Data hound: Analysing non-English data smells in large code datasets

Bogdan-Mihai Buzatu[1]

Supervisor(s): Prof. Dr. Arie van Deursen[1]**, Assistant Prof. Dr. Maliheh Izadi**[1]**, ir. Jonathan Katzy**[1] **and ir. Razvan-Mihai Popescu**[1]

[1]**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: Bogdan-Mihai Buzatu
Final project course: CSE3000 Research Project
Thesis committee: Prof. Dr. Arie van Deursen, Assistant Prof. Dr. Maliheh Izadi, Associate Prof. Dr. Avishek Anand, ir. Jonathan Katzy and ir. Razvan-Mihai Popescu,

Data hound: Analysing non-English data smells in large code datasets

Bogdan-Mihai Buzatu

Delft University of Technology
Delft, the Netherlands

Abstract

Large Language Models (LLMs) are increasingly used for code-centric tasks. However, their training data often exhibits data smells that may hinder downstream quality. This research focuses on the "Uneven Natural Languages" smell and the presence of non-English text in source code and investigates its effect on LLM-based code generation and summarisation. We construct a three-stage (Detection, Generation, Evaluation) pipeline that annotates every character in a file with its predicted language using Tree-sitter, FastText, and pycld2; masks target spans via causal masking and Fill-in-the-Middle (FIM) and prompts using three chosen models (SmolLM2, StarCoder 2, and Mellum-4B). The Heap dataset is used for the pipeline; however, this research only focuses on the Java subset of the Heap.

In 3.35 million Java files, we find that English tokens account for more than 90% of comments, strings, and identifiers, while Chinese, Spanish, Portuguese, and French form a long-tailed minority. Despite this skew, LLMs achieve marginally higher BLEU, METEOR, ROUGE, and Exact Match scores when non-English elements are present or masked. Mellum consistently yields the most fluent continuations; StarCoder 2 retains broader token recall; SmolLM2 lags on both axes, reflecting its smaller capacity.

Our publicly available code enables reproducible assessment of multilingual data smells and lays the groundwork for cleaner, language-aware pre-training corpora and more robust multilingual code assistants.

Keywords

Data Smell, Multilingual, Code Generation, Code Summarisation, Large Language Models

1 Introduction

Large Language Models (LLMs) have been increasingly utilised in various tasks beyond natural language generation. One of the primary use cases of LLMs is code generation and summarisation for different programming languages. Given that LLMs are trained in open-source code repositories for code generation purposes, data smells [1] may introduce challenges that affect future reasoning capabilities of the model and overall model quality.

In this research, we examine the presence of non-English languages [2] within the source code, considering that LLMs are typically pre-trained on English data [3, 4]. The "Uneven Natural Languages" data smell, which refers to the inclusion of non-English text in code, has been identified and studied, along with strategies to remove it, while being tested alongside other data smells [2]. However, there is a research gap in evaluating the effect of "Uneven Natural Languages" data smell at inference time. Building on this identified gap, we aim to understand the impact of non-English code elements on code generation and summarisation. In particular, we would like to focus on the following:

- **RQ1** What languages are commonly used in the code in addition to English?
- **RQ2** What is the distribution of English and non-English elements across the Heap?
- **RQ3** What is the effect on Large Language Models' code generation when non-English code elements are present in the prompt dataset?
- **RQ4** How is Large Language Models' code summarisation influenced when non-English code elements are masked in the prompt dataset?

To address the identified challenges, we design a pipeline that detects, masks, and then prompts the masked code to an LLM. This process is applied to a dataset, The Heap [5], which is completely decontaminated from the training data of SmolLM2 [6], StarCoder2 [7] and Mellum [8] to ensure the reliability of our results. For each code sample, we identify the smell in non-English language using Abstract Syntax Trees generated by Tree-sitter [9] and language detection tools such as FastText [10, 11]. Once the smell is detected, it is masked using Casual Masking and Fill-in-the-middle [12]. The modified code is passed to SmolLM2 [6], StarCoder2 [7] and Mellum [8] to complete the code or summarise it. The resulting completion is then evaluated using metrics such as BLEU [13] and exact match.

Based on the developed pipeline, we find that English tokens account for over 90 % of all comments, string literals and identifiers present in the 3.35 Milion files Java subset of the Heap. Moreover, after sampling between 1000 and 2000 files from both English and non-English datasets and performing code generation and summarisation tasks, we find that the models perform slightly better on non-English data.

With this work, we contribute in the following ways:

(1) We develop a three-stage pipeline consisting of Detection, Generation, and Evaluation through which the "Uneven Natural Languages" data smell can be evaluated in tasks like code generation and summarisation using metrics like BLEU and exact match.
(2) Our Detection stage produces character-level annotations, stating the language of each specific character for each character present in the content of the data point. These annotations can be leveraged for further evaluation and deeper analysis of the effects of non-English data smell on the Heap dataset.
(3) The repository used for the development of the pipeline is publicly available to reproduce the research and the results. The repository contains the code used for the pipeline, along

with samples of the dataset and the results from inferring those samples. [1]

2 Background Research

Data smells [1] are identified as data quality issues that may lead to future problems for LLMs as they are present inside the training data of the models. They usually appear as a result of avoiding using recommended best practices, their suspicion not being dependent on a specific context. Based on a defined taxonomy [2], there is a class for smells related to the language used inside the code. The 'Uneven Natural Language' sub-category is associated with the presence of different languages inside the code other than the ones that the model has been usually pre-trained on. Since English is the most used language for the training of the LLMs, the 'Uneven Natural Language' data smell is related to the 'Long-Tailed Distribution' data smell category, which highlights the higher representation of a particular class compared to other classes inside the dataset.

Currently, most research on this data smell focuses on detecting it using various methods. Among others, we can highlight the removal of the instances where English is not present, or the probability of being present is low [1, 14] using existing libraries for language detection or by removal of the instances where non-ASCII characters [15]. In the case of FineWeb [14], filtering out non-English content is done to improve downstream performance. However, there is no significant research done to address the influence of 'Uneven Natural Language' data smell at inference time.

In the following subsection, we will go through detection methods of non-English language, FIM used for masking and multiple ways of evaluating the result (such as Exact Match, BLEU Score and Human Evaluation)

2.1 Detection

There were a few highlighted methods for identifying the language of a given text to solve the problem of Uneven Natural Language inside the code block, based on the Systematic Literature Review conducted [2] (langid, cld3, and fasttext). In the following bullet points, we are going to go through multiple methods of detecting non-English language that can be used to detect the data smell inside a code block:

(1) Dictionary-Based Lookup [16] is a lexicon-based approach in which a wordlist is stored for each language. Based on the tokens from the input text, each token is checked to see which dictionary is part. The dictionary that has the most tokens inside the input text is considered the language of that text. It is an efficient and transparent approach as it is fast, and there is no need for training or an external model. However, there are cases where words can be part of multiple languages, and this approach would fail in this case.
(2) Langid (Character N-Gram + Naive Bayes) [17, 18] is a model that converts text into character n-grams (e.g., trigrams: fun, unc, nct, etc.)and then the input is applied to a Naive Bayes classifier trained to distinguish languages by their n-gram frequencies. This model can work well with variable names inside the code (e.g., isOpened, closedTickets), as the tokenisation is independent (cite the survey). In addition to Being Tokenisation-Independent, the model is fully self-contained with no external dependencies, and it provides deterministic results.
(3) pycld2 and pycld3 (Google's Compact Language Detector 2 and 3) [19] use a neural model trained on massive multilingual web data (CLD3 made by Google), through which it computes character-level embeddings and produces top k languages with probabilities. It can be very accurate on short, web-like text (e.g., comments and short literals), and it supports many languages. However, it is a closed model with biases coming from the training data (the model might favour more web-dominant languages).
(4) FastText (Facebook Short Neural Model) [10, 11, 20] is a Facebook AI-developed shallow neural network, initially developed for text classification. The language identification model is a specific application of this architecture. The model used for language identification is trained based on Common Crawl & Wikipedia for 170+ languages. When receiving an input string, it is extracted into n-grams, which are further converted into a vector. Furthermore, the input is embedded by averaging the vector and then passes through a linear layer where the language of the input string is predicted. It achieves good accuracy and efficiency for most input texts, providing reliable confidence scores for the resulting languages.

2.2 Masking

For the masking of smells, we are going to use the 'Fill-in-the-middle' (FIM) paradigm. 'Fill-in-the-middle' [12] is a paradigm used to generate the middle part of a text based on the given prefix and suffix for the LLMs. When using the FIM paradigm, we encode both the prefix and suffix, and we provide the prompt for the Large Language Model as: <PRE> Enc(prefix) <SUF> Enc(suffix) <MID>. Where <PRE>, <SUF> and <MID> are sentinel tokens used to mark the boundaries between the given prefix and suffix. The generated response is prompted until the <EOT> token is generated, which signals that the middle part has been generated successfully.

The paradigm can be used without affecting the model's ability to generate tokens from left to right. This property is called "FIM-for-free" as defined by Bavarian et al. [12], where there is no actual loss on standard task (including code completion) by adding this new capability. Moreover, FIM can work at both the document level (where the transformation occurs after fragmentation) and at the context level (where the transformation takes place before tokenisation and fragmentation), which can impact the generation of the middle depending on the size of the input and the splitting of both the prefix and suffix.

Using FIM, language models can generate the missing/masked part of the code at a specific location within the code while conditioning on both a prefix and a suffix. This is particularly useful for tasks such as code completion, generation of docstrings or infill generation, where edits are made mid-function or mid-comment.

[1] https://github.com/bogdanbuzatu04/Research-Project

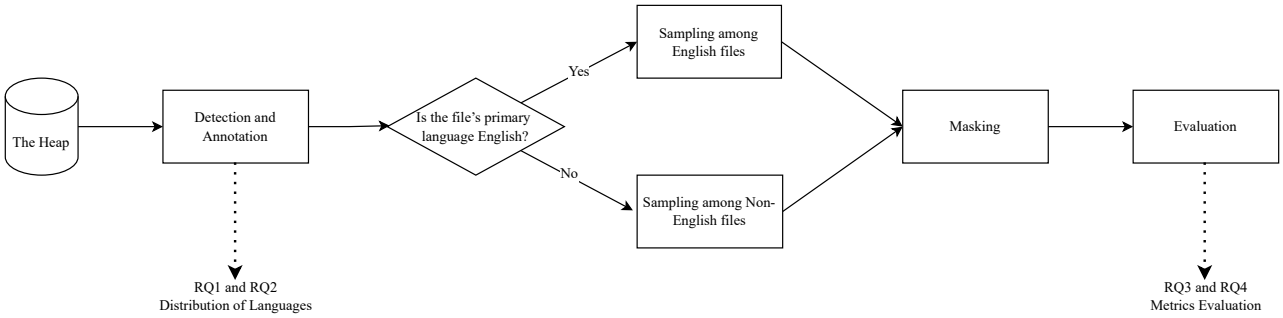

Figure 1: Pipeline of this research

2.3 Evaluation of the result

Taking into account that the focus of this research is to determine the effect of masking certain data smells identified in the code, the evaluation of the result focuses on either the code generated or the docstring based on the provided prefix and suffix. In the following bullet points, we are going to present different evaluation methods that have been used in the past for evaluating the result of a given prompt to a Large Language Model:

(1) BLEU (Bilingual Evaluation Understudy)[13] measures n-gram overlap (typically up to 4-grams) between a generated string and one or more references. It has been widely used in the field of natural language processing tasks, offering fast performance and ease of computation.
(2) METEOR Metric for Evaluation of Translation with Explicit ORdering) [21] adds up on the n-gram overlap measured by the BLEU score by taking into account also fragmentation, stemming, and synonymy. In this way, it captures recall and partial, and it offers a better sentence-level correlation with human judgments than BLEU.
(3) ROUGE (Recall-Oriented Understudy for Gisting Evaluation) [22] consists of a suite of metrics built for summarisation and can be used for LLM-generated summarisation evaluation. The ROUGE-1 score is calculated based on the number of overlapping unigrams divided by the number of total reference unigrams. In contrast, ROUGE-2 is calculated taking into account the bigrams (sequences of two consecutive words). In addition to the overlapping n-grams, ROUGE-L is calculated based on the largest common subsequence between the result and the reference, the subsequence not being consecutive but still in order.
(4) The Levenshtein distance measures the number of single-character edits needed to transform one string into another. It is often normalised by $1 - \left(\frac{\text{distance}}{\max(\text{len(ref)},\text{len(pred)})}\right)$. It is simple to interpret and useful when the exact surface form matters (e.g., identifier names).
(5) Exact Match compares the predicted and the reference string, and then one is returned in case the two strings are the same. Taking into account its structure, it is clear and strict, while it is also appropriate for deterministic completions (e.g., variable names, short docstrings).

3 Methodology

To conduct our experiments, we use *The Heap* dataset [5], which contains code in multiple programming languages and is suitable for evaluating LLMs. *The Heap* is contamination-free with respect to several public pre-training datasets, such as *The Stack* v1 [23] and *The Stack* v2 [7], both used for the training of SmolLM2 [6], StarCoder2 [7] and Mellum [8]. Therefore, the dataset is suitable for a fair evaluation of the chosen LLMs on unseen code. In our experiments, we prompt chosen LLMs with code samples in which we mask the English and non-English targets, allowing us to determine the influence of *Uneven Natural Languages* data smell on the models' performance. In general, the methodology of this research follows three essential parts: *Detection and annotation*, *Masking* and *Evaluation*

3.1 Detection and annotation

We start the detection by using Tree-sitter [9, 24] to produce concrete Abstract Syntax Trees that preserve all code details. By querying inside the generated AST, we look into specific parts of code like string literals, comments, variables, and class and method names where non-English content typically appears. The detection of the used language is addressed in two different ways, depending on the part of the code being tackled.

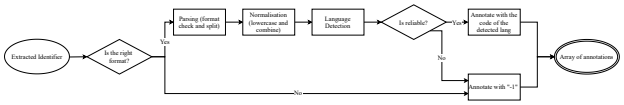

Figure 2: Language Detection for Identifiers

Identifiers. Based on good practices, identifiers for methods and variables should adhere to the following formats: Camel Case (blueRedYellow), Snake Case (blue_red_yellow) and Kebab Case (blue-red-yellow). The identifiers can be easily parsed and split into multiple parts (e.g. blue, red, yellow) using a regular expression, where each part is lowercase and then combined into a string with spaces in between elements. The resulting string is then passed to `detect()` `pycld2` and `FastText` [10, 11] for secondary check. In case the result is reliable, each of the characters inside the identifiers is annotated with the code of the resulting language or `"-1"` otherwise.

Figure 3: Language Detection for Comments and Strings

String literals, comments, and docstrings. Since comments and docstrings may contain annotations and tags specific to programming languages, the input text is first cleaned using regular expressions to remove the particular annotations and tags. With the cleaned text, sliding windows are used to determine the language for smaller parts of the text using `FastText` [10, 11]. The result of the sliding window is the list of the top 3 languages and their confidence scores. For each character, the detection algorithm computes the weighted sum of confidence scores of the languages for the windows where that character is present. In case the language with the highest confidence score is bigger than the threshold, assign that language to the character. Otherwise, perform a fallback using `pycld2` where the character is in the centre of the input window.

After detecting the language, we annotate the data smell to have a general overview of the existence of such data in the format of an array which contains the language code or `"-1"`. By default, the other parts of the code that are not tackled by the detection algorithm have their annotations set to `"-1"`. Based on the annotations, we can determine the share of natural languages across the dataset, directly answering *RQ1* and *RQ2*.

3.2 Masking

To evaluate the influence of *Uneven Natural Languages* data smell in completing code and generating comments given partial context, we apply two masking strategies: FIM masking [12] and causal masking. In both cases, the unmasked context (either containing both prefix and suffix or only the prefix) is drawn from the original reference code, and the model is tasked with filling in the masked region.

Comments and docstrings. Since comments and docstrings describe the implementation written in the code, we would like to determine the influence of the data smell on code generation by masking the first line after the comments or the docstring. The other case is to mask the entire docstring or block comment to evaluate the summarisation of the code.

After annotating each file for language content, we split our dataset into English-only and non-English subsets. From each subgroup, we draw random samples for our experiments. We then apply the same masking procedures to all files, regardless of whether they exhibit the *Uneven Natural Languages* data smell so that we can directly compare models' performance on clean versus contaminated code. For each file in the sample, we make generations for every comment (for FIM, we use block comments and docstrings, while for casual masking all the comments).

3.3 Evaluation

Based on the result of the inference, we compute Exact Match, BLEU [13], METEOR [21] and ROUGE [22] to compare the similarity of the result against the ground truth target. We use these metrics to determine how *Uneven Natural Languages* data smell influences the code summarization and generation, directly answering *RQ3* and *RQ4*.

Number of files where the language is solely present		
Language	ISO-639-1 Code	Files
English	en	1907629
Chinese	zh	2000
Spanish	es	1355
Portuguese	pt	1005
French	fr	771
Italian	it	454
German	de	433
Czech	cs	189
Japanese	ja	183
Polish	pl	177
Russian	ru	173
Danish	da	170
Korean	ko	150
Dutch	nl	143
Turkish	tr	114

Table 1: Natural languages used across the Heap

4 Results

In this section, we report findings for RQ1–RQ4. RQ1: among 3.35 million Java files, 57 % use only English; the rest mix languages or use a single non-English tongue—chiefly Chinese, Spanish, Portuguese and French (Table 1). RQ2: across comments, strings and identifiers, English accounts for over 90 %, with a minor tail of Romance and Germanic languages. RQ3: on 8 457 English versus 1 911 non-English next-line generations, BLEU and ROUGE scores are near zero but slightly higher for non-English; Mellum and StarCoder 2 outperform SmolLM2 in both recall and fluency. RQ4: masking non-English terms boosts Mellum's BLEU at the expense of coverage, while StarCoder 2 sustains broader token recall (ROUGE-1/ROUGE-L); both better reconstruct non-English comments than English ones.

4.1 RQ1: What languages are commonly used in the code in addition to English?

Based on a subset of 3 350 000 Java files, 1 907 629 (about 57 %), as shown in Table 1, contain only English. The remaining files either mix multiple languages or use a different single language. More specifically, other languages present solely in the files include Chinese, Spanish, Portuguese, and French, which occur in more than 500 files each.

4.2 RQ2: What is the distribution of English and non-English elements across the Heap?

Figures 6, 7 and 8 all confirm that English utterly dominates code-adjacent language use, with token counts an order of magnitude greater than any other language—hence the logarithmic 'y' axes. In comments (Figure 6), English accounts for approximately 92.4% of all monolingual entries, dwarfing Chinese (≈ 3.0%), Galician (≈

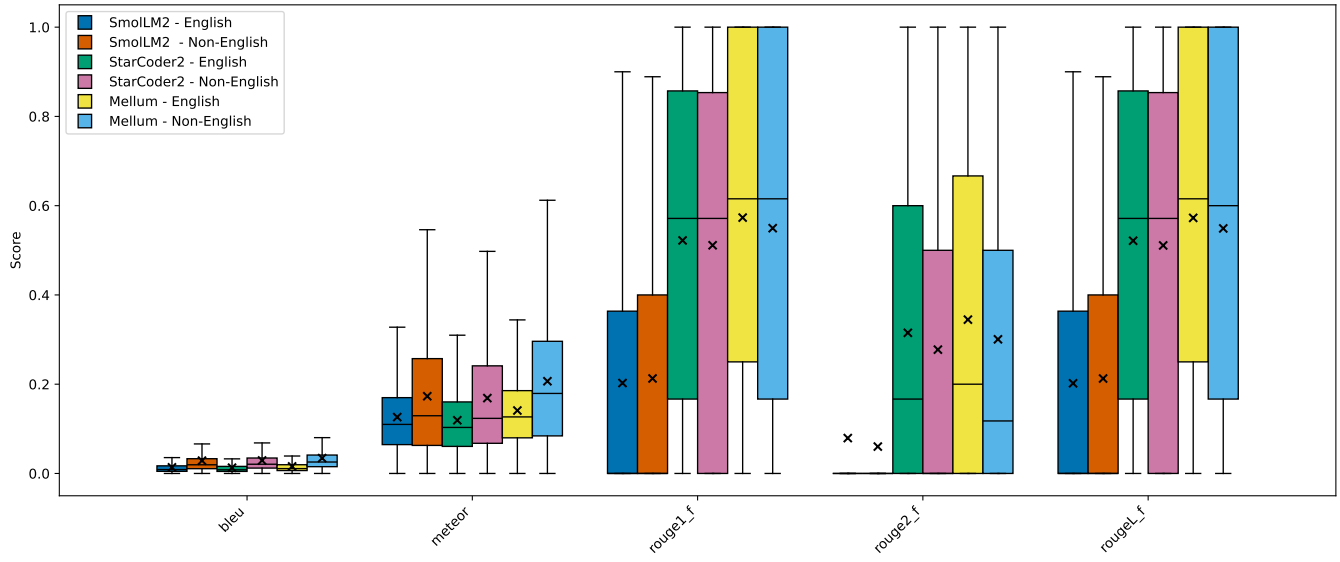

Figure 4: Metrics evaluation for Next Line Generation

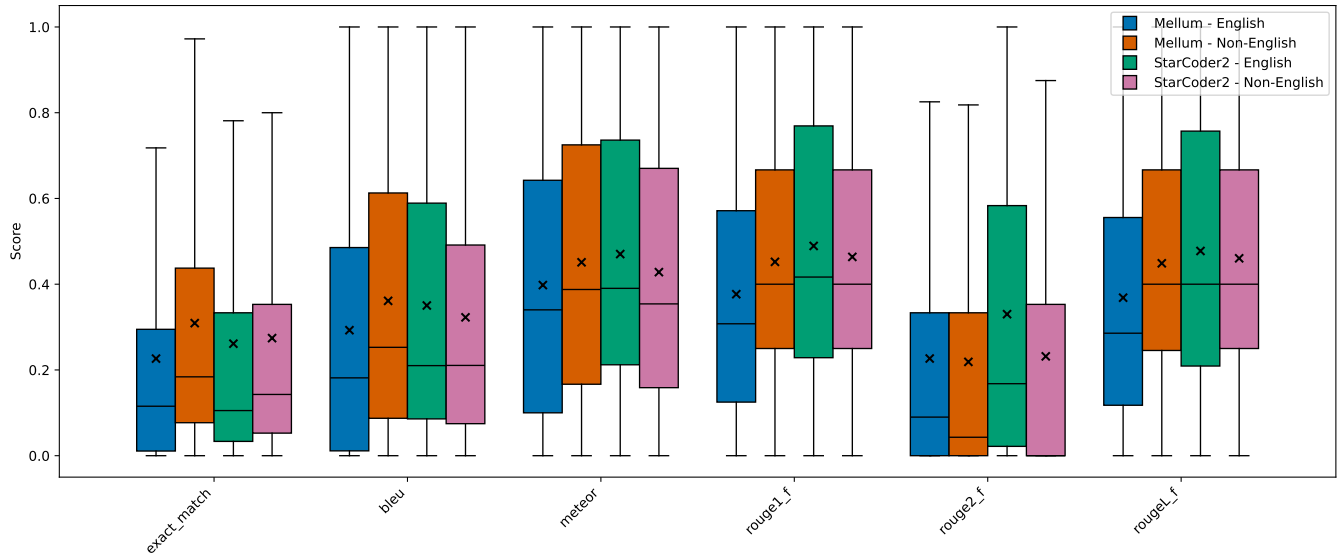

Figure 5: Metrics evaluation for Code Summarisation

1.1%), French (≈ 0.6%), German (≈ 0.6%), Spanish (≈ 0.6%), Japanese (≈ 0.5%), Portuguese (≈ 0.4%), Italian (≈ 0.4%) and Russian (≈ 0.3%). Identifier names (Figure 8) show an even stronger skew: English comprises about 97.8% of tokens, while the next-most-frequent tongues—Spanish (≈ 0.49%), Italian (≈ 0.44%), French (≈ 0.42%), German (≈ 0.26%), Portuguese (≈ 0.23%), Dutch (≈ 0.14%), Czech (≈ 0.10%), Swedish (≈ 0.10%) and Catalan (≈ 0.09%)—barely register by comparison. String literals (Figure 7) similarly feature English at roughly 92.4%, versus Chinese (≈ 1.3%), German (≈ 1.1%), Danish (≈ 1.1%), French (≈ 0.9%), Portuguese (≈ 0.7%), Italian (≈ 0.7%), Spanish (≈ 0.7%), Latin (≈ 0.6%) and Dutch (≈ 0.5%).

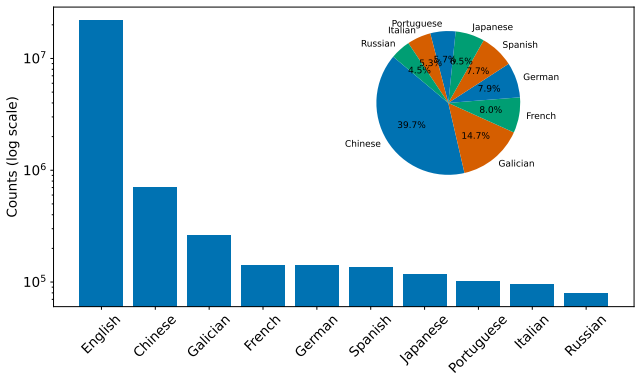

Figure 6: Comments containing a single language

Beyond English, most of the nine next-most-common languages in each category can be clustered into two principal families, excluding Chinese and Japanese. The six Romance languages (French, Italian, Spanish, Portuguese, Galician and Catalan) collectively contribute to a significant amount of non-English tokens, reflecting a shared lexicon and broad adoption among developer communities in Southern Europe and Latin America. Meanwhile, four Germanic languages—Danish, Swedish, Dutch, and German—also feature prominently, underscoring the influence of Northern European languages in open-source code. These patterns suggest that, although English remains the unchallenged lingua franca across the Heap dataset, secondary languages used in comments, identifiers, and strings tend to mirror broader familial affinities, with developers favouring terminology drawn from their native or closely related languages.

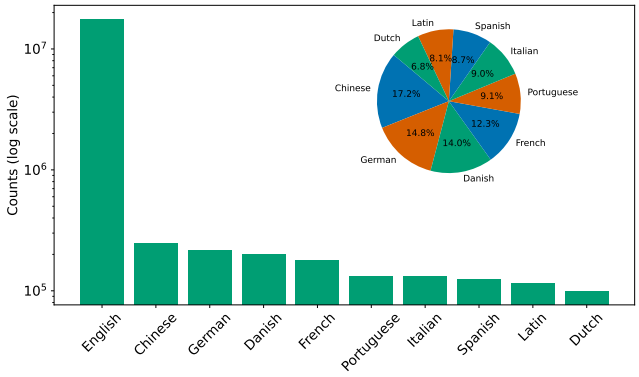

Figure 7: Strings containing a single language

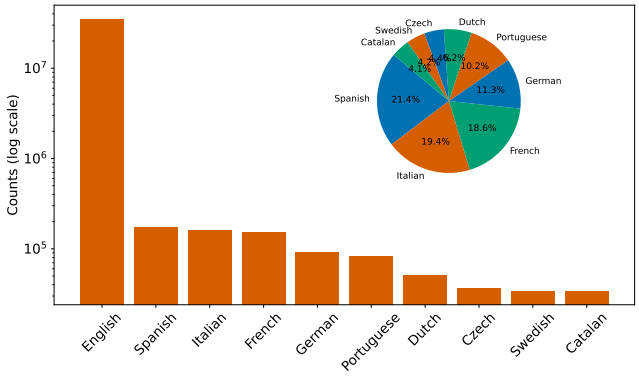

Figure 8: Identifiers containing a single language

Sample	Number of files
Containing only English language	1 913 782
Containing only English language with at least one comment	1 362 819
Randomly selected from the English subset for Next Line generation	2 000
Randomly selected from the English subset for FIM generation	1 000
Containing a singular non-English language	8 897
Containing a singular non-English language with at least one comment	3 928
Randomly selected from the non-English subset for Next Line generation	1 000
Randomly selected from the non-English subset for FIM generation	1 000

Table 2: Number of files used for sampling

4.3 RQ3: What is the effect on Large Language Models' code generation when non-English code elements are present in the prompt dataset?

To evaluate the effect on code summarisation, we randomly take 1000 file samples from the non-English subset and 2000 file samples from the English subset. Figure 2 shows a detailed view of the sampling procedure, taking into account the total number of monolingual files that use either English or non-English languages.

Based on 8457 generations for English and 1911 generations for non-English languages, Figure 5 shows that BLEU is essentially zero for all models in both English and non-English, indicating almost no exact n-gram overlap in next-line predictions. There are also zero values for the exact-match, which have been removed from the chart, but they are visible in the Appendix of the paper. Looking at ROUGE-1, Mellum seems to have the highest recall, while StarCoder 2 has a solid recall, and SmolLM2 has the lowest overlap. When it comes to fluency, the order is maintained, with Mellum producing the most coherent bigram sequences, StarCoder 2 offering moderate continuity, and SmolLM2 being last.

Moreover, the means (marked by "×" in Figure 5) closely track the medians for every model and language, indicating that extreme outliers do not drive these central tendencies. SmolLM2's IQR is very tight—its next-line performance is uniformly low—. In contrast, Mellum and StarCoder 2 show wider IQRs and longer whiskers, signalling greater variability (sometimes quite strong, sometimes weak) across examples. Occasional high-scoring outliers, particularly for Mellum on non-English ROUGE-2 and ROUGE-L, underscore that a subset of lines can be reconstructed extremely well even when the median remains modest.

In general, non-English following lines are easier for every model and metric, with consistently higher scores on non-EN than English, reflecting the models' slightly stronger handling of shorter or more formulaic comment patterns in those languages. Moreover, the order of performance between the models is maintained for both English and non-English. The lower scores of SmolLM2 can be

justified by the much lower number of parameters (135 M) [6] compared to Mellum [8] and StarCoder 2 [7], which limits its capacity both to recall tokens and to generate fluent continuations.

4.4 RQ4: How is Large Language Models' code summarisation influenced when non-English code elements are masked in the prompt dataset?

To evaluate the effect on code summarisation, we randomly take 1000 file samples from the non-English subset and 1000 file samples from the English subset.

Based on 2531 generations for English and 1222 generations for non-English languages, Figure 5 shows that neither English nor non-English comments are recreated verbatim, with exact-match hovering around 10–15 % in every condition. Mellum sacrifices some token recall (ROUGE-1) to boost local cohesion (BLEU), yielding smoother two-gram continuity at the expense of overall coverage. Additionally, Mellum seems to perform a more fluent phrasing, ignoring completeness: its median BLEU and Meteor scores both exceed those of StarCoder 2 by several percentage points, especially on non-English samples. On the other hand, StarCoder 2 has a slightly broader recall, given its marginally higher ROUGE-1 score and superior ROUGE-L performance, which indicate it retains a larger fraction of the original comment's vocabulary and ordering. Looking at Meteor, Mellum's candidates are "closer" in word order and content on average, particularly for non-English, where its median Meteor rises by nearly 0.10 compared to English.

Here again, the close alignment of means and medians confirms that these differences are robust rather than skewed by outliers. The interquartile ranges for Mellum and StarCoder 2 are wide, highlighting that masking induces a variety of outcomes—some comments are very faithfully reconstructed, others much less so. Rare but notable high-end outliers in the non-English distributions point to specific cases where FIM recovers nearly complete comments.

When comparing the English and non-English generations, both models generally reconstruct non-English comments more accurately, achieving higher medians and narrower interquartile ranges across all metrics. However, StarCoder 2 has slightly higher scores for BLEU, METEOR, and ROUGE in English, as evidenced by its elevated mean values (depicted with "×" in Figure 5), suggesting that despite lower overall fluency, it captures a broader array of English comment tokens with greater consistency.

5 Discussion

We now discuss the main findings in more detail from the perspective of the research questions and their implications. We provide a transparent summary and comment on the limitations of the existing research in the latter part of the section and discuss future work.

5.1 Language Detection

Looking at the 2024 Octoverse report by GitHub[2], we see that the United States, United Kingdom and Canada, three predominantly English-speaking countries—rank first, fifth and tenth, respectively, among GitHub's Developer Community. Other large developer populations include China, Brazil (Portuguese), Germany and Japan. This global distribution partially mirrors our Java subset analysis of the Heap: English, as the de facto lingua franca of programming, unsurprisingly represents the vast majority of the code-adjacent text.

As shown in Figure 6, Galician appears with unexpectedly high frequency among monolingual comments, despite having only around 2.4 million native speakers in northwestern Spain[3]. A manual inspection of the annotated dataset reveals that many occurrences of "Galician" are artefacts of Eclipse's internationalisation markers, e.g. '//$NON-NLS-1$'—which the detector misclassifies as Galician rather than recognising them as tool-generated tokens[4][5]. Moreover, the maximum monolingual comment length for Galician is only 610 characters, versus 647 658 for English, 7 022 for Spanish or 5 278 for Portuguese, further evidence that these are false positives due to detector fallback behaviour and pycld2's known limitations as fallback method for the detection [6].

A parallel issue arises with strings flagged as Latin: many of these literals contain significant non-alphabetic noise (e.g. UUIDs, log-format placeholders), which again confuses the language detector. In both cases, our findings underscore a key limitation of automated language identification in source code contexts, especially when tool-specific markers or noisy tokens are present.

5.2 Code generation and summarisation

The evaluation metrics, BLEU, ROUGE and METEOR, align with human expert rankings across languages and preserve their exact ordering. When presented with noisy comments, their scores trend towards zero [25], demonstrating their ability to flag truly incoherent outputs. However, because they quantify only surface overlap, they do not account for semantic equivalence between reference and generation. In non-English settings, absolute scores drop even further (due to inflectional variation, compounding, or segmentation issues), so language-specific tokenisation and per-language threshold calibration are critical to interpret and compare results fairly.

It is worth mentioning that across the Java subset of the Heap, the maximum size of the Monolingual comment in English is 647658 characters, while for non-English, the maximum number of characters is around 90000 while taking into account that these numbers include spaces, special character and random noises. Additionally, looking at the samples selected for the generation, there is a higher number of comments on average per file for English files compared to non-English (EN: 2000 files - 8457 line, block comments or java docs, non-EN: 1000 files - 1911 line, block comments or java docs)

With these in mind, we now turn to two complementary tasks: next-line generation and code summarisation.

5.2.1 Next line generation. The near-zero BLEU and exact-match scores confirm that predicting arbitrary "next lines" in code comments is inherently challenging; even state-of-the-art models rarely reproduce the author's phrasing verbatim. Medians and means for

[2]https://github.blog/news-insights/octoverse/octoverse-2024/

[3]https://minorityrights.org/communities/galicians/

[4]https://www.eclipse.org/articles/Article-Internationalization/how2I18n.html

[5]https://stackoverflow.com/questions/654037/what-does-non-nls-1-mean

[6]https://github.com/pemistahl/lingua-py

these metrics sit at or near zero, with very narrow interquartile ranges, indicating uniformly poor overlap across examples. This suggests that practical code-completion systems should rely on fuzzy matching or semantic retrieval rather than literal regeneration.

Models exhibit a clear recall-vs-fluency trade-off: Mellum > StarCoder 2 > SmolLM2 on ROUGE-2 (bigram continuity) and the inverse ordering on ROUGE-1 (token coverage). Wider interquartile ranges and occasional high-scoring outliers for the larger models hint at cases of surprisingly accurate predictions, while SmolLM2's tight IQR reflects consistently low performance. All models perform better on non-English snippets, implying that shorter, more stereotypical patterns in those languages are easier to predict and yield higher median scores with reduced variance.

5.2.2 Code summarisation. FIM's 10–15 % exact-match rate shows that, even with contextual masks, verbatim recovery is rare. The close alignment of means and medians for exact-match and BLEU underscores that this limitation is consistent rather than driven by outliers. Yet moderate ROUGE-1 and BLEU scores indicate models still capture the local phrasing.

Non-English masked comments perform better in all metrics. However, a select few high-value outliers within the non-English distributions demonstrate a tendency to over-fit simplistic templates. While specific comments are reconstructed with remarkable accuracy, others significantly underperform relative to the median. This variability highlights the need for a more in-depth examination of language-specific patterns and the development of adaptive masking strategies.

5.3 Limitations

(1) The research has been conducted on a subset of the Heap dataset. Consequently, it is hard to have a general overview of the languages present while also lacking generalisation about the influence of non-English languages on code summarisation and generation
(2) Pycld2 is not fully reliable when detecting the Language. The library has been chosen for this research, mainly because of the lack of support of pycld3 for python versions >= 3.10 [7]
(3) The evaluation of the generations does not take into account the semantics when comparing the reference solution to the generated one. This is caused by how the chosen metrics are calculated

5.4 Future work

Based on the findings and the limitations of our current research, we define further areas of development around this research:

(1) There can be an improvement of the existing pre-processing done on comments and strings so that we can strip or normalise such artefacts, which would yield more accurate language distributions.
(2) We should change pycld2 with other libraries for language detection. This way, we can improve the fallback method used for comments and strings, as well as the general detection used for identifiers.
(3) We should run the pipeline across the entire Heap dataset, as it might reveal a slightly different distribution of languages and the influence of non-English on LLMs' code summarisation and generation.
(4) We should also tackle generations around identifiers and strings, as they are essential parts of the code that contain non-English Language.

[7] https://pypi.org/project/pycld3/

6 Conclusion

In this study, we designed and implemented a three-stage (Detection, Generation, Evaluation) pipeline to investigate the impact of the "Uneven Natural Languages" data smell on LLM-based code generation and summarisation. Our pipeline incorporates character-level language annotations for every code file in the dataset, enabling us to gain a more comprehensive understanding of the languages used in each file and across the entire dataset.

Our experiments focus on the Java portion of the Heap dataset, comprising 3.35 million files. Once annotated, we find that English tokens account for over 90 % of all comments, string literals and identifiers. At the same time, Romance languages (e.g. Spanish, French, Portuguese) and Germanic languages (e.g. German, Dutch, Danish) constitute a long-tail minority.

Based on the annotated subset, we derive four sample subsets (three of which consist of 1000 files and another of 2000 files) to use for generation. Based on the presence of data smell, we define the context and the target for Fill-in-Middle and Casual Masking while utilising StarCoder2, SmolLM2, and Mellum for the generation. In code generation and summarisation tasks based on the derived samples, English remains overwhelmingly the most pervasive, yet the presence of non-English components informs LLMs' behaviour. In particular, the non-English samples perform slightly better on specific tasks, as indicated by BLEU, METEOR, and ROUGE scores.

7 Responsible Engineering

While conducting this research, we put the accent on the integrity, transparency and reproducibility of the research by having:

Transparent and reliable research process. In the 10 weeks of this research, the progress has been continuously supervised by 2 PhD students with knowledge and expertise in the field of research. During the process, they have provided us with feedback on the work that has been done. Moreover, a responsible professor verified our progress midway through the research while fellow students provided feedback on specific parts of the research.

Open-Source Dataset. The Heap dataset [5] is a publicly available dataset and can be used by anyone to research testing the LLM's generation capabilities.

Reproducibility of the results. The repository used for the development of the pipeline is publicly available to reproduce the research and the results. The repository contains the code used for the pipeline, along with samples of the dataset and the results

from inferring those samples. [8]. This ensures that the research is reproducible.

References

[1] Harald Foidl, Michael Felderer, and Rudolf Ramler. Data smells: Categories, causes and consequences, and detection of suspicious data in ai-based systems, 2022.

[2] Antonio Vitale, Rocco Oliveto, and Simone Scalabrino. A catalog of data smells for coding tasks. *ACM Trans. Softw. Eng. Methodol.*, December 2024. Just Accepted.

[3] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] Tree-sitter Contributors. Tree-sitter Documentation: An Incremental Parsing System. https://tree-sitter.github.io/tree-sitter/, 2025. Accessed: 2025-05-06.

[10] Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Hérve Jégou, and Tomas Mikolov. Fasttext.zip: Compressing text classification models. *arXiv preprint arXiv:1612.03651*, 2016.

[11] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*, 2016.

[12] Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models to fill in the middle, 2022.

[13] 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.

[14] 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.

[15] Lin Shi, Fangwen Mu, Xiao Chen, Song Wang, Junjie Wang, Ye Yang, Ge Li, Xin Xia, and Qing Wang. Are we building on the rock? on the importance of data preprocessing for code summarization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 107–119, New York, NY, USA, 2022. Association for Computing Machinery.

[16] Tommi Jauhiainen, Marco Lui, Marcos Zampieri, Timothy Baldwin, and Krister Lindén. Automatic language identification in texts: A survey. *CoRR*, abs/1804.08186, 2018.

[17] Marco Lui. langid.py: Stand-alone Language Identification System. https://github.com/saffsd/langid.py, 2011. Accessed: 2025-05-06.

[18] Marco Lui and Timothy Baldwin. langid.py: An Off-the-shelf Language Identification Tool. In *Proceedings of the ACL 2012 System Demonstrations*, pages 25–30, Jeju Island, Korea, 2012. Association for Computational Linguistics.

[19] Benjamin Solomon. pycld3: Python Bindings for Compact Language Detector v3 (CLD3). https://github.com/bsolomon1124/pycld3, 2024. Accessed: 2025-05-06.

[20] Facebook AI Research. FastText Language Identification. https://fasttext.cc/docs/en/language-identification.html, 2024. Accessed: 2025-05-06.

[21] Alon Lavie and Abhaya Agarwal. METEOR: An automatic metric for MT evaluation with high levels of correlation with human judgments. In Chris Callison-Burch, Philipp Koehn, Cameron Shaw Fordyce, and Christof Monz, editors, *Proceedings of the Second Workshop on Statistical Machine Translation*, pages 228–231, Prague, Czech Republic, June 2007. Association for Computational Linguistics.

[22] 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.

[23] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 tb of permissively licensed source code. *Preprint*, 2022.

[24] Tree-sitter Contributors. Tree-sitter Source Code Repository: Incremental Parsing System. https://github.com/tree-sitter/tree-sitter, 2025. Accessed: 2025-05-06.

[25] Jonathan Katzy, Yongcheng Huang, Gopal-Raj Panchu, Maksym Ziemlewski, Paris Loizides, Sander Vermeulen, Arie van Deursen, and Maliheh Izadi. A qualitative investigation into llm-generated multilingual code comments and automatic evaluation metrics, 2025.

[8] https://github.com/bogdanbuzatu04/Research-Project

A Other Language charts

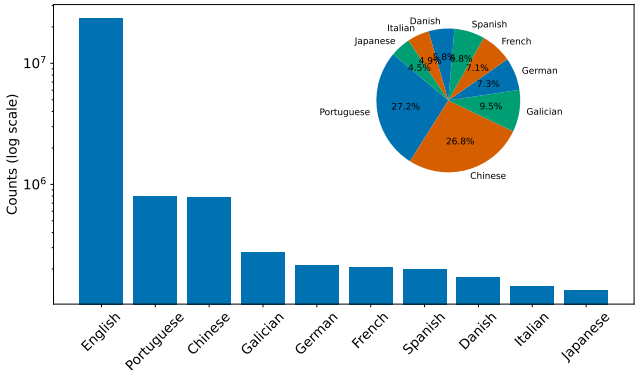

Figure 9: Distribution of languages in comments containing one or more languages

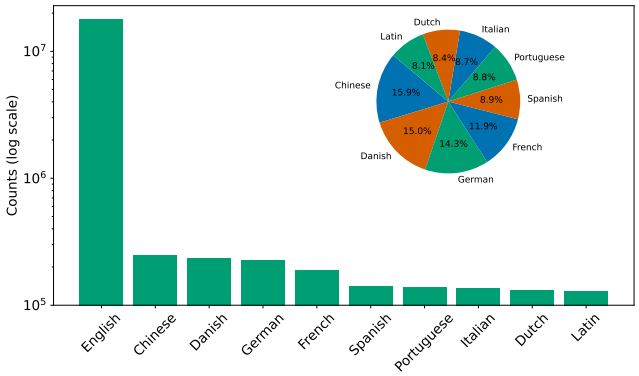

Figure 10: Distribution of languages in string literals containing one or more languages

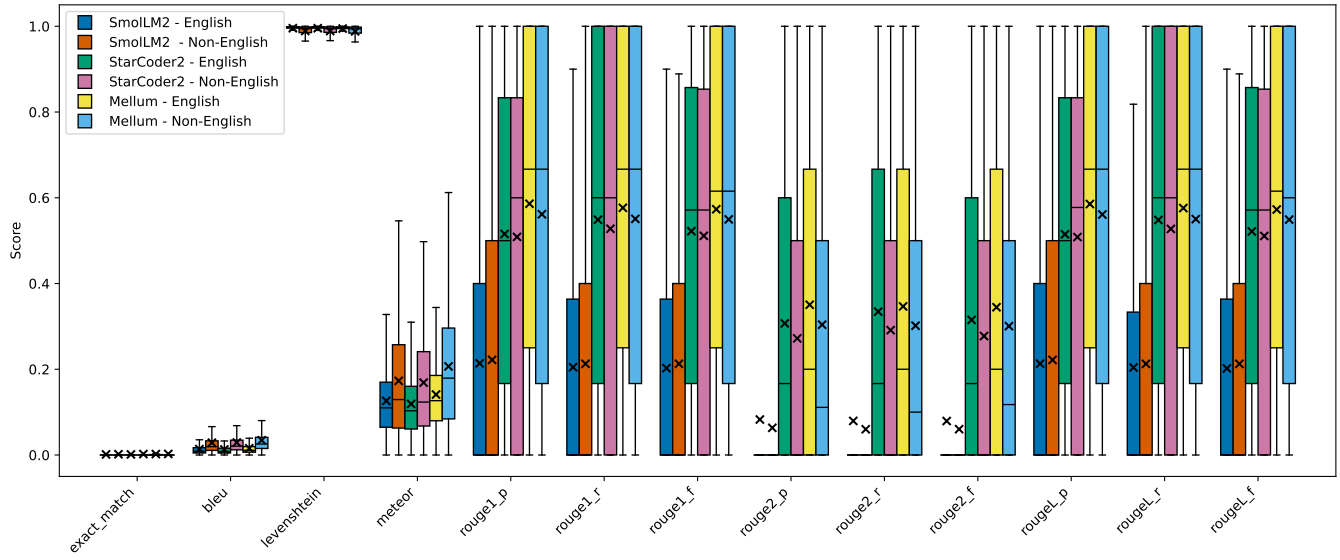

Figure 11: Metrics evaluation for Next Line Generation

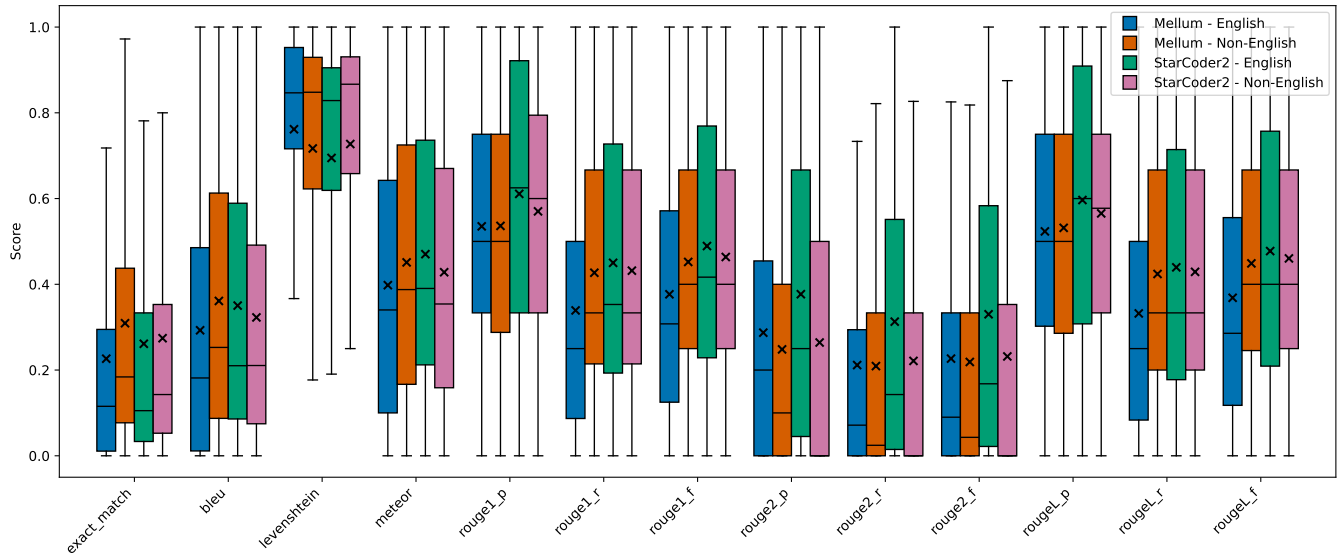

Figure 12: Metrics evaluation for Code Summarisation

B Fully-Detailed Metrics Evaluations