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

Anonymous ACL submission

Abstract

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 standard Python code derived from Stack Overflow. Comprising 3,402 meticulously crafted examples by Python experts, our dataset is designed for both model finetuning and standalone evaluation. The examples have been carefully refined to reduce data contamination, a process confirmed by the performance of three leading models: Mistral 7B, CodeLLAMA 13B, and Starcoder 15B. This dataset not only involves an average of three unit tests but also categorizes examples in order to get more fine grained analysis, enhancing the understanding of models' strengths and weaknesses in specific coding tasks. 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 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 pioneering 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 comprehensive 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 an invaluable tool for understanding and enhancing the capabilities of LLMs in real-world programming contexts.

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 an in-depth statistical analysis of the dataset, highlighting its diverse applications. In Section 4, the dataset's efficacy is showcased through evaluations using various LLM baselines. Lastly, Section 5 situates CodeInsight within the broader landscape of code generation datasets, underscoring its unique contributions to aiding software development.

2 Dataset Construction

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

2.1 Data Sources

To cultivate a high-quality dataset conducive to the task of code generation, it is critical to source from platforms that mirror the nuanced challenges faced by developers in real-world scenarios and guarantee an effective alignment between descriptive language and functional code. Stack Overflow stands out as a cornerstone platform for such an undertaking.

Veritable Developer Queries The platform operates as a dynamic repository of queries authentically posed by developers, mirroring the real-time conundrums encountered in modern software development. This feature ensures that the collected dataset is a genuine reflection of the typical inquiries and solutions sought by developers, offering invaluable insights into their problem-solving processes.

Language-Code Alignment Stack Overflow's model, rooted in community engagement, inherently promotes precision and clarity. This communal scrutiny is pivotal in curating language-code pairs that are not only syntactically correct but semantically coherent, forming the bedrock for training advanced code generation models.

Balanced Code Complexity The structured nature of Stack Overflow allows for a strategic curation of code snippets, maintaining a consistent degree of complexity. Such deliberate selection is instrumental in creating a dataset that represents a spectrum of programming tasks, devoid of bias towards either trivial or overly complex samples.

Despite Stack Overflow's comprehensive repository of developer questions, not all contributions align with the 'how-to' structure crucial for our dataset. A 'how-to' question typically presents a clear, task-oriented query where the developer seeks a step-by-step solution or a method to accomplish a specific programming task. These questions are distinctly actionable and contain a direct request for code that achieves a particular objective. Following the analysis in Yin et al. (2018), only 36% of Python-tagged inquiries exhibit this 'how-to' format, rendering them suitable for our dataset's intention to support practical code generation.

To surmount the challenge of sourcing applicable examples, we have leveraged the CoNaLa dataset (Yin et al., 2018), which constitutes a refined compilation of potential "how-to" examples from Stack Overflow, filtered through a probabilistic methodology. The CoNaLa corpus predominantly contains Python code snippets that are directly representative of the tasks at hand and contain minimal external dependencies.

To broaden the scope and applicability of our dataset, we have deliberately 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 strategic 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 comprehensiveness to accommodate a wider range of computational operations.

The procedure for enriching our dataset 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 7,301 raw examples to serve as the foundation for our dataset.

2.2 Data Filtering

In the critical juncture between sourcing and preparing data, meticulous filtering is essential. The transition into the data preparation phase necessitates a discerning approach to select examples from the source, acknowledging that not all contributions

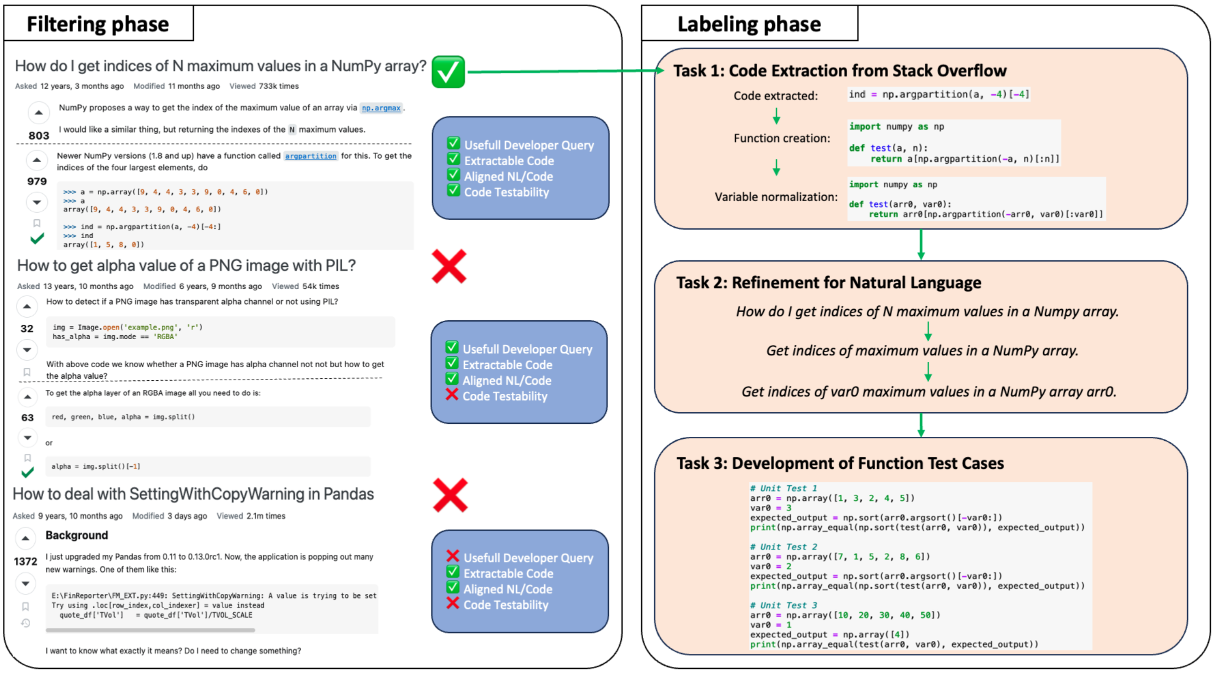

Figure 1: Curation Workflow from Stack Overflow to Dataset - The filtering phase (left) screens questions based on usefulness, code extractability, alignment, and testability, with one example advancing. The labeling phase (right) details the annotation of this example: extracting and standardizing code,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

Source	Initial number of examples	Number of examples after filtering
Total	7237	2707
SO_conala	5437	1993
SO_pandas	600	294
SO_numpy	600	242
SO_regex	600	178

Table 1: Exploitability of examples

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 stringent 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 statement 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.

These standards are meticulously upheld to refine the dataset and are instrumental in achieving our ambition to forge a dependable tool for code generation assistance.

During our filtering process, as detailed in Table 1, we distilled the initial compilation down to 2,707 distinct problems, which equates to merely 37% of the original volume meeting our criteria. This low rate can be attributed to several factors.

Predominantly, a portion of the CoNaLa dataset did not satisfy our criteria for testability or was excessively specialized, requiring extensive reworking for practical application. Additionally, some entries sourced from CoNaLa did not consist of Python code, as the collection methodology included automated systems with a broader capture net. The nuanced complexity of specific queries, especially those involving sophisticated libraries such as pandas, numpy, and regex, added another layer of challenge. These high-caliber queries, while esteemed within the Stack Overflow ecosystem for their specialized advice, often do not lend themselves to generalization without significant developer intervention, as in cases like *'Apply pandas function to column A to create multiple new columns'*.

Our approach deliberately sidesteps narrowly scoped inquiries to concentrate on those with a potential for broader application. The comprehensive filtering process is visually encapsulated on the left side of Figure 1, demonstrating the stringent yet necessary measures we employ to cultivate our source.

2.3 Data Labeling

Our data labeling workflow is carefully designed to preclude model memorization and instead cultivate genuine problem-solving skills within the generated dataset. Through a structured multi-stage annotation process, we transform selected examples from the filtering phase into clearly delineated, context-rich, and varied learning instances. This systematic refinement and evaluation of each example diminish the likelihood of models learning by rote and enhance their ability to generalize across diverse programming challenges. 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 outline our comprehensive annotation strategy:

Task 1 - Code Extraction from Stack Overflow This initial phase of annotation 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, annotators construct a Python function named `test`, transforming the snippet into a standardized format with arguments named systematically (e.g., `vari` for variables, `arri` for arrays, etc. See Appendix B for all normalized names) to maintain consistency across the dataset.

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 forged in Task 1. The challenge lay in harmonizing the language descriptions with the Python code's logic and argument structure. 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 unique test cases for each `test` function, designed to rigorously assess the function's operational integrity and accuracy. These test cases are instrumental in ascertaining the practical utility of the code and establishing the dataset's veracity. Once the test cases have been passed, annotator can proceed the next example.

As illustrated on the right-hand side of Figure 1, each task forms an integral component of our comprehensive annotation procedure. A team of five data science professionals, each boasting a minimum of four 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 extensive process yielded a compendium of 3,402 examples derived from 2,702 distinct problem statements formulated by seasoned developers. These examples, meticulously revised and recontextualized from their origins on StackOverflow, preventing complete memorization, offer a repository of unique and rigorously testable instances suitable for advancing code generation models.

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 number of unit tests per label, are presented to demonstrate the dataset's depth and the rigor of its composition.

3.1 Packages Statistics

In Table 2, our analysis presents a thorough comparative study of the CodeInsight dataset, underscoring its distinctive features and contributions,

Package	Item Count		Avg. Prob Words		Avg. Code Lines		Avg. Unit Tests	
	CodeInsight	DS-1000	CodeInsight	DS-1000	CodeInsight	DS-1000	CodeInsight	DS-1000
Full dataset	3,402	1,000	12.57 ± 4.25	140.0	4.58 ± 2.31	3.6	2.89 ± 0.54	1.6
`Pandas`	819	291	14.08 ± 4.15	184.8	3.59 ± 1.87	5.4	3.04 ± 0.35	1.7
`Numpy`	591	220	12.19 ± 3.25	137.5	5.25 ± 1.99	2.5	2.99 ± 0.20	2.0
`Scikit-learn`	19	115	13.79 ± 5.51	147.3	8.11 ± 7.41	3.3	3.00 ± 0.00	1.5
`Scipy`	8	106	13.00 ± 4.42	192.4	5.50 ± 1.32	3.1	3.00 ± 0.00	1.6
`NoImport`	1,557	-	12.10 ± 4.03	-	3.59 ± 1.87	-	3.04 ± 0.35	-
`Re`	241	-	12.20 ± 2.10	-	5.53 ± 0.77	-	3.01 ± 0.19	-
Other	167	-	12.46 ± 3.20	-	6.07 ± 2.80	-	3.03 ± 0.10	-
`Matplotlib`	-	155	-	21.1	-	3.0	-	1.0
`TensorFlow`	-	45	-	192.4	-	3.1	-	1.6
`Pytorch`	-	68	-	133.4	-	2.1	-	1.7

Table 2: 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 different packages like `Itertools`, `Collections`, `Operator`, etc.

particularly in the area of developmental aid and coding challenges. This table illustrates the expansive scope of CodeInsight, which encompasses a wide variety of packages, notably `Pandas`, `Numpy`, `Regex`, among others, culminating in a total of 3,402 examples. This extensive collection is indicative of CodeInsight's diverse problem types and coding methodologies.

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.

The dataset's diversity is further evident in the range of code complexity it presents. This is reflected in the average number of code lines and the standard deviations associated with them, demonstrating the broad spectrum of complexity within CodeInsight.

Towards the end of the analysis, we draw a comparison with the DS-1000 dataset, comprising 1,000 examples and featuring advanced data science packages like TensorFlow and Pytorch, which are not included in CodeInsight. Unlike CodeInsight, DS-1000 maintains the full context of Stack Overflow queries, which is reflected in its larger average problem word count. Despite these differences, there is a noticeable alignment in the average number of code lines and their standard deviations between the two datasets, suggesting a comparable level of complexity.

A striking observation from our study is the higher average number of unit tests in CodeInsight,

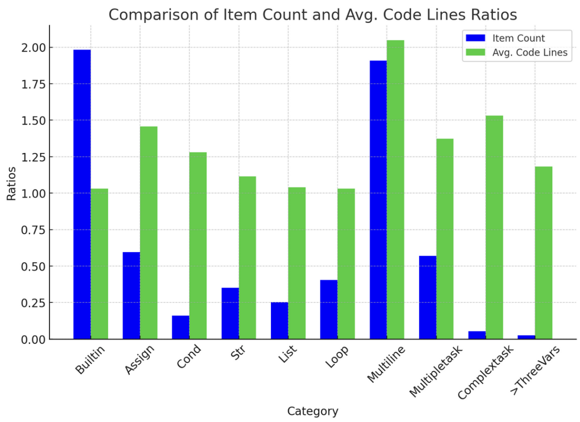

Figure 2: Ratios in *CodeInsight Categories* (listed in Appendix C). This figure presents the ratio of positive (belonging to a specific category) to negative (not belonging to the category) examples for each of the 10 distinct categories focusing on item count and average code lines. Detailed statistical data supporting this analysis can be found in Appendix D.

indicating a more robust testing methodology. This feature is vital for training models that require a deep understanding of code functionality and correctness.

In summary, this comparative analysis highlights the complementary nature of the CodeInsight and DS-1000 datasets in the field of code generation. Both datasets contribute uniquely to the development in this area, bringing their own strengths and focal points, thereby enriching the domain of computational linguistics and code generation research.

3.2 Labels Statistics

In our study, we identified 10 distinct *CodeInsight Categories* to enhance our analysis and gain a better understanding of our dataset. These predefined categories not only facilitated a nuanced analysis but also will 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 comprehensive definition of all *CodeInsight 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 and average code lines, 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.

Regarding the average code lines, depicted by green bars, categories like COMPLEXTASK, MULTIPLETASK, and >THREEVARS demonstrate significantly higher ratios, underscoring the complexity and extensive nature of the code in these tasks. Contrary to expectations, the LOOP category does not exhibit a greater number of lines. A closer examination reveals that this is due to the prevalent use of Python list comprehensions within this category, which accounts for the fewer lines of code than initially anticipated.

This multifaceted analysis offers an understanding of the CodeInsight dataset, revealing a harmonious blend of linguistic diversity and computational intricacy. These findings are crucial for developing computational models tailored to the dataset's unique properties, ensuring their effectiveness in various linguistic and programming scenarios.

In summary, the detailed metrics analysis underscores the intricate composition of the CodeInsight dataset. The CodeInsight dataset, with its diverse and well-structured composition, emerges as a valuable resource for advancing the frontiers of computational linguistics, particularly in the realm of code-related natural language processing.

4 Baselines

In this section, we test our dataset using state-of-the-art LLMs for code generation. Considering the volume and unique nature of our dataset as a tool for development, we explore various model evaluation methodologies. Initially, we employ a zero-shot evaluation framework, augmenting it with tailored pre-prompts to align closely with our specialized task. Subsequently, we experiment with diverse partitioning strategies of the dataset for model fine-tuning, followed by assessments on the remaining data. This allows us to critically analyze and contrast the effectiveness of fine-tuning and strategic prompting in enhancing model performance.

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 broad 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.

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

Split	Pass@1	BLEU	codeBLEU
20-80	$48.93 \pm 0.58\%$	50.03 ± 0.15	42.54 ± 0.07
40-60	$52.62 \pm 0.77\%$	58.14 ± 0.41	48.83 ± 0.40
60-40	$53.43 \pm 1.03\%$	57.93 ± 0.81	48.75 ± 0.66
80-20	$53.12 \pm 1.67\%$	57.94 ± 1.35	48.55 ± 1.16

Table 4: Scores for Different Splits of CodeLLaMA over five different seed. We report the mean and standard deviation for each metric.

4.2 Prompting Evaluation

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 3, indicate a stark 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-emptive text 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 significantly 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 various fine-tuning configurations to discern their impact on model efficacy.

Splitting Method For the assembly of our test subset, we meticulously curated a collection of 3,094 unique problems, each bolstered by at least three unit tests to ensure a thorough assessment of model performance. This selection criterion is grounded in the necessity for extensive test case coverage, which is instrumental in evaluating model robustness across a wide array of scenarios. Moreover, the exclusivity of problems in the test set serves to prevent potential memorization biases that could arise if models were exposed to these problems during training. 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

Dataset	Problems	Evaluation	Avg. Test Cases	Avg. P Words	Avg. Lines of Code Solution	Data Source
HumanEval	164	Test Cases	7.7	23.0	6.3	Hand-Written
MBPP	974	Test Cases	3.0	15.7	6.7	Hand-Written
APPS	5000	Test Cases	13.2	293.2	18.0	Competitions
JulCe	1981	Exact Match + BLEU	–	57.2	3.3	Notebooks
DSP	1119	Test Cases	2.1	71.9	4.5	Notebooks
CoNaLa	500	BLEU	–	13.8	1.1	StackOverflow
Odex	945	Test Cases	1.8	14.5	3.9	Stack Overflow + Hand-Written
DS-1000	1000	Test Cases + Surface-Form Constraints	1.6	140.0	3.6	StackOverflow
CodeInsight	1860	Test Cases	3.0	12.6	4.7	StackOverflow

Table 5: Comparison of Test Set Statistics for CodeInsight with Classic Code Generation Datasets

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

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

Finally, we chose the 40-60 split to perform our final evaluation on our baselines. We report the result in Table 6. 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.

5 Related Works

We present a comparative analysis highlighting how our evaluation set is designed to benchmark against existing code generation datasets in Table 5, many of which focus predominantly on evaluation data and may lack a specialized training set. Notably, the average number of unit tests in CodeInsight is much larger than other data science related datasets like DSP (Chandel et al., 2022), DS-1000 (Lai et al., 2023) and ODEX (Wang et al., 2022). More importantly, the problems in CodeInsight represent unique diverse and naturalistic intent and context formats that cannot be seen in any other datasets as we reformate the intent but also the code to create a function. Unlike generic Python code generation benchmarks (MBPP (Austin et al., 2021) and HumanEval (Chen et al., 2021b)), we note that other data science code generation benchmarks have fewer test cases in general since the annotators need to define program inputs with complex objects such as square matrices, classifiers, or dataframes rather than simple primitives, such as floats or lists. Our dataset contains 3 test cases for each test set problem which show the importance of our work to test all type of possibilities.

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 3,400 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 best 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.

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 Example of final exploitability from the CoNaLa dataset for filtering phase

we include two tables that analyze the exploitability of examples from the CoNaLa dataset. The Table 7)presents the 10 examples with the highest probability of exploitability, highlighting their votes, titles, and whether they are exploitable. The Table 8 displays a random selection of 10 examples from the same dataset, also detailing their exploitability probability, votes, and titles.

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	Non
0.84	+20	How to convert a Date string to a DateTime object?	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

Table 7: 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	+16	get index of character in python list	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 8: Exploitability of 10th random from CoNaLa dataset

B Normalized variable names

Label	Condition
`vari`	Variable
`dicti`	Dictionary
`arri`	Array
`dfi`	Dataframe
`stri`	String
`lsti`	List
`mati`	Matrix
`inti`	Int

Table 9: List of normalized variable names used in our dataset

The Table 9 lists normalized variable names used in the dataset. These names, like `vari` for 'Variable', `dicti` for 'Dictionary', and others, standardize the naming convention across the dataset. Note that `vari` can be used everytime, even when alternative names could also be used and it will not affect test case outcomes. However, this might introduce a slight increase in difficulty for models to correctly interpret and process the code, given the variability in naming conventions.

C CodeInsight 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 10: 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 11 covers various Python packages like Pandas, Numpy, and Regex, detailing the item count, average problem words, code lines, and unit tests. The second Table 12 analyzes different labels such as Builtin, Assign, Cond, and others, also including their item count and average metrics. Both tables reveal the dataset's complexity and diversity, offering insights 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	12.57 ± 4.25	4.58 ± 2.31	2.89 ± 0.54
`NoImport`	1557	12.10 ± 4.03	3.59 ± 1.87	3.04 ± 0.35
`Pandas`	819	14.08 ± 4.15	5.40 ± 1.81	3.00 ± 0.22
`Numpy`	591	12.19 ± 3.25	5.25 ± 1.99	2.99 ± 0.20
`Re`	241	12.20 ± 2.10	5.53 ± 0.77	3.01 ± 0.19
`Scikit-learn`	19	13.79 ± 5.51	8.11 ± 7.41	3.00 ± 0.00
`Scipy`	8	13.00 ± 4.42	5.50 ± 1.32	3.00 ± 0.00
`Itertools`	55	11.80 ± 3.46	6.40 ± 3.13	3.00 ± 0.38
`Collections`	39	13.05 ± 3.46	6.79 ± 2.55	3.03 ± 0.16
`Operator`	43	13.37 ± 2.99	5.02 ± 1.41	3.16 ± 0.48
`String`	8	9.00 ± 1.80	5.75 ± 1.09	3.00 ± 0.00
`Random`	14	12.00 ± 1.96	5.36 ± 2.41	2.86 ± 0.52
`Math`	8	13.13 ± 4.70	6.00 ± 1.94	2.88 ± 0.33

Table 11: 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. Unit Tests
Full dataset	3402	12.57 ± 4.25	4.58 ± 2.31	2.89 ± 0.54
BUILTIN	2261	12.70 ± 3.83	4.73 ± 2.20	3.02 ± 0.28
NOBUILTIN	1141	12.42 ± 3.62	4.59 ± 1.43	3.01 ± 0.29
ASSIGN	1269	13.16 ± 3.93	5.77 ± 2.35	3.00 ± 0.22
NOASSIGN	2133	12.26 ± 3.64	3.96 ± 1.40	3.03 ± 0.31
COND	471	13.39 ± 3.81	5.76 ± 2.85	3.05 ± 0.34
NOCOND	2931	12.49 ± 3.75	4.50 ± 1.80	3.01 ± 0.27
STR	885	12.80 ± 3.53	5.06 ± 2.03	3.02 ± 0.26
NOSTR	2517	12.55 ± 3.87	4.54 ± 2.01	3.02 ± 0.29
LIST	685	12.75 ± 3.76	4.83 ± 3.00	3.04 ± 0.30
NOLIST	2717	12.59 ± 3.78	4.65 ± 1.63	3.01 ± 0.27
LOOP	981	12.82 ± 3.83	4.78 ± 2.80	3.03 ± 0.28
NOLOOP	2421	12.53 ± 3.75	4.64 ± 1.53	3.01 ± 0.28
MULTILINE	2232	12.80 ± 3.73	5.51 ± 1.91	3.00 ± 0.24
NOMULTILINE	1170	12.20 ± 3.86	2.69 ± 0.46	3.06 ± 0.35
MULTIPLETASK	1236	13.16 ± 3.77	5.61 ± 2.52	3.01 ± 0.25
NOMULTIPLETASK	2166	12.27 ± 3.74	4.09 ± 1.39	3.02 ± 0.29
COMPLEXTASK	169	13.15 ± 3.76	6.98 ± 2.80	2.96 ± 0.27
NOCOMPLEXTASK	3233	12.59 ± 3.78	4.56 ± 1.88	3.02 ± 0.28
>THREEVARS	82	16.91 ± 4.17	5.52 ± 1.20	2.95 ± 0.38
<=THREEVARS	3320	12.51 ± 3.70	4.67 ± 2.02	3.02 ± 0.28

Table 12: Statistical analysis of Labels in CodeInsight. We report including Item Count, Average Problem Words, Code Lines, and Unit Tests with Standard Deviations.