Abstract

The aim of this project is to use popular open-weight LLMs to analyze a corpus of Alloy code and identify common patterns and analyse Alloy code generated by such LLMs. A corpus of Alloy code is scraped from GitHub and uses the models in prompts to LLMs to get information about the structure of the models.

Introduction

This project aims to use LLMs to identify prevalent patterns in Alloy code. Alloy is a declarative specification language based on first-order logic, which is used for model-checking. It builds on previous work[cite] which used classical parsing techniques to determine the frequencies of code patterns in Alloy code found on GitHub.

Problem statement

Research Questions:

- Can fine-tuned LLMs consistently produce valid Alloy code? (i.e. code that runs on the Alloy Ananlyzer)
- What are the characteristics of the code produced by LLMs? Are certain patterns and structures favoured/disfavoured when compared to human-written Alloy code?
- Can AI recognize the presence of patterns in given code?
- Does the AI recognize patterns which were hitherto unidentified by classical techniques?

Scope reduction:

Since computational resources were limited, the project had to be downsized to include only the third research question.

Ground Truth

The facts about the models being analysed by the LLMs are determined via conventional parsing techniques, as defined in [paper]. The same tools used to extract information in [paper] are re-used on a different corpus, scraped from GitHub. This is then compared to the results generated by parsing the outputs of the LLMs.

Translation to prompts

When examining a certain aspect of an Alloy model, the aspect is defined in a manner compatible with classical parsing technique in [paper]. Since the goal of the project is to study LLMs in particular, the inputs given to LLMs are given in plain english. The purpose behind each regular expression and SDT statement in the parser used in [paper] is outlined in the github repository associated with it. For this project, these statements were used to construct equivalent prompts in plain English.

For instance, the task of counting the number of signatures in a model is translated into:

```
prompt = "This is sample Alloy code. How many signatures exist? Answer in a
single sentence, use symbols for numbers, do not use "," to separate the
numbers\n"+contents
```

The scripts used to invoke the LLM were developed with reference to https://arxiv.org/pdf/2307.13106 (Step 3, pages 3-4).

A significant challenge was to ensure that the outputs of the prompts were meaningful. Since the problem of counting occurances of sigs is relatively simple, validating the output is as simple as checking if numbers exist explicitly in it. To avoid conflicts with the use of words linked to numbers (like "twice", used in other parts of an answer with an actual number as the main response), the prompt required the use of symbols for the answer. Any answer completely lacking in such symbols were treated as illegible.

LLMs

fine-tuning

Since the LLMs being used in the experiment were already trained on examples from Github (which presumably included Alloy code in particular and declarative programming in general), and since fine-tuning is computationally much more expensive than data gathering via prompts, it was determined that gathering more data from the base implementations of the LLMs is a more economical use of the already limited compute available.

parameter count

Due to limitations in RAM, the models chosen were limted to small parameter counts. Furthermore, quantization of the parameters was required to compress the models further into a size that would fit in the RAM of the computer.

Data collection

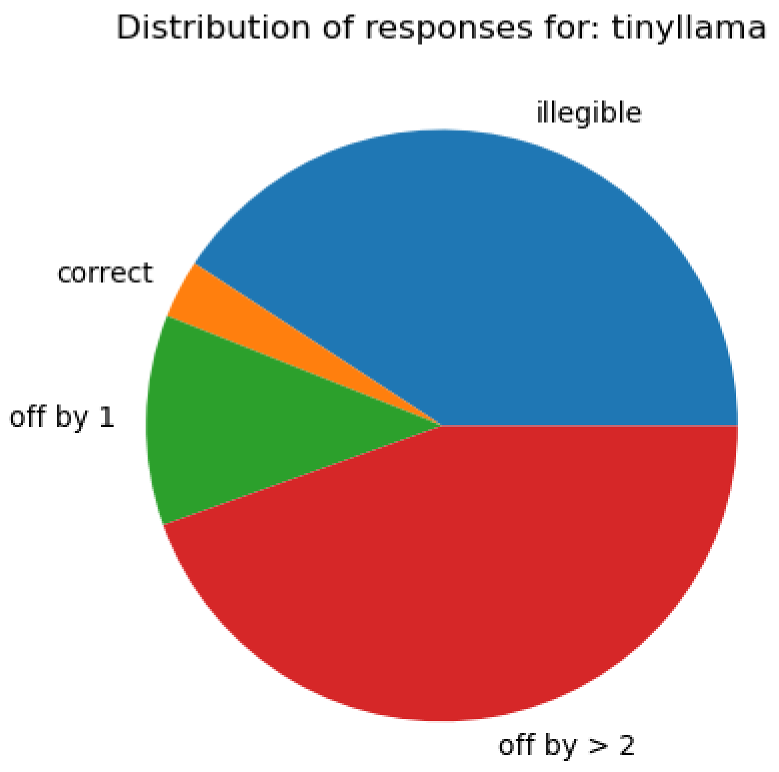
Since the LLMs need to be loaded into the RAM/VRAM to run, the process controlling the LLMs needs to be resistant to crashes which may cause a loss of gathrered data. The data gathered is in the form of text, which needs to be moved out from temporary to permanent memory on a continuous basis. However, since accessing memory is expensive, this process needs to be performed only when strictly necessary.

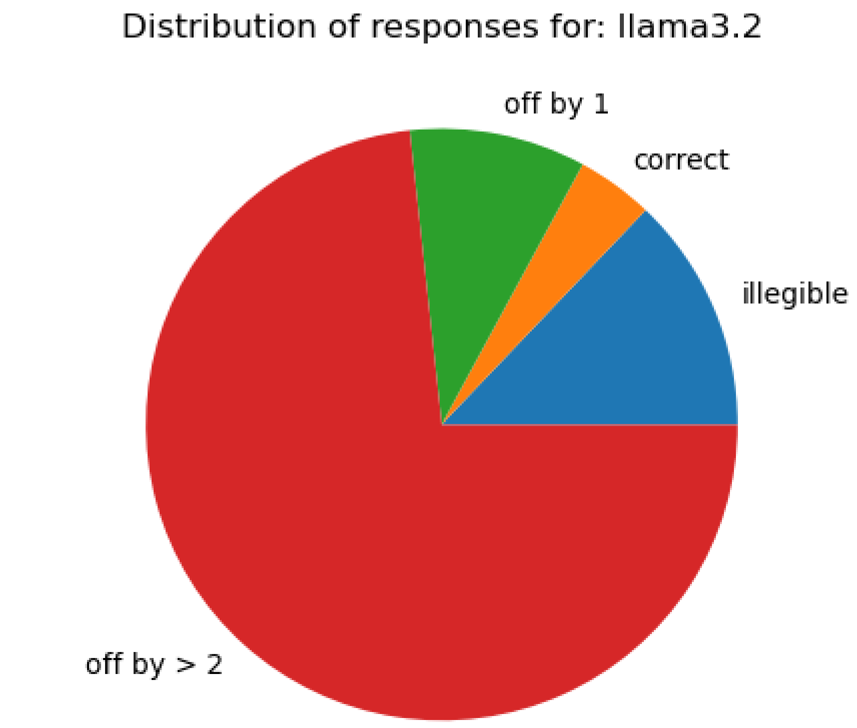
If the process does not write LLM outputs until the very end, it is vulnerable to complete data loss in the event of a crash. Furthermore, as the outputs require RAM to store, the compute power keeps decreasing as more and more responses are gathered. Eventually, when there is more space required than is avaliable in the RAM, the process crashes.

To prevent this, the process is designed to periodically dump the gathered data into memory, triggered between two successive prompts to an LLM.

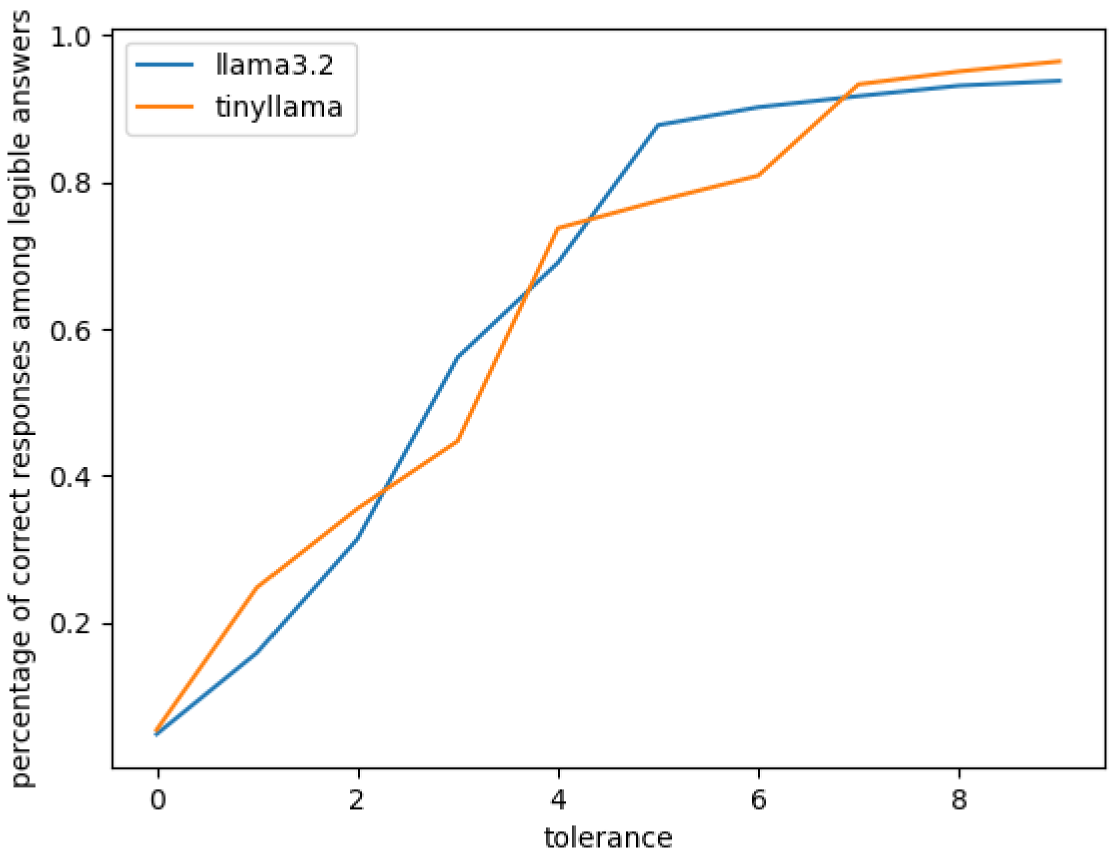
The data is dumped into a `.stream` file, which are then merged during the data analysis phase, after all the data is gathered.

Evaluation of results:

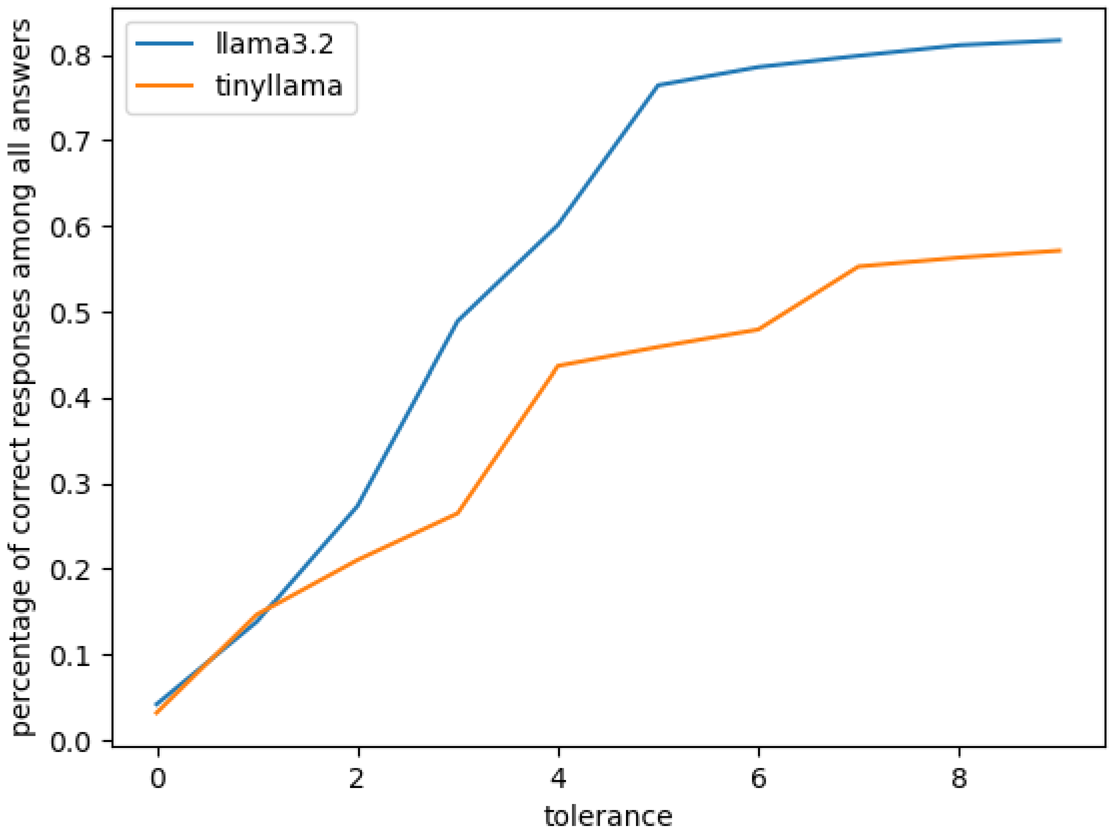

Distribution of responses for: tinyllama
illegible
correct
off by 1
off by > 2


Distribution of responses for: llama3.2
off by 1
correct
illegible
off by > 2

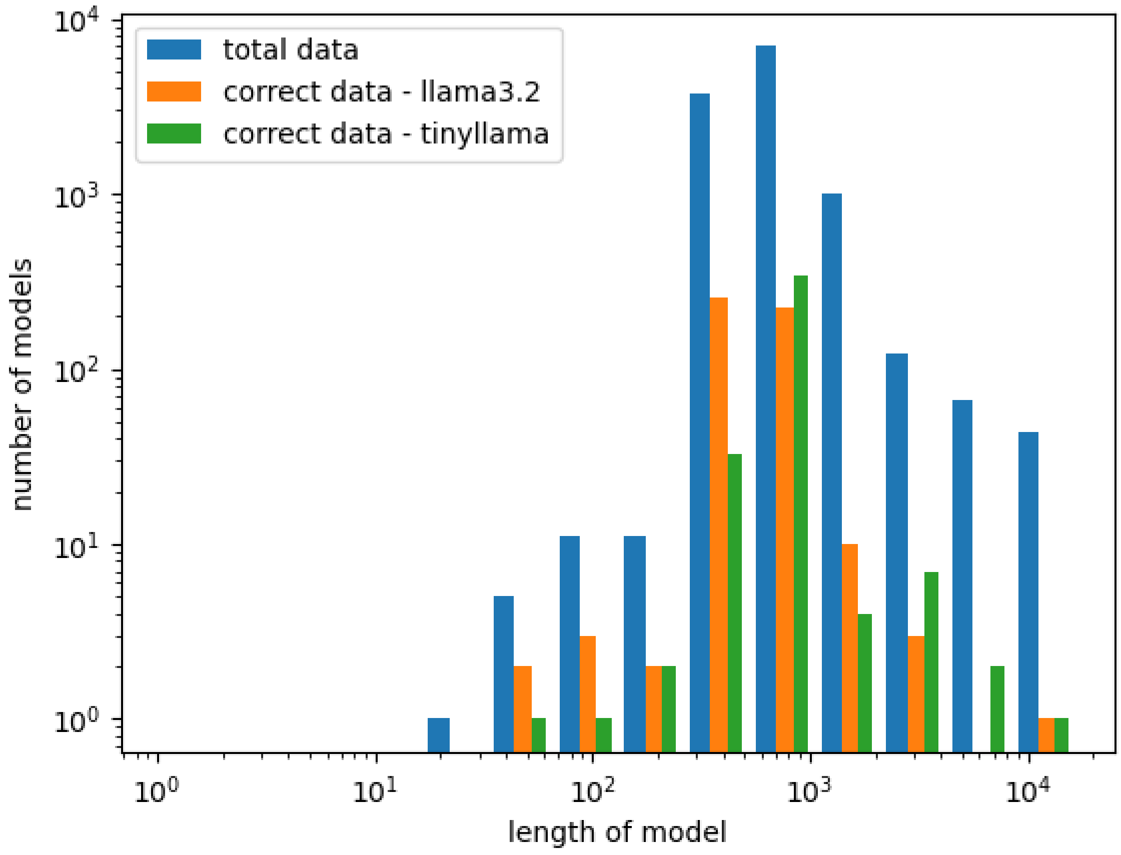
When given a relatively simple problem of detecting the presence of signatures and counting them, the accuracy of the LLMs were low. Llama3, which had a larger parameter count, gave more legible answers (i.e. answers which had number in them). However, the percentage of correct answers (compared to the total number of queries) was not significantly different between the two LLMs, with Llama3.2 being slightly more accurate than tinyllama.



When the restrictions for accuracy are relaxed to measure the percentages of responses which miscount the number of signatures by x, where x lies between 0 and 10, one observes that there is no significant difference between the two models when restricting the domain to the set of responses which are legible.

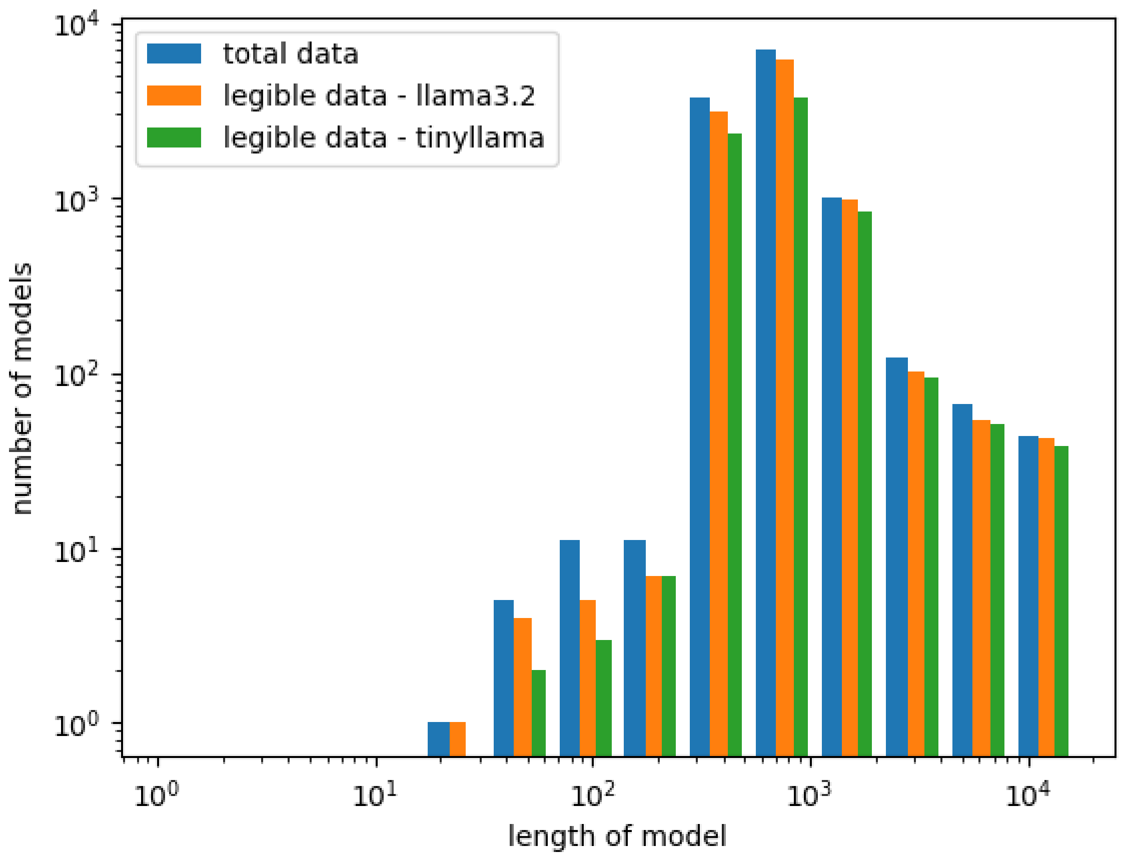


However, when considering all the responses, the larger llama3.2 outperforms the smaller tinyllama LLM.

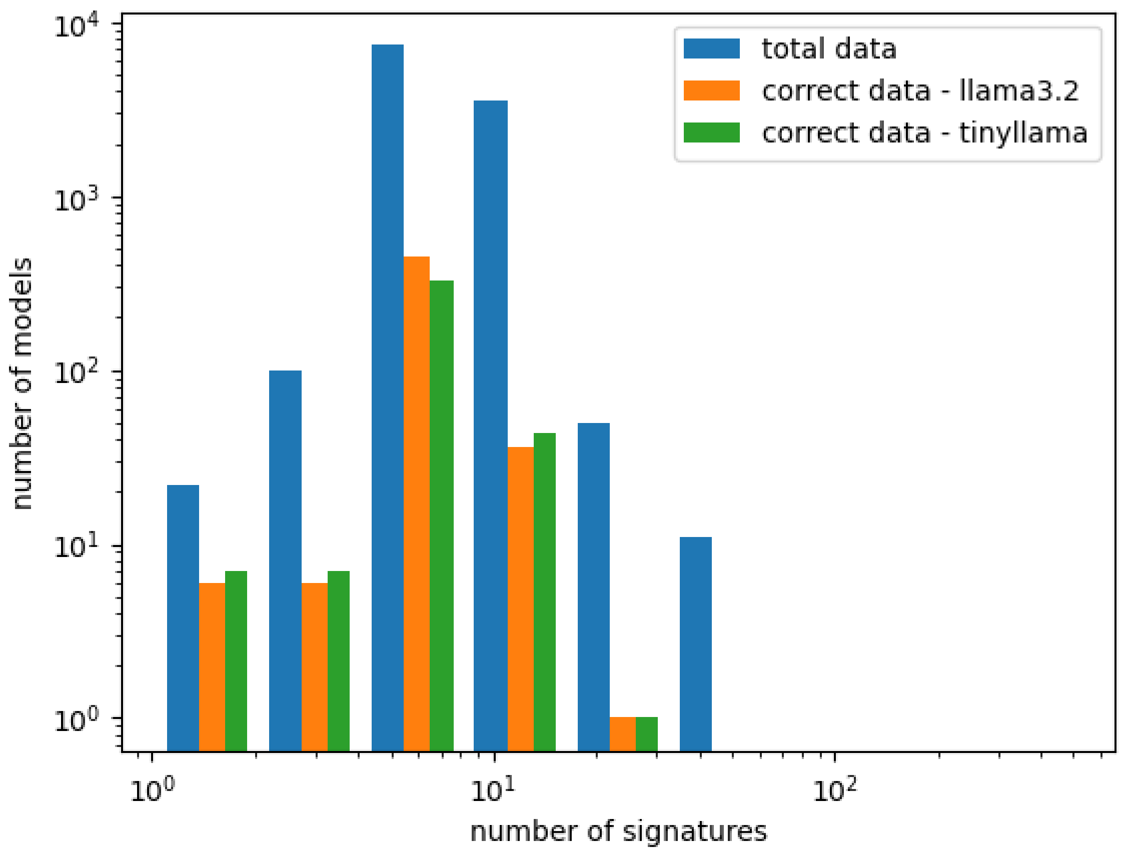


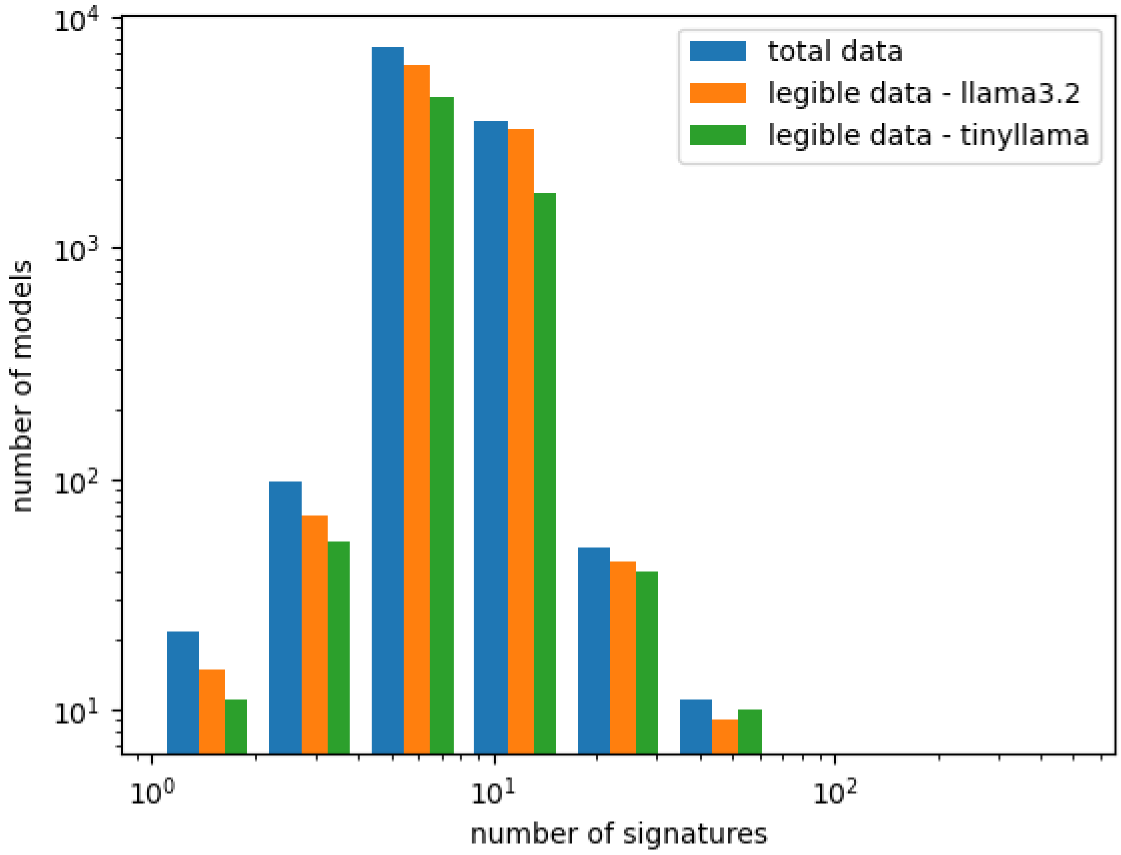
When breaking down the data by model size, a correlation between model size and accuracy arises - Both Llama3.2 and tinyllama do significantly worse on larger models, particularly when the number of characters crosses 1024. For models of high length, the accuracy drops off to negligible levels.

☑ consists of a logarithmic scale for clarity. The relative sizes of the bars do not directly represent the percentage of correct responses.



When similar analysis is done for legibility, we observe no correlation with size. However, it is clear that Llama3.2 consistently produces more legible responses than tinyllama at every size of model.


total data
correct data - llama3.2
correct data - tinyllama
number of models
number of signatures
10^0
10^1
10^2
10^3
10^4

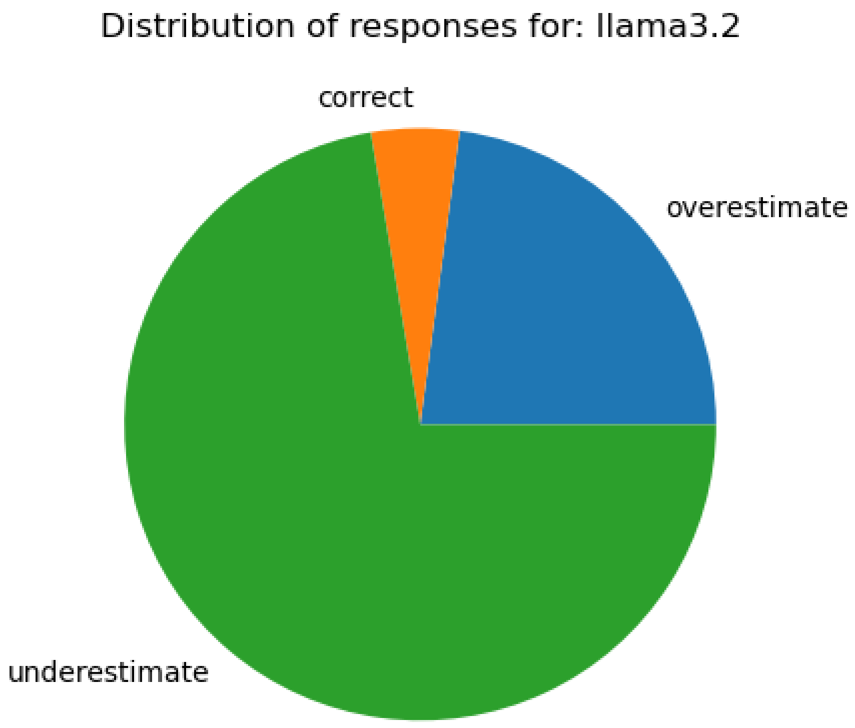

total data
legible data - llama3.2
legible data - tinyllama
number of models
number of signatures
10^0
10^1
10^2
10^3
10^4

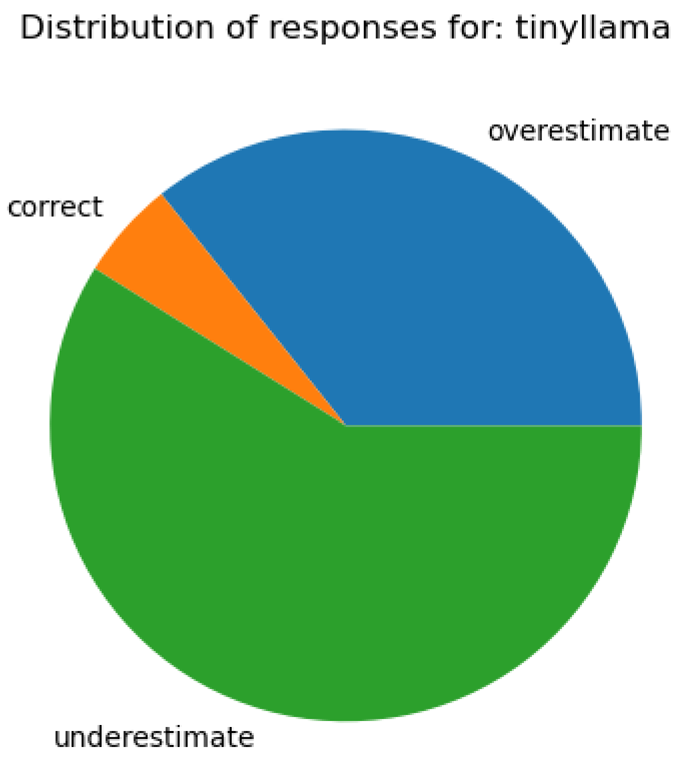
When similar analysis is done by grouping models based on the number of signatures that the LLMs need to count, we see a sharp decline in correctness when the number crosses 10. This suggests that LLMs cannot reliably count large numbers of occurances of keywords in a model. This result is in line with the results in https://arxiv.org/html/2412.18626v1

Threats to Validity:

Due to constraints on compute, the project used only low-parameter models for analysis.The results could have been significantly different using different models like mixtral (with a mixture of experts architecture) or DeepSeek-r1 (with reasoning capabilities on par with or exceeding OPenAI o1). In particular, DeepSeek-r1 seems particularly promising at reasoning through a question about a model's structure, while dealing with large contexts.

Each prompt was issued exactly once, due to restrictions on compute. Since the output is not necessarily deterministic for LLMs, a more accurate picture can be derived from repeating each prompt multiple times.





When the nature of the errors are analysed further, most of the quantities given by the LLMs are underestimates of the true amount. Overestimates are relatively rarer than underestimates, with tinyllama having a fairer mixture of errors than Llama3.2

Conclusions:

1. The accuracy is very low, even for simple tasks like counting sigs.
2. Counting tasks seem more likely to result in underestimation than overestimation.
3. When accounting for margin of error, larger LLMs do slightly better than smaller ones.
4. LLMs seem to perform much worse with longer Alloy models.