UTRECHT UNIVERSITY

Department of Information and Computing Science

Artificial Intelligence Master Thesis

Code Generation on a Diet: A Comparative Evaluation of Low-Parameter Large Language Models

First examiner:

Gatt, Albert

Candidate:

Carboni, Leonardo

Second examiner:

Dalpiaz, Fabiano

June 29, 2024

Abstract

In the constantly evolving field of software development, the demand for automated code generation has significantly increased since the release of AI based tools like ChatGPT and GitHub Copilot. These tools, powered by Large Language Models (LLMs), typically require server requests due to their closed source nature and substantial computational costs. This thesis investigates the potential of smaller, locally runnable low-parameter LLMs in the context of code generation. The research begins with an overview of the state of the art of coding and its anticipated evolution thanks to AI integration. It continues by analyzing the current landscape of LLMs by explaining the underlying mechanisms of these models and listing several of the most important low-parameter models, such as Mistral, CodeLlama and DeepSeek-Coder. The study also examines the impact of techniques like fine-tuning, instruction-tuning and quantization on improving performance and efficiency. Additionally, it reviews the available code evaluation techniques, focusing on match-based and functional metrics, and discusses the datasets used to evaluate the models. The methodology employed involves selecting suitable datasets and models, generating code samples, and evaluating them using both types of metrics. The evaluation highlights the limitations of match based metrics in capturing the LLM's true code generation performance and emphasizes the importance of functional metrics like pass rates. The findings indicate that while larger models generally outperform smaller ones, the performance gap is narrowing, thanks to higher quality and more domain-specific training data. Moreover the study confirms the effectiveness of the aforementioned fine tuning and quantization techniques in improving the model's capabilities and lowering the requirements needed to run them. The thesis concludes by suggesting that with continuous advancements, smaller models could play a crucial role in making high quality code generation more accessible and sustainable.

Contents

Chapter 1

Motivation Behind the Research

In the constantly evolving field of software development, the demand for automated code generation has grown exponentially since the public release of highly capable large language models and tools like ChatGPT and GitHub Copilot. Some of these models, however, are only accessible via APIs because of their closed source nature and computational costs. More in detail, OpenAI's flagship model, GPT4, reaches a summed parameter count of around 1.76 trillion, spread across its 8 sub-models connected by a Mixture of Experts architecture, according to an industry leak [59]. This research aims to discover if and how smaller models can generate quality code on par with the best available LLMs.

1.1 LLMs are more used than Stack Overflow

It is undeniable that, since the release of Copilot and ChatGPT, artificial intelligence has become one of the most useful tools in the software development community [1][21]. In this field, it has always been common to take inspiration from someone else's code via GitHub, Stack Overflow or whichever blog about computer science. However, the code one can find online would need to be adapted to the developer's use case in the majority of occasions. On the contrary, feeding LLMs the right prompt and enough context, like the rest of the code, or a good and detailed explanation of what we need to implement and how, allows the system to give the

right solution in a good percentage of cases. This allows for a faster development of systems, and, in some cases, even a better quality of the code. Moreover, users can interact with LLMs by asking the AI to tweak parts of the generated code or to explain how it works, without the need to wait for an answer or the inconvenience of going through a list of similar threads until the doubts have been cleared up.

1.2 The advantages of smaller models

It is generally safe to assume that larger models tend to outperform smaller ones, a conclusion that can be reached without extensive technical knowledge. This assertion is supported by researches on scaling laws such as the study conducted by Kaplan et al. (2020) [34], which provides empirical evidence of the performance benefits associated with larger models.

The landscape of large language models currently spans a broad and rapidly expanding spectrum, including both open-source and proprietary models of varying sizes and architectures [79]. Leading the industry are major corporations like Google, Meta, and the collaboration between OpenAI and Microsoft engaged in a competitive effort to develop the most accurate models possible, alongside emerging startups that often focus on more specific sectors like efficiency or multilingualism. OpenAI's GPT-4 [47] is a proprietary model with a high parameter count, accessible through its API or a subscription-based chat service. Google's latest high-performing model, Gemini [63], also closed-source, is available in various configurations named Nano, Pro, and Ultra, with the latter achieving parity with OpenAI's flagship model across numerous tasks. Conversely, Meta's LLaMa (all the different versions) [65] [2] represents an open-source and permissively licensed LLM available in different sizes (7, 34, 70 and 400 billion parameters). LLaMa's approach is different from its counterparts because, despite not leading in performance metrics, it is highly regarded within the open-source community for encouraging the development of specialized adaptations that contribute significantly to AI research advancements. Overall, the models proposed by both big tech and smaller companies

span from fewer than 10 billion parameters to several hundred billion parameters. A very comprehensive analysis of the LLM landscape is presented in a 2023 survey by Zhao et al. [79].

Nevertheless, smaller models, thanks to their open-source nature and the support from the developer community, have made significant advancements in every field, suggesting that they are on a path to becoming a relevant part of the market too, leveraging their key strengths to close the gap with larger models. Additionally, on top of the architectural improvements that brought small models into practical use, research introduced the concept of quantization, further simplifying the model inference via the approximation of the LLM's weights, reducing the computational requirements for running models on systems with limited performance capabilities [13].

Locally runnable models, despite not being as reliable as top level paid ones, present quite a list of advantages:

- **Prices**: There is no fee for API calls because the smaller versions can operate on desktop computers and laptops with medium-to-high performance. As a result, they are easier to access for both businesses and individuals.
- **Control**: Low parameter models can be easily and affordably fine-tuned by the user or the enterprise for certain tasks or coding styles. A software development business, for instance, might employ a custom model to adhere to their own production code style or use an internal library. On a more sensitive note, the models can also be adjusted with uncensored data; this eliminates the need to adhere to the safety norms that a publicly traded company must follow.
- **Privacy**: Running a model locally means that the interaction history between LLM and user are in full control of the user itself and cannot be used in any way by third parties for analytical purpose.
- **Environmental Sustainability**: Less computing power required means less energy spent, which translates to less carbon emissions [19] than

the ones of high parameters LLMs that run on often coal-powered [62] data centers.

This research focuses on exploring the available open source models with smaller parameters counts, that offer a good performance for little to no cost, investigating the methods in which generated code can be evaluated and understanding which Large Language Model is best at solving coding tasks.

1.3 Research Questions

The research questions that this thesis aims to answer are:

RQ1. How do low-parameter large language models' coding capabilities compare to the ones of bigger models?

RQ2. What code evaluation methods can be used to enable comparative analysis?

RQ3. Which small LLM is the overall best at coding?

RQ4. How much does quantization influence the quality of the generated code?

Chapter 2

Current Landscape of Large Language Models for Code Generation

This section contains all the primary concepts that are explored in this research. First of all, an analysis of the state of the art of coding and why AI is always going to be more relevant in the field. Secondly, it contains an overview of the main LLMs that will be examined, the fine tuning and quantization techniques used to improve performance and reduce computational requirements of the LLMs. Finally, it discusses the available metrics that will be taken into consideration and the open source code completion datasets currently used to evaluate the models.

2.1 Why coding is relevant

In the constantly evolving field of technology, coding remains one of the most fundamental skills, shaping nearly every kind of industry. The programming task, that initially contemplated the use of punching cards and was limited to a small number of very specialized scientists, has seen an exponential growth after the creation of computers, that made it more accessible via the invention of programming languages. The first languages that came to life in the 1950s and 1960s were all developed to solve scien-

tific, mathematical, engineering and business related problems. After the 1970s, with the invention of languages such as C, programming became useful for a potentially infinite number of tasks, like the creation of operating system and applications. Every programming language has to follow a particular logic, which defines what and how different ideas can be implemented. This means that programming languages can be placed on a scale that defines the level of abstraction that a language can achieve. Lower level languages, like Assembly, C and C++ make use of logic that is as close as possible to the machine language (which is the most basic set of instruction that the computer can execute) and revolves around basic operations regarding arithmetic and memory management. High level languages, on the other side, are more comprehensible to humans and allow for the implementation of different programming paradigms like Object Oriented Programming. OOP has opened the door to reusable code, allowing programmers to develop more complicated systems that also made use of interactive graphical user interfaces. Some languages, however, even if high level, are structured differently than others, due to their scripting nature. A scripting language, like Python or Perl can be used to automate repetitive tasks and run autonomous systems.

2.1.1 The new coding era

The programming landscape has undergone a transformation with recently introduced technologies that enhance productivity and accessibility. Two pivotal developments are chatbot interactions and code completion tools, both leveraging artificial intelligence to change how programmers interact with code.

AI chatbots have introduced a new layer of functionality and convenience, with their ability to translate natural language to code. Giving detailed instructions to tools such as ChatGPT, it is possible to obtain a snippet of code in almost every existing programming language, that attempts to solve the specified problem. This application helps programmers to quickly prototype ideas, explore solutions, and learn new programming

concepts without having to write code from scratch.

Code Completion tools, like GitHub Copilot [22], offer contextual code suggestions, predicting the following line(s) of code that the programmer would need to write by hand. Thus this tools are not that suitable for problem solving, or at least not like chatbots can be, their main advantage is the performance increase the developer gains by not having to rewrite repetitive or boilerplate code.

Other AI applications not explored in this research touch topics like:

- **Code Optimization**: AI can analyze code and suggest optimizations for performance, memory usage, or readability. For example, given a function in python that makes use of for loops and matrix multiplications, a well trained LLM could suggest the use of optimized methods from specific libraries like `numpy`, or refactor the code to take advantage of list comprehension, a python feature that allows for faster list population. The study from Wu et al. [72], explains how, thanks to Reinforcement Learning with Human Feedback, their agent CodeZero is able to rewrite code to minimize compiler instructions.
- **Bug Detection**: Machine learning models can be trained to detect and fix bugs in code by analyzing patterns from existing codebases. This means that the LLM could notify the developer regarding potential flaws like the possibility of division by 0 or the absence of exception handling.
- **Code Translation**: LLMs can translate code from one programming language to another, facilitating cross-platform development and legacy code migration.
- **Code Refactoring**: AI can assist in refactoring code by identifying opportunities for improvement and suggesting changes to enhance code quality and maintainability. For instance, it could help prevent errors related to variables sharing the same name and improve readability by extracting repeated or similar code parts into a single function.

- **Code Documentation**: Language Models can generate documentation for code by analyzing its structure and functionality, reducing the manual effort required. These tools have been introduced well before the possibilities to implement them became available [56] and are now widely diffused in the developer community.
- **Test Case Generation**: AI can generate test cases for software based on the requirements, reducing the need for manual test design and ultimately delivering more robust software.
- **Test Automation**: AI can automate various testing tasks, such as UI testing, load testing, and regression testing, improving test coverage and efficiency.

2.1.2 Transition to no-code

During a presentation about new AI chips, NVIDIA's CEO Jensen Huang made a bold prediction, suggesting that AI will eventually take over coding, making programming as we know it optional [67]. According to Huang, future development processes will diverge significantly from current practices. People and companies will be able to implement their ideas just by explaining them to AI models. This shift could not only enhance performance but also unlock a myriad of possibilities currently beyond our consideration. This statement aligns with the recent emergence of DevinAI[1], a groundbreaking AI system developed by Cognition Labs, hailed as the first autonomous AI software engineer capable of tackling complex engineering tasks from start to finish without human intervention. When evaluated on the SWE-Bench benchmark [33], which tests an AI's ability to solve real-world GitHub issues, DevinAI correctly resolved 13.86% of the issues unassisted, far surpassing the previous state-of-the-art model's performance of 1.96% unassisted and 4.80% assisted. DevinAI, controlled by a fine-tuned LLM, autonomously tackles tasks for review, using a comprehensive suite of developer tools in a safe sandbox environment. If future

[1] Devin's introduction blog post available at `https://www.cognition.ai/blog/introducing-devin`

versions of AI software developers will keep getting better, it is very likely these tools will follow the success gained by ChatGPT and Copilot. However, in order to obtain better results, it is still very important to develop and discover techniques regarding the main component of such AIs, the Large Language Models.

2.2 The Foundation: Language Models

In the last years, the field of Natural Language Processing has come a long way. The objective of this AI research field is to build models of the human use of language that serve in various technological applications. The language model, considered one of the foundational components of this technology, is a probabilistic model that assigns a probability to a sentence or word sequence $P(word_1, word_2, ..., word_{n-1}, word_n)$, estimating how likely it is to occur in natural language.

Natural Language Processing, is a research field that focuses on different tasks related to the understanding and generation of text. In order for a large language model to be considerable good, it must have good performances in both NLU (Natural Language Understanding) and NLG (Natural Language Generation).

Some NLU applications include:

- **Sentiment Analysis**: Determining the sentiment expressed in a text, such as capturing if a review is positive or not.
- **Entity Recognition and Relationship Extraction**: Identifying key elements in text like names of people and the relationships between subjects.
- **Text Classification**: Categorizing text into one or more predefined topics based on their content. One example is spam detection in email systems.

A good understanding of language can therefore lead to a good text sequence generation, that can be used for different applications like:

- **Machine Translation**: translation of an input text in a different target language. The LM is used to decide the words and the order in which they have to appear.
- **Speech Recognition**: transcript text from acoustic signal. The sentence probabilities help decide between similar-sounding options.
- **Summarization**: generate shorter paragraphs from longer textual inputs.
- **Dialogue Systems**: communicate with user through conversation.
- **Content Generation**: given an initial prompt, the model generates a sequence of words that reflects the solution of the problem described in the prompt. Code generation, the topic of this research, falls under this prominent area of AI applications.

2.3 Scaling Up: Large Language Models

Large Language Models, distinguished from standard LMs for their abilities in general-purpose language generation, are commonly trained via self-supervised and semi-supervised training techniques based on big datasets. Currently, the most performing LLMs are influenced by the transformer architecture, presented in Google's "Attention Is All You Need" [68] in 2017, which gave birth to the widespread adoption of BERT in 2018 and, subsequently, larger and better models like the OpenAI's GPT family, Meta's LLaMa, Google's PaLM, Gemini and others.

The transformer is a neural architecture, originally developed for machine translation, that employs positional encoding and an attention mechanism to derive dependencies between input and output elements. Introduced with an encoder-decoder implementation, it has been adapted to other types of layer structures. The encoder's function is to process the input by converting words into a contextual representation, using positional encoding to maintain sequence information that would be lost during parallel processing. It makes use of self attention, allowing words to interact with each other, in order to enhance the model's understanding of the context.

The decoder, based on the encoded input and the previous outputs, does the opposite job of the encoder, employing both self attention for coherence and cross attention to make the output accurately reflect the input context. The transformer can handle parallel sequences of data, improving performance and efficiency over the recurrent neural networks used for most language tasks.

While the original transformer model makes use of both encoder and decoder, different variations that specialize in various aspects of language modeling have been proposed. For example, BERT (Bidirectional Encoder Representations from Transformers) utilizes only the encoder part to comprehend the context of words in text [15], making it very suitable for natural language understanding tasks. Conversely, models like T5 [54] and BART [35] are examples of encoder-decoder architectures that are designed for a broader range of tasks like text generation, summarization, and translation. Meanwhile, models such as the GPT series [6], based on decoder-only architectures, focus on generating coherent and contextually relevant output text based on the input they receive, making them the most aligned with the language modeling objective previously introduced.

In the last year, researchers have been trying to mix the two major architectures, RNNs and transformers, to obtain even better results, developing Receptance Weighted Key Value (RWKV) models, that use a linear attention mechanism, simpler than the one in the transformer, combined with an RNN. RWKV is obtaining similar results to the transformer based alternatives with similar parameter count, while keeping the use of the resources significantly lower [51].

As introduced before, OpenAI's GPT4 is still considerable the most powerful and accurate LLM available, but since LLaMa's weights leak in march 2023 [69], the community has made big steps in improving models and bridging the gap between open-source and closed-source models. As for the current situation, different companies populate the LMSYS Chatbot Arena Leaderboard [80] with one or more of their own models.

2.4 Efficiency by Design: Low-Parameter LLMs

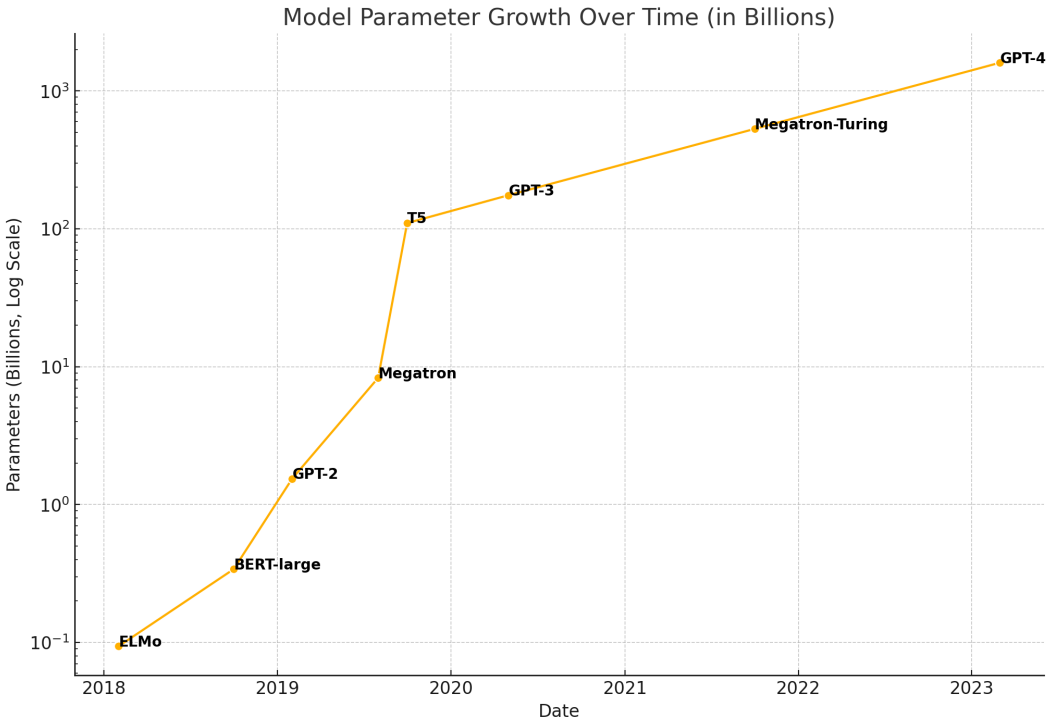


Figure 2.1: Large Language Models' exponential growth across the years 2018-2023

As mentioned earlier, it is proven that a higher number of parameters and a bigger training set generally tend to give better performance [34], resulting in higher capabilities in capturing the nuances of language more effectively, leading to a superior level of generalization and understanding. Furthermore, the relationship between model size and performance is not linear but shows diminishing returns as the models become exceedingly large, creating a power-law region outside of which the learning curve plateaus [30]. The visualisation of this region is shown in figure 2.2, directly taken from the study by Hestness et al.[30]. The scaling laws apply not only to LLMs [11] but also to neural models across various architectures and domains, such as computer vision [75] and machine translation [24].

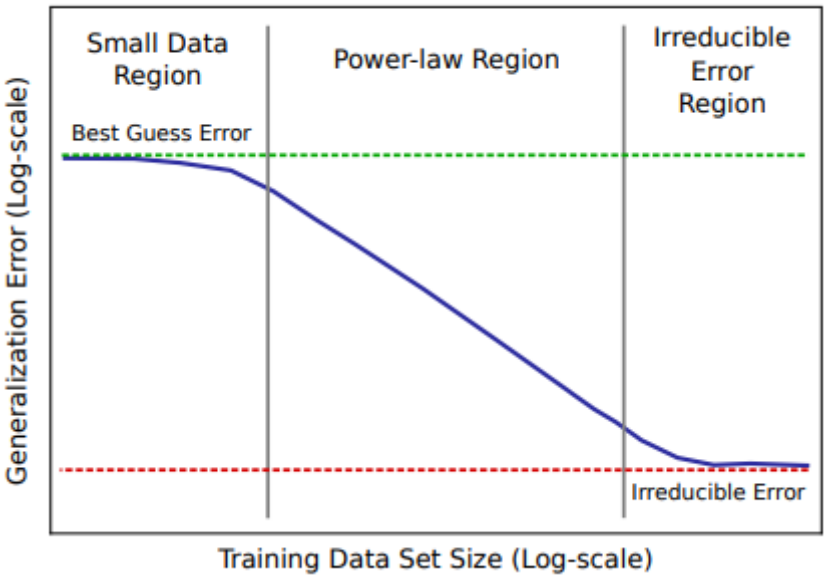


Figure 2.2: Sketch of the power-law region learning curve, from Hestness et al.[30]

Smaller models, however, thanks to different optimization techniques, are now capable of yielding similar results to the ones of counterparts with 10 or 20 times the number of parameters, especially when comparing them on specific tasks. For example, a smaller LLM trained on more code than natural language, like CodeLlama, is supposed to achieve a performance comparable to GPT 3.5 on coding challenges, outperforming the base Llama model which should yield better results on more natural language related tasks. First of all, it is important to separate low parameters models into two categories: the downscaled version of a bigger models, like LLaMa2 7B, which is derived from the 70B parameter version [65], and models that are designed to be small, like Mistral 7B, which is only available in one size [32]. Models that employ more efficient data encoding techniques and attention mechanisms, like parameter sharing [40] and FlashAttention [9], can process information more effectively, improving performance without linearly scaling the dimensions of the model. Training techniques are also an important factor when trying to get the best out of limited sized models. Knowledge distillation, for example, is a technique that allows smaller models to learn from the capabilities of their larger counterparts, learning to emulate their behaviour without scaling the size [25]. Finally, other

more practical techniques like fine tuning and domain specific instruction tuning can improve the quality of the generated output. Both topics are are explained more in detail in the relative sections 2.7 and 2.8.

2.5 Even Smaller: Quantized Models

Until now, the term 'size' has referred to the dimensions of the model in terms of its parameter count. This sections talks about less computationally demanding version of the same LLM, where size is the actual memory used by the system to load the model and run inference on it.

An LLM's weights are the learned parameters that the model uses to generate text, and are typically stored as floating point numbers. The precision of these numbers, that is the quantity of bits used to represent an information, influences both the memory usage and the precision of the model. For instance, weights are often stored in 32 bit floating point format (FP32), which allows for a very detailed representation of the information but also consumes substantial memory. To reduce computing requirements, models can employ lower precision formats like FP16 (half precision floating point) and Bfloat 16, which is a format optimized for machine learning that makes use of a different distribution in how each bit represents the different part of the number. Different floating point representation formats have been proposed by companies and/or regulatory agencies [71], each optimized for specific tasks.

In the last years a big effort has been done by the community to create even more accessible models, using different compression methods. This technique, called quantization involves reducing the precision of the model's weights and activations, compressing the model's actual size without substantially compromising its performance or reducing the number of parameters. This makes small models even easier to run locally requiring less resources, while still giving almost the best results possible. Lower precision models also allow for substantially cheaper requirements for fine tuning, thanks to Quantized Low Rank Adapters (qLoRa [14]). The existing

different quantization techniques can be categorized in two main branches: Post-Training Quantization (PTQ), that recalculates the weights after the training phase, and Quantization-Aware Training (QAT), that adapts the training process to output directly the quantized weights [44]. QAT lets the model take into consideration the quantization during the training phase, allowing better performance than PTQ techniques. However, quantizing weights after they have been calculated is less computationally demanding than retraining a model, making post training quantization a more diffused option. PTQ methods are divided in naive, hybrid and selective methods [10]: the first methods quantize all the values to the same output type (e.g. INT32 to INT8), hybrid quantization allows for mixed output types, converting only specific operators to a lower rank type. The selective approach further refines the hybrid one, enabling the quantization of specific operators with different calibration methods and granularity, offering flexibility to optimize accuracy and latency for different parts of the neural network. The first significant quantization method that maintained high quality for models with over 7 billion parameters, called LLM.int8() [13], is based on the 8 bit matrix multiplication, only limited to non outlier values. Since this important breakthrough in the quantization field, different and more refined methods have been developed, reaching even lower bits multiplications and overall lighter models. Pre-quantized models are typically available in three main formats [27]:

- **GPTQ**: compresses all weights to 4 or another specified number of bits minimizing squared error between the full-sized weights and the quantized version [20]. During inference, it dequantizes the weights back to FLOAT16, improving performance, especially on GPU equipped machines.
- **GGUF** (GPT-Generated Unified Format, formerly GGML): a more versatile format, that allows better inference with CPUs and Apple's new Metal architecture.
- **AWQ**: similar to GPTQ, but uses a selective approach, assuming that not all weights are equally important, allocating more space and

higher precision to the more critical ones [39].

Large language models can ultimately be quantized to almost any quantity of bits, with the most relevant being 4 and 8 bit. Other quantization approaches have been proposed, like binary quantization. This new approximation technique exists in different variants, like the ternary value based weights, as seen in **BitNet b1.58** [46] and a more hybrid "AWQ-like" approach that binarizes only the least important weights to values of 0 or 1 [60].

2.6 Large Language Models for code generation

Despite their linguistic origins, LLMs have proven to be able to handle code as well, due to several factors:

- Similarities between natural language and code: both are communication means, following their own grammatical and syntactical rules. Just like other languages, the model will learn to recognize and replicate pattern found in code too.
- Structured nature of code: compared to natural language, code tends to be more structured and less ambiguous, making it even easier to generate.
- Abundance of training data: thanks to publicly available repositories, like the ones on GitHub, there is an immense amount of available, high quality and well documented code snippets, spanning every language and sector.

Given the fast paced evolution that the NLP sector is experiencing, this section will not attempt an exhaustive description of all the LLMs currently available. Instead, it identifies the key models that are representative of the state of the art, acknowledging that the specific LLMs analyzed may evolve with the progress of the study.

- **Mistral** 7B: A 7 billion parameters LLM engineered for performance and efficiency [32] by Mistral AI. The model is capable of reaching

very good performance while being optimized for real-time applications. It makes use of sliding window attention and grouped-query attention that bring down inference time and memory requirements. The model also has a bigger context window than most of its rivals, managing up to 32k tokens.

- **Mixtral** 8x7B: An evolution of Mistral that uses the Sparse Mixture of Experts architecture, keeping the parameter count low but improving quality of answers [3]. The MoE architecture loads 2 of the 8 submodels on the device, allowing it to maintain computational requirements low. Despite this, the 8x7B model is still big compared to the 7B version, making it less accessible in terms of computing demand. Another even bigger version of the model has been introduced, that relies on 8x22B parameters sub-models, with double the context window (64k tokens).
- **CodeLLaMa** 7B, 13B and 34B: Version of LLaMa-2 from Meta specifically designed for code, available in different sizes [58]. LLaMa-2 [65] won't be analyzed because of lower performance on code due to the availability of a version optimized for code.
- **CodeQwen 1.5** 7B: The Chinese model from Alibaba Cloud can manage up to 64k tokens and demonstrates great performance on 92 different coding languages [53].
- **DeepSeek-Coder** 6.7B and 33B: These open source models give notable results on various automated test, due to their training data mostly being high quality project level code [37].
- **Eagle RWKV-v5** 7.25B: Open source model based on top of the RWKV-v5 architecture with 10-100x lower inference costs that performs on par with other 7B models in multi-lingual benchmarks [51].
- **Phi-2** 2.7B: This small model from Microsoft Research achieves near state of the art performance on various tasks thanks to the high quality "textbook-grade" data it was trained on. The research paper for Phi-1, "Textbooks are all you need", states that the quality of the train-

ing dataset influences the model's performances more than the number of parameters, demonstrating the potential of strategically trained small language models to match or exceed the capabilities of significantly larger models [28].

These models have also been fine tuned on even more code, improving further their performance on code generation:

- **Phind-CodeLLaMa** 34B: Fine tuned version of CodeLLaMa 34B from Phind AI, trained on 1.5B tokens more than the standard version [4]. The models gives prominent results, but given its size, its requirements are still higher than most standard or even high end desktop devices.
- **Vicuna** 13B and 33B: Similar to Phind-CodeLlama, Vicuna is a fine tuned version of LLaMa2 which takes advantage of distillation knowledge. It is trained on conversations between user and GPT4 [81].
- **OpenHermes 2.5** 7B: Mistral Fine-tune trained on GPT4 conversations and open source datasets [64].
- **MagiCoder** CL-7B and DS-6.7B: this models are respectively based on CodeLLaMa and DeepSeek, fine tuned with 75000 synthetic instruction via a new method called OSS-Instruct [70]. The authors of the paper claim its superiority over WizardCoder, previously known as the best coding LLM.

2.7 Fine Tuning

Fine tuning is a transfer learning technique that retrains the weights of a model using new data. This leads to higher performance on the newer type of data, without distancing too much from the base model. In the field of NLP, it is used to extend the knowledge of the model regarding some topics like, for example, code generation. Using a model fine tuned on a certain programming language or a certain set of operations will lead to higher performance on the specific task itself. Even for code inclined models like CodeLLaMa, fine tuning can yield higher results, both in over-

all coding capabilities, like Phind, with a 25% performance increase from the base model, and specific capabilities like code style and use of custom libraries. A fine tuned large language model's performance is also dependent on the size of the model itself and the specific task it is fine tuned for, according to a study from Zhang et al. [76]. While fine tuning is a very computationally demanding task, it is now easier and more accessible than ever thanks to custom open source frameworks like Axolotl [8].

2.8 Instruction Tuning

Instruction tuning is a fine tuning technique for large language models that teaches the model to generate a response to an instruction, in order to get more practical results and less discursive answers. Models are generally trained on generic texts, mimicking the human discursiveness when answering a question. This means that the answers might include unintended sections like biased, toxic, untruthful or unnecessary text. In practice, instruction tuning helps to generate a better and more structured output, facilitating the developments of pipelines that makes use of LLMs instances and increasing the model's accuracy. To obtain the best results from a query to an instruction tuned LLM, the only necessary tweak needed is to give a structure to the prompt.

Input:

```
"<s>[INST] Generate a Python function that takes a list of integers
and returns the sum of its elements.  [/INST]"
```

Output:

```
def sum_of_elements(int_list):
    """
    This function takes a list of integers and returns the sum
   of its elements.

    Parameters:
    int_list (list): A list of integers

```

```
    Returns:
    int: The sum of the elements in the list
    """
    return sum(int_list)

# Example usage
numbers = [1, 2, 3, 4, 5]
print(sum_of_elements(numbers))  # Output: 15
```

</s>"

This example shows the prompt template of Mistral 7B instruction tuned model, where <s> and </s> delimit the beginning and end of a sentence, while [INST] and [/INST] define the section for the instruction itself. The instruction provided a clear and concise task for the LLM to perform, and the LLM responded with a well-structured Python function that meets the specified requirements. It also included a docstring and an example usage, demonstrating a thorough understanding of the instruction.

Instruction tuning does not further train the model on its language modeling objectives, but helps to align the output to better match the specific input instructions. While this does not extend the model's foundational language skills, it can lead to the acquisition of new capabilities. This is mostly achieved via a fine tuning technique called reinforcement learning with human feedback, often referenced as RLHF, where each generated response is evaluated by a human operator that triggers a reward mechanism, improving the model's performance by learning from these evaluations. As shown in the research from Ouyang et al. (2022) [49], instruction tuned models' completions are generally preferred by human evaluators. In this study, an instruction tuned version of an LLM will be used instead of the base version or the chat-specific fine-tuned variant, when available.

2.9 What's next for LLMs

During the first part of 2024, we have seen an evolution regarding how LLMs interact with the user. While the multi turn question answering system is both efficient and user friendly, a new type of LLMs is emerging.

These new LLMs, like OpenAI's GPT-4o [48] and Google's Project Astra [23], introduce the capability of understanding and processing speech in near real time, allowing for a more discursive chat between model and user. Moreover they implement the possibility of processing real time video too and are both deeply more aware of the tone of the conversation, showing sparks of what could be called emotional intelligence. Most Large Language Models are trained on a split of natural language and code, where the natural language part prevails due to their intent being modelling human language. Because of this, these models are capable of a broad variety of tasks, which makes them not optimal for automated code generation. Some models, however, are designed with the intent to be good coders, like **DeepSeek Coder**, which is trained from scratch on 2T tokens, with a composition of 87% code and 13% natural language in both English and Chinese [37]. The applications of coding LLMs are broad and span from auto completion to fully autonomous AI Software Developer. For this particular kind of task, which requires extensive coding knowledge, it is important to have a reliable model that minimizes the mistakes. The field is still in expansion, but given the potential it has, it is likely that it will develop very rapidly.

2.10 Code evaluation methods

Code evaluation is a very complicated software engineering related tasks. When assessing the quality of code it is important to take into considerations some key aspects:

- **Similarity to a ground truth solution**: comparing the generated completion to a reference code can, in some cases, be a good way to determine the quality of the code snippet by analyzing the similarity to the ground truth solution.
- **Adherence to language rules**: the completion should follow the coding conventions in terms of syntax, usage of data structures and programming paradigms to the specific language.

- **Syntactical correctness**: the code must also be free of syntax errors, allowing it to compile or run without issues.
- **Functional correctness**: the generated code has to fulfill the intended purpose specified in the prompt and provide the expected outcome in different conditions, including the edge cases.

Different metrics have been proposed to evaluate text in the context of natural language generation. Various of the existing ones, however, are match-based metrics, that compare the generated text to a ground truth string. While these techniques are helpful for assessing machine translated text in several languages, they may not be as indicative when applied to code. This arises from the limited set of keywords used by programming languages, and is aggravated by the multiple distinct approaches that can be taken when solving a given coding problem. The most relevant match based code evaluation metrics are mostly based on precision, recall and accuracy, with some of them employing language models:

- **Exact Match Ratio**: measures the proportion of total predictions that were correct. In the context of code, it assesses the percentage of matches between the generated snippet and the ground truth.
- **Edit Distance/TER**: calculates the number of insertions, deletions, or substitutions required to transform the generated code into the reference code. It provides a measure of similarity that accounts for the possible minor modifications needed to correct the output, reflecting the closeness of the generated code to the expected solution [61].
- **BLEU**: measures the similarity between two code snippets by counting overlapping n-grams [50].
- **CodeBLEU**: evolution of BLEU that also considers the structure of the code [57].
- **CrystalBLEU**: based on BLEU, but able to reduce the noise caused by trivially shared n-grams [17].
- **Ruby**: considers lexical, syntactical, and semantic representations of source code [66].

- **BERTScore** and **CodeBERTScore**: use contextual embeddings (BERT) to measure similarity of text [78] and code [82].
- **COMET**: neural framework that exploits informations from both the source input and a target-language reference translation in order to more accurately predict MT quality [55].
- **Rouge** (Recall-Oriented Understudy for Gisting Evaluation): a recall oriented set of metrics born to evaluate summarized text comparing it to a reference [38]. It consists of Rouge-N, based on n-grams co-occurrences, Rouge-L, that uses the longest common sub-sequence of words in the two sentences to determine their similarity, Rouge-W (similar to the previous one with the introduction of weights) and Rouge-S, which measures the overlap of skip bi-grams between the two sentences.
- **YiSi**: a metric that makes use of sentence representation to calculate the similarity between a machine translation and human references by aggregating the weighted distributional lexical semantic similarities and optionally incorporating shallow semantic structures. It offers different variants depending on the resources available for evaluation [45].
- **Meteor**: computes a score based on explicit word-to-word matches, synonyms and simple morphological variants of a word, taking into account both precision and recall [12].
- **CHRF**: considers character-level precision, recall, and F1 score [52].

Code can also be evaluated via **execution**: running the code and the relative tests can be enough to check if the code actually resolve the problem or not. This is, however, more intricate to implement and can suffer from non exhaustiveness of test cases. The most common metric to evaluate execution is *pass@k*, which quantifies the probability that at least one sample out of the k best ones is correct (passes the tests) [7].

$$pass@k := \mathbb{E}_{Problems}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]$$

Where:

k is the number of best samples considered

n is the total number of generated samples

c is the number of correct samples

$\mathbb{E}_{Problems}$ is the expected value over the problems

This metric allows us to understand if the model, given a number of samples k, is able to solve the presented problem or not, by calculating the complement of the probability that all k picks are wrong among the $n - c$ incorrect samples.

2.11 Datasets

Models are evaluated on the data they generate, therefore selecting which data to synthesize is as important as choosing which metric to use. Next are listed the most important code evaluation datasets, containing the problem in natural language, a ground truth solution and eventually a list of test cases:

- **HumanEval**: The most famous code evaluation benchmark, proposed by OpenAI in 2021 [7]. Contains only 164 docstrings and relative solutions, with a list of test cases that need to pass in order to confirm that the prompt has been interpreted correctly. The topics of the questions encompass language comprehension, algorithms, and basic mathematics, with some being similar to ones that could be encountered in software interviews.
- **APPS**: The Automated Programming Progress Standard consists of 10k coding problems, 131k test cases and 232k ground-truth solu-

tions written by humans. Problems vary in difficulty and can be quite complicated to understand, with the average length of each prompt being around 293 words [29].

- **MBPP**: Mostly Basic Python Programming consists of around 1000 entry level Python programming problems, covering programming fundamentals, standard library functionality, and so on. Each problem consists of a task description, code solution and 3 automated test cases. [5]
- **CoNaLa**: the Code/Natural Language Challenge dataset is filled with 2879 examples from Stack Overflow questions. Each example includes a natural language intent paired with a corresponding Python snippet, typically consisting of a single line of code. In addition to the manually annotated dataset, there are also 598,237 mined intent-snippet pairs [73]
- **CodeContests**: DeepMind's dataset containing competition level code used to train the AlphaCode model. The data is derived from different sources and the problems contained in the dataset include both correct and incorrect human solutions in different languages, other than the relative test cases. [36]
- **EvalPlus**: EvalPlus is a code synthesis evaluation benchmark based on HumanEval and MBPP. The HumanEval+ dataset contains altered versions of ambiguous questions and extends the test cases up to 80 times the original number present in the standard HumanEval. The overall performance over the standard version is 19.3-28.9% lower, though the landscape is more varied now, with some smaller models outperforming GPT-3.5 [42].
- Other Datasets: There are alternative coding datasets available, although their relevance to the research may be limited. For instance, Card2code Hearthstone [41] requires generating a corresponding class based on a given Hearthstone Card description. Another example is NAPS [74], which utilizes the Unified Abstract Syntax Tree, an abstraction layer over the PSI of various programming languages

designed for the Java Virtual Machine [31]. Finally, Natural2Code is a coding dataset, supposedly similar to HumanEval, developed by Google and shown in the Gemini presentation, which is not yet been leaked online.

Considering the differences between chat-oriented instruction-tuned models and standard code completion models, it is important to make distinction between the prompt oriented evaluation sets already introduced and the benchmarks that evaluate a model's ability to suggest only the missing snippet. Different datasets have been proposed in order to answer the question "Can Language Models give the right suggestion?". These datasets, namely **CrossCodeEval** [16], **RepoBench** [43] and **RepoCoder** [77], differ from the aforementioned sets like MBPP and HumanEval because they don't specify the problem in natural language, but feed as prompt the code that surrounds the missing section. Due to the nature of this evaluation methods, the context fed to the LLM is extended to the whole repository, and contains retrieved code from other relevant files.

Finally, **RepoQA**, developed by the EvalPlus group, is a benchmark for evaluating long-context code understanding in large language models, oriented on real-world repository based coding tasks in five programming languages. It is based on the "Needle in the Haystack" problem, which requires the identification of specific and relevant information from a large textual context assessing LLMs' ability of long-context code understanding and retrieval [18]. The image 2.3, from RepoQA's Homepage, shows the structure of the context provided to the model. It begins with an initial instruction outlining the task, similar to the one used for the other task, followed by the code context, and then a description of the specific function to be retrieved. Finally, the main instruction is repeated to reinforce the model's understanding of the task. The output, which should contain the requested function, is then evaluated using BLEU and a similarity threshold.

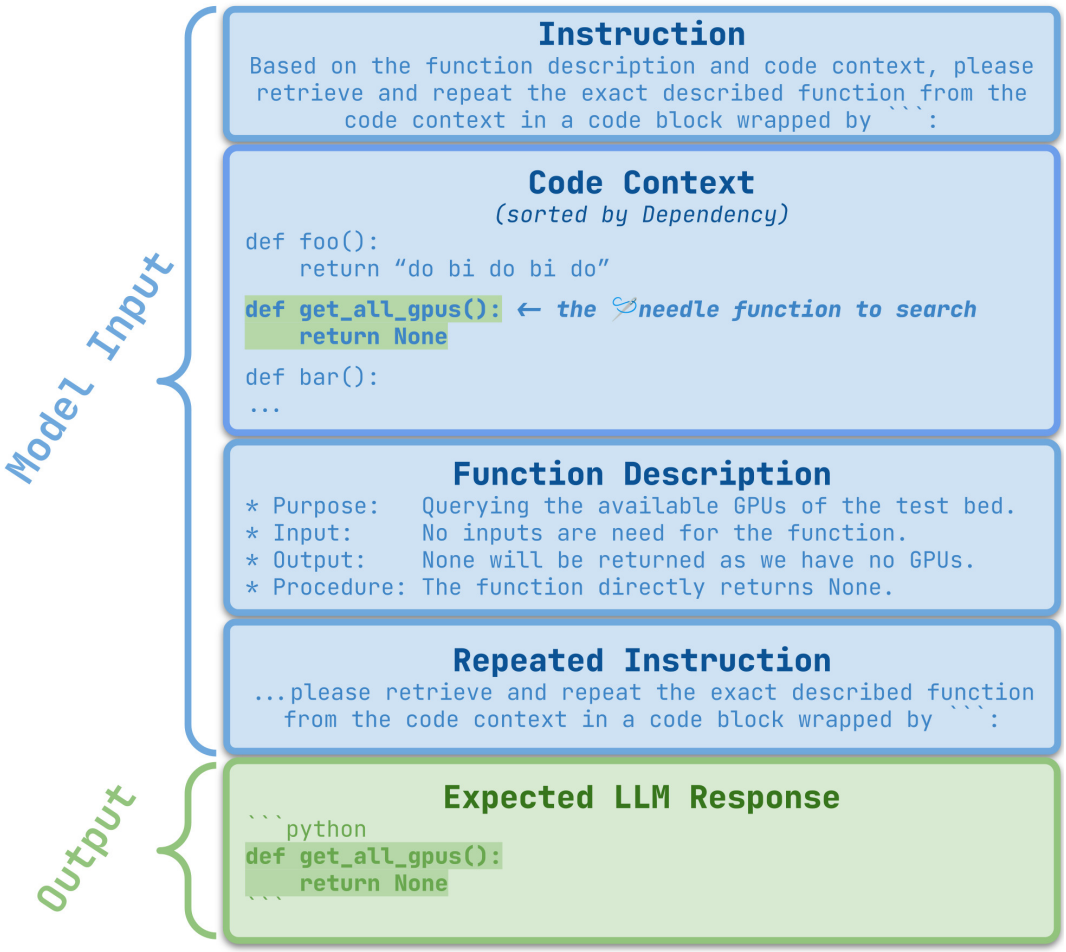


Figure 2.3: RepoQA's context structure, including the model's response

2.12 Summary

This chapter has analyzed the state of the art of coding, reflecting upon how it will evolve in the future thanks to the possibilities introduced by artificial intelligence based tools that support and eventually might replace the role of developer. Code, being a subset of language, is modeled in ways similar to the latter. The transformer, the most relevant neural architecture in today's language modeling landscape, has opened the doors for a new era of code synthesis, based on pretrained large language models. These models vary in size, openness and performance on different tasks. Thus bigger models tend to outperform smaller ones, it is also true that, with time, efficiency and optimization allow to extrapolate more from the small open source ones, making their use more accessible and demo-

cratic. To further improve the efficiency of these models, various techniques have been introduced, each with a different goal. Quantisation allows the model to run on approximated weights, making it even less computationally onerous. Fine tuning is used to further train the model on unseen data in order to extend its knowledge, while instruction tuning helps the LLM generate generally more coherent text in the context of multi turn question answering.

The chapter has also given an overview of the most diffused and interesting low parameters models, each characterized by their own architecture and training data. Finally, we reflected upon the techniques employed to evaluate an LLM, restricted to the field of programming. While it is impossible to evaluate a model by itself, it is feasible to evaluate the code it generates and compare it to others. To generate code, various evaluation datasets have been introduced, which require the model to output a code snippet that solves the presented problem. In order to automate the evaluation of code, multiple metrics have been introduced. While match based ones, originating from the field of multilingual machine translation, should give a sense of the quality of the generated code, functional metrics appear as a more robust and indicative way to validate code and the model generating it.

Chapter 3

Methods

The research is divided in four parts: Datasets and Model Selection, Sample Generation, Evaluation and Analysis.

3.1 Datasets and Model Selection

The evaluation datasets referenced in Section 2.11 presents a diverse landscape that requires careful selection due to different levels of relevance and applicability to our specific research objectives. A critical assessment of these datasets reveals several limitations and mismatches with the goals of our study:

- **APPS**: This dataset features a long and intricate prompt structure, which consequently yields suboptimal results during the evaluation phase. The complexity of APPS makes it a less indicative evaluation harness.
- **CoNaLa**: Restricted solely to evaluations based on match-based metrics, CoNaLa offers limited utility for code analysis. While it will be incorporated to some extent, its application remains constrained by its evaluative methodology.
- **CrossCodeEval**, **RepoCoder**, and **RepoBench**: These datasets are tailored for Fill In the Middle (FIM) models, which are specifically optimized to consider both preceding and subsequent contexts. This

models are not what the research focuses on, thus diminishing the datasets' relevance.

- **Card2Code**: The exclusion of Card2Code is due to its lack of unique features and real world applicability, which limits its utility in providing distinctive insights.

Given these considerations, the datasets selected for use in this study are primarily **HumanEval**, **MBPP**, along with their enhanced versions **HumanEval+** and **MBPP+**, and **RepoQA**. These datasets have been chosen for their compatibility with the research objectives, focusing on standard LLMs, and their proven efficacy in rigorous evaluative contexts. More in detail, HumanEval, MBPP, HumanEval+ and MBPP+ are composed of a **prompt** describing the problem, a **ground truth solution** and a list of **test cases**. A more in depth analysis of how each problem is structured, along with examples, is provided in Appendix B. RepoQA, on the other hand, makes use of 50 repositories in 5 programming languages to evaluate the LLM's context retrieval capabilities.

Dataset	Type	Problems	Size	Language	Article
HumanEval	Prompt/Solution/Tests	164	83.9 kB	Python	Link
MBPP	Prompt/Solution/Tests	974	351 kB	Python	Link
HumanEval+	Prompt/Solution/Tests	164	2.9 MB	Python	Link
MBPP+	Prompt/Solution/Tests	378	1.13 MB	Python	Link
RepoQA	Information Retrieval	500	10.3 MB	Python, C++, Java, TypeScript, Rust	Link

Table 3.1: Dataset used and their relative info

In order to run inference on each problem, a system prompt has been defined as follows:

```
Write code to solve the following coding problem. WRAP YOUR
CODE USING ‘‘‘ OR IT WON’T BE EXECUTED! DO NOT GIVE EXAMPLES!
```

This string states the task the model will have to tackle (solving the coding problem) and instructs it to use special characters to wrap code. These characters are needed to automatically extract only the code from the answer. The instruction and the prompt are formatted in the correct template before being encoded and fed to the model. The whole pipeline has been illustrated in figure 3.1 in chapter 3.2. The list of templates used can be

found in appendix C.

Before acting, it is important to carefully choose which models to test and how. The landscape of the LLMs for code generation listed in section 2.6, contains a wide variety of models, which span different sizes and come from different AI companies. While this list references the most popular ones, there are significantly more models to be considered.

Thanks to the public availability of a leaderboard directly provided by the EvalPlus Team, available on the EvalPlus Website, the research can benefit from a very wide selection of LLMs to compare. However, due to computational and time constraints, the more in depth study regarding quantization only makes use of a limited number of models, chosen for their relevance in the landscape of LLMs for coding:

- **Mistral 7B**: the highly regarded European model offers very fast and efficient completions, making it one of the most used models in the selected size range for multiple applications.
- **CodeLlama 7B and 13B**: Both the smaller versions of CodeLlama are easily accessible in terms of costs. It is also interesting to understand how much a code oriented model performs compared to less optimized but generally more innovative ones.
- **Llama 3 8B**: The latest Llama model, announced as a GPT-4 competitor, is expected to deliver good results while still being locally runnable. Thanks to the popularity of its predecessors it has the potential to become a very diffused option.
- **MagiCoder-S-DS 6.7B**: This DeepSeek-Coder based model, which features a training set predominantly made of code, can also give insight regarding the efficacy of fine tuned models.
- **Phi-2 3B**: The small model based on the claim that the quality of training data influences the abilities of a model more than its size has a lot to prove.

3.2 Sample Generation

In this phase, the focus is on running the models to generate the samples for each prompt. After obtaining all the datasets, these have been adapted to a standardized data retrieval method (as explained in appendix B). A pipeline has been designed to iterate over every dataset and query the prompts to each of the selected models. The sampled completions are then saved for evaluation. It is important to note that each LLM is prompted using the correct template in order to take advantage of instruction tuning where available.

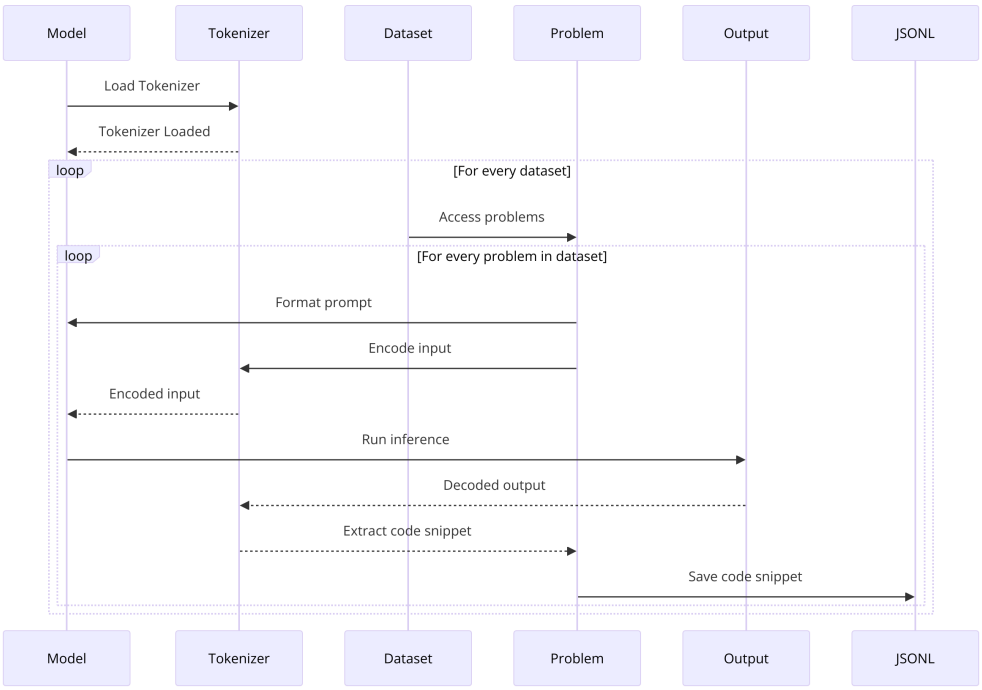


Figure 3.1: Sequence Diagram of the pipeline implemented: each problem is formatted and encoded before being passed as argument to the inference function of the LLM. The output is then decoded and, after extracting the relevant section of code, saved to a JSONL file.

At this time it is very important to:

- Decide the right amount of sample to generate (n value): every model is required to generate n completions for each problem. Despite being 100 samples the ideal number of completions, it could've resulted in a very computationally onerous process and a possible waste of resources. This n value also ideally represents the maximum number of

queries a user would try to prompt the model before understanding that the required solution cannot be synthesized. For this reason, a n value of 10 provides a compromise between insufficient testing and excessive resource utilization.

- Adapt the prompt to each LLM: as already said, using instruction tuned models, when possible, is supposed to yield better results thanks to the higher degree of understanding from the LLM. Each model, however, being trained on different datasets, requires different templates. Even though the transformer library allows for automatic template formatting, not all models are updated enough to have it as a feature. This implies the need of a dictionary that parses each model to its own template, obtained via the HuggingFace description page of the model[1]. The templates used are listed in appendix C.

3.3 Evaluation

In this phase, the generated code snippets have been evaluated using the aforementioned metrics. While the match based metrics are as easy to implement as importing their specific library, the functional metric requires the implementation of a sandbox environment where the code will be run and tested. Given that the chosen metric, pass@k, has been introduced with HumanEval, it made sense to adapt the provided evaluation harness to our needs, also for the sake of repeatability.

3.4 Analysis

After all the metrics have been calculated, they have been compared to analyze the eventual correlation between match based ones and the execution metric. The comparison between the different metrics allowed us to get better closure on the topic of code evaluation itself. The results from

[1]When an instruction tuned model is presented on HuggingFace, it is common to show the prompt format like the developers did for MagiCoder-S-DS (`https://huggingface.co/ise-uiuc/Magicoder-S-DS-6.7B`)

each model have been systematically compared by placing their scores side by side. This approach has allowed for a comprehensive analysis, highlighting the differences and similarities in their performance. Moreover, some of the most important models have been compared to their 4 and 8 bit quantized version, to understand how much performance degradation is introduced by quantization and, more specifically, the GPTQ algorithm.

Chapter 4

Results Analysis

This chapter contains the results obtained from the analysis conducted, from getting a deeper understanding of the metrics, to figuring out which model performs better on coding tasks, to an investigation regarding how quantisation affects the generation of code. More specifically, after analyzing the correlation between different metrics (section 4.1), we compare the main unquantized versions of the most relevant models on the selected datasets (section 4.2). This will allow us to also visualize how the scaling law applies and how instruction tuned models perform against their base version. In addition, we also present an analysis regarding the fine tuning of code oriented models, to evaluate the performance delta introduced by this technique (section 4.2.1).

After that, we show how quantization influences the main LLMs on the task of code generation and, more in depth, how each model is affected by weight approximation. Finally, we present the results obtained by running the RepoQA test on different models, in order to understand which model has the best context retrieval skills, once again analyzing how quantization affects the performance of the model.

4.1 Match Based Metrics correlation

To evaluate the effectiveness of match based metrics, we put them in relations with the functional ones. The selected metrics, chosen for their widespread use and reliability, include: BLEU, TER, Rouge (1, 2 and L), ChrF, Comet, BERTScore (Precision, Recall and F1) and CodeBERTScore (Precision, Recall and F1). The following tests have been run using Mistral 7B, its quantized version and Mixtral 8x7B, chosen for their architectural similarity.

After generating $n = 10$ samples for each problem in HumanEval and MBPP, we calculated the match based metrics scores based on the provided standard reference solution (or multiple solutions for particular metrics that take into account different references). Then, using the implemented sandbox environment, we run every generated snippet of code/-function on all of the proposed texts. After saving which samples pass the problem and which don't, we calculated a pass rate metric using the formula **pass_rate** $= \frac{\#passed}{n}$. Finally, we also considered the metric **passed**, which is an indicator variable set to 1 if any of the n samples pass, and set to 0 otherwise.

It is important to emphasize that, in alignment with the *pass@k* philosophy, we put in relations the best performing samples of each problem. For all the metrics except for TER, which is based on an inverse scale, we consider the highest value among the n samples. Similarly, the *passed* metric considers a problem solved if at least one of the samples solves it. Unlike the other metrics, *pass@k* is not compared directly because it evaluates performance across multiple problems, whereas our method averages the best outcomes from each individual problem.

The correlation matrices, calculated using Spearman's rank coefficient, shown in figures 4.1, 4.2a and 4.2b show how every metric relates to the others across different models.

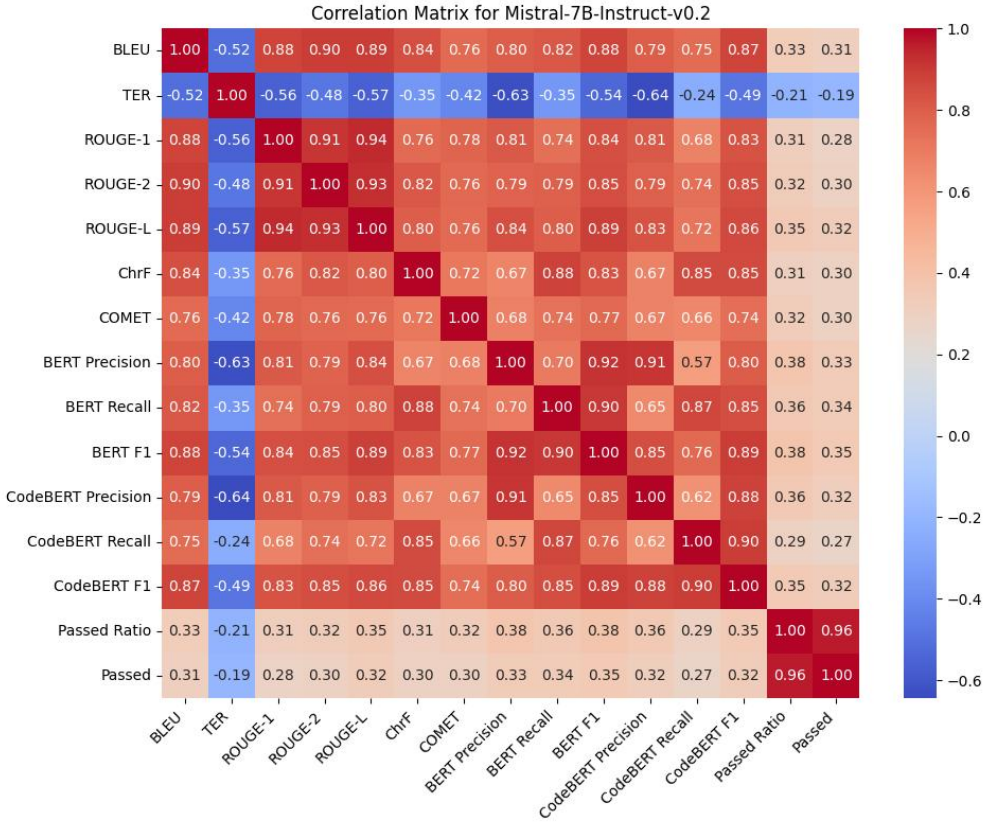
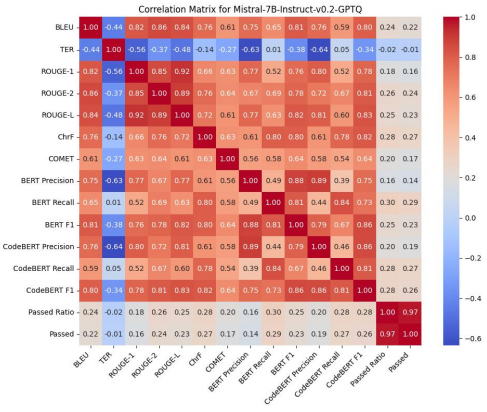
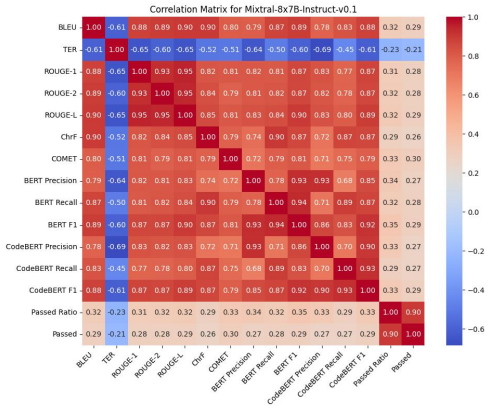


Figure 4.1: Spearman Correlation Matrix between Match Based Metrics and Functional Metrics on Mistral 7B (unquantized)



(a) Spearman Correlation Matrix between Match Based Metrics and Functional Metrics on Mistral 7B (quantized)



(b) Spearman Correlation Matrix between Match Based Metrics and Functional Metrics on Mixtral 8x7B (unquantized)

Figure 4.2: Correlation matrices comparison

As expected, the functional metrics (Passed Ratio and Passed) and the Match Based ones relate very poorly to each other in all models tested. This is due to the nature of match based metrics, that can be observed in the distributions plot (figure 4.3). All the match based metrics, in some order, end up having a similar and gaussian-like distribution, with a very small number of outliers. The functional metrics, however, due to their nature, have a very polarized distribution, with a high count of values being 0 (the unsolved problems). The TER metric stands out from the group because it has an inverse scoring nature; a lower TER score indicates a stronger match between generated and reference code. Consequently, TER negatively correlates with all other metrics.

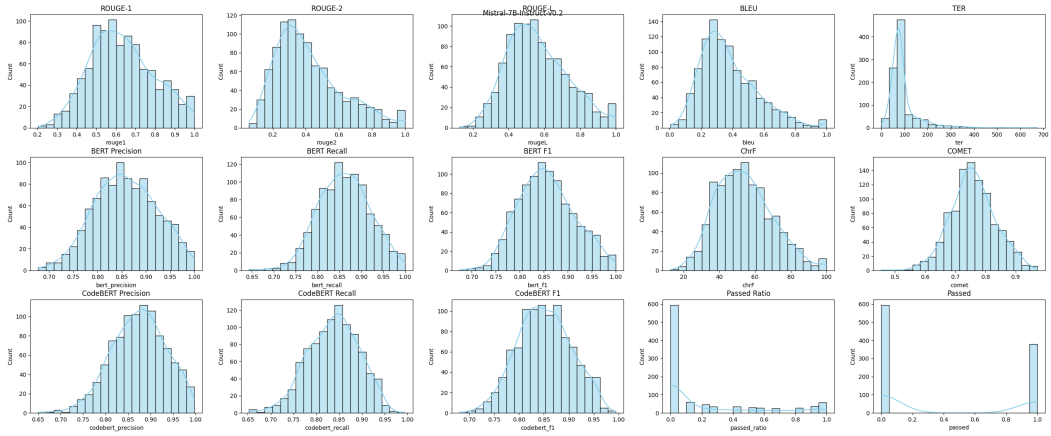


Figure 4.3: Metrics Distribution on Mistral 7B (unquantized)

The observed lack of correlation between the match-based metrics and the functional metrics, along with the Gaussian-like distribution of MBMs, suggests that they may not be indicative of the performance dimension that is captured by the functional metrics.

4.2 Models Comparison

As said before, due to computational constraints, the overall evaluation of the base models is taken from the https://evalplus.github.io/leaderboard.html. The table 4.1 represents the most important models present in the leaderboard, together with their sizes, their level of openness, whether they are prompted or not (as in whether they are instruction tuned models) and the relative scores for both standard and Plus version of HumanEval and MBPP.

As reported in table 4.1, the top performing models remain big and closed source ones, with GPT-4 Turbo still being the undefeated reference LLM. Anthropic's Claude 3 Opus confirms the theory of how closed source models benefit from private investments that allow for more refined training datasets and tweaks. In contrast with this, the lowest step of the podium sees CodeQwen 1.5 on top of it. This model shows impressive performance especially if we consider the number of parameters. Following Alibaba's top coding model, we can see DeepSeek-Coder and a some models mentioned earlier, like Mistral, Mixtral, CodeLlaMa, LlaMa 3 and others.

	Model	*Version*	*#P*	*PR*	*OS*	*HE*	*HE+*	*MBPP*	*MBPP+*	*AVG*
OpenAI	GPT-3.5	May 2023		Yes	None	73.20	66.50			69.85
	GPT-3.5-Turbo	Nov 2023		Yes	None	76.80	70.70	82.50	69.70	74.93
	GPT-4	May 2023		Yes	None	88.40	79.30			83.85
	GPT-4-Turbo	April 2024		Yes	None	90.20	86.60			88.40
	GPT-4-Turbo	Nov 2023		Yes	None	85.40	81.70	85.70	73.30	81.53
Anthropic	claude-2	Mar 2024		Yes	None	69.50	61.60			65.55
	claude-3-haiku	Mar 2024		Yes	None	76.80	68.90	80.20	68.80	73.68
	claude-3-opus	Mar 2024		Yes	None	82.90	77.40	89.40	73.30	80.75
	claude-3-sonnet	Mar 2024		Yes	None	70.70	64.00	83.60	69.30	71.90
	claude-instant-1	Mar 2024		Yes	None	57.30	50.60			53.95
Google	Gemini Pro	1		Yes	None	63.40	55.50	75.40	61.40	63.93
	Gemini Pro	1.5		Yes	None	68.30	61.00			64.65
Mistral AI	Mistral	Large		Yes	None	69.50	62.20	72.80	59.50	66.00
Google	Gemma	Gemma 2B	2	No	None	25.00	20.70	41.80	34.10	30.40
		Gemma 2B - IT	2	Yes	None	17.70	15.20			16.45
		Gemma 1.1 2B - IT	2	Yes	None	22.60	17.70	29.80	23.30	23.35
		Gemma 1.1 7B - IT	7	Yes	None	42.70	35.40	57.10	45.00	45.05
		Gemma 7B	7	No	None	35.40	28.70	52.60	43.40	40.03
		Gemma 7B - IT	7	Yes	None	28.70	25.00	47.10	36.80	34.40
	CodeGemma	CodeGemma 2B	2	No	None	26.80	20.70	55.60	46.60	37.43
		CodeGemma 7B	2	No	None	44.50	41.50	65.10	52.40	50.88
		CodeGemma 7B - IT	7	Yes	None	60.40	51.80	70.40	56.90	59.88
Mistral AI	Mistral	Mistral 7B	7	No	None	28.70	23.80	51.90	42.10	36.63
		Mistral v0.2 7B - IT	7	Yes	None	42.10	36.00	44.70	37.00	39.95
	Mixtral	Mixtral 8x7B - IT	13	Yes	None	45.10	39.60	59.50	49.70	48.48
		Mixtral 8x22B - IT	44	Yes	None	76.20	72.00	73.80	64.30	71.58
Meta	CodeLlama	CodeLlama-7B	7	No	None	37.80	35.40	59.50	46.80	44.88
		CodeLlama-13B	13	No	None	42.70	38.40	63.50	52.60	49.30
		CodeLlama-34B	34	No	None	51.80	43.90	69.30	56.30	55.33
		CodeLlama-70B	70	No	None	55.50	50.60			53.05
		CodeLlama-70B-IT	70	Yes	None	72.00	65.90			68.95
	Llama 3	Llama3-8B	8	No	None	33.50	29.30	61.40	51.60	43.95
		Llama3-8B-IT	8	Yes	None	61.60	56.70	70.10	59.30	61.93
		Llama3-70B-IT	70	Yes	None	77.40	72.00	82.30	69.00	75.18
Qwen	CodeQwen	CodeQwen1.5-7B	7	No	None	51.80	45.70	73.50	60.80	57.95
		CodeQwen1.5-7B-Chat	7	Yes	None	83.50	78.70	79.40	69.00	77.65
	Qwen	Qwen1.5-72B-Chat	72	Yes	None	68.30	59.10	72.50	61.60	65.38
Salesforce	Code T5	CodeT5+-2B	2	No	Full	25.00	22.00	48.40	38.10	33.38
		CodeT5+-6B	6	No	Full	29.30	24.40	52.90	41.50	37.03
		CodeT5+-16B	16	No	Full	31.70	26.80	56.60	47.10	40.55
DeepSeek AI	DeepSeek-Coder	DS-C 1.3B	1	No	None	28.70	25.60	56.90	47.90	39.78
		DS-C 1.3B IT	1	Yes	None	65.90	60.40	65.30	54.80	61.60
		DS-C 6.7B	7	No	None	47.60	39.60	72.00	58.70	54.48
		DS-C 6.7B IT	7	Yes	None	74.40	71.30	74.90	65.60	71.55
		DS-C v1.5 7B IT	7	Yes	None	75.60	71.30	75.20	62.20	71.08
		DS-C 33B	33	No	None	51.20	44.50			47.85
		DS-C 33B IT	33	Yes	None	81.10	75.00	80.40	70.10	76.65
Microsoft	Phi-3	Phi-3-mini - IT	4	Yes	None	64.60	59.10	65.90	54.20	60.95
Databricks	DBRX	dbrx-instruct	36	Yes	None	75.00	70.10	67.20	55.80	67.03
Bud	Code-Millennials	code-millenials-34B	34	Yes	None	74.40	70.70	76.20	64.60	71.48
BigCode	StarCoder 2	StarCoder2-3B	3	No	Full	31.70	27.40			29.55
		StarCoder2-7B	7	No	Full	35.40	29.90			32.65
		Starcoder2-15B - IT	15	Yes	Full	67.70	60.40	78.00	65.10	67.80

Table 4.1: EvalPlus Leaderboard, grey models are fully closed source. `#P` = Parameter Count, `PR` = Prompted, `OS` = Open Source Data

From a more detailed analysis, we can confirm that, relatively to the same model, the scaling law applies, with bigger versions of LLMs performing better than their smaller counterparts, as represented in figure 4.4. However, the rate of improvement and the shape of the growth curve differs significantly depending on the model. This variability suggests that some architectures are more efficient at leveraging more computational resources than others.

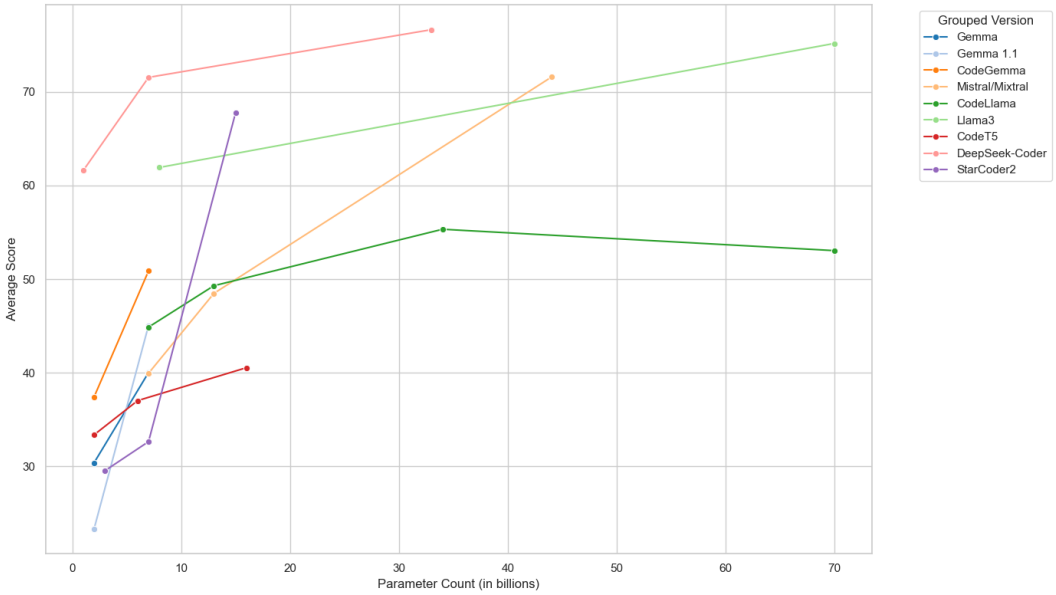


Figure 4.4: Analysis of performance increase different sizes of the same model

It is also confirmed that, in the context of coding and solving problems based on prompts, instruction tuning influences positively the quality of the response. The average performance increase between base and instruction tuned (prompted) model stands at 11.4% and a graphical analysis is reported in figure 4.5. The bar plot illustrates the performance comparison between the different model versions. Less performing models, such as Gemma and Mistral, show minimal improvement or even a decline in accuracy, while the models that already perform decently, benefit more from instruction tuning.

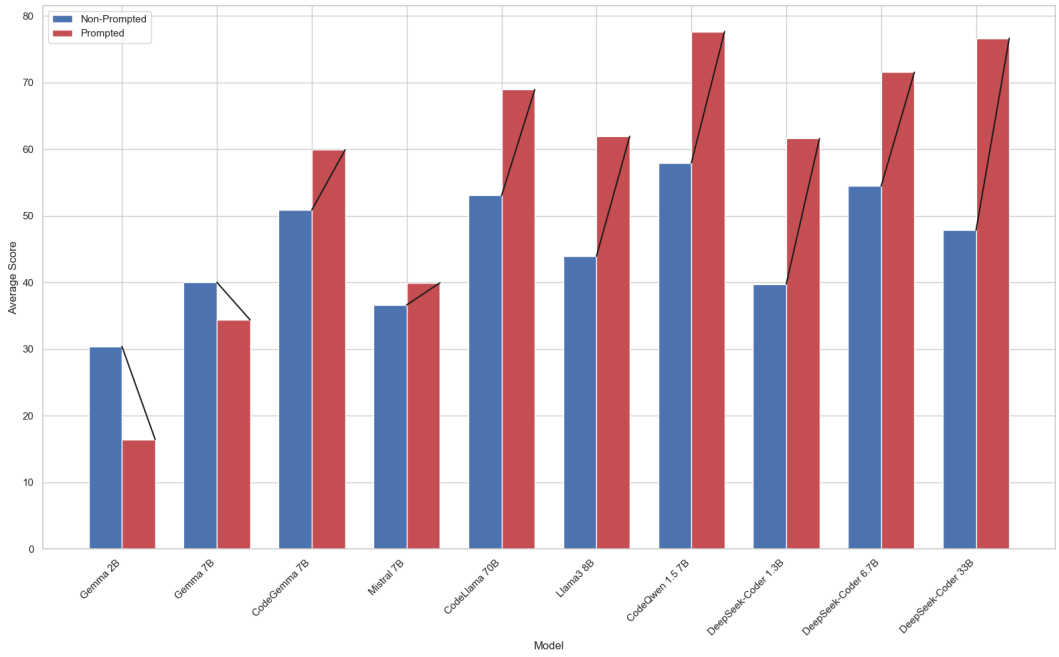


Figure 4.5: Analysis of performance increase between Base and Instruction Tuned Models

Another relevant aspect to take into consideration is the fact that EvalPlus Scores shuffle the leaderboard by quite a lot, with closed source models performing better on the standard datasets than their more refined and exhaustive version. This might or might not be an indication of the fact that closed source models are trained on evaluation datasets too, making them appear as more performing in the leaderboards.

4.2.1 Fine tuned models

The data presented in table 4.2, supports the efficacy of fine-tuning a coding LLM with additional high quality code. When compared with the results scored by the base models, the fine tuned version generally yields better results. For example, Mistral 7B, that scores an average of 36.63% as shown in table 4.1, gets widely outperformed by OpenChat 3.5, that solves almost twice as many problems as the base model. Similarly, both CodeLlama and DeepSeek-Coder exhibit potential for more accurate code generation if further trained on high quality code. It is also confirmed that, when using the same dataset to fine tune the models, DeepSeek-Coder remains the most performing one compared to Mistral/CodeLlama based

LLMs.

	Model	*Base Model*	*#P*	*PR*	*OS*	*HE*	*HE+*	*MBPP*	*MBPP+*	*AVG*
Magicoder	Magicoder-S-CL-7B	*CodeLlama*	7	Yes	Partial	70.70	67.70	70.60	60.10	67.28
	Magicoder-S-DS-6.7B	*DeepSeek-Coder*	7	Yes	Partial	76.80	71.30	79.40	69.00	74.13
WizardCoder	WizardCoder-Python-34B-V1.0	*CodeLlama*	34	Yes	None	73.20	64.60	75.10	63.20	69.03
	WizardCoder-Python-7B-V1.0	*CodeLlama*	7	Yes	None	50.60	45.10	58.50	49.50	50.93
	WizardCoder-15B-V1.0	*DeepSeek-Coder*	15	Yes	None	56.70	50.60	64.30	54.20	56.45
	WizardCoder-33B-V1.1	*DeepSeek-Coder*	33	Yes	None	79.90	73.20			76.55
Phind AI	Phind-CodeLlama-34B-v2	*CodeLlama*	34	No	None	71.30	67.10			69.20
Artigenz	Artigenz-Coder-DS-6.7B	*DeepSeek-Coder*	7	Yes	None	75.60	72.60	80.70	69.60	74.63
Dolphin	dolphin-2.6-mixtral-8x7b	*Mixtral 8x7B*	13	Yes	Partial	64.00	57.30	70.60	59.00	62.73
HuggingFace	starchat2-15b-v0.1	*StarCoder2*	15	Yes	Full	73.80	71.30	74.90	64.60	71.15
Speechless AI	speechless-code-mistral-7B-v1.0	*Mistral*	7	Yes	Partial	48.20	41.50	57.40	48.70	48.95
	speechless-coder-ds-6.7B	*DeepSeek-Coder*	7	Yes	Partial	71.30	65.90	75.90	64.40	69.38
	speechless-starcoder2-7b	*StarCoder2*	7	Yes	Full	56.10	51.80	66.70	56.30	57.73
	speechless-starcoder2-15b	*StarCoder2*	15	Yes	Full	67.10	62.80	73.50	62.40	66.45
	speechless-codellama-34B-v2.0	*CodeLlama*	34	Yes	Partial	77.40	72.00	73.80	61.40	71.15
OpenChat	OpenChat-3.5-7B-0106	*Mistral*	7	Yes	Partial	72.60	67.70	63.80	54.50	64.65
WaveCoder	WaveCoder-Ultra-6.7B	*DeepSeek-Coder*	7	Yes	None	75.00	69.50	74.90	63.50	70.73
WhiteRabbitNeo	WhiteRabbitNeo-33B-v1	*DeepSeek-Coder*	33	Yes	None	72.00	65.90	79.40	66.90	71.05
XwinCoder	XwinCoder-34B	*CodeLlama*	34	Yes	None	75.60	69.50	77.00	64.80	71.73

Table 4.2: EvalPlus learderboard for FineTuned Models. `#P` = Parameter Count, `PR` = Prompted, `OS` = Open Source Data

4.3 Quantisation Comparison

After understanding that open source models are capable of generating quality code on par with closed-source one, and comparing them to each other, we can explore the possibility of making the models even easier to run, by putting them against their quantized version. For the experiment, the models are sampled in both their full sized version and their 4-bits quantized version, using GPTQ as the quantization framework. 4-bits models bring a very substantial decrease in precision compared to their full sized counterpart. This however is justified by the quantity of memory needed to run the model, which, in the case of Mistral, is reduced from at least 16 GB to less than 5 for the 4 bit counterpart. Other quantized versions, using 3 to 8 bits to represent data, are still valuable and relevant models that could be employed in different applications, just like the use of other quantization formats. However this research focuses on 4-bit quantized models, which can now run on limited capabilities laptops and desktop PCs.

In table 4.3 we can see the results obtained by sampling the models on standard HumanEval and MBPP. Aside from the interested open-source model, the table reports the score obtained using the same sampling technique on GPT-3.5 Turbo and GPT-4 Turbo. The scores obtained by these models are close to the ones previously reported, but are overall lower, because of the way the prompt was structured, which was kept as simple as possible, in order to mimic at best how a human user would interact with the model and minimize compilation error.

We can observe that the performance degradation that quantization introduces depends from the model itself:

- Mistral appears to be the most affected by quantization, with an average relative decrease of **77.7%** on HumanEval and **20.7%** on MBPP (absolute decrease at 32.2 and 13.5).
- CodeLlaMa, on both versions, has a very low performance degradation, with an overall increase in performance on HumanEval (+6.4%,

mostly due to the pass@10 score obtained by the 13B model) and a very low decrease in MBPP (around 1.7%).

- LlaMa-3 shows constant performance decrease between the unquantized and quantized version, standing at **15.2%** on HumanEval and **15.9%** on MBPP. In this case, the trade off between resources saved and performance lost appears to be balanced enough, especially considering that the quantized version still performs better than most models tested.
- MagiCoder-S (DeepSeek): The quantized version of the overall best performing model exhibits a **1%** increase in accuracy only on HumanEval, together with a more aligned **8%** decrease on MBPP.
- Phi-2, the smallest tested model, gives overall unimpressive results on both the unquantized and quantized version. Thus his size is limited, the quantized version seems to be performing decently on HumanEval with a **13.6%** decrease from the full weight model and not as good on MBPP, where the relative difference jumps to **39%**.

Model	Size	Quantized	HumanEval		MBPP	
			pass@1	*pass@10*	*pass@1*	*pass@10*
GPT 3.5 Turbo	N/A	No	62.80%	-	52.87%	-
GPT 4 Turbo	N/A	No	85.37%	-	59.03%	-
Mistral	7	No	34.02%	51.22%	20.56%	38.91%
		Yes	4.21%	16.46%	13.83%	35.52%
CodeLlama	7	No	35.61%	54.88%	31.40%	46.41%
		Yes	32.44%	54.88%	31.96%	46.20%
	13	No	36.10%	53.05%	34.60%	49.08%
		Yes	38.29%	68.29%	31.98%	48.67%
LlaMa 3	8	No	56.59%	79.88%	38.35%	54.31%
		Yes	45.06%	71.95%	27.77%	51.95%
Magicoder-S-DS	6.7	No	59.45%	77.44%	51.25%	70.00%
		Yes	60.12%	82.93%	46.37%	65.20%
Phi-2	2.7	No	7.07%	32.32%	14.41%	42.30%
		Yes	6.34%	26.83%	6.97%	30.65%

Table 4.3: Comparison between models quantized and unquantized

To better understand the reason of this performance decrease, table 4.4 shows in detail how each model's generated snippets act when executed. The proportion shown in the table takes into consideration every single sampled solution, without making any distinction based on the problem

it is a solution of. In this case too, models react differently to quantization. In the instances of Mistral and LlaMa-3, which are very comparable due to their general purpose nature, quantization introduces a high level of inaccuracy regarding the task of generating code itself, with a higher compilation error than their non-quantized counterpart and less correct solutions overall.

As demonstrated in table 4.3, both quantized and unquantized version of CodeLlaMa 7B and 13B are very comparable, with the approximated versions showing an even lower percentage of non-compilable code. This robustness could be attributed to the architecture's ability to maintain functionality despite quantization and pruning, as explained by Gromov et al. [26]. The study implies that either current pretraining methods do not fully utilize the parameters in deeper network layers, making them less relevant, or that the shallower layers are the most crucial for retaining knowledge. Finally, Magicoder and phi, following what shown in table 4.3, exhibit a decrease in correct solutions, with a higher percentage of compiled snippets that eventually fail the tests.

Although quantized models may exhibit lower correctness, some of them—namely CodeLlama, DeepSeek-Coder and Phi-2—tend to produce a higher proportion of executable code. This could be accredited to the standardization introduced by approximating weights, which reduces susceptibility to stochastic variations.

To get an even deeper knowledge of how quantization influences the quality of the generated code, both full sized model and its 4-bits quantized version are put in relation with an 8-bits quantized variant. More in detail, we compared Magicoder-S DS on both the HumanEval and MBPP test sets.

Model	*Passed*	*Failed*	*Non Compiled*
gpt-3.5-turbo (1 sample)	54.31%	32.16%	13.53%
gpt-4-0125-preview (1 sample)	62.83%	27.68%	9.49%
Mistral-7B-Instruct-v0.2	22.50%	30.99%	46.50%
Mistral-7B-Instruct-v0.2-GPTQ	12.44%	17.75%	69.81%
CodeLlama-7B-Instruct	32.00%	37.59%	30.40%
CodeLlama-7B-Instruct-GPTQ	31.77%	42.64%	25.59%
CodeLlama-13B-Instruct	34.82%	37.77%	27.42%
CodeLlama-13B-Instruct-GPTQ	32.89%	37.72%	29.39%
Llama-3-8B-Instruct-HF	40.98%	39.31%	19.71%
Llama-3-8B-Instruct-GPTQ	30.26%	34.70%	35.04%
Magicoder-S-DS-6.7B	55.99%	23.49%	20.53%
Magicoder-S-DS-6.7B-GPTQ	48.36%	34.07%	17.57%
phi-2	3.35%,	5.49%	91.16%
phi-2-GPTQ	7.15%,	10.62%	82.23%

Table 4.4: Differentiation between passed all test, failed tests and didn't compile

The models differ in number of bit used to represent a weight and group size, which is a parameter that determines how the weights of the model are grouped together for quantization. A smaller group size generally leads to better accuracy but higher memory usage, while a larger group size reduces memory usage at the cost of some accuracy. The 8 bits / 32 gs version is the most resources demanding of the available, while the 4 bits / 128 gs puts efficiency first. The required VRAM to run inference on the three versions of the model is (approximately) 20 GB for the unquantized version, 8 for the 8 bit and 4 for the 4 bit.

Model		**HumanEval**		**MBPP**	
Bits	**Group Size**	**pass@1**	**pass@10**	**pass@1**	**pass@10**
16 bits	Full model	59.45%	77.44%	51.25%	70.00%
4 bits	128	60.12%	82.93%	43.27%	63.27%
8 bits	32	60.30%	76.83%	47.12%	60.62%

Table 4.5: Comparison between 4 bit, 8 bit quantized and 16 bit unquantized Magicoder-S-DS

As explained in section 2.5, the size difference between BF16 and GPTQ's 4-bit and 8-bit precision format is substantial, allowing to run 4 or more instances of the 4-bits quantized version with the same requirements of the base model or 2 with almost the same VRAM the 8-bits instance needs.

Table 4.5 shows the results of the three version of the same model achieving very similar results, with quantized models almost outperforming the base model on HumanEval and demonstrating significantly inferior performance on the more understanding and reasoning oriented MBPP, similarly to what saw in table 4.3.

Focusing on just the two quantized versions, the lower bit version outperforms the more computationally onerous one over the 10 samples, while the 8 bit model seems to be slightly more precise and reliable when only considering one try.

The last dataset tested, RepoQA, based on the needle in the haystack test, helps us understand how the models are able to retrieve information from context. The table reported below shows how the models score based on a similarity threshold. Every model tested, with a context limited to its maximum input, shows very prominent signs of understanding degradation. The highlighted value is the average across all languages with a match similarity threshold of 0.8, enough to ensure a high level of precision in matching the "needle" (specific information or answer) within the "haystack" (large body of text or repository of data).

Table 4.6 compares the standard and 4-bits quantized version of Mistral 7B using 16k characters of context and Llama 3 8B, restricted to 6000 characters because of its limited context window of 8k tokens. The results reflect the landscape obtained after the functional evaluation, showing a solid decrease in performance when using the approximated version of the weights.

Lastly, comparing 4 bit and 8 bit quantized version of the same models yields interesting results on the Search Needle Function tests. As shown in table 4.7, the difference between the two variants is close to none, proving that the difference in requirements does not reflect the one in performance.

Threshold	**0.0**	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**	**1.0**
Mistral 7B Instruct V0.2											
AVG	85.8	75.8	69.8	66.0	61.2	58.0	55.8	52.0	*46.6*	42.2	29.8
python	92.0	80.0	69.0	62.0	55.0	51.0	46.0	42.0	38.0	31.0	23.0
cpp	80.0	71.0	68.0	63.0	61.0	58.0	58.0	52.0	49.0	46.0	33.0
java	87.0	78.0	70.0	66.0	64.0	62.0	60.0	57.0	47.0	45.0	37.0
typescript	92.0	89.0	86.0	84.0	78.0	74.0	72.0	67.0	60.0	51.0	24.0
rust	78.0	61.0	56.0	55.0	48.0	45.0	43.0	42.0	39.0	38.0	32.0
Mistral 7B Instruct V0.2 GPTQ											
AVG	22.2	14.4	13.2	13.0	13.0	13.0	12.2	12.0	11.8	11.4	9.6
python	23.0	13.0	11.0	11.0	11.0	11.0	9.0	9.0	8.0	8.0	8.0
cpp	27.0	21.0	21.0	20.0	20.0	20.0	19.0	18.0	18.0	18.0	18.0
java	13.0	7.0	5.0	5.0	5.0	5.0	4.0	4.0	4.0	4.0	4.0
typescript	35.0	28.0	26.0	26.0	26.0	26.0	26.0	26.0	26.0	24.0	15.0
rust	13.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0
Llama 3 8B Instruct [6k context]											
AVG	86.4	83.2	81.0	79.8	79.0	78.2	76.4	75.4	74.8	71.0	57.2
python	89.0	86.0	84.0	83.0	82.0	81.0	78.0	78.0	77.0	74.0	71.0
cpp	77.0	72.0	68.0	68.0	68.0	68.0	66.0	64.0	63.0	58.0	34.0
java	86.0	84.0	82.0	81.0	81.0	80.0	79.0	78.0	78.0	75.0	68.0
typescript	94.0	92.0	91.0	89.0	87.0	86.0	85.0	84.0	83.0	77.0	44.0
rust	86.0	82.0	80.0	78.0	77.0	76.0	74.0	73.0	73.0	71.0	69.0
Llama 3 8B Instruct GPTQ [6k context]											
AVG	37.6	26.4	25.4	24.6	24.4	24.2	23.0	22.4	22.4	21.8	18.2
python	28.0	15.0	14.0	14.0	14.0	14.0	14.0	14.0	14.0	13.0	11.0
cpp	55.0	49.0	48.0	47.0	47.0	47.0	46.0	46.0	46.0	44.0	28.0
java	24.0	14.0	13.0	11.0	11.0	11.0	11.0	11.0	11.0	11.0	11.0
typescript	32.0	16.0	15.0	14.0	13.0	13.0	11.0	10.0	10.0	10.0	10.0
rust	49.0	38.0	37.0	37.0	37.0	36.0	33.0	31.0	31.0	31.0	31.0

Table 4.6: Comparison between unquantized and 4-bit quantized version of Mistral 7B and LlaMa 3 8B on the RepoQA dataset

Threshold	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
Magicoder S DS [8k context]											
AVG	81.0	72.2	68.8	65.4	63.6	62.4	60.8	59.4	*57.0*	52.6	44.2
python	91.0	81.0	76.0	70.0	69.0	68.0	66.0	64.0	59.0	56.0	52.0
cpp	74.0	63.0	60.0	59.0	56.0	55.0	53.0	52.0	51.0	48.0	40.0
java	85.0	79.0	77.0	75.0	75.0	74.0	71.0	70.0	69.0	62.0	56.0
typescript	87.0	82.0	79.0	76.0	74.0	73.0	73.0	70.0	67.0	59.0	36.0
rust	68.0	56.0	52.0	47.0	44.0	42.0	41.0	41.0	39.0	38.0	37.0
Magicoder S DS GPTQ [4bit] [8k context]											
AVG	32.6	21.6	20.2	19.8	19.4	19.0	18.0	17.8	*17.6*	16.8	14.2
python	24.0	10.0	10.0	10.0	9.0	9.0	6.0	6.0	5.0	5.0	5.0
cpp	42.0	32.0	31.0	31.0	31.0	29.0	27.0	26.0	26.0	26.0	26.0
java	25.0	14.0	11.0	9.0	9.0	9.0	9.0	9.0	9.0	9.0	9.0
typescript	45.0	41.0	40.0	40.0	39.0	39.0	39.0	39.0	39.0	36.0	24.0
rust	27.0	11.0	9.0	9.0	9.0	9.0	9.0	9.0	9.0	8.0	7.0
Magicoder S DS GPTQ [8bit] [8k context]											
AVG	32.4	21.2	19.8	19.4	19.0	18.6	17.6	17.4	17.2	16.8	14.2
python	24.0	10.0	10.0	10.0	9.0	9.0	6.0	6.0	5.0	5.0	5.0
cpp	42.0	32.0	31.0	31.0	31.0	29.0	27.0	26.0	26.0	26.0	26.0
java	25.0	14.0	11.0	9.0	9.0	9.0	9.0	9.0	9.0	9.0	9.0
typescript	45.0	40.0	39.0	39.0	38.0	38.0	38.0	38.0	38.0	36.0	24.0
rust	26.0	10.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	7.0
CodeLlama 7B Instruct [8k context]											
AVG	73.8	59.6	52.6	51.2	49.4	48.2	46.2	44.0	41.8	38.4	30.4
python	79.0	65.0	57.0	56.0	53.0	53.0	50.0	46.0	43.0	41.0	35.0
cpp	76.0	51.0	45.0	44.0	41.0	39.0	38.0	37.0	36.0	32.0	25.0
java	70.0	56.0	50.0	49.0	48.0	47.0	46.0	46.0	43.0	38.0	33.0
typescript	78.0	67.0	63.0	59.0	58.0	56.0	55.0	52.0	48.0	42.0	22.0
rust	66.0	59.0	48.0	48.0	47.0	46.0	42.0	39.0	39.0	39.0	37.0
CodeLlama 7B Instruct GPTQ [4 bit] [8k context]											
AVG	32.6	22.4	21.0	20.6	20.2	19.8	18.8	18.6	18.2	17.6	14.8
python	24.0	14.0	14.0	14.0	13.0	13.0	10.0	10.0	8.0	8.0	7.0
cpp	42.0	33.0	32.0	32.0	32.0	30.0	28.0	27.0	27.0	27.0	27.0
java	25.0	14.0	11.0	9.0	9.0	9.0	9.0	9.0	9.0	9.0	9.0
typescript	45.0	40.0	39.0	39.0	38.0	38.0	38.0	38.0	38.0	36.0	24.0
rust	27.0	11.0	9.0	9.0	9.0	9.0	9.0	9.0	9.0	8.0	7.0
CodeLlama 7B Instruct GPTQ [8 bit] [8k context]											
AVG	32.4	21.2	19.8	19.4	19.0	18.6	17.6	17.4	17.2	16.8	14.2
python	24.0	10.0	10.0	10.0	9.0	9.0	6.0	6.0	5.0	5.0	5.0
cpp	42.0	32.0	31.0	31.0	31.0	29.0	27.0	26.0	26.0	26.0	26.0
java	25.0	14.0	11.0	9.0	9.0	9.0	9.0	9.0	9.0	9.0	9.0
typescript	45.0	40.0	39.0	39.0	38.0	38.0	38.0	38.0	38.0	36.0	24.0
rust	26.0	10.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	8.0	7.0

Table 4.7: Comparison between unquantized and quantized (4 bit and 8 bit) version of Magicoder-S DS 6.7B and CodeLlaMa 7B on the RepoQA dataset

4.4 Key Findings

The analysis conducted helped us getting a thorough understanding of how Large Language Model perform in code generation tasks:

- **Evaluation Metrics**: it is clear that match based metrics and functional ones can't be used for the same purpose. While execution-related metrics give us a more practical understanding of the capabilities of a coding model, match-based ones are still a valid way to evaluate code given a reference. This means that metrics like BLEU, BERTScore, or even better, CodeBERTScore, can be employed in various specific tasks, such as in context retrieval evaluation, as seen with RepoQA. However, they are not as indicative as the pass@k metric when used to evaluate code generation itself because of their sensibility to variations that might not be relevant.
- **Closed Source LLMs** are still undefeated: as shown in table 4.1, the best results are achieved by big closed source models, with GPT-4 still dominating the leaderboard.
- **DeepSeek-Coder** 6.7B and **CodeQwen** 1.5 7B are the best performing low parameter count open source models available: the two models show impressive results compared to other models with the same size.
- A **bigger version** of a model will perform better: while the scaling law is confirmed, the curve of how a model scales varies greatly depending on the LLM itself. While some models greatly benefit from a higher parameter count, others show very low performance increase that doesn't justify the higher computational requirements.
- **Instruction Tuning** is effective: the graph presented in figure 4.5 is a solid proof that a model optimized for multi-turn question answering provides better code than the standard completion model.
- **Fine Tuning** can improve a model's ability to generate code: when trained on more high quality code, LLMs will produce better code.
- **Quantization** introduces an amount of degradation that depends

widely on the model: different models respond in different ways to quantization, with some like CodeLlama not being affected as much as others. In general, the decrease in performance introduced is a trade off that needs to be carefully evaluated before deciding which version to use.

- The **number of bits** used to quantize a model is not as relevant as it may seem: from the analysis done with HumanEval, MBPP and RepoQA, it emerges that the difference between 4 and 8 bits models is less substantial than anticipated, while the performance delta between quantized and unquantized model is very high, especially in context retrieval tasks.

Chapter 5

Conclusion

This research has explored different techniques developed to asses the quality of the code generated by large language models and compared the main LLMs currently available to understand the state of the art of AI code synthesis. The research investigated how the scaling law and the quantization techniques affect the quality of generated code.

RQ1. The comparative analysis revealed that while larger models generally outperform smaller counterparts in complex coding tasks, the lower parameter count models can achieve competitive performance, especially if trained on high quality data. Despite closed source models still being at the top of the leaderboards, which is dominated by GPT-4 Turbo and Cloud 3 Opus, the gap between them and highly specialized open source ones is closing, allowing for a less expensive and more accessible code generation, that opens the doors for new locally runnable automatic program syntheses.

RQ2. Match based metrics have been found not as indicative as functional metrics when it comes to evaluation of generated code. While metrics like BLEU are useful in specific tasks such as retrieval evaluations, functional metrics provide a more practical way to understand if the generated code is right or wrong. This discrepancy is evident from the presented correlation matrix and the score distribution.

RQ3. Out of all the open source models analyzed to evaluate their skills in

code generation, the most reliable ones are CodeQwen 7B and DeepSeek-Coder 6.7B for the lower end of parameter count range. However, if we extend the size range up to 70B parameters, Mixtral 8x22B, Llama 3 70B and DeepSeek-Coder 33B give the highest results. While 70B parameters are 10 times bigger than the smallest counterpart, therefore being less accessible and harder to run in a local configuration, it is also important to take into consideration the rapid scaling that even retail Graphic Processing Units are experiencing. This is making running bigger models easier thanks to the faster speeds and higher quantities of memory built in the hardware, highlighting the possibility of sacrificing a small part of efficiency for a performance improvement, even if not substantial.

RQ4. The process of weight precision reduction leads to a performance degradation that varies across different models. Some models, like CodeLlaMa, maintain high robustness against pruning and quantization, showing consistent performance across all different levels of data precision. Conversely, other models like Mistral exhibit a significant decrease in correctness when quantized, underscoring the need for careful consideration when choosing the version of an LLM to use for code generation.

5.1 Future Work

Future research can build upon this findings by exploring the development of hybrid evaluation metrics based on a small LLM like DeepSeek-Coder. This would involve creating a more refined and context-aware evaluation metric to better understand and improve the performance of different LLMs, particularly in specialized domains such as code generation and natural language processing. Further exploration of model robustness, particularly in the context of quantization and pruning, will also be valuable to optimize LLM performance under various constraints.

5.2 Limitations of the study

While the analysis has been structured to be the most comprehensive possible, it could benefit from a broader selection of models to test. The lists of available models and datasets are also constantly expanding, making any kind of similar research eventually out of date.

5.3 Acknowledgements

I would like to express my appreciation to my supervisor, Professor Albert Gatt, whose expertise and patience added considerably to my graduate experience, allowing me to do research on a topic that I am strongly passionate about. I am also grateful to my peers for their wonderful collaboration and stimulating discussions that we had during this time we shared together.

Bibliography

[1] *2023 Devloper Survey*. en. URL: https://survey.stackoverflow.co/2023.

[2] Meta AI. *Introducing Meta Llama 3: The most capable openly available LLM to date*. URL: https://ai.meta.com/blog/meta-llama-3/.

[3] Mistral AI. *Mixtral of Experts*. Dec. 2023. URL: https://mistral.ai/news/mixtral-of-experts/.

[4] Phind AI. *Beating GPT-4 on HumanEval with a Fine-Tuned CodeLlama-34B*. Aug. 2023. URL: https://www.phind.com/blog/code-llama-beats-gpt4.

[5] Jacob Austin et al. *Program Synthesis with Large Language Models*. 2021. arXiv: 2108.07732 [cs.PL].

[6] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL].

[7] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG].

[8] OpenAccess ai collective. *axolotl: Go ahead and axolotl questions*. URL: https://github.com/OpenAccess-AI-Collective/axolotl.

[9] Tri Dao et al. *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*. 2022. arXiv: 2205.14135 [cs.LG].

[10] Deci.AI. *Model Quantization and Quantization-Aware Training: Ultimate guide*. Jan. 2024. URL: https://deci.ai/quantization-and-quantization-aware-training.

[11] DeepSeek-AI et al. *DeepSeek LLM: Scaling Open-Source Language Models with Longtermism*. 2024. arXiv: 2401.02954.

[12] Michael Denkowski and Alon Lavie. "Meteor Universal: Language Specific Translation Evaluation for Any Target Language". In: *Proceedings of the Ninth Workshop on Statistical Machine Translation*. Ed. by Ondřej Bojar et al. Baltimore, Maryland, USA: Association for Computational Linguistics, June 2014, pp. 376–380. DOI: 10.3115/v1/W14-3348. URL: https://aclanthology.org/W14-3348.

[13] Tim Dettmers et al. *LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale*. 2022. arXiv: 2208.07339 [cs.LG].

[14] Tim Dettmers et al. *QLoRA: Efficient Finetuning of Quantized LLMs*. 2023. arXiv: 2305.14314 [cs.LG].

[15] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL].

[16] Yangruibo Ding et al. *CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion*. 2023. arXiv: 2310.11248 [cs.LG].

[17] Aryaz Eghbali and Michael Pradel. "CrystalBLEU: Precisely and Efficiently Measuring the Similarity of Code". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2023. ISBN: 9781450394758. DOI: 10.1145/3551349.3556903. URL: https://doi.org/10.1145/3551349.3556903.

[18] EvalPlus. URL: https://evalplus.github.io/repoqa.html.

[19] Ahmad Faiz et al. *LLMCarbon: Modeling the end-to-end Carbon Footprint of Large Language Models*. 2023. arXiv: 2309.14393 [cs.CL].

[20] Elias Frantar et al. *GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers*. 2023. arXiv: 2210.17323 [cs.LG].

[21] Sue Gee. *Stack overflow announces AI-powered features*. July 2023. URL: https://www.i-programmer.info/news/99-professional/16487-stack-overflow-announces-ai-powered-features.html.

[22] GitHub. *GitHub Copilot*. URL: https://github.com/features/copilot.

[23] Google. *Gemini breaks new ground with a faster model, longer context, AI agents and more*. 2024. URL: https://blog.google/technology/ai/google-gemini-update-flash-ai-assistant-io-2024.

[24] Mitchell A Gordon, Kevin Duh, and Jared Kaplan. *Data and Parameter Scaling Laws for Neural Machine Translation*. Online and Punta Cana, Dominican Republic, Nov. 2021. DOI: 10.18653/v1/2021.emnlp-main.478. URL: https://aclanthology.org/2021.emnlp-main.478.

[25] Jianping Gou et al. "Knowledge Distillation: A Survey". In: *International Journal of Computer Vision* 129.6 (Mar. 2021), 1789–1819. ISSN: 1573-1405. DOI: 10.1007/s11263-021-01453-z. URL: http://dx.doi.org/10.1007/s11263-021-01453-z.

[26] Andrey Gromov et al. *The Unreasonable Ineffectiveness of the Deeper Layers*. 2024. arXiv: 2403.17887 [cs.CL].

[27] Maarten Grootendorst. Nov. 2023. URL: https://open.substack.com/pub/maartengrootendorst/p/which-quantization-method-is-right.

[28] Suriya Gunasekar et al. *Textbooks Are All You Need*. 2023. arXiv: 2306.11644 [cs.CL].

[29] Dan Hendrycks et al. "Measuring Coding Challenge Competence With APPS". In: *NeurIPS* (2021).

[30] Joel Hestness et al. *Deep Learning Scaling is Predictable, Empirically*. 2017. arXiv: 1712.00409.

[31] JetBrains. *UAST - Unified Abstract Syntax Tree*. URL: https://plugins.jetbrains.com/docs/intellij/uast.html.

[32] Albert Q. Jiang et al. *Mistral 7B*. 2023. arXiv: 2310.06825 [cs.CL].

[33] Carlos E. Jimenez et al. *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?* 2024. arXiv: 2310.06770 [cs.CL].

[34] Jared Kaplan et al. *Scaling Laws for Neural Language Models*. 2020. arXiv: 2001.08361 [cs.LG].

[35] Mike Lewis et al. *BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension*. 2019. arXiv: 1910.13461 [cs.CL].

[36] Yujia Li et al. "Competition-level code generation with AlphaCode". In: *Science* 378.6624 (2022), pp. 1092–1097. DOI: 10.1126/science.abq1158. eprint: https://www.science.org/doi/pdf/10.1126/science.abq1158. URL: https://www.science.org/doi/abs/10.1126/science.abq1158.

[37] Daya Guo Qihao Zhu Dejian Yang Zhenda Xie Kai Dong Wentao Zhang Guanting Chen Xiao Bi Y. Wu Y.K. Li Fuli Luo Yingfei Xiong Wenfeng Liang. *DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence*. 2024. URL: https://arxiv.org/abs/2401.14196.

[38] Chin-Yew Lin. "ROUGE: A Package for Automatic Evaluation of Summaries". In: *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, July 2004, pp. 74–81. URL: https://aclanthology.org/W04-1013.

[39] Ji Lin et al. *AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration*. 2023. arXiv: 2306.00978 [cs.CL].

[40] Ye Lin et al. *Understanding Parameter Sharing in Transformers*. 2023. arXiv: 2306.09380 [cs.LG].

[41] Wang Ling et al. *Latent Predictor Networks for Code Generation*. 2016. arXiv: 1603.06744 [cs.CL].

[42] Jiawei Liu et al. "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation". In: *Thirty-seventh Conference on Neural Information Processing Systems*. 2023. URL: https://openreview.net/forum?id=1qvx610Cu7.

[43] Tianyang Liu, Canwen Xu, and Julian McAuley. *RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems*. 2023. arXiv: 2306.03091 [cs.CL].

[44] Zechun Liu et al. *LLM-QAT: Data-Free Quantization Aware Training for Large Language Models*. 2023. arXiv: 2305.17888 [cs.CL].

[45] Chi-kiu Lo. "YiSi - a Unified Semantic MT Quality Evaluation and Estimation Metric for Languages with Different Levels of Available Resources". In: *Proceedings of the Fourth Conference on Machine Translation (Volume 2: Shared Task Papers, Day 1)*. Ed. by Ondřej Bojar et al. Florence, Italy: Association for Computational Linguistics, Aug. 2019, pp. 507–513. DOI: 10.18653/v1/W19-5358. URL: https://aclanthology.org/W19-5358.

[46] Shuming Ma et al. *The Era of 1-bit LLMs: All Large Language Models are in 1.58 Bits*. 2024. arXiv: 2402.17764 [cs.CL].

[47] OpenAI. *GPT-4 Technical Report*. 2023. arXiv: 2303.08774 [cs.CL].

[48] OpenAI. *Hello, GPT-4o*. 2024. URL: https://openai.com/index/hello-gpt-4o/.

[49] Long Ouyang et al. *Training language models to follow instructions with human feedback*. 2022. arXiv: 2203.02155 [cs.CL].

[50] Kishore Papineni et al. "Bleu: a Method for Automatic Evaluation of Machine Translation". In: *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Ed. by Pierre Isabelle, Eugene Charniak, and Dekang Lin. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, July 2002, pp. 311–318. DOI: 10.3115/1073083.1073135. URL: https://aclanthology.org/P02-1040.

[51] Bo Peng et al. *RWKV: Reinventing RNNs for the Transformer Era*. 2023. arXiv: 2305.13048 [cs.CL].

[52] Maja Popović. "chrF: character n-gram F-score for automatic MT evaluation". In: *Proceedings of the Tenth Workshop on Statistical Machine Translation*. Ed. by Ondřej Bojar et al. Lisbon, Portugal: Association for Computational Linguistics, Sept. 2015, pp. 392–395. DOI: 10.18653/v1/W15-3049. URL: https://aclanthology.org/W15-3049.

[53] Team Qwen. *Code with CODEQWEN 1.5*. 2024. URL: https://qwenlm.github.io/blog/codeqwen1.5/.

[54] Colin Raffel et al. *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. 2023. arXiv: 1910.10683 [cs.LG].

[55] Ricardo Rei et al. "COMET: A Neural Framework for MT Evaluation". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Ed. by Bonnie Webber et al. Online: Association for Computational Linguistics, Nov. 2020, pp. 2685–2702. DOI: 10.18653/v1/2020.emnlp-main.213. URL: https://aclanthology.org/2020.emnlp-main.213.

[56] Ehud Reiter, Chris Mellish, and John Levine. "Automatic generation of technical documentation". In: *Applied Artificial Intelligence an International Journal* 9.3 (1995), pp. 259–287.

[57] Shuo Ren et al. *CodeBLEU: a Method for Automatic Evaluation of Code Synthesis*. 2020. arXiv: 2009.10297 [cs.SE].

[58] Baptiste Rozière et al. *Code Llama: Open Foundation Models for Code*. 2023. arXiv: 2308.12950 [cs.CL].

[59] Maximilian Schreiner. *GPT-4 architecture, datasets, costs and more leaked*. July 2023. URL: https://the-decoder.com/gpt-4-architecture-datasets-costs-and-more-leaked/.

[60] Yuzhang Shang et al. *PB-LLM: Partially Binarized Large Language Models*. 2023. arXiv: 2310.00034 [cs.LG].

[61] Matthew Snover et al. "A Study of Translation Edit Rate with Targeted Human Annotation". In: *Proceedings of the 7th Conference of the Association for Machine Translation in the Americas: Technical Papers*. Cambridge, Massachusetts, USA: Association for Machine Translation in the Americas, Aug. 2006, pp. 223–231. URL: https://aclanthology.org/2006.amta-papers.25.

[62] Emma Strubell, Ananya Ganesh, and Andrew McCallum. *Energy and Policy Considerations for Deep Learning in NLP*. 2019. arXiv: 1906.02243 [cs.CL].

[63] Gemini Team et al. *Gemini: A Family of Highly Capable Multimodal Models*. 2023. arXiv: 2312.11805 [cs.CL].

[64] Teknium1. *OpenHermes 2.5 - Mistral*. 2023. URL: https://huggingface.co/teknium/OpenHermes-2.5-Mistral-7B.

[65] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023. arXiv: 2307.09288 [cs.CL].

[66] Ngoc Tran et al. "Does BLEU Score Work for Code Migration?" In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, May 2019. DOI: 10.1109/icpc.2019.00034. URL: http://dx.doi.org/10.1109/ICPC.2019.00034.

[67] Arya Vaishnavi. *Nvidia CEO thinks AI would kill coding, says "everybody is now a programmer"*. 2024. URL: https://www.hindustantimes.com/world-news/us-news/nvidia-ceo-thinks-ai-would-kill-coding-says-everybody-is-now-a-programmer-101708965034169.html.

[68] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL].

[69] James Vincent. *Meta's powerful AI language model has leaked online - what happens now?* Mar. 2023. URL: https://www.theverge.com/2023/3/8/23629362/meta-ai-language-model-llama-leak-online-misuse.

[70] Yuxiang Wei et al. *Magicoder: Source Code Is All You Need*. 2023. arXiv: 2312.02120 [cs.CL].

[71] Wikipedia contributors. *Bfloat16 floating-point format — Wikipedia, The Free Encyclopedia*. [Online; accessed 28-May-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Bfloat16_floating-point_format&oldid=1216247964.

[72] Jialong Wu et al. *Supercompiler Code Optimization with Zero-Shot Reinforcement Learning*. 2024. arXiv: 2404.16077 [cs.PL].

[73] Pengcheng Yin et al. "Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow". In: *International Conference on Mining Software Repositories*. MSR. ACM, 2018, pp. 476–486. DOI: https://doi.org/10.1145/3196398.3196408.

[74] Maksym Zavershynskyi, Alex Skidanov, and Illia Polosukhin. *NAPS: Natural Program Synthesis Dataset*. 2018. arXiv: 1807.03168 [cs.LG].

[75] Xiaohua Zhai et al. *Scaling Vision Transformers*. 2022. arXiv: 2106.04560.

[76] Biao Zhang et al. "When Scaling Meets LLM Finetuning: The Effect of Data, Model and Finetuning Method". In: *arXiv preprint arXiv:2402.17193* (2024).

[77] Fengji Zhang et al. *RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation*. 2023. arXiv: 2303.12570 [cs.CL].

[78] Tianyi Zhang et al. *BERTScore: Evaluating Text Generation with BERT*. 2020. arXiv: 1904.09675 [cs.CL].

[79] Wayne Xin Zhao et al. *A Survey of Large Language Models*. 2023. arXiv: 2303.18223 [cs.CL].

[80] Lianmin Zheng et al. *Judging LLM-as-a-judge with MT-Bench and Chatbot Arena*. 2023. arXiv: 2306.05685 [cs.CL].

[81] Lianmin Zheng et al. *Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena*. 2023. arXiv: 2306.05685 [cs.CL].

[82] Shuyan Zhou et al. *CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code*. 2023. arXiv: 2302.05527 [cs.SE].

Appendix A

Ethics and Privacy Quick Scan

The Ethics and Privacy Quick Scan of the Utrecht University Research Institute of Information and Computing Sciences was conducted and classified this research as low-risk with no fuller ethics review or privacy assessment required.

Appendix B

Dataset details

B.1 Dataset Examples

B.1.1 HumanEval

The HumanEval dataset, available on GitHub is structured as follows:

- **task_id**: id of the task containing the dataset name and the problem number (i.e. **HumanEval/0**)
- **prompt**: function declaration stating function name, arguments, return type, a docstring that contains what the function is intended to do and a list of examples.

```
from typing import List
def has_close_elements(numbers: List[float], threshold: float) -> bool:
    """ Check if in given list of numbers, are any two numbers closer to each other than given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True
    """
```

- **entry_point**: the function name, useful for automatic execution during the functional evaluation phase (i.e. `has_close_elements`)

- **canonical_solution**: the ground truth solution proposed by the authors.

```
    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True
    return False
```

- **test**: a list of tests saved directly in python code, for convenience during the execution.

```
def check(candidate):
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3)
    == True
    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.05)
    == False
    assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.95) ==
    True
    assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.8) ==
    False
    assert candidate([1.0, 2.0, 3.0, 4.0, 5.0, 2.0], 0.1)
    == True
    assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 1.0) ==
    True
    assert candidate([1.1, 2.2, 3.1, 4.1, 5.1], 0.5) ==
    False
```

B.1.2 MBPP

To simplify and unify the sampling process, the MBPP dataset has been modified from its original version, obtained from GitHub to the same exact structure as HumanEval.

The main differences between the two sets are:

- **prompt**: MBPP's prompt is a string containing the natural language description.

`Write a function to find the minimum cost path to reach (m, n) from (0, 0) for the given cost matrix cost[][] and a position (m, n) in cost[][].`

- **canonical_solution**: the solution contains the whole code

```
R = 3
C = 3
def min_cost(cost, m, n):
    ttc = [[0 for x in range(C)] for x in range(R)]
    tc[0][0] = cost[0][0]
    for i in range(1, m+1):
        tc[i][0] = tc[i-1][0] + cost[i][0]
    for j in range(1, n+1):
        tc[0][j] = tc[0][j-1] + cost[0][j]
    for i in range(1, m+1):
        for j in range(1, n+1):
            tc[i][j] = min(tc[i-1][j-1], tc[i-1][j], tc[i
   ][j-1]) + cost[i][j]
    return tc[m][n]
```

- **entry_point**: While standard MBPP does not have a predefined function, the canonical solution entry point is included in the prompt, so that the automatic execution harness could be used to evaluate both sets (in this example `min_cost`)

B.2 Unused Dataset

The motivation for why the other proposed datasets have not been utilised for model evaluation is reported in section 3.1. However, it is important to specify that APPS and CoNaLa have been used for some preliminary tests before being discarded.

B.2.1 APPS

While APPS is conceptually similar to both HumanEval and MBPP, the main reason why it did not get included in the study is the poor results that all models tested achieved. The following table shows how 5 differ-

ent models' performance on APPS, indicating the average test pass rate limited to the compiled codes, the percentage of problems solved with all test passing and the percentage of compilation and runtime errors derived from faulty code generation.

Model	Test Case Average	Strict Accuracy	Compile Errors	Runtime Errors
CodeLlama 70B*	15.04%	4.5%	23.0%	24.5%
Gemma 7B*	13.18%	1.93%	9.2%	41.3%
Mistral 8x7b*	13.38%	3.54%	33.0%	22.7%
CodeLlama 7B Instruct GPTQ	9.67%	3.07%	19.4%	25.5%
Mistral 7B GPTQ	7.43%	2.38%	44.2%	27.5%

Table B.1: Results on APPS dataset on different models. The models marked with a * are sampled on a limited number of problems.

Despite the results still aligning with the results presented in tables 4.1 and 4.3, the overall low percentages relative to accuracy make them less indicative, not properly highlighting the gaps between models as much as HumanEval and MBPP. This is also due to the complexity of the problems of APPS. Moreover, APPS is composed of 5000 different problems, making the sampling phase very onerous in terms of resources and time.

B.2.2 CoNaLa

The CoNaLa dataset, composed of 2880 problems, is structured as follows:

- **intent**: an extremely short explanation of the problem.
- **rewritten intent**: a more specific explanation of the problem that needs to be solved
- **solution**: a short solution, often being just single line of code.

The first major noticeable difference from the other datasets is the absence of test cases, which make the automatic execution based evaluation impossible. Another major difference is the presence of a rewritten intent. This intent gives the actual description relative to the solution, as shown in the example below.

As we can see, a problem can have multiple different interpretations, which are explained in the rewritten intent field. The problem descriptions, how-

id	intent	rewritten_intent	snippet
1,476	express binary literals	convert 173 to binary string	bin(173)
1,476	express binary literals	convert binary string '01010101111' to integer	int('01010101111', 2)
1,476	express binary literals	convert binary string '010101' to integer	int('010101', 2)
1,476	express binary literals	convert binary string '0b0010101010' to integer	int('0b0010101010', 2)
1,476	express binary literals	convert 21 to binary string	bin(21)
1,476	express binary literals	convert binary string '11111111' to integer	int('11111111', 2)
1,854	What OS am I running on	get os name	import platform platform.system()
1,854	What OS am I running on	get os version	import platform platform.release()
1,854	What OS am I running on	get the name of the OS	print(os.name)

ever, are very broad, while the solutions, being so short, can vary quite substantially. Although this would not be a problem in the context of executable code, it definitely is a discriminating factor when evaluating the generated code using match-based metrics, which are more sensitive to these variations. The absence of standardized functions in the solutions poses a significant issue. Different models, due to variations in their training, tend to encapsulate most solutions within a function. When match-based metrics are used to evaluate the differences between the generated and reference strings, this discrepancy leads to lower scores, despite the underlying code being substantially similar.

Appendix C

Used LLMs Details

C.1 Templates

The templates used for each model during the sampling phase are reported as follows:

- Mistral-7B:

```
<s>[INST] {system_message}: {prompt} [/INST]
```

- CodeLlama (7B and 13B):

```
[INST] {system_message} :
{prompt}
[/INST]
```

- Llama 3 8B:

```
<|begin_of_text|><|start_header_id|>system<|end_header_id|>
{system_message}<|eot_id|>

<|start_header_id|>user<|end_header_id|>
{prompt}<|eot_id|>

<|start_header_id|>assistant<|end_header_id|>
```

- Magicoder-S-DS-6.7B:

```
{system_message}:

@@ Instruction
{prompt}

@@ Response
```

- Phi-2 3B:

```
Instruct: {system_message}. {prompt}
Output:
```