Data Hound: Linking Educational Value to LLM Code Completion Performance During Inference

Boris Annink
Supervisor(s): Arie van Deursen, Maliheh Izadi, Jonathan Katzy, Razvan Popescu
EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Boris Annink
Final project course: CSE3000 Research Project
Thesis committee: Arie van Deursen, Maliheh Izadi, Jonathan Katzy, Razvan Popescu, Avishek Anand

Data Hound: Linking Educational Value to LLM Code Completion Performance During Inference

Boris Annink

Delft University of Technology
Delft, South-Holland, Netherlands

Abstract

This paper investigates the relation between the educational value of input code and the subsequent inference performance of code large language models (LLMs) on completion tasks. Results were attained using The Heap dataset and using SmolLM2, StarCoder 2 and Mellum models. Performance was measured by comparing the generated outputs with the ground truth, where high similarity indicates high performance. We analyse how factors such as language, model size, task type and granularity of educational value affect performance across educational value. We find that most factors do not have a relation with education value, as most metrics plateau except for exact-match. It is observed to have a consistent negative correlation with educational value. Additionally, a consistent turning point is seen around an educational value of 1.75, before which, performance tends to have a more positive relation with educational value. Results highlight the influence of input quality on LLM behaviour and offer insights for more effective training and evaluation strategies.

Keywords

Large Language Models, Educational Value, Data Smells, Performance, Code Completion

1 Introduction

In the race to build ever more capable language models, the quality and not just the quantity of training data is emerging as a key driver of performance. One method to improve the quality of the training dataset is to remove undesirable data, i.e. data smells [1]. Recent papers introduce educational value as a key data smell for improving the dataset [2]. This value represents how suitable the sample is for an academic assignment.

However, these papers only explore the impact of educational value for training and not inference. Since educational value is relevant during training, it is possible that relationships between educational value and performance also exist for testing. Additionally, models trained on selective ranges of educational value might perform differently on non-selected ranges. Insights into these relations could suggest new methods to improve code LLMs.

This paper addresses the following research questions:

RQ1 *What is the distribution of educational value in The Heap dataset?*

RQ2 *What is the impact of educational value on code completion performance of large language models during inference?*

RQ3 *How do task type, programming language, model size, and educational value granularity influence this impact?*

To answer these questions, we use The Heap dataset [3], which was decontaminated with respect to popular datasets (e.g. Stack v2 [4]). The models used were SmolLM2, StarCoder 2 and Mellum [4–6], which were selected for their relatively small size and accessibility. Educational value is computed using the classifiers developed for SmolLM2, which classify code on a 0-5 range. The performance this paper focuses on is measured as the similarity between the re-inferred and ground truth (original) code. This similarity is measured using exact-match, normalised edit similarity, BLEU, METEOR, and ROUGE-L [7–9]. Lastly, the models were prompted to infer next-line and next-20-token prediction on Java and Python subsets of The Heap.

We observe that most languages in the heap have a normal distribution of educational value with a mean near 2.25 and a standard deviation of 0.45. Additionally, for the relation between educational value and performance, it was observed that after a turning point of 1.75 educational value, little difference in performance is observed for all metrics, excluding exact-match. Exact-match, however, was shown to have a consistent drop in performance, suggesting that small differences are more likely as educational value increases. Before the aforementioned turning point of 1.75, most metrics exhibit a more positive correlation. The likely cause is that the model's prediction is of higher code quality than the ground truth, causing a mismatch. This is further supported by the results of investigating the influence of different educational value granularities, as this trend was most strongly observed for educational value specific to the ground truth. Other factors, such as model size and language, affect overall performance, but no relationship was observed with respect to educational value.

Further research into whether the claim that code scoring low in educational value drops in performance due to bad and unpredictable code holds. It could lead to advancements for models. For example, LLMs in an IDE setting could be discouraged from inferring on low educational value, preventing errors, and saving both computing time and the user's time. Other than preventing inference, it could also lead to stronger models for such code. This finding could also be used to improve testing datasets, allowing the dataset to be fine-tuned to highlight or avoid low educational value.

2 Background Research

In this section, background information for the project is provided. Aiming to give the reader a better understanding of the relevant concepts.

2.1 Large Language Models

With the release of Transformer models in 2017 [10], Natural Language Processing (NLP) was revolutionised. It allows for the efficient processing of large amounts of text. This led to the creation of large language models such as ChatGPT, Gemini, and Llama [11–13].

LLMs are developed by training neural networks on extensive text datasets. During training, the models learn statistical patterns in language, enabling them to generate coherent and contextually relevant text. This process involves optimising model parameters using large-scale machine learning algorithms. The resulting LLMs can perform various language-related tasks, such as translation, summarisation, and code generation.

Based on the provided context, the models respond by iteratively predicting the most likely next word, or more specifically, token.

Since the popularisation of LLMs, much research has been conducted to find methods that help improve performance. To provide more context on how LLMs work and 'think', the following will be dedicated to such found methods. Chain-of-Thought is an answering pattern where the model first talks to itself. This pattern attempts to capture the in-between steps that the model would be required to take to get the correct result consistently. Fill-In-the-Middle, popularised by StarCoder [14], is an input pattern where both context before and after the generation location is provided. The contexts and desired generation location are combined into a single input by marking the borders using special tokens.

2.2 Data Smells and Educational Value

Training an LLM requires large amounts of data. This makes handpicking the training dataset virtually impossible. This causes certain undesired data qualities to be included within the dataset and, by extension, the model. These undesired data qualities are referred to as data smells. These data smells can negatively impact the performance and reliability of AI-based systems [1]. These data smells are frequently defined and removed from the dataset to enhance the model.

The data smell that this paper focuses on is called educational value. Educational value is a quantitative, subjective measure that estimates how suitable the data is for a learning environment. This paper uses the classifiers made for SmolLM2. These classifiers have five criteria, each rewarding up to 1 point, resulting in a 0-5 score. These criteria are: validity of code, practicality, complexity (i.e. advanced programming topics), self-contained, and educationally worded (i.e. it reads as an assignment). Figure 1 shows an example for both low and high educational value. Since it is difficult to quantify these criteria, a judge LLM was used. It would be trained on correct scores of examples, allowing it to capture the desired criteria. More details on the training process of the classifier model can be found in SmolLM 2 paper [5] as well as the appendix section A.

Educational value is relevant to LLMs, as it was shown that removing low-scoring data significantly improved the performance of the model. This method was popularised by phi-1 [2] and has been adopted into many other models and datasets [5, 13, 15, 16].

```python
import math
import os
import sys
import random
import json

print(random.randint(1, 10)
```

(a) Low educational value (score: 1.72). No comments, unused imports, and syntax issue.

```python
def calculate_total_price(prices, tax_rate):
    """
    Calculates the total price including tax.

    Args:
        prices (list of float): Item prices.
        tax_rate (float): Tax rate as a decimal.

    Returns:
        float: Total price after tax.
    """
    subtotal = sum(prices)
    tax = subtotal * tax_rate
    return subtotal + tax

# Example usage
print(calculate_total_price([19.99, 5.49, 3.50], 0.07))
# Expected result: 31.008599999999998
```

(b) High educational value (score: 4.27). Clear structure, documentation, and example usage.

Figure 1: Examples illustrating low vs. high educational value in code.

2.3 Performance Measuring

This paper defines performance as high similarity between the ground truth and a model's prediction. Ground truth or reference refers to the original version that was regenerated. The regenerated output is the prediction. There are many ways to determine similarity between these two. This paper focuses mostly on three distinct levels. Exact-match, resulting only in a score of 1 if they exactly match each other. N-gram similarity, by metrics BLEU, METEOR and ROUGE, focusing more on the word level. And lastly, normalised edit similarity, a score derived from Levenshtein distance [17], which is the amount of character insertions, deletions and substitutions required to become an exact match. For this paper, this is normalised by the maximum possible changes and mirrored such that high values mean high similarity, in line with the other metrics.

This is not the only way to define performance. Other definitions include evaluating whether the generated code passes code tests (e.g. HumanEval and MultiPL-E [18, 19]) and code quality measured by static analysers.

3 Methodology

This section describes the selection of datasets and the design of two distinct prediction pipelines: Next-Line and Next-20-Token prediction. Additionally, it explains how educational value was computed and utilised. The general pipeline involves selecting a context and a corresponding target. The context is provided to the model as input. The models inferred output is then used as the prediction. During inference, no sampling is applied, ensuring deterministic prediction. High similarity between prediction and ground truth indicates strong model performance. The maximum context size was capped at 2000 tokens, as 85% of files are smaller and larger contexts substantially slow inference.

3.1 Dataset, Model, and Language Selection

The Heap dataset was chosen due to its minimal overlap with popular datasets. Any overlap that exists is explicitly annotated, facilitating easy removal. To avoid bias from models trained on the Stack 1 and Stack 2 datasets, samples from these datasets were excluded. Random samples were then selected based on the following criteria:

- No individual line of code may exceed 160 characters, preventing excessively long targets in Next-Line prediction.
- Each sample must contain at least five lines of code, ensuring sufficient context for model inference.

The selected models are SmolLM2 (135M, 360M, 1.7B), StarCoder 2 (3B, 7B), and Mellum (4B). They were selected for their availability, relevance, and size.

The main languages that were used in this paper are Python and Java. However, for evaluating the distribution of educational value in The Heap, additional languages were included. The following is a complete list of used languages: C, C++, C#, Go, Java, JavaScript, PHP, Python, Ruby, Rust, Swift, SQL. These were selected as they all have a subset in The Heap and a corresponding classifier available.

3.2 Educational Value

Educational value was computed using classifiers from the SmolLM2 model, taking the entire file content of each sample as input and producing a numerical value between 0 and 5. This value was utilised to determine the overall distribution of educational value across the dataset and also served as an independent variable in subsequent analyses. Additionally, an educational value mask was created specifically for Next-Line prediction, where each character in a file received a float corresponding to the educational value of the smallest encompassing code definition (classes, enums, or methods in Java, and classes or functions in Python). This mask enabled separate calculations of the mean educational value for context and target line. This mask was not used for next-20-token prediction, because it would be complex to deterministically find the corresponding character ranges for the token ranges.

3.3 Next-Line Prediction

A random line of code, excluding lines containing only whitespace or comments and the first line of the file, is selected as the target. The context consists of preceding lines, beginning at a line boundary, up to a maximum of 2000 tokens. The model predicts the next line, and the first complete line generated by the model is taken as the prediction. Due to generally low token counts, comparisons are performed on decoded text after removing whitespace characters (\n, \t, \r), focusing exclusively on code content.

3.4 Next-20-Token Prediction

The Next-20-Token pipeline evaluates LLM performance from a token-level perspective. A random sequence of 20 consecutive tokens is selected as the target, with context formed by preceding tokens, capped at 2000 tokens. The model generates exactly 20 tokens that are directly used as the prediction. Comparisons between predictions and references are performed both at the token and decoded text levels.

3.5 Comparing

Comparisons utilise multiple metrics. Exact-match assesses whether predictions match targets exactly. An edit-distance-derived score (normalised edit similarity) is calculated as $1 - \frac{\text{Levenshtein distance}}{\text{max text length}}$, yielding scores between 0 and 1, where 1 denotes a perfect match.

Token-level comparisons employ BLEU, METEOR, and ROUGE-L metrics:

- **BLEU** evaluates the precision of n-gram overlaps, penalising shorter outputs. For BLEU, method4 smoothing function was used, a commonly applied default smoothing method in BLEU evaluations. No smoothing function would result in lower scores, even if some similarity was observed.
- **METEOR** integrates precision, recall, and synonym matching, providing more nuanced evaluations.
- **ROUGE-L** assesses the longest common subsequence, reporting recall, precision, and F1-score.

To quantitatively determine correlation, Pearson's correlation [20] was computed using binned means of results, employing a bin size of 0.1.

4 Results

This section presents findings from the three main experiments: (1) an analysis of educational value distributions across languages in The Heap dataset, (2) an evaluation of how educational value correlates with model performance across various metrics and models, and (3) a deeper investigation into how this relationship is changes with different task type, model size, and educational value granularity. Together, these results provide insight into how educational value influences code generation performance and under what conditions this effect is most pronounced.

Most results were either obtained using a sample set of n = 100,000 or $n = 10\% \; of \; Heap$. When binning was applied, the binsize would be 0.1, and both the means and the standard deviation would be provided. For the plots the mean is represented as the full line, while a dashed line marks ±1 standard deviation. The results only show the results for the educational value range of 1-4. Full graphs can be found in the appendix section B.

4.1 Distribution of Educational Value in the Heap

To analyse how educational value is distributed across languages in The Heap dataset, we sampled 10% from each language subset. Figure 2 shows violin plots of these samples, revealing that most distributions are approximately normal with medians clustered between 2 and 2.5.

Table 1: Summary statistics of educational value for each Heap language subset (10% sample).

Language	Count	Mean	Std
C	307,647	1.84	0.45
Ruby	66,291	1.93	0.37
C++	446,982	2.02	0.40
JavaScript	190,780	2.06	0.45
Java	516,819	2.12	0.43
C#	325,745	2.14	0.46
Go	232,852	2.19	0.41
Python	159,591	2.23	0.53
PHP	331,024	2.24	0.42
Rust	80,270	2.27	0.52
SQL	4,080	2.38	0.44
Swift	43,484	2.42	0.54

Table 1 reports the sample sizes, means, and standard deviations. Most languages have a mean educational value between 2.0 and 2.25, with a standard deviation in the range of 0.4 to 0.5. This indicates that extremely low (near 0) or high (near 5) educational value samples are rare.

4.2 Impact of Educational Value on Model Performance

This section explores how model performance varies with educational value across different evaluation metrics. We analyse the divergence between exact match and edit similarity, identify a turning point within the data around 1.75, and highlight outlier behaviours such as Mellum's precision-recall imbalance.

4.2.1 More Positive Correlation Before the 1.75 Threshold. When splitting the data at an educational value of $x = 1.75$, we observe that the Pearson correlation between performance and educational value is almost always more positive in the lower range ($x \leq 1.75$) than in the upper range ($x > 1.75$). This pattern holds regardless of whether the individual correlations are positive or negative. Pre-turning-point correlations consistently show a stronger or more positive relationship.

This effect is observed consistently across both next-20-token and next-line prediction tasks, and for both Java and Python subsets. Table 2 highlights the trend for the Python subset with next-20-token prediction. While the exact shape of the curves varies, this relative shift in correlation remains stable. Additional plots supporting this finding are available in the appendix (Figure B).

4.2.2 Divergence Between Exact Match and Other Metrics Across Tasks and Languages. Figure 3 shows the difference between normalised edit similarity and exact match across educational value levels for the SmolLM2 1.7B model. Results are presented for both Python and Java subsets ($n = 100{,}000$ each), using next-20-token and next-line prediction with a bin size of 0.1.

As educational value increases, the difference between edit similarity and exact match consistently widens. This distance is larger with next-20-token prediction. This suggests that model outputs become more approximate: fewer exact matches but closer to the correct. The trend is visible across both languages and tasks, with Java consistently showing slightly larger differences than Python. Importantly, this widening is not unique to edit similarity; exact match also diverges from other metrics such as BLEU, ROUGE-L, and METEOR. This pattern holds invariably across all evaluated models, indicating that as educational value increases, outputs tend to become more partially correct rather than fully accurate, regardless of model size or architecture.

4.2.3 Mellum Scores High in Precision but Low in Recall. Figure 4 shows ROUGE-L precision and recall scores for Mellum 4B and StarCoder 2 3B across educational value levels on the Python subset. While StarCoder maintains a consistent balance between precision and recall, Mellum exhibits a strikingly different pattern: consistently high precision but markedly low recall.

4.3 Factors Influencing Educational Value Performance Trends

This section shows whether various factors influence the relationship between educational value and model performance. Specifically, we analyse the impact of task type, programming language, model size, and the granularity of educational value computation. These analyses provide insight into when and how educational value correlates with performance, and which configurations yield stronger or more consistent trends.

4.3.1 Effect of Task and Language on Educational Value Trends. Figure 5 compares model performance across educational value for two different programming languages (Python and Java) and two task setups (next-20-token and next-line prediction). The SmolLM2 1.7B model was used on subsets of 100,000 samples each, with a binning size of 0.1.

Overall, performance is consistently higher on Java than on Python, regardless of task setup. For the next-20-token prediction task, model performance increases steadily with educational value up to around $x = 1.75$, after which it plateaus. In contrast, the next-line task begins relatively stable but gradually decreases past the same turning point.

4.3.2 Larger Models Consistently Outperform Smaller Ones. Figure 6 compares exact match performance across educational value levels for models of varying sizes on the Python subset, using next-20-token prediction. Performance improves consistently with model size across the full range of educational values.

This trend is especially clear when comparing SmolLM2 135M, SmolLM2 360M, SmolLM2 1.7B, and StarCoder 2 3B. The difference becomes less pronounced between StarCoder 2 3B and StarCoder 2 7B, though the larger model still performs slightly better. Mellum 4B was excluded from this figure due to its odd performance (see Section 4.2.3 for details).

4.3.3 Educational Value Source Has Minimal Impact After 1.75. This experiment compares performance when educational value is computed at three levels: file (directly), context, and line (both averaged over the educational value mask). As shown in Figure 7, the choice of educational value source has a minor impact for samples with $x \leq 1.75$, where context-level values correlate with higher performance than file- or line-level values. In this range, the order typically is as follows: $line < file \leq context$.

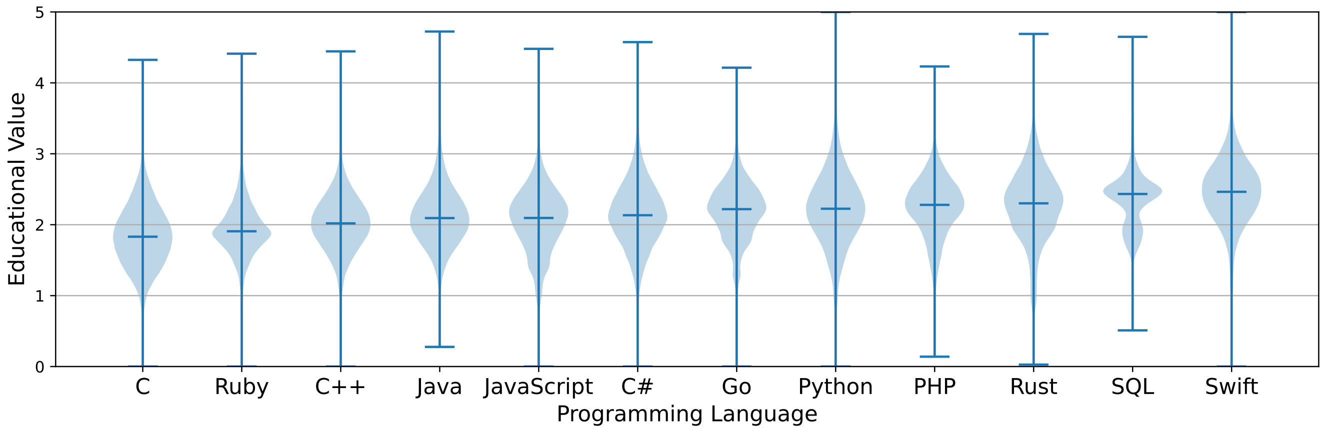


Figure 2: Violin plots of educational value for 10% samples from each Heap language subset. Most distributions are roughly normal with medians around 2–2.5.

Table 2: Pearson correlation coefficients (with p-values) between educational value and model performance for different metrics, before and after the turning point at $x = 1.75$. Each cell reports the correlation for samples with $x \leq 1.75$ and $1.75 < x < 3.5$, respectively. A strong drop in correlation after the threshold supports the presence of an L-shaped trend. Results are based on the Python subset ($n = 100{,}000$) using next-20-token prediction. Correlations were computed over binned means with a bin size of 0.1.

Model	ExactMatch	EditDistance	BLEU	METEOR	ROUGE L F1
SmolLM2 135M	0.88 (<0.01) / -0.87 (<0.01)	0.82 (0.01) / 0.32 (0.22)	0.79 (0.02) / -0.59 (0.01)	0.73 (0.04) / -0.18 (0.48)	0.71 (0.05) / -0.18 (0.50)
SmolLM2 360M	0.88 (<0.01) / -0.84 (<0.01)	0.84 (0.01) / -0.09 (0.74)	0.86 (0.01) / -0.62 (0.01)	0.79 (0.02) / -0.31 (0.22)	0.76 (0.03) / -0.14 (0.58)
SmolLM2 1.7B	0.92 (<0.01) / -0.88 (<0.01)	0.97 (<0.01) / 0.46 (0.07)	0.97 (<0.01) / -0.18 (0.50)	0.95 (<0.01) / 0.37 (0.14)	0.92 (<0.01) / 0.47 (0.05)
StarCoder 2 3B	0.94 (<0.01) / -0.88 (<0.01)	0.83 (0.01) / -0.54 (0.03)	0.87 (0.01) / -0.88 (<0.01)	0.83 (0.01) / -0.73 (<0.01)	0.81 (0.02) / -0.85 (<0.01)
StarCoder 2 7B	0.93 (<0.01) / -0.71 (<0.01)	0.78 (0.02) / 0.31 (0.23)	0.81 (0.02) / -0.15 (0.57)	0.71 (0.05) / 0.15 (0.57)	0.74 (0.04) / -0.03 (0.92)
Mellum 4B	0.89 (<0.01) / -0.55 (0.02)	0.93 (<0.01) / 0.87 (<0.01)	0.93 (<0.01) / 0.19 (0.46)	0.93 (<0.01) / 0.45 (0.07)	0.91 (<0.01) / 0.61 (0.01)

However, beyond $x = 1.75$, the difference between the performances diminishes. Suggesting that the granularity of educational value matters in lower ranges but less in higher educational value.

5 Discussion

This section aims to answer the research questions posed in the introduction and attempts to explain the underlying reason for the results and their significance.

RQ1: Distribution of Educational Value in The Heap Educational value in The Heap generally follows a normal distribution with most languages centred around a mean of 2–2.5 and a standard deviation of 0.4–0.5. Extreme values (near 0 or 5) are rare. This aligns with expectations, as most code in public repositories is functional, somewhat documented, and covers practical use cases, consistent with moderate educational value. Higher-level languages like Python, Swift, and SQL tend to have slightly higher educational values than lower-level ones like C and C++, likely due to their readability and abstraction level. Some exceptions (e.g., Ruby, Rust, and JavaScript) deviate slightly from this trend but still remain within a moderate range. These deviations may reflect differences in common usage patterns or community practices, such as less consistent documentation or varying code quality norms across ecosystems.

RQ2: Relation between inputs' educational value and code completion performance of LLMs Most metrics plateau or improve slightly with educational value, except exact match, which declines. This suggests that as educational value increases, the model's outputs remain similar in meaning but diverge more in exact form, likely due to greater complexity and variability in higher-quality code. A turning point is consistently observed at $x = 1.75$, after which the correlation weakens or reverses. This may be caused by the fact that low-education samples often contain flawed or inconsistent code, making them harder to mimic accurately, as it may prompt the model to produce improved but non-matching alternatives.

Mellum shows high precision but low recall, likely due to its token repetition and tokenisation. Its tokeniser includes a standalone space (' ') and many space-prefixed tokens (e.g., ' return'). Repeating ' ' often aligns with the reference, even when content is incorrect, inflating ROUGE-L precision. Since the model overgenerates whitespace more than the ground truth contains it, this skews results toward precision.

RQ3: Additional influences for performance across Educational Value

Java consistently outperforms Python, likely due to its structural predictability and boilerplate-heavy syntax. This difference would

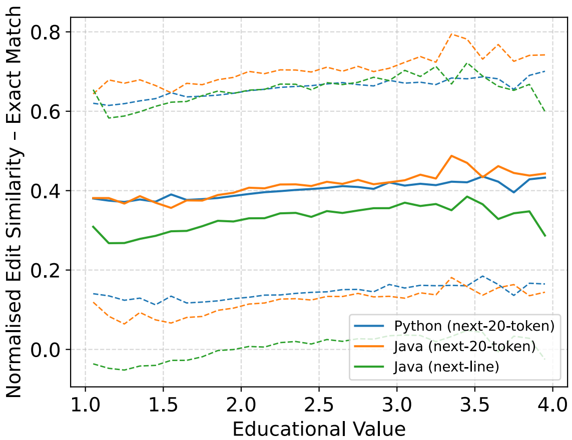


Figure 3: Difference between normalised edit similarity and exact match across educational value for the SmolLM2 1.7B model on Python and Java subsets (n = 100,000 each). Results are shown for both next-20-token and next-line prediction with a bin size of 0.1. Dashed lines indicate ±1 standard deviation. The difference between exact-match and normalised edit similarity widens as educational value increases.

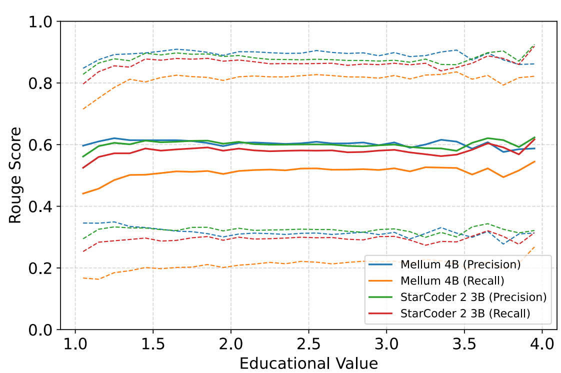


Figure 4: ROUGE-L precision and recall scores across educational value for Mellum 4B and StarCoder 2 3B on the Python subset (n = 100,000), using next-20-token prediction and a bin size of 0.1. Dashed lines indicate ±1 standard deviation. While StarCoder maintains a balance between precision and recall, Mellum exhibits notably higher precision but significantly lower recall across all educational value levels.

show effect throughout the whole educational value spectrum. An alternative reason could be that the used model, SmolLM2 1.7B, performs better on Java for code-completion tasks.

Task type also matters: next-line prediction shows a more negative trend with educational value. Closer inspection revealed that the model often predicted the correct line but with a comment marker prepended. Since evaluation strips whitespace, the comment marker captures leading spaces, causing a disproportionate

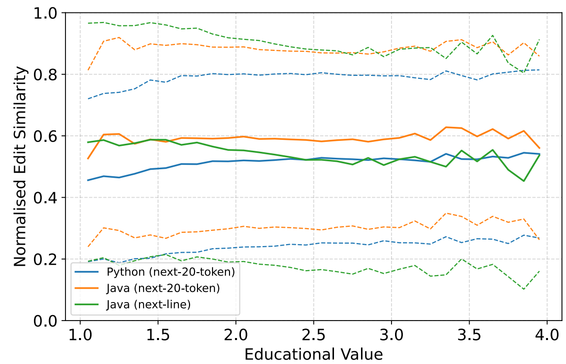


Figure 5: normalised edit similarity across educational value for Python and Java subsets (n = 100,000 each), using next-20-token and next-line prediction with a bin size of 0.1. Dashed lines indicate ±1 standard deviation. Results are shown for the SmolLM2 1.7B model. Models perform consistently better on Java than Python. While next-20-token performance shows a gradual increase with educational value, next-line prediction on Java exhibits a notable decline.

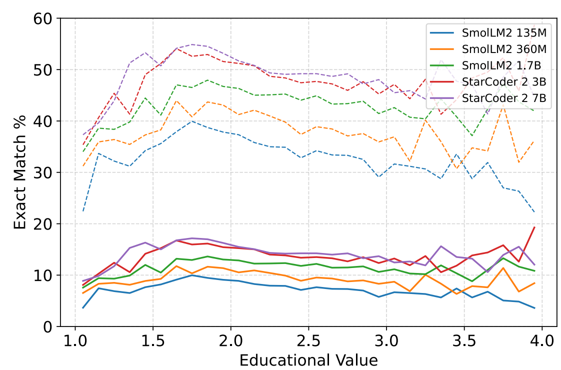


Figure 6: Exact match performance across educational value levels for different model sizes on the Python subset (n = 100,000), using next-20-token prediction with a bin size of 0.1. Dashed lines indicate ±1 standard deviation. Larger models consistently outperform smaller ones across the entire educational value spectrum, highlighting the benefits of scaling.

drop in score. This effect is likely a quirk of SmolLM2, caused by ending the input with a newline.

Model size has a clear positive impact—larger models perform better across the board. The difference between the performance of the different models does not seem to change much. This suggests that increasing the model size does not benefit a specific range of educational value.

Granularity of educational value becomes negligible after the 1.75 threshold, but below that, line-level values lead to lower performance than file- or context-level values. This supports the idea

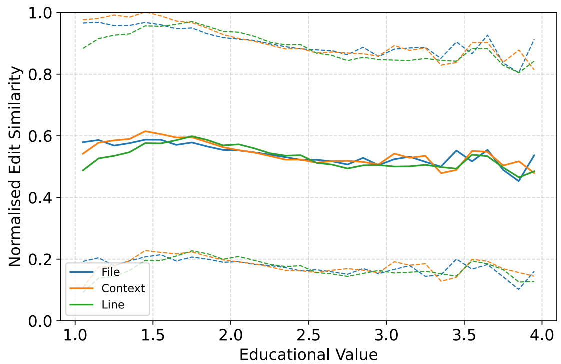


Figure 7: Normalised Edit Similarity scores by SmolLM2 1.7B for next-line prediction across varying educational value, using the Java subset (n = 100,000) with a bin size of 0.1. The three curves represent educational value computed at file, context, and line level. Dashed lines indicate ±1 standard deviation. A clear difference is visible for $x \leq 1.75$, after which the distinctions diminish.

that models may generate "improved" lines that diverge from the original, which would most strongly affect line-level comparisons.

It is important to note that these results, with the exception of model size, were only obtained using the models of SmolLM2. Thus, trends and relevant reasoning may not be representative of all models.

Impact of findings. These findings can be useful in a few practical cases. One such case would be for LLMs in code IDEs. As it could be used to prevent the model from inferring when educational value is too low, saving computing power and the users' time. Another application would be improving testing datasets for code completion. Though the exact reason why educational value below 1.75 scores poorly remains speculative. It can be asked whether such samples should belong in the testing dataset. If the claim that they perform poorly due to low code quality holds, then educational value could be a fast metric for filtering the dataset.

6 Limitations and Future Work

This project was heavily limited by time and computing power due to its dependency on generating using LLMs. Additionally, due to the low number of samples with extreme educational values, this paper couldn't fully explore the correlations across the full educational value spectrum. These were big factors in limiting the scope of this project. Another limitation is the random sampling strategy used to select the subset. In hindsight and for future research, it would be better both in terms of performance and data distribution to select the subset such that the distribution of educational value is more uniform.

This project could be extended by including more (larger) models and more extreme samples. Other methods for expanding the paper would include using different prediction methods (e.g., method-body prediction), looking into models with unique properties such as Chain-Of-Thought and Fill-In-the-Middle, measuring performance in additional ways, such as time to generate and code quality, and using different tasks for the model (e.g., bug-fixing). Furthermore, for future work, the results show a turning point near 1.75 educational value. The exact reason is speculatively assumed to be low-quality ground truth mismatching an improved prediction. Additional research would be required to confidently determine whether this claim holds.

7 Conclusion

This paper explored the relationship between educational value and the performance of code LLMs. Educational value in The Heap tends to follow a normal distribution. The exact mean and variance differ slightly between programming languages. Results show that metrics tend to plateau, but there are exceptions to the rule. Exact-match consistently correlates negatively with educational value, implying that near-misses are more likely for higher educational value. Additionally, an odd turning point is observed around a value of 1.75. The exact reason is unclear, but it is speculated to be caused by the model outperforming the ground truth regarding code quality. This can be supported by the finding that plotting performance against the different educational value granularities shows divergence before 1.75, where the line educational value performed the worst. Factors like language and model size influence the performance, but with little respect to educational value. Lastly, the results show that the target for what to predict heavily influences the trend line, as next-line prediction has a significantly more negative correlation than next-20-token prediction. However, these results may be caused by model-specific quirks and thus might not be representative of all code LLMs. The findings suggest that educational value could be used to prevent LLMs like IDEs from inferring on low-scoring code and prevent errors. It also hints that educational value is a possible data smell for not only training, but also testing datasets.

8 Responsible Research

During the project and the writing of the paper, the fundamentals of FAIR: Findable, Accessible, Interoperable, and Reusable were mostly followed. The datasets and models that were used are all publicly available on HuggingFace. Additionally, the code used to attain results will also be made publicly available in the following repository: https://github.com/AISE-TUDelft/data-hound. The fundamentals were not fully followed as the seeds used to select the subset were lost, but different seeds should yield similar, if not the same, results.

The data used in this project solely comes from The Heap dataset. This dataset was created by scraping GitHub according to the relevant section of 'GitHub Acceptable Use Policies'. On the following link https://github.com/AISE-TUDelft/The-Heap-opt-out, owners can submit a request to opt out of the dataset. However, due to the nature of scraping and opt-out, the dataset may contain sensitive data such as Personal Identifiers, private keys or other information the owner does not want to be included. These would be extremely rare, and during closer examination of the datasets, they were not seen.

(Generative) AI was used for multiple goals. For the paper, it was used to find errors in spelling and grammar and help with LaTeX formatting. For the project, it was used to elaborate on key concepts and generate code. Importantly, it was **not** used to replace critical thinking or analysis. Lastly, and most notably, it was used to generate results for evaluation, as it is a central element within this paper. All outputs were validated for accuracy. Examples showing how AI was used are included in the appendix section C.

References

[1] Harald Foidl, Michael Felderer, and Rudolf Ramler. Data Smells: Categories, Causes and Consequences, and Detection of Suspicious Data in AI-based Systems . In *2022 IEEE/ACM 1st International Conference on AI Engineering – Software Engineering for AI (CAIN)*, pages 229–239, Los Alamitos, CA, USA, May 2022. IEEE Computer Society.

[2] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. Textbooks are all you need, 2023.

[3] Jonathan Katzy, Razvan Mihai Popescu, Arie van Deursen, and Maliheh Izadi. The heap: A contamination-free multilingual code dataset for evaluating large language models, 2025.

[4] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder 2 and the stack v2: The next generation, 2024.

[5] Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Hynek Kydlíček, Agustín Piqueres Lajarín, Vaibhav Srivastav, Joshua Lochner, Caleb Fahlgren, Xuan-Son Nguyen, Clémentine Fourrier, Ben Burtenshaw, Hugo Larcher, Haojun Zhao, Cyril Zakka, Mathieu Morlon, Colin Raffel, Leandro von Werra, and Thomas Wolf. Smollm2: When smol goes big – data-centric training of a small language model, 2025.

[6] Nikita Pavlichenko, Iurii Nazarov, Ivan Dolgov, Ekaterina Garanina, Karol Lasocki, Julia Reshetnikova, Sergei Boitsov, Ivan Bondyrev, Dariia Karaeva, Maksim Sheptyakov, Dmitry Ustalov, Artem Mukhin, Semyon Proshev, Nikita Abramov, Olga Kolomyttseva, Kseniia Lysaniuk, Ilia Zavidnyi, Anton Semenkin, Vladislav Tankov, and Uladzislau Sazanovich. Mellum-4b-base, 2025.

[7] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In Pierre Isabelle, Eugene Charniak, and Dekang Lin, editors, *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics.

[8] Satanjeev Banerjee and Alon Lavie. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 65–72, Ann Arbor, Michigan, June 2005. Association for Computational Linguistics.

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

[10] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[11] OpenAI. Gpt-4 technical report, 2023.

[12] Google DeepMind and Google Research. Gemini: A family of highly capable multimodal models, 2023.

[13] AI@Meta. Llama 3 Model Card. https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md, 2024. Accessed: 2025-05-17.

[14] 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, Joel Lamy-Poirier, Joao Monteiro, Nicolas Gontier, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Ben Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason T Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Urvashi Bhattacharyya, Wenhao Yu, Sasha Luccioni, Paulo Villegas, Fedor Zhdanov, Tony Lee, Nadav Timor, Jennifer Ding, Claire S Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, 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. Starcoder: may the source be with you! *Transactions on Machine Learning Research*, 2023. Reproducibility Certification.

[15] Guilherme Penedo, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. The fineweb datasets: Decanting the web for the finest text data at scale, 2024.

[16] Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, Alon Benhaim, Misha Bilenko, Johan Bjorck, Sébastien Bubeck, Martin Cai, Qin Cai, Vishrav Chaudhary, Dong Chen, Dongdong Chen, Weizhu Chen, Yen-Chun Chen, Yi-Ling Chen, Hao Cheng, Parul Chopra, Xiyang Dai, Matthew Dixon, Ronen Eldan, Victor Fragoso, Jianfeng Gao, Mei Gao, Min Gao, Amit Garg, Allie Del Giorno, Abhishek Goswami, Suriya Gunasekar, Emman Haider, Junheng Hao, Russell J. Hewett, Wenxiang Hu, Jamie Huynh, Dan Iter, Sam Ade Jacobs, Mojan Javaheripi, Xin Jin, Nikos Karampatziakis, Piero Kauffmann, Mahoud Khademi, Dongwoo Kim, Young Jin Kim, Lev Kurilenko, James R. Lee, Yin Tat Lee, Yuanzhi Li, Yunsheng Li, Chen Liang, Lars Liden, Xihui Lin, Zeqi Lin, Ce Liu, Liyuan Liu, Mengchen Liu, Weishung Liu, Xiaodong Liu, Chong Luo, Piyush Madan, Ali Mahmoudzadeh, David Majercak, Matt Mazzola, Caio César Teodoro Mendes, Arindam Mitra, Hardik Modi, Anh Nguyen, Brandon Norick, Barun Patra, Daniel Perez-Becker, Thomas Portet, Reid Pryzant, Heyang Qin, Marko Radmilac, Liliang Ren, Gustavo de Rosa, Corby Rosset, Sambudha Roy, Olatunji Ruwase, Olli Saarikivi, Amin Saied, Adil Salim, Michael Santacroce, Shital Shah, Ning Shang, Hiteshi Sharma, Yelong Shen, Swadheen Shukla, Xia Song, Masahiro Tanaka, Andrea Tupini, Praneetha Vaddamanu, Chunyu Wang, Guanhua Wang, Lijuan Wang, Shuohang Wang, Xin Wang, Yu Wang, Rachel Ward, Wen Wen, Philipp Witte, Haiping Wu, Xiaoxia Wu, Michael Wyatt, Bin Xiao, Can Xu, Jiahang Xu, Weijian Xu, Jilong Xue, Sonali Yadav, Fan Yang, Jianwei Yang, Yifan Yang, Ziyi Yang, Donghan Yu, Lu Yuan, Chenruidong Zhang, Cyril Zhang, Jianwen Zhang, Li Lyna Zhang, Yi Zhang, Yue Zhang, Yunan Zhang, and Xiren Zhou. Phi-3 technical report: A highly capable language model locally on your phone, 2024.

[17] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[18] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri 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, Josh 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. Evaluating large language models trained on code, 2021.

[19] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691, 2023.

[20] Karl Pearson. Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, 58:240–242, 1895.

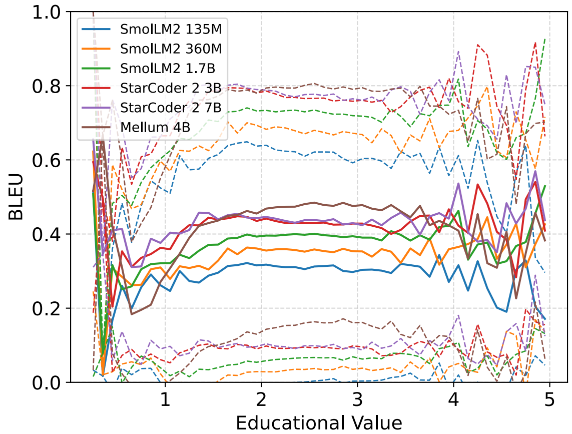
A How educational value (Stack-Edu) is evaluated

Educational value is computed using language-specific classifiers trained with the StarEncoder model on the StarCoder2Data dataset. The dataset is annotated with scores made by Llama3-70B-Instruct. The following is the prompt that was used to generate the annotated score for Python scripts.

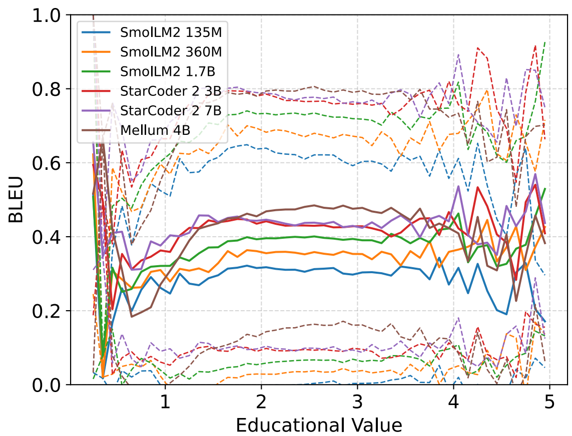
```
Below is an extract from a Python program. Evaluate
 whether it has a high educational value and could help
 teach coding. Use
the additive 5-point scoring system described below.
 Points are accumulated based on the satisfaction of
 each criterion:
- Add 1 point if the program contains valid Python code,
 even if  i t s  not educational, like boilerplate code,
 configs, and niche
concepts.
- Add another point if the program addresses practical
 concepts, even if it lacks comments.
- Award a third point if the program is suitable for
 educational use and introduces key concepts in
 programming, even if the
topic is advanced (e.g., deep learning). The code should
 be well-structured and contain some comments.
- Give a fourth point if the program is self-contained
 and highly relevant to teaching programming. It should
 be similar to a
school exercise, a tutorial, or a Python course section.
- Grant a fifth point if the program is outstanding in
 its educational value and is perfectly suited for
 teaching programming. It
should be well-written, easy to understand, and contain
 step-by-step explanations and comments.
The extract: <EXAMPLE>
After examining the extract:
- Briefly justify your total score, up to 100 words.
- Conclude with the score using the format: "Educational
 score: <total points>
```

Figure 8: Python prompt used to attain initial educational value scores.

B Results

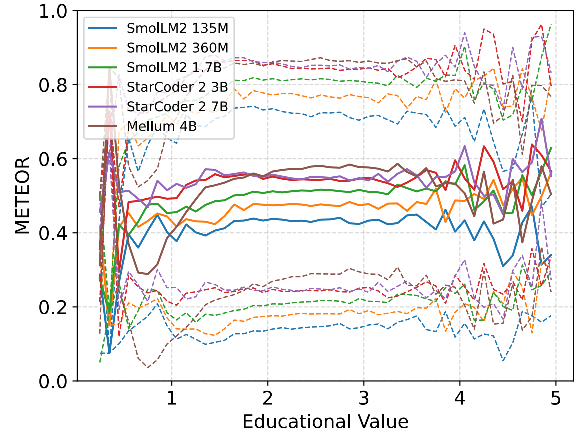


Python (next-20-token)

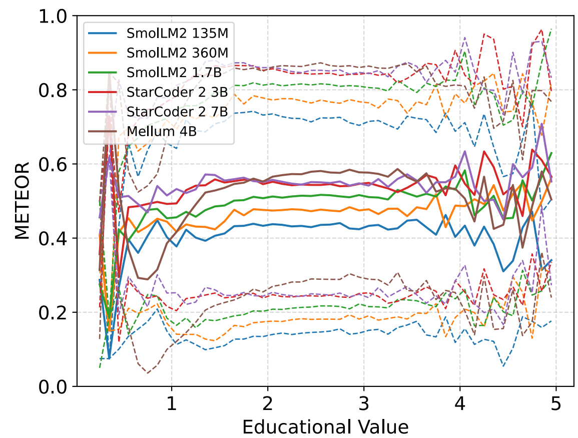


Java (next-20-token)

Figure 9: BLEU scores (not available for j100k).

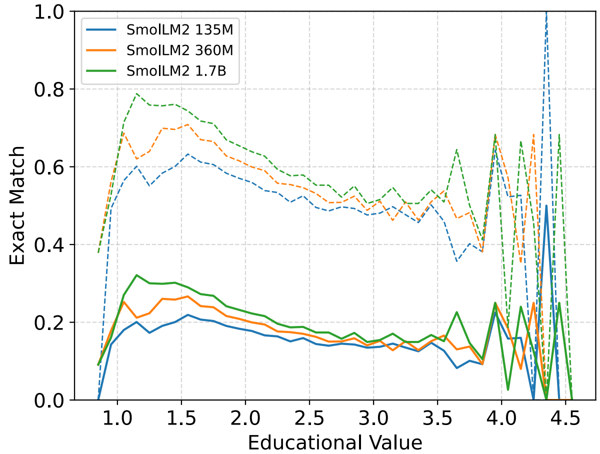


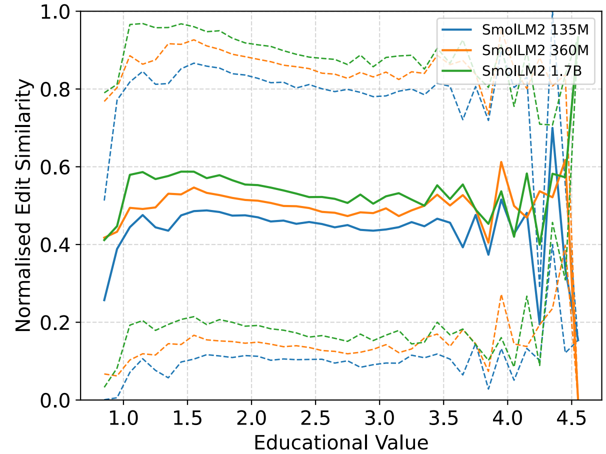
Python (next-20-token)



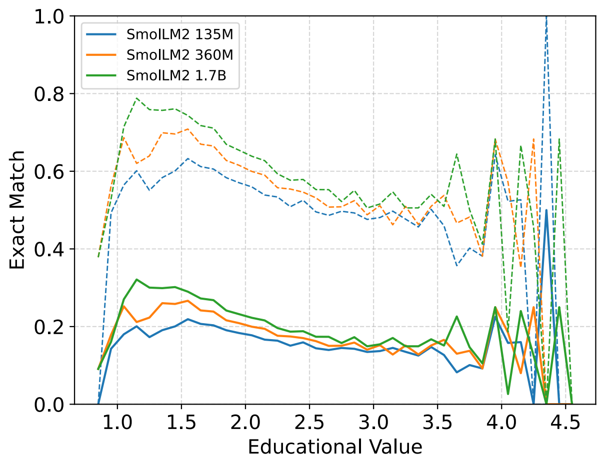
Java (next-20-token)

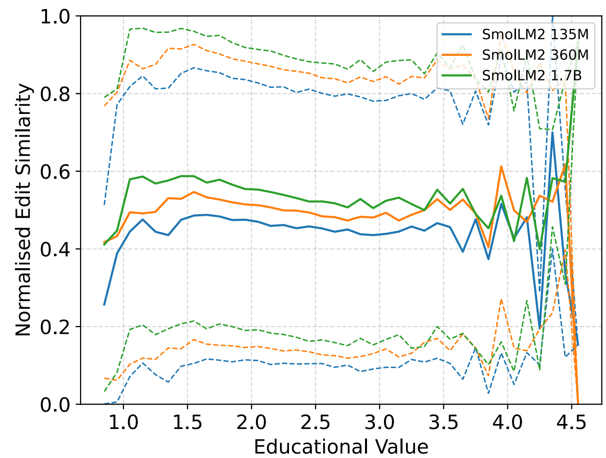
Figure 10: METEOR scores (not available for j100k).



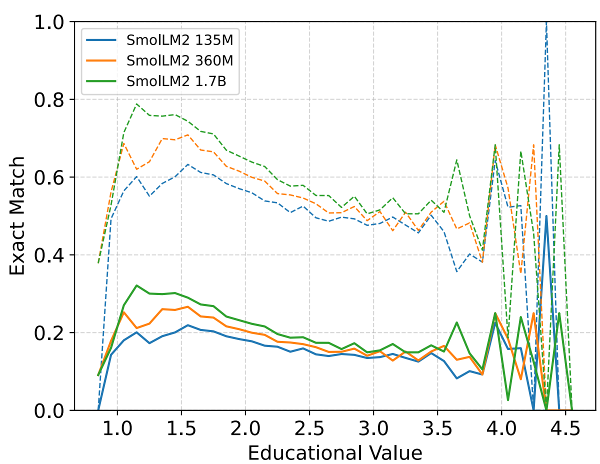


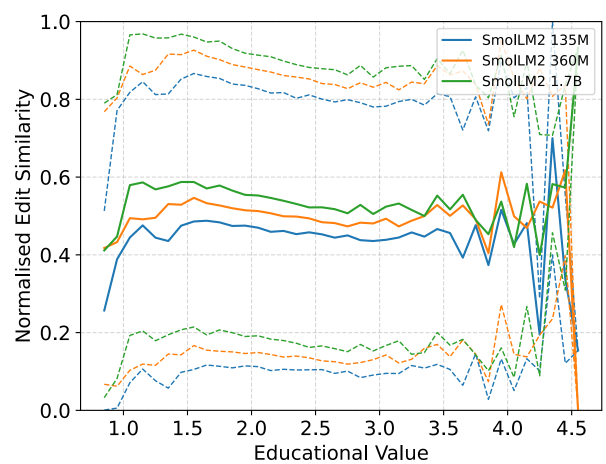
Java (next-line) **Java (next-line)**





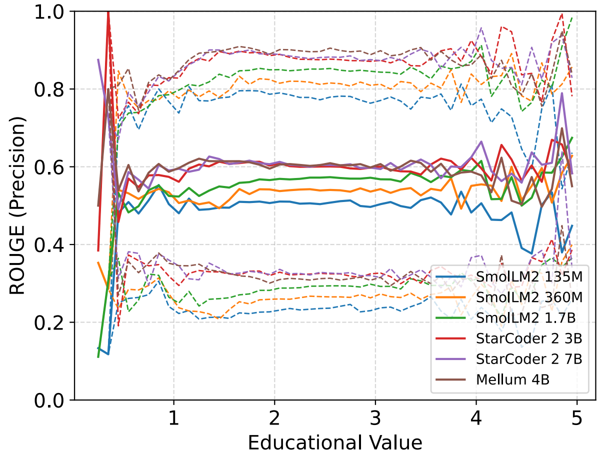
Python (next-20-token) **Python (next-20-token)**



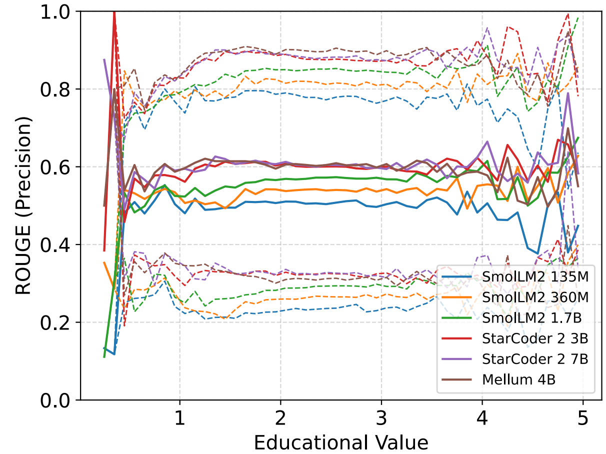


Java (next-20-token) **Java (next-20-token)**

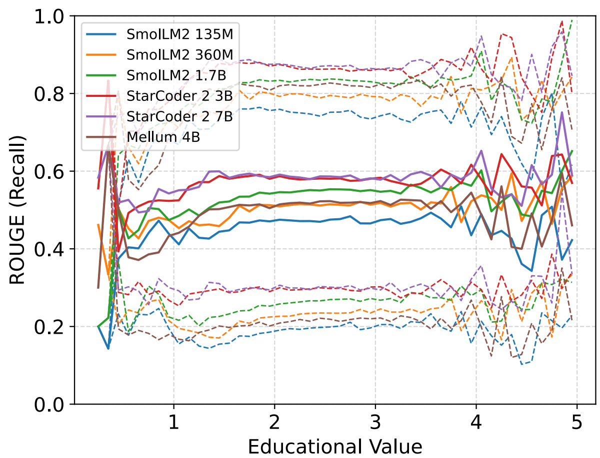
Figure 11: Exact Match and Normalised Edit Similarity across all settings.



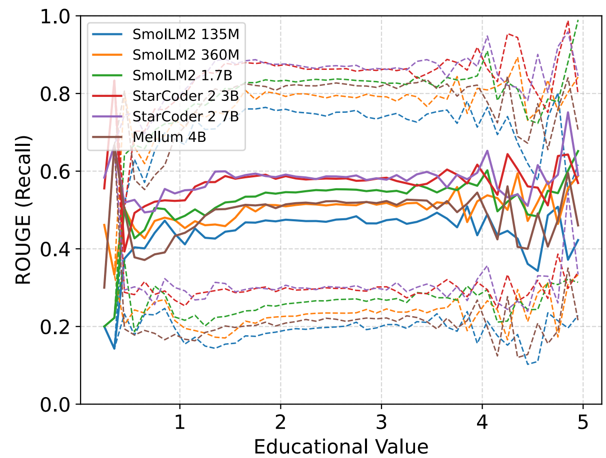
Python (next-20-token)



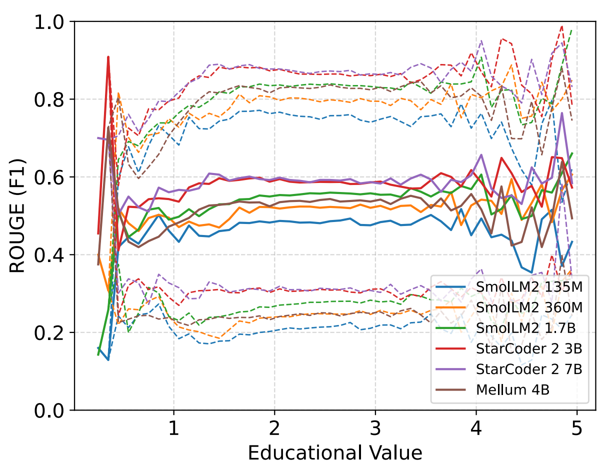
Java (next-20-token)



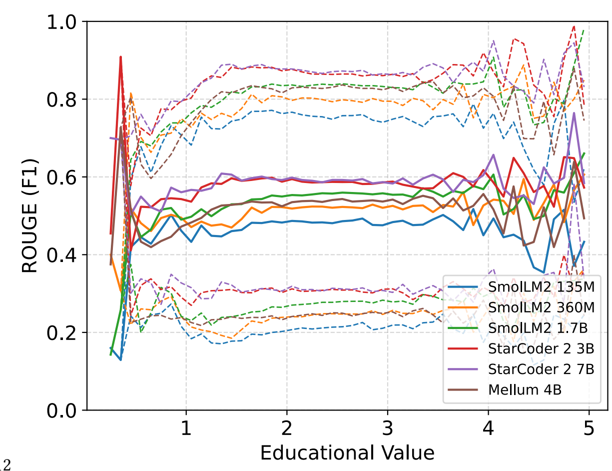
Python (next-20-token)



Java (next-20-token)



Python (next-20-token)



Java (next-20-token)

Figure 12: ROUGE-L scores (Precision, Recall, F1) for Python and Java (next-20-token).

C Example Use of Generative AI

The following examples show the use of AI in the project. It was mainly used to save time.

how exactly does rouge-l compute scores? include both recall, precision and f1

"ROUGE-L is based on the **Longest Common Subsequence (LCS)** between reference and candidate texts.

Let:

* X: reference * Y: candidate * $LCS(X, Y)$: length of longest common subsequence

Then:

* **Recall**:

$$R_{LCS} = \frac{LCS(X, Y)}{\text{length}(X)}$$

* **Precision**:

$$P_{LCS} = \frac{LCS(X, Y)}{\text{length}(Y)}$$

* **F1 Score**:

$$F_{LCS} = \frac{(1 + \beta^2) \cdot R_{LCS} \cdot P_{LCS}}{R_{LCS} + \beta^2 \cdot P_{LCS}}$$

Usually $\beta = 1$, making it the harmonic mean."

can you modify this plot by adding the std to it?

```
def update_plot(*args):
    with out:
        clear_output(wait=True)
        plt.figure(figsize=(8,5))
        for y_name in [f'{mo}-{me}' for mo in
         model_checklist.value for me in
         metric_checklist.value]:
            x, y = bin(ds['eduv'], ds[y_name],
             statistic='mean')
            plt.plot(x, y, label=y_name)
        labels('eduv', 'score')
        plt.ylim((0,1))
        plt.legend()
        plt.show()
```

"Add ‘std‘ like this:

```
def update_plot(*args):
with out:
    clear_output(wait=True)
    plt.figure(figsize=(8,5))
    for y_name in [f'{mo}-{me}' for mo in
     model_checklist.value for me in
     metric_checklist.value]:
        x, y_mean = bin(ds['eduv'], ds[y_name],
         statistic='mean')
        _, y_std = bin(ds['eduv'], ds[y_name],
         statistic='std')
        plt.plot(x, y_mean, label=y_name)
        plt.fill_between(x, y_mean - y_std, y_mean +
         y_std, alpha=0.2)
    labels('eduv', 'score')
    plt.ylim((0,1))
    plt.legend()
    plt.show()
```

can you modify it to instead make a dashed line marking +- 1 std

"Sure, replace ‘fill_between‘ with two dashed ‘plot‘ lines for ±1 std:

```
def update_plot(*args):
    with out:
        clear_output(wait=True)
        plt.figure(figsize=(8,5))
        for y_name in [f'{mo}-{me}' for mo in
         model_checklist.value for me in
         metric_checklist.value]:
            x, y_mean = bin(ds['eduv'], ds[y_name],
             statistic='mean')
            _, y_std = bin(ds['eduv'], ds[y_name],
             statistic='std')
            line, = plt.plot(x, y_mean, label=y_name)
            plt.plot(x, (y_mean + y_std),
             linestyle='--', linewidth=1,
             color=line.get_color())
            plt.plot(x, (y_mean - y_std),
             linestyle='--', linewidth=1,
             color=line.get_color())
        labels('eduv', 'score')
        plt.ylim((0,1))
        plt.legend()
        plt.show()
```