CloneWars: The Effects of Deduplication on LLM Code Completion Performance

Aiman Abdul Wahab, Alexandru-Radu Moraru, Ana Ţerna, Dean Polimac,
Floris van Veen, Nada Mouman, Zuji Zhou
Delft University of Technology Delft, Netherlands
aawahab@tudelft.nl, amoraru@tudelft.nl, aterna@tudelft.nl, dpolimac@tudelft.nl,
bwmfvanveen@tudelft.nl, nmouman@tudelft.nl, zzhou-38@tudelft.nl

***Abstract*—As the usage of Large-Language Models (LLMs) grows, the volume of underlying data in the wild, often used for training, also expands, raising the risk of duplication. Data duplication can introduce biases and affect model performance, making deduplication efforts essential. This study investigates the effect of duplicate training data on CodeParrot's code completion performance for Python, analyzing both exact and near-deduplication across different levels: repository, file, and function. Our findings show that more granular deduplication increases perplexity, highlighting data contamination in the training sets. A Jaccard threshold of 0.75 seems to strike an optimal balance between eliminating duplicates and minimizing false positives, while changes to n-gram window size had little impact.**

***Index Terms*—deduplication, code completion, large-language models, MinHashLSH**

I. Introduction

Today, large language models (LLMs) are used more than ever for code-related tasks such as code completion, code generation, bug detection, and code refactoring [1]–[3]. Consequently, one of the important points in the creation of LLMs is preparing training datasets. LLMs designed for code-related tasks are typically trained on datasets containing an abundance of code files from public repositories. These datasets may include duplicate or near-duplicate code[1], as developers frequently reuse code snippets across projects [4] or copy-paste code blocks with minor modifications from online sources like StackOverflow. Given the importance of preventing biases and enhancing model reliability, it is crucial to address duplicate code during dataset preprocessing. For example, in LLMs, duplicated code with security flaws, like outdated cryptography or poor input validation, can lead the model to generate insecure or biased code, reinforcing vulnerabilities or simply poorly written code [5].

Typically, different techniques are used to remove duplicate code in the training data, including exact deduplication and near-deduplication. The primary research question that we want to answer is how exact and near-deduplication affect the performance of LLMs on the code completion task. Additionally, we want to analyze how variations in MinHash Locality-Sensitive Hashing (LSH), such as the n-gram size and Jaccard similarity thresholds, could influence the performance of LLMs.

A. Contributions

- An investigation into the impact of exact and near-deduplication on the CodeParrot model[2] for code completion.
- An exploration of different MinHashLSH configurations, and their impact on code completion performance.
- A public code repository[3] that was used to obtain the results in this paper.

B. Research Questions

We hereby formalize the main research questions of this paper:

- **RQ1**: *To what extent do different code deduplication methods affect the performance of an LLM on the code completion task?*
- **RQ2**: *How do different MinLSH parameters affect the performance of an LLM on the code completion task?*

II. Problem Statement and Motivation

The main problem we wanted to address is how different variants of code deduplication, namely exact and near, affect the performance of LLMs in code-related tasks. Due to time constraints, we focused exclusively on code completion, however, we believe this is a good starting point for ramifications into other code-related tasks. Data deduplication is of paramount importance because training datasets for LLMs typically contain very similar code snippets, which can negatively impact the model's learning process, and further impact its ability to generate diverse, and secure code.

Besides the aforementioned ideas, this topic intrigues us for several reasons. Firstly, while duplication techniques have been studied before to improve model generalization and training data diversity, limited effort has been spent comparing different levels of duplication in LLM performance when considering specific coding tasks. Secondly, we are motivated to explore different MinHashLSH configurations for near deduplication, as these may provide optimizations that improve model performance and help mitigate underlying biases.

[1] In this paper, near-duplicates refer to fragments of code that are similar, yet not exactly identical. These similarities could occur in code structure, logic, different variable names and comments, or just operation sequencing.

[2] https://huggingface.co/codeparrot

[3] https://github.com/120L021101/LLM_paper

III. Background and Related Work

While far from being exhaustive, this background section is intended to provide sufficient context for the reader and lays the foundation for our analysis and future research directions. This section covers a summary of LLMs (III-A), their use in various coding tasks (III-B), details on pretraining and fine-tuning processes (III-C), data contamination issues (III-D), the impact of code duplication (III-E), and different deduplication strategies (III-F).

A. Overview of LLMs

Large Language Models are deep neural networks designed to process and generate human-like text. These models are typically based on transformer architectures [6], which allow for efficient handling of sequential data. LLMs are pre-trained on vast amounts of text data, enabling them to learn patterns, syntax, and semantics across a wide range of languages and domains. Due to their scale and versatility, LLMs have been successfully applied in various tasks such as natural language understanding, text generation, machine translation, and code-related tasks [7], showing significant advancements in both accuracy and performance compared to previous state-of-the-art approaches.

B. LLMs for Coding Tasks

Automating code-related tasks can boost software development by improving efficiency, reliability, and understanding. We highlight four key tasks, namely code generation, code completion, bug detection, and code summarization:

Code generation involves automatically creating fully functional code snippets or entire programs based on given specifications or natural language descriptions.

Code completion predicts and suggests the next part of a code snippet based on the context. This functionality helps developers write code more efficiently by reducing the need for repetitive typing, enhancing productivity and reducing errors.

Bug detection automatically identifies potential errors or bugs in the code. For example, when trying to divide the variable a by b, a bug detection model might flag an issue when $b = 0$, which would result in a division by zero error. Identifying such bugs early ensures software reliability, improves code quality, and prevents runtime errors.

Code summarization generates high-level descriptions of code to improve understanding and documentation based on the context given. This task aids developers in quickly understanding code functionality, making it easier to maintain and collaborate on projects.

Due to time constraints, we will narrow our exploration to just one going forward, namely code completion. However, a common factor for all these tasks is that benchmarks such as HumanEval+ [8] provide crucial metrics to assess performance and, therefore, progress in the field.

C. Pretraining and Fine-Tuning in LLMs

Pretraining and fine-tuning are crucial processes in developing LLMs. Firstly, pretraining involves using large datasets to *teach* the model generic language representations through self-supervised learning, with notable examples like BERT [9] and GPT-3 [10] proving the effectiveness of this approach.

On the other hand, fine-tuning can be considered a *post-training* step, wherein an LLM is adapted to specific tasks or domains by training it further on considerably smaller, task-specific datasets using supervised learning. This process *fine-tunes* the underlying model's parameters to align with the new task or simply to address its shortcomings.

In the context of code deduplication, a model can be fine-tuned on a deduplicated dataset to nudge it into *forgetting* the biases learned during pretraining. This can further improve generalization across diverse coding tasks, avoid suggesting poorly written or insecure code, or mitigate privacy risks in language models [11].

D. Data Contamination

When evaluating a model, it is crucial to ensure that the evaluation dataset is independent of the training dataset, otherwise, the results may be biased. For example, CodeLiveBench findings [12] indicate that some large models may have been exposed to portions of the HumanEval+ [8] evaluation dataset during training, leading to biased outcomes. Therefore, we plan to apply a repository-level deduplication method to maximize the validity of the evaluation dataset.

E. The effect of code duplication

As the demand for machine learning models grows, so does the need for vast and high-quality training data. Training a model like GitHub Copilot requires extensive data scraping from GitHub, which leads to significant duplication. This duplication often stems from different versions of the same repository or users copying code snippets from online sources. As a result, the dataset includes repetitive code segments, ranging from entire files to smaller functions.

This duplication can introduce bias in models, which can result in inflated evaluated performance relative to the true performance [13]. This is introduced in the form of inaccurate representation, where evaluations show improved performance (e.g., lower perplexity) vs. true unbiased performance. This can be due to test data and training data being exact copies of each other. There exist methods to detect and remove duplicates from the data, but we must also specify what type of duplication we want to remove: exact and near. Exact refers to exact copies of each other and near refers to snippets which may include slight differences but in essence, the functionality is almost identical. In these cases, it is often best to remove duplicate instances to get a better evaluation of the true accuracy.

Recent studies have shown that repeated data used during training can significantly undermine a model's performance and generalization ability. For instance, Hernandez et al. [14] explore how repeated data during training can harm large language models, showing that even a small amount of repetition can significantly reduce performance by causing the model to memorize instead of generalize. It finds that repeated data impacts key internal mechanisms and can lead to a noticeable drop in accuracy, even when most of the training data is unique. The paper highlights that repeating just 0.1% of the training data 100 times can cause an 800M parameter model to perform as poorly as a model half its size (400M parameters), even though 90% of the training data remains unique.

Moreover, existing LLM datasets contain near-duplicate fragments and repetitive substrings. Lee et al. [15] identified that near-duplicate and exact duplicate content are pervasive in large-scale datasets, with some text repeated thousands of times. By removing duplicate examples through techniques like substring matching and MinHash-based approximate matching, the researchers were able to reduce model memorization rates by a factor of 10 and eliminate substantial train-test overlap, which otherwise inflates evaluation metrics. Specifically, models trained on deduplicated datasets achieve lower perplexity on unique validation examples, indicating that deduplication enhances generalization by reducing overfitting and encouraging the learning of robust patterns instead of memorization.

F. Deduplication Techniques

Deduplication is a common step in many pre-processing pipelines adopted by recent research. For instance, Kocetkov et al. [16] used the following deduplication pipeline: they applied MinHash with 256 permutations of all documents and used Locality Sensitive Hashing (LSH) [17] to find clusters of duplicate documents. They also used the Jaccard similarity threshold of 0.85 to further reduce the size of the clusters by removing files with high similarity.

The MinHash LSH process begins by breaking down each document into sets of shingles. These sets are then transformed into compact MinHash signatures by applying a series of hash functions. For each hash function H, the smallest hash value among all elements x in set A is selected to form the MinHash signature, as shown in Eq. 1.

$$a_{\text{min}} = \min_{x \in A} H(x) \tag{1}$$

The similarity between two items is computed by comparing their signatures. The number of matching values between two signatures approximates the Jaccard similarity of the corresponding sets. To increase efficiency, LSH groups similar items into buckets with a high probability before making direct comparisons. The combination of MinHash and LSH significantly reduces the number of comparisons, making the process more computationally efficient [18].

Allal et al. [19] further explored the interplay between the MinHash LSH parameters, concluding that using 5-grams and a Jaccard threshold of 0.7 strikes an effective balance between false positives and false negatives. They observed that lower Jaccard thresholds increase the rate of false negatives, while higher thresholds (e.g., > 0.85) increase the risk of incorrectly removing valuable duplicates.

Deduplication of code datasets can also be performed at various levels. The repository-level approach is the most commonly used in well-known model pretraining [20]. However, for more fine-grained preprocessing, combining file-level and repository-level deduplication has proven to eliminate a higher number of retained tokens. For example, [21] evaluated file-level deduplication on two datasets (i.e., HumanEval and MBPP) during pretraining and found that it significantly improved pass@k results compared to repository-level deduplication. They also discovered that file-level deduplication could reduce up to 52 billion token equivalents between files on the datasets that were used. Furthermore, while they explored deduplication at the chunk level, no substantial benefits were observed. Ultimately, the most effective approach remains repository-level deduplication, followed by fuzzy deduplication at the file level.

IV. Approach

This section outlines the methodology used to examine the impact of code deduplication on the performance of large language models in code completion tasks. The approach is structured into three phases: dataset deduplication, data preprocessing for code completion, and performance evaluation. Our analysis focuses on various deduplication strategies applied to the CodeParrot model specifically, a model tailored for code generation and comprehension. The evaluation dataset [22], comprises over 2.2 million Python files from 49,562 repositories.

In Figure 3, we provide an overview of the execution pipeline, which integrates multiple stages to ensure a robust methodology for evaluating the impact of code deduplication on model performance. The primary objective is to minimize contamination between the evaluation dataset and the training data of the CodeParrot model[4], thereby ensuring a fair and unbiased assessment of the model's capabilities. The pipeline is divided into three main stages: deduplication, data preprocessing, and performance evaluation. Overall, the pipeline is designed to progressively minimize contamination and provide a comprehensive understanding of how deduplication influences model performance.

First, the deduplication stage addresses the risk of data contamination, which can lead to artificially inflated performance metrics and significantly lowered perplexity as mentioned in Section III-E. The combination of repository-level and file-level deduplication clearly results in improved performance, as outlined in Section III-F. Hence, we begin with repository-level deduplication, where the evaluation dataset is filtered to exclude files originating from repositories already present in the training data. This step is critical because repositories often contain multiple related files, and their inclusion in both datasets could create unintentional overlap. Building on this,

[4] https://huggingface.co/datasets/codeparrot/codeparrot-clean

we perform exact-file deduplication by hashing all code files in both datasets and removing any files in the evaluation set that share identical hash values with files in the training data. This ensures that no identical files are reused, further reducing the risk of contamination. To refine this process, we also conduct near-deduplication on the remaining data. Using MinHash LSH, we explore various parameter configurations to identify the most effective setup for detecting and removing files that are highly similar but not identical. This step ensures that even partially overlapping content, such as slight variations or different versions of the same file, is accounted for. By testing different combinations of MinHash parameters, we aim to strike a balance between eliminating redundant data and retaining sufficient diversity in the evaluation dataset.

After deduplication, the filtered dataset undergoes preprocessing to prepare it for the code completion task. This involves extracting the appropriate data for the model's input pipeline and ensuring it aligns with the specific requirements of the CodeParrot model.

Finally, we evaluate the processed datasets on the CodeParrot model using the perplexity metric. Perplexity measures the model's confidence in predicting the next token in a sequence and provides insights into how well the model performs on a given evaluation dataset. In this study, our goal is to observe how perplexity evolves as we progressively deduplicate the data.

A. Dataset Deduplication

In this section, we delve deeper into the implementation of both exact and near-duplicate detection as part of the deduplication process. Exact deduplication is achieved by computing the SHA256 hash for each Python file, effectively removing identical files.

For near deduplication, we utilize the MinHashLSH algorithm via the *datasketch* Python library[5]. As shown in Figure 1, the process begins by parsing the source code. For file-level deduplication, files are tokenized and represented as n-gram shingles, where the value of n determines the sequence length and granularity of the comparison. For function-level deduplication, individual Python functions are extracted, tokenized, and similarly converted into shingles.

Next, MinHash generates compact hash signatures for each shingle, enabling scalable similarity comparisons. These signatures are indexed using locality-sensitive hashing, which clusters similar shingles into bands, minimizing the computational overhead associated with exhaustive comparisons. Using Jaccard similarity, near-duplicate code fragments are identified by measuring the overlap between shingle sets. The process outputs a mapping of near-duplicate files or function pairs, enabling the removal of redundant data while still preserving meaningful diversity in the dataset.

B. Model Evaluation

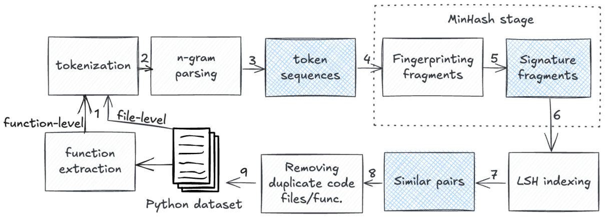

Fig. 1. Near file/function deduplication process based on MinHash LSH. The outcome of each computational stage is depicted in blue boxes, while white boxes represent the implementation.

In Figure 2, we provide an in-depth overview of the evaluation pipeline. The evaluation process is iterative, with the output dataset of each deduplication step serving as the input for the subsequent step. Hence, this ensures that the impact of each deduplication strategy is analyzed in a structured and cumulative manner.

During near file-level deduplication, we explore the effects of varying two key MinHash LSH parameters: the Jaccard similarity threshold and the n-gram size. These variations result in six distinct deduplicated datasets, each representing a unique configuration of the near-deduplication process. For each configuration, we conduct evaluations to identify the dataset that demonstrates the most promising results in terms of deduplication efficiency. Building on the insights gained from this step, we refine the pipeline further by applying function-level near deduplication. The MinHash LSH parameters identified during the file-level analysis are inherited for consistency.

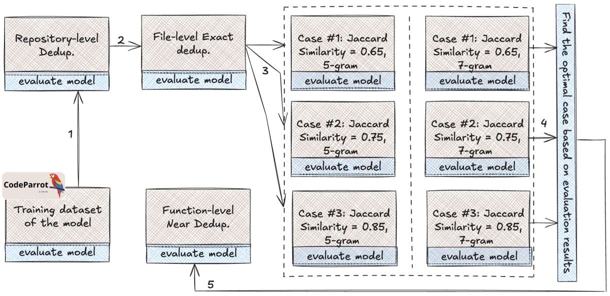
Repository-level Dedup.
evaluate model
2
File-level Exact dedup.
evaluate model
3
Case #1: Jaccard Similarity = 0.65, 5-gram
evaluate model
Case #2: Jaccard Similarity = 0.75, 5-gram
evaluate model
Case #3: Jaccard Similarity = 0.85, 5-gram
evaluate model
Case #1: Jaccard Similarity = 0.65, 7-gram
evaluate model
Case #2: Jaccard Similarity = 0.75, 7-gram
evaluate model
Case #3: Jaccard Similarity = 0.85, 7-gram
evaluate model
4
Find the optimal case based on evaluation results
1
CodeParrot
Training dataset of the model
evaluate model
Function-level Near Dedup.
evaluate model
5

Fig. 2. Evaluation pipeline employed in this work. At each step, we apply the respective deduplication method against the dataset from the previous step, evaluate the model and record the perplexity metric.

V. Experiment Setup

In this section, we outline the detailed configurations used to obtain the results, including the evaluation setup and dataset pre-processing steps.

A. Evaluation Setting and Metrics

In this research, we hypothesize that the presence of duplicate code in the evaluation data with regard to the testing data reduces perplexity, whereas training on deduplicated code results in higher perplexity. This is because the model is exposed to novel problem statements and solutions, which is particularly relevant for code completion tasks. In essence, a lower perplexity value suggests better generalization across

[5] https://github.com/ekzhu/datasketch

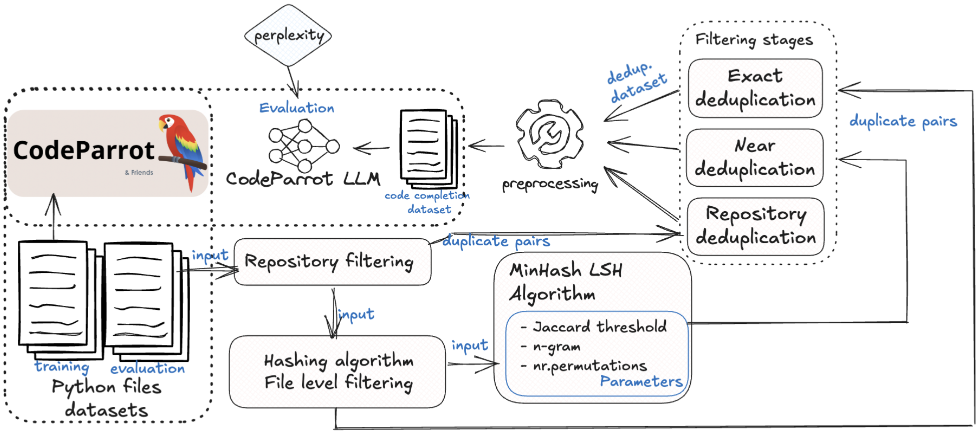

Fig. 3. Overview of the stages in the research approach, illustrating how repository, exact (using hashing) and near (using MinHash LSH) deduplication methods are applied to the evaluation and the CodeParrot's training dataset comprised of Python code, followed by their impact analysis on the perplexity of the CodeParrot model.

diverse scenarios, indicating that the model is less "surprised" by its predictions. This naturally means that we aim to record higher perplexity on deduplicated datasets and lower on duplicated data, which we measure with Eq. 2.

$$PPL = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log P(w_i \mid w_{<i})\right) \tag{2}$$

Here N refers to the total number of available tokens, where w_i is the i-th token in the sequence, and $P(w_i \mid w_{<i})$ as the probability of the i-th token given the sequence of tokens leading up to it.

While perplexity is our primary evaluation metric, we also considered alternative metrics such as ROUGE similarity, edit distance, and BLEU score based on [23]. ROUGE similarity measures semantic similarity between code snippets, which could provide a deeper understanding of redundancy beyond syntactic duplication. Additionally, BLEU scores, commonly used in natural language processing, were deemed less relevant because they prioritize surface-level token overlaps rather than evaluating the model's ability to generate semantically meaningful code completions. These metrics were not included in our final evaluation framework due to computational overhead, lack of relevancy, and limited alignment with the specific objectives of code completion tasks.

B. Configuration and Implementation Details

1) Hardware setup: The evaluations of the deduplicated datasets were conducted on the DelftBlue high-performance computing cluster [24]. Each evaluation used a single NVIDIA A100 GPU with 80 GB of video memory and 8 CPU cores, each with 7 GB of RAM. These configurations were necessary because of the size of both the CodeParrot model and the datasets. We initially deduplicated and preprocessed the datasets using only CPUs, and then we used the GPU to evaluate the tokenized datasets. This setup facilitated us to parallelize the evaluations of the datasets, ensuring the experiment was completed within our time constraints.

2) Raw Dataset: As mentioned before in Section IV, the raw dataset [22] consists of 2.2 million Python files that were scraped from Github.

3) Deduplication: This subsection describes the implementation details. For a high-level overview, refer to section IV.

For repository-level deduplication, we removed the repositories in our raw dataset contained in the CodeParrot training dataset. Specifically, we compared the 'repo_name' column in our dataset with the one in CodeParrot. If a match was found, the corresponding repository was removed from our dataset. This refined dataset was then used in the next step.

For file-level deduplication, the refined dataset and the CodeParrot dataset were used in this stage. The 'content' column in both datasets contains the file data. These file contents were hashed using *hash_256* and the *hashlib* module[6], and then stored in distinct sets, ensuring each file's content is uniquely distinguishable. The hashes in the refined dataset that were also present in the CodeParrot were filtered out, and thus, only the unique files of the refined dataset were kept. The final dataset consists of unique files that are **not present word-for-word** in the CodeParrot dataset.

For near-deduplication, various experiments were carried out as shown in Figure 2. For file-level near deduplication, we tested 50 permutations, 5- and 7-gram, and Jaccard similarity thresholds of 0.65, 0.75, and 0.85. The optimal case was then selected for the function-level near-deduplication evaluation. For near-deduplicate, the files were firstly tokenized into n-grams (using different n-gram sizes for each experiment) and converted into unique sets. Each set was then processed using the MinHash technique with 50 permutations, generating a signature of length 50 for each n-gram set. After generating the hashes, we created an LSH object using 50 permutations and a different threshold for Jaccard similarity based on the experiment. We stored all dissimilar functions and files in a dictionary, which was transformed into a database.

Function-level near-deduplication follows the same approach, with the only exception that instead of performing shingling on the files, we do it on the functions contained in each file. The functions are extracted using the Abstract Syntax Tree (AST) module[7] which extracts functions from

[6] https://docs.python.org/3/library/hashlib.html
[7] https://docs.python.org/3/library/ast.html

files and returns them as tuples in the format (function_name, function_code)

4) Dataset preprocessing: For dataset preprocessing, the collected Python code was reformatted into function completion tasks to align with the requirements of code completion based on a similar paper [25]. Using Tree-sitter [8], each file was parsed to identify all functions. From these, functions containing a docstring were randomly selected. The portion of the file up to the end of the docstring was used as the context, while the corresponding function body served as the groundtruth. The remaining parts of the file were discarded to focus solely on meaningful completion tasks. To minimize preprocessing time on the already large dataset, a maximum of three functions were extracted per class. During evaluation, the context was provided as input to the model, prompting it to generate the function body. Functions with docstrings were specifically chosen to ensure that the context was well-defined and supported the inference. Additionally, the dataset was filtered to include contexts between 64 and 768 tokens in length, with ground-truths shorter than 256 tokens, to match the model's generation capabilities.

5) Evaluation: To evaluate the impact of dataset deduplication on the model's performance, we utilize a systematic approach centered on perplexity as the primary metric. The evaluation pipeline is implemented using the Hugging Face *transformers* library[9], which offers tools to streamline the evaluation process of pre-trained language models. Needless to say, the pre-trained CodeParrot model is used throughout all experiments. The tokenized evaluation dataset is processed in full, without truncation, to ensure comprehensive assessment across all examples. To focus purely on inference, gradient updates are disabled, and only the prediction loss is computed during evaluation. This loss is subsequently used to calculate perplexity, which provides insight into how well the model generalizes to deduplicated datasets.

These configurations, such as consistent tokenization, dataset splitting, and evaluation settings, are maintained throughout all experiments to ensure a fair comparison of the model's performance across different deduplication strategies and parameter settings.

VI. Results

This section highlights the results obtained by running the experiments as described in Sections IV and V. As previously stated, we consider different thresholds for the Jaccard similarity and observe their impact on the model performance, as well as the difference between using 5-gram and 7-gram when considering near-file deduplication. In Tables VI and VI, observe a trend between the rigour of each deduplication level and the perplexity score, where having a higher Jaccard similarity threshold, i.e. allowing for more similarity, leads to high perplexity scores. The results suggest that there is a negative correlation between the size of the dataset and the perplexity score of the model. As mentioned in previous sections, we assume that lower perplexity scores are an indicator of the model being trained on previously seen data, in other words, the original training set was contaminated. In contrast to just the raw size in gigabytes, we also take into account the number of rows after each deduplication stage. We observe a significant difference in the number of rows compared to the original dataset, and the dataset obtained after applying near-file deduplication with the similarity threshold $\theta = 0.65$. This further reinforces our hypothesis that the dataset used for training the baseline model was contaminated. In Table VI it is also suggested that even for simple repository and exact-file deduplication strategies, there is a significant decrease in the number of rows left, while the perplexity score increases.

Lastly, we would like to emphasize that the Jaccard similarity threshold $\theta = 0.65$ is used for function-level near-deduplication as it was previously observed to give optimal results, which was the proposed approach in Figure 2.

VII. Discussion

As we progress through our evaluation pipeline, we observe a linear increase in perplexity compared to the baseline results. For example, the baseline perplexity of 4.99 (i.e., original dataset without deduplication) rises to a maximum of 6.57 after applying near-function-level deduplication, highlighting a noticeable difference. Additionally, the dataset size steadily decreases, reinforcing the argument that both evaluation and training datasets exhibit a degree of contamination. Specifically, repository-level and exact file-level deduplication via hashing account for a 29.7% reduction in contaminated code, while near deduplication achieves an additional 32.5% reduction.

Among the near-deduplication configurations, a Jaccard threshold of 0.65 with a 5-gram parameter resulted in the highest perplexity, whereas a threshold of 0.75 produced the lowest, even lower than 0.85. We hypothesize that smaller thresholds increase the risk of removing false positives alongside true duplicates, while higher thresholds focus on strict duplicates but might overlook false negatives. Overall, a 0.75 threshold appears to strike a balance by preserving meaningful diversity and eliminating redundancy, aligning with findings from Allal et al. [19]. In contrast, the 7-gram configuration resulted in different dynamics of perplexity among the Jaccard thresholds. For instance, a 0.65 Jaccard threshold is less aggressive on 7-gram than on 5-gram parameter because as the sequence size increases, the number of similar pairs decreases. However, that cannot be said with regard to the 0.75 parameter. Thus, no explicit causation can be established for the variations observed across different MinHash LSH parameters. We recommend that future work explore a broader range of parameters while evaluating their effectiveness in identifying true duplicates. This could be facilitated by using a dataset that accurately tracks true positives. Lastly, we notice a significant decrease in dataset size after near function level deduplication, namely 63.8% which we advise against this technique as this might have resulted in removing an excessive number of false

[8] https://tree-sitter.github.io/tree-sitter/

[9] https://huggingface.co/docs/transformers/index

Metric	Baseline	Repo-level	Exact-file
Perplexity	4.99	5.13	5.86
Size (GB)	12.4	11.5	10.7
Row Count	2,290,182	1,779,816	1,609,841

Fig. 4. Comparison of data across Baseline and different levels of deduplication: Repository-level (Repo-level), and Exact-file on Perplexity, Size (GB), and Row Count

Metric	Near-file (5-gram)			Near-file (7-gram)			Near-func
Parameters	$\theta = 0.65$	$\theta = 0.75$	$\theta = 0.85$	$\theta = 0.65$	$\theta = 0.75$	$\theta = 0.85$	$\theta = 0.65$
Perplexity	6.56	6.13	6.35	6.31	6.36	6.24	6.57
Size (GB)	6.56	6.76	6.99	6.61	6.82	7.03	0.85
Row Count	1,086,339	1,138,289	1,190,453	1,118,052	1,161,319	1,205,940	693,090

Fig. 5. Comparison of dataset across varying Jaccard similarity thresholds (θ) for different deduplication strategies: Near-file using 5-gram and 7-gram, and Near-function deduplication. Metrics are reported for Perplexity, Size (GB), and Row Count

positives or we could argue that some repetitive functions in fact can provide more context to the code files especially for code completion tasks. Overall, we provide the following answers to the stated **RQs**:

RQ1: *To what extent do different code deduplication methods affect the performance of an LLM on the code completion task?*

Answer: Each deduplication technique increases the perplexity of the model which supports the contamination claim. We notice that repository-level and exact file-level deduplication lead to a significant dataset reduction whilst near-deduplication further filters out syntactically similar code. Moreover, near-function-level deduplication drastically reduces dataset size, potentially harming its versatility.

RQ2: *How do different MinLSH parameters affect the performance of an LLM on the code completion task?*

Answer: Overall, different MinHash parameters increase the perplexity of the model. A Jaccard threshold of 0.75 strikes a balance by preserving diversity while removing code clones, whereas low thresholds risk removing false positives and higher thresholds may overlook duplications. Additionally, we did not notice a considerable difference on variations of n-gram values.

A. Implications

The implications of these findings suggest that methods which operate on more precise scales result in higher levels of deduplication. Repository level results in a smaller perplexity increase than the exact file-level, and functions which remove near similarities on a file level result in higher levels of perplexity. Researchers may make use of the results found to make more informed decisions on looking into which deduplication methods would fit them best, or to look into the best thresholds to use. Along with this, they can find evidence that simply decreasing the threshold beyond a certain point may not improve the performance of models, and as such will still need to look to find the optimal value.

From the given results, we can see that the found perplexity increases as we apply deduplication techniques to the initial dataset. This gives clear evidence of contamination in our gathered dataset, and that these techniques are successful in removing part of it. In the wider field, data contamination can be found in all non curated datasets and should be cleaned using one of these techniques. Models that have been trained and evaluated using the contaminated data should be re-evaluated to see if the reported performance is being inflated by the contamination.

At the scope of the field, comparing the effectiveness of multiple different deduplication methods against each other is relatively sparse, however there are several instances of research done on the effectiveness on individual 'levels' such as from Lee et al [15], Allmanis [13] and Allal et al [19]. There have been studies done into comparing repository level vs exact file-level deduplication from Huang et al [21], but our research compares these methods effectiveness against near file and function level deduplication techniques. Along with this, there is relatively little research currently done into near function level deduplication, and if it is a viable form of deduplication going forward. This research fills in a potential gap in research by displaying a clear comparison of performance between each method and which is more effective.

Going forward, data contamination should be taken more seriously when making use of large data sets to train large models. It is present in large amounts in large datasets, and the results of figure VI show that for near file deduplication even with a Jaccard threshold of 0.75 roughly half the files were removed. This can decrease the amount of duplicated data being run in the model, along with allowing for more accurate evaluation [13], as the models will not be tested on the same data that they were trained on. It also gives the potential for more efficient training of LLMs, as this removes a significant amount of data which the model needs to be trained on without losing valuable information.

B. Threats to Validity

One internal threat to validity originates from the choice of the CodeParrot model for the experiments. We selected this smaller, GPT-2-based model (1.5B paremeters) due to the limited project context. However, in the last couple of years more powerful models capable of generating code in various programming languages have emerged. Although we assume that the deduplication techniques discussed here can apply to other models when considering different programming languages or code tasks altogether, a systematic evaluation is needed to assess the full extent of generalization.

Building on this, another threat is that our experiment is not generalizable since we evaluated code completion tasks using only one LLM (i.e. CodeParrot) on one dataset consisting of only Python files. Due to time constraints, we had to limit the scope of the experiment, but this could affect the reliability of our results in other programming languages, code tasks, LLMs, or datasets. Further investigation is necessary to confirm our results with different LLMs, datasets, or code tasks.

As we mentioned in Section V-B4, we only extracted a maximum of three functions from each file in the datasets due to lack of time. This limited number of functions could not be sufficient to fully represent the characteristics of datasets, therefore impacting the reliability of our results. Extracting a bigger amount of functions might lead to more dependable results which capture better the datasets.

Because of limited time, the near-deduplicated 5-gram datasets were split into three batches and evaluated in parallel. Then, the results were averaged out to get the perplexity for the whole dataset. While this approach helped to reduce the evaluation time of the datasets, averaging the results could have led to an incomplete representation of the dataset since some of the batches could have a more biased evaluation. Some batches may have produced more skewed results than the other and averaging out the results could have smoothed out these characteristics and not fully represented the dataset's performance. Thus, this could affect the accuracy and generalisability of our results.

C. Future Work

This study examines the impact of repository-level, exact file-level, and near deduplication on the performance of the CodeParrot LLM in code completion. However, several promising areas remain for future exploration, including semantic deduplication, identifying optimal candidates for duplicate code, and applying safety fine-tuning. These directions could build on our findings to broaden understanding and improve the utility of deduplication in LLMs for code.

1) Semantic Deduplication: Unlike exact or near-duplicate detection, which relies on syntactic similarity, semantic deduplication aims to identify code fragments that are functionally equivalent but differ in structure or syntax. This approach could help uncover latent redundancies that traditional methods miss. For instance, semantic deduplication can leverage techniques such as program analysis, graph-based representations, or advanced neural models to identify equivalences at a deeper level.

2) Good Code Duplication: It would be also interesting to look into potential ideas on how code duplication can be leveraged to increase or enrich the training data. For instance, building on the work of Hashimoto et al. [26], combining retrieval methods to find similar code snippets, using semantic deduplication for instance, and perform edits over those examples which then can be reused as evaluation or training data is a promising direction. This approach aligns with the idea of not just filtering out duplicates but also understanding how they can be transformed or utilized to enhance model training and downstream tasks.

Inspired by observations from Kapser et al. [27], future work could explore how to embrace certain types of code clones rather than eliminating them entirely. Code clones are not inherently detrimental, in fact, they often provide developers with flexibility in project evolution. A nuanced approach to deduplication that distinguishes between harmful and beneficial duplicates could open new directions for improving dataset quality without sacrificing diversity. An intriguing question raised in the literature is: "*Can we usefully exploit near-duplicates to produce better software engineering tools?*". Instead of treating near-duplicates solely as artifacts to be removed, future studies could investigate how these instances might inform tasks such as bug detection, refactoring recommendations, or code completion. For example, near-duplicates could serve as valuable training examples for models focused on generating contextually appropriate variations of code.

3) Safety Fine-Tuning: We also propose exploring safety fine-tuning strategies, such as adapting the ForgetFilter algorithm [28], to address malicious code duplicates in LLM training data. This could reduce bias in LLM output while maintaining performance in tasks like code completion. However, methods like ForgetFilter have limitations and can be bypassed using adversarial techniques [29]. Researchers have even developed an algorithm called Prompt Automatic Iterative Refinement (PAIR) [30], which uses another LLM to automatically generate prompts that can bypass the security mechanisms of a target LLM, despite having only black-box access. To that end, RMCBench [31] assesses LLMs' resilience against generating malicious code. Similarly, the CodeSecEval [32] dataset evaluates the security-awareness of code LLMs in tasks like code repair and generation. Both are valuable, recently published benchmarks for assessing security-related LLM performance. We suggest future work to explore variations of the research questions outlined in Section I-B, such as: "*How do different code deduplication methods impact an LLM's ranking on RMCBench and CodeSecEval?*".

VIII. Conclusion

In this work, we evaluated both exact and near-deduplication across various levels: function, file, and repository. Although these techniques have been explored in other contexts, this study is one of the first to compare LLM performance, specifically CodeParrot with 1.5B parameters, in code completion under different conditions, including n-grams, Jaccard similarity thresholds (θ), and the specified code levels.

Our results show that as we dive deeper into the code structure during deduplication, i.e. from the repository level down to the function level, CodeParrot achieves higher perplexity. This suggests that deduplication efforts prove data contamination in the training set, therefore skewing the evaluation of genuine LLM performance when deduplication is not applied. For optimal results, we recommend using $\theta = 0.75$, as it provides a balanced approach between removing true duplicates and minimizing false positives. Additionally, while we experimented with different n-gram windows, this did not yield any significant effects.

Finally, although our evaluation was relatively limited in scope, we are intrigued by potential alternative techniques and extensions of our work. In particular, semantic deduplication and the possibility of retaining and leveraging some duplicate code in the training data could prove beneficial. While LLMs for code tasks are valuable, they still continue to face several challenges, such as bias mitigation, performance enhancement, providing explanations for their suggestions, and ensuring privacy and security. While each of these issues deserves its own investigation, they are arguably connected with deduplication and merit a holistic approach, examining them in conjunction.

References

[1] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," 2024. [Online]. Available: https://arxiv.org/abs/2406.00515

[2] J. Wang and Y. Chen, "A review on code generation with llms: Application and evaluation," in *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, 2023, pp. 284–289.

[3] N. Raihan, C. Newman, and M. Zampieri, "Code llms: A taxonomy-based survey," 2024. [Online]. Available: https://arxiv.org/abs/2412.08291

[4] M. Gharehyazie, B. Ray, M. Keshani, M. Zavosht, A. Heydarnoori, and V. Filkov, "Cross-project code clones in github," *Empirical Software Engineering*, vol. 24, 06 2019.

[5] Z. Yang, Z. Zhao, C. Wang, J. Shi, D. Kim, D. Han, and D. Lo, "Unveiling memorization in code models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. ACM, Apr. 2024, p. 1–13. [Online]. Available: http://dx.doi.org/10.1145/3597503.3639074

[6] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[7] M. U. Hadi, R. Qureshi, A. Shah, M. Irfan, A. Zafar, M. B. Shaikh, N. Akhtar, J. Wu, S. Mirjalili *et al.*, "A survey on large language models: Applications, challenges, limitations, and practical usage," *Authorea Preprints*, 2023.

[8] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," 2023. [Online]. Available: https://arxiv.org/abs/2305.01210

[9] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019. [Online]. Available: https://arxiv.org/abs/1810.04805

[10] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[11] N. Kandpal, E. Wallace, and C. Raffel, "Deduplicating training data mitigates privacy risks in language models," in *Proceedings of the 39th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, Eds., vol. 162. PMLR, 17–23 Jul 2022, pp. 10 697–10 707. [Online]. Available: https://proceedings.mlr.press/v162/kandpal22a.html

[12] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica, "Livecodebench: Holistic and contamination free evaluation of large language models for code," 2024. [Online]. Available: https://arxiv.org/abs/2403.07974

[13] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," 2019. [Online]. Available: https://arxiv.org/abs/1812.06469

[14] D. Hernandez, T. Brown, T. Conerly, N. DasSarma, D. Drain, S. El-Showk, N. Elhage, Z. Hatfield-Dodds, T. Henighan, T. Hume, S. Johnston, B. Mann, C. Olah, C. Olsson, D. Amodei, N. Joseph, J. Kaplan, and S. McCandlish, "Scaling laws and interpretability of learning from repeated data," 2022. [Online]. Available: https://arxiv.org/abs/2205.10487

[15] K. Lee, D. Ippolito, A. Nystrom, C. Zhang, D. Eck, C. Callison-Burch, and N. Carlini, "Deduplicating training data makes language models better," 2022. [Online]. Available: https://arxiv.org/abs/2107.06499

[16] D. Kocetkov, R. Li, L. B. Allal, J. Li, C. Mou, C. M. Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, D. Bahdanau, L. von Werra, and H. de Vries, "The stack: 3 tb of permissively licensed source code," 2022. [Online]. Available: https://arxiv.org/abs/2211.15533

[17] E. Zhu, F. Nargesian, K. Q. Pu, and R. J. Miller, "Lsh ensemble: Internet-scale domain search," 2016. [Online]. Available: https://arxiv.org/abs/1603.07410

[18] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of Massive Datasets*, 2nd ed. USA: Cambridge University Press, 2014.

[19] L. B. Allal, R. Li, D. Kocetkov, C. Mou, C. Akiki, C. M. Ferrandis, N. Muennighoff, M. Mishra, A. Gu, M. Dey, L. K. Umapathi, C. J. Anderson, Y. Zi, J. L. Poirier, H. Schoelkopf, S. Troshin, D. Abulkhanov, M. Romero, M. Lappert, F. D. Toni, B. G. del Río, Q. Liu, S. Bose, U. Bhattacharyya, T. Y. Zhuo, I. Yu, P. Villegas, M. Zocca, S. Mangrulkar, D. Lansky, H. Nguyen, D. Contractor, L. Villa, J. Li, D. Bahdanau, Y. Jernite, S. Hughes, D. Fried, A. Guha, H. de Vries, and L. von Werra, "Santacoder: don't reach for the stars!" 2023. [Online]. Available: https://arxiv.org/abs/2301.03988

[20] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "Deepseek-coder: When the large language model meets programming – the rise of code intelligence," 2024. [Online]. Available: https://arxiv.org/abs/2401.14196

[21] S. Huang, T. Cheng, J. K. Liu, J. Hao, L. Song, Y. Xu, J. Yang, J. H. Liu, C. Zhang, L. Chai, R. Yuan, Z. Zhang, J. Fu, Q. Liu, G. Zhang, Z. Wang, Y. Qi, Y. Xu, and W. Chu, "Opencoder: The open cookbook for top-tier code large language models," 2024. [Online]. Available: https://arxiv.org/abs/2411.04905

[22] Razvan27/ml4se-python · datasets at hugging face. [Online]. Available: https://huggingface.co/datasets/Razvan27/ML4SE-Python

[23] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," 2020. [Online]. Available: https://arxiv.org/abs/2005.08025

[24] Delft High Performance Computing Centre (DHPC), "DelftBlue Supercomputer (Phase 2)," https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2, 2024.

[25] H. Ding, V. Kumar, Y. Tian, Z. Wang, R. Kwiatkowski, X. Li, M. K. Ramanathan, B. Ray, P. Bhatia, S. Sengupta, D. Roth, and B. Xiang, "A static evaluation of code completion by large language models," 2023. [Online]. Available: https://arxiv.org/abs/2306.03203

[26] T. B. Hashimoto, K. Guu, Y. Oren, and P. Liang, "A retrieve-and-edit framework for predicting structured outputs," 2018. [Online]. Available: https://arxiv.org/abs/1812.01194

[27] C. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful," in *2006 13th Working Conference on Reverse Engineering*, 2006, pp. 19–28.

[28] J. Zhao, Z. Deng, D. Madras, J. Zou, and M. Ren, "Learning and forgetting unsafe examples in large language models," 2024. [Online]. Available: https://arxiv.org/abs/2312.12736

[29] S. Jain, E. S. Lubana, K. Oksuz, T. Joy, P. H. S. Torr, A. Sanyal, and P. K. Dokania, "What makes and breaks safety fine-tuning? a mechanistic study," 2024. [Online]. Available: https://arxiv.org/abs/2407.10264

[30] P. Chao, A. Robey, E. Dobriban, H. Hassani, G. J. Pappas, and E. Wong, "Jailbreaking black box large language models in twenty queries," 2024. [Online]. Available: https://arxiv.org/abs/2310.08419

[31] J. Chen, Q. Zhong, Y. Wang, K. Ning, Y. Liu, Z. Xu, Z. Zhao, T. Chen, and Z. Zheng, "Rmcbench: Benchmarking large language models' resistance to malicious code," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 995–1006. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1145/3691620.3695480

[32] J. Wang, X. Luo, L. Cao, H. He, H. Huang, J. Xie, A. Jatowt, and Y. Cai, "Is your ai-generated code really safe? evaluating large language models on secure code generation with codeseceval," 2024. [Online]. Available: https://arxiv.org/abs/2407.02395

IX. Appendix: Task Assignment Sheet

Alexandru-Radu Moraru:

- Wrote the abstract, introduction, problem statement and motivation, and conclusion sections.
- Wrote "*Overview of LLMs*", "*Pretraining and Fine-Tuning in LLMs*" subsections, along with the initial part of the "*Threats to Validity*" and part of the "*Future Work*" subsection.
- General rewriting and integrating Peer feedback.
- Coded the evaluation setup, alongside its corresponding scripts and guide to running it, either locally or on DelftBlue.
- General code refactoring, including improving code quality and adding user guides.

Aiman Abdul Wahab

- Wrote '*deduplication techniques*' in the background section.
- Wrote '*3) Deduplication*' in 'section: V Experiment Setup'
- Wrote the code for near-deduplication at file level and function level.
- Ran near-deduplication at file-level for 5 and 7-gram with different jaccard threshholds.
- Ran near-deduplication at function level for 5-gram.

Ana Terna

- In the "*Background*" section wrote "*Data Contamination*", and "*The effect of code duplication*".
- Wrote the Approach section.
- In Experiment Setup wrote Evaluation Setting and Metrics, and in Configuration and Implementation Details wrote Dataset preprocessing and Evaluation.
- Wrote also the discussion of results and answers to RQs and part of Future Work.
- For code, implemented the data preprocessing stage and part of the evaluation pipeline.

Dean Polimac

- Wrote the "*Research Question*" Section for the midterm report, which was later merged with the "*Introduction*" Section.
- Wrote the "*Result*" section.
- Researched & implemented the evaluation metrics in the pipeline.
- Provided general feedback on parts of the report, mainly the "*Experiment Setup*" Section.

Floris van Veen

- One of the members that worked on background and related work, added much of the information on deduplication effects. Slightly edited/condensed LLM section
- "*Implications*" subsection of the report.
- Wrote the Data scraper code.
- Worked on the deduplication methods in the code. Primary section made was repo level dedup but helped in the other sections as well with debugging, etc. Also cleaned up the deduplication folders in general + added repo readme instructions, requirements, etc.
- Made scripts to run and save relevant data

Nada Mouman

- Wrote the introduction for the midterm report.
- Wrote Threats of Validity and Hardware Setup sections.
- Implemented exact-dedup at file level
- Ran the job to create the exact-deduped dataset at file-level.
- Wrote script for running jobs in DelftBlue
- Created setup for the local environment to run jobs in DelftBlue.
- Debugged and fixed preprocessing code to include a maximum number of functions to be extracted
- Ran jobs to preprocess and evaluate the following datasets: baseline, repo-level, exact-file dedup, near-file dedup (7-gram).

Zuji Zhou

- One of the members that worked on background and related work, added most of the info on LLM's and their relevant coding tasks + pretraining.
- Implemented exact-dedup at function level
- Helped with implementing near-dedup
- Helped improving preprocessing step
- Ran jobs to get results of deduplicated dataset & evaluations.