

Code Generation Results Summary

Part 1. SLMs without fine-tuning

January 25, 2024

1. Code used to generate the below results

Code: <https://github.com/agnedil/code-generation>

2. Challenges

- The Replicate API library would not work directly, so I had to use its version within another library – LangChain.
- I used the code from this repo for HumanEval evaluation of all models: <https://github.com/openai/human-eval>. The multiprocessing module executed with an error: "AttributeError: Can't pickle local object in Multiprocessing". **I had to modify the original code to fix it.**
- The final check of the code correctness in the original OpenAI code is done by combining the problem (otherwise called starter code or function docstring) and completion: **problem["prompt"] + completion** which means the completion shouldn't contain the problem. Two issues: 1) incorrect indentation when joining the problem and completion – the code generates error when running, 2) some models may still misunderstand and include the problem into the completion (Llama 3 Instruct trained for chat, does it for 33% of cases). Therefore, I am using an additional prompt to ask LLMs to **include the problem definition (function docstring) into the completion**, and I exclude **problem["prompt"]** from the evaluated string. **I had to modify [the original code](#) to fix this.**
- **Summary of code modifications** (all in execution.py):
 - **Add class DillProcess** to fix the pickling issue (uses dill instead of pickle).
 - **Modify function check_correctness()** to have an extra argument use_prompt which controls the exclusion or addition of problem["prompt"] to the Python program to be run. The body of the function is modified to accommodate for the use of use_prompt.
 - Modify exception handling to **add error tracebacks** (helps when the error message is empty).
- SLMs tend to output **additional explanations** and clarifications like: "Here is the requested code completion:" etc. which break the automatic code execution during the verification stage. Adding more specific instructions like: "Complete the following code. Output only the runnable code and nothing else:" would still lead to non-runnable content like triple backticks in the output. As a result – **I had to provide minimum help to some models** by removing the initial or trailing triple backticks or strings like ````python`. Or by adding `from typing import List` as this was removed in the process (when LLM forgets to include it into the repeated func definition)

- **General approach to evaluate all models:** create one extensive and comprehensive prompt for all models. If any model fails to fully understand it and outputs code with human phrases or non-runnable symbols, it should be considered the drawback of the model.

3. Results

- **Llama 3 8B** – promising results.
- Non-chat optimized model **"meta/meta-llama-3-8b"** - several cases of hallucinations when functions are repeated and the code is incomplete in the end (stops at the middle of a function). See Appendix
- **Nous-hermes-2-solar-10.7b** – tries to explain the solution if no prompt is used (func docstring as prompt) – not runnable. 25.61% when using a prompt.
- **Gemma 7B** – incomprehensible output whether I include the prompt prefix or not.
- **Code Gemma 7b IT** (when asked to output the full func for HumanEval) – a) code generation template (per HG docs): unusable output – patchy pieces of code, sometimes 1 or 2 random lines, b) chat generation template: more usable output, but still a lot of errors in the first 5 problems: repeats def in the end, 2 out of 3 outputs were completions w/out func signature and docstring, 2 others contained extraneous text, e.g. the word “def” after the func was already provided, etc. Decided not to waste compute units – the leaderboard performance is still only 55%.
- **Phixtral** – generates code, but contains extraneous text (Here is a solution). If I do additional post-processing, this will be a disadvantage for other models. Also, it is very slow – up to 2 minutes per test case (5 hours for the entire run)
- **GPT-J-6B** – not fit for the task as the model is too weak, outputs hallucinations that remind of the expected output only very remotely (trained in 2021).
- **Yi-6B** is a bilingual (Chinese) model – pass@1 = 3% if not using prompt (function docstring as prompt), otherwise if using a prompt the model outputs some irrelevant snippets of code and. Asking to output the starter code concatenated with the completion doesn't help – the output still includes the completion without the beginning in most cases (<https://huggingface.co/01-ai/Yi-6B>).
- **Flan-T5** outputs complete nonsense that resembles code – completely not runnable.
- **Phi** – not designed for code completion. Outputs incomprehensible combinations of letters (“em”, “emlen”, “A”, “A.A.A.A.”, etc.) as generated code with or without a prompt (if with prompt, the model repeats the entire prompt before the incomprehensible output).
- **Phixtral-2x2_8** – MoE of two Phi models (4.5B), follows the instruction much better than Phi, reached Pass@1 = ~15%. Still outputs irrelevant human-like output although asked specifically not to do that: e.g. here's the code, here's the concatenated code, etc. Also, it takes ~1 min per API call which is a lot, considering there are 500 data points in the MBPP dataset.
- **Qwen1.5-7b** (replicate.com) – demonstrated a good result on HumanEval Pass@1 at ~44%, but only 20% on MBPP. The main challenge with this model is that it takes 200-300 s per one API call - took 1 day to run MBPP on replicate. This is unacceptable for experiments with agents as I will have to make several API calls per one agent call + run this for all 500 MBPP data points again – will take more than a day per experiment.
- **Mistral 7B** provided the expected result. The model would strip any docstrings from the functions – only the definition def was left. I helped the model by removing triple backticks

from start / end, “```python”, and adding “from typing import List” because the model would strip this import most of the times while the import is specific to the HumanEval dataset.

- **Codestral Mamba** – showed the best result on my leaderboard, followed by **Ministral 8B** and, surprisingly, **Ministral 3B**. The latter is the smallest model that I tried, but it outperformed many other models that are 2 to 2.5 times bigger, and even the models with 22B parameters (7+ times bigger). Other models from the Mistral family also showed good results which, on average, make this group of models as the leading one among all other models.
- **OpenCodeInterpreter-DS-6.7B** and **Artigenz-Coder-DS-6.7B** – when asked to output the entire function, keeps saying “Here is the completed function” (even if I ask not to do it in the prompt). Model needs extra help by getting the code placed between ```python and ```. **May be better at pure code completion?**
- **Mamba 2.8B** (replicate.com): if not using a prompt (func docstring as prompt) – the model tries to generate a completion, but then follows a paragraph of hallucinations that look like human free-form text with how-to questions about software development. When using a prompt – the model doesn’t even try to complete the code – it starts hallucinating right away (see saved file with examples).
- Gemma 7B, Gemma 2B, Flan-T5, Phi, Mamba 2.8B (replicate.com) – incoherent output.
- **Deepseek-Coder-6.7B-Instruct** – scored great on HumanEval, but did only 1% on MBPP, mainly because the model outputs unnecessary explanations, although it is explicitly asked not to do that. Example: “Sure, here is the Python function that calculates.” This is done for every data point. Somewhat similar numbers are for OpenCodeInterpreter-DS-6.7B. Reason is same: unnecessary clarifications when asked not to do it: “Here is the Python function that satisfies the given tests:” *Solution – maybe decrease temperature?*
- **Llama 3.1 8B Instruct** – released fall 2024. Inference takes an average of 2 minutes for Human Eval and 0.75 min for MBPP. Both tasks required 4 hours to finish running in Google Colab on an A100 GPU which is the best available. This may be too long for subsequent experiments, but I was able to get this model work for the Reflection workflow – **TODO: add timer for the entire notebook.**
- **Phixtral** on Replicate – takes 100 to 200 seconds per API call. Running this model for Big Code Bench (500 data points) took well all night and up to the lunch time of the next day. Considering that the results from this model are very low in general, I will discontinue using it for agent experiments as they will take even more time due to several API calls per iteration. Maybe use the HuggingFace version of the model? What if the HF version is more up-to-date and faster?
- **NxCode** is not only one of the best models quality-wise, but it also ran much faster than the models that started at the same time or even earlier (as measured on plain agent on MBPP) – OpenCodeInterpreter was considerably slower while Artigenz was the slowest - twice as slow (and one of the worst quality-wise). Llama is also relatively fast – it finished after NxCode, but before OpenCodeInterpreter. Another slow model – DeepseekCoder which was twice as slow as CodeQwen. Code Gemma ran even faster than CodeQwen.

All models received slight help by stripping ``` backticks at edges including the ```python string + adding “from typing import List” which is often stripped by SLMs.

The pass@1 scores below are provided for my run (first number) and then for the result shown on the Big Code Leaderboard (second number), if it is available:

<https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard>

Table 1. Pass@1 Score for Testing SLMs on Multiple Datasets

Model	Hosted By	Model Size	Human-Eval Full Func (Me / Big Code)	H-E Compl	MBPP	LBPP	Big Code Bench	Rank	Temp / top_p	Cost \$, full func	Notes
Small Language Models (SLMs)											
Nxcode-CQ-7B-orpo	Google Colab	7.25B	82.93 / 87.23	75.61%	73%	22.84%	24%	1	1.0 / 1.0	\$50/month	License needed if > 100M commercial users
Codestral Mamba	mistral.ai	7.3B	75.61% / 75%	60.37%	39.4%	26.54%	23%	3	0.7 / 1.0	0.02	Apache 2
Ministral 8B	mistral.ai	8B	72.56% / 76.8% (instruct)	71.34%	56.2%	22.22%	24.6	2	0.3 / 1.0	0.01	Mistral Commercial License, Mistral Research License
Deepseek-Coder-6.7B-Instruct	Google Colab	6.7	65.24% / 80.22%	70.73%	1%	0% (extra words!)	32.2%	9	1.0 / 1.0	\$50/m	Free responsible use license
Ministral 3B	mistral.ai	3B	64.63% / 77.4% (instruct)	61.59% / 77.4% (instruct)	51.8%	20.99%	26.8%	5	0.3 / 1.0	0.01	Mistral Commercial License
Mistral-Nemo-Instruct-2407	mistral.ai	12B	58.54%/67%	53.05%	47.4%	21.6%	17.2%	7	0.3 / 1.0	0.01	Apache2
Llama 3.1 8B Instruct	Google Colab	8B	65.9% / 72.6%	55.47%	56.8%	21.6%	29.8%	4			Commercial use license if > 700M users
CodeQwen1.5-7B-Chat	Google Colab	7B	50% / 87.2%	54.88%	55.2%	19.75%	26.8%	6	1.0 / 1.0	\$50/m	Requires license if > 700M users
OpenCodeInterpreter-DS-6.7B	Google Colab	6.7	41% / 73.2%	71.95 / 73.2%	5.4%	8%	32.2%	8	1.0 / 1.0	\$50/m	Apache 2

Model	Hosted By	Model Size	Human-Eval Full Func (Me / Big Code)	H-E Compl	MBPP	LBPP	Big Code Bench	Rank	Temp / top_p	Cost \$, full func	Notes
Mistral 7B, open-mistral-7b	mistral.ai	7B	31.1% / 30.5%	35.98% / 30.5%	13.6%	9.9%	15.6%	11	0.7 / 1.0	0.01	Apache2
Nous-hermes-2-solar-10.7b	replicate.com	10.7B	25.61%	28.65%	30.4%	8%	43.4%	10	0.95 / 1	0.61	Available on HF as NousResearch/Nous-Hermes-2-SOLAR-10.7B Apache 2 License
Phixtral-2x2_8 (4.5B)	replicate.com	4.5B	14.64%	34.756 %	14.6%	4.9	20.6%	14	0.95 / 1	2.77	Available on HF as mlabonne/p hixtral-2x2_8 (4.5B). See also mlabonne/p hixtral-4x2_8 (7.8B) MIT License
Artigenz-Coder-DS-6.7B	Google Colab	6.7B	1.22% / 70.89%	73.17%	0.2%	4.32%	13.4%	13	1.0 / 1.0	\$50/m.	Free responsible use
Code Gemma 7b IT	Google Colab	7B	0% (? model)	27.44%	51% (chat model)	19.14% (chat model)	2.4%	12	1.0 / 1.0		Free use
Slightly Bigger SLMs											
Mistral-Small-2409	mistral.ai	22B	70.73% / 80%	64.63%	60.2%	25.3%	20.6%		0.7 / 1.0	0.03	
Codestral latest	mistral.ai	22.2B	26.83% / 81.1%	64.63%	37%	48.15	12.8%		0.7 / 1.0	0.15	

Model	Hosted By	Model Size	Human-Eval Full Funk (Me / Big Code)	H-E Compl	MBPP	LBPP	Big Code Bench	Rank	Temp / top_p	Cost \$, full func	Notes
Mixtral-8x7B-v0.1	mistral.ai	12B active (47B total)	16.46% / 40.2%	33.54%	0%	5.6%	11.8%		0.7 / 1.0	0.05	
Not Useful SLMs											
Qwen1.5-7b	replicate.com	7B	43.9%	200 s per API call	19.4%	200 s per API call	200 s API call	200 s API call	0.95 / 1	3.55	
Llama 3 8B	Replicate	8B	51.83% / 45.65%	API Error	API Error	API Error	API Error		0.95 / 1	0.29	
Yi 6B	replicate.com	6B	3%	3%	0.2%				0.95 / 1	0.44	
Gemma 7B	replicate.com	7B	0 %	0%	0%			0%	0.95 / 1	0.05	
Gemma 2B	replicate.com	2B	0 %	4.26%	4.6%				0.95 / 1	0.05	
Flan-T5	replicate.com		0%						0.95 / 1		
Phi-2	replicate.com		0%	Incoherent (even if temp=0.25)					0.95 / 1		
Mamba 2.8B	replicate.com	2.8B	n/a	Incoherent (even if temp=0.25)	0%			0	0.95 / 1	0.02 (20 calls)	

Notes

1. The second number in the HumanEval Full Funk column is the performance of a model on the HuggingFace's Big Code Models Leaderboard (<https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard>) – not to be confused with the Big Code Benchmark dataset because HuggingFace presents only the humanEval results in its leaderboard.

2. HuggingFace transformer models' default temperature and top_p parameters are explained here: https://huggingface.co/docs/transformers/v4.22.2/en/main_classes/text_generation. Usually they are 1.0 and 1.0, respectively, and can be checked by running model.config.temperature and model.config.top_p.

Table 2. Model Versions (Table Composed on January 25, 2025)

Model	Hosted By	Model Size	Model Version
NTQAI/Nxcode-CQ-7B-orpo	Google Colab	7.25B	Model version as seen on https://huggingface.co/Artigenz/Artigenz-Coder-DS-6.7B/commits/main (from model card click on Files and Versions and then on History: 7 commits (3 commits may be different)): 74f3b3c06de36b261af9ef857279d6e33f893336, commit of May 30, 2024
Codestral Mamba	mistral.ai	7.3B	Endpoint: open-codestral-mamba. Version: v 0.1
Ministral 8B	mistral.ai	8B	Endpoint: ministral-8b-latest. Version: 24.10
deepseek-ai/deepseek-coder-6.7b-instruct	Google Colab		Model version as seen on https://huggingface.co/google/codegemma-7b-it/commits/main): e5d64add26a6a1db0f9b863abf6ee3141936807, commit of Feb 1, 2024
Ministral 3B	mistral.ai	3B	Endpoint: ministral-3b-latest. Version: 24.10
Mistral-Nemo-Instruct-2407	mistral.ai	12B	Endpoint: open-mistral-nemo. Version: 24.07
meta-llama/Meta-Llama-3.1-8B-Instruct	Google Colab	8B	Model version as seen on https://huggingface.co/google/codegemma-7b-it/commits/main): 0e9e39f249a16976918f6564b8830bc894c89659, commit of Sep 25, 2024
Qwen/CodeQwen1.5-7B-Chat	Google Colab		Model version as seen on https://huggingface.co/google/codegemma-7b-it/commits/main): 7b0cc3380fe815e6f08fe2f80c03e05a8b1883d8, commit of April 30, 2024
m-a-p/OpenCodeInterpreter-DS-6.7B	Google Colab		Model version as seen on https://huggingface.co/google/codegemma-7b-it/commits/main): 60b89884df814590abd76757a6db4a527cbdfc91, commit of Mar 3, 2024
Mistral 7B	mistral.ai	7B	Endpoint: open-mistral-7b. Version: v0.3
Nous-hermes-2-solar-10.7b	replicate.com	10.7B	nateraw/nous-hermes-2-solar-10.7b:1e918ab6ffd5872c21fba21a511f344fd12ac0edff6302c9cd260395c7707ff4 (1 year ago)
Phixtral-2x2_8 (4.5B)	replicate.com	4.5B	lucataco/phixtral-2x2_8:25d7b93bb0ec9e8dd94fcc69adc786759243a5628ba5574bd9609d6abafe57cf (11 months, 2 weeks ago)
Artigenz/Artigenz-Coder-DS-6.7B	Google Colab		Model version as seen on https://huggingface.co/google/codegemma-7b-it/commits/main):

Model	Hosted By	Model Size	Model Version
			a0dea4a1c6cfdef8043c8accffa803887f444f45, commit of April 16
google/codegemma-7b-it	Google Colab	7B	Model version as seen on https://huggingface.co/google/codegemma-7b-it/commits/main : 078cdc51070553d1636d645c9a238f3b0914459a, commit of Aug 7, 2024
Slightly Bigger SLMs			
Mistral-Small-2409	mistral.ai	22B	Endpoint: mistral-small-latest. Version: 24.09
Codestral latest	mistral.ai	22.2B	Endpoint: codestral-latest. Version: 25.01
Mixtral-8x7B-v0.1	mistral.ai	12B active (47B total)	Endpoint: open-mixtral-8x7b. Version: v0.1
Not useful SLMs			
Qwen1.5-7b	replicate.com	7B	lucataco/qwen1.5-7b:f85bec5b21ba0860e0f200becf5af9d5a65b974b9f99e36eb036d21eab884de (11 months, 2 weeks ago)
Llama 3 8B	Replicate	8B	No version shown on replicate.com – hence the API error
Yi 6B (non-chat)	replicate.com	6B	01-ai/yi-6b:d302e64fad6b4d85d47b3d1ed569b06107504f5717ee1ec12136987bec1e94f1 (1 year 2 months ago)
Gemma 7B	replicate.com	7B	google-deepmind/gemma-7b:2ca65f463a2c0cfef4dbc4ba70d227ed96455ef6020c1f6983b2a4c4f3ecb4ec (11 months ago)
Gemma 2B	replicate.com	2B	google-deepmind/gemma-2b:26b2c530f16236a4816611509730c2e6f7b27875a6d33ec5cff42961750c98d8 (11 months ago)
Flan-T5	replicate.com		replicate/flan-t5-xl:eec2f71c986dfa3b7a5d842d22e1130550f015720966bec48beaae059b19ef4c (1 year 9 months ago)
Phi-2	replicate.com		lucataco/phi-2:740618b0c24c0ea4ce5f49fcfef02fcd0bdd6a9f1b0c5e7c02ad78e9b3b190a6 (11 months, 3 weeks ago)
Mamba 2.8B	replicate.com	2.8B	adirik/mamba-2.8b:571abd73203a3dd3d7071f1c0380a3502c427aba98a2fb5edf2f7cfdeea1676c (11 months, 2 weeks ago)

Source of model versioning information:

1. https://docs.mistral.ai/getting-started/models/models_overview/
2. <https://replicate.com/explore>

3. To determine versions for HuggingFace models, see the sha hash + date for the latest commit here: from **model card** click on **Files and versions**, then click on **History: 7 commits** (# commits may be different)

4. Analysis

- **Nxcode-CQ-7B-orpo** stands out as the top-scoring SLM overall.
- Other strong contenders include **Ministral 8B**, **Deepseek-Coder-6.7B**, and **Llama 3.1**.
- Several mid-tier models (e.g., CodeQwen1.5-7B-Chat, Mistral 12B Nemo, OpenCodeInterpreter) show moderate but inconsistent performance.
- Several models fall into low or nearly unusable categories due to extremely low pass rates or major practical limitations (long latencies, API errors, or nonsense outputs).
- **Larger parameter counts** do **not** always guarantee higher pass@1!

Top Performers

1. **Nxcode-CQ-7B-orpo**
 - *Human-Eval pass@1*: 82.93% / 87.23% (very high)
 - Solid MBPP (~73%) and LBPP (~22–24%)
 - Overall among the highest marks on multiple benchmarks.
2. **Ministral 8B**
 - *Human-Eval pass@1*: 72.56% / 76.8%
 - MBPP around 71%, mid-50% on other tasks, overall strong.
3. **Deepseek-Coder-6.7B-Instruct**
 - *Human-Eval pass@1*: 65.24% / 80.22%
 - Does well on MBPP (~70%) but struggles on LBPP (1%).
 - Not as consistently high as Nxcode, but still a strong contender.
4. **Ministral 3B** (with “instruct” variant)
 - *Human-Eval pass@1*: up to ~64.63% / 77.4% in instruct mode
 - Fairly good MBPP (~61–77%), decent LBPP (~20–27%).
 - Punches above its parameter count.
5. **Llama 3.1 8B Instruct**
 - *Human-Eval pass@1*: ~65.9% / 72.6%
 - MBPP and LBPP in the mid 50–60% range, a respectable showing.

These top models generally exceed ~60% pass@1 on Human-Eval (sometimes well above 70–80%), plus moderate to good results on MBPP and other code tasks.

Mid Performers

1. **CodeQwen1.5-7B-Chat**
 - *Human-Eval pass@1*: 50% / 87.2% (unclear if the 87.2% is a different setting)
 - MBPP ~55%, LBPP ~19–26%.
 - Results are somewhat mixed but places it in a middle tier on average.

2. **Mistral-Nemo-Instruct-2407 (12B)**

- *Human-Eval pass@1*: ~58.5% / 67%
- MBPP ~47%, LBPP ~21–22%.
- Decent but not top-tier.

3. **OpenCodeInterpreter-DS-6.7B**

- *Human-Eval pass@1*: ~41% / 73.2%
- Good MBPP (~72%), but quite low performance on some tasks like LBPP (5–8%).

4. **Mistral 7B** (open-mistral-7b)

- *Human-Eval pass@1*: ~31%
- MBPP ~13.6%, LBPP ~9.9%.
- Sits lower than the ones above, but still not in the “near-zero” group.

Many of these mid-range models have partial strong points (e.g., decent MBPP or decent instruct performance) but are inconsistent across benchmarks.

Low Performers

A number of models show **very low** pass@1 on Human-Eval (often near 0–25%) or produce mostly irrelevant outputs:

- **Nous-hermes-2-solar-10.7b** (~25.6% Human-Eval)
- **Phixtral-2x2_8 (4.5B)** (~14.64%)
- **Artigenz-Coder-DS-6.7B** (1.22% / 70.89% in some mode, but near 0% in others)
- **Code Gemma 7b IT** (0% on some tasks)

And several models from the “Not Useful SLMs” section with near-zero performance or major usability problems (API errors, extremely long latencies, or nonsense outputs):

- **Qwen1.5-7B** (unusable due to 200 s API calls)
- **Llama 3 8B (Replicate)** (API errors)
- **Yi 6B, Gemma 7B, Gemma 2B, Flan-T5, Phi, Mamba 2.8B** all show 0–3% pass@1 or produce irrelevant outputs.

5. Summary of what was done

- Table contains much more data now.
- Finished the **first and second HumanEval runs and the MBPP run** – CONSIDERABLE EFFORT as the dataset has 500 data points which means the code needs to be generated and verified 500 times.
- According to (Matton et al. 2024), **data leakage** in code generation occurs when popular evaluation benchmarks (like HumanEval and MBPP) appear in a model’s training data and, whether intentionally or unintentionally, compromise the validity of test scores. Therefore, two more runs were done on the **LBPP** and **Big Code Bench** dataset for all models.
- Conducted an initial experiment with the **Reflection agentic workflow**.
- Need to continue applying various agentic workflows.
- Need to finish the **Methodology** section

References

Matton A., Tom Sherborne, Dennis Aumiller, Elena Tommasone, Milad Alizadeh, Jingyi He, Raymond Ma, Maxime Voisin, Ellen Gilsenan-McMahon, Matthias Gallé. 2024. **On Leakage of Code Generation Evaluation Datasets.**

Visuals

1. Pass@1 Scores Visualized by Model / Dataset

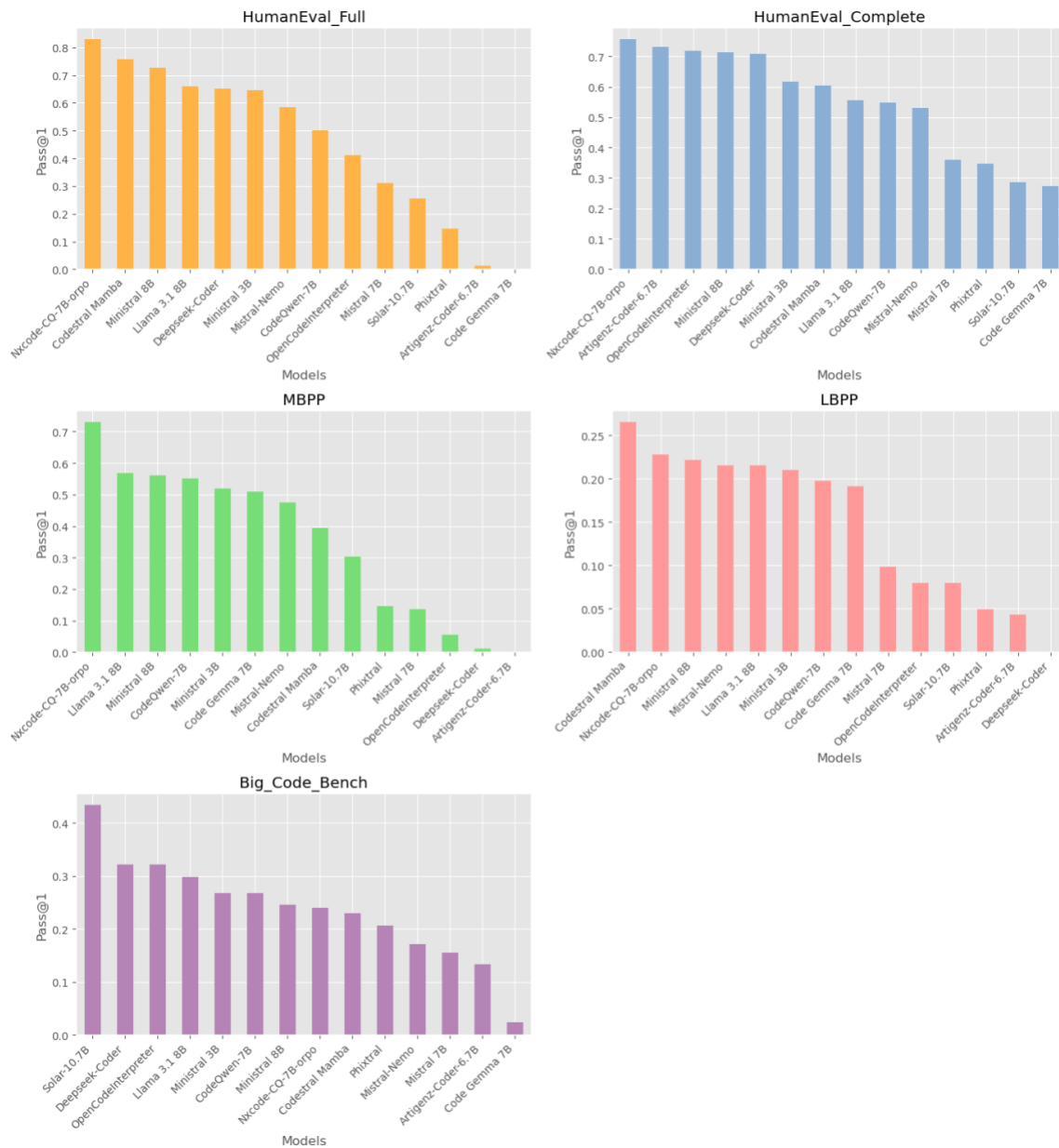


Figure 1. Pass@1 Scores Visualized by Model / Dataset

2. Initial Results for Models by Dataset

	Model	HumanEval_Full	HumanEval_Complete	MBPP	LBPP	Big_Code_Bench
0	Nxcode-CQ-7B-orpo	0.8293	0.75610	0.730	0.2284	0.240
1	Codestral Mamba	0.7561	0.60370	0.394	0.2654	0.230
2	Ministral 8B	0.7256	0.71340	0.562	0.2222	0.246
3	Deepseek-Coder	0.6524	0.70730	0.010	0.0000	0.322
4	Ministral 3B	0.6463	0.61590	0.518	0.2099	0.268
5	Mistral-Nemo	0.5854	0.53050	0.474	0.2160	0.172
6	Llama 3.1 8B	0.6590	0.55470	0.568	0.2160	0.298
7	CodeQwen-7B	0.5000	0.54880	0.552	0.1975	0.268
8	OpenCodeInterpreter	0.4100	0.71950	0.054	0.0800	0.322
9	Mistral 7B	0.3110	0.35980	0.136	0.0990	0.156
10	Artigenz-Coder-6.7B	0.0122	0.73170	0.002	0.0432	0.134
11	Code Gemma 7B	0.0000	0.27440	0.510	0.1914	0.024
12	Solar-10.7B	0.2561	0.28650	0.304	0.0800	0.434
13	Phixtral	0.1464	0.34756	0.146	0.0490	0.206

Table 1. Initial Results for Models by Dataset

3. Normalized Results with Final Ranking

	Model	HumanEval_Full	HumanEval_Complete	MBPP	LBPP	Big_Code_Bench	Average
0	Nxcode-CQ-7B-orpo	1.000000	1.000000	1.000000	0.860588	0.526829	0.877483
2	Ministral 8B	0.874955	0.911356	0.769231	0.837227	0.541463	0.786846
1	Codestral Mamba	0.911733	0.683621	0.538462	1.000000	0.502439	0.727251
6	Llama 3.1 8B	0.794646	0.581897	0.777473	0.813866	0.668293	0.727235
4	Ministral 3B	0.779332	0.708947	0.708791	0.790882	0.595122	0.716615
7	CodeQwen-7B	0.602918	0.569649	0.755495	0.744160	0.595122	0.653469
5	Mistral-Nemo	0.705897	0.531659	0.648352	0.813866	0.360976	0.612150
8	OpenCodeInterpreter	0.494393	0.924019	0.071429	0.301432	0.726829	0.503620
3	Deepseek-Coder	0.786688	0.898692	0.010989	0.000000	0.726829	0.484640
12	Solar-10.7B	0.308815	0.025119	0.414835	0.301432	1.000000	0.410040
9	Mistral 7B	0.375015	0.177289	0.184066	0.373022	0.321951	0.286269
11	Code Gemma 7B	0.000000	0.000000	0.697802	0.721176	0.000000	0.283796
10	Artigenz-Coder-6.7B	0.014711	0.949346	0.000000	0.162773	0.268293	0.279025
13	Phixtral	0.176534	0.151879	0.197802	0.184627	0.443902	0.230949

Table 2. Normalized Results. See **Average** column for final ranks (after min max scaling).

Next Steps

(This was not submitted on January 25 – for my reference only)

1. Finish the **temperature & top_k** experiments – decide which models to discontinue (Phixtral and Artigenx? Or more?).
2. Finalize the format of the **Reflection workflow**: keep simple as I did with Llama 3.1 or use Langchain or another format?
3. Run Reflection workflow for **all models all datasets**.
4. Do 1 experiment with **another agentic workflow** (Llama?)
5. Finish the Chapter 3 **Methodology**.

EVERYTHING ABOVE WAS SUBMITTED ON JANUARY 25, 2025. THIS IS THE LATEST REVISION OF THIS INFORMATION.

Part 2. Agents

2.1 February 8, 2025 Submission

In the last two week I did multiple runs of the plain reflection agent on 7 models across 4 datasets. Each run takes some 1 to 2 hours. Here are the current results, and I will share more insights and analysis in the coming weeks.

Model	HumanEval_Complete	HumanEval_Complete_Plain_reflection	MBPP	MBPP_Plain_reflection	LBPP	LBPP_Plain_reflection	Big_Code_Bench	Big_Code_Bench_Plain_reflection
Nxcode-CQ-7B-orpo	0.7561	0.7865	0.7300	0.4720	0.2284	22.2200	0.2400	0.2520
Deepseek-Coder	0.7073	0.6768	0.0100	0.0900	0.0000	0.0061	0.3220	0.1280
Llama 3.1 8B	0.5547	0.5731	0.5680	0.5200	0.2160	19.1400	0.2980	0.4180
CodeQwen-7B	0.5488	0.5487	0.5520	0.4960	0.1975	0.1790	0.2680	0.2300
OpenCodeInterpreter	0.7195	0.0000	0.0540	0.1860	0.0800		0.3220	0.0000
Artigenz-Coder-6.7B	0.7317	13.4100	0.0020	0.0380	0.0432	0.0123	0.1340	0.0560
Code Gemma 7B	0.2744	35.9700	0.5100	0.4780	0.1914	16.7000	0.0240	0.0740

2.2 February 21, 2025 Submission

My current, Feb 21 submission is Section 2.2 which also starts on page 14 of the attached file and spreads over the next 12 pages.

My current submission is an error analysis of implementing the reflection agentic workflow. I went over a lot of log files and analyzed the most typical errors. Based on this, I created the three TODOs at the very end of Section 2.2 which serve as my current plan to move forward:

- TODO #1 contains a list of typical errors made by ML models when generating the code. I could use some additional means to correct such errors programmatically. As I mentioned I already started implementing code for TODO #1 in this Jupyter notebook: <https://github.com/agnedil/code-generation/blob/main/notebooks/Generated%20code%20validation%20playground.ipynb>. The plan is to finish the implementation after I submit Chapter 3.
- Then, I will move on to TODO #2 containing some non-agentic improvements of the process.
- And then finally to TODO #3 which constitutes the next step in my agentic workflow implementation - the collaboration agent that will hopefully improve the final results based on the lessons learned from implementing the reflection agentic workflow.

Analysis of Errors in Plain Reflection Agents

Part 1. MBPP

- **OpenCodeInterpreter-DS-6.7B** (MBPP dataset): There are some 20% of cases when the model outputs only code and nothing else. Although asked specifically in the prompt not to do this, the model often adds human-like explanations before and after the code + code fences + test cases:
 - ````python ... ````. Sometimes twice – first for the func per se, second time for a testing func or assert statements.
 - **Human-style clarifications:**
 - Here is the improved solution to the problem:
 - Here is the python function that adds pairwise for tuples:
 - The code uses the built-in `zip` function to pair the elements in the tuples. `zip` pairs the elements of the tuples in parallel
 - Here is a Python function for ... It takes three parameters: an array, an integer `n`, and another integer `m`. It calculates the product of ...
 - Note: This solution corrects the errors in the proposed solution and removes redundant code.
 - The proposed solution is correct and can drop empty items from a given dictionary. Here is the improved solution:
 - **Testing code:**
 - ```
print(tuple_intersection([(3, 4), (5, 6), (9, 10), (4, 5)], [(5, 4), (3, 4), (6, 5), (9, 11)]))
```
    - **Prints this after the func's return statement:**

Example 1:

```
print(find_adverb_position("clearly!! we can see the sky"))
print(find_adverb_position("seriously!! there are many roses"))
print(find_adverb_position("unfortunately!! sita is going to home"))
```

Example 2:

```
def upper_ctr(s):
 return sum(1 for c in s if c.isupper())
print(upper_ctr('PYthon'))
print(upper_ctr('BigData'))
print(upper_ctr('program'))
```

**To catch this I need to know the func name.**
    - Simple assert statements: `assert drop_empty({'c1': 'Red', 'c2': 'Green', 'c3': None})=={'c1': 'Red', 'c2': 'Green'}`.  
**Sometimes everything is fine (no explanations or code fences), but only the assert statements need to be removed.**
    - More complex test code:
 

```
def tests():
 assert decimal_to_Octal(10) == "12"
 assert decimal_to_Octal(2) == "2"
 assert decimal_to_Octal(33) == "41"
 print("All tests passed!")
```

Example 2:

```
```python
def decimal_to_Octal(decimal):
    return oct(decimal)[2:]
```



```
def tests():
    assert decimal_to_Octal(10) == "12"
    assert decimal_to_Octal(2) == "2"
    assert decimal_to_Octal(33) == "41"
    print("All tests passed!")

tests()
'''
```

Not sure what to do with this one. If this pattern repeats with other modes, I may introduce this case into the cleab_code() func.

- **NxCode**

- **Exceptional prompt following ability** – many cases when only the code outputs and nothing but the code (as asked in the prompt)
- ````python OR ```python...```` - multiple cases
- Clarifications:
 - Here is the improved solution:
- **Testing code:**

Example 1

```
assert max_length_list([[0], [1, 3], [5, 7], [9, 11], [13, 15, 17]])==(3, [13, 15, 17])
assert
max_length_list([[1,2,3,4,5], [1,2,3,4], [1,2,3], [1,2], [1]])==(5, [1,2,3,4,5])
assert
max_length_list([[3,4,5], [6,7,8,9], [10,11,12]])==(4, [6,7,8,9])

def max_length_list(l1):
    max_length_sublist = max(l1, key=lambda sublist:
len(sublist))
    return (len(max_length_sublist), max_length_sublist)
```

Example 2:

Here is the improved solution:

```
```python
def find_parity(x):
 x ^= x >> 1
 x ^= x >> 2
 x ^= x >> 4
 x ^= x >> 8
 x ^= x >> 16
 return "Even Parity" if x & 1 else "Odd Parity"
print(find_parity(12))
print(find_parity(7))
print(find_parity(10))
MULTIPLE CASES OF PRINT(FUNC_NAME(...)) ANS ASSERT STATEMENTS
```

- Wrong test logic:  
EXAMPLE 1

```
def neg_nos(lst):
 return [i for i in lst if i < 0]
```

```
neg_nos([-1, 4, 5, -6])
neg_nos([-1, -2, 3, 4])
neg_nos([-7, -6, 8, 9])
```

MISSING ASSERT KEYWORD

### EXAMPLE 2

Here is the improved Python code:

```
def rectangle_area(length, width):
 return length * width
```

MISSING PYTHON KEYWORD

### EXAMPLE 3

Improved Completion:

```
res = [sub[0] for sub in test_list]
return (res)
```

MISSING FUNC HEADER

### EXAMPLE 4

Improved Completion:

```
radius * 2
```

- **Llama 3.1 & CodeGemma (MBPP, LBPP)** - follow instructions in an excellent way – no human-like text, no test cases, nothing extraneous, just the code. ALL cases are clean like that! So in the end, it's how correct the clean code is.
- **Artigenz (MBPP, LBPP)** – just a talkative model, every time it says something like “Here is an improved solution:...”. NOTE: in most cases, the model uses either ````python ...```` or ````Python ...```` and also a lot of assert statements and `print(func_name())` statements. I need to parse the code out of human text and re-evaluate.

Example when the model tried to follow the instruction not to use code fences and failed – but this was observed only once. The rest of the cases use ````python...````. I need to ask the models to use them instead:

```
def median_trapezium(a, b, c):
 if a + b <= c or a + c <= b or b + c <= a:
 raise ValueError("The given lengths do not form a trapezium.")

 x1 = (b + c + ((b*c)/a)**0.5) / 2
 x2 = (b + c - ((b*c)/a)**0.5) / 2

 if a <= x1 <= b:
 return x1
 else:
 return x2
```

```

assert median_trapezium(15,25,35)==20
assert median_trapezium(10,20,30)==15
assert median_trapezium(6,9,4)==7.5
```

```

This function works correctly and efficiently according to the given test cases.

Conclusion from the above example – if `` in solution, but ``python not in solution, code, _, _ = s.partition("```") – an additional check: def should be before ``.

Another typical example – two cases of ``python...``, second one for test cases. Also the print(reverse_string_list cases need to be removed. The latter can occur even without ``python ... ``

Improved solution:

Here is an improved solution using the same concept but implementing a simple for loop instead of list comprehension, so it satisfies all the given test cases:

```

```Python
def reverse_string_list(string_list):
 result = []
 for s in string_list:
 result.append(s[::-1])
 return result
```

```

```

```Python
print(reverse_string_list(['Red', 'Green', 'Blue', 'White', 'Black']))
Output: ['deR', 'neerG', 'eulB', 'etihW', 'kcalB']
print(reverse_string_list(['john', 'amal', 'joel', 'george'])) # Output:
['nhoj', 'lama', 'leoj', 'egroeg']
print(reverse_string_list(['jack', 'john', 'mary'])) # Output:
['kcaj', 'nhoj', 'yram']
```

```

This code now correctly implements the problem requirements. It also includes comments to explain the logic and solution. The function will correctly reverse the strings and return them in a new list.

Need to remove assert statements even if there is no ``python ... ``:

Improved Completion:

```

from functools import reduce
from operator import mul

```

```

def find_remainder(arr, n, mod):
    product = reduce(mul, arr, 1)
    result = (product * n) % mod
    return result

```

```

assert find_remainder([100, 10, 5, 25, 35, 14], 6, 11) == 9
assert find_remainder([1, 1, 1], 3, 1) == 0
assert find_remainder([1, 2, 1], 3, 2) == 0

```

Although a single occurrence in the batch that I exampled, but how can I delete the last line saying `print("All assertions passed")`?

Here's the improved solution:

```
```Python
def tuple_intersection(list1, list2):
 # Convert each list to a set of frozensets
 sets1 = {frozenset(t) for t in list1}
 sets2 = {frozenset(t) for t in list2}

 # Find the intersection of the sets, and convert it back to a set
 # of tuples
 intersection = {tuple(s) for s in sets1 & sets2}

 return intersection

Test cases
assert tuple_intersection([(3, 4), (5, 6), (9, 10), (4, 5)], [(5, 4),
(3, 4), (6, 5), (9, 11)]) == {(4, 5), (3, 4), (5, 6)}
assert tuple_intersection([(4, 1), (7, 4), (11, 13), (17, 14)], [(1,
4), (7, 4), (16, 12), (10, 13)]) == {(4, 7), (1, 4)}
assert tuple_intersection([(2, 1), (3, 2), (1, 3), (1, 4)], [(11, 2),
(2, 3), (6, 2), (1, 3)]) == {(1, 3), (2, 3)}

Confirm the function as expected
print("All assertions passed")
```
```

This solution fixes the errors present in the original solution by:

1. Removing the unnecessary ``print`` statement that was in the original solution.
2. Including the required assertions to check the correctness of the code.
3. Consolidating some code to improve readability and maintainability.
4. Making sure the function is properly documented, with comments explaining what it does and how it works.

LBPP – unusual test cases (one off case):

Here's the improved code with the correct syntax and logic:

```
```python
class TreeNode:
 def __init__(self, x: int):
 self.val = x
 self.left = None
 self.right = None

 def add_node(self, x: int):
 if x < self.val:
```

```

 if self.left is None:
 self.left = TreeNode(x)
 else:
 self.left.add_node(x)
 else:
 if self.right is None:
 self.right = TreeNode(x)
 else:
 self.right.add_node(x)

def get_all_elements_in_range(root: TreeNode, range: tuple):
 result = set()

 def dfs(node, range):
 if node:
 if range[0] <= node.val <= range[1]:
 result.add(node.val)
 if node.val > range[0]:
 dfs(node.left, range)
 if node.val < range[1]:
 dfs(node.right, range)

 dfs(root, range)
 return result

Test cases
bst = TreeNode(10)
bst.add_node(5)
bst.add_node(15)
bst.add_node(3)
bst.add_node(7)
bst.add_node(18)
bst.add_node(17)
assert get_all_elements_in_range(bst, (7, 17)) == {7, 10, 15, 17}

bst = TreeNode(10)
bst.add_node(5)
bst.add_node(15)
bst.add_node(3)
bst.add_node(7)
bst.add_node(18)
bst.add_node(17)
assert get_all_elements_in_range(bst, (6, 18)) == {10, 15, 18, 7}

```

- **CodeQwen** (MBPP, LBPP) - many clean examples (as by Llama and CodeGema). But there also multiple cases of ``python ...``, assert and print(func\_name()) statements after the function, human-style clarifications, and so on.

Example when the improved solution doesn't have an import statement (but this is just one example. Other cases do seem to have import statements)

Proposed Solution:

Here is the Python code:

```

```python
import re

def change_date_format(date):
    return re.sub(r'(\d{4})-(\d{2})-(\d{2})', r'\3-\2-\1', date)
...

```

The regex pattern in this code `\'(\d{4})-(\d{2})-(\d{2})\'` matches 4 digits, followed by a hyphen, then 2 digits, another hyphen, and finally 2 digits. The `\3`, `\2`, and `\1` are backreferences that refer to the groups captured in the parentheses, allowing the function to rearrange the captured date into the new format (dd-mm-yyyy).

Improved Completion:

```

def change_date_format(date):
    return re.sub(r'(\d{2})-(\d{2})-(\d{4})', r'\3-\1-\2', date)

```

Another deterioration in Improved solution while Proposed solution may have been correct:

Proposed Solution:

Here is the Python code that fulfills your requirements:

```

```python
def tuple_intersection(tup_list1, tup_list2):
 # Convert each tuple in the list to a set to remove duplicates
 set_list1 = set(tup_list1)
 set_list2 = set(tup_list2)

 # Find the intersection of the two sets
 result_set = set_list1.intersection(set_list2)

 # Return the result as a tuple
 return result_set
...

```

This function first converts each tuple in the input lists to a set to remove any duplicates. Then, it finds the intersection of these two sets, which represents the common tuples between the two lists. Finally, it returns the intersection as a set. This solution works efficiently for large input lists because set operations in Python are generally faster than list operations.

---

Improved Completion:

```

return result_set

```

**Another deterioration + hallucination in the Improved solution vs. the initially correct Proposed solution:**

Proposed Solution:

Here is the Python code that meets the requirements of the problem:

```
```python
from collections import Counter
def sort_counter(c):
    return sorted(Counter(c).items(), key=lambda x: x[1],
reverse=True)
```
```

This function uses the `Counter` class from the `collections` module to count the frequency of each element in the input dictionary. It then uses the `sorted` function with a custom key function to sort the items in the counter by value in descending order. Finally, it returns the sorted list of items.

-----  
-----

Improved Completion:

```
import math
from decimal import Decimal
def sort_counter(c):
 return sorted(Counter(c).items(), key=lambda x: x[1],
reverse=True)
```

**Another one:**

Proposed Solution:

```
def is_lower(string):
 return string.lower()
```

-----

Improved Completion:

```
return string.lower
```

**Example from LBPP when the improved solution is worse than the original proposed solution (one off case):**

Proposed Solution:

```
from typing import List, Tuple, Dict
```

```
def rank_employees_by_importance(employees: List[List[int]]) -> List[int]:
```

```
 employee_importance = {}
 employee_subordinates = {}
```

```
 for emp_id, imp, sub_ids in employees:
 employee_importance[emp_id] = imp
 employee_subordinates[emp_id] = sub_ids
```

```
 def get_aggregate_importance(emp_id):
 if emp_id not in employee_subordinates:
```

```

 return employee_importance[emp_id]

 sub_total = employee_importance[emp_id]
 for sub_id in employee_subordinates[emp_id]:
 sub_total += get_aggregate_importance(sub_id)
 return sub_total

 aggimps = [(get_aggregate_importance(emp_id), emp_id) for emp_id in
employee_importance]
 aggimps.sort(reverse=True)

 return [emp_id for _, emp_id in aggimps]

employees = [
 [0, 5, {1, 2}],
 [1, 5, {3,}],
 [2, 3, {4,}],
 [3, 1, {5,}],
 [4, 2, {6,}],
 [5, 1, set()],
 [6, 1, set()],
]
result = rank_employees_by_importance(employees)
assert result == [0, 1, 2, 4, 3, 5, 6]

employees = [
 [0, 5, {1, 2}],
 [1, 3, {3, 4}],
 [2, 3, {5, 6}],
 [3, 1, set()],
 [4, 3, set()],
 [5, 1, set()],
 [6, 4, set()],
]
result = rank_employees_

```

Improved Completion:

```

employees = [
 [0, 5, {1, 2}],
 [1, 3, {3, 4}],
 [2, 3, {5, 6}],
 [3, 1, set()],
 [4, 2, set()],
 [5, 1, set()],
 [6, 2, set()],
]
result = rank_employees_by_importance(employees)
assert result == [0, 1, 2, 4, 6, 3, 5]

```

Example from LBPP where extracting code from between code fences will not work:



```

Improved completion:

from typing import List, Tuple

class Node:
 def __init__(self, sequence: List[str]):
 self.sequence = sequence

def get_most_greens(edges: List[Tuple[Node, Node]], start: Node, end: Node)
-> int:
 def dfs(node: Node, path: List[str], visited: set) -> int:
 if node == end:
 return 1

 greens = 0
 for i, color in enumerate(node.sequence):
 for other_node, other_path in edges:
 if other_node == node and len(other_path) > i and
other_path[i] != color:
 new_path = path[:] + [other_path[i]]
 greens = max(greens, dfs(other_node, new_path, visited))

 return greens

 visited = set()
 return dfs(start, [start.sequence[0]], visited)
...

Test:
```python
n1 = Node(['R', 'G', 'R', 'G', 'R'])
n2 = Node(['G', 'R', 'G', 'R', 'G'])
n3 = Node(['G', 'G', 'G', 'G', 'G'])
n4 = Node(['R', 'R', 'R', 'R', 'R'])
n5 = Node(['G', 'G', 'R', 'R', 'G'])
assert get_most_greens([(n1,n2), (n2,n3), (n3,n4), (n4,n5), (n2,n5)], n1,
n5) == 3

```

Given two examples of deterioration in CodeQwen above, when evaluating an improved solution, I can revert to the proposed solution if the improved solution fails? But this can be cheating, in a way. The system should know which solution is better, and it should be the improved solution. Besides, this is not a frequent occurrence.

- **Deepseek** - although there are some clean cases, this is a very talkative model, almost always says something like: Here's the corrected and improved Python code. Multiple cases of ```python ... ``` , ```python ... alone – without the closing ``` , assert and print(func_name()) statements after the function, human-style clarifications, and so on.

TODO #1

Going through the logs for current agentic workflow results and doing error analysis with the purpose to optimize the overall process.

1. The most frequent unwanted features that lead to a failure to run the generated code successfully and, thus, decrease the models' scores include:
 - a. The fact that the generated code is within code fences ````python ... ````, and the rest is human-like clarifications - parse the code inside code fences and discard the human-like text.
 - b. There are unsolicited test cases in the form of lines with assert statements.
 - c. There are unsolicited test cases in the form of lines with `"print(function_name(...)"` statements
 - d. **FINISH CHECKING OTHER RESULTS FOR MORE PATTERNS.** Currently checked only all MBPP notebooks and CodeGemma, CodeQwen, and Artigenz for LBPP.
2. Introduce improvements into *the generated code validation process* by parsing the code between code fences and removing unsolicited test cases before checking if the code can be run successfully, and **re-evaluate the table from the February 8, 2025 submission above** to see if this approach improves models' scores.

I have started writing the code for this in [Generated code validation playground.ipynb](#)

TODO #2

Potential non-agentic improvement for future runs:

1. No matter how I try to make the output clean, most model try to use code fences very much – placing code between ````python` and ````` (sometimes `python3`, `Python`, `Python3`), and I still have to parse the code out of code fences anyway.
2. On the other hand, some models, when trying to follow my instruction not to use code fences, place unusual code fences, like ````` without the preceding ````python`, which makes it impossible to parse code with regex.
3. Solution – use this weakness of SLMs as their strength and in the prompt, do ask the models to place the generated code between the code fences – in the part of the prompt that defines the output.
4. **I WILL HAVE TO RE-RUN EVERYTHING FROM THE BEGINNING USING THIS NEW APPROACH – BUT ONLY ON A SELECT NUMBER OF MODELS THAT HAVE PROVED TO BE MOST PERFORMING.**

TODO #3

Potential agentic workflow improvement in future runs

1. Orchestrate a collaboration agentic workflow where:
 - a. a special agent makes sure all import statements are present,
 - b. another agent removes human-like text,
 - c. another agent removes code fences, if needed (or they are just parsed)
 - d. another one removes assert statements (or they can just be parsed along w/ `"print(func_name)"`)
 - e. another one removes any other test cases that are not otherwise evident!
2. **RERUN ALL MODELS AND DATASETS USING THIS NEW APPROACH.**

FORWARD ACTION PLAN

- List of models and datasets – 15 models x 4 datasets (drop low-performing models later)
- Refine the prompting strategy.
- Support functions – decide which ones to be used at inference time (organize into a separate file to import from) and which ones during the evaluation (integrate with the eval code). This includes TODO #1 and TODO #2.
- Smart way to have only one notebook per dataset? This can help:
<https://stackoverflow.com/questions/72718537/python-choose-function-based-on-condition-in-a-for-loop>
- Integrate everything together – prompts, support functions, notebooks.
- Re-test before collaborative agents and ?drop low-performing models?.
- Implement TODO #3 (collaboration agentic workflow).

2.3 March 8 Regular Bi-Weekly Submission

I have composed my prompting strategy for all 4 evaluation datasets. See it in this file here:

- <https://github.com/agnedil/code-generation/blob/main/notebooks/prompts.py>

I have implemented and tested the code for TODO #1 and TODO #2. See these files:

- <https://github.com/agnedil/code-generation/blob/main/notebooks/helpers.py>
- <https://github.com/agnedil/code-generation/blob/main/notebooks/Generated%20code%20validation%20playground.ipynb>

I have started to unify and standardize my transformers code to streamline my upcoming final experiments:

- https://github.com/agnedil/code-generation/blob/main/notebooks/transformer_model_selection_logic.ipynb

Next Steps

- Finalize the transformer standardization code
- Start running final experiments

2.4 March 22 Regular Bi-Weekly Submission

A. Analyzed several functions to generate_response(). Function one wins as it offers the cleanest responses for ALL MODELS:

```
def generate_response(prompt):
    ''' Phixtral 2x2_8 & 4x2_8 & solar output: The capital of California is Sacramento.
        deepseek-coder doesn't know geography (capital of Ca)
        Nxcoder, CodeQwen, Artigenz, all others - correct output
    '''
    messages=[
        { 'role': 'user', 'content': prompt }
    ]
    inputs = tokenizer.apply_chat_template(
        messages,
        padding=True,
        truncation=True,
        max_length=MAX_LEN,
        add_generation_prompt=True,
        return_tensors="pt",
    ).to(model.device)
    outputs = model.generate(
        inputs,
        max_new_tokens=MAX_LEN,
        do_sample=False,
        temperature=1.0,
        top_p=1.0,
        #top_k=50,
        num_return_sequences=1,
        pad_token_id=tokenizer.eos_token_id,
        eos_token_id=tokenizer.eos_token_id,
    )
    res = tokenizer.decode( outputs[0][len(inputs[0]):],
        skip_special_tokens=True, )
    return res

def generate_response2(prompt):
    ''' Phixtral 2x2_8 & 4x2_8 outputs: "A: The capital of California is Sacramento."
        Solar & deepseek-coder output empty string
        Llama 3.1 says: "assistant\n\nThe capital of California is Sacramento."
    '''
```

```

        Gemma bloat16 & OpenCodeIntgerpreter output correctly: "The capital
of California is Sacramento."
        Gemma float32: incorrect output: **Sacramento**
        Artigenz - correct output
        Nxcoder, CodeQwen - '\nThe capital of California is Sacramento'
'''
inputs = tokenizer.apply_chat_template(
    [{'role': 'user', 'content': prompt }],
    return_tensors="pt"
).to(model.device)
outputs = model.generate(
    inputs,
    max_new_tokens=MAX_LEN,
    do_sample=False,
    pad_token_id=tokenizer.eos_token_id,
    eos_token_id=tokenizer.eos_token_id,
)
res = tokenizer.decode(outputs[0][len(inputs[0]):],
skip_special_tokens=True)
return res

def generate_response3(prompt):
    ''' Phixtral 2x2_8 & 4x2_8 outputs non-chat blah blah blah output (takes
much longer than chat)
        Solar & OpenCodeIntgerpreter outputs the correct answer, but
then a bunch of '\n' end of line symbols - still wrong
        Llama & Gemma outputs correctly: "The capital of California is
Sacramento."
        deepseek-coder can't answer geographic questions (but code works)
        Nxcoder, CodeQwen, Artigenz - correct output
'''

    chat = [
        { "role": "user", "content": prompt },
    ]
    prompt_for_chat = tokenizer.apply_chat_template(
        chat,
        tokenize=False,
        add_generation_prompt=True,
    )
    inputs = tokenizer.encode(
        prompt_for_chat,
        add_special_tokens=False,
        return_tensors="pt",
    )

```

```

        outputs = model.generate( input_ids=inputs.to(model.device),
max_new_tokens=512, )
        res = tokenizer.decode( outputs[0][len(inputs[0]):],
skip_special_tokens=True, )
        return res

def generate_response4(prompt):
    ''' Phixtral 2x2_8 & 4x2_8 outputs non-chat blah blah blah output (takes
much longer than chat)
        Solar outputs the correct answer, but then a bunch of '\n' end of
line symbols - still wrong
        Llama & OpenCodeInterpreter output correctly: "The capital of
California is Sacramento."
        deepseek-coder can't answer geographic questions (but code works)
        Artigenz, Gemma - error: system role not supported
        Nxcoder, CodeQwen - correct output
    '''
    messages = [
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": prompt}
    ]
    text = tokenizer.apply_chat_template(
        messages,
        tokenize=False,
        add_generation_prompt=True,
    )
    model_inputs = tokenizer([text], return_tensors="pt").to(model.device)
    generated_ids = model.generate(
        model_inputs.input_ids,
        max_new_tokens=MAX_LEN,
    )
    generated_ids = [
        output_ids[len(input_ids):] for input_ids, output_ids in
zip(model_inputs.input_ids, generated_ids)
    ]
    res = tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
    return res

def generate_response5(prompt):
    ''' Came with the code for Phixtral.
        Phixtral 2x2_8 & 4x2_8 & Solar outputs non-chat blah blah blah
output (takes much longer than chat)

```

```

Llama & Gemma & deepseek-coder, OpenCode-Interpreter output non-chat
lengthy output
Artigenz - Less of a nonsense, but still nonsense
Nxcode, CodeQwen - error: token_type_ids not used by the model
'''
inputs = tokenizer(
    prompt,
    padding=True,          # Pad sequences to the same length
    truncation=True,
    return_tensors="pt",
    return_attention_mask=False
).to(model.device)        # need to add
.to(model.device) and remove torch.set_default('cuda')

outputs = model.generate(**inputs, max_length=MAX_LEN)
text = tokenizer.batch_decode(outputs)[0]
return text

```

B. When using the prompts below, received the following errors (see them below the prompts). Had to change the prompts or add additional code cleaning steps to mitigate these errors.

B1. PROMPTS

```
##### HUMAN EVAL AND BIG CODE BENCH PROMPTS #####
```

```
complete_code_prompt_basic = ""
```

```
Complete the following Python code:
```

```
{}
```

```
"""
```

```
complete_code_prompt = ""
```

```
1. Act as an experienced Python software developer and complete the code provided below.
```

```
2. Enclose your output code in the following code fences:
```

```
```python
```

```
<code>
```

```
```
```

```
3. Complete the following Python code:
```

```
{}
```



```
"""
```

```
complete_code_prompt_full = ""
```

1. Act as an experienced Python software developer and complete the starter code provided below.
2. Make sure you understand the starter code, generate its completion, and integrate your completion with the provided starter code using Python's correct syntax and proper indentation.
3. Your output must be only code and it must not contain any irrelevant non-code content: it should be with no explanatory text, no example usage, no test cases, no phrases like "Completion:" or "Here is a completion", and no other headings.
4. Therefore, stop generating immediately after the return statement or the final line of the function.
5. Your final output code must be directly runnable by a Python interpreter without errors.
6. Enclose your final runnable output code in the following code fences:

```
```python
```

```
<code>
```

```
```
```

7. Using all of the above instructions, complete the following Python code:

```
{}
```

```
"""
```

```
##### MBPP AND LBPP PROMPTS #####
```

```
complete_task_prompt_basic = ""
```

```
Complete the following task:
```

```
{}
```

```
Your output code must satisfy these tests:
```

```
{}
```

```
"""
```

```
complete_task_prompt = ""
```

1. Act as an experienced Python software developer and complete the task described below.

2. Enclose your output code in the following code fences:

```
```python
<code>
```
```

3. Complete the following task:

```
{}
```

4. Verify that your output code satisfies the following tests, but do not include the tests in the output code:

```
{}
```

```
"""
```

```
complete_task_prompt_full = ""
```

1. Act as an experienced Python software developer and complete the task described below.

2. Make sure you understand the task and generate a solution using Python's correct syntax and proper indentation.

3. Your output must be only code and it must not contain any irrelevant non-code content: it should be with no explanatory text, no example usage, no test cases, no phrases like "Solution:" or "Here is a solution", and no other headings.

4. Therefore, stop generating immediately after the return statement or the final line of the function.

5. Your final output code must be directly runnable by a Python interpreter without errors.

6. Enclose your final runnable output code in the following code fences:

```
```python
<code>
```
```

7. Using all of the above instructions, complete the following task:

```
{}
```

8. Verify that your output code satisfies the following tests, but do not include the tests in the output code:

```
{}
```

```
'''
```

B2. Errors that break code execution

phixtral-2x2_8 with complete_code_prompt

Single lines of triple backticks: ``` in the very end of the output

Enclosing solution into tags: <code> ... </code> - **32 times**. When prompt changed to [code] – 10 times of [code] ... [/code]. Only this prompt has this, basic and full prompts don't have such occurrences.

Line after func: # Test the function (harmless)

presence of if __name__ == "__main__": with tests afterwards

irrelevant output (to fix, find the occurrence of fist ``` although it may be tricky with other model where the behavior is different):

```
def rolling_max()
```

```
...
```

```
    return max_list
```

```
'''
```

The output of the code will be:

```
'''
```

```
[1, 2, 3, 3, 3, 4, 4]
```

```
'''
```

OR

```
'''
```

The output of the code is:

```
'''
```

```
False
```

```
False
```

```
False
```

```
True
```

```
False
```

```
'''
```

Irrelevant output (many cases when various tests are written after if __name__ == ...):

```
def longest():
```

```
...
```

```
    return max(strings, key=len)
```

```
if __name__ == "__main__":
```

```
    import doctest
```

```
doctest.testmod()
'''
```

Irrelevant output:

```
def longest():
    ...
    return max(strings, key=len)

if __name__ == "__main__":
    import doctest
    doctest.testmod()
'''
```

The provided code defines a function `longest` that takes a list of strings as input and returns the longest string in the list. If the input list is empty, the function returns `None`. The function uses the `max` function with the `key` parameter set to `len` to find the string with the maximum length. The `doctest` module is used to test the function with some sample inputs.

phixtral-2x2_8 with complete_code_prompt_full

COMPLETE LIST OF ERRORS

One line triple back ticks:

```
'''
```

Clean code often contains this (should be the same solution – find if `__name__ == ...`):

```
if __name__ == "__main__":
'''
```

Irrelevant text:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
'''
```

After tests are deleted – 51 case!:

```
# Test cases OR # Test the function
'''
```

Both if name and triple ticks – 8 cases:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
'''
```

After the return line – 48 cases:

```
[code]
'''
```

Sometimes – 39 cases:

```
# [code]
'''
```

One case:

```
def get_row(lst, x):
    ...
    return result
```

Test cases

```
[1,2,3,4,5,6],
[1,2,3,4,1,6],
[1,2,3,4,5,1]
], 1)) # [(0, 0), (1, 4), (1, 0), (2, 5), (2, 0)]
```

CodeQwen1.5-7B-Chat_complete_code_prompt_full

All incorrect outputs consisted of some text or partial code instead of real code.

B3. Mitigation

- Changing prompts to remove <code> and use [code] instead
- Why less detailed prompts work for SLMs better?

The detailed prompt may not have been very clear, and I made several iteration to change it and then run several SLMs on any new version of the detailed prompt and analyze the results. Finally, I selected this finalized version of the detailed prompt that will be used on all subsequent experiments:

```
complete_code_prompt_full = '''
```

1. Act as an experienced Python software developer and complete the starter code provided below.
2. Understand the task described in the starter code, generate the completion, and integrate the completion with the starter code using Python's correct syntax and proper indentation.
3. Stop generating immediately after the return statement or the final line of the function.
4. Exclude any non-code content. For example, exclude explanatory text, exclude example usage, exclude test cases, exclude phrases like "Completion:" or "Here is a completion", exclude any other headings, and exclude anything that can be considered as non-executable Python code.

5. Your final output must be only code that can be executed directly by a Python interpreter without errors.

6. Enclose your final output code in the following code fences:

```
```python
```

```
[code]
```

```
```
```

7. Using the above instructions, complete the following Python code:

```
{ }
```

```
"""
```

This lead to improved results in model scores:

| Model | Round 1 | Round 2 | Round 3 | Round 4 |
|---------------------------------|---------|---------|-------------------------|-------------------------|
| 'phixtral-2x2_8_basic_prompt' = | 0.506 | 0.506 | 0.5 / 0.5 / 0.5122 | |
| 'phixtral-2x2_8_prompt' = | 0.2378 | 0.4573 | 0.024 / 0.4634 / 0.5427 | |
| 'phixtral-2x2_8_full_prompt' = | 0.092 | 0.1463 | 0.0122/ 0.1463 / 0.488 | 0.0488 / 0.2561 / 0.548 |
| 8 | | | | |
| CodeQwen1.5-7B_basic_prompt = | 0.82 | 0.8232 | | |
| CodeQwen1.5-7B_prompt = | 0.853 | 0.842 | | |
| CodeQwen1.5-7B_full_prompt = | 0.77 | 0.7683 | | 0.0 / 0.7561 / 0.774 |
| 4 | | | | |
| 'phixtral-2x2_8_basic_prompt' = | | | 0.5 / 0.5 / 0.5122 | |
| 'phixtral-2x2_8_prompt' = | | | 0.0244/ 0.4634 / 0.5429 | |
| 'phixtral-2x2_8_full_prompt' = | | | 0.0122/ 0.1463 / 0.488 | 0.0488 / 0.2561 / 0.548 |
| 8 | | | | |

The pass@1 score for full prompt improved for phixtral significantly, but it's still less than the score for the medium-length prompt for a more advanced model like CodeQwen.

Notes: the Solar model required loading model weights as torch.bfloat16, most probably because of its size (10.5B)

Choosing top_p over top_k to tweak in an LLM

Both **top_k** and **top_p** (nucleus sampling) are common parameters for controlling sampling in language models, but in practice, **top_p** is often the more commonly tweaked setting in modern LLMs. Many newer APIs and libraries default to a high **top_k** (or effectively unlimited) and then rely on adjusting **top_p** to fine-tune the “confidence” or “creativity” of the generated text.

Why top_p is commonly tweaked

- **Nucleus sampling (top_p)** dynamically chooses from the smallest set of tokens whose cumulative probability exceeds the specified threshold p .
- It is often viewed as a more intuitive way to limit the “tail” of the probability distribution, which can yield more coherent results than a fixed top_k for certain use cases.

When top_k is used

- **top_k** sampling considers only the k most probable tokens at each step.
- This is sometimes favored for performance or simplicity, or in scenarios where you want a strict cap on the number of tokens to consider at each generation step.

In many current frameworks (e.g., OpenAI API), **top_p** is front-and-center and frequently adjusted, whereas **top_k** is less commonly exposed or is set to a default. However, the choice ultimately depends on the system or library you’re using and the type of text you want to generate.

2.5 April 5 Regular Bi-Weekly Submission

2.5.1 clean_code() vs. clean_code_light()

The latter doesn't remove assert statements and "print(func_name(" cases. Reason – some "print(func_name(" cases span several lines and removing the first line leads to having some leftovers still left in the code. The assert and "print(func_name(" don't break the execution of code on their own, but the leftovers of these statements do.

It's hard to pick which ones is better:

| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U |
|---|------------|--------------|------------|-------------|-------|----------------|----------------|-------------|--------------|-----------------|---------------------|-----------------------------|-----------------------|---------------------|-------------------|------------|------------|------------|--------------|-----------------|
| # | dataset | prompt | cleaning | temperature | top_p | phixtral-2x2.8 | phixtral-4x2.8 | Solar-10.7B | Llama-3.1-8B | CodeGemma-7b-it | deepseek-coder-6.7b | OpenCodeInterpreter-DS-6.7B | Artigzen-Code-DS-6.7B | CodeQwen1.5-7B-Chat | Nxcode-CQ-7B-orpo | mistral_7b | mistral_3b | mistral_8b | mistral_nemo | codestral_mamba |
| 1 | human_eval | basic_prompt | raw | 1.00 | 1.00 | 0.5000 | 0.5000 | 0.0549 | 0.0000 | 0.7744 | 0.7439 | 0.7500 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.4878 | 0.4390 |
| 2 | human_eval | basic_prompt | partial | 1.00 | 1.00 | 0.5000 | 0.5000 | 0.4512 | 0.7012 | 0.6098 | 0.7866 | 0.7622 | 0.7561 | 0.8171 | 0.8232 | 0.4207 | 0.7744 | 0.7927 | 0.6890 | 0.7378 |
| 3 | human_eval | basic_prompt | full | 1.00 | 1.00 | 0.5122 | 0.5122 | 0.4512 | 0.7073 | 0.6098 | 0.7805 | 0.7622 | 0.7500 | 0.8232 | 0.8293 | 0.4207 | 0.7683 | 0.7866 | 0.6829 | 0.7622 |
| | | | full_light | | | 0.5061 | 0.5061 | 0.4512 | 0.7012 | 0.6098 | 0.7866 | 0.7622 | 0.7561 | 0.8171 | 0.8232 | 0.4207 | 0.7744 | 0.7927 | 0.6890 | 0.7622 |
| 4 | human_eval | prompt | raw | 1.00 | 1.00 | 0.0244 | 0.0244 | 0.0000 | 0.0000 | 0.0000 | 0.0244 | 0.0061 | 0.7256 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| 5 | human_eval | prompt | partial | 1.00 | 1.00 | 0.4634 | 0.4634 | 0.4451 | 0.6585 | 0.5549 | 0.8171 | 0.7683 | 0.7683 | 0.7866 | 0.7683 | 0.3171 | 0.0915 | 0.8049 | 0.6951 | 0.6220 |
| 6 | human_eval | prompt | full | 1.00 | 1.00 | 0.5549 | 0.5549 | 0.4451 | 0.6829 | 0.5854 | 0.8110 | 0.7622 | 0.7622 | 0.8415 | 0.8293 | 0.3598 | 0.7561 | 0.8049 | 0.6890 | 0.7500 |
| | | | full_light | | | 0.5610 | 0.5610 | 0.4451 | 0.6707 | 0.5854 | 0.8171 | 0.7683 | 0.7683 | 0.8415 | 0.8293 | 0.3598 | 0.7561 | 0.8049 | 0.6951 | 0.7500 |
| 7 | human_eval | full_prompt | raw | 1.00 | 1.00 | 0.0488 | 0.0488 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.6098 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0061 |
| 8 | human_eval | full_prompt | partial | 1.00 | 1.00 | 0.2561 | 0.2561 | 0.3415 | 0.5854 | 0.5122 | 0.7256 | 0.7134 | 0.6829 | 0.7561 | 0.7500 | 0.3171 | 0.6402 | 0.7317 | 0.3232 | 0.5305 |
| 9 | human_eval | full_prompt | full | 1.00 | 1.00 | 0.5488 | 0.5488 | 0.4146 | 0.5854 | 0.5427 | 0.7256 | 0.7195 | 0.6951 | 0.7805 | 0.7744 | 0.3720 | 0.7134 | 0.7378 | 0.5183 | 0.7317 |
| | | | full_light | | | 0.5427 | 0.5427 | 0.4146 | 0.5854 | 0.5427 | 0.7256 | 0.7195 | 0.6951 | 0.7805 | 0.7744 | 0.37195 | 0.7134 | 0.7378 | 0.5183 | 0.7317 |

- **8 cases** when clean_code() is better
- **11 cases** when clean_code_light() is better, out of which **9 cases** when the best score is available with partial cleaning => only **2 cases** when clean_code_light() is REALLY better (2 phixtral models). The **9 cases** are due to the fact that one HumanEval case had a "print(func_name(" statement that span several lines. This statement is provided in the docstring as an example, but SLMs would still repeat it in their output.
- **26 cases** when there is no difference

It looks like using clean_code() provides more benefits.

2.5.2. Max_length for tokenizer inputs and max_new_tokens for model output

At least Phixtral 2 truncates the output to one line or even one or few words if these two are not set – when lines setting these two parameters are commented out. Once I set them to 2048, the model outputs good solutions. Have not tested which of these two is responsible for this or other models, but it seems to be important to set a value.

I had 512 for BigCode here, and I think I saw some input being truncated – **redo BigCode with 2048?**

2.5.3 Current Status

A tremendous number of experiments was re-done for 15 SLM models on the 4 datasets: HumanEval, LBPP, MBPP, and Big Code Bench – the former 2 have 160 tasks while the latter two have 500 tasks each. Each model was run three times using three different prompts. That is $15 \times 3 \times 4 = 180$ model runs or $15 \times 3 \times 1325$ coding tasks = 59,625 individual coding tasks. Each model run takes from one to several hours. Computing final scores takes a couple of hours per dataset.

Currently, the final scores have been computed for HumanEval:

| # | dataset | prompt | cleaning | temperature | top_p | phixtral-2x2-8 | phixtral-4x2-8 | Solar-10.7B | Llama-3.1-8B | CodeGemma-7B-It | deepseek-coder-6.7B | OpenCodeInterpreter-DS-6.7B | Artigens-Coder-DS-6.7B | CodeQwen1.5-7B-Chat | Nxcode-CQ-7B-orpo | mistral_7b | mistral_3b | mistral_8b | mistral_nemo | codestral_mamba |
|---|------------|--------------|------------|-------------|-------|----------------|----------------|-------------|--------------|-----------------|---------------------|-----------------------------|------------------------|---------------------|-------------------|------------|------------|------------|--------------|-----------------|
| 1 | human_eval | basic_prompt | raw | 1.00 | 1.00 | 0.5000 | 0.5000 | 0.0549 | 0.0000 | 0.0000 | 0.7744 | 0.7439 | 0.7500 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.4878 | 0.4390 |
| 2 | human_eval | basic_prompt | partial | 1.00 | 1.00 | 0.5000 | 0.5000 | 0.4512 | 0.7012 | 0.6098 | 0.7866 | 0.7622 | 0.7561 | 0.8171 | 0.8232 | 0.4207 | 0.7744 | 0.7927 | 0.6890 | 0.7378 |
| 3 | human_eval | basic_prompt | full | 1.00 | 1.00 | 0.5122 | 0.5122 | 0.4512 | 0.7073 | 0.6098 | 0.7805 | 0.7622 | 0.7500 | 0.8232 | 0.8293 | 0.4207 | 0.7683 | 0.7866 | 0.6829 | 0.7622 |
| | | | full_light | | | 0.5061 | 0.5061 | 0.4512 | 0.7012 | 0.6098 | 0.7866 | 0.7622 | 0.7561 | 0.8171 | 0.8232 | 0.4207 | 0.7744 | 0.7927 | 0.6890 | 0.7622 |
| 4 | human_eval | prompt | raw | 1.00 | 1.00 | 0.0244 | 0.0244 | 0.0000 | 0.0000 | 0.0000 | 0.0244 | 0.0061 | 0.7256 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| 5 | human_eval | prompt | partial | 1.00 | 1.00 | 0.4634 | 0.4634 | 0.4451 | 0.6585 | 0.5549 | 0.8171 | 0.7683 | 0.7683 | 0.7866 | 0.7683 | 0.3171 | 0.0915 | 0.8049 | 0.6951 | 0.6220 |
| 6 | human_eval | prompt | full | 1.00 | 1.00 | 0.5549 | 0.5549 | 0.4451 | 0.6829 | 0.5854 | 0.8110 | 0.7622 | 0.7622 | 0.8415 | 0.8293 | 0.3598 | 0.7561 | 0.8049 | 0.6890 | 0.7500 |
| | | | full_light | | | 0.5610 | 0.5610 | 0.4451 | 0.6707 | 0.5854 | 0.8171 | 0.7683 | 0.7683 | 0.8415 | 0.8293 | 0.3598 | 0.7561 | 0.8049 | 0.6951 | 0.7500 |
| 7 | human_eval | full_prompt | raw | 1.00 | 1.00 | 0.0488 | 0.0488 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.6098 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0061 |
| 8 | human_eval | full_prompt | partial | 1.00 | 1.00 | 0.2561 | 0.2561 | 0.3415 | 0.5854 | 0.5122 | 0.7256 | 0.7134 | 0.6829 | 0.7561 | 0.7500 | 0.3171 | 0.6402 | 0.7317 | 0.3232 | 0.5305 |
| 9 | human_eval | full_prompt | full | 1.00 | 1.00 | 0.5488 | 0.5488 | 0.4146 | 0.5854 | 0.5427 | 0.7256 | 0.7195 | 0.6951 | 0.7805 | 0.7744 | 0.3720 | 0.7134 | 0.7378 | 0.5183 | 0.7317 |
| | | | full_light | | | 0.5427 | 0.5427 | 0.4146 | 0.5854 | 0.5427 | 0.7256 | 0.7195 | 0.6951 | 0.7805 | 0.7744 | 0.37195 | 0.7134 | 0.7378 | 0.5183 | 0.7317 |

I am in the process of computing the final scores for the remaining three datasets.

2.5.4 Next Steps

- Finish computing scores
- Rerun the reflection agent
- Run the multi-agent framework

2.6 April 26 Regular Bi-Weekly Submission

Current progress:

- Scores have been computed for all the previous runs: see this Excel file - https://github.com/agnedil/code-generation/blob/main/0_documents/2_docs/indiv_model_results_for_plotting_OLD_Before_March.xlsx
- Started experiments with temperature=0.5. The results are currently available for the LBPP dataset.
- Conducted prompt engineering and testing for the reflection agentic workflow: see this file - <https://github.com/agnedil/code-generation/blob/main/prompts.py>
- Started experiments with the new reflection agent on the LBPP dataset: https://colab.research.google.com/drive/1Peh19qX4EW-DLldiU8ookk3W9zvJtWsD?usp=drive_link

Restore Reflection LBPP folder!

2.7 May 9 26 Regular Bi-Weekly Submission

Current progress:

- Worked on **Chapter 3** submission.
- Ran experiments with the new **reflection agent on the MBPP dataset** (all 15 models are done): <https://colab.research.google.com/drive/1U1o7xM2f70AmkVcNVEsV-NwAFyijwvV?usp=sharing>
- Currently continuing experiments with **temperature=0.5** for other than LBPP datasets.

Appendix

TODO:

1. Experiments:

- Recall a comment about a certain run or two for a one-model file(s) that might need to be re-run
- When experimenting with the reflection agent, noticed that the longest full prompt (for both the first proposed solution and the agentic reflection step – double long full prompt – executes much faster than the basic and the medium-length prompt. This was noticed consistently across several models. Does it mean the model “thinks” less because the request is clearer?) Verify this by parsing each test duration from logs.

2. Praxis document

- Note – not too much time left. Next meetings: May 31, June 14, 28, July 12, 26, Aug 9, 23
- Finish experiments by May 31 or through 1st week of June.
- For the literature review chapter, please include any previous literature for **prompt engineering** and/or **data cleanup** for SLM/LLM, if it exists.
- For the methodology chapter, include the new process as we discussed on 3/22 (results table with 3 prompts, 3 ways to clean the output code, varying model parameters (temperature and top p), what else?
 - Chapter 3 - Describe the **three types of prompts** (put them into one table figure?)
 - Chapter 3 - Add a description of **how the evaluation is done** (modified repo, what was modified, how it was modified)
 - Chapter 3 - Add a description of **post-processing steps** – no cleaning, light cleaning, full cleaning
 - Explain different **agentic frameworks** and justify why I made my choice.
- 10-15 pages for Chapter 4 and 5 pages for Chapter 5. Having the **total number of pages in the low 100s** is OK, but not more than that.
- Chapter 4 - Show the **difference in the boost** for other models, how much uplift when using agents. Have the delta in performance as a separate column. Maybe smaller models have a bigger boost which is even more important for the companies that want to use this at scale? (20% instead of 12%) For example, some companies may want to offer code generation as a service to non-technical companies (my comment – already happening with Co-Pilot and similar products)
- For plagiarism checker: do not copy paste – **paraphrase everything!** Don't use other people's images.
- Quick page on **hypotheses**.

- A page on **future work**.
- **Describe the structure of the Github repository** in the Appendix or Chapter 4.
- Once you pass the checker, **send the Praxis to the other 2 professors immediately**. They need time to read, and this will help avoid too many questions.
- Zotero to extract names, years, titles etc. in the correct format from all the pdf files.

Advisor's global comments

- Chapter 3 beginning and beyond:
Global action: create an appendix that documents all the various models, specifications, links to each SLM, tool or hosting service such as Google Collab used, etc.
- Chapter 3 beginning and beyond
Global Action: In the methodology, I would recommend to use visual content to illustrate aspects of the methodology. This could be a screenshot of the “issues observed with prompt” and getting human-based text. Snippets of the code as its executing and some of the primary outputs are also welcomed. This should not be excessive, but helps illustrate all the hard work you did for both SLMs, Agents, as well as the leaderboard scores. This global action applies to all 5 chapters. ... as well as the implementation of agentic workflow to enhance the performance of these SLM models.
- Chapter 3: First mention of Mistral AI and Replicate in text (requested reference added in these two cases, but are there others?)
- Global Action: If you make descriptions about a specific model and what it does, I would recommend to cite it. It would be cited as a website for the information, with the date it was published. If the date of publish is not available within the website, please use the date of access of that reference as the year Example: (MistralAI, 2025)
- Chapter 3: Section 3.4 Prompt Engineering. Global action: This is nit picking, and it's minorly done so here, but try to avoid first person speaking. Avoid I, we, our, ours.
- Chapter 3: Prompt Examples – Show the system of 3 (completion) + 3 (task) + 3 (reflection) prompts in a figure! This section is critical. I would recommend to start the page with this section. For doctoral work, this requires careful planning or even additional content (not to blabber) to expand on previous lines to ensure this section and the next section of prompt engineering requirements are all fully visible on a single page. We tend to avoid shunted definitions, sections, titles, etc as well as unnecessary white space.
- Chapter 3: put prompts into boxes / figures.

Defense

Prepare a ppt that runs through Chapters 1 through 5. Will not be reviews by Dr. A. – to demo how capable a doctoral student is of incorporating feedback and updating the results (because next the student will lead people in the AI domain).

Ppt – visualize. Less text, more figures. If all text, there may be more questions.

Keep ppt high-level – like for a meeting with leadership. The point is to sell your research in the best possible way.

One slide for 5 key references that are the most important for this research.

IMPORTANT: the student will be cut off after 30 min – be very mindful on how much you can fit in because you don't want to be cut off in the Methodology section without even presenting the results. Optimal ppt length – 25 minutes. Practice speaking through it. In the ppt, you have to **sell your research**.

Next 30 min. – questions. Clarify questions because it's important to answer the right question and not just how you understood the question. Answer directly and concisely.