CodeInsight: A Curated Dataset of Practical Coding Solutions from Stack Overflow

Anonymous ACL submission

Abstract

001 We introduce a novel dataset tailored for code generation, aimed at aiding developers in common tasks. Our dataset provides examples that include a clarified intent, code snippets associated, and an average of three related unit tests. It encompasses a range of libraries such as `Pandas`, `Numpy`, and `Regex`, along with more than 70 standard libraries in Python code derived from Stack Overflow. Comprising 3,402 crafted examples by Python experts, our dataset is designed for both model finetuning and standalone evaluation. To complete unit tests evaluation, we categorize examples in order to get more fine grained analysis, enhancing the understanding of models' strengths and weaknesses in specific coding tasks. The examples have been refined to reduce data contamination, a process confirmed by the performance of three leading models: Mistral 7B, CodeLLAMA 13B, and Starcoder 15B. We further investigate data-contamination testing GPT-4 performance on a part of our dataset. The benchmark can be accessed at `anonymized address`.

1 Introduction

In the dynamic landscape of software engineering, developers frequently confront the challenge of translating conceptual ideas into functional code. While navigating this process, the gap between intention and implementation can often be a hurdle, even for experienced programmers. Traditionally, developers have turned to online resources like Stack Overflow, searching for solutions in natural language to address their specific coding dilemmas.

The emergence of large language models (LLMs) trained on code has heralded a new era in this domain. Innovations like Codex (Chen et al., 2021a) have revolutionized the field by providing real-time code suggestions in Integrated Development Environments (IDEs). Similarly, models such as ChatGPT and CodeLLAMA (Rozière et al., 2023) demonstrate the potential for integrating into IDEs, offering developers context-aware assistance in initiating and refining code, thereby enhancing the efficiency of the software development cycle.

However, the ascent of code generation through LLMs underscores the heightened need for datasets that emphasize precision, context-awareness, and syntactic accuracy. While existing datasets have propelled advancements in this arena, they are not without limitations. The shift towards LLM-focused datasets has led to a decreased emphasis on traditional training sets, directing attention towards evaluation sets. This shift challenges the training of models from scratch or for specific task fine-tuning. Moreover, while datasets like HumanEval (Chen et al., 2021b) or APPS (Hendrycks et al., 2021) provide valuable insights, they often fall short of mirroring the real-world coding challenges developers encounter.

Addressing these gaps, this paper introduces the CodeInsight dataset, a resource specifically tailored for Python code generation. This focus is anchored in Python's widespread adoption in key sectors like data science, machine learning, and web development. The dataset, comprising 3,402 unique, expert-curated Python examples, spans basic programming to complex data science challenges, complete with unit tests for evaluation. The CodeInsight dataset stands out in its ability to provide a nuanced balance between breadth and depth, offering a finely-tuned resource for training and evaluating LLMs in Python code generation. By bridging the gap between natural language and code, CodeInsight presents a tool for understanding and enhancing the capabilities of LLMs in real-world programming contexts.

The dataset provides three primary innovations, uniquely combined within this resource:

- It includes a unit test based evaluation, offering a more robust evaluation metric than traditional methods such as BLEU score.

Findings of the Association for Computational Linguistics ACL 2024, pages 5935–5947
August 11-16, 2024 ©2024 Association for Computational Linguistics

- Examples are annotated to facilitate a deeper analysis of its strengths and weaknesses.
- It provides a training set in addition to a test set, with each example being manually curated to ensure high quality, supporting efficient fine-tuning.

Organized as follows, this paper first details the dataset construction process in Section 2, including our sources, selection criteria, and annotation methods. Section 3 presents astatistical analysis of the dataset, highlighting its diverse applications. In Section 4, the dataset's is tested through evaluations using LLM baselines. Lastly, Section 5 situates CodeInsight within the broader landscape of code generation datasets.

2 Dataset Construction

Our pipeline for building CodeInsight consists of three pivotal steps. Initially, we identified the sources to retrieve examples. Subsequently, from these sources, we filtered the most relevant natural language-code pairs. The final phase involved annotating these pairs and crafting associated unit tests. This section provides a breakdown of each of these stages.

2.1 Data Sources

To develop a dataset for code generation aimed at aiding development, we prioritize sources that closely mirror real-world development challenges, ensuring a match between natural language and code. We chose Stack Overflow due to its extensive collection of real-world programming questions and solutions, featuring balanced complexity and contributed by a broad and experienced community.

Despite Stack Overflow's extensive collection of developer queries, only 36% of Python-tagged questions fit the *how-to* format essential for our dataset, as identified in Yin et al. (2018). A *'how-to'* question typically presents a clear, task-oriented query where the developer seeks a method to accomplish a specific programming task.

To address the challenge of identifying relevant examples, we utilized the CoNaLa dataset (Yin et al., 2018), a curated collection of Python *'how-to'* examples from Stack Overflow. This dataset features 2,379 examples that have been manually reviewed and corrected by annotators, alongside approximately 600,000 unrefined examples ranked by their likelihood of fitting the 'how-to' criteria. Our selection encompassed the 2,379 hand-written examples and the top-ranked 3,121 unrefined examples.

To broaden the scope and applicability of our dataset, we have incorporated an additional 600 samples from Stack Overflow, emphasizing the use of packages like `Pandas`, `Numpy`, and `Regex`. The integration of these packages is a decision to align the dataset with the emergent code generation demands in data science, both in academic research and industry applications. Moreover, `Regex`'s inclusion enhances the dataset's to accommodate a wider range of specialized problems.

The sourcing procedure began with the elimination of redundancies and the filtration of issues based on a baseline of community engagement—measured by votes and views—and the presence of accepted answers. We then prioritized the problems using a weighted ranking system that accounts for the temporal dimension, recognizing that older issues may naturally garner more attention over time.

Finally, from our selection process, we gathered a total of 7,300 raw examples to serve as the foundation for our dataset.

2.2 Data Filtering

The transition into the data filtering phase necessitates a strategy to select examples from the source, acknowledging that not all contributions from the Stack Overflow community are directly amenable to our goals, as underscored by Yin et al. (2018); Lai et al. (2023). To illustrate, the most upvoted question on pandas is *'How to iterate over rows in a DataFrame in Pandas'*, yet the consensus answer advises against iteration, highlighting the complexity inherent in the selection process.

To navigate these intricacies, we established criteria for inclusion:

Authenticity of Developer Inquiries Only those questions that present realistic programming scenarios are considered, ensuring the dataset's relevance to the actual needs of developers.

Direct Extractability of Code We require that the code snippet can be unambiguously identified and extracted from the accompanying explanatory text.

Natural Language and Code Alignment A robust correspondence between the problem state-

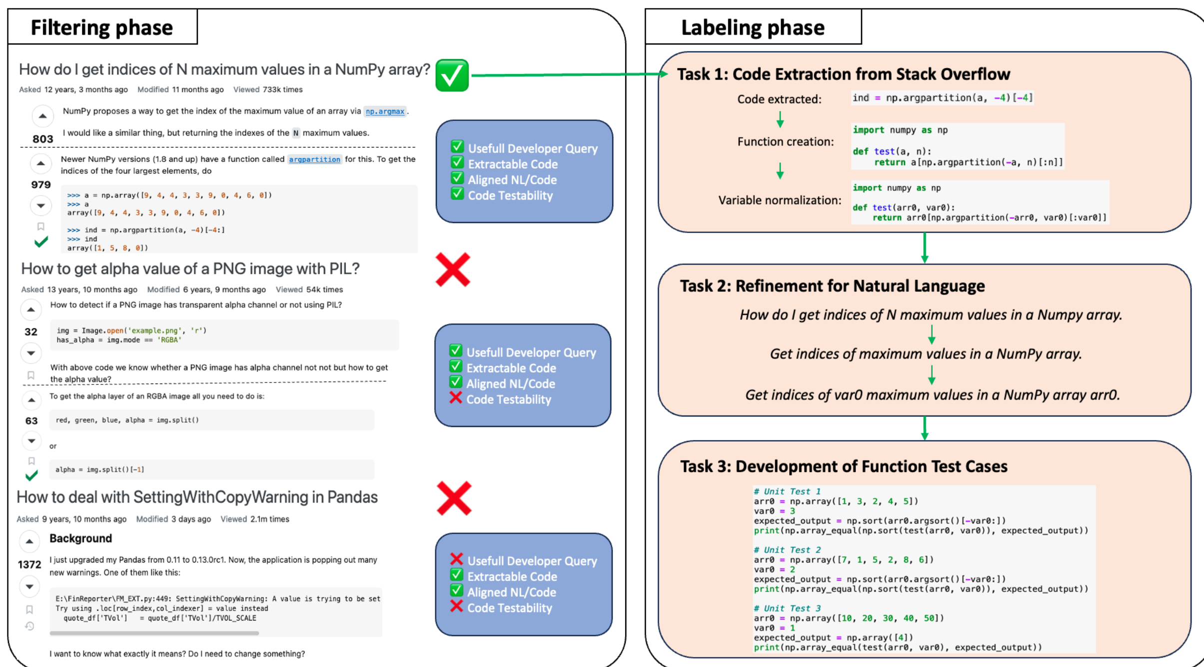

Figure 1: Curation Workflow from Stack Overflow to Dataset - The filtering phase (left) screens questions based on usefulness, code extractability, alignment, and testability, with one example advancing. The labeling phase (right) details the annotation of this example: extracting and standardizing code, refining the question for clarity with normalized terms, and developing unit tests to validate the function.

ment and the code solution is necessary for maintaining semantic integrity.

Executable Code Samples The code must be functionally valid, capable of running in a designated environment, which is essential for both verifying its effectiveness and constructing unit tests. We decide to exclude code where we need to open or save a file.

After implementing our filtering process, we refined our initial collection of 7,300 examples to 2,707 distinct problems, constituting about 37% of the original raw examples. This significant reduction is represented across various sources: from Stack Overflow's CoNaLa dataset, we retained 1,993 out of 5,500 examples; in the `Pandas`, `Numpy`, and `Regex` categories, the numbers were pruned down to 294, 242, and 178 from their respective totals of 600.

The low retention rate in my dataset can be attributed to various factors. For instance, some CoNaLa dataset examples were either non-testable or too specialized, necessitating extensive modification for practical use. Additionally, certain examples offered best practice advice or warnings rather than direct code solutions. The complexity of queries involving advanced libraries like `Pandas`, `Numpy`, and `Regex` also posed challenges. While these queries provide valuable specialized advice on Stack Overflow, they often require significant adaptation for generalization. The second and third examples on the left of Figure 1 illustrate these challenges: one involves extracting features from an image using a library, which is relevant but difficult to test due to the need for incorporating and processing images. The other example from the Pandas library focuses on best practices rather than direct coding solutions, not aligning with the dataset's aim for concrete developer tasks. More examples of what we consider as real-world problems, overly specialized queries, or edge cases are detailed in Appendix A.

2.3 Data Annotation

Our data annotation workflow is designed to prevent model memorization and instead cultivate problem-solving skills within the generated dataset. Through a multi-stage annotation process presented on the right of Figure 1, we curate selected examples from the filtering phase into delineated instances, which are then tested against specially crafted unit tests to ensure their correctness. By refining natural language focusing on the semantic relationships between functions and their descrip-

tions, we diminish the likelihood of models trained on massive datasets to merely replicate solutions seen in their training data, a concern highlighted by Lai et al. (2023) regarding examples sourced from Stack Overflow. To maintain focus and efficiency, annotators are allocated a strict twenty-minute window per example to ensure timely progression and a broad coverage of examples. The ensuing steps show our annotation strategy:

Task 1 - Code Extraction from Stack Overflow This initial phase entailed the extraction of code solutions from Stack Overflow in response to developers' inquiries. When the question admits more than one valid response, annotators are expected to capture alternate solutions as well, creating a supplementary example for the same intent. Upon extraction, they transform these snippets into a standardized Python function named `test`, systematically renaming arguments (e.g., `vari` for variables, `arri` for arrays, etc. See Appendix B for all normalized names). This normalization approach aligns with Yin et al. (2018), recognizing, as pointed out by Beau and Crabbé (2022), the significant influence this method has on models performance.

Task 2 - Refinement for Natural Language and Code Consistency During this stage, annotators refined the natural language descriptions to precisely correspond with the `test` function created in Task 1. The challenge lay in harmonizing the language descriptions with the Python code's logic, ensuring they are concise yet informative. Annotators were also tasked with incorporating normalized argument names into these descriptions to bolster the dataset's internal coherence and force the alignment.

Task 3 - Development of Function Test Cases The concluding annotation task involved the generation of 3 unique test cases for each `test` function, designed to rigorously assess the function's operational integrity and accuracy. These test cases include a normal scenario, an edge case, and an error situation, providing comprehensive coverage. This ensures a thorough yet time-efficient evaluation. Once the test cases have been passed, annotator can proceed the next example.

A team of five data science professionals, each with a minimum of five years of experience, contributed to the labeling of the filtered examples. They managed to complete the annotation in an average time of twelve minutes per example, amounting to a collective annotation effort of over 540 hours.

This process yielded a compendium of 3,402 examples derived from 2,702 distinct problem statements formulated by seasoned developers.

3 Dataset Statistics

This section outlines the statistical framework of our dataset, highlighting the diversity of programming tasks and the complexity of the included code samples. We approach the analysis from two angles: the representation of code libraries and different labels representing the characteristics of code. Key metrics such as item count, average words per natural language problem, and lines per code sample, alongside the depth of Abstract Syntax Tree (AST) to complete code analysis, are presented to give keys of the difficulty of the dataset.

3.1 Packages Statistics

The Table 1 illustrates the scope of CodeInsight, which encompasses a variety of packages.

A key aspect of CodeInsight is its focus on concise and precise problem descriptions, a departure from datasets that retain extensive problem contexts. This approach is aimed at reducing the word count in problem descriptions without sacrificing clarity and specificity, a crucial factor for effective code generation.

Code complexity is evaluated using two primary quantitative metrics: the mean line count of code alongside the depth of ASTs. The latter, a measure of syntactic structure complexity, serves to augment the insights gained from line count data. Analysis of AST depth within our dataset reveals a trend: more intricate coding structures, characterized by nested or conditional logic, are associated with deeper ASTs, whereas simpler, linear code correlates with shallower ASTs. Notably, across different packages in our dataset, the AST depth remains relatively consistent, with minor variations observed in packages like `Scipy` and `Scikit-learn`, potentially attributable to their smaller sample sizes.

Our dataset is compared with the DS-1000 dataset (Lai et al., 2023), the latter comprising 1,000 evaluation instances and utilizing specialized data science tools from Stack Overflow, complete with extensive problem descriptions. Despite similarities in code line counts and complexity, differences in AST depths, particularly in Pandas,

Package	Item Count		Avg. Prob Words		Avg. Code Lines		Avg. Depth AST	
	CodeInsight	DS-1000	CodeInsight	DS-1000	CodeInsight	DS-1000	CodeInsight	DS-1000
Full dataset	3,402	1,000	12.6 ± 4.3	140.0	4.6 ± 2.3	3.6	8.6 ± 1.5	8.5
`Pandas`	819	291	14.1 ± 4.2	184.8	3.59 ± 1.9	5.4	8.7 ± 1.4	10.7
`Numpy`	591	220	12.2 ± 3.3	137.5	5.3 ± 1.2	2.5	8.0 ± 1.4	8.1
`Scikit-learn`	19	115	13.8 ± 5.5	147.3	8.1 ± 7.4	3.3	7.6 ± 0.7	7.6
`Scipy`	8	106	13.0 ± 4.4	192.4	5.5 ± 1.3	3.1	6.5 ± 1.8	8.3
`NoImport`	415	-	12.1 ± 4.0	-	3.6 ± 1.9	-	8.2 ± 1.2	-
`Re`	241	-	12.2 ± 2.1	-	5.5 ± 0.8	-	8.1 ± 1.4	-
Other	1309	-	12.5 ± 3.2	-	6.1 ± 2.8	-	8.7 ± 0.9	-
`Matplotlib`	-	155	-	21.1	-	3.0	-	6.5
`TensorFlow`	-	45	-	192.4	-	3.1	-	7.8
`Pytorch`	-	68	-	133.4	-	2.1	-	8.2

Table 1: Comparative Analysis of Package Statistics in CodeInsight and DS-1000 Datasets. Standard deviations are reported where applicable. "-" indicates the package is not included in the dataset. Other contains 78 distinct packages like `Itertools`, `Collections`, `Operator`, etc. Detailed statistical data can be found in Appendix D.

indicate nuanced syntactic complexity variances. CodeInsight features a higher average number of unit tests per example, suggesting a more thorough evaluation methodology. Unlike DS-1000, which lacks detailed analysis of code model failures in unit tests, we provide a statistical breakdown of code categories to enhance understanding of model performance in the next Section.

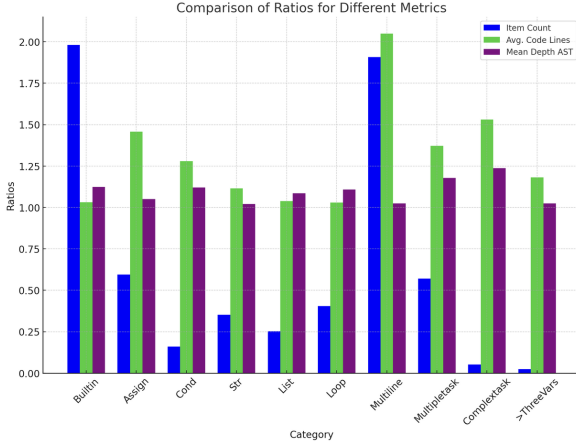

Figure 2: Ratio of positive (belonging to a specific category) to negative (not belonging to the category) examples for each of the 10 distinct *Categories* focusing on item count, average code lines and AST depths. Detailed statistical data supporting this analysis can be found in Appendix D.

3.2 Labels Statistics

In our study, we identified 10 *Categories* to enhance our analysis and gain a better understanding of our dataset. These predefined categories provide insights into the conditions under which models were successful or not. These categories vary from basic indicators like BUILTIN denoting the use of Python's built-in functions, to ASSIGN marking variable assignments. More complex categories include COMPLEXTASK for codes with multiple imports, and >THREEVARS for functions with over three arguments. Each example in the dataset is binary annotated—marked as positive if it falls under a category and negative otherwise. For a precise definition of all *Categories* , refer to Appendix C.

Figure 2 illustrates the ratio of positive to negative examples for each category to highlight the impact of each category. For example, we compare the ASSIGN category against all examples that do not include variable assignments. Our analysis primarily focuses on the most striking ratios, namely the item count, average code lines and average AST depth, as we found that the unit tests and average problem words exhibit minimal variation across the dataset. Detailed statistical data is provided in Appendix D.

The blue bars in the chart, representing item count ratios, significantly highlight the volume and distribution of data in each category. This showcases the prevalence of certain coding practices; for instance, the BUILTIN category, with nearly twice as many instances as its counterpart, suggests frequent utilization of built-in functions, indicative of a Pythonic approach in our dataset. In contrast, labels like COND and LOOP exhibit more balanced distributions, reflecting a diverse representation of these elements. Notably, categories such as COMPLEXTASK and >THREEVARS are less represented, aligning with the expectation of their complexity.

In the context of average code lines, represented by green bars in the graph, specific categories

such as COMPLEXTASK, MULTIPLETASK, and >THREEVARS exhibit notably higher ratios. This finding suggests a more intricate and voluminous nature of code associated with these tasks. Contrary to initial expectations, the LOOP category does not show an increased number of lines. Further investigation indicates that this outcome can be attributed to the frequent utilization of Python list comprehensions in this category, which typically reduces the number of code lines. In terms of AST depth, it remains relatively consistent for MULTILINE and >THREEVARS categories. This observation implies that longer codes or handling multiple variables do not necessarily correlate with increased syntactic complexity. However, in the cases of MULTIPLETASK and COMPLEXTASK, there is a correlation between the number of code lines and higher syntactic complexity. For other categories, the complexity levels remain to be stable.

Overall, this analysis underscores the diverse nature of coding practices and the value of categorization in understanding code complexity and coding styles in a nuanced manner. This category-based perspective evaluation on code analysis can be used in general to understand better model mistakes and way to improve model development.

4 Baselines

In this section, we test our dataset using state-of-the-art LLMs for code generation. Considering the volume and nature of our dataset, we explore various model evaluation methodologies. Initially, we employ a zero-shot evaluation framework, augmenting it with pre-prompts to align better with our specialized task. Subsequently, we experiment diverse partitioning strategies of the dataset for model fine-tuning, followed by evaluation on the remaining data. Additionally, we conduct a comparative performance analysis of GPT-4 on a subset of CoNaLa examples and their modified versions by our annotators, to understand the impact of language and code remodeling relatively to data contamination.

4.1 Experimental Setup

Models We evaluate the following pre-trained language models: Mistral 7B (Jiang et al., 2023) ; CodeLLAMA 13B (Rozière et al., 2023) and Starcoder 15B (Li et al., 2023). These models have been selected to provide a perspective on the scalability of model performance in relation to their size and the intricacies of code understanding and generation.

Evaluation Metrics We follow Lai et al. (2023) and measure the execution accuracy using the pass@1 metric i.e. we generate one code and test it agains all unit tests. We also use the BLEU score (Papineni et al., 2002) and the codeBLEU score (Ren et al., 2020) to complete our evaluation.

Model input For evaluation, we give to the model the intent in natural language and its associated function header with its arguments. Once the generation is finished, we automatically detect the end of the function -when it exists- to get the whole code and test it.

4.2 Prompting Evaluation

	Mistral	CodeLLAMA	Starcoder
Without Prompt	4.7%	44.7%	**45.1%**
First Prompt	4.9%	40.3%	**45.1%**
Second Prompt	10.1%	**48.1%**	46.8%

Table 2: Baselines result varying prompt method. We report the percentage of all unit tests passed (pass@1 score).

Without prompt Initially, the models were evaluated using the entire dataset without any additional context added to the natural language intent. The results, as presented in the Table 2, indicate a contrast in performance. Mistral showed notably lower efficiency compared to CodeLLAMA and Starcoder, which both passed nearly 45% of the unit tests. A key observation was the absence of a return statement in a significant proportion of the generated code. While Python allows for scenarios where not returning an explicit value is acceptable, such as actions or modifications without a return value, our dataset did not align with these scenarios. Mistral particularly exhibited a tendency (25% of the cases) to end functions with print statements instead of return statements, affecting its accuracy.

First prompt In an attempt to steer the models towards generating return statements for development aid tasks, a pre-prompt was introduced: *"You are a powerful code generation model. Your job is to convert a given natural language prompt into Python function code and return the result."*. Surprisingly, this prompt only marginally improved Mistral's performance, with a slight increase in return statement generation. However, it did not sig-

Split	Pass@1	BLEU	codeBLEU
20-80	$48.9 \pm 0.6\%$	50.0 ± 0.2	42.5 ± 0.1
40-60	$52.6 \pm 0.8\%$	58.1 ± 0.4	48.8 ± 0.4
60-40	$53.4 \pm 1.0\%$	57.9 ± 0.8	48.8 ± 0.7
80-20	$53.1 \pm 1.7\%$	57.9 ± 1.4	48.6 ± 1.2

Table 3: Scores for Different Splits of CodeLLaMA over five different seed. We report the mean and standard deviation for each metric.

nificantly affect the performance of CodeLLAMA and Starcoder. Notably, CodeLLAMA's performance even dropped to 40%, indicating that this prompting method might not be optimal.

Second prompt Aiming to further encourage the generation of return statements, a different prompt, *"Return the Result."* was added to the end of the natural language intent. This change led to an overall improvement in performance across all models, with CodeLLAMA outperforming Starcoder. Mistral, although still lagging, showed an improvement, successfully passing 10.1% of the unit tests.

4.3 Fine-Tuning Evaluation

This segment delves into fine-tuning configurations to discern their impact on model efficacy.

Splitting Method For the assembly of our test subset, we curated a collection of 3,094 unique problems, each along with at least three unit tests to ensure an assessment of model performance. This selection criterion is grounded in the necessity for test case coverage, which is important in evaluating model robustness. Out of this repository, we allocated different subset to evaluate the need of a train set to perform on test set.

Fine-Tuning Details We finetuned using Lora with $r = 16$ and $\alpha = 16$. The LoRA layer incorporated a dropout rate of 0.05 and was configured without bias adjustments. The batch size was established at 128, encompassing a warmup phase of 100 steps and an overall training regimen of 400 steps. The learning rate was set at 3×10^{-5}, with the optimization executed using the AdamW algorithm. To optimize computational efficiency, training was conducted using half-precision computation (FP16) on an a100 GPU with 40GB memory.

We crafted four distinct training/test splits - 20-80, 40-60, 60-40, and 80-20 - to fine-tune the CodeLLaMa model. Each split was evaluated over five different seeds, and the results are depicted in the following table.

In our analysis, we noticed that the performance scores for CodeLLaMa exhibit minimal variation when the training set ranges between 40% to 80%. Interestingly, these scores surpass those achieved through prompting alone. It appears that fine-tuning with just 20% of the dataset approaches the performance levels seen with prompting methods, yet it falls short by approximately 4 percentage points in the pass@1 metric and at least 6 points in both BLEU and codeBLEU scores. Given our objective to maximize the utilization of unit tests, we have determined that a 40-60 split represents the most optimal division for the final configuration of the CodeInsight dataset. This decision is grounded in achieving a balanced approach between training efficacy and test coverage.

4.4 Results

Finally, we chose the 40-60 split to perform our final evaluation on our baselines. We report the result in Table 5. The Table highlights that fine-tuning has a varied impact on different models. Fine-tuning yields comparable outcomes for Starcoder and CodeLLaMa, each passing slightly over half of the problems. Notably, Starcoder excels in complex tasks like COMPLEXTASK and >THREEVARS, though it drops to 30% in logical complex tasks. Regex, being a distinct language, poses challenges for all models. Interestingly, Mistral shows significant improvement post-finetuning, adapting well to the task with 38.4% test pass rate. However, Mistral struggles with complex tasks and Regex, likely due to its non-code-specific pre-training, unlike the other two models.

We provide a deeper error analysis of CodeLLaMa in Appendix E.

4.5 Exploring remodeling relatively to data contamination

In Section 2.3, we detail our approach to mitigate data contamination by rephrasing natural language intents and converting code snippets into function formats. Out of 2,379 CoNaLa handwritten examples, we annotated 812 for analysis. Considering the possibility of these examples being included in GPT-4's training set—a model not open for fine-tuning—we evaluated its zero-shot performance on them, achieving a BLEU score of 58.8.

Further analysis was conducted on the 812 examples post-annotation to assess the impact of our

Dataset	Problems	Evaluation	Avg. Test Cases	Avg. P Words	Avg. Lines of Code	Data Source	Train Set
HumanEval	164	Test Cases	7.7	23.0	6.3	Hand-Written	No
MBPP	974	Test Cases	3.0	15.7	6.7	Hand-Written	No
APPS	5000	Test Cases	13.2	293.2	18.0	Competitions	Yes
JulCe	1981	Exact Match + BLEU	–	57.2	3.3	Notebooks	No
DSP	1119	Test Cases	2.1	71.9	4.5	Notebooks	No
CoNaLa	500	BLEU	–	13.8	1.1	StackOverflow	Yes
Odex	945	Test Cases	1.8	14.5	3.9	Stack Overflow	No
DS-1000	1000	Test Cases	1.6	140.0	3.6	StackOverflow	No
CodeInsight	1860	Test Cases	3.0	12.6	4.7	StackOverflow	Yes

Table 4: Comparison of Test Set Statistics for CodeInsight with recent Code Generation Datasets

Category	Total	Starcoder	CodeLLAMA	Mistral
Full Dataset	1860	52.5%	**53.1%**	38.4%
Labels				
MultiLine	1258	**51.8%**	50.2%	42.0%
Assign	703	47.0%	**48.2%**	40.5%
MultipleTask	692	**44.5%**	42.2%	39.8%
Builtin	1292	**51.2%**	49.8%	41.9%
Cond	260	46.7%	**47.6%**	38.3%
Loop	573	**48.9%**	47.8%	40.4%
List	408	49.0%	**49.5%**	41.2%
>ThreeVars	47	**53.5%**	53.1%	42.3%
ComplexTask	90	**35.6%**	34.5%	23.1%
Packages				
`Pandas`	458	**56.0%**	55.2%	44.8%
`Numpy`	335	**53.6%**	52.8%	43.2%
`NoImport`	775	**54.1%**	53.9%	44.0%
`Regex`	133	37.5%	**38.3%**	26.2%

Table 5: Baselines Result on final Test Set split 40-60. We report the pass@1 for all models.

modifications. This evaluation resulted in a BLEU reduction to 47.6. Remarkably, without fine-tuning, GPT-4 passed 64% of unit tests for these examples, indicating its effectiveness in understanding natural language.

The performance of GPT-4, despite a drop in BLEU score, suggests its coding capabilities rather than full data leakage from its training phase. The contrast in BLEU scores before and after annotation suggests our approach's impact. Appendix F presents GPT-4's predictions, illustrating potential memorization.

5 Related Works

We introduce a comparative analysis, as detailed in Table 4, to assess our evaluation set against prevailing code generation datasets. This analysis clusters HumanEval (Chen et al., 2021b), MBPP (Austin et al., 2021), and APPS (Hendrycks et al., 2021) due to their emphasis on resolving comprehensive programming challenges. Our dataset, however, is distinguished by its focus on development assistance, which is typically characterized by a lower average line count in the provided code examples, reflecting a different use case compared to the aforementioned datasets. It is important to note that many datasets are designed primarily for the purpose of evaluating LLMs, which aligns with the prevalent trend in the field. However, the CodeInsight dataset sets itself apart by offering an average number of unit tests per example that exceeds those found in datasets oriented towards data science, such as DSP (Chandel et al., 2022), DS-1000 (Lai et al., 2023), and ODEX (Wang et al., 2022). This difference is largely due to the specific requirements of data science code generation tasks, which often necessitate fewer but more complex test cases, dealing with sophisticated input objects like square matrices, classifiers, or dataframes.

Featuring three distinct unit tests per example, a specialized training set, and predefined labels for in-depth performance analysis, the CodeInsight dataset represents an unparalleled resource. It supports fine-tuning and a comprehensive evaluation of code generation models, offering a novel approach to enhance their development and assessment.

6 Conclusion

In conclusion, CodeInsight proposes a new framework for testing code generation, specialized in assisting developers. It adeptly links natural language and code in more than 3,402 problems, providing a robust platform for model training and evaluation. The dataset's strength lies in its diversity, expert annotation, and focus on practical coding scenarios, making it a valuable asset in the intersection of computational linguistics and code generation research. Thanks to its categories, it allows a more precise comprehension of code generation model on this task and is completely compatible with other datasets for development aid.

Limitations

The CodeInsight dataset, while innovative, presents several limitations. Firstly, its specialized nature in development aid may not fully represent the broader spectrum of coding challenges. Expert annotations, while valuable, could introduce biases and may not capture diverse coding methodologies. Additionally, the dataset's current scope may limit its adaptability to evolving programming languages and practices. Furthermore, its reliance on Python restricts its applicability across different programming environments. These limitations suggest areas for future expansion and improvement to enhance the dataset's comprehensiveness and applicability in diverse coding contexts.

References

Jacob Austin, Augustus Odena, Maxwell I. 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. *CoRR*, abs/2108.07732.

Nathanaël Beau and Benoît Crabbé. 2022. The impact of lexical and grammatical processing on generating code from natural language. In *Findings of the Association for Computational Linguistics: ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 2204–2214. Association for Computational Linguistics.

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri 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, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew 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. *CoRR*, abs/2107.03374.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri 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, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021b. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

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. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b. *CoRR*, abs/2310.06825.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 18319–18345. PMLR.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Moustafa-Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra,

Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you! *CoRR*, abs/2305.06161.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*, pages 311–318. ACL.

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

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2022. Execution-based evaluation for open-domain code generation. *CoRR*, abs/2212.10481.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 476–486. ACM.

A Detailed overview of filtering phase

We include two tables that analyze the exploitability of examples from the CoNaLa dataset. The Table 6 presents the 10 examples with the highest probability of exploitability, highlighting their votes, titles, and whether they are exploitable. The Table 7 displays a random selection of 10 examples from the same dataset, also detailing their exploitability probability, votes, and titles.

We provide a detailed description of an accepted example, a rejected example and a borderline case for passing the filtering phase.

Accepted example We detailed the accepted example which is the first one on the left of the Figure 1. This particular example, a query about finding the largest values in a numpy array, demonstrates a typical developer's question due to unfamiliarity with specific numpy functions. Its solution, involving the `argpartition` function, is directly responsive to the query and easily testable, making it a perfect fit for our dataset.

P(expl)	Vote	Title	Exploitability
0.87	+8	Sort a nested list by two elements	Yes
0.85	+61	Converting integer to list in python	Yes
0.85	+37	Converting byte string in unicode string	Yes
0.85	+7	List of arguments with argparse	No
0.84	+20	How to convert a Date string to a DateTime object?	Yes/No
0.82	+64	Converting html to text with Python	Yes
0.81	+8	Ordering a list of dictionaries in python	Yes
0.81	+4	Two Combination Lists from One List	No
0.80	+4	Creating a list of dictionaries in python	No
0.79	+16	get index of character in python list	Yes

Table 6: Exploitability of the 10th examples with highest P(exploitability) from CoNaLa dataset

P(expl)	Vote	Title	Exploitability
0.75	+11	How can I plot hysteresis in matplotlib?	No
0.67	+499	How can I get list of values from dict?	Yes
0.71	+7	How do I stack two DataFrames next to each other in Pandas?	Yes
0.56	+4	List sorting with multiple attributes and mixed order	No
0.10	+7	Set x-axis intervals(ticks) for graph of Pandas DataFrame	No
0.26	+6	pandas binning a list based on qcut of another list	No
0.05	+1989	Determine the type of an object?	Yes
0.03	+11	Saving an animated GIF in Pillow	No
0.02	+5	Quiver or Barb with a date axis	No
0.018	+6	Can't pretty print json from python	No
0.008	+31	For loop - like Python range function	No

Table 7: Exploitability of 10th random from CoNaLa dataset

Rejected example The *"List sorting with multiple attributes and mixed order"* question on Stack Overflow from Table 7 presents an excessive level of specificity for inclusion whereas it has a high P(expl) value. This question delves into sorting a list by different attributes of a particular class, emphasizing the specific class's complexity rather than a broader understanding of sorting functions. The high level of detail in both the problem and its solution complicates the extraction of universally applicable code examples. Therefore, including it may not aptly represent the range of coding tasks and challenges.

Edge example An example such as *"How to convert a Date string to a datetime object?"* presented in Table 6 necessitates a more specific reformulation for a precise coding answer, like "How to compare a date string in ISO format to a datetime object." This demands the annotator's understanding of ISO format data. These are considered edge cases in our dataset and are included based on the annotator's expertise, who are constrained to a maximum of 20 minutes per annotation process.

LABEL	CONDITION
vari	Variable
dicti	Dictionary
arri	Array
dfi	Dataframe
stri	String
lsti	List
mati	Matrix
inti	Int

Table 8: List of normalized variable names used in our dataset

B Normalized variable names

The Table 8 outlines the standardized variable names utilized in the dataset, such as `vari` for 'Variable' and `dicti` for 'Dictionary', where i correspond to the number of the element appearing. This approach also allows for evaluating model efficacy with or without these normalized names. Note that `vari` is employed universally, even when alternatives might be applicable, without affecting test outcomes.

C Code Categories

Label	Condition Description
ASSIGN	Includes variable assignment.
BUILTIN	Uses a built-in function.
COND	Has conditional statement(s).
LOOP	Contains 'for' or 'while' loops.
STR	Performs string operation(s).
LIST	Uses list method(s).
MULTILINE	Code exceeds two lines.
MULTIPLETASK	Has $\geq$3 other Labels.
>THREEVARS	Function with >3 parameters.
COMPLEXTASK	Has $\geq$2 imports

Table 9: Detailed Labels for Automated Annotation

D CodeInsight Statistics

The two tables provide a detailed statistical analysis of the CodeInsight dataset, breaking down by Packages and Labels. The Table 10 covers various Python packages like Pandas, Numpy, and Regex, detailing the item count, average problem words, code lines, and unit tests. The second Table 11 analyzes different labels presented in Appendix C such as Builtin, Assign, Cond, and others, also including their item count and average metrics. Both tables gives insight on the dataset's complexity and diversity into the typical problem structure and testing framework associated with different programming constructs and packages.

	Item Count	Avg. Prob Words	Avg. Code Lines	Avg. Unit Tests
Full dataset	3,402	12.6 ± 4.3	4.6 ± 2.3	3.0 ± 0.4
NoImport	415	12.1 ± 4.0	3.6 ± 1.9	3.0 ± 0.4
Pandas	819	14.1 ± 4.2	5.4 ± 1.8	3.0 ± 0.2
Numpy	591	12.2 ± 3.3	5.3 ± 2.0	3.0 ± 0.2
Re	241	12.2 ± 2.1	5.5 ± 0.8	3.0 ± 0.2
Scikit-learn	19	13.8 ± 5.5	8.1 ± 7.4	3.0 ± 0.0
Scipy	8	13.0 ± 4.4	5.5 ± 1.3	3.0 ± 0.0
Itertools	55	11.8 ± 3.5	6.4 ± 3.1	3.0 ± 0.4
Collections	39	13.1 ± 3.5	6.8 ± 2.6	3.0 ± 0.2
Operator	43	13.4 ± 3.0	5.0 ± 1.4	3.2 ± 0.5
String	8	9.0 ± 1.8	5.8 ± 1.1	3.0 ± 0.0
Random	14	12.0 ± 2.0	5.4 ± 2.4	2.9 ± 0.5
Math	8	13.1 ± 4.7	6.0 ± 1.9	2.9 ± 0.3

Table 10: Statistical analysis of Packages in CodeInsight. We report including Item Count, Average Problem Words, Code Lines, and Unit Tests with Standard Deviations.

	Item Count	Avg. Prob Words	Avg. Code Lines	Avg. AST depth
Full dataset	3402	12.6 ± 4.3	4.6 ± 2.3	3.0 ± 0.4
BUILTIN	2261	12.7 ± 3.8	4.7 ± 2.2	8.7 ± 1.5
NOBUILTIN	1141	12.4 ± 3.6	4.6 ± 1.4	7.7 ± 1.2
ASSIGN	1269	13.2 ± 3.9	5.8 ± 2.4	8.6 ± 1.4
NOASSIGN	2133	12.3 ± 3.6	4.0 ± 1.4	8.2 ± 1.5
COND	471	13.4 ± 3.8	5.8 ± 2.9	9.2 ± 1.3
NOCOND	2931	12.5 ± 3.8	4.5 ± 1.8	8.2 ± 1.4
STR	885	12.8 ± 3.5	5.1 ± 2.0	8.5 ± 1.6
NOSTR	2517	12.6 ± 3.9	4.5 ± 2.0	8.3 ± 1.5
LIST	685	12.8 ± 3.8	4.8 ± 3.0	8.9 ± 1.3
NOLIST	2717	12.6 ± 3.8	4.7 ± 1.6	8.2 ± 1.5
LOOP	981	12.8 ± 3.8	4.8 ± 2.8	9.0 ± 1.3
NOLOOP	2421	12.5 ± 3.8	4.6 ± 1.5	8.2 ± 1.5
MULTILINE	2232	12.8 ± 3.7	5.5 ± 1.9	8.5 ± 1.5
NOMULTILINE	1170	12.2 ± 3.9	2.7 ± 0.5	8.3 ± 1.4
MULTIPLETASK	1236	13.2 ± 3.8	5.6 ± 2.5	8.9 ± 1.5
NOMULTIPLETASK	2166	12.3 ± 3.7	4.1 ± 1.4	7.8 ± 1.4
COMPLEXTASK	169	13.2 ± 3.8	7.0 ± 2.8	10.4 ± 2.3
NOCOMPLEXTASK	3233	12.6 ± 3.8	4.6 ± 1.9	8.3 ± 1.5
>THREEVARS	82	16.9 ± 4.2	5.5 ± 1.2	8.6 ± 1.4
<=THREEVARS	3320	12.5 ± 3.7	4.7 ± 2.0	8.4 ± 1.5

Table 11: Statistical analysis of Labels in CodeInsight. We report including Item Count, Average Problem Words, Code Lines, and AST depth with Standard Deviations.

E Code Error Analysis

We present in Table 12 an error analysis of different CodeLLaMa outputs on our dataset.

For the first scenario, involving scikit-learn, the task is to generate a textual representation of a decision tree with specified feature names for readability. Despite providing the model with import statements and the function signature, the generated code omits the necessary argument syntax `feature_names=feature_names0`, resulting in a TypeError due to excessive positional arguments. This minor but pivotal error exemplifies a scenario where a developer could struggle with argument specification.

In the second case, the task is to split a string while retaining separators, such as punctuation or

Statement	Generated Code	Reference Code

Extract decision rules from a scikit-learn decision tree tree0 with feature_names feature_names0	`from sklearn.tree import export_text`
	`def test(tree0, feature_names0):`
	`    return export_text(tree0, feature_names0)`
		`from sklearn.tree import export_text`
		`def test(tree0, feature_names0):`
		`    return export_text(tree0, feature_names=feature_names0)`

Split a string str0 and keep the separators	`import re`
	`def test(str0: str) -> list:`
	`    return re.split(r’\W+’, str0)`
		`import re`
		`def test(str0: str) -> list:`
		`    return [x for x in re.split(r’(\W+)’, str0) if x]`

Reset index in a pandas dataframe df0	`import pandas as pd`
	`def test(df0):`
	`    df0.reset_index(inplace=True)`
	`    return df0`
		`import pandas as pd`
		`def test(df0):`
		`    return pd.DataFrame(df0.values, columns=df0.columns)`

Table 12: Error analysis of CodeLLaMa outputs on our final evaluation set.

special tokens like newline characters. The generated code, however, omits an essential parenthesis in the `re.split` function, leading to a split that excludes the separators. This highlights the dual complexity of understanding both Python and Regex syntaxes.

The final example presents an annotation discrepancy. It involves resetting the index of a Pandas dataframe without specific instructions on handling the old index. The model correctly employs the `reset_index` function, typically retaining the old index as a new column. However, the reference code, and consequently the unit tests, do not preserve the old index. Thus, while the generated code aligns with the stated task, it fails unit tests due to the discrepancy in index handling. This case underscores the need for nuanced dataset analysis and exemplifies the challenges of borderline scenarios in dataset construction.

F GPT-4's prediction

We investigate data contamination within the CoNaLa and our dataset by examining GPT-4's outputs on both dataset. This can offer information in re-writting example during data annotation phase to mitigate data contamination.

Intent	**Reference Solution**	**GPT-4 Prediction**
Convert a list of integers x into a single integer	`r = int(’’.join(map(str, x)))`	`r = int(’’.join(map(str, x)))`
Convert a DateTime string back to a DateTime object of format %Y-%m-%d %H:%M:%S	`datetime.strptime(’2010-11-13 10:33:54’, ’%Y-%m-%d %H:%M:%S’)`	`datetime.datetime.strptime(date_string, ’%Y-%m-%d %H:%M:%S’)`
Reverse sort dictionary `d` *based on its values*	`sorted(list(d.items()), key=lambda k_v: k_v[1], reverse=True)`	`sorted(list(d.items()), key=lambda k_v: k_v[1], reverse=True)`

Table 13: GPT-4's Outputs Comparison for CoNaLa

Intent	**Reference Solution**	**Prediction**
Convert a list of integers `lst0` *into a single integer.*	`def test(lst0):` `    return int(''.join(map(str, lst0)))`	`def test(lst0):` `    return int(''.join(map(str, lst0)))`

Convert a datetime string `str0` *back to datetime object of format %Y-%m-%d %H:%M:%S.*	`from datetime import datetime`
	
	`def test(str0):`
	`    return datetime.strptime(str0, "%Y-%m-%dT%H:%M:%S")`
		`from datetime import datetime`
		
		`def test(str0):`
		`    return datetime.strptime(str0, "%Y-%m-%dT%H:%M:%S")`

Sort in reversing order the items in dictionary `dict0` *by their first values.*	`def test(dict0):` `    return dict(sorted(dict0.items(), key=lambda item: item[1], reverse=True))`	`def test(dict0):` `    return dict(sorted(dict0.items(), key=itemgetter(1), reverse=True))`

Table 14: GPT-4's Outputs for CodeInsight

Discussion This discussion presents an analysis of GPT-4's outputs on equivalent examples from the CoNaLa dataset and our dataset, CodeInsight, as detailed in Tables 13 and 14.

For the CoNaLa dataset, the analysis of GPT-4's predictions reveals interesting observations for the first and third examples. Specifically, GPT-4 autonomously includes an assignment to variable `r` in its prediction for the first example, despite the absence of such instruction in the original example. Similarly, in the third example, GPT-4 employs the `k_v` variable, an uncommon choice, demonstrating a potential memorization to variable naming.

Conversely, the second example highlights GPT-4's ability to generalize. The model infers the use of `date_string` even though the specific datetime format is not explicitly mentioned in the intent, showcasing its adeptness at filling in contextual gaps based on the provided intent.

Regarding the outputs on CodeInsight for the same rewritten intents, GPT-4's accuracy remains consistent for the first example. The second example further underscores GPT-4's precision, where the model's prediction aligns exactly with the reference, attributed to a more clearly defined intent or to a memorization from CoNaLa.

The third example diverges in response using itemgetter instead of lambda function but notably passes our unit tests, illustrating GPT-4's capacity for generating viable alternative solutions. This indicates that despite potential differences, GPT-4's inherent generalization capabilities enable it to offer valid code solutions, reflecting its understanding of programming concepts without memorizing data.