Execution-based Evaluation for Data Science Code Generation Models

Junjie Huang[1]*, **Chenglong Wang**[3], **Jipeng Zhang**[5]*, **Cong Yan**[3], **Haotian Cui**[6]*,
Jeevana Priya Inala[3], **Colin Clement**[4], **Nan Duan**[2], **Jianfeng Gao**[3]
[1] Beihang University [2] Microsoft Research Asia [3]Microsoft Research Redmond
[4] Microsoft [5] Hong Kong University of Science and Technology [6] Toronto University
[1]huangjunjie@buaa.edu.cn,
[2,3,4]{chenwang,coyan,janala,coclemen,nanduan,jfgao}@microsoft.com
[5]jzhanggr@connect.ust.hk, [6]ht.cui@mail.utoronto.ca

Abstract

Code generation models can benefit data scientists' productivity by automatically generating code from context and text descriptions. An important measure of the modeling progress is whether a model can generate code that can correctly execute to solve the task. However, due to the lack of an evaluation dataset that directly supports execution-based model evaluation, existing work relies on code surface form similarity metrics (e.g., BLEU, CodeBLEU) for model selection, which can be inaccurate.

To remedy this, we introduce ExeDS, an evaluation dataset for execution evaluation for data science code generation tasks. ExeDS contains a set of 534 problems from Jupyter Notebooks, each consisting of code context, task description, reference program, and the desired execution output. With ExeDS, we evaluate the execution performance of five state-of-the-art code generation models that have achieved high surface-form evaluation scores. Our experiments show that models with high surface-form scores do not necessarily perform well on execution metrics, and execution-based metrics can better capture model code generation errors. [1]

1 Introduction

Code generation models (Chen et al., 2021a; Tunstall et al., 2022) have shown promising results to improve developer productivity by generating code from natural specifications (Le et al., 2020; Al-Hossami and Shaikh, 2022). These promising results also bring interest to code generation for data scientists, who program data analysis scripts in interactive notebook environments like Jupyter Notebooks (Kluyver et al., 2016) where programs are written interactively in loosely organized program cells (Figure 1 (1)). This domain and style

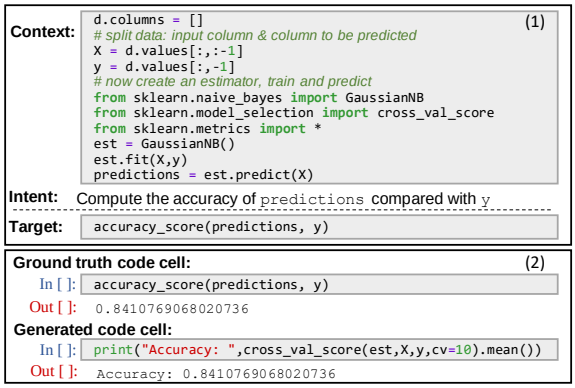

Figure 1: An example from ExeDS. The first block describes the task of data science code generation with code context and NL intents. The second block compares the code and output of reference and generation.

differences motivates new modeling resources, e.g., new datasets (e.g. JuiCe (Agashe et al., 2019)) and models (e.g. JuPyT5 (Chandel et al., 2022)) specific to data science tasks.

However, we still lack a good methodology to evaluate data science (DS) code generation models. JuiCe dataset uses the BLEU (Papineni et al., 2002) and Exact Match (EM), the prevailing metrics in code generation, to measure semantic similarity between the generated and reference code. However, these two surface-form metrics have limitations: the former neglects code syntactic features and the latter is too strict (Ren et al., 2020). Execution-based metrics are another widely accepted line of metrics in general software engineering (SE) domain, where the correctness of generated functions is determined by whether the outputs are consistent with oracle input-output data/unit tests. For DS problems, however, collecting an executable dataset and performing execution-based evaluation are challenging. DS notebooks usually do not come with their own set of unit tests and existing datasets like JuiCe do not track the input data (such as tables) needed to run the notebooks. In addition, the outputs from notebook cells are often not "pure"

*Work done during internship at Microsoft Research Asia.
[1]Source code and data can be found at https://github.com/Jun-jie-Huang/ExeDS.

Proceedings of the Fourth Workshop on Data Science with Human-in-the-Loop (Language Advances), pages 28 - 36

values (e.g., numbers, strings, or lists) like the outputs of the functions in SE problems. The DS notebook cell outputs are meant for human understanding and hence, may contain complex data structures (e.g., data frames, plots) accompanied with texts; thus simply checking whether outputs are the same is too strict to capture cases when the generated cell output is semantically correct but formatted differently from the reference (Figure 1-(2)).

In this paper, we provide a dataset for evaluating DS code generation models, dubbed ExeDS, which contains 534 data science problems built on JuiCe (Agashe et al., 2019). We collect ExeDS by first crawling data dependencies from original GitHub repositories for the notebooks and filtering out notebooks with runtime errors; then, we curated 534 high-quality problems with sufficient code context and human-written natural language (NL) to describe tasks as the testset. With ExeDS, we can evaluate execution correctness by comparing outputs from generated code with desirable outputs.

We experiment with 5 existing code generation models on ExeDS to identify their execution performance. Experiment results show that (1) models with high/low surface-form scores do not necessarily generate execution-correct code – for example, while Codex (Chen et al., 2021a) is low in BLEU, it achieves high execution scores. (2) Execution-based metrics can better capture code errors which might be helpful for model improvements.

2 Related Works

Data science (DS) refers to the practice of analyzing data and acquiring insights with computational methods (Donoho, 2017). With the goal to improve productivity, there are increasing interests in building systems to solve a variety of DS tasks, including code synthesis (Agashe et al., 2019), code synthesis for visualization (Chen et al., 2021b) and data preparation (Yan and He, 2020), documentation (Liu et al., 2021; Wang et al., 2021), etc. In our work, we focus on code generation in DS, which generates code with code, NL and data context.

Code generation benchmarks are predominantly evaluated by matching code surface forms(Papineni et al., 2002; Lin, 2004; Ren et al., 2020). These datasets evaluate explicit code generation with different input specifications, including natural language (Wang et al., 2015; Oda et al., 2015; Zhong et al., 2017; Yin et al., 2018; Yu et al., 2018; Lin et al., 2018), unfinished code (Iyer et al., 2018; Lu et al., 2021), and input-output examples (Polosukhin and Skidanov, 2018; Zavershynskyi et al., 2018). However, surface form metrics are unable to assess code as programmers, who focus on the functionality and execution correctness in practice.

Dataset	#	Domain	Evaluation
APPS (Hendrycks et al., 2021)	10,000	SE	Unit Test
MBPP (Austin et al., 2021)	974	SE	Unit Test
HumanEval (Chen et al., 2021a)	164	SE	Unit Test
DSP (Chandel et al., 2022)	1,139	DS	Unit Test
JuiCe (Agashe et al., 2019)	1,981	DS	Surface Form
PlotCoder (Chen et al., 2021b)	894	DS	Surface Form
ExeDS (ours)	534	DS	Output Match

Table 1: Comparisons of code generation testsets.

Consequently, recent works turn to execution-based metrics instead, where the code would be correct if it passes a set of unit tests defined by humans (Roziere et al., 2020; Kulal et al., 2019; Austin et al., 2021; Chen et al., 2021a; Hendrycks et al., 2021). However, the complex output data and scarcity of units tests in DS limit its application in DS code generation. Chandel et al. (2022) explore applying unit tests in DS, but they only focus on educational problems. Table 1 compares ExeDS with various related datasets.

3 ExeDS for Execution Evaluation

As mentioned in Section 1, the lack of executable environments for notebooks is a key limiting factor of execution-based model evaluation for data science tasks. Thus we first construct an evaluation dataset ExeDS and analyze its characteristics. Then describe the methods for execution evaluation.

Dataset Collection ExeDS contains 534 problems with code context, NL task description, reference code and target execution output, which is built upon JuiCe (Agashe et al., 2019) with 659K publicly available Python Jupyter notebooks from GitHub. We create ExeDS in the following steps.

Step1: Crawling Data Context and Execution. Programming problems in DS often deal with data, which are often stored in files (e.g., `.csv`) and loaded by code. Executing notebooks needs such data dependencies, which are not provided in JuiCe. Thus, we first crawl dependent data for notebooks from their GitHub repositories. Notebooks with inaccessible data or using libraries not present in Python standard library and default DS environment are removed. With data dependency, we execute notebooks with a time limit of 1000 sec-

Function Type	%	Examples
Data statistic	40	Avg., var., p-value, ...
Explore data value	19	Min/max value, ...
Explore data property	10	Dtype, shape, ...
Machine learning	16	Loss, train, predict, ...
Simple math	6	Arithmetic, ...
Data changing	5	Sort, sample, ...
Data displaying	4	Head/tail columns, ...

Table 2: Function types of target code in ExeDS.

onds per cell. After execution, code cells have three types of outputs: (1) *displaying data* with a figure; (2) *execute result* with a textual execution output; and (3) *stream output* with a printed textual output through streams. Since it's hard to compute figure similarities, in this paper, we only evaluate execution correctness on textual outputs and construct ExeDS with *execute result* and *stream output*.

Step 2: Dataset Filtering and Intent Curation. As some cells are overly complex for code generation, for simplicity, we remove examples with more than 5 lines or using customized methods in target code cells. To keep diversity, we downsample cells with frequent outputs, e.g. `df.summary()`, `df.info()`, `df.shape`, etc. To ensure sufficient context is provided, we remove the target code whose variables are absent in the previous 5 cells.

Since some cells lack sufficient descriptions for the problems, for clarity, we recruit two university students with Python and notebook experience to manually write NL descriptions for each example. After viewing the context, target code and output, they are asked to write descriptions containing information in two aspects: (1) the functions of target code; (2) the instructions to print outputs. We discard examples that annotators feel hard to describe.

Finally, we obtain 534 problems from 278 notebooks for ExeDS, each with code context, NL description, target code, and desired execution output.

Dataset Statistics Table 2 shows the function types in ExeDS. We found the majority of target codes are computing statistics (40%), exploring data value (19%) or property (10%), and for machine learning (16%), which are popular DS tasks.

Table 4 presents the types of execution output in all 534 problems. We find the majority of execution output are numbers, which is not surprising considering the fraction of data statistics and exploring data value in code functions. Also comparing numbers is less complicated than comparing other types of data like strings or data frames, which helps easier evaluation of execution outputs.

Library	# problems
pandas	534
numpy	473
matplotlib	431
sklearn	287
seaborn	211
scipy	135
statsmodels	57
math	46
datetime	42
re	39

Table 3: Frequency of most common 10 libraries used in 534 examples of ExeDS.

Table 3 displays the most common libraries used in ExeDS. We find the majority of them use data science libraries and all of them use `pandas`, which indicates our focus on data science code generation.

Evaluation Metrics In ExeDS, we measure the execution correctness by comparing the reference outputs with outputs from generated code, which is called output exact match (OutputEM). However, as a variety of examples produce outputs in numbers, we convert all numbers in `string` type to the `float` type with two decimal spaces to better match numbers. Similarly, we remove the explanation string when printing outputs for better comparison.

4 Evaluating Code Generation on ExeDS

Based on ExeDS, we evaluate the models' performance on data science code generation and compare both surface-form code and execution output.

DS Code Generation We investigate the task of target code cell generation in notebooks with context. Figure 1 presents an example of the task. For each target code cell, we prepare a source-target example, conditioned on prior multimodal context and natural language intent. The context includes: (1) the closest three cells prior to the target cell, regardless of code or markdown; (2) a code state-

Output Type	%	Examples
Single number	55	0.841076906802073; 68
List/tuple/array	34	(256, 10); ['UserID', 'Gender']
Dataframe	11	Weight 26.25 Speed 36.70 dtype: float64

Table 4: Types of ground truth outputs in ExeDS.

	BLEU	CodeBLEU	EM	OutputEM
GPT-style framework				
GPT-neo-125M	3.4	17.2	0.0	1.5
GPT-neo-1.3B	9.2	26.2	0.0	10.7
GPT-neo-2.7B	9.1	28.8	0.4	13.3
CodeGPT	26.4	28.6	1.5	12.7
CodeGPT-adapted	25.1	26.8	3.3	13.1
Codex*	3.9	23.5	0.0	27.7
encoder-decoder framework				
PyMT5	25.7	35.8	2.8	19.7
JuPyT5	**35.3**	**41.1**	**6.2**	**31.6**

Table 5: Evaluation results of surface form metrics and execution metric. * denotes a zero-shot setting.

ment to define the columns names of data in the format of `df.columns=['a', 'b']`.

Baseline Models We test five code generation models: (1) PyMT5 (Clement et al., 2020) is an encode-decoder transformer (Vaswani et al., 2017) pretrained on Python corpus. (2) JuPyT5 (Chandel et al., 2022) is an encoder-decoder transformer pretrained on Jupyter notebooks with the code-infilling objective. (3) CodeGPT and CodeGPT-adapted (Lu et al., 2021) are two GPT-style models (Solaiman et al., 2019) pretrained on CodeSearchNet Python functions (Husain et al., 2019), where the former is trained from scratch and the latter is trained from GPT-2 checkpoint. (4) GPT-neo (Black et al., 2021) is a GPT-style model pretrained on The Pile (Gao et al., 2021), a dataset with a variety of text sources including 8% GitHub code. We evaluate three GPT-neo models with different parameters, including 125M, 1.3B, and 2.7B. (5) Codex (Chen et al., 2021a) is the state-of-the-art model trained on 159G GitHub Python files from GPT-3 (Brown et al., 2020). We test its zero-shot performance due to the inaccessibility of model weights.

Finetuning For training and validation, we filter a set of 123K source-target examples from JuiCe with data dependencies, where the target is any code cell and the source is the prior multimodal context as in ExeDS. We randomly select 4K examples for validation and leave the rest for finetuning. More details can be found in Appendix A.

Metrics We report results with OutputEM, which is the proportion of examples with correct output, and surface-form metrics, i.e. BLEU, CodeBLEU (Ren et al., 2020), and Exact Match (EM).

Error Category	%	Exception Examples
Use undefined variable	45	NameError...
Use undefined API	16	AttributeError...
Use wrong schema	22	KeyError, ValueError, IndexError...
Wrong Syntax	8	IndentationError, SyntaxError ...
Other errors	9	No message, ImportError, ...

Table 6: Qualitative error analysis on examples that raise exceptions during execution. Some representative exception types for each error category are listed.

5 Evaluation Results

In this section, we show and analyze evaluation results to show the advantages of our ExeDS dataset.

5.1 Main Results

Table 5 shows the results of different baseline models in surface form metrics and execution correctness. We have the following main observations.

(1) For all models, the surface form EM is close to zero while the OutputEM is in a normal range. This suggests that surface form EM often fails to evaluate code correctness, while the execution metric is better which covers more correct cases and shows correctness beyond matching code strings.

(2) Surprisingly, zero-shot Codex achieves compatible results with finetuned JuPyT5 in OutputEM, but it performs badly with surface-form metrics. This finding suggests the strength of Codex to generate correct code and understand the multimodal context. In addition, the difference between surface-form scores and OutputEM again shows the superiority of measuring code with execution correctness.

(3) Encoder-decoder models perform better than GPT-style models with all metrics, which indicates their strength in generating code. Also, JuPyT5 achieves the best performance with all metrics. One possible reason is that JuPyT5 is pretrained on a large corpus of notebooks, which learns the necessary knowledge from the notebook context.

5.2 Error Analysis

We give two error analyses of execution results to investigate examples with raised execution exceptions and erroneous outputs. The code examples are produced by our top-performing model JuPyT5. Detailed examples can be found in Appendix B.

Exception Types Table 6 shows five exception types from 154 examples. We find for 45% cases, the model fails to capture data-flow and uses undefined variables in context. For 16% cases, the

Error Category	%	Examples
Incorrect Code	56	Figure 3 & 4
No Output	8	Figure 5
Partially Correct	12	Figure 6
To Many Output	24	Figure 7 & 8

Table 7: Analysis of 50 examples with wrong outputs.

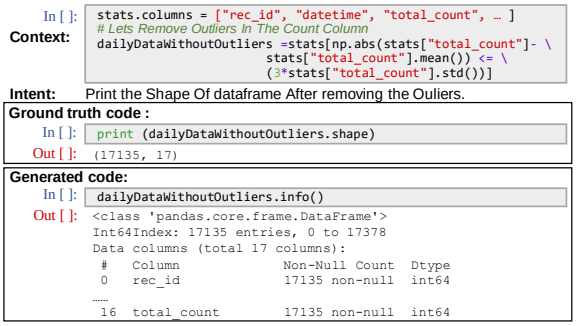

Figure 2: An incorrect example with high surface form metrics scores but low execution metrics scores. Surface form metrics are deficient to evaluate code correctness.

model misuses API methods and often leads to `AttributeError` , possibly due to version differences and calling methods without import. 22% cases misuse the data schema of dataframes, which indicates the need to improve code generation models with such multimodal context, especially how to incorporate the data schema context. Only 8% cases have syntax problems, suggesting the model's strong ability to generate syntax-correct code.

Output Errors Table 7 shows four types of output errors from 50 examples. We find 56% cases have incorrect code. The challenging NL description and context might be hard for models to understand and generate correct code. 8% cases complete the correct functions but do not call `print()` to output. 12% of cases are partially correct, where the output mismatch is caused due to some missing details, for example, the absence of some parameters. Finally, 24% cases produce too many outputs.

6 Case Study

We give an example predicted by JuPyT5 with a high BLEU score but erroneous outputs in Figure 2, to show the advantages of execution evaluation for DS code generation. The example is a typical DS task which intends to explore the shape of a dataframe. But the model misunderstands the intents and generates code to display all dataframe information. Although we can find the expected shapes from the output, i.e., 17135 entries and 17 columns, the output is not exactly correct. However, as the code is short while the variable name is long, which leads to a high overlap between prediction and ground truth, the generated code obtains above average BLEU and CodeBLEU scores. This example reveals the deficiency of surface form metrics to evaluate code correctness.

7 Conclusion

In this paper, we propose an evaluation dataset to support execution correctness evaluation for data science code generation dubbed ExeDS, which consists of 534 typical data science problems from Jupyter Notebooks, each with code context, task description, target code, and desired execution output. By performing experiments with five strong code generation models on ExeDS, we find models that achieve high surface-form scores do not necessarily produce execution correct code, and execution-based metrics could capture more detailed code generation errors. We expect our efforts to attract more attention to code execution correctness and generating executable code.

Limitations

Firstly, only the test set examples have high quality of human annotation and verification. Thus the training set might be too noising to train a robust code generation model. Secondly, the execution metric is insufficient to show other information like semantic relatedness, variable naming, and API usages, which are also important in evaluating a good code. Thirdly, our datasets and metrics focus on Python code in data science domain. It's unclear whether is applicable to general software code. Fourth, our execution-based automatic evaluation is more time-consuming to compute and evaluate than other surface-form metrics like EM, BLEU. At last, evaluating generated code is far different from evaluating natural languages. The final goal of code generation is to generate execution and functional correct code. Though with many limitation, our work could be a pilot study which provides insights and possible solutions on how to better evaluate code generation models.

References

Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. 2019. JuICe: A large scale distantly supervised

dataset for open domain context-based code generation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5436–5446, Hong Kong, China. Association for Computational Linguistics.

Erfan Al-Hossami and Samira Shaikh. 2022. A survey on artificial intelligence for source code: A dialogue systems perspective. *ArXiv*, abs/2202.04847.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *ArXiv*, abs/2108.07732.

Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Shubham Chandel, Colin B. Clement, Guillermo Serrato, and Neel Sundaresan. 2022. Training and evaluating a jupyter notebook data science assistant. *ArXiv*, abs/2201.12901.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021a. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374.

Xinyun Chen, Linyuan Gong, Alvin Cheung, and Dawn Song. 2021b. PlotCoder: Hierarchical decoding for synthesizing visualization code in programmatic context. 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)*, pages 2169–2181, Online. Association for Computational Linguistics.

Colin Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. 2020. PyMT5: multi-mode translation of natural language and python code with transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9052–9065, Online. Association for Computational Linguistics.

David L. Donoho. 2017. 50 years of data science. *Journal of Computational and Graphical Statistics*, 26:745 – 766.

Leo Gao, Stella Rose Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2021. The pile: An 800gb dataset of diverse text for language modeling. *ArXiv*, abs/2101.00027.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *NeurIPS*.

Hamel Husain, Hongqi Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *ArXiv*, abs/1909.09436.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium. Association for Computational Linguistics.

Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E. Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B. Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter Development Team. 2016. Jupyter notebooks - a publishing format for reproducible computational workflows. In *ELPUB*.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alexander Aiken, and Percy Liang. 2019. Spoc: Search-based pseudocode to code. In *NeurIPS*.

Triet Huynh Minh Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep learning for source code modeling and generation. *ACM Computing Surveys (CSUR)*, 53:1 – 38.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain. Association for Computational Linguistics.

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. 2018. NL2Bash: A corpus and semantic parser for natural language interface to the linux operating system. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan. European Language Resources Association (ELRA).

Xuye Liu, Dakuo Wang, April Wang, Yufang Hou, and Lingfei Wu. 2021. HAConvGNN: Hierarchical attention based convolutional graph neural network for code documentation generation in Jupyter notebooks. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 4473–4485, Punta Cana, Dominican Republic. Association for Computational Linguistics.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation (t). *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *ACL*.

Illia Polosukhin and Alexander Skidanov. 2018. Neural program search: Solving data processing tasks from description and examples. In *ICLR 2018*.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, M. Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *ArXiv*, abs/2009.10297.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems*, volume 33, pages 20601–20611. Curran Associates, Inc.

Irene Solaiman, Miles Brundage, Jack Clark, Amanda Askell, Ariel Herbert-Voss, Jeff Wu, Alec Radford, and Jasmine Wang. 2019. Release strategies and the social impacts of language models. *ArXiv*, abs/1908.09203.

Lewis Tunstall, Leandro von Werra, and Thomas Wolf. 2022. *Natural Language Processing with Transformers: Building Language Applications with Hugging Face*. O'Reilly Media, Incorporated.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

April Yi Wang, Dakuo Wang, Xuye Liu, and Lingfei Wu. 2021. Graph-augmented code summarization in computational notebooks. In *IJCAI*.

Yushi Wang, Jonathan Berant, and Percy Liang. 2015. Building a semantic parser overnight. In *ACL*.

Cong Yan and Yeye He. 2020. Auto-suggest: Learning-to-recommend data preparation steps using data science notebooks. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 476–486.

Tao Yu, Rui Zhang, Kai-Chou Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Z Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *EMNLP*.

Maksym Zavershynskyi, Alexander Skidanov, and Illia Polosukhin. 2018. Naps: Natural program synthesis dataset. *ArXiv*, abs/1807.03168.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *ArXiv*, abs/1709.00103.

A Finetuning Details

We finetune all the baseline models, except Codex, on our cleaned training set and select the best checkpoint with the perplexity score on dev set for testing. All models are trained on 16 Tesla V100 32GB GPUs. The hyper parameter are presented in Table 8.

At inference time, we use beam search decoding with a beam size of 5.

B More Examples

In this section, we present 6 examples to show the typical types of errors with erroneous outputs in Figure 3 - Figure 8. We also give an example with a typical type of errors causing exceptions in Figure 9.

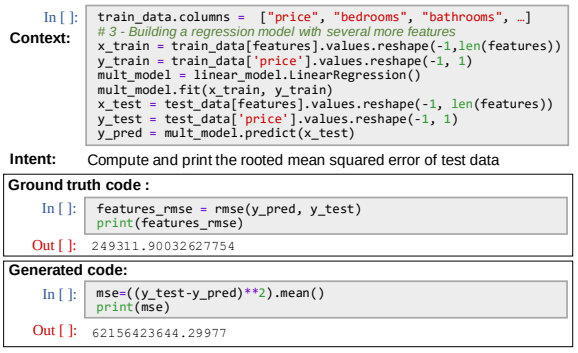

Figure 3: An example with incorrect code. The NL intent is too challenging and the generated code misses the key information to compute the (rooted) error. More powerful models to understand NL intent are required.

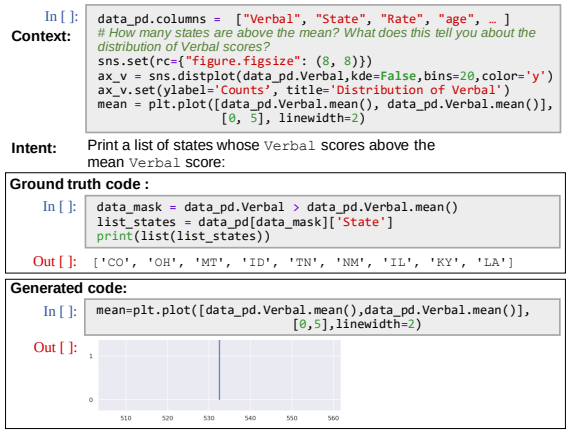

Figure 4: An example with incorrect code. The model fails to perform contextual reasoning over such multi-modal context.

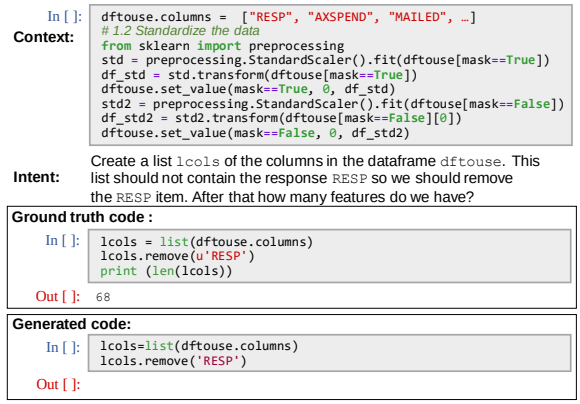

Figure 5: An example with a no output error. the generated code satisfies the intent to create a list and remove the item. But it fails to produce the output, i.e., the length of the feature list.

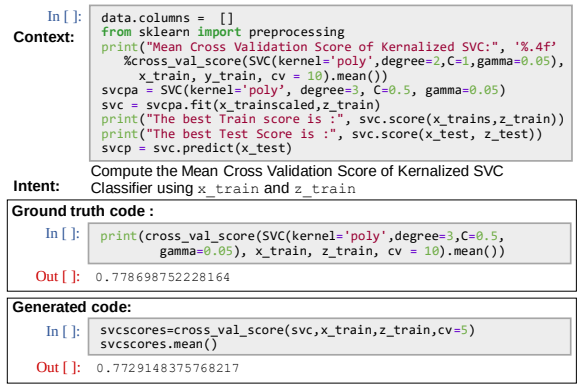

Figure 6: An example with a partially correct error. The code is actually correct but the parameter for cv is different, resulting in the difference between the ground truth and execution outputs.

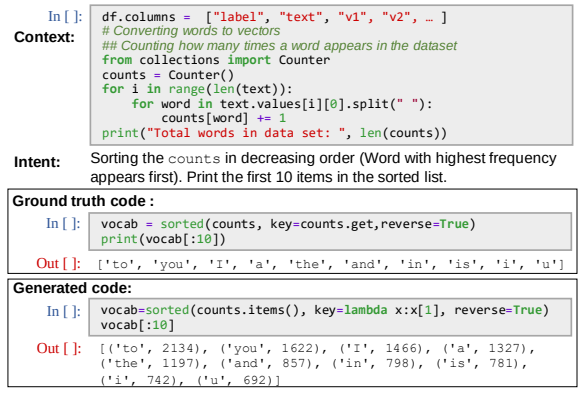

Figure 7: An example with too many output. The correct output actually exists in the execution output, but the excessive output causes the inexact match and decline in ExeF1.

Hyperparameter	CodeGPT	CodeGPT-adapted	GPT-neo 125M	GPT-neo 1.3B	GPT-neo 2.7B	PyMT5	JuPyT5
# vocab size	50001	50260	50257	50257	50257	50337	50340
# parameters	124M	124M	125M	1.3B	2.7B	374M	374M
# hidden size	768	768	768	2048	2560	1472	1472
# layers	12	12	12	16	20	12	12
# heads	12	12	12	24	32	12	12
dropout	0.1	0.1	0.1	0.1	0.1	0.1	0.1
optimizer	AdamW	AdamW	AdamW	AdamW	AdamW	Adam	Adam
learning rate	5e-05	5e-05	5e-05	5e-05	5e-05	1e-4	1e-4
batch size	16	16	3	1	1	1	1
epochs	30	30	10	10	10	10	10
max tokens	512	512	2048	2048	1536	3600	3600

Table 8: Details of the hyperparameters used during fine-tuning for the code generation task in this paper.

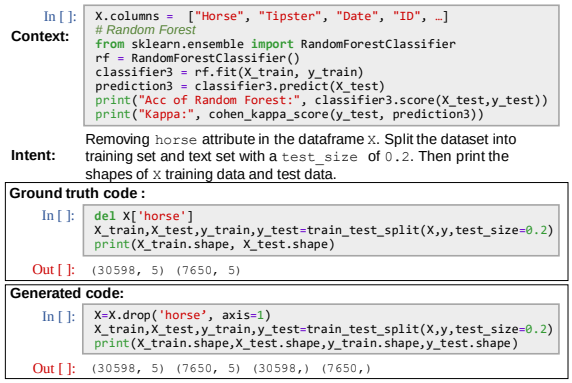

Figure 8: Another example with too many output.

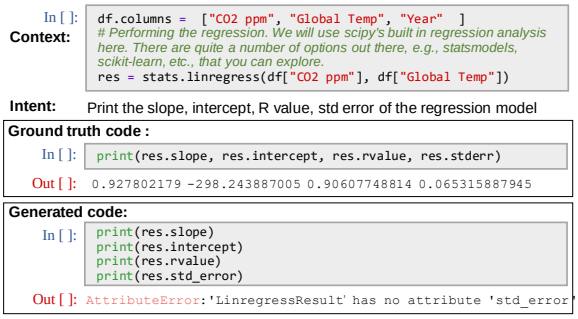

Figure 9: An example running with exceptions. The model misuses the attribute to call the standard errors.