



University of Minho
School of Engineering

Simão Pedro Sá Cunha

Towards a Platform for Benchmarking Large Language Models



University of Minho
School of Engineering

Simão Pedro Sá Cunha

Towards a Platform for Benchmarking Large Language Models

Dissertação de Mestrado
Master's Dissertation in Informatics Engineering

Trabalho efetuado sob a orientação de
João Alexandre Baptista Vieira Saraiva
Francisco José Torres Ribeiro

Abstract

This dissertation details the creation of a platform designed for evaluating Large Language Models (LLMs) through benchmarking coding tasks using well-known datasets such as Human-Eval. The primary goal of this platform is to assess the effectiveness and energy efficiency of LLMs in addressing diverse challenges, with a specific focus on code generation and software development. The document thoroughly examines the project's context, motivation, and objectives, emphasizing the transformative influence of LLMs like GPT-4 and LLAMA-2 on the field of software development. Furthermore, it delves into the challenges posed by the energy consumption of LLMs when represented as quantized models, analyzing their size and weight differences. This exploration underscores the fundamental principles of sustainability within software engineering. Through an extensive analysis of the performance of existing LLMs, this dissertation actively contributes to the progression of both efficient and sustainable software development practices.

Keywords Large Language Models, Energy consumption, Green Software

Resumo

Esta dissertação detalha a criação de uma plataforma projetada para avaliar Large Language Models (LLMs) através de uma análise comparativa. O objetivo principal desta plataforma é avaliar a eficácia e eficiência energética de LLMs ao lidar com diversos desafios de programação de *datasets* conhecidos como o Human-Eval. Este documento aborda o contexto, motivação e objetivos do projeto, realçando a influência de LLMs como o GPT-4 e o LLAMA-2 no desenvolvimento de software. Além disso, explora os obstáculos apresentados pelo consumo de energia de LLMs representados sob a forma de modelos com *quantization* e analisa as diferenças nos seus tamanhos, destacando os princípios fundamentais de sustentabilidade na engenharia de software. Através de uma análise abrangente do desempenho de LLMs existentes, esta dissertação contribui ativamente para o avanço de práticas eficientes e sustentáveis no desenvolvimento de software.

Palavras-chave Large Language Models, Consumo de energia, Green Software

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation	2
1.3	Main Objectives	2
1.4	Expected results	3
2	State of the Art	4
2.1	Green Software	4
2.2	Large Language Models (LLMs)	6
2.3	Large Language Models for Code	8
2.4	Evaluating LLMs' code generations	13
2.4.1	HumanEval	15
2.4.2	HumanEval-X	17
2.5	Executing LLMs Locally	20
3	The problem and its challenges	21
4	Development	22
5	Experiments	25
5.1	Experiment setup	25
5.2	Results	26
5.3	Discussion	27
6	Conclusions and future work	28
7	Planned Schedule	29
7.1	Activities	29

List of Figures

1	System Architecture for Execution and Measurement	24
2	Execution time and HumanEval-X pass@1 of llama-2-7b.Q2_K	26
3	Execution time and HumanEval-X pass@1 of llama-2-7b.Q3_K_L	26
4	CPU Energy and HumanEval-X pass@1 of llama-2-7b.Q2_K	27
5	CPU Energy and HumanEval-X pass@1 of llama-2-7b.Q3_K_L	27

List of Tables

1	LLMs examples from Naveed et al. (2023)	6
2	Use-cases examples of LLMs in the real world, according to Kaddour et al. (2023)	7
3	LLMs for code examples from Agarwal (2023)	13
4	LLMs benchmarks examples - adapted from examples provided on the Llama 2 webpage.	14
5	HumanEval score (%) - <i>pass@k</i> - from Chen et al. (2021)	16
6	HumanEval-X score (%) - <i>pass@k</i> - Code generation - from Zheng et al. (2023)	19
7	Examples of LLMs execution methods	20
8	Dependencies required for the HumanEval-X	23
9	Specifications of the computer used in the experiment	25
10	Activities Plan	29

Acronyms

AI Artificial Intelligence.

CPU Central Process Unit.

CSV Comma Separated Values.

GPU Graphics Processing Unit.

IDE Integrated Development Environment.

IT Information Technology.

JSON JavaScript Object Notation.

LLM Large Language Model.

NLP Natural Language Processing.

PDR Preliminary Dissertation Report.

SOA State of the Art.

Chapter 1

Introduction

1.1 Context

Large Language Models (**LLM**), such as GPT-4 supporting ChatGPT, are transforming the way we create software. These models can aid programmers in code development, much like advanced integrated development environments (**IDE**) utilizing artificial intelligence (**AI**) support as a coPilot (just as ChatGPT relies on an LLM at its core). LLMs have exhibited significant potential as highly competent AI assistants excelling in intricate reasoning tasks that demand expert knowledge across a broad spectrum of fields, including specialized domains like programming and creative writing. They facilitate interaction with humans through user-friendly chat interfaces, leading to swift and widespread adoption among the general public ([Touvron et al. \(2023\)](#)). The integration of these comprehensive language models is transforming how we approach software development, marking the advent of a new era characterized by heightened efficiency and innovation.

Due to the significant success of these large language models, more models have emerged, such as Meta's LLAMA family ([Llama 2](#)), varying in size and capabilities. According to [Naveed et al. \(2023\)](#), from October 2019 until December 2023, approximately 55 LLMs were released.

Creating and training LLMs are computationally intensive tasks, leading to substantial energy consumption ([Naveed et al. \(2023\)](#)). In fact, the principles of sustainability in software engineering and LLMs seem to be moving in opposite directions.

As the number of LLMs continues to grow, it becomes crucial to assess their performance not only in generating or completing programs in various programming languages but also in terms of their energy efficiency. The objective is to define a set of problems commonly addressed by LLMs, expressed as prompts that serve as their inputs. This thesis will conduct a comprehensive study on the performance of all available LLMs in solving these problems.

While computer manufacturers began developing energy-efficient hardware during the early stages of

the global adoption of mobile computing devices, it is only recently that the software engineering community has started expressing concerns about the energy efficiency of software systems ([Pinto and Liu \(2014\)](#)). Today, this topic has become a focal point of research, with defined estimations provided by the [Intel RAPL](#), frameworks such as Qualcomm TrepN ([Abdulsalam et al. \(2015\)](#)), or [CodeCarbon](#), and even the utilization of external energy measurement devices ([McIntire et al. \(2012\)](#)).

The energy efficiency of each LLM will be measured using one of the referenced frameworks, providing insights into which model is the most sustainable for addressing a specific problem, considering factors such as energy consumption and maintenance requirements.

1.2 Motivation

The motivation of this study is to investigate the concept of coding in a more environmentally friendly manner using large language models. While LLMs have demonstrated remarkable capabilities in assisting developers, it is crucial to assess their energy efficiency alongside their programming prowess. Understanding the environmental implications of utilizing LLMs can contribute to the adoption of more sustainable software development practices. By identifying the most energy-efficient models for specific programming tasks, developers can make informed decisions, aligning technological advancements with environmental sustainability.

At its core, this research is motivated by the need to code not just efficiently, but also responsibly, acknowledging the ecological footprint of software development in the age of LLMs.

1.3 Main Objectives

The main objective of this dissertation is a comparative analysis of Large Language Models, taking into account their energy and memory consumption, execution time, and accuracy when evaluated on well-known benchmarks. To achieve this, the following topics need to be addressed:

- Literature review on LLMs, benchmarks, and frameworks for measuring the energy consumption of systems during LLM execution.
- Investigation of the computational resources required to conduct the experiment.
- Coding of scripts that automatically apply various LLMs, ensuring the reproducibility of the obtained results.

- Gathering and storing measurements (e.g., Comma Separated Values - **CSV**).
- Statistical analysis of the obtained results.
- Creating an environment for easy recreation of experiments (e.g. Docker image).

1.4 Expected results

The expected outcome of this dissertation involves conducting a multi-criteria analysis that evaluates the energy and memory consumption, execution time, and accuracy of each large language model across all benchmarks.

Chapter 2

State of the Art

2.1 Green Software

The continuous expansion of software and **IT** usage plays a crucial role in sustaining the dynamism of our society and managing individual lives. However, this growth is accompanied by a substantial increase in energy demand. As highlighted in both [Verdecchia et al. \(2021\)](#) and [Podder et al. \(2020\)](#), it is projected that data centers alone will consume approximately 10% of the global electricity supply by the year 2030. Acknowledging that end-users can only make use of what is provided to them, the responsibility lies with the community of software developers to actively embrace ecological practices. This is where the concept of green software assumes a pivotal role.

As defined by [Bener et al. \(2014\)](#), *greening in software* aims to mitigate the environmental impact caused by the software itself. Green specifications provide a means to quantify a service's carbon footprint and, eventually, specify operational constraints to allow more flexibility during service provisioning.

According to the [Green Software Foundation](#), the advocacy for three primary actions to reduce the carbon emissions of software includes:

- Using fewer physical resources;
- Using less energy;
- Using energy more intelligently.

Using energy more intelligently involves both using lower-carbon energy sources and employing electricity in a way that speeds up the transition to a low-carbon future.

[Ardito et al. \(2015\)](#) elaborated on seven principles for making software environmentally friendly:

1. Clean up obsolete code and data for energy efficiency and maintainability;

2. Manage software lifecycle to prevent unnecessary energy drain from immortal processes or threads (e.g. *starvation*);
3. Monitor and understand hardware power requirements, optimizing software accordingly for energy conservation;
4. Refactor based on common usage scenarios to perceptibly save energy in specific situations;
5. Prioritize higher-level structures in refactoring for more significant impact on **CPU** and memory;
6. Beware of energy-consuming loops; detect and refactor to save energy, especially on battery-powered devices;
7. Optimize data transfer in distributed systems, using compression or aggregation for energy-efficient operations.

An eighth principle arises in the paper by [Pereira et al. \(2021\)](#). The study benchmarks 27 languages across 10 problems, concluding that distinct programming languages result in varying energy consumption values.

[Verdecchia et al. \(2023\)](#) conducted a systematic review of Green AI literature, finding a focus on monitoring model footprint, tuning hyperparameters for sustainability, and benchmarking. This leads to the ninth principle: *choosing the right AI model*.

Finally, [Green Software Foundation](#) introduces the tenth principle: monitor real-time power consumption during development using techniques like dynamic code analysis. Gathering critical data helps understand gaps between design choices and actual energy profiles. Tools such as [Intel RAPL](#) (Running Average Power Limit) or [CodeCarbon](#) can assist in this process.

2.2 Large Language Models (LLMs)

According to [Hadi et al. \(2023\)](#), Large Language Models (LLMs) *are a type of artificial intelligence (AI) that has emerged as powerful tools for a wide range of tasks, including natural language processing (NLP), machine translation, and question-answering.* These models derive their efficacy from being "large" indicating extensive training on vast datasets.

The training process of LLMs involves exposing them to substantial amounts of data, often sourced from the Internet. While the quantity of data is substantial, the quality plays a crucial role in determining the model's proficiency. In some cases, a curated dataset is employed to ensure a higher standard of learning ([Rozière et al. \(2023\)](#)). The training aims to develop a nuanced comprehension of human language and intricate data, empowering Large Language Models to decode the complexities embedded in characters, words, and sentences.

Following the initial training, LLMs may undergo a fine-tuning or prompt-tuning process. This phase tailors the model to excel in specific tasks, aligning with the programmer's objectives ([Kaddour et al. \(2023\)](#)).

In the following table, it will be presented some examples of LLMs and their respective creators are presented:

LLM	Creator
GPT-4	OpenAI
Llama 2	Meta
Bard	Google
Claude	Anthropic
Vicuna	LMSYS
Bloom	BloomAI

Table 1: LLMs examples from [Naveed et al. \(2023\)](#)

Large Language Models find applications in various domains. In the following table, we will provide a brief explanation of each area, highlight common use cases for LLMs within these domains, and identify what LLMs are commonly used in each domain area:

Domain area	Domain area description	LLMs used
Chatbots	Information retrieval, multi-turn interaction, and text generation (including code) between human and an AI agent	GPT-3.5 & GPT-4
Computational Biology	Protein embeddings generations from amino-acid or genomic sequence inputs and genomic analysis by an AI agent	ESM-1 & ESM-2
Computer Programming	Generation and completion of computer programs in various programming languages	Codex & Code Llama
Creative Work	Story, script and screenplay generation without human interaction	Chinchilla 70B & GPT-3.5
Knowledge Work	Cognitive tasks requiring analysis, problem-solving, and intellectual capabilities define knowledge work	GPT-3.5 & Galactica
Law	Legal question answering, legal information extraction and case outcome prediction	GPT-3.5 & GPT-4
Medicine	Medical question answering, clinical information extraction, indexing, triage and management of health records	PaLM & PubMedGPT
Reasoning	Mathematical and algorithmic tasks done by an AI agent	Codex & GPT-3
Robotics	Giving instructions to robots	Codex & GPT-3
Social Sciences	Modeling and analyzing human and LLM behavior and simulating relationships with LLMs	GPT-3 & BERT
Synthetic Training Data	Datasets generation	GPT-4

Table 2: Use-cases examples of LLMs in the real world, according to [Kaddour et al. \(2023\)](#)

In the domain of modern technology, numerous applications are built around the foundational integration of Large Language Models into their core functionalities. The forthcoming list will highlight various applications that prominently incorporate LLMs:

- [ChatGPT](#) - AI-powered chatbot - conversational interactions with advanced language understanding.
- [Google Bard](#) - AI-powered chatbot - facilitates interactive and dynamic user conversations.
- [Github CoPilot](#) - AI developer tool - enhances coding productivity with intelligent auto-completion.

- [Microsoft Bing Browser](#) - AI search tool - optimizes speed, accuracy, and efficiency in search engine usage.
- [ChatPDF](#) - AI-powered chatbot - interacts based on user content within PDF files.
- [Tabnine](#) - AI developer tool - boosts coding efficiency with powerful auto-completion capabilities.
- [Amazon Codewhisperer](#) - AI developer tool - enhances coding experience with intelligent auto-completion.

2.3 Large Language Models for Code

We perceive *programming* as a technical procedure in which humans provide instructions in a specific dialect to computers for solving a given problem. This involves creating a file containing a set of instructions that is then presented to a compiler or an interpreter of a programming language, enabling the computer to comprehend the steps required to address a particular problem. Programmers (or developers), individuals who compose these instructions, have the flexibility to code in various programming languages such as C, C++, C#, Python, Java, JavaScript, Ruby, Rust, Fortran, Pascal, and more.

Advantages of using Large Language Models (LLMs) for programming were highlighted by [Tarassow \(2023\)](#), including code creation, documentation, explanation, and description. For example, LLMs can aid developers in tasks like naming variables and functions, and assisting with code refactoring and bug¹ fixing.

In the table below, [Rozière et al. \(2023\)](#) utilized Code Llama - Instruct 34B to exemplify the usage of Large Language Models (LLMs) in programming assistance. This model was specifically fine-tuned using instructions in natural language, making it preferable for code generation tasks, as it was optimized to generate helpful and secure responses in natural language.

¹ It refers to an error, flaw, or unintended behavior in a software program or system that causes it to produce incorrect or unexpected results.

Prompt: This function should return a list of lambda functions that compute successive powers of their input, but it doesn't work:

```
1 def power_funcs(max_pow):  
2     return [lambda x: x**k for k in range(1, max_pow+1)]  
3
```

The function should be such that `[h(2) for f in powers(3)]` should give `[2, 4, 8]`, but it currently gives `[8, 8, 8]`. What is happening here?

Response: The problem is that the lambda functions are all using the same variable 'k', which is being modified by the loop. By the time the lambda functions are called, the loop has already finished and 'k' is equal to 'max_pow+1'. Therefore, all the lambda functions are using the same value for 'k', which is why they all return the same result.

To fix this, you can use a default argument to create a new variable for each lambda function:

```
1 def power_funcs(max_pow):  
2     return [lambda x, k=k: x**k for k in range(1, max_pow+1)]  
3
```

This will create a new variable 'k' for each lambda function, with the value of 'k' at the time the lambda function was created.

It is relevant to highlight an important concept after analyzing the previous table, which is *prompt engineering*. According to [Chen et al. \(2023\)](#), prompt engineering is *the process of structuring input text for LLMs and is a technique integral to optimizing the efficacy of LLMs*. They also present some ideas that we have to consider when we are doing prompt engineering:

- **Giving instructions:** We must type a text (a *prompt*) to provide guidance to the LLM on how to perform a specific task, i.e., explaining what we want to achieve.
- **Be clear and precise:** Formulate prompts that are unambiguous and precise, guiding the model toward generating the desired output.

Prompt: I want to understand some cutting-edge aspects of technology

Response: Here are some of the key areas:

1. Artificial intelligence and machine learning: These technologies are becoming increasingly sophisticated, with advancements in areas such as natural language processing (like the technology you're interacting with right now), computer vision, and reinforcement learning. Techniques such as transfer learning and federated learning are being used to make machine learning models more efficient and less data-hungry.
2. Quantum computing: Quantum computers use the principles of quantum mechanics to process information. They hold the potential to solve certain types of problems much more efficiently than classical computers. Companies like Google, IBM, and many startups are investing heavily in this area.
3. 5G and beyond: 5G wireless technology began widespread deployment in 2020 and 2021, bringing faster speeds, lower latency, and the ability to connect more devices simultaneously. Research is also being done on 6G, which is expected to integrate even more deeply with AI and potentially bring about more transformative changes.
4. Biotechnology: ...

Giving instructions with more precision (from [Chen et al. \(2023\)](#))

Prompt: I want to understand the cutting edge of technology, **specifically related to artificial intelligence and machine learning. Recently, artificial intelligence has become extremely popular, especially the large language models which have amazed me. Please analyze the possible reasons behind the rapid advancement of this technology from at least three perspectives.**

Response: Absolutely, the rise of AI and machine learning, especially with regard to large language models like GPT-4, can be seen as a convergence of several key factors. I'll analyze this from three perspectives: technological advancements, availability of data, and socio-economic demand.

1. Technological advancements: The growth of AI and machine learning models has been facilitated by several technological innovations. For starters, we've seen incredible improvements in computational power, particularly in Graphics Processing Units (GPUs) and now even more specialized AI chips. These advances have made it ...
2. Availability of data: ...
3. Socio-economic demand: ...

- **Do role-prompting:** Assign a specific role to the LLM to enable more accurate and detailed responses.

Role prompting example (from [Chen et al. \(2023\)](#))

Prompt: You are an expert in artificial intelligence specializing in large language models...

Response: Here are five basic methods to quickly get started with large language models:

1. Experiment with web-based interfaces:
2. ...

- **Use of triple quotes:** It is recommended to use triple quotes to separate different parts of a prompt or to encapsulate multi-line strings. This is commonly used in *docstrings*² for automatic code repairing.

² A brief text within code to describe a function, method, or module's purpose, providing documentation for better understanding and usage by developers.

- **Try several times:** Due to the non-deterministic nature of LLMs, it is often beneficial to try generating responses several times.
- **One-shot vs few-shot prompting:** In one-shot prompting, the model is given a single example to learn from, while in few-shot prompting, multiple examples are provided for learning.

One-shot prompting example (from [Chen et al. \(2023\)](#))

Prompt: Q: Yes or no: is it common to see frost during some college commencements? A: The answer is yes. Q: Yes or no: would a pear sink in water? **Response:** A: The answer is No

Few-shot prompting example (from [Tam \(2023\)](#))

Prompt: Text: Today the weather is fantastic Classification: Pos Text: The furniture is small. Classification: Neu Text: I don't like your attitude Classification: Neg Text: That shot selection was awful Classification:

Response: Text: Today the weather is fantastic Classification: Pos Text: The furniture is small. Classification: Neu Text: I don't like your attitude Classification: Neg Text: That shot selection was awful Classification: Neg

[Tarassow \(2023\)](#) suggested that, due to limited research, high-resource general-purpose programming languages, such as Python or C++, generally outperform their low-resource counterparts. This is attributed to insufficient data for fine-tuning LLMs, making them more challenging to process compared to high-resource languages.

Building on this observation, a response to the identified challenges has been the emergence of LLMs specifically tailored for coding tasks. The models outlined in Table 3 exemplify this trend, demonstrating a focused approach to understanding and generating code. These LLMs aim to address the intricacies of programming languages while mitigating the limitations associated with low-resource environments.

LLM	Authors	Year
Code2Vec	Alon et al. (2019)	2018
CodeBERT	Feng et al. (2020)	2020
Codex	Chen et al. (2023)	2021
CodeT5	Wang et al. (2021)	2021
PLBart	Ahmad et al. (2021)	2021
Code Llama	Rozière et al. (2023)	2023

Table 3: LLMs for code examples from [Agarwal \(2023\)](#)

To illustrate the effectiveness of employing an LLM to assist developers in addressing code issues, [Ribeiro et al. \(2022\)](#) utilized CodeGPT — LLM specifically designed for coding. This model application automatically repaired faulty programs by leveraging its code completion capabilities. The authors also introduced strategies for identifying optimal positions for code completion within defective lines and addressed the challenge of seamlessly integrating the newly generated code.

2.4 Evaluating LLMs' code generations

As highlighted in Table 2, Large Language Models (LLMs) boast numerous applications in daily life. The performance of an LLM varies significantly depending on the specific task it is deployed for. For example, if an LLM is extensively trained with coding data, it will likely provide more accurate responses to programming questions compared to inquiries in common sense areas.

Hence, a brief overview of LLMs benchmarking in various areas is presented in the following table:

Domain area	Benchmark name	Benchmark description
Language Understanding	MMLU - Massive Multitask Language Understanding (Hendrycks et al. (2021))	Developed comprehensive multitask test with diverse multiple-choice questions.
Reading Comprehension	TriviaQA (Joshi et al. (2017))	Challenging dataset with complex questions and diverse syntactic variability.
Open-domain QA	Natural Questions (Kwiatkowski et al. (2019))	Requires comprehension of entire Wikipedia articles, enhancing realism.
Math Problem Solving	GSM8K - Grade School Math 8K (Cobbe et al. (2021))	Grade school math problems, 2-8 steps, elementary arithmetic.
Program Synthesis	HumanEval (Chen et al. (2021))	Gauges correctness in program synthesis from docstrings.
Artificial General Intelligence	AGIEval (Zhong et al. (2023))	Assesses models in human-centric exams for comprehensive evaluation.
Question Answering	BoolQ (Clark et al. (2019))	Yes/no questions, 15942 examples, annotated for comprehensibility and accuracy.
Commonsense Reasoning	HellaSwag - Harder Endings (Zellers et al. (2019))	Grounded commonsense inference, 70k questions predicting next events.
Open-book Exam	OpenBookQA (Mihaylov et al. (2018))	5957 questions, 1326 science facts, mirroring open-book exam scenarios.
Dialog-driven QA	QuAC - Question Answering in Context (Choi et al. (2018))	Designed for information-seeking tasks, interactive conversations, open-ended queries.
Commonsense Reasoning	Winogrande (Sakaguchi et al. (2019))	44k problems refining Winograd Schema Challenge, evaluating machine capabilities.

Table 4: LLMs benchmarks examples - adapted from examples provided on the [Llama 2](#) webpage.

In the following sections, we will discuss the benchmarks considered for the study, with a primary focus on LLMs for code.

2.4.1 HumanEval

HumanEval (Chen et al. (2021)) is a dataset designed to evaluate functional correctness and measure the problem-solving capabilities of large language models, with a specific focus on synthesizing programs from docstrings. Notably, the programming problems are composed in Python and incorporate English natural text within comments and docstrings³. Developed by engineers and researchers at OpenAI, the dataset consists of 164 hand-written programming problems. Each problem includes a function signature, docstring, body, and several unit tests, with an average of 7.7 tests per problem. The tasks within the HumanEval dataset assess language comprehension, reasoning, algorithms, and basic mathematics. Meticulously crafted, this dataset aims to facilitate the evaluation of functional correctness and problem-solving capabilities in large language models.

The emergence of a metric from this dataset is noteworthy: *pass@k*. It measures the proportion of problems for which at least one out of *k* generated samples passes the unit tests. In other words, if a large language model generates *k* solutions to a programming problem, *pass@k* measures the percentage of those solutions that pass the unit tests, selecting the best solution from multiple generated samples. In the following table, we present scores - higher scores indicate better performance - from some LLMs obtained in the HumanEval benchmark, as reported in the study by Chen et al. (2021).

³ The GitHub repository code can be found here: <https://github.com/openai/human-eval/tree/master>.

Model	k = 1	k = 10	k = 100
GPT-NEO 125M	0.75	1.88	2.97
GPT-NEO 1.3B	4.79	7.47	16.30
GPT-NEO 2.7B	6.41	11.27	21.37
GPT-J 6B	11.62	15.74	27.74
TABNINE	2.58	4.35	7.59
CODEX-12M	2.00	3.62	8.58
CODEX-25M	3.21	7.1	12.89
CODEX-42M	5.06	8.8	15.55
CODEX-85M	8.22	12.81	22.4
CODEX-300M	13.17	20.37	36.27
CODEX-679M	16.22	25.7	40.95
CODEX-2.5B	21.36	35.42	59.5
CODEX-12B	28.81	46.81	72.31

Table 5: HumanEval score (%) - $pass@k$ - from [Chen et al. \(2021\)](#)

Each Large Language Model (LLM) confronted with this dataset received a JavaScript Object Notation (**JSON**) file containing entries crucial for this benchmark. Each entry follows a specific structure, as illustrated in Listing [A.1](#):

- *task_id*: identifier for the data sample;
- *prompt*: input for the model containing function header and docstrings;
- *canonical_solution*: solution for the problem in the prompt;
- *test*: contains function to test generated code for correctness;
- *entry_point*: entry point for test.

When utilizing the GitHub repository code from [Chen et al. \(2021\)](#), specific steps must be followed to obtain the $pass@k$ value:

1. **Generating Samples File:** Employing the provided Python code recommended by the authors, a dictionary with two elements - *task_id* (the identifier for the data sample) and *completion* (the

code generated by the LLM without the header and docstring) - needs to be created and saved in a JSON file. It is essential to note that the variable `num_samples_per_task` repeats entries for a specific prompt as many times as its value. The code for generating samples is as follows:

```
1 from human_eval.data import write_jsonl, read_problems
2
3 problems = read_problems()
4
5 num_samples_per_task = 200
6 samples = [
7     dict(task_id=task_id, completion=generate_one_completion(problems[task_id]["prompt"]))
8     for task_id in problems
9     for _ in range(num_samples_per_task)
10 ]
11 write_jsonl("samples.jsonl", samples)
```

Listing 2.1: Samples Generation Python Code

2. **Evaluating samples:** Execute the command:

```
1 python evaluate_functional_correctness.py data/example_samples.jsonl --problem_file=
   data/example_problem.jsonl
2
```

In this context, *evaluate_functional_correctness.py* is the script responsible for the functional correctness evaluation task, *data/example_samples.jsonl* is the JSON file generated earlier, and *data/example_problem.jsonl* is the original dataset. The output will resemble the following:

```
1 Reading samples...
2 6it [00:00, 3397.11it/s]
3 Running example suites...
4 100%|...| 6/6 [00:03<00:00, 1.96it/s]
5 Writing results to data/example_samples.jsonl_results.jsonl...
6 100%|...| 6/6 [00:00<00:00, 6148.50it/s]
7 {'pass@1': 0.4999999999999999}
```

Listing 2.2: HumanEval Benchmark Output Example

3. **Analysing pass@k values:** With all the *pass@k* values obtained, the next step is left to the user: either simply analyze the obtained values or extract the values to place in another file (e.g., CSV file) for further comparative analysis.

2.4.2 HumanEval-X

Zheng et al. (2023) introduced an enhanced version of the HumanEval benchmark⁴. Unlike the pre-

⁴ The GitHub repository code can be found here: <https://github.com/THUDM/CodeGeex/tree/main/codegeex/benchmark>.

vious iteration, which exclusively featured Python code, the newly developed HumanEval-X benchmark aims to assess the capabilities of multilingual code models in tasks such as code generation and translation, focusing on functional correctness. This expansion facilitates the exploration and enhancement of pre-trained (multilingual) code models.

In the original HumanEval dataset, limited to Python, the team manually transformed each problem into a multilingual context. For every problem initially designed solely for Python, prompts, canonical solutions, and test cases were meticulously rewritten in C++, Java, JavaScript, and Go. HumanEval-X encompasses a comprehensive set of 820 handcrafted problem-solution pairs, comprising 164 distinct problems, each accompanied by solutions in five languages.

When revising each problem, adherence to a set of guidelines was imperative:

- Employ established naming conventions, such as `CamelCase` in Java, Go, and JavaScript, and `snake_case` in C++;
- Place docstrings before the function declaration in Java, JavaScript, C++, and Go;
- Adjust symbols within docstrings, including the substitution of single quotes with double quotes in certain languages, and the replacement of keywords like `True/False` or `None`;
- Modify test cases to align with language-specific behaviors, steering clear of imposing uniform outcomes across diverse programming languages;
- Acknowledge and accommodate language-specific behaviors. For instance, when converting an integer to a binary string, the Python method `bin` appends the prefix `"0b"` before the string, a distinction from the Java method `Integer.toString`, which lacks this prefix. These nuances are considered and adjusted in test cases for each programming language.

In this benchmark, two distinct tasks are available: code generation and code translation. In the code generation task, the model receives a problem description as input and generates the corresponding solution in the selected language. On the other hand, the code translation task involves taking the implementation of a problem in the source language and generating its equivalent implementation in the target language.

Similar to the HumanEval benchmark in [Chen et al. \(2021\)](#), the metric used to assess the functional correctness of code generated by an LLM is *pass@k*. Table 6 presents HumanEval-X scores for all five programming languages across six LLMs during the code generation task:

Language	Metric	GPT-J -6B	GPT-NeoX -20B	InCoder -6.7B	CodeGen-Multi -6B	CodeGen-Multi -16B	CodeGeeX -13B
Python	pass@1	11.10%	13.83%	16.41%	19.41%	19.22%	22.89%
	pass@10	18.67%	22.72%	26.55%	30.29%	34.64%	39.57%
	pass@100	30.98%	39.56%	43.95%	49.63%	55.17%	60.92%
C++	pass@1	7.54%	9.90%	9.50%	11.44%	18.05%	17.06%
	pass@10	13.67%	18.99%	19.30%	26.23%	30.84%	32.21%
	pass@100	30.16%	38.75%	36.10%	42.82%	50.90%	51.00%
Java	pass@1	7.86%	8.87%	9.05%	15.17%	14.95%	20.04%
	pass@10	14.37%	19.55%	18.64%	31.74%	36.73%	36.70%
	pass@100	32.96%	42.23%	40.70%	53.91%	60.62%	58.42%
JavaScript	pass@1	8.99%	11.28%	12.98%	15.41%	18.40%	17.59%
	pass@10	16.32%	20.78%	22.98%	27.92%	32.80%	32.28%
	pass@100	33.77%	42.67%	43.34%	48.81%	56.48%	56.33%
Go	pass@1	4.01%	5.00%	8.68%	9.98%	13.03%	14.43%
	pass@10	10.81%	15.70%	13.80%	23.26%	25.46%	25.68%
	pass@100	23.70%	32.08%	28.31%	41.01%	48.77%	47.14%

Table 6: HumanEval-X score (%) - *pass@k* - Code generation - from [Zheng et al. \(2023\)](#)

To enable submissions of LLMs to this dataset and benchmark, a Docker image⁵ is provided in their GitHub repository. This image, or alternatively, a Dockerfile⁶, includes all necessary components for running the application. The authors strongly recommend using a secure environment.

This dataset consists of several JSON files, similar to the HumanEval dataset. Each entry follows a structured format - in listings [A.2](#), [A.3](#), [A.4](#), [A.5](#), and [A.6](#), the same prompt is presented for all five programming languages with the appropriate treatment in JSON format:

- *task_id*: Programming language and numerical problem ID (e.g., Python/0 represents the 0-th problem in Python).
- *declaration*: Function declaration, including necessary libraries or packages.
- *docstring*: Description specifying the functionality and example input/output.

⁵ Lightweight, standalone, and executable software package that includes all the necessary code, runtime, libraries, and dependencies for running an application.

⁶ Script containing instructions to build a Docker image.

- *prompt*: Function declaration and its docstring.
- *canonical_solution*: Verified solution to the problem.
- *test*: Test program, including test cases.

Given that the metric is the same as used in the HumanEval benchmark and this dataset is an extended version of that, the methodology to obtain the $pass@k$ values is very similar.

2.5 Executing LLMs Locally

Executing large language models poses a significant challenge due to their substantial computational and memory requirements. To address this challenge and enable the execution of LLMs on user devices with limited computational power, various solutions have been proposed by different authors. Table 7 presents examples of such solutions:

System	Author
QiGen	Pegolotti et al. (2023)
LLaMa.cpp	Gerganov (2023)
LLM.int8()	Dettmers et al. (2022)
GPTQ	Frantar et al. (2023)
SmoothQuant	Xiao et al. (2023)

Table 7: Examples of LLMs execution methods

Chapter 3

The problem and its challenges

In this thesis, the primary objective is to compare Large Language Models (LLMs) by executing several benchmarks discussed in Chapter 2. Specifically, I addressed the HumanEval and HumanEval-X benchmarks at the point of the PDR delivery, with the expectation of extending the comparison to include more benchmarks, such as CyberSecEval¹. The goal is to analyze both energy consumption and execution time to provide an overview of whether a particular LLM is more suitable for a given task compared to other LLMs. This comparison encompasses various factors, including energy consumption, execution time, CPU, RAM, GPU power usage, and the estimated CO_2 emissions. These factors are estimated using the CodeCarbon framework, while other tools such as Intel RAPL provide less factors. However, different frameworks will be explored in the future. Another established objective is to create a Docker image containing all the necessary packages for results recreation. This image will also enable users to test a specific LLM with a designated prompt, capturing information on execution time and energy consumption. However, a detailed exploration of this aspect will be undertaken at a later stage of this study.

An identified challenge lies in the limited storage available to all the Large Language Model files. Currently, we were utilizing the Hugging Face website, which provides various .gguf files, specifically those from the Llama-2-7B². To employ all these models simultaneously, a minimum of 51.22 GB of available disk space is required. Considering the ideal scenario involves running numerous model variants from various LLMs locally, the scarcity of memory presents a genuine challenge. This challenge may be effectively addressed by utilizing a server (for example), accessible through an SSH connection, equipped with better GPUs and VRAM/RAM and disk offering the greatest available memory capacity for use in my study.

¹ The GitHub repository can be found here: <https://github.com/facebookresearch/PurpleLlama/tree/main/CybersecurityBenchmarks>

² .gguf files from Llama-2-7B can be found here: <https://huggingface.co/TheBloke/Llama-2-7B-GGUF/tree/main>

Chapter 4

Development

This section outlines the procedure for generating a CSV file containing all the necessary measurements. The initial step involved selecting a system capable of executing various Large Language Model (LLM) families, such as Llama, GPT, Vicuna, etc. From the models referenced in Table 7, `llama.cpp` was chosen.

In the `llama.cpp` GitHub repository ([Gerganov \(2023\)](#)), I identified a [Python binding for LLaMa.cpp](#) mentioned in the documentation, and I followed the provided instructions for installation on my computer. In Listing 4.1, an example illustrates the execution of an LLM by providing a prompt and configuring additional parameters.

```
1 from llama_cpp import Llama
2 llm = Llama(model_path="./models/7B/llama-model.gguf")
3 output = llm(
4     "Q: Name the planets in the solar system? A: ", # Prompt
5     max_tokens=32, # Generate up to 32 tokens
6     stop=["Q:", "\n"], # Stop generating just before the model would generate a new question
7     echo=True # Echo the prompt back in the output
8 )
9 print(output) # Dictionary with response details
```

Listing 4.1: Llama.cpp example usage via [Python binding for LLaMa.cpp](#)

Having established a method for executing LLMs, the subsequent step involved determining the benchmarks suitable for evaluating the LLMs. For the PDR phase, I opted to utilize HumanEval-X instead of the original benchmark created by OpenAI, as detailed in [Chen et al. \(2021\)](#). HumanEval-X represents an upgraded version, encompassing evaluations in Python, C++, Go, Java, and JavaScript. In order to execute this benchmark, it is recommended by the authors to use a secure environment like a Docker image, since it requires the following packages:

Dependency	Version
Python	3.8.12
JDK	18.0.2.1
Node.js	16.14.0
js-md5	0.7.3
C++	11
g++	7.5.0
Boost	1.71.0
OpenSSL	3.0.0
go	1.18.4

Table 8: Dependencies required for the HumanEval-X

A Docker image has been provided by the authors, accessible via the command `docker pull rishubi/codegeex:latest`. However, after spending several hours attempting to execute the benchmark, I observed that the repository inside the image was outdated¹, making it impossible to run. Additionally, I identified a code bug reported by another user along with a proposed solution², but as of now, it has not been addressed and fixed. As a solution, I took the initiative to update the Dockerfile provided, incorporating the necessary bug fixes. Following these updates, I was successful in obtaining the desired `pass@k` values.

The next step involves selecting a tool for measuring (or estimating) the energy consumption of Large Language Models (LLMs) during the execution of various datasets. Among the identified frameworks, namely [Intel RAPL](#) and [CodeCarbon](#), I opted for the latter due to its provision of more comprehensive information. Notably, CodeCarbon offers details such as CO_2 emissions and CO_2 emissions rate, which are not available in RAPL. CodeCarbon provides two measurement methods: online and offline. Although differences between these methods will be explored in future work, for the current study, I chose the offline measurement method to eliminate the need for an internet connection.

In Listing 4.2, an example illustrates the usage of CodeCarbon. It requires a mandatory argument, `country_iso_code`, representing the ISO country code where the LLM would hypothetically be executed in the real world. I believe this parameter is used to calculate CO_2 emissions in a real-world context, but further investigation is needed.

¹ See more details here: <https://github.com/THUDM/CodeGeeX/issues/223> (Accessed: 16 January 2024)

² See more details here: <https://github.com/THUDM/CodeGeeX/pull/214> (Accessed: 16 January 2024)

```

1 country_code = None # Define ISO country code
2 tracker = OfflineEmissionsTracker(country_iso_code=country_code)
3 tracker.start()
4 # Code under measurement
5 tracker.stop()
6 print(tracker) # Construct that contains all the measurements details

```

Listing 4.2: CodeCarbon example usage

In this phase, the following metrics will be collected using CodeCarbon:

- Execution time (s)
- CPU Energy (J)
- RAM Energy (J)
- GPU Energy (J)
- CPU Power (W)
- RAM Power (W)
- GPU Power (W)
- CO_2 emissions (Kg)
- CO_2 emissions rate (Kg/s)

In the following figure, a simplified architecture of the developed code is presented.

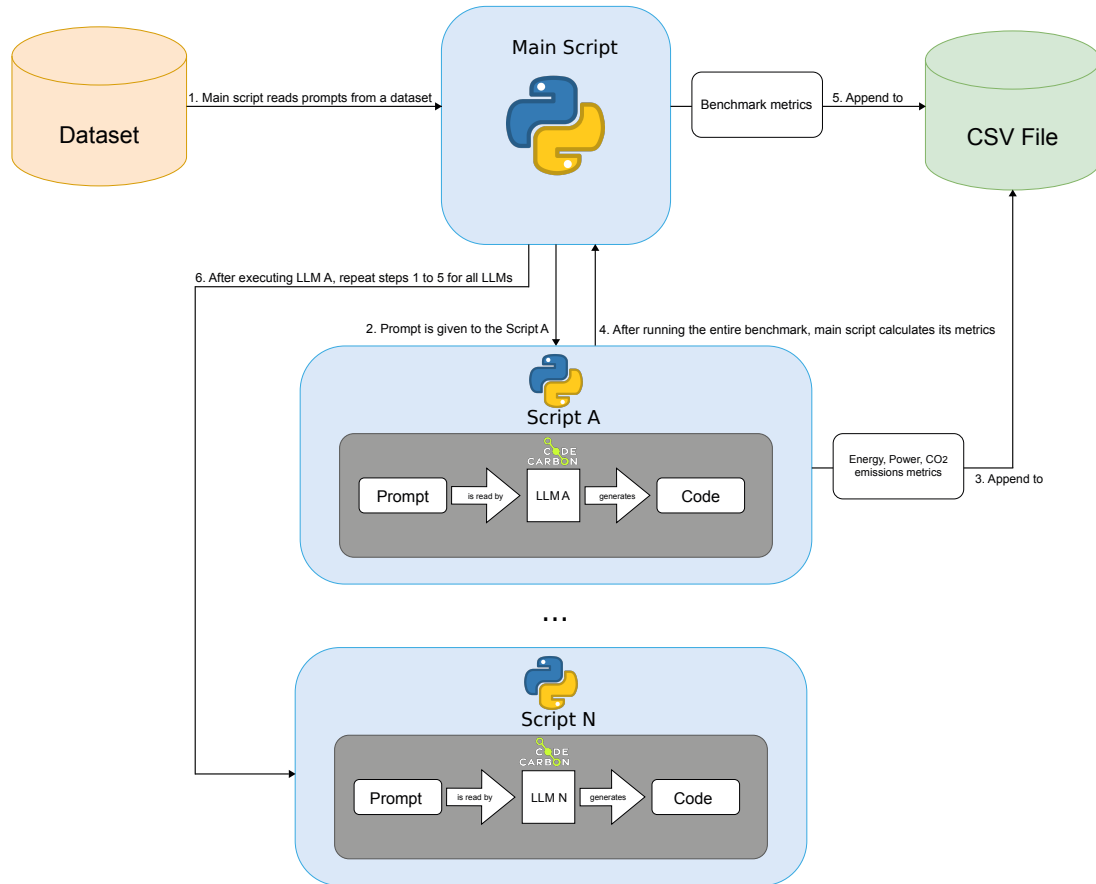


Figure 1: System Architecture for Execution and Measurement

Chapter 5

Experiments

5.1 Experiment setup

In order to conduct preliminary tests on the developed code, I executed it on my personal computer. It is important to note that all the LLMs were built exclusively for CPU usage. In the future, Graphics Processing Units (**GPU**) will be used instead. Another crucial point to observe is that, during the program's execution, only the essential processes for minimal execution need to be active. However, in this experiment, I only took the following steps to minimize unnecessary energy consumption:

- Disable internet connection.
- Disable Bluetooth connections.
- Turn off keyboard LEDs.
- Remove all peripherals connected to the computer.
- Set minimum brightness on the computer.

Operating System	Linux Ubuntu 22.04
Processor	Intel(R) Core(TM) i7-8750H
Clockspeed	2.2 GHz
Turbo Speed	4.1 GHz
Cores	6
Threads	12
RAM	16GB
RAM Speed	2666 mt/s
Cache Size	L1: 384K, L2: 1.5MB, L3: 9MB

Table 9: Specifications of the computer used in the experiment

5.2 Results

In this experiment, we will use two LLMs (llama-2-7b.Q2_K and llama-2-7b.Q3_K_L) and will compare the HumanEval-X pass@1 score, considering both energy consumption and execution time during output generation.

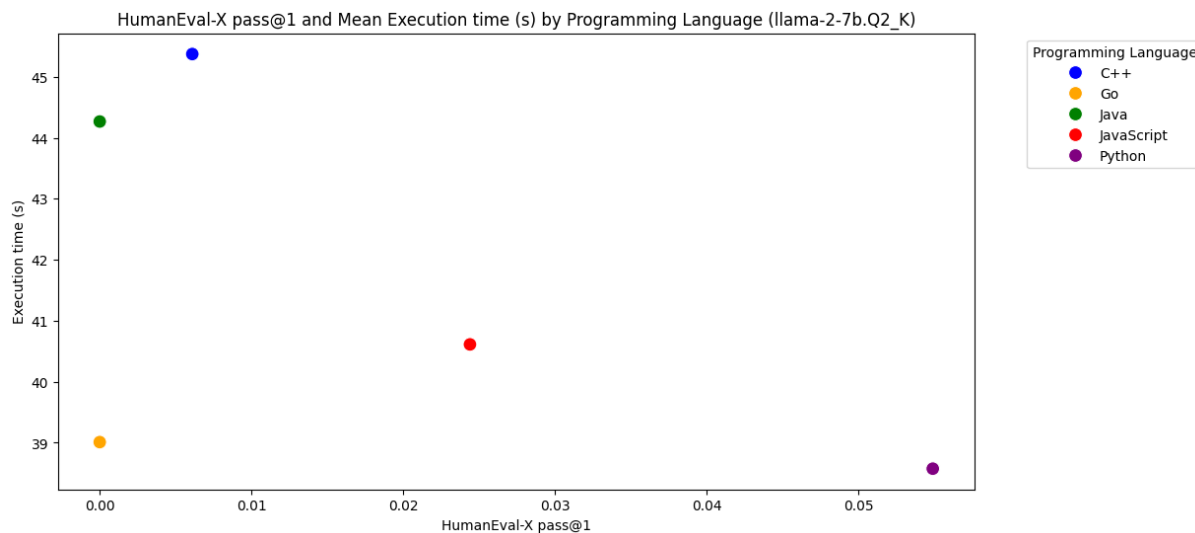


Figure 2: Execution time and HumanEval-X pass@1 of llama-2-7b.Q2_K

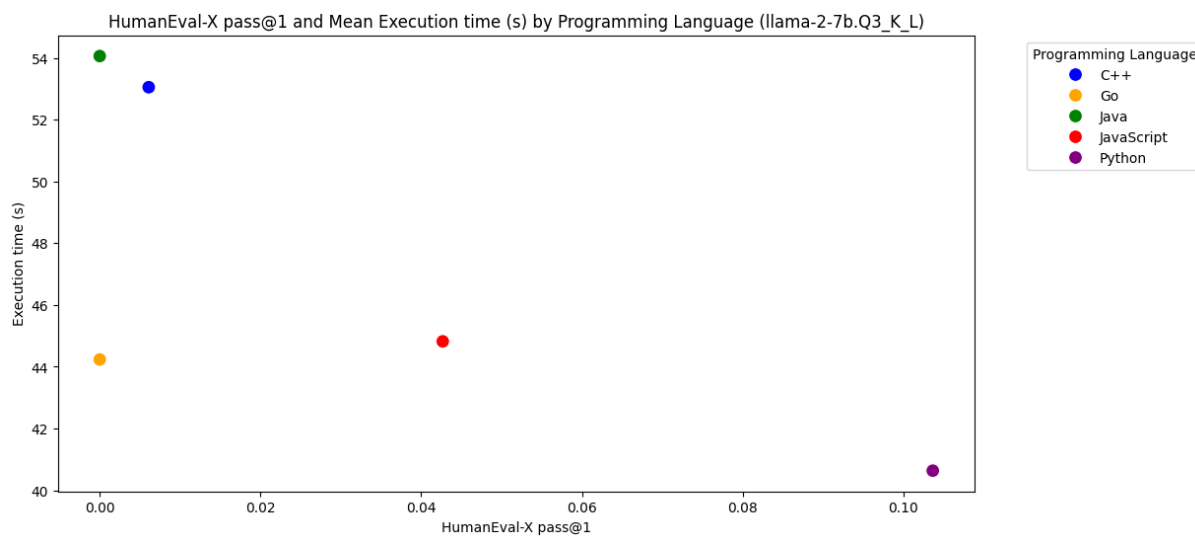


Figure 3: Execution time and HumanEval-X pass@1 of llama-2-7b.Q3_K_L

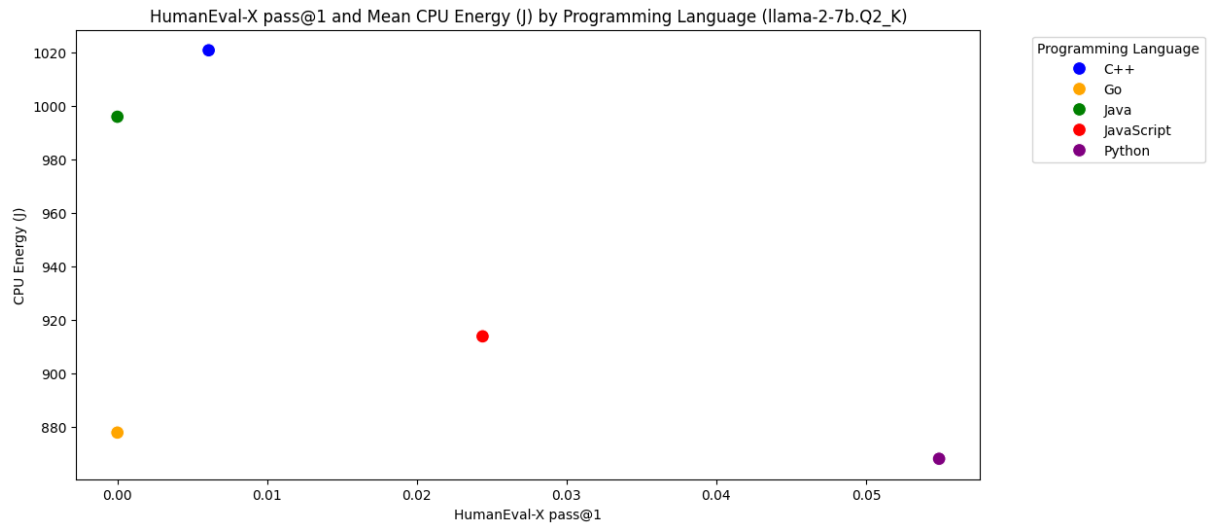


Figure 4: CPU Energy and HumanEval-X pass@1 of llama-2-7b.Q2_K

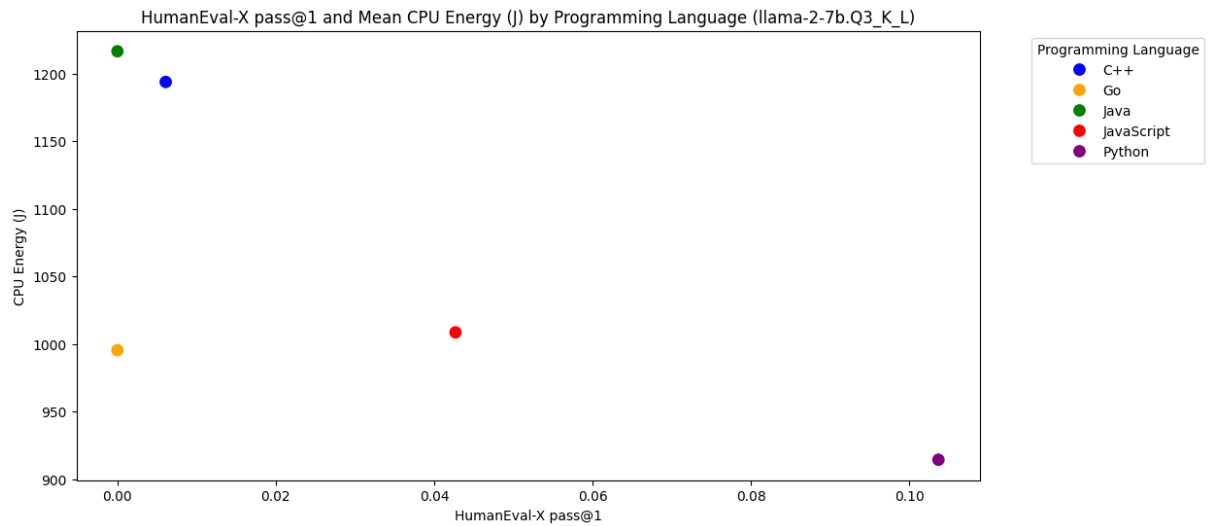


Figure 5: CPU Energy and HumanEval-X pass@1 of llama-2-7b.Q3_K_L

5.3 Discussion

Analyzing scatterplots from Figures 2, 3, 4, and 5, we can observe that Python achieves the highest score in the HumanEval-X benchmark, consuming less energy and taking less time to generate code in this language. Both LLMs under consideration generate code with a null HumanEval-X score in Go and Java. llama-2-7b.Q2_K and llama-2-7b.Q3_K_L generate Java code that takes longer to generate and produce C++ code with the highest energy consumption.

Chapter 6

Conclusions and future work

In this phase of the study, we obtained some preliminary results. They are not reliable, as they likely involve consuming more energy than necessary for the tasks. However, the next steps in the study will involve validating these results in a more robust manner. Additional benchmarks, such as CyberSecEval and MBPP (Mostly Basic Python Programming), should be included in the study. Moreover, more LLM families need to be studied for local execution using Python, necessitating the inclusion of additional computational resources. To accommodate the storage of all LLMs on my computer, a new server with larger space storage will be required to streamline the removal of unnecessary processes that consume extra energy.

The significance of open science highlights our dedication to offering comprehensive resources for replicating our study, accompanied by clear instructions on their utilization. Therefore, the subsequent step involves creating a Docker image containing all the necessary libraries to execute the LLMs and benchmarks utilized in the study. This initiative will empower others to benchmark their LLMs regarding energy consumption, benchmark scores, execution time, and CO_2 emissions.

Chapter 7

Planned Schedule

7.1 Activities

Task	Sep	Oct	Nov	Dec	Jan	Feb	Mar	Apr	May	Jun	Jul
Background and SOA	•	•	•	•	•						
Resource Investigation			•	•	•						
Implementation			•	•	•	•	•	•	•		
PDR Writing			•	•	•						
Statistical Analysis							•	•	•	•	•
Dissertation Writing			•	•	•	•	•	•	•	•	•

Table 10: Activities Plan

Bibliography

- Sarah Abdulsalam, Ziliang Zong, Qijun Gu, and Meikang Qiu. Using the greenup, powerup, and speedup metrics to evaluate software energy efficiency. In *Proc. of the 6th Int. Green and Sustainable Computing Conf.*, pages 1–8. IEEE, 2015.
- Shubham Agarwal. Cracking the code llms: How code llms progressed from rnns to transformers. *Towards Data Science*, Nov 2023. URL <https://towardsdatascience.com/cracking-the-code-llms-354505c53295>. Published in Towards Data Science. Accessed: January 9, 2024.
- Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation, 2021.
- Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, January 2019. ISSN 2475-1421. doi: 10.1145/3290353. URL <http://doi.acm.org/10.1145/3290353>.
- Amazon Codewhisperer. Amazon codewhisperer website, 2022. URL https://aws.amazon.com/pt/codewhisperer/resources/#Getting_started/. Accessed: January 3, 2024.
- Luca Ardito, Giuseppe Procaccianti, Marco Torchiano, and Antonio Vetrò. Understanding green software development: A conceptual framework. *IT Professional*, 17(1):44–50, 2015. doi: 10.1109/MITP.2015.16.
- Bard. Bard google model website, 2023. URL <https://bard.google.com/chat>. Accessed: January 3, 2024.
- Ayse Basar Bener, Maurizio Morisio, and Andriy Miranskyy. Green software, 2014. ISSN 0740-7459. Published by the IEEE Computer Society.
- Bloom. Bloom bigscience model huggingface website, 2022. URL <https://huggingface.co/bigscience/bloom>. Accessed: January 3, 2024.

ChatGPT. Chatgpt website, 2022. URL <https://chat.openai.com/>. Accessed: January 3, 2024.

ChatPDF. Chatpdf website, 2022. URL <https://www.chatpdf.com/>. Accessed: January 3, 2024.

Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. Unleashing the potential of prompt engineering in large language models: a comprehensive review, 2023.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

Eunsol Choi, He He, Mohit Iyyer, Mark Yatskar, Wen tau Yih, Yejin Choi, Percy Liang, and Luke Zettlemoyer. Quac : Question answering in context, 2018.

Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. Boolq: Exploring the surprising difficulty of natural yes/no questions, 2019.

Claude. Claude anthropic model website, 2023. URL <https://claude.ai>. Accessed: January 3, 2024.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021.

CodeCarbon. Codecarbon framework website, 2021. URL <https://codecarbon.io/>. Accessed: January 2, 2024.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale, 2022.

- Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.
- Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers, 2023.
- G. Gerganov. llama.cpp: Low-Latency Audio Streaming Library for C++. <https://github.com/ggerganov/llama.cpp>, 2023. Accessed: January 16, 2024.
- Github CoPilot. Github copilot website, 2022. URL <https://github.com/features/copilot>. Accessed: January 3, 2024.
- Google Bard. Google bard website, 2022. URL <https://bard.google.com/chat>. Accessed: January 3, 2024.
- GPT-4. Gpt-4 openai model website, 2023. URL <https://openai.com/gpt-4>. Accessed: January 3, 2024.
- Green Software Foundation. What is green software?, 2021. URL <https://greensoftware.foundation/articles/what-is-green-software>. Green Software Foundation Newsletter.
- Muhammad Usman Hadi, Qasem Al Tashi, Rizwan Qureshi, et al. A survey on large language models: Applications, challenges, limitations, and practical usage. *TechRxiv*, July 2023. doi: 10.36227/techrxiv.23589741.v1.
- Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding, 2021.
- Intel RAPL. Intel rapl overview website, 2022. URL <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html>. Accessed: January 2, 2024.
- Mandar Joshi, Eunsol Choi, Daniel S. Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension, 2017.
- Jean Kaddour, Joshua Harris, Maximilian Mozes, Herbie Bradley, Roberta Raileanu, and Robert McHardy. Challenges and applications of large language models, 2023.

- Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Matthew Kelcey, Jacob Devlin, Kenton Lee, Kristina N. Toutanova, Llion Jones, Ming-Wei Chang, Andrew Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. Natural questions: a benchmark for question answering research. *Transactions of the Association of Computational Linguistics*, 2019.
- Llama 2. Llama 2 meta models website, 2023. URL <https://ai.meta.com/llama/>. Accessed: January 2, 2024.
- Dustin McIntire, Thanos Stathopoulos, Sasank Reddy, Thomas Schmidt, and William J. Kaiser. Energy-efficient sensing with the low power, energy aware processing (leap) architecture. *ACM Trans. Embed. Comput. Syst.*, 11(2):27:1–27:36, July 2012.
- Microsoft Bing Browser. Microsoft bing browser website, 2022. URL <https://www.bing.com/>. Accessed: January 3, 2024.
- Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? a new dataset for open book question answering, 2018.
- Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A comprehensive overview of large language models, 2023.
- Tommaso Pegolotti, Elias Frantar, Dan Alistarh, and Markus Püschel. Qigen: Generating efficient kernels for quantized inference on large language models, 2023.
- Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, 2021. ISSN 0167-6423. doi: <https://doi.org/10.1016/j.scico.2021.102609>. URL <https://www.sciencedirect.com/science/article/pii/S0167642321000022>.
- Gustavo Pinto and Fernando Castor and Yu David Liu. Mining questions about software energy consumption, 2014.
- S. Podder, A. Burden, S. Kumar Singh, and R. Maruca. How green is your software? *Harvard Business Review*, September 2020. URL <https://hbr.org/2020/09/how-green-is-your-software>. Accessed: January 9, 2024.

Python binding for LLaMa.cpp. Llama 2 meta models website, 2023. URL <https://github.com/abetlen/llama-cpp-python>. Accessed: January 16, 2024.

Francisco Ribeiro, Rui Abreu, and João Saraiva. Framing program repair as code completion. In *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 38–45, 2022. doi: 10.1145/3524459.3527347.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2023.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale, 2019.

Tabnine. Tabnine website, 2022. URL <https://www.tabnine.com/>. Accessed: January 3, 2024.

Adrian Tam. What are zero-shot prompting and few-shot prompting, July 20 2023. URL <https://machinelearningmastery.com/what-are-zero-shot-prompting-and-few-shot-prompting/>. Accessed: January 10, 2024.

Artur Tarassow. The potential of llms for coding with low-resource and domain-specific programming languages, 2023.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan

- Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.
- Roberto Verdecchia, Patricia Lago, Christof Ebert, and Carol de Vries. Green it and green software. *IEEE Software*, 38(6):7–15, 2021. doi: 10.1109/MS.2021.3102254.
- Roberto Verdecchia, June Sallou, and Luís Cruz. A systematic review of green ai, 2023.
- Vicuna. Vicuna lmsys model website, 2023. URL <https://lmsys.org/blog/2023-03-30-vicuna/>. Accessed: January 3, 2024.
- Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021.
- Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.
- Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence?, 2019.
- Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x, 2023.
- Wanjun Zhong, Ruixiang Cui, Yiduo Guo, Yaobo Liang, Shuai Lu, Yanlin Wang, Amin Saied, Weizhu Chen, and Nan Duan. Agieval: A human-centric benchmark for evaluating foundation models, 2023.

Appendix A

Listings

```

1 {
2     "task_id": "HumanEval/0",
3     "prompt": "from typing import List\n\nndef has_close_elements(numbers: List[float],\nthreshold: float) -> bool:\n    \"\"\" Check if in given list of numbers, are any two\nnumbers closer to each other than\n    given threshold.\n    >>> has_close_elements([1.0,\n2.0, 3.0], 0.5)\n    False\n    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3\n)\n    True\n    \"\"\"\n",
4     "entry_point": "has_close_elements",
5     "canonical_solution": "    for idx, elem in enumerate(numbers):\n        for idx2, elem2\nin enumerate(numbers):\n            if idx != idx2:\n                distance = abs(elem -\n                elem2)\n            if distance < threshold:\n                return True\n    return False\n",
6     "test": "\n\nMETADATA = {\n    'author': 'jt',\n    'dataset': 'test'\n}\n\nndef check(\ncandidate):\n    assert candidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3) == True\n    assert\ncandidate([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.05) == False\n    assert candidate([1.0, 2.0,\n5.9, 4.0, 5.0], 0.95) == True\n    assert candidate([1.0, 2.0, 5.9, 4.0, 5.0], 0.8) ==\nFalse\n    assert candidate([1.0, 2.0, 3.0, 4.0, 5.0, 2.0], 0.1) == True\n    assert\ncandidate([1.1, 2.2, 3.1, 4.1, 5.1], 1.0) == True\n    assert candidate([1.1, 2.2, 3.1, 4.\n1, 5.1], 0.5) == False\n\n"
7 }

```

Listing A.1: HumanEval prompt file entry example

```

1 {
2     "task_id": "CPP/0",
3     "prompt": "/*\nCheck if in given vector of numbers, are any two numbers closer to each\nother than\ngiven threshold.\n>>> has_close_elements({1.0, 2.0, 3.0}, 0.5)\nfalse\n>>>\nhas_close_elements({1.0, 2.8, 3.0, 4.0, 5.0, 2.0}, 0.3)\ntrue\n*/\n#include<stdio.h>\n#include<vector>\n#include<math.h>\nusing namespace std;\nbool has_close_elements(vector<\nfloat> numbers, float threshold){\n",
4     "canonical_solution": "    int i,j;\n    for (i=0;i<numbers.size();i++)\n        for\n(j=i+1;j<numbers.size();j++)\n            if (abs(numbers[i]-numbers[j])<threshold)\n                return true;\n    return false;\n}\n",
5     "test": "#undef NDEBUG\n#include<assert.h>\nint main(){\n    vector<float> a={1.0, 2.0, 3.\n9, 4.0, 5.0, 2.2};\n    assert (has_close_elements(a, 0.3)==true);\n    assert (\nhas_close_elements(a, 0.05) == false);\n    assert (has_close_elements({1.0, 2.0, 5.9, 4\n.0, 5.0}, 0.95) == true);\n    assert (has_close_elements({1.0, 2.0, 5.9, 4.0, 5.0}, 0.8)\n==false);\n    assert (has_close_elements({1.0, 2.0, 3.0, 4.0, 5.0}, 2.0) == true);\n    assert (has_close_elements({1.1, 2.2, 3.1, 4.1, 5.1}, 1.0) == true);\n    assert (\nhas_close_elements({1.1, 2.2, 3.1, 4.1, 5.1}, 0.5) == false);\n}\n",
6     "declaration": "#include<stdio.h>\n#include<vector>\n#include<math.h>\nusing namespace std\n;\n#include<algorithm>\n#include<stdlib.h>\nbool has_close_elements(vector<float> numbers,\nfloat threshold){\n",
7     "example_test": "#undef NDEBUG\n#include<assert.h>\nint main(){\n    assert (\nhas_close_elements({1.0, 2.0, 3.0}, 0.5) == false && \"failure 1\");\n    assert (\nhas_close_elements({1.0, 2.8, 3.0, 4.0, 5.0, 2.0}, 0.3) && \"failure 2\");\n}\n"
8 }

```

Listing A.2: HumanEval-X prompt file entry example for C++

```

1 {
2     "task_id": "Go/0",
3     "prompt": "import (\n    \"math\"\n)\n\n// Check if in given list of numbers, are any two\nnumbers closer to each other than given threshold.\n// >>> HasCloseElements([]float64{1.0,\n2.0, 3.0}, 0.5)\n// false\n// >>> HasCloseElements([]float64{1.0, 2.8, 3.0, 4.0, 5.0, 2.0\n}, 0.3)\n// true\nfunc HasCloseElements(numbers []float64, threshold float64) bool {\n",

```

```

4  "import": "import (\n    \"math\"\n)\n",
5  "docstring": "/* Check if in given list of numbers, are any two numbers closer to each other than given threshold.\n */\n\n >>> HasCloseElements([float64{1.0, 2.0, 3.0}], 0.5)\n\n false\n\n >>> HasCloseElements([float64{1.0, 2.8, 3.0, 4.0, 5.0, 2.0}], 0.3)\n\n true\n",
6  "declaration": "\nfunc HasCloseElements(numbers []float64, threshold float64) bool {\n",
7  "canonical_solution": "    for i := 0; i < len(numbers); i++ {\n        for j := i + 1; j < len(numbers); j++ {\n            var distance float64 = math.Abs(numbers[i] - numbers[j])\n            if distance < threshold {\n                return true\n            }\n        }\n    }\n    return false\n}\n",
8  "test": "func TestHasCloseElements(t *testing.T) {\n    assert := assert.New(t)\n    assert.Equal(true, HasCloseElements([float64{11.0, 2.0, 3.9, 4.0, 5.0, 2.2}], 0.3))\n    assert.Equal(false, HasCloseElements([float64{1.0, 2.0, 3.9, 4.0, 5.0, 2.2}], 0.05))\n    assert.Equal(true, HasCloseElements([float64{1.0, 2.0, 5.9, 4.0, 5.0}], 0.95))\n    assert.Equal(false, HasCloseElements([float64{1.0, 2.0, 5.9, 4.0, 5.0}], 0.8))\n    assert.Equal(true, HasCloseElements([float64{1.0, 2.0, 3.0, 4.0, 5.0, 2.0}], 0.1))\n    assert.Equal(true, HasCloseElements([float64{1.1, 2.2, 3.1, 4.1, 5.1}], 1.0))\n    assert.Equal(false, HasCloseElements([float64{1.1, 2.2, 3.1, 4.1, 5.1}], 0.5))\n}\n",
9  "test_setup": "package main\n\nimport (\n    \"testing\"\n    \"github.com/stretchr/testify/assert\"\n)\n",
10 "example_test": "func TestHasCloseElements(t *testing.T) {\n    assert := assert.New(t)\n    assert.Equal(false, HasCloseElements([float64{1.0, 2.0, 3.0}], 0.5))\n    assert.Equal(true, HasCloseElements([float64{1.0, 2.8, 3.0, 4.0, 5.0, 2.0}], 0.3))\n}\n"
11 }

```

Listing A.3: HumanEval-X prompt file entry example for Go

```

1  {
2  "task_id": "Java/0",
3  "prompt": "import java.util.*;\nimport java.lang.*;\n\npublic class Solution {\n    /**\n    Check if in given list of numbers, are any two numbers closer to each other than given threshold.\n    >>> hasCloseElements(Arrays.asList(1.0, 2.0, 3.0), 0.5)\n    false\n    >>> hasCloseElements(Arrays.asList(1.0, 2.8, 3.0, 4.0, 5.0, 2.0), 0.3)\n    true\n    */\n    public boolean hasCloseElements(List<Double> numbers, double threshold) {\n",
4  "canonical_solution": "        for (int i = 0; i < numbers.size(); i++) {\n            for (int j = i + 1; j < numbers.size(); j++) {\n                double distance = Math.abs(numbers.get(i) - numbers.get(j));\n                if (distance < threshold) return true;\n            }\n        }\n        return false;\n    }\n}\n",
5  "test": "public class Main {\n    public static void main(String[] args) {\n        Solution s = new Solution();\n        List<Boolean> correct = Arrays.asList(\n            s.hasCloseElements(new ArrayList<>(Arrays.asList(11.0, 2.0, 3.9, 4.0, 5.0, 2.2)), 0.3),\n            !s.hasCloseElements(new ArrayList<>(Arrays.asList(1.0, 2.0, 3.9, 4.0, 5.0, 2.2)), 0.05),\n            s.hasCloseElements(new ArrayList<>(Arrays.asList(1.0, 2.0, 5.9, 4.0, 5.0)), 0.95),\n            !s.hasCloseElements(new ArrayList<>(Arrays.asList(1.0, 2.0, 5.9, 4.0, 5.0)), 0.8),\n            s.hasCloseElements(new ArrayList<>(Arrays.asList(1.0, 2.0, 3.0, 4.0, 5.0, 2.0)), 0.1),\n            s.hasCloseElements(new ArrayList<>(Arrays.asList(1.1, 2.2, 3.1, 4.1, 5.1)), 1.0),\n            !s.hasCloseElements(new ArrayList<>(Arrays.asList(1.1, 2.2, 3.1, 4.1, 5.1)), 0.5)\n        );\n        if (correct.contains(false)) {\n            throw new AssertionError();\n        }\n    }\n}\n",
6  "text": "/* Check if in given list of numbers, are any two numbers closer to each other than given threshold.\n */\n\n >>> hasCloseElements(Arrays.asList(1.0, 2.0, 3.0), 0.5)\n\n false\n\n >>> hasCloseElements(Arrays.asList(1.0, 2.8, 3.0, 4.0, 5.0, 2.0), 0.3)\n\n true\n",
7  "declaration": "import java.util.*;\nimport java.lang.*;\n\npublic class Solution {\n    public boolean hasCloseElements(List<Double> numbers, double threshold) {\n",
8  "example_test": "public class Main {\n    public static void main(String[] args) {\n        Solution s = new Solution();\n        List<Boolean> correct = Arrays.asList(\n            !s.hasCloseElements(new ArrayList<>(Arrays.asList(1.0, 2.0, 3.0)), 0.5),\n            s.hasCloseElements(new ArrayList<>(Arrays.asList(1.0, 2.8, 3.0, 4.0, 5.0, 2.0)), 0.3)\n        );\n        if (correct.contains(false)) {\n            throw new\n        }\n    }\n}\n",
9  }

```

Listing A.4: HumanEval-X prompt file entry example for Java

```

1  {
2  "task_id": "JavaScript/0",
3  "prompt": "/* Check if in given list of numbers, are any two numbers closer to each other than\n given threshold.\n */\n\n >>> hasCloseElements([1.0, 2.0, 3.0], 0.5)\n\n false\n\n >>> hasCloseElements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)\n\n true\n\n */\n\nconst hasCloseElements = (numbers, threshold) => {\n",

```

```

4  "canonical_solution": "  for (let i = 0; i < numbers.length; i++) {\n    for (let j = 0; j
    < numbers.length; j++) {\n      if (i !== j) {\n        let distance = Math.abs(numbers[i]
    - numbers[j]);\n        if (distance < threshold) {\n          return true;\n        }\n      }\n    }\n  }\n  return false;\n}\n\n",
5  "test": "const testHasCloseElements = () => {\n  console.assert(hasCloseElements([1.0, 2.0
, 3.9, 4.0, 5.0, 2.2], 0.3) === true)\n  console.assert(\n    hasCloseElements([1.0, 2.0,
3.9, 4.0, 5.0, 2.2], 0.05) === false\n  )\n  console.assert(hasCloseElements([1.0, 2.0, 5.
9, 4.0, 5.0], 0.95) === true)\n  console.assert(hasCloseElements([1.0, 2.0, 5.9, 4.0, 5.0]
, 0.8) === false)\n  console.assert(hasCloseElements([1.0, 2.0, 3.0, 4.0, 5.0, 2.0], 0.1)
=== true)\n  console.assert(hasCloseElements([1.1, 2.2, 3.1, 4.1, 5.1], 1.0) === true)\n
  console.assert(hasCloseElements([1.1, 2.2, 3.1, 4.1, 5.1], 0.5) === false)\n}\n\n\ntestHasCloseElements()\n",
6  "declaration": "\nconst hasCloseElements = (numbers, threshold) => {\n",
7  "example_test": "const testHasCloseElements = () => {\n  console.assert(hasCloseElements([
1.0, 2.0, 3.0], 0.5) === false)\n  console.assert(\n    hasCloseElements([1.0, 2.8, 3.0, 4
.0, 5.0, 2.0], 0.3) === true\n  )\n}\n\ntestHasCloseElements()\n"
8 }

```

Listing A.5: HumanEval-X prompt file entry example for JavaScript

```

1  {
2    "task_id": "Python/0",
3    "prompt": "from typing import List\n\ndef has_close_elements(numbers: List[float],
threshold: float) -> bool:\n    \"\"\" Check if in given list of numbers, are any two
numbers closer to each other than\n    given threshold.\n    >>> has_close_elements([1.0,
2.0, 3.0], 0.5)\n    False\n    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3
)\n    True\n    \"\"\"\n",
4    "canonical_solution": "  for idx, elem in enumerate(numbers):\n      for idx2, elem2
in enumerate(numbers):\n          if idx != idx2:\n              distance = abs(elem -
elem2)\n              if distance < threshold:\n                  return True\n\n  return False\n",
5    "test": "\n\nMETADATA = {\n  'author': 'jt',\n  'dataset': 'test'\n}\n\ndef check(
has_close_elements):\n    assert has_close_elements([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.3)
== True\n    assert has_close_elements([1.0, 2.0, 3.9, 4.0, 5.0, 2.2], 0.05) == False\n
    assert has_close_elements([1.0, 2.0, 5.9, 4.0, 5.0], 0.95) == True\n    assert
has_close_elements([1.0, 2.0, 5.9, 4.0, 5.0], 0.8) == False\n    assert has_close_elements
([1.0, 2.0, 3.0, 4.0, 5.0, 2.0], 0.1) == True\n    assert has_close_elements([1.1, 2.2, 3.
1, 4.1, 5.1], 1.0) == True\n    assert has_close_elements([1.1, 2.2, 3.1, 4.1, 5.1], 0.5)
== False\n\ncheck(has_close_elements)",
6    "text": "  Check if in given list of numbers, are any two numbers closer to each other
than\n  given threshold.\n  >>> has_close_elements([1.0, 2.0, 3.0], 0.5)\n  False\n
  >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)\n  True",
7    "declaration": "from typing import List\n\ndef has_close_elements(numbers: List[float],
threshold: float) -> bool:\n",
8    "example_test": "def check(has_close_elements):\n    assert has_close_elements([1.0, 2.0,
3.0], 0.5) == False\n    assert has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3) ==
True\n\ncheck(has_close_elements)\n"
9 }

```

Listing A.6: HumanEval-X prompt file entry example for Python

