Data Hound: Analyzing Boilerplate Code Data Smell on Large Code Datasets

Stefan Minkov[1]
Supervisor(s): Arie van Deursen[1], Maliheh Izadi[1]
Jonathan Katzy[1], Razvan 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: Stefan Minkov
Final project course: CSE3000 Research Project

Data Hound: Analyzing Boilerplate Code Data Smell on Large Code Datasets

Stefan Minkov

Delft University of Technology
Delft, The Netherlands

Abstract

As Large Language Models become an ever more integral part of Software Engineering, often assisting developers on coding tasks, the need for an unbiased evaluation of their performance on such tasks grows [1]. Data smells [2] are reported to have an impact on a Large Language Model's ability on such tasks [3]. Boilerplate code is considered to be a subcategory of said smells. In this paper, we investigate a specific type of this smell, boilerplate API usage patterns. We analyze their prevalence in The Heap dataset [1] and examine how they may bias reference-based evaluation of Large Language Models on code generation tasks. Our findings show that while this data smell is relatively rare, instances containing it are significantly easier for LLMs to predict. We attribute this to partial memorization of common boilerplate patterns, which inflates perceived model performance.

1 Introduction

Are there biases in the evaluation of Large Language Models (LLMs) hidden just under our noses? As LLMs are increasingly becoming an integral part of Software Engineering (SE), among multiple domains, the need for a clear, unbiased explanation and evaluation of these models grows. However, their performance, as well as the performance of AI-based systems they power, is reported to be impacted from different types of data quality issues, including latent ones [2, 4]. Data Smells, a term coined to represent indications of latent data quality issues, has therefore been an active field of research in recent years, mostly centred around providing a theoretical and ontological base for the topic [2, 3]. Boilerplate code (BC), a popular term from the SE domain characterizing repetitive but necessary code fragments, is identified as a subcategory of the Irrelevant Code Data Smell [3]. Smells in this category are believed to be uninformative for an LLM and are commonly removed from the data for downstream tasks when detected [3, 5].

Nevertheless, the academic literature lacks a quantitative description of the impact of BC presence on the performance of LLMs and on what biases in their evaluation it may introduce. Our current paper directly addresses this research gap.

Our approach consists of two main components. The first, a pipeline based on the MARBLE tool [6], focuses on detecting API-usage patterns BC for eight popular Java APIs within The Heap, a dataset deduplicated against popular training datasets, which is made for evaluation purposes [1]. The second concentrates on inferencing three open-sourced LLMs for coding and evaluating their performance on a total of five different tasks. The chosen models are SmolLM-135M [7], Starcoder-3B [8] and MellumBase-4B [9].

This approach aims to provide answers to the following Research questions:

- RQ1 How widespread is API usage pattern Boilerplate Code across The Heap?
- RQ2 How does Boilerplate Code affect the code completion performance of an LLM when present in the context window or the target of an inference?
- RQ3 Is Boilerplate Code memorized by LLMs?

We found that while the presence of API usage pattern boilerplate code is rare throughout The Heap, it is very common in files containing any of the eight target APIs. We also show that for code generation tasks, LLMs perform better when Boilerplate code is present in the target and worse, when it is present in the context of an inference. We also present evidence that up to 16% of BC is partially memorized by the models we run our evaluation on, a number that we hypothesize would be much higher for bigger LLMs, citing Scaling Laws [10].

Our additional contributions include 1) a pipeline for the detection of API usage pattern BC, and 2) a generic pipeline for inferencing different LLMs and conducting a reference-based evaluation on them.

The organization of this paper is as follows: Section 2 reviews relevant background research. Section 3 describes the methodology. Section 4 presents the results, followed by a discussion in Section 5. Section 6 concludes the paper and provides a summary of the main takeaways. Appendix A provides all results from our inference and evaluation experiments with LLMs.

Keywords: Data Smells, Boilerplate Code, Large Language Models, Memorization in Large Language Models

2 Background Research

In order to address the research questions of this paper, we need solid background knowledge in four domains. Section 2.1 provides insight into the current research of Data Smells, section 2.2 elaborates on the topic of Boilerplate code and existing methods for its detection, section 2.3 describes what LLMs are and how they are evaluated, and section 2.4 introduce the current state of research in the domain of LLM memorization problem.

2.1 Data smells

Data smells are a recent but important topic of interest in the research community due to their implication for Artificial Intelligence models. They were first mentioned as a term in the informal scientific literature in 2014 [11] and in the academic literature in 2018[12], described as consequences of bad practices in data handling. Later, they were systematically examined and defined by Foidl et al. [2] as

```
@Override
protected void onPostExecute(final String ContentText) {
        String tweetText = null;

        if (this.dialog.isShowing()) {
                this.dialog.dismiss();
        }

        if (ContentText == null) {
                tweetText = getString(R.string.fetching_error);
```

Figure 1: Annotated boilerplate code for an API usage pattern of the mask.android.app.ProgressDialog API for Java

"context-independent, data value-based indications of latent data quality issues may lead to problems in the future", analogous to code smells in the software engineering domain. They are reported to be caused by poor practices, such as inadequate data management and handling, or poor data source quality. The potential risks of their presence in the context of AI-based systems include maintenance issues or failures and defect. In our paper, we are interested in those data smells that are induced by poor source quality, particularly those, that arise as a result of quality issues in the source codebases.

A taxonomy of the data smells for code-generation tasks has been created, dividing them in nine subcategories [3]. One of those subcategories, *Limited informativeness*, deals with different kinds of code that brings little to no additional information to the model, e.g. noise tokens or irrelevant code. A common approach to address the "irrelevant code" data smell is to remove the instances containing it [5, 13, 14]. However, it is not described how such an action would affect the performance of an LLM for code generation tasks. A type of irrelevant code is reported to be boilerplate code.

2.2 Boilerplate code

The definition of boilerplate code, a popular term from the SE domain, has been subjective and vague throughout the literature [6]. Systematically derived defining properties include:

Repetitive and standartized. Boilerplate code is usually found in identical forms throughout multiple repositories [15, 16].
Undesirable but necessary. Due to the infeasibility of a certain functionality to be implemented in any other way, the boilerplate approach becomes a single option, in spite of going against software quality standards [6, 15, 16].
Localized. Boilerplate code snippets are usually found in methods and files which are close in proximity rather than spread throughout a whole repository [6].

A snippet considered by us to be boilerplate code is shown in Figure 1.

Due to their repetitive nature boilerplate code fragments are very similar to code clones. Nonetheless, they differ as the former is known to be non-(trivially) avoidable and localized [16], i.e., boilerplate code usually occurs in one place or multiple places with high proximity, rather than throughout a whole repository. While multiple specialized tools exist for the detection of duplicated code [17] and boilerplate text [18–20], almost none are available for boilerplate code detection in particular. One of them is MARBLE [6], a frequency-based tool, built to mine API-usage-pattern boilerplate code for popular Java APIs. It utilizes the Probabilistic API Miner [21] to extract possible candidates and then an AST-based algorithm to filter the potential candidates. For every target API, the pipeline finds boilerplate usage patterns specific to this API. Throughout the thirteen target APIs that the MARBLE paper examines, 1072 out of 9989 files were reported to include boilerplate snippets, accounting for approximately 10.8% of the files.

2.3 Evaluation of Large Language Models

LLMs, powered by the Transformer architecture [22], along with their training and evaluation processes, have become in most recent years the central topic in Artificial Intelligence and Computer Science. The process of forward pass, pivotal in both training and inferencing the model, predicts the most probable next token, an encoded textual fragment, with respect to a given context. A generation includes a prediction of one or more tokens with an optional upper bound.

LLMs are evaluated on their generations. This is most commonly done via reference-based, benchmark, or human evaluation. Benchmarks are complex, used only to test already trained models, and contain significant biases and limitations [23, 24] while human evaluation needs to be performed manually. However, reference-based evaluation is often used in the training process, specifically during fine-tuning, and is a simple, automatic operation which compares a generation with a ground truth, also referred as the *target* of the generation. The output is measured in terms of various evaluation metrics, including, among others, exact match (EM), Levenshtein distance (LD) and recall-oriented understudy for gisting evaluation (ROUGE). Nevertheless, such estimation could also be biased, because of data contamination, i.e. a part of the test data appears in the training data, which the model could overfit on.

A step towards addressing this bias is the release of The Heap [1], a dataset which was deduplicated on the file level against popular training datasets, e.g. The Stack [8], and explicitly dedicated for the purposes of evaluation. Boilerplate code, however, usually appears on the method level, so such leakage could occur even in The Heap, referring back to the repetitive property of boilerplate code. It is noteworthy to report that Data Smells in general are reported to impact the performance of LLMs for code generation tasks [3], but it is not quantitatively described how.

2.4 Memorization in Large Language Models

Due to the vast amounts of data used in the training process of LLMs and their tendency to overfit on it, memorization in the context of LLMs has become an increasingly explored topic. It is defined as a sequence which is outputted verbatim as the ground which is included in the training data [25]. In order to prove memorization we use the notion of k-extractability[26].

Definition. A string (suffix) x of length l is said to be *k-extractable* from a language model f if all of the following conditions hold:

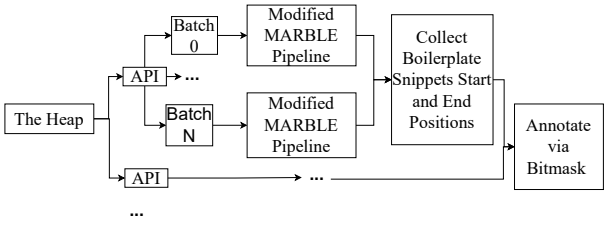

Figure 2: Detection and Annotation pipeline

(1) There exists a length-k context $p = (x_{-k}, \ldots, x_{-1})$ such that the concatenation $p \,||\, x$ appears in the model's training data.
(2) When f is prompted with p, it reproduces x *exactly* using greedy decoding.

3 Methodology

An approach consisting of two phases was developed to address the objectives of this research. The first tackles the detection and annotation of Boilerplate code within The Heap, and is thoroughly described in section 3.1. The second phase aims to evaluate the impact of the presence of the explored data smell on code completion tasks for LLMs. Its details are shown is section 3.2. A diagram showcasing each of them can be found on Figure 2 and Figure 3, respectively.

3.1 Detection and Annotation

The Detection and Annotation component of our experimental setup finds the exact positions of API usage pattern boilerplate code within The Heap and outputs the contaminated files annotated on the character level with a bitmask for their boilerplate presence.

It begins with filtering only files containing a declaration of any of the eight selected Java APIs from our The Heap. Due to its large size and the high time complexity of the MARBLE tool, we divided each of the eight resulting subsets into batches of roughly 1,000 files each. The eight APIs, as well as their presence in The Heap and the number of resulting batches, can be found on Table 1.

Each of those batches was then fed into a modified, pipelined version of the MARBLE tool, which takes the filtered and batched files and outputs the boilerplate files, which contain Boilerplate code along with the exact positions of the boilerplate snippets within those files. The outputs of all batches related to an API are joined, the eight resulting files are then fed to our Annotation tool, and a final output is produced of a bitmask that encodes as 1 each character being part a boilerplate code snippet and as 0 every other character. For example, each red character in Figure 1 was masked as 1 by our pipeline.

3.2 Evaluating the impact of Boilerplate code on LLMs

In order to evaluate how boilerplate code affects LLM generation for coding tasks, we perform five experiments on a total of three models. They all use reference-based evaluation to quantify the results, with four evaluation metrics being calculated for each generation.

Name of Java API	Number of Occurrences	Number of Batches
android.app.ProgressDialog	5,147	4
android.database.sqlite	7,979	8
android.support.v4.app.ActivityCompat	1,630	2
com.squareup.picasso	2,032	2
java.beans.PropertyChangeSupport	1,973	2
java.io.BufferedReader	49,161	48
javax.xml.parsers	13,667	10
javax.xml.transform	8,876	8
TOTAL	**80,465**	**84**

Table 1: Number of files containing each API and resulting batches

Experiments. A total of five experiments were designed to evaluate and explain the impact of the Boilerplate code on code generation tasks and check whether their repetitive nature results in memorization in/by the LLMs.

First, we set up a Next-Token-Prediction task, which applies a standard forward pass process with iteratively increasing context. Similar evaluation is used some fine-tuning training procedures for an LLM, with the sole difference that in this case, instead of a predicted weights metric, e.g. cross-entropy, we use token-based evaluation approaches. In order to conduct the experiment, for each file, we randomly sampled one method that has boilerplate code within and one that does not, if such exist. Then, starting from the first until the penultimate token, we use it along with all previous ones for context and try to predict the next one. This aims to show whether and how BC biases this most standard evaluation technique.

Second, we construct a Contextual Next-Line Prediction experiment, as line completion is one of the most common tasks for LLMs for code generation. For each line, we pick itself and a context consisting of up to 10 lines above it, and then categorize it in four possible groups: 1) both the line and the context have boilerplate code within, 2) the line has boilerplate code within but the context is free of any, 3) the line is free of boilerplate code but the context contains some, 4) both the line and the context are free of boilerplate code. For each category, we randomly sample one pair, if such exists. We causally mask the context and generate up to 160 tokens, with a stopping criterion being the existence of a full, non-empty line which has been generated. By undergoing the described process, we additionally aim to show how the presence or absence of boilerplate code within the context or the target of a generation impacts the accuracy of a model's prediction for a bigger target.

Third, we create a experimental set-up for method-level prediction given only the signature as context. For this task we use FIM masking [27], since the higher complexity of the task, as well as a stopping criterion that checks whether a full method has been generated using the property of bracket symmetry in the Java syntax. We again sampled one method containing and one method free of boilerplate code, if such exist. Then, the method's first line, i.e. the signature, and its last line, i.e. the closing bracket, including or excluding the return statement, are picked as prefix and suffix, respectively. A Fill-In-Middle inference is then performed on the method, with a constraint for maximum generated tokens of 1,500.

This number was chosen as it is an approximation of a M+1.5xSTD measure covering 90% of all methods across The Heap, with M being the mean of all the token sizes of all methods throughout the dataset and STD being the standard deviation of that measurement. The goal is to check whether methods that include boilerplate code are repetitive enough to be predicted by the model with only their signature as context.

Fourth, we created a setup for a full context method-level prediction. It is identical to the setup of the third experiment, with the sole difference that this time, we add everything in the file before the method signature to the prefix and everything in the file after the closing bracket to the suffix. By doing this, we want to further explore the properties that the other three experiments are observing, and above all discover what changes more context will introduce compared to the previous experiment.

Fifth, we perform a k-extractability experiment on randomly sampled snippets of 80 tokens from methods containing boilerplate code. We take k = 64 and try to predict the next 16 tokens, taking only generations that perfectly match the ground truth for positives. This is a setup similar to what Biderman et al. used to predict memorization in LLMs [10]. We then use the data portrait of The Stack v.2 [1] to check for occurrences of the exactly predicted sequences with the actual model. By doing so, we want to approximate what part of the boilerplate code within The Heap is memorized from The Stack in spite of the first being deduplicated against the second.

Models and Data. Three models were selected to carry out the generations. Those are SmolLM-135M [7], StarCoder2-3B [8] and MellumBase-4B [9]. They have all been trained on data that The Heap was deduplicated against, namely The Stack v.2 [8], i.e. we expect no consequences of data contamination on the file level to be observed and only such induced by repetitive code in the method level, including boilerplate code, to be observed. In the case of MellumBase-4b, we need to note that The Stack was only its main but not sole source for training data and it is not disclosed what the other sources are. Experiments involving FIM inference were run only on the two bigger models, namely Starcoder2-3B and MellumBase-4B, as SmolLM-135M is not pretrained for FIM inference. We configured all models for greedy decoding only.

The Heap was used as an evaluation dataset. The files annotated to contain the boilerplate code data smell were accounted as one of the two inputs to our Generation and Evaluation pipeline. The other input was a set of 10,000 files uniformly sampled from the Heap, which contains usages of none of the thirteen APIs reported to contain high frequencies of boilerplate usage patterns [6]. It was used to provide a baseline for results in the experiments.

Evaluation metrics. Each generation is evaluated right after its creation against four evaluation metrics. Exact matching observes what part of the generated tokens are identical and in the same positions. BLEU evaluates what part of the derived 4-grams from the prediction and reference overlp. Levenshtein distance evaluates the edit distance between the ground truth and the predicted text. ROUGE evaluates the overlap of n-grams for different values of n in precision, recall and F1 scores. We calculate this metric for 1-grams, 2-grams and L-grams, with L being the longest matching sequence.

[1] https://stack-v2.dataportraits.org/

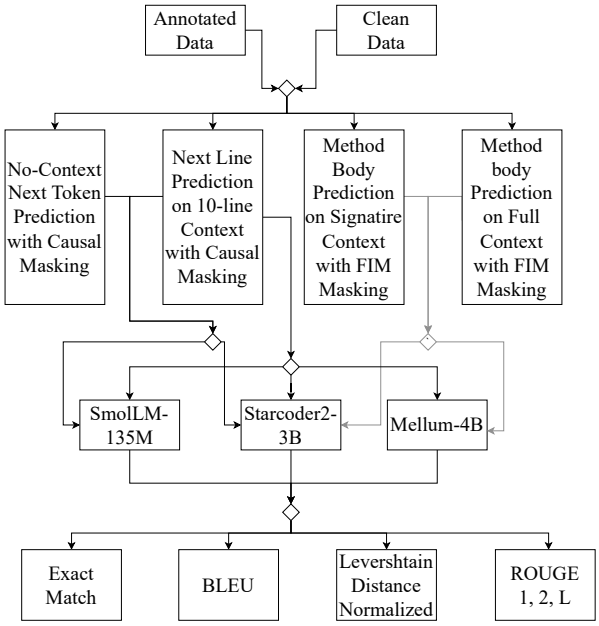

Figure 3: Generation and Evaluation pipeline

4 Results

This section presents the key findings derived from the implemented methodology, highlighting the most significant outcomes in relation to the research objectives. It is organized in three subsections, each addressing one of the three research questions. Section 4.1 shows how much is BC contained in The Heap and describes the properties of the detected files, section 4.2 outlines how does BC affect the code generation performance of the three selected LLMs, and section 4.3 elaborates on whether and how well is BC memorized.

4.1 Presence of Boilerplate code within the Heap

Utilizing our Detection and Annotation pipeline, we have investigated a total of 80,465 files across the Heap, targeting a total of eight APIs in search for boilerplate usage patterns. For each target API, the number of files in which boilerplate usage patterns, i.e. boilerplate code, was found, is listed in Table 2.

The API associated with the largest number of boilerplate usage patterns was found to be java.io.BufferedReader, with over 10000 files with boilerplate code present. The android.app.ProgressDialog API is shown to induce boilerplate usage patterns to the greatest extent, with approximately 51% of all files in which it occurs containing some boilerplate code. The summative detection rate across all files is approximately 21%. The relative presence of the data smell in files for each API can be found in Figure 4. This makes the Boilerplate code data smell for API usage patterns for the eight targeted APIs present in approximately 0.3% of all files within The Heap in its Java subset.

Name of Java API	Number of Boilerplate Files
android.app.ProgressDialog	1,871
android.database.sqlite	1,433
android.support.v4.app.ActivityCompat	826
com.squareup.picasso	92
java.beans.PropertyChangeSupport	32
java.io.BufferedReader	10,174
javax.xml.parsers	1,352
javax.xml.transform	764
TOTAL	**16,544**

Table 2: Number of files containing boilerplate code for each API

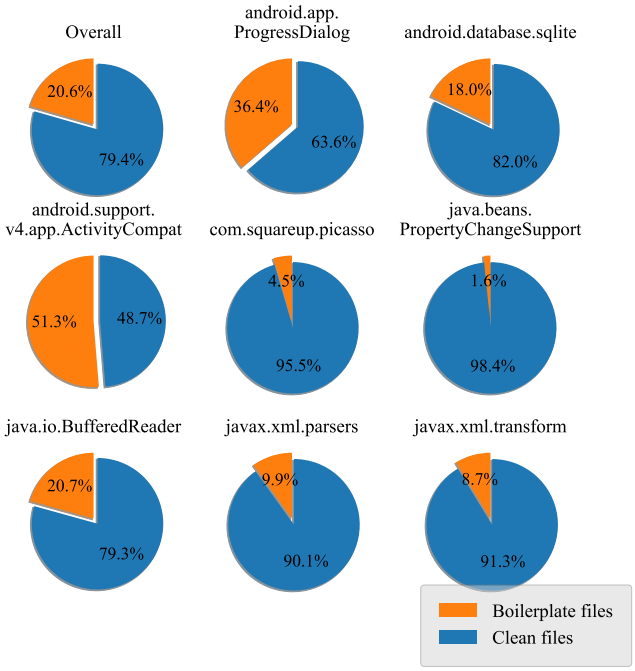

Figure 4: Percentage of Boilerplate code content within files

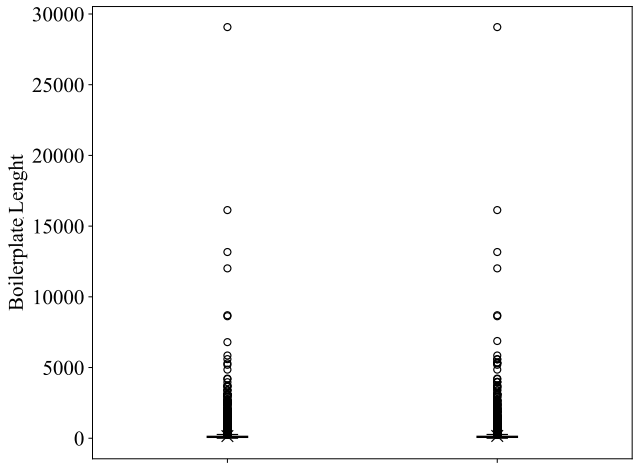

Figure 5: Length of Boilerplate code content within methods (left) and files (right)

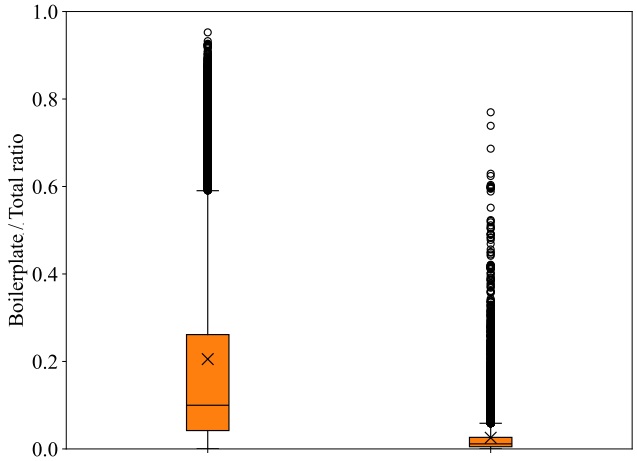

Figure 6: Percentage of Boilerplate code content within methods (left) and files (right)

Moreover, our findings show that the length of almost all of the boilerplate usage patterns within a file is contained within 1,000 characters, with only several exceeding 5,000 characters and only four exceeding 10,000 characters, see Figure 5. The maximal length of boilerplate code within a file we found to be 29,074 characters for an android.database.sqlite usage pattern.

These code snippets we discovered to make up, in most cases, no more than 5% of the respective file they are found in. They also build up on average 21% of the methods they are used in. Boxplots of this data can be seen on Figure 6. The maximum found boilerplate part within a file was 76,97% for a usage pattern for the java.io.BufferedReader API.

More fine-grained statistics for each of the target APIs can be found in Appendix A.

4.2 Boilerplate code impact on LLMs' code generation performance

The generations on which the following results are based are performed on L-4 GPUs with 22.5GB of VRAM, except for the fourth experiment, which was conducted on A-100 GPUs with 40GB of VRAM.

The third and fourth experiment were run on two datasets of 1,000 files each, randomly sampled from the annotated and the baseline data, respectively. The second and fifth ones were conducted on two datasets of 10,000 files each, sampled in the same manner. Due to the high computational cost, the first was run only on the two smaller models, i.e. SmolLm-135M and Starcoder2-3B, on samples of 1,000 and 600, respectively, produced in the same manner. The results presented for Tables 3-6 represent the averaged respective

metric	bp	fr	base
exact match	0.670	0.582	0.530
bleu	0.474	0.351	0.292
exact match	0.801	0.722	0.679
bleu	0.624	0.524	0.467

Table 3: Results for the Next Token Prediction Experiment. The first group of rows show SmolLM results, while the second - Starcoder2

metric	bp	fr	base
exact match	0.004	0.007	0.007
bleu	0.036	0.058	0.052
rouge-l-p	0.503	0.397	0.342
rouge-l-r	0.134	0.228	0.278
exact match	0.000	0.018	0.008
bleu	0.002	0.026	0.060
rouge-l-p	0.462	0.365	0.341
rouge-l-r	0.085	0.136	0.299

Table 4: Results for the Limited Context Method Body Prediction Experiment. The first group of rows show Starcoder2 results, while the second - MellumBase

mean across all generations. The standard error for each velue was calculated to be below 0.01.

We found that LLMs perform significantly better for next-token prediction tasks for methods are which contaminated with BC data smell, as can be seen in Table 3. Estimating the performance via the exact match metric, we found that the two LLMs under investigation perform with a difference of between 8 and 9 percentage points compared to the BC-free methods within the same files and between 12 and 14 percentage points compared to the baseline, as shown in Figure 7.

The ability to predict a method body based on the context around it, i.e. experiments three and four, is measured to follow two important trends, see Table 4 and Table 5.

First, naturally, giving more context to the LLM boosts the performance across all metrics but ROUGE. However, we show that with respect to these metrics, prediction of BC-contaminated methods results in negligibly worse generations, showing that the investigated LLMs' capabilities to predict a method's body on its signature do not generally depend on the presence of BC in the method itself. Patterns of BC-contaminated methods being more predictable with respect to EM, BLEU and LD, nevertheless, slightly emerge when the LLMs are given more context. In this setting, we observe with the BLEU score showing approximately 5.5% improvement over the baseline across both models and slightly less over the average for BC-free methods in the contaminated files. The highest spike for BLEU we show to be for BC-contaminated methods on MellumBase-4B which totals 33 percentage points improvement.

metric	bp	fr	base
exact match	0.015	0.014	0.010
bleu	0.235	0.222	0.180
rouge-l-p	0.505	0.421	0.361
rouge-l-r	0.444	0.559	0.612
exact match	0.101	0.062	0.043
bleu	0.333	0.292	0.279
rouge-l-p	0.531	0.499	0.534
rouge-l-r	0.385	0.481	0.579

Table 5: Results for the Full Context Method Body Prediction Experiment. The first group of rows show Starcoder2 results, while the second - MellumBase

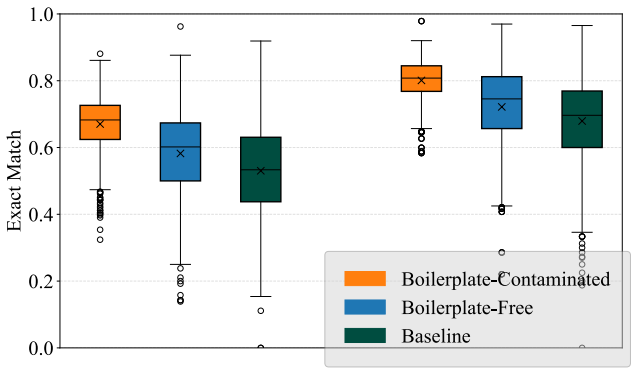

Figure 7: Exact Match Metrics for Next-Token Prediction. The first group of boxplots (left) show SmolLM results, while the second (right) - Starcoder2

Second, the ROUGE-L score did not show a major change between the two experiments, neither nominally nor relatively. However, it revealed a pattern of higher precision on predicting BC-contaminated method bodies, and higher recall on methods from the baseline dataset, as well as methods within BC-containing files that are BC-free. This directly implies that for methods with BC within, LLMs predicted more of the reference n-grams, but a lesser number of the predicted n-grams were contained in the reference solution.

The results for the second experiment are shown in Table 6 where the column names represent BC-contaminated target with BC-contaminated context (mm), BC-contaminated target with BC-free context (mu), BC-free target with BC-contaminated context (um), BC-free target with BC-free context (uu), all from the annotated dataset and a BC-free target with BC-free context from the baseline dataset (base), respectively. Taking the EM, BLEU and normalized Levenshtein distance metrics into consideration, we show that the two bigger models generally performed significantly better than the smaller one, with Starcoder2-3B scoring better than the bigger MellumBase-4B. Differentiating between context and target of the generations, with respect to where BC is present, we can see that the pattern of BC-contaminated targets scoring higher than the

metric	mm	um	mu	uu	base
exact match	0.308	0.305	0.181	0.251	0.258
bleu	0.261	0.275	0.128	0.194	0.191
levenshtein	0.671	0.665	0.800	0.728	0.719
exact match	0.585	0.518	0.399	0.444	0.472
bleu	0.519	0.362	0.294	0.340	0.366
levenshtein	0.401	0.435	0.579	0.541	0.512
exact match	0.566	0.505	0.390	0.441	0.455
bleu	0.501	0.347	0.282	0.335	0.344
levenshtein	0.418	0.449	0.589	0.548	0.529

Table 6: Results for the Next Line Prediction Experiment. The first group of rows show SmollLM results, the second - Starcoder2, and the third - MellumBase

Model Name	Eligible Candidates	Perfect Matches (% of Eligible)
SmolLM-135M	7,825	260 (3.3%)
Starcoder2-3B	7,382	800 (10.8%)
MellumBase-4B	8,523	2,038 (23.9%)

Table 7: Number of perfect matches found in our k-extractability experiment against the total number of generations for valid references

Model Name	Sampled Snippets	Partially Matched Candidates	Perfectly Matched Candidates
SmolLM-135M	50	40	28
Starcoder2-3B	50	43	22
MellumBase-4B	50	33	13

Table 8: Number of Perfect and Partial Matches in The Stack out of Perfectly Matched Generations

baseline is maintained across all models. We report a slightly better performance for the um compared to the mm category for SmolLM-135M. This behaviour is majorly reversed for the two bigger models. The results for the uu category across the three models are slightly lower than those for the baseline, a difference contained within 1% in most cases. However, we report a drop in the performance of all LLMs on the um category generations, which is shown to be over 20 percentage points compared to the mm category for the BLEU score on the two larger models.

4.3 Memorization of Boilerplate code

We conducted our fifth, k-extractability, experiment for all three models on a randomly sampled dataset consisting of 10,000 entries which was again uniformly sampled from our set of annotated, BC-contaminated files. We used L-4 GPUs with 22.5GB of VRAM for this purpose.

For each model, we show the number of generations, which were evaluated to be a perfect match with their references and the total number of tested sequences, along with the calculated proportion between the two. A sampled candidate was considered eligible only if it consisted of more than 80 tokens for the respective model and all ineligible were discarded. This information can be seen in Table 7. While it reinforces the claim that the bigger an LLM is, the better it performs, the data also implies that up to 24% of the code generated in this experiment by the biggest model could have been memorized by it.

We checked whether the perfectly matching code occurs verbatim in the Stack v2 via the Data Portrait provided by the creators of the dataset. It is important to note that the tool is reported to not be updated with the latest checkpoints of the dataset. Therefore, the results of this check are to be considered a lower bound of the actual ones. Out of the 50 entries that we uniformly sampled from the results scoring EM of 1.00 for each of the models, we list the partially and the fully matched snippets in Table 8. Assuming they resemble the actual distribution, we can report that while only a minority of the perfectly matched by the model snippets have a perfect match in The Stack, between 66% and 86% of them have a partial match.

5 Discussion

In the following we present a discussion about our findings. Section 5.1 presents our interpretation of the results, section 5.2 discussed in-depth their implication and section 5.3 elaborates on this paper's limitations and possible threats to its validity.

5.1 Interpretation

Presence of Boilerplate Code. Focusing on API usage pattern BC, we clearly showed that some APIs induce the necessity of BC substantially more than others. These findings confirm the results presented by Nam et al. [6]. These imbalances are mainly attributed to the lack of suitable abstractions which can replace the usage patterns with simple consise statements. This owes to either a conscious decision by the developers of such APIs to allow more "manual" control over the APIs functionality within a client code or their inability to provide such.

Boilerplate code impact on LLMs. Our results for reference-based evaluation across multiple experiments proved beyond reasonable doubt that BC has a significant impact on an LLM's performance for code generation tasks.

In almost all experiments, with the sole exception of experiment three, we saw that LLMs perform much better when they have to predict code which is fully or partially contaminated with BC. This is the first indication of potential memorization of BC by LLMs, which we can attribute to its repetitive nature.

We also showed that both BC-containing and BC-free contexts prompt the BC-containing target to be predicted better than the targets from the baseline data for the smaller model. For the two larger models, the former proves to cause significantly better results, while the latter type of context induces similar results to those of the baseline. Taking into consideration the scaling law for LLMs stating that bigger models tend to memorize more of their training data [10, 28], we claim that this spike is due to said memorization.

Nevertheless, an analysis of our results discovers that BC-contaminated context prompts a drop in performance when an LLM is expected to predict a BC-free line directly after it. Our interpretation of this outcome is that, given this context, the model is likely to continue

generating boilerplate code, an idea which resonates with the already established claims of memorization. Similar is the case with an imbalance between precision and recall in the ROUGE metric in experiments three in four. A continuous generation of boilerplate code inside the predicted method body, given only the method's signature, results in more tokens of the reference being guessed correctly, but fewer of the guessed tokens being correct themselves.

Analyzing the information derived from experiments one and four, we showed the pattern that models perform slightly better in predicting the BC-free methods within BC-contaminated files than methods of the baseline dataset. This outcome we accredit to class and method coherence. While this code is not considered to be a part of the usage pattern, as the pattern itself appears in almost identical form, it will have a higher degree of similarity than code which has no implicit cohesive influence of BC. These slight patterns of similarity, encoded by the LLMs during training, show up in the results from our experiments.

Finally, we address the results of experiment three, where the prediction of BC-contaminated methods based on their signature alone, yielded worse evaluation scores than those of BC-free or baseline dataset methods on all metrics but ROUGE precision scores. We claim that there is no explicit relation between a method's signature and the usage of specific API usage patterns, which are considered boilerplate code. This is further reinforced by our observations that this kind of BC builds in most cases only a minor part of a method.

Memorization. The evidence produced by our k-extractibility experiment clearly shows that a non-insignificant part of the BC-contaminated methods is memorized by LLMs. We also provided evidence that this part grows with the the size of the explored model, as reported in previous research [10, 28]. However, in the majority of cases, only partial matches were found in their training data. We conclude that due to small syntactic deviations of the BC-usage patterns, mostly partial statements of the usage pattern were memorized verbatim, which were later encoded in closest proximity by the LLMs, enabling them to predict those fragments more than efficiently.

5.2 Implications

Although our findings show that only 0.3% of The Heap is contaminated with BC data smell, we report that the eight targeted Java APIs induce the smell in more than 20% of the files in which they are used. This implies that if the set of targeted APIs grow, the total presence of BC throughout The Heap, as well as other large code datasets, could end up being much higher.

The reported impact of boilerplate code on LLM code generation tasks implies a considerable bias in the reference-based evaluation of LLMs, as well as in their fine-tuning. This requires the scientific community to produce a way of dealing with this very bias. The proposed removal approach for irrelevant code data smells, namely clearing all files from methods containing the smell, reduces the said bias but does not remove it completely. Developing an effective approach to address this problem is out of the scope of the current research and should be addressed in future work.

Lastly, our results show that LLMs, especially bigger ones, tend to partially memorize API usage pattern boilerplate code. This implies a higher performance for BC targets of AI-based software development tools that are powered by such models. However, it also constitute a legal threat to users of such tools with respect to intellectual property copyright [1], as well as a vulnerability for data extraction attacks [10].

5.3 Limitations and Threats to Validity

There are three main limitations of the current research.

First, our results are based on just one type of boilerplate code in the Java language, namely API usage patterns. Extending it to other types of boilerplate code within java, i.e. getter and setter methods, may yield different results.

Second, due to constraints on computing power and time, we investigated how the smell affects relatively small LLMs. Whereas we expect the same behaviour to be maintained and even amplified for larger models due to scaling laws, we do not provide a guarantee for this. This limitation of the current paper, as well as the aforementioned one, needs to be addressed in future research.

Third, we relied on external tools for the detection of the data smell [6]. Defects on those tools, if found, could undermine the implications of the current work.

6 Conclusion

There are three main takeaways in our research.

First, while the files we found to contain boilerplate code data smell in The Heap are a small portion of the dataset, we reported that popular Java APIs induce the smell in large proportions. Moreover, we found that API usage pattern BC makes up for only a minor part of the methods it is contained in.

Second, we demonstrated that boilerplate code significantly impacts LLMs on code generation tasks, as they tend to predict it much better than baseline code. Therefore, we concluded that the data smell introduces a positive bias on the reference-based evaluation of such models and emphasized the need to address it.

Third, we discovered that LLMs partially memorize boilerplate code during their training process. Consequently, we discussed the implications of this finding, with respect to both industry and data privacy.

Responsible research

This research was conducted in accordance with high ethical standards, with a strong focus on transparency, reproducibility, and responsible AI usage.

All experiments were designed for reproducibility: a fixed random seed was used for sampling, and all code and results have been open-sourced and are freely available on GitHub. This ensures that others can verify, replicate, and build upon our work.

The datasets used are publicly available and covered by open copyright or permissive licenses. When handling data containing personal information, we took care to respect privacy and minimize risk, following appropriate ethical guidelines.

We also considered the implications of using large language models (LLMs) in software development tasks. LLMs can partially memorize boilerplate or commonly seen code patterns. This raises concerns about the unintended reproduction of sensitive or copyrighted material. We therefore stress the importance of responsible

use, especially in production environments or when handling proprietary data.

Within the scope of the current research Large Language Models were used as a tool only for the purposes of code documentation and language polishing with respect to writing.

References

[1] Jonathan Katzy, Razvan Mihai Popescu, Arie van Deursen, and Maliheh Izadi. The heap: A contamination-free multilingual code dataset for evaluating large language models, January 2025. arXiv preprint arXiv:2501.09653.

[2] 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.

[3] Antonio Vitale, Rocco Oliveto, and Simone Scalabrino. A catalog of data smells for coding tasks. *ACM Trans. Softw. Eng. Methodol.*, 34(4), April 2025.

[4] Yuxing Cheng, Yi Chang, and Yuan Wu. A survey on data contamination for large language models, 2025.

[5] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. An empirical study on the usage of transformer models for code completion. *IEEE Transactions on Software Engineering*, 48(12):4818–4837, 2022.

[6] Daye Nam, Amber Horvath, Andrew Macvean, Brad Myers, and Bogdan Vasilescu. Marble: Mining for boilerplate code to identify api usability problems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 615–627, 2019.

[7] Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Leandro von Werra, and Thomas Wolf. Smollm - blazingly fast and remarkably powerful, 2024.

[8] 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.

[9] 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.

[10] Stella Biderman, USVSN PRASHANTH, Lintang Sutawika, Hailey Schoelkopf, Quentin Anthony, Shivanshu Purohit, and Edward Raff. Emergent and predictable memorization in large language models. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 28072–28090. Curran Associates, Inc., 2023.

[11] Jacob Harris. Distrust your data, May 2014.

[12] Tushar Sharma, Marios Fragkoulis, Stamatia Rizou, Magiel Bruntink, and Diomidis Spinellis. Smelly relations: measuring and understanding database schema quality. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, page 55–64, New York, NY, USA, 2018. Association for Computing Machinery.

[13] Matteo Ciniselli, Luca Pascarella, and Gabriele Bavota. To what extent do deep learning-based code recommenders generate predictions by cloning code from the training set? In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 167–178, New York, NY, USA, 2022. Association for Computing Machinery.

[14] 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.

[15] Chaiyong Ragkhitwetsagul. Code similarity and clone search in large-scale source code data. 10 2018.

[16] Christina Zacharoula Chaniotaki. An empirical investigation of boilerplate code. 1 2022.

[17] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1157–1168, 2016.

[18] Matthias Gallé and Jean-Michel Renders. Boilerplate detection and recoding. pages 462–467, 04 2014.

[19] Roland Schäfer. Accurate and efficient general-purpose boilerplate detection for crawled web corpora. *Lang. Resour. Eval.*, 51(3):873–889, September 2017.

[20] Marcos Fernández-Pichel, Manuel Prada-Corral, David E. Losada, Juan C. Pichel, and Pablo Gamallo. An unsupervised perplexity-based method for boilerplate removal. *Natural Language Engineering*, 30(1):132–149, 2024.

[21] Jaroslav Fowkes and Charles Sutton. Parameter-free probabilistic api mining across github. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 254–265, New York, NY, USA, 2016. Association for Computing Machinery.

[22] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[23] Sourav Banerjee, Ayushi Agarwal, and Eishkaran Singh. The vulnerability of language model benchmarks: Do they accurately reflect true llm performance?, 2024.

[24] James Fodor. Line goes up? inherent limitations of benchmarks for evaluating large language models, 2025.

[25] Chiyuan Zhang, Daphne Ippolito, Katherine Lee, Matthew Jagielski, Florian Tramèr, and Nicholas Carlini. Counterfactual memorization in neural language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[26] Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramer, and Chiyuan Zhang. Quantifying memorization across neural language models, 2023.

[27] 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.

[28] Ali Al-Kaswan, Maliheh Izadi, and Arie van Deursen. Traces of memorisation in large language models for code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

A Full Result Tables for Inference Experiments

Table 9: Results for the No-Context Next Token Prediction Experiment

Metric	bp	fr	base
exact match	0.670	0.582	0.530
bleu	0.474	0.351	0.292
levenshtein	0.335	0.431	0.488
meteor	0.660	0.583	0.541
rouge-1-p	1.000	1.000	1.000
rouge-1-r	1.000	1.000	1.000
rouge-1-f	1.000	1.000	1.000
rouge-2-p	1.000	1.000	1.000
rouge-2-r	1.000	1.000	1.000
rouge-2-f	1.000	1.000	1.000
rouge-l-p	1.000	1.000	1.000
rouge-l-r	1.000	1.000	1.000
rouge-l-f	1.000	1.000	1.000
exact match	0.801	0.722	0.679
bleu	0.624	0.524	0.467
levenshtein	0.422	0.384	0.367
meteor	0.781	0.734	0.687
rouge-1-p	1.000	1.000	1.000

Metric	bp	fr	base
rouge-1-r	1.000	1.000	1.000
rouge-1-f	1.000	1.000	1.000
rouge-2-p	1.000	1.000	1.000
rouge-2-r	1.000	1.000	1.000
rouge-2-f	1.000	1.000	1.000
rouge-l-p	1.000	1.000	1.000
rouge-l-r	1.000	1.000	1.000
rouge-l-f	1.000	1.000	1.000

Table 10: Results for the Next Line Prediction Experiment

metric	mm	um	mu	uu	base
exact match	0.308	0.305	0.181	0.251	0.258
bleu	0.261	0.275	0.128	0.194	0.191
levenshtein	0.671	0.665	0.800	0.728	0.719
meteor	0.389	0.394	0.248	0.339	0.339
rouge-1-p	0.353	0.418	0.263	0.348	0.365
rouge-1-r	0.340	0.375	0.234	0.328	0.339
rouge-1-f	0.337	0.382	0.236	0.327	0.339
rouge-2-p	0.232	0.279	0.137	0.207	0.210
rouge-2-r	0.231	0.275	0.135	0.205	0.205
rouge-2-f	0.230	0.274	0.135	0.203	0.204
rouge-l-p	0.352	0.417	0.262	0.347	0.364
rouge-l-r	0.339	0.375	0.234	0.328	0.339
rouge-l-f	0.337	0.382	0.235	0.327	0.338
exact match	0.585	0.518	0.399	0.444	0.472
bleu	0.519	0.362	0.294	0.340	0.366
levenshtein	0.401	0.435	0.579	0.541	0.512
meteor	0.634	0.584	0.468	0.520	0.550
rouge-1-p	0.613	0.617	0.483	0.523	0.567
rouge-1-r	0.595	0.602	0.462	0.508	0.544
rouge-1-f	0.597	0.600	0.461	0.506	0.543
rouge-2-p	0.505	0.493	0.335	0.375	0.403
rouge-2-r	0.502	0.494	0.332	0.371	0.396
rouge-2-f	0.501	0.489	0.331	0.369	0.394
rouge-l-p	0.613	0.616	0.482	0.523	0.567
rouge-l-r	0.595	0.602	0.461	0.508	0.543
rouge-l-f	0.597	0.600	0.461	0.505	0.542
exact match	0.566	0.505	0.390	0.441	0.455
bleu	0.501	0.347	0.282	0.335	0.344
levenshtein	0.418	0.449	0.589	0.548	0.529
meteor	0.618	0.569	0.457	0.516	0.528
rouge-1-p	0.596	0.605	0.477	0.520	0.547
rouge-1-r	0.580	0.588	0.453	0.503	0.524
rouge-1-f	0.581	0.587	0.453	0.501	0.523
rouge-2-p	0.489	0.478	0.327	0.368	0.380
rouge-2-r	0.487	0.480	0.323	0.365	0.374
rouge-2-f	0.485	0.475	0.323	0.363	0.373
rouge-l-p	0.596	0.604	0.476	0.520	0.546
rouge-l-r	0.579	0.588	0.452	0.502	0.523
rouge-l-f	0.581	0.587	0.453	0.500	0.522

Table 11: Results for the Limited Context Method Body Prediction Experiment

metric	bp	fr	base
exact match	0.004	0.007	0.007
bleu	0.036	0.058	0.052
levenshtein	0.985	0.977	0.976
meteor	0.098	0.188	0.216
rouge-1-p	0.521	0.408	0.353
rouge-1-r	0.137	0.231	0.280
rouge-1-f	0.180	0.229	0.246
rouge-2-p	0.145	0.142	0.109
rouge-2-r	0.050	0.117	0.148
rouge-2-f	0.058	0.095	0.100
rouge-l-p	0.503	0.397	0.342
rouge-l-r	0.134	0.228	0.278
rouge-l-f	0.176	0.225	0.242
exact match	0.000	0.018	0.008
bleu	0.002	0.026	0.060
levenshtein	0.993	0.977	0.979
meteor	0.038	0.117	0.233
rouge-1-p	0.507	0.365	0.351
rouge-1-r	0.090	0.136	0.302
rouge-1-f	0.145	0.164	0.262
rouge-2-p	0.121	0.054	0.121
rouge-2-r	0.020	0.033	0.161
rouge-2-f	0.032	0.041	0.113
rouge-l-p	0.462	0.365	0.341
rouge-l-r	0.085	0.136	0.299
rouge-l-f	0.137	0.164	0.257

Table 12: Results for the Full Context Method Body Prediction Experiment

metric	bp	fr	base
exact match	0.015	0.014	0.010
bleu	0.235	0.222	0.180
levenshtein	0.954	0.961	0.969
meteor	0.380	0.409	0.385
rouge-1-p	0.514	0.425	0.365

metric	bp	fr	base
rouge-1-r	0.452	0.563	0.615
rouge-1-f	0.425	0.406	0.382
rouge-2-p	0.321	0.277	0.229
rouge-2-r	0.307	0.422	0.486
rouge-2-f	0.276	0.276	0.254
rouge-l-p	0.505	0.421	0.361
rouge-l-r	0.444	0.559	0.612
rouge-l-f	0.418	0.403	0.379
exact match	0.101	0.062	0.043
bleu	0.333	0.292	0.279
levenshtein	0.920	0.941	0.952
meteor	0.476	0.487	0.534
rouge-1-p	0.541	0.504	0.538
rouge-1-r	0.391	0.484	0.581
rouge-1-f	0.423	0.454	0.523
rouge-2-p	0.341	0.340	0.367
rouge-2-r	0.264	0.367	0.463
rouge-2-f	0.278	0.324	0.384
rouge-l-p	0.531	0.499	0.534
rouge-l-r	0.385	0.481	0.579
rouge-l-f	0.415	0.450	0.521