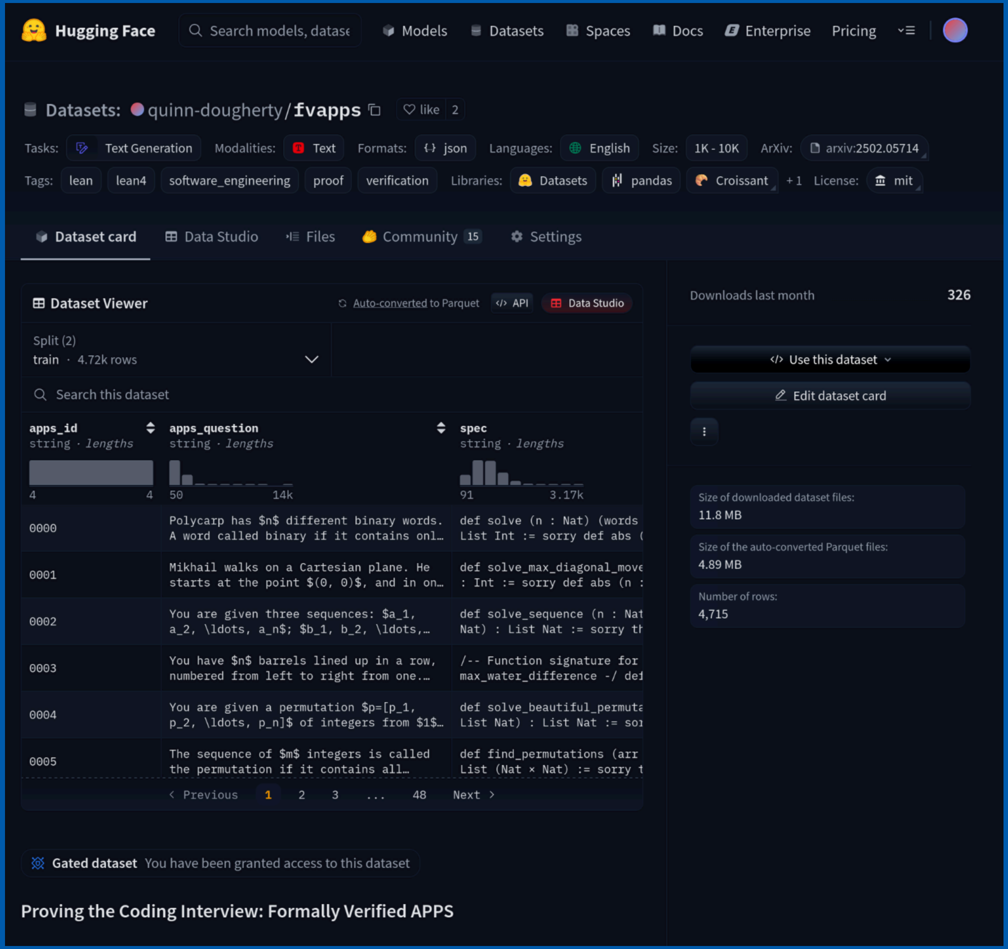
Proving the Coding Interview: A Benchmark for Formally Verified Code Generation

Quinn Dougherty[1], Ronak Mehta[2]

[1]Beneficial AI Foundation

[2]Coordinal Research

May 03, 2025


Hugging Face
Search models, datase
Models
Datasets
Spaces
Docs
Enterprise
Pricing
Datasets: quinn-dougherty/fvapps
like 2
Tasks: Text Generation
Modalities: Text
Formats: json
Languages: English
Size: 1K - 10K
ArXiv: arxiv:2502.05714
Tags: lean lean4 software_engineering proof verification
Libraries: Datasets pandas Croissant +1
License: mit
Dataset card
Data Studio
Files
Community 15
Settings
Dataset Viewer
Auto-converted to Parquet
API
Data Studio
Split (2)
train · 4.72k rows
Search this dataset
apps_id string · lengths 4 4
apps_question string · lengths 50 14k
spec string · lengths 91 3.17k
0000 Polycarp has n different binary words. A word called binary if it contains onl… def solve (n : Nat) (words List Int := sorry def abs (
0001 Mikhail walks on a Cartesian plane. He starts at the point $(0, 0)$, and in on… def solve_max_diagonal_move : Int := sorry def abs (n :
0002 You are given three sequences: $a_1, a_2, \ldots, a_n$; $b_1, b_2, \ldots,… def solve_sequence (n : Nat Nat) : List Nat := sorry th
0003 You have n barrels lined up in a row, numbered from left to right from one.… /-- Function signature for max_water_difference -/ def
0004 You are given a permutation $p=[p_1, p_2, \ldots, p_n]$ of integers from 1… def solve_beautiful_permuta List Nat) : List Nat := so
0005 The sequence of m integers is called the permutation if it contains all… def find_permutations (arr List (Nat × Nat) := sorry t
Previous 1 2 3 ... 48 Next
Gated dataset You have been granted access to this dataset
Proving the Coding Interview: Formally Verified APPS
Downloads last month 326
Use this dataset
Edit dataset card
Size of downloaded dataset files: 11.8 MB
Size of the auto-converted Parquet files: 4.89 MB
Number of rows: 4,715

Motivation

FVAPPS: Formally Verified APPS

- We want more proof certificates per synthetic LoC.

FVAPPS: Formally Verified APPS

- We want more proof certificates per synthetic LoC.
- Previously, APPS ([1]) scraped "job interview" style coding puzzles to be completed by Python synthesis.

FVAPPS: Formally Verified APPS

- We want more proof certificates per synthetic LoC.
- Previously, APPS ([1]) scraped "job interview" style coding puzzles to be completed by Python synthesis.
- In FVAPPS, we convert these **Python** problems to **Lean** problems, and state correctness theorems.

FVAPPS: Formally Verified APPS

- We want more proof certificates per synthetic LoC.
- Previously, APPS ([1]) scraped “job interview” style coding puzzles to be completed by Python synthesis.
- In FVAPPS, we convert these **Python** problems to **Lean** problems, and state correctness theorems.
- FVAPPS ships 4715 questions, each consisting of a function definition and 2-5 theorems, with a curated subset of 1083 samples.

FVAPPS: Formally Verified APPS

- We want more proof certificates per synthetic LoC.
- Previously, APPS ([1]) scraped "job interview" style coding puzzles to be completed by Python synthesis.
- In FVAPPS, we convert these **Python** problems to **Lean** problems, and state correctness theorems.
- FVAPPS ships 4715 questions, each consisting of a function definition and 2-5 theorems, with a curated subset of 1083 samples.

[1] D. Hendrycks *et al.*, "Measuring Coding Challenge Competence With APPS," *ArXiv*, 2021, [Online]. Available: https://api.semanticscholar.org/CorpusID:234790100

Formal verification and proof assistants: quality assurance

QA Process	Blindspot
Unit tests	What did I forget to test?
Fuzzing/property-based tests	Cases are non-exhaustive
Formal verification	The “world-spec gap” (i.e. sidechannels)

Formal verification and proof assistants: quality assurance

QA Process	**Blindspot**
Unit tests	What did I forget to test?
Fuzzing/property-based tests	Cases are non-exhaustive
Formal verification	The "world-spec gap" (i.e. sidechannels)

Proof assistants accomplish this degree of assurance by *exploiting inductive structure.*

Formal verification and proof assistants: compile time knowledge

- Python is ruled by *runtime knowledge*: the absence of an initial error message is tiny evidence that your program is correct
- A dependent type theory like Lean is ruled by *compiletime knowledge*: the absence of an error message is strong evidence that your program is correct.

Formal verification and proof assistants: out with math, in with software

- Most formal proof automation effort is invested into mathematics (i.e. MiniF2F)
- Instead, we could focus on **software** to bring the assurances of type theory to the real world
- This benchmark is a babystep in that direction

Benchmark Generation

Scaffold

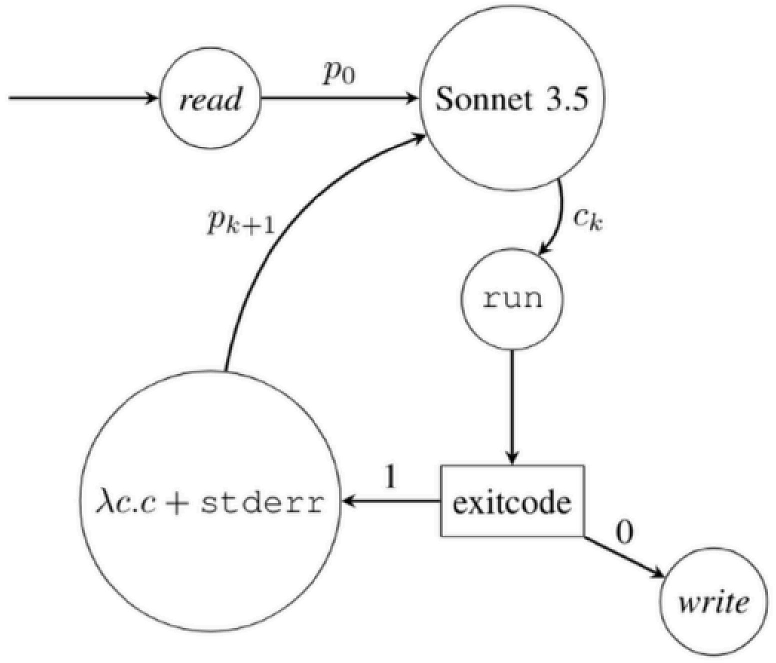


Fig. 3. Our generic scaffolding loop used at various stages of our pipeline. The `run` element is replaced with the `python`, `pytest`, `lean`, or `lake build` executable respectively.

- A *scaffold* or *agent* is an architecture involving LLM calls and observations (*tool use*).

Scaffold

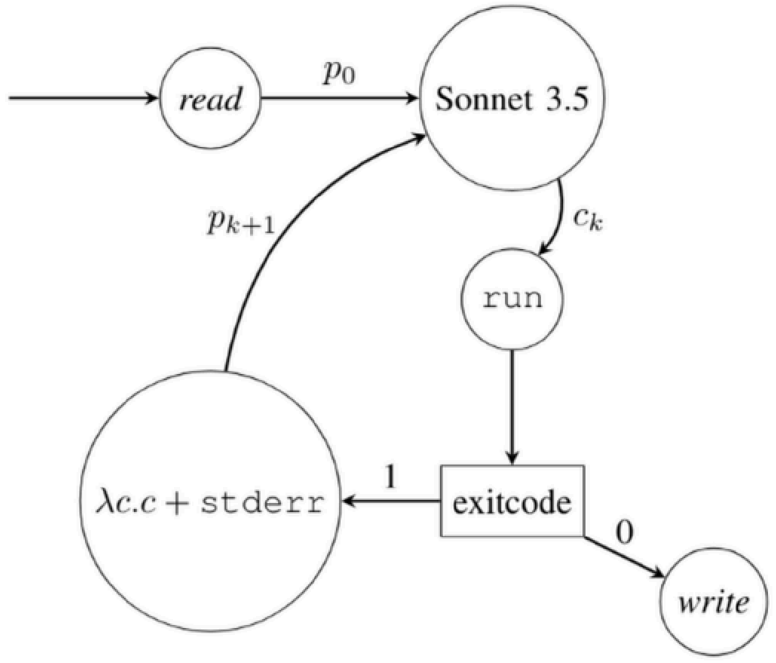


Fig. 3. Our generic scaffolding loop used at various stages of our pipeline. The `run` element is replaced with the `python`, `pytest`, `lean`, or `lake build` executable respectively.

- A *scaffold* or *agent* is an architecture involving LLM calls and observations (*tool use*).
- The simplest possible architecture, which is all we need, is a **loop**.

Benchmark Generation Pipeline

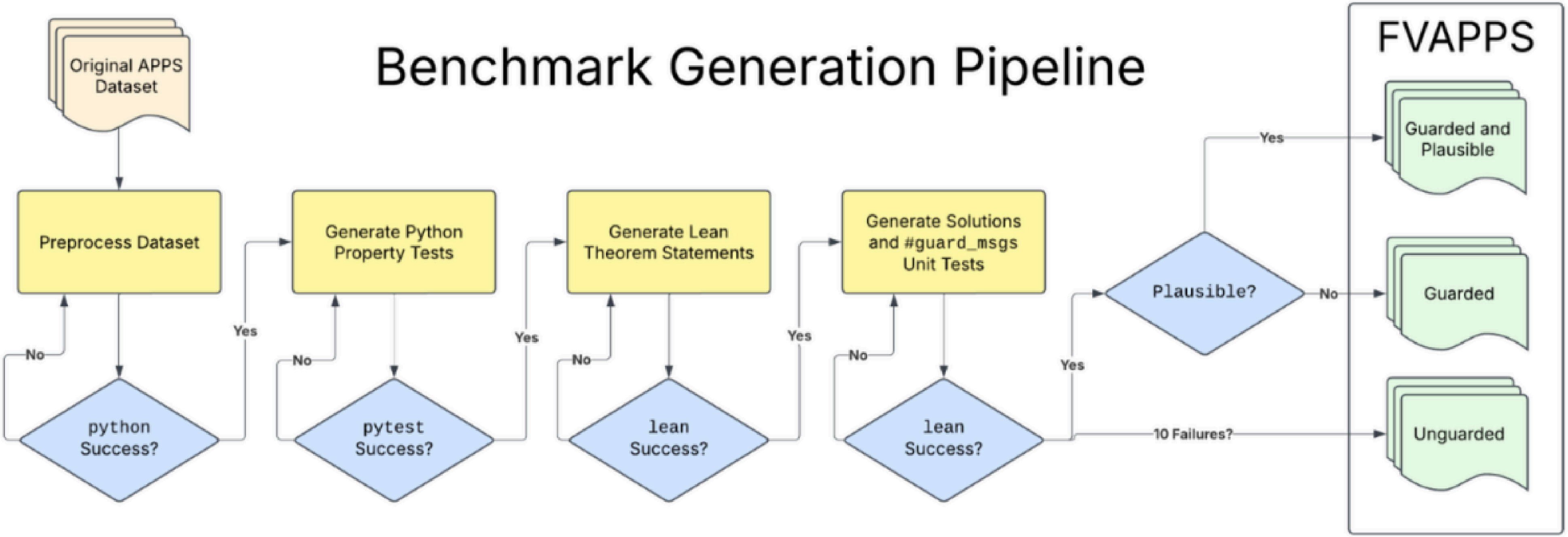


Fig. 1. Benchmark generation pipeline for creating coding interview theorem statements in Lean from APPS questions and solutions.

Example Sample

```
def solve_elections (n : Nat) (voters : List (Nat × Nat)) : Nat := sorry

theorem solve_elections_nonnegative (n : Nat) (voters : List (Nat × Nat)) : solve_elections n
    voters >= 0 := sorry

theorem solve_elections_upper_bound (n : Nat) (voters : List (Nat × Nat)) : solve_elections n
    voters <= List.foldl (λ acc (pair : Nat × Nat) => acc + pair.2) 0 voters := sorry

theorem solve_elections_zero_votes (n : Nat) (voters : List (Nat × Nat)) : (List.all voters
    (fun pair => pair.1 = 0)) -> solve_elections n voters = 0 := sorry

theorem solve_elections_single_zero_vote : solve_elections 1 [(0, 5)] = 0 := sorry
```

Fig. 2. FVAPPS sample 23, derived from train sample 23 of APPS source. The `def` is where the solver implements the function, each `theorem` is a correctness specification.

Theorems per sample

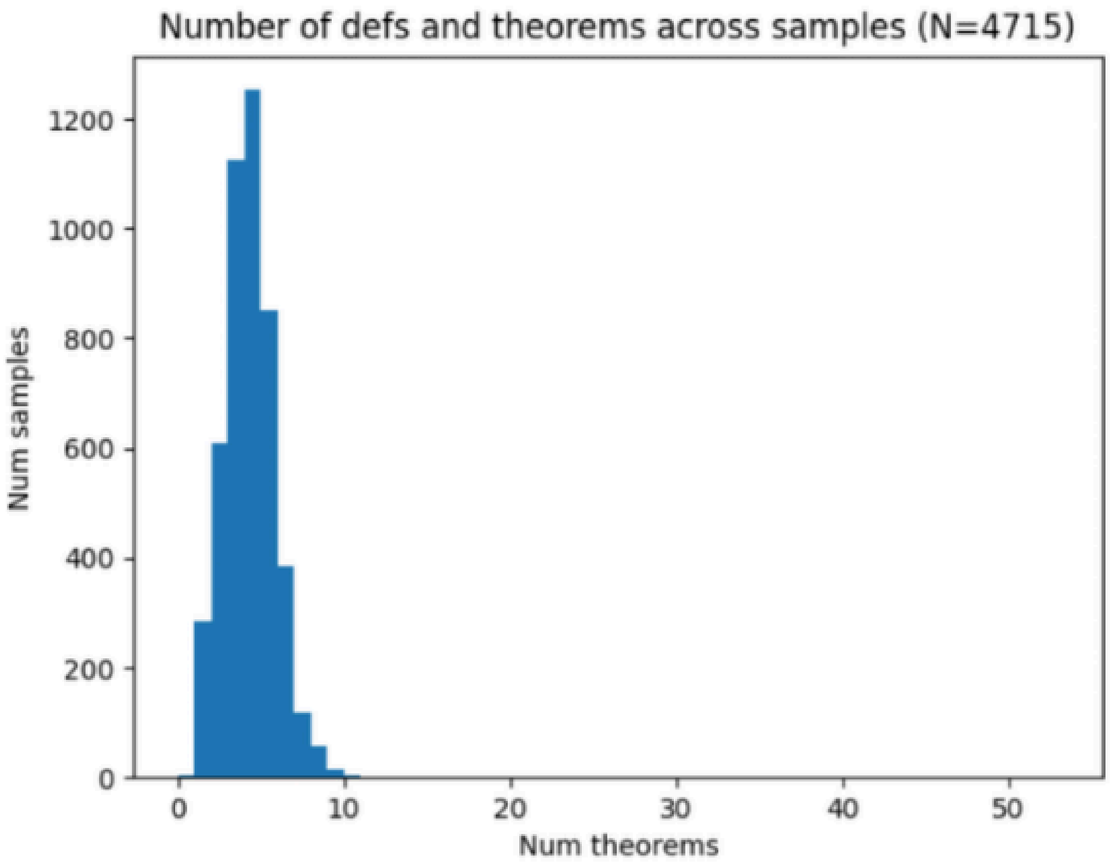


Fig. 4. Total number of defs and theorems across FVAPPS samples.

Baselines

What did we test?

- LLMs were given a loop scaffold similar to that in the generation.

What did we test?

- LLMs were given a loop scaffold similar to that in the generation.
- We measured Sonnet 3.5 (October 2024) and Gemini 1.5 Pro (retrieved November 2024)

What did we test?

- LLMs were given a loop scaffold similar to that in the generation.
- We measured Sonnet 3.5 (October 2024) and Gemini 1.5 Pro (retrieved November 2024)
- A human baseliner attempted one sample for 10 hours

Model baselines

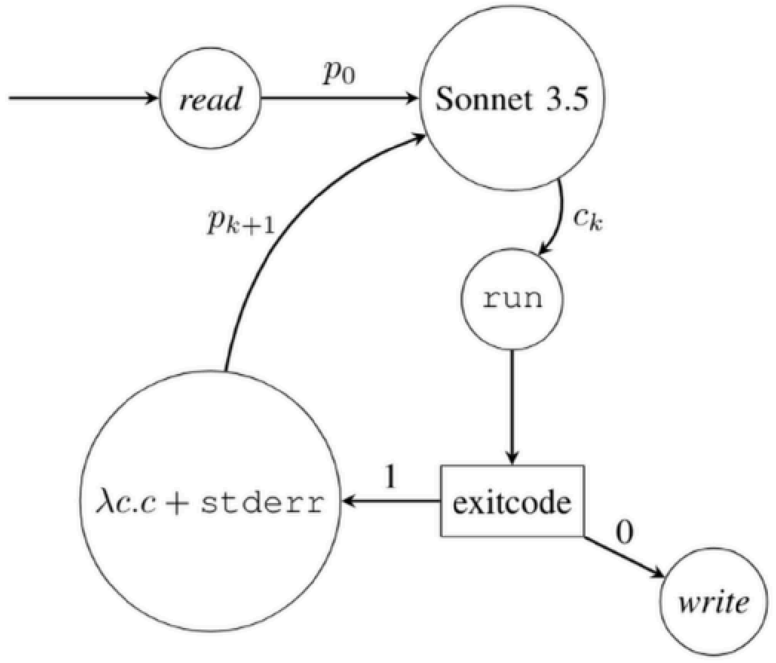


Fig. 3. Our generic scaffolding loop used at various stages of our pipeline. The `run` element is replaced with the `python`, `pytest`, `lean`, or `lake build` executable respectively.

406 theorems were attempted across 101 randomly selected samples

Each sample requires a function definition to be filled in before theorems can be attempted

On these, Sonnet proved 30% and Gemini proved 18%

Model baselines

	Baseline Counts	Sonnet Successes	Gemini Successes
Unguarded	69	41	28
Guarded	18	12	11
Guarded and Plausible	14	7	4
Total	101	60	43

TABLE II

BASELINE RESULTS SPLIT ACROSS QUALITY ASSURANCE STAGES OF OUR GENERATION PIPELINE.

Model baselines

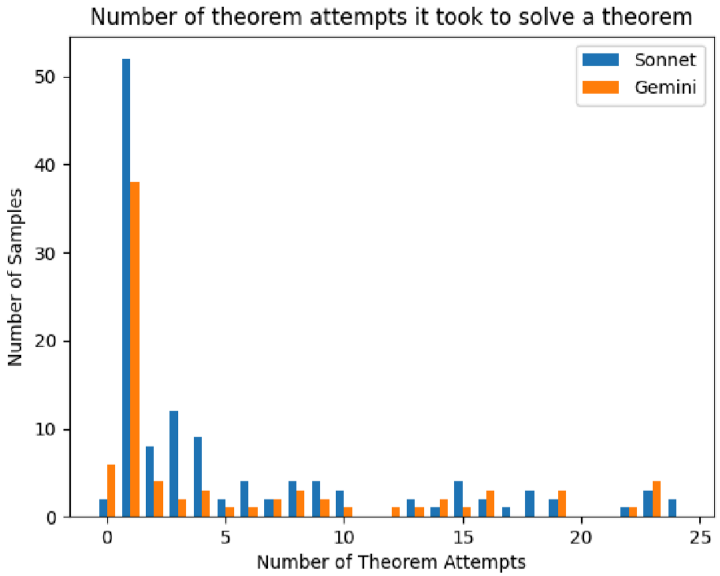


Fig. 7. Number of theorem attempts it took to solve a theorem, conditional on that theorem succeeding.

Of the theorems that got eventually completed, roughly 20% of each model's were done in zero or one iteration of the loop.

Model baselines

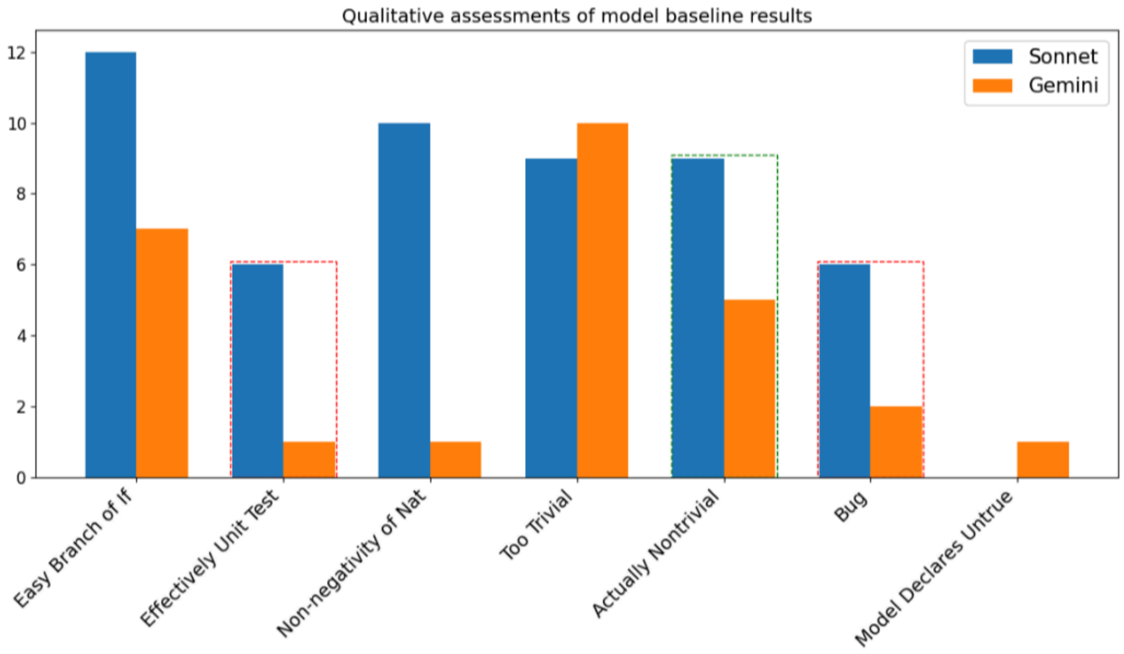


Fig. 6. Qualitative categories of theorem solutions in the 100 samples, first two theorems each sample. The red box shows completely spurious results, either bugs or a substitution of a quantified variable with a single value. The green box shows the most nontrivial results. The other categories are neither spurious nor impressive, though they require some syntactic fluency that many language models would fail at.

Future

Future

- Quality control for **soundness** (no false positives) could be improved

Future

- Quality control for **soundness** (no false positives) could be improved
- Harvesting property tests from the real world and turning them into Lean theorems (go from job interview code to real code)

References

[1] D. Hendrycks *et al.*, "Measuring Coding Challenge Competence With APPS," *ArXiv*, 2021, [Online]. Available: https://api.semanticscholar.org/CorpusID:234790100