity of 1.82 for the Python programming language. Codex [4], another language model, based on the GPT-3 was also able to achieve state-of-the-art performance. Codex generates python code from a docstring, and is used in GitHub Copilot. Results show that Codex is able to solve easy programming interview questions. Trained on 54 million lines of code, totalling a dataset of 159GB of code, it is able to solve 28.8% of the problems presented. By sampling multiple times from the same model, the solvability increased to 70.2%.

B. Improving code completions

1) Using dataflow graphs: A dataflow graph represents a dependency relation between variables. The nodes in this graph represent a variable, and edges represent where the value of each variable comes from [5]. In order to generate a dataflow graph, first an AST is created which can then be transformed into a data flow graph. GraphCodeBERT is a pre-trained model that has three main input components: in addition to the bimodal source code and comment data, it also has the data flow graph as input. The resulting model GraphCodeBERT ended up achieving state-of-the-art performance in comparison to other models such as CodeBERT [6] and RoBERTa [7], performing better on the tasks of *natural language code search*, *code clone detection*, *code translation*, and *code refinement*.

2) Using trees: Trees can also be used to represent code structure. TreeGen [8] solves the long dependency problem by introducing a novel AST reader (the encoder in the transformer model), achieving state-of-the-art performance. The AST reader combines grammar rules with the AST structure. The model attempt to predict grammar rules, where programs are decomposed to their grammar rules, and can be parsed as an AST.

C. Code completion problems

1) Code completion poisoning: Automatic code completion models are trained on large code datasets from a large number of code repositories, but this may lead to problems. Usually, this data is then split into two categories: unimodel and bimodal [6], code that only contains the source, and code that is also paired with a natural language representation, respectively. However, this code may not contain the best solution to a problem, or perhaps even an insecure solution to a problem [9]. Auto completion models could perhaps suggest outdated code practises, such as recommending an older SSL/TLS protocol version, like SSLv3. Even without automatic code completion, developers tend to also pick outdated/insecure code practises, based on surface features around a StackOverFlow question [10].

2) Vocabulary issues: Large vocabularies introduced by source code affect the performance of Natural Language Models (NLMs) [11]. In the paper, an open vocabulary NLM is presented, which is able to scale to such large vocabularies (but not limited to large corpora), and achieving state-of-the-art performance, outperforming n-gram LMs. The open vocabulary NLM also has a small hardware footprint, it can be executed on a consumer-grade GPU and predicting tokens takes a fraction of a second, fast enough to use in modern IDEs. Furthermore, the presented NLM also outperforms on the task of bug detection, where LMs detect defects as "unnatural" code.

3) Insufficient quality of code completions: While code completion models may produce very accurate results on theoretical evaluations, they may perform bad in a real world scenario. NL2Code [12] is presented, a plugin for the PyCharm IDE, evaluating the results in such a setting. Results show that the plugin had little impact on the developer, and developers would still search for code externally, rather than accepting the code provided by the IDE plugin. Furthermore, the study showed that the snippets produced by the plugin were of insufficient quality, even though the accuracy scores obtained during automatic evaluations were state-of-the-art.

The efforts of improving code completions using different techniques are great, and using the right context in addition to the left context is one of them. This study will examine the performance gained by using this additional context, using a novel dataset for Python and JavaScript.

TABLE I
TOP-1000 MOST-STARRED PYTHON AND JAVASCRIPT FILE STATISTICS

Files	N
All	327 972
−*Duplicates*	286 705

.

IV. METHODOLOGY

Our approach consists of two steps, data collection and the empirical assessment of the pretrained InCoder-1B model. For the former, we have collected and processed 2000 open-source GitHub repositories. For the latter, we used existing state-of-the-art metrics to compute accurate and reproducible results.

The P1K-22 and JS1K-22 datasets are created by collecting the top-1000 GitHub repositories for Python and JavaScript code, respectively. GitHub provides a Search API[2], from which the most starred repositories can be fetched. This resulted in a combined dataset of 327 972 individual files. To prevent oversampling the files were filtered for duplicates, and the results of the filtering can be found in table I. From these files, multiple samples can be made where different statements can be predicted. For each file, 10 samples are made for different statement completions, on different lines in the code. Each of these samples will be selected on a random token within the line, and every token to the left of this selected token is the left context, and everything starting from the next newline is the right context. The text in between the end of the left context, and start of the right context, is the ground truth.

By default, the stopping condition for these models is based on a maximum number of tokens they are able to

[2]https://docs.github.com/en/rest/search#search-repositories

TABLE II
STOP TOKENS FOR THE STATEMENT PREDICTION TASK

Token ID	Text Representation
205	\n
284	\n\n
353	\r\n
536	\n\n\n
994	\r\n\r\n
3 276	\n\n\n\n
4 746	\r\n\r\n\r\n
15 471	\n\n\n\n\n
16 027	\n\n\n\n\n\n\n\n
28 602	\r\n\r\n\r\n\r\n
40 289	\n\n\n\n\n\n
43 275	\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n
50 517	<|endofmask|>

.

predict. InCoder has a token window of 2048 tokens, which means that the input plus the to be generated tokens need to fit in this window. On the dataset collected, the average token length was 10 per line with empty lines excluded. In order to have some buffer, an input length of 2000 tokens was chosen, giving InCoder the chance to generate up to 48 tokens for each sample. InCoder has an end-of-mask token, indicating the end of the model's statement generation. Since token prediction is expensive and relatively slow, an early stopping condition was made to stop as soon as the end-of-mask token is predicted. In addition, generation is stopped when a newline has been predicted by the model. The InCoder token vocabulary contains in total 12 tokens with newlines (and these happen to also consist of only newlines and carriage returns, but more within one token), and table II lists the token ids and their text representation.

To produce the predictions of the InCoder model, the Delft High Performance Computing Centre (DHPC) [13] was used. In total, 4 tasks were ran in parallel on the DHPC, computing the predictions for the P1K-22 dataset, P1K-22 without comments, the JS1K-22 dataset, and JS1K-22 without comments. For each task, 4 parallel jobs were scheduled, each using 4 NVIDIA v100 GPUs.

Language Model Metrics

The metrics chosen are state-of-the-art for language models, and each of them give a score based on a different method.

1) BLEU-4: BLEU [14] compares an n-gram of the prediction against an n-gram of the ground truth. BLEU-4 computes the weighted uniform sum of the 1, 2, 3 and 4-grams precision. However, the n-gram precision is modified, and capped to the maximum number of matches in the ground truth. This prevents models which are generating a single correct token over and over from receiving a perfect score. To make BLEU-4 better on the sentence-level [15], the smoothing function proposed in (Lin and Och, 2004) was chosen, adding 1 to the total n-gram count for n ranging from 2 to N.

2) ROUGE-L F1: ROUGE-L [16] applies a longest common substring (LCS) method. The longer the LCS is between the prediction and the ground truth, the more similar they are. Based on the LCS accuracy, a F1-score can be computed. A benefit over BLEU is that ROUGE-L does not require any specific n-gram, because the longest common substring is already the longest n-gram.

3) Exact Match: The exact match is relatively simple, it returns 1 if the prediction exactly matches with the ground truth, and 0 if there is no exact match.

4) Edit Similarity (Levenshtein Distance): The Levenshtein distance [17] is a metric for computing the amount of changes one has to make to convert one string into another, limited by three character operations: insertions, deletions and substitutions. When normalised, this Levenshtein distance becomes the edit similarity.

5) METEOR: METEOR [18] creates alignments between the prediction and ground truth, where an alignment maps each unigram to at most 1 unigram in the other. Multiple alignments can be created, and the one with the least crossing mappings is chosen. This is done several stages, and after the final stage a precision and recall can be computed. Using the precision, recall, and a penalty function, the METEOR score can be computed.

The metrics were computed on DHPC, using 36 CPU cores. An overview of the whole workflow can be found in Figure 1.

V. Experiment Design

The main research question is "What is the performance of the state-of-the-art model *InCoder* of JavaScript and Python statement completions, when evaluated on only the left context versus both contexts". To answer this question, a novel dataset has been introduced in the last section, allowing a large corpus of data to be used for inferencing. Since the dataset creates a test-pair (only the left context and both contexts) for each test case, the results should be fair to compare against each other. Python and JavaScript were chosen since these were the main languages InCoder was trained on.

The main question can be divided in the following research questions:

RQ1*: What is the performance in terms of GPU inference time, BLEU-4, ROUGE-L, Exact Match, Levenshtein distance and METEOR for JavaScript and Python statement completions for statements with both contexts?*

InCoder has the ability to use both contexts in the prompt, and computing the metrics with both left and right contexts given to the model would be the baseline, and an accurate real-world scenario, where code exists both to the left of the prediction and further down a file.

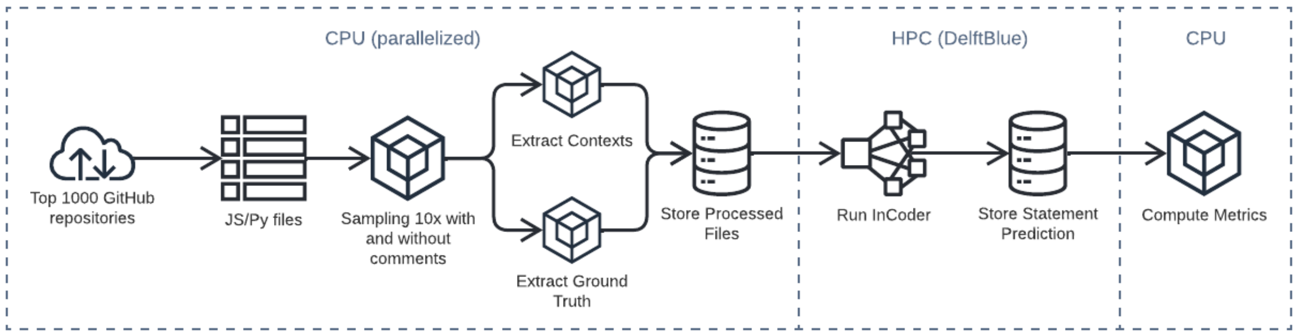

Fig. 1. Overview of the workflow of this study.

RQ2: *What is the difference in terms of GPU inference time, BLEU-4, ROUGE-L, Exact Match, Levenshtein distance and METEOR for JavaScript and Python statement completions for statements using only the left context?*

Both contexts is not required for the infill task, and it is also possible to generate tokens based on only the left context. To generate infills using only the left context, the right context can be set as an empty string. This subquestion analyses the results in case only the left context is given to the model.

RQ3: *What is the difference in terms of GPU inference time, BLEU-4, ROUGE-L, Exact Match, Levenshtein distance and METEOR for JavaScript and Python statement completions for statements on Trigger Points?*

Since the dataset is split on a random token within a line, the last few tokens of the leftcontext could compose a Trigger Point (TP), a so-called point defined to trigger autocompletions. These positions are character sequences after which an IDE would suggest an autocompletion. The TPs chosen here are composed of some of the keywords and operators from the Python and JavaScript language. Some keywords were removed, as predicting after such keyword will always result to the same output, e.g. for JavaScript in the case of the "**do**" keyword, which is always followed by an opening bracket, "{".

For the Python language, the following keywords and operators were chosen:

await, **assert**, **raise**, **del**, **lambda**, **yield**, **return**, **while**, **for**, **if**, **elif**, **else**, **global**, **in**, **and**, **not**, **or**, **is**, **with**, **except**, **.**, **+**, **-**, *, **/**, **%**, **, <<, >>, **&**, |, ˆ, **==**, **!=**, <=, >=, **+=**, **-=**, **=**, <, >, **;**, **,**, **[**, **(**, {, **~**

For the JavaScript language the following were chosen:

await, **assert**, **throw**, **delete**, **const**, **var**, **let**, **yield**, **return**, **while**, **for**, **if**, **else**, **new**, **with**, **.**, **+**, **-**, *, **/**, **%**, <<, >>, **&**, |, ˆ, **==**, **!=**, <=, >=, **+=**, **-=**, **=**, <, >, **;**, **,**, **[**, **(**, {, **~**

RQ4: *Case study: how well does InCoder perform with both contexts when used in an IDE plugin for the Python and JavaScript language?*

This question can be answered by using the data collected by the *Code4Me IDE Plugin*[3], which collects the prediction and ground truth in the same fashion as this automatic evaluation study. With this data, the same metrics can be computed, and shown how well the InCoder model performed in a real-world setting, with new unseen data. Code4Me automatically collects the prediction and ground truth from the user anonymously, and whether the completion has been accepted by the user. In order to retrieve the ground truth, Code4Me tracks the changes the user made over a period of 30s after the prediction has been suggested or accepted.

VI. Results

The dataset statistics can be found in table III, as well as the same dataset with only the Trigger Points. Interestingly, the input context length is on average much lower than the maximum token window of the InCoder model.

The results of running the P1K-22, and JS1K-22 dataset can be found in table IV, and also the same dataset with only the Trigger Points filtered out. On average, an improvement of 9.9% on the exact match was seen when using both contexts instead of only the left context, and similar large improvements using the other metrics. The most staggering improvement was seen on the JS1K-22 TP only dataset, an improvement of 12.47% on exact match. On the same note, Python code with the comments stripped from the source code seem to perform slightly better than the raw Python code containing comments. Removing comments from the code seem to also have a benefit on the average token prediction time. This can also be seen in the JS1K-22 TP only dataset, with an average token length of 321 for the left context, and 527 for both contexts: 8ms less per token on average.

Finally, table V shows the results of the Code4Me plugin over a period of 25 days. Most notably, when a user writing Python code has accepted a suggested statement completion, the scores are higher than in the automatic evaluation setting using the P1K-22 dataset.

[3]https://github.com/code4me-me/code4me

TABLE III
STATISTICS ON THE DATASET USED FOR THE INCODER EVALUATION

Dataset	N	Context	ITL	T/line	C/T
P1K-22	1 040 938	LC	532	8.05	5.05
		BC	800	8.04	5.04
- w/o comments	1 017 212	LC	503	7.53	4.93
		BC	767	7.53	4.97
P1K-22 (TP only)	63 373	LC	590	7.89	5.13
		BC	850	7.90	5.12
- w/o comments	63 394	LC	552	7.32	5.01
		BC	806	7.34	5.07
JS1K-22	1 368 282	LC	431	9.28	4.17
		BC	620	8.31	4.19
- w/o comments	1 330 895	LC	382	9.00	3.89
		BC	552	8.03	3.99
JS1K-22 (TP only)	162 730	LC	347	7.50	4.55
		BC	566	7.19	4.42
- w/o comments	165 168	LC	294	7.09	4.20
		BC	487	6.87	4.18

Statistics of the novel dataset, showing the total number of test cases (N), the input token length (ITL), tokens per line (T/line) and characters per token (C/T) for the left context only (LC) and both contexts (BC).

VII. DISCUSSION

The JS1K-22 dataset contains around 327k more test cases for the both left context only and both contexts, this can be explained due to the repository size differences in the scraped repositories from the top-1000 github repositories. However, since the size of the dataset is quite large, and the source code is different for the JavaScript and Python dataset, it can be neglected. The input is considerably lower than the maximum token window from InCoder, and combined with the knowledge that adding the right context to the left context improves the scores significantly, this could mean that adding even more context from multiple documents could greatly improve the score. However, the model needs to be adapted to be able to work with multiple contexts from across different documents.

On average, a large improvement was seen by using both contexts. However, even with both contexts, the actual input size seems to be around 1265 tokens shorter than the maximum context window of InCoder. This finding suggests that modifying the model to allow for more (extra) contexts could result in even better results for the InCoder model.

Furthermore, the Code4Me IDE plugin shows that whenever a user accepts a suggested completion, the scores are better than in the automatic evaluation. When a suggestion has not been accepted, still relatively good scores are seen. These scores suggest that InCoder was exactly right 21.94% of the time for the Python user base and 12.43% for JavaScript. However, a worthy mention is that the study performed using the Code4Me IDE plugin was mainly focused on recruiting a Python user base, and therefore the numbers for JavaScript are significantly lower.

Threats to validity

Internal threats to validity are factors which unintentionally affect the results of the study. In this study, the variables remained constant throughout the process, and the datasets were treated both in the same way. The InCoder tokenizer was used for both datasets, and the metrics were configured and calculated in the same manner.

An external threat present in the project could be that there might exist an overlap of the code present in the dataset and the code used for training with the InCoder model. However, it is unlikely that InCoder has seen the exact same input data, since the token infills chosen within each file were chosen at random in the P1K-22 and JS1K-22 datasets.

Threats to construct validity are factors which affect the validity of the measurements performed. In this work, well established metrics have been used in this particular field of research [14], [16]–[18]. These metrics were configured the same for all computed values throughout this study, and can be found on the GitHub repository[4].

VIII. CONCLUSION AND FUTURE WORK

In conclusion, using both contexts produce significantly better results in all cases: on the raw dataset, without comments, and only on trigger points. Furthermore, stripping comments from the source code result in better scores for the P1K-22 dataset. In the Code4Me plugin, when a prediction has been accepted by the user, the scores are higher than in the automatic evaluation setting.

For future research, InCoder could be improved by allowing more contexts to be present in the input context window. Another improvement can be made by optimising the tokenizer used for calculating the metrics, which can be improved by combining knowledge of the source code language used.

IX. RESPONSIBLE RESEARCH

The results in this research should be fairly easy to reproduce, all code performing the steps in the methodology and experiment design has been published on GitHub[4]. The repositories used in the dataset can be downloaded using the *incoder-analysis-java* module. Finalizing the dataset, running InCoder and computing the metrics can be done using the *incoder-analysis-python* module.

Furthermore, the full configuration and exactly which implementation for the metrics was used can also be found in the public repository.

X. ACKNOWLEDGEMENTS

I would like to thank Maliheh Izadi for supervising the process of my study and providing helpful feedback throughout the challenges solved in this project. In addition, I would like to thank Georgios Gousios for providing feedback and for providing the Larisa server which were used for testing and creating the files necessary for the P1K-22 and JS1K-22 dataset. Lastly I want to thank Marc Otten for providing the key insights of the Code4Me IDE Plugin, and to thank Tim van Dam, Jorit de Weerdt and Mika Turk for providing feedback on the project.

[4]https://github.com/FrankHeijden/incoder-analysis

TABLE IV
METRICS OF THE INCODER MODEL ON THE STATEMENT PREDICTION TASK

Dataset	Context	IT	PL	EM	Edit Sim	BLEU-4	ROUGE-L	METEOR
P1K-22	LC	28.92	5.94	41.01	70.82	45.13	56.94	49.47
	BC	30.71	**6.02**	51.23	78.09	54.01	66.12	58.92
- w/o comments	LC	**28.11**	5.83	40.83	71.00	45.31	57.00	49.72
	BC	29.84	5.90	**52.04**	**78.99**	**54.97**	**67.03**	**59.99**
P1K-22 (TP only)	LC	26.93	5.59	42.09	72.06	46.83	63.79	51.68
	BC	28.83	**5.64**	50.89	78.15	54.46	71.13	59.50
- w/o comments	LC	**26.27**	5.56	41.63	72.17	46.78	63.99	51.73
	BC	28.30	5.62	**51.06**	**78.69**	**54.95**	**71.75**	**60.13**
JS1K-22	LC	28.96	8.65	43.83	70.84	46.39	51.83	51.03
	BC	29.82	8.82	**52.00**	76.48	**53.47**	**59.32**	58.60
- w/o comments	LC	**28.28**	8.62	41.15	69.99	44.96	49.37	49.81
	BC	28.94	**8.77**	50.79	**76.63**	53.34	58.30	**58.80**
JS1K-22 (TP only)	LC	20.29	6.35	39.44	70.43	44.76	59.02	49.75
	BC	21.35	6.39	**51.91**	**80.02**	**56.45**	**72.13**	**62.40**
- w/o comments	LC	**19.50**	6.43	33.79	66.49	41.25	54.46	46.32
	BC	20.39	**6.44**	48.09	77.87	55.10	69.95	61.35

Results of computing the various metrics after running the InCoder-1B model. The contexts are split up among the left context only (LC), and both the left and right context (BC). All metrics are an average over multiple predictions: inference time (IT) as ms·token^{-1}, prediction length (PL) as the amount of tokens generated, exact match (EM), Edit Similarity, BLEU-4, ROUGE-L and METEOR.

TABLE V
METRICS OF THE CODE4ME IDE PLUGIN USING INCODER

Language	N	IT	PL	EM	Edit Sim	BLEU-4	ROUGE-L	METEOR
Python	4166	10.38	**5.60**	21.94	52.72	26.93	42.85	30.11
- chosen only	663	**9.87**	4.79	**62.14**	**83.03**	**59.90**	**80.14**	**65.99**
JavaScript	724	8.47	**6.53**	12.43	50.57	24.38	33.21	27.77
- chosen only	95	**6.70**	4.95	**42.11**	**79.89**	**54.89**	**74.78**	**60.56**

Results of the Code4Me IDE Plugin using the InCoder-1B model, with both contexts. These metrics are computed in the same manner as Table IV

REFERENCES

[1] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," 2022.

[2] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," *arXiv preprint arXiv:2005.08025*, 2020.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), vol. 30, Curran Associates, Inc., 2017.

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

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

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

[7] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[8] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang, "Treegen: A tree-based transformer architecture for code generation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 8984–8991, 2020.

[9] R. Schuster, C. Song, E. Tromer, and V. Shmatikov, "You autocomplete me: Poisoning vulnerabilities in neural code completion," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 1559–1575, 2021.

[10] D. Van Der Linden, E. Williams, J. Hallett, and A. Rashid, "The impact of surface features on choice of (in) secure answers by stackoverflow readers," *IEEE Transactions on Software Engineering*, no. 01, pp. 1–1, 2020.

[11] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code!= big vocabulary: Open-vocabulary models for source code," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 1073–1085, IEEE, 2020.

[12] F. F. Xu, B. Vasilescu, and G. Neubig, "In-ide code generation from natural language: Promise and challenges," *arXiv preprint arXiv:2101.11149*, 2021.

[13] Delft High Performance Computing Centre (DHPC), "Delftblue supercomputer (phase 1)." https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase1, 2022.

[14] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, (USA), p. 311318, Association for Computational Linguistics, 2002.

[15] B. Chen and C. Cherry, "A systematic comparison of smoothing techniques for sentence-level BLEU," in *Proceedings of the Ninth Workshop on Statistical Machine Translation*, (Baltimore, Maryland, USA), pp. 362–367, Association for Computational Linguistics, June 2014.

[16] C.-Y. Lin and F. J. Och, "Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics,"

in *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*, (Barcelona, Spain), pp. 605–612, July 2004.

[17] P. E. Black, “Algorithms and theory of computation handbook.” https://www.nist.gov/dads/HTML/Levenshtein.html, 1999.

[18] S. Banerjee and A. Lavie, “METEOR: An automatic metric for MT evaluation with improved correlation with human judgments,” in *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, (Ann Arbor, Michigan), pp. 65–72, Association for Computational Linguistics, June 2005.