Evaluation of Large Language Models Trained on Code

Srinivasan Viswanathan

Sbv5221@psu.edu

Overview

- Codex – GPT-3 based language model
  - Scalable sequence prediction
  - Writes python code given docstrings.
  - Codex powers the GIThub co-pilot.
  - Solves 29% of the problems with 1 sample per problem.
  - 70% with 100 samples per problem.
  - Solves very simple problems
    - Eg., increment a list etc.,
- BTW, Codex is deprecated as of last week ! GPT3.5 seems to have included that model or improved on that considerably.

Scalable Sequence Prediction models

- Use cases in
  - NLP
  - Computer vision
  - Biology
  - Audio/Speech
  - Multi-modal

- With this -- > in code generation as well
  - More natural fit due to "Coding Language"!
  - But has to be very correct - more responsibility unlike a natural language sentence.

Language Models for code

- Even GPT3 produced some code
  - Who knows if they were correct
- GPT3.5 - experiment Demo
  - It did not work with Incoder
  - GPT3.5 talked to me on how to run the code !
- Is it due to Reasoning or it is a Large compressed Intelligent semantic based search engine ?
  - Much like any math equation representing a wealth of data related information in one equation
  - A neural network model with 170Billion parameters obviously will have lots more data ?

Problem Solving Capability

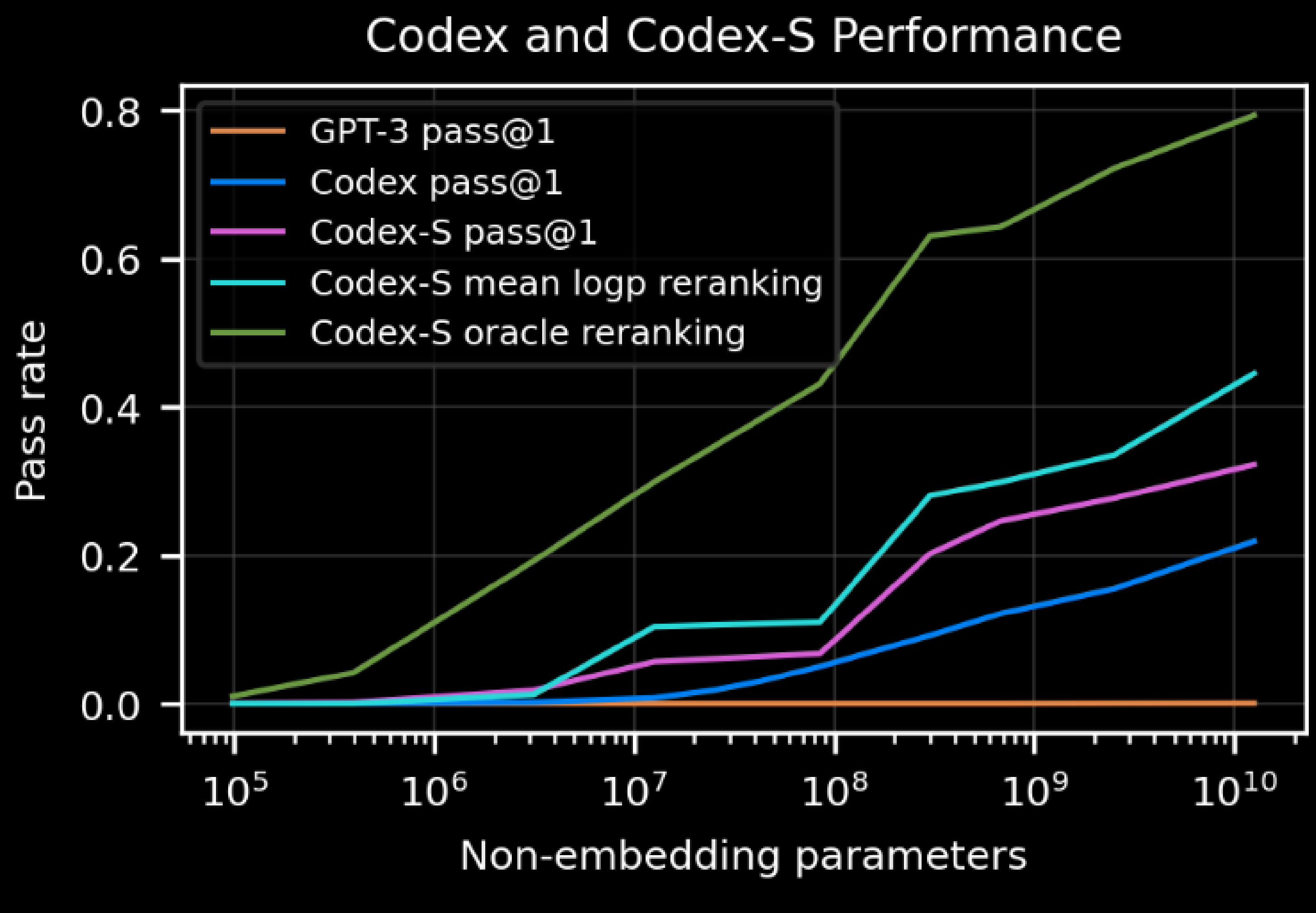


Figure 1. Pass rates of our models on the HumanEval dataset as a

How it works ?

- Docstrings --> code
- Code correctness using "Humanval" benchmark
- Humanval
  - 164 hand written programs with unit tests
  - Assess – language comprehension, algorithm, math, interview (?)
  - Each problem function, docstring, body, tests,
  - 7.7 tests per problem.

How it works ? - One sample

- To solve a problem
  - Generate samples Let us say "1"
  - Check if it passes the humanval unit tests

Problem solving with one sample

Approach: to solve a problem, generate samples and check if any pass the unit tests

With one sample:

Codex (12 billion parameters)	solves	28.8%	of problems
Codex (300 million parameters)	solves	13.2%	of problems
GPT-J (6 billion parameters)	solves	11.4%	of problems
Other GPT models	solve	≈ 0%	of problems

To improve performance, Codex is fine-tuned on correctly implemented functions

Codex-S (12 billion parameters)	solves	37.7%	of problems

More Samples

- No one gets the code right first time !
- So, they created 100 samples (some of which could be wrong !)
  - Rationale seems right ?
- But it works ! With that Codex-S solves ie. Generates one correct function for 77.5 % of the problems.
- Thoughts !!
  - So a programmer model will be – give some input to codex-S, wait for 100 samples, wait for tests, pick the correct one ? Practical? Fast?
- So, now they move to highest mean log-probability (sample that has one!) now it solves 44.5 % of the problems !

Evaluation Framework – should be right !!

- BLEU – in natural language completion (or samples) match with the human sentence and has a value of 0 to 1.
- Programs cannot be verified that way !!
- Sample is correct ONLY WHEN IT WORKS ! Ie., functional correctness.
  - What about - non-functional correctness, reliability, scalability, readability, enhancibility, performance, stability etc., ?
  - Humans do test driven development !! (I test every 10 lines of code as I develop)
- Paper introduces a new metric pass@k metric. (what is the merit of this metric ?)
- Humanval benchmark described in slide 6.

Evaluation Framework – should be right !!

- Pass@k metric !
  - Generate "k" samples
  - Find out which all passes the unit tests
  - Fraction of the "k" samples that passes is reported.
  - Expectation of that random variable is pass@k
  - There is a math way they calculate (not described here! But described in the appendix of the paper)
    - Generate "n" samples >= k
    - Correct samples = 'c'
    - Pass@k = function_of(n, c, k) --> returns a value between 0 and 1.
    - Ie., probability atleast one model is correct.

Evaluation Framework – should be right !!

- Pass@k metric !
  - Generate "k" samples
  - Find out which all passes the unit tests
  - Fraction of the "k" samples that passes is reported.
  - Expectation of that random variable is pass@k
  - There is a math way they calculate (not described here! But described in the appendix of the paper)
    - Generate "n" samples >= k
    - Correct samples = 'c'
    - Pass@k = function_of(n, c, k) --> returns a value between 0 and 1.
    - Ie., probability atleast one model is correct.

Evaluation Framework – how will you test ?

- Random (may be correct !) code is generated by some hidden blackbox that is nuclear packed with billions of bits of data from internet !
  - Is it trustworthy ?
  - Is it secure ?
  - Hope data has been cleaned thoroughly in the training set !
- They use "gvisor" a container framework for isolating this process.
  - Firewall
  - Control groups
  - Resource limitation
  - Absolutely no visibility of anything outside
  - Aha.. How will you test client/server programs then ? Wont we generate code for them ?

Codex – fine tuning

- Codex is fine tuned GPT model with 12B parameters.
  - Same model parameters as GPT.

  - Better tokenizer to accommodate coding languages (eg., large white spaces)
  - White spaces of different lengths reduced tokens by 30%
  - Nucleus Sampling (top p=0.95)
  - Much better performance than GPT

Data Cleaning

- Data Collection – cleaning, filtering
  - Remove the following files:
    - No automatic generated code (eg., django or oracle sql query in c or python)
    - Line length > 100
    - Lower percentage of alphanumerics
    - Everything similar to what we probably do in a spam email or bad code or corruptions etc.,
    - Be ultra careful here !

Prompting to compute pass@k

Each HumanEval problem is assembled into a prompt

signature

function header

```python
def incr_list(l: list):
    """Return list with elements incremented by 1.
    >>> incr_list([1, 2, 3])
    [2, 3, 4]
    >>> incr_list([5, 3, 5, 2, 3, 3, 9, 0, 123])
    [6, 4, 6, 3, 4, 4, 10, 1, 124]
    """
    return [i + 1 for i in l]
```

docstring

Codex 12B: $\text{pass}@1 = 0.9$

Sampling continues until one of the following tokens is encountered:

`'\nclass'` `'\ndef'` `'\n#'` `'\nif'` `'\nprint'`

(otherwise, Codex will keep generating additional functions and statements)

Nucleus sampling (with top $p = 0.95$) is used for all sampling evaluation

Multi-function prompts

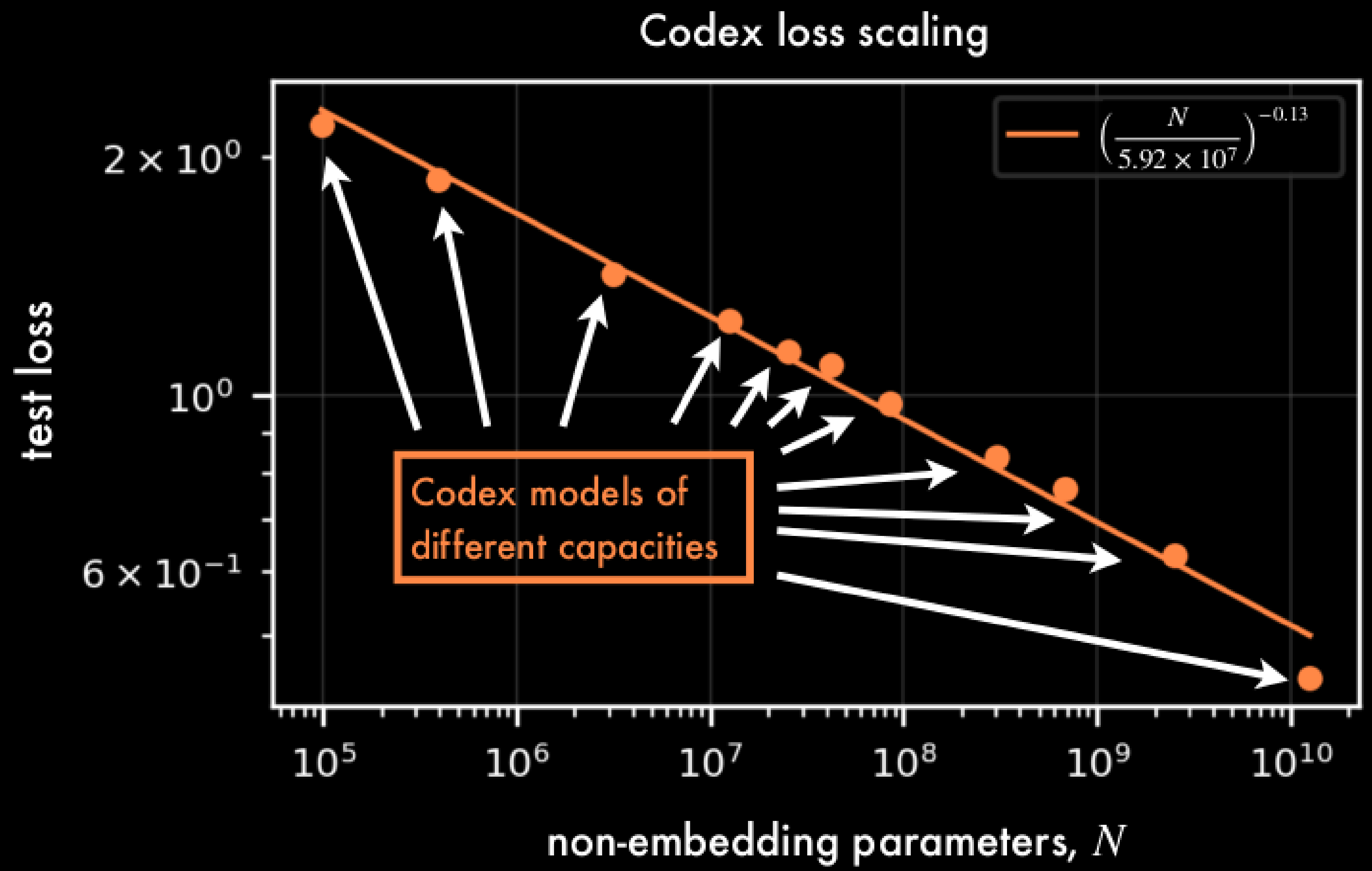
```python
def encode_cyclic(s: str):
    """
    returns encoded string by cycling groups of three characters.
    """
    # split string to groups. Each of length 3.
    groups = [s[(3 * i):min((3 * i + 3), len(s))] for i in range((len(s) + 2) // 3)]
    # cycle elements in each group. Unless group has fewer elements than 3.
    groups = [(group[1:] + group[0]) if len(group) == 3 else group for group in groups]
    return "".join(groups)

def decode_cyclic(s: str):
    """
    takes as input string encoded with encode_cyclic function. Returns decoded string.
    """
    # split string to groups. Each of length 3.
    groups = [s[(3 * i):min((3 * i + 3), len(s))] for i in range((len(s) + 2) // 3)]
    # cycle elements in each group.
    groups = [(group[-1] + group[:-1]) if len(group) == 3 else group for group in groups]
    return "".join(groups)
```

Codex 12B: pass@1 = 0.005

Loss scaling

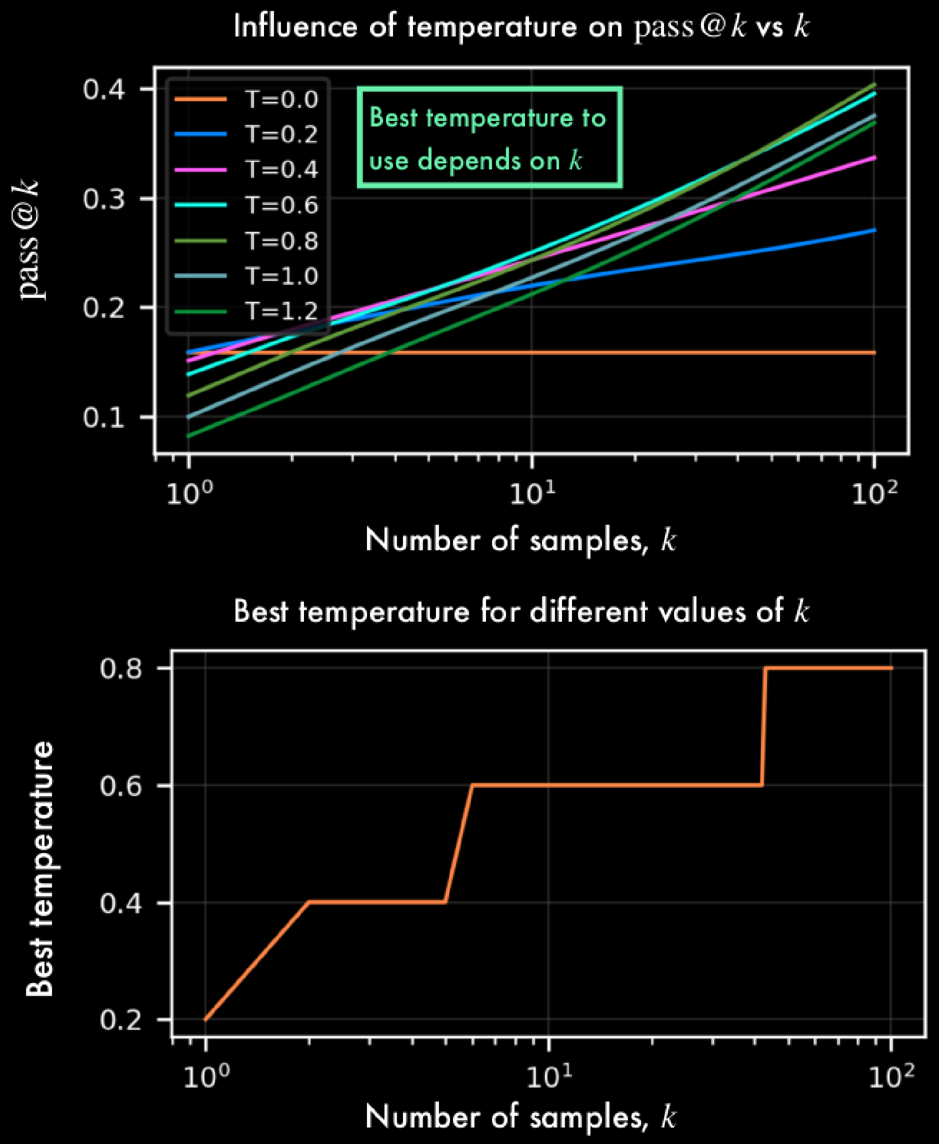
Language model losses appear to follow a power law (Kaplan et al., 2020)

Similarly, plot Codex test loss on a held-out val set of GitHub corpus:



Takeaway: Codex fine-tuning appears to follow a power law with model size

Sampling temperature



For larger k, higher temperatures (higher diversity) work better

pass@k only rewards whether the model generates any solution

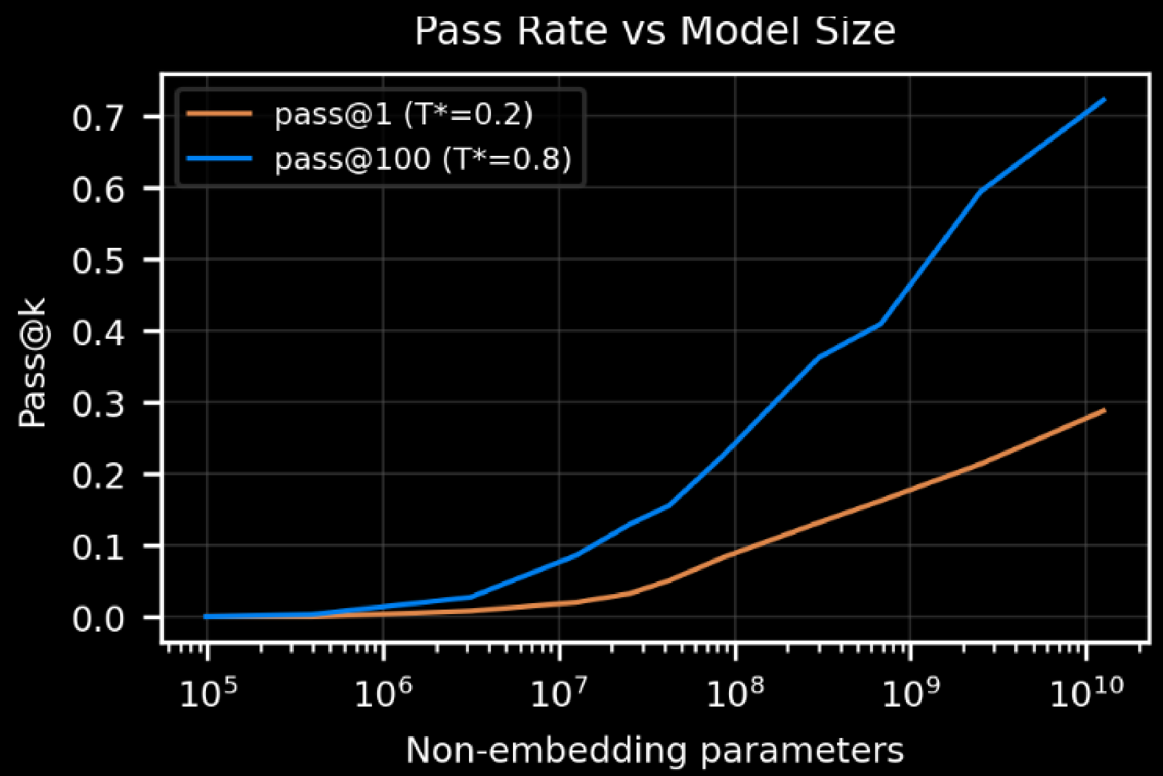


Figure 6. Using the optimal temperatures 0.2 and 0.8 for pass@1 and pass@100, we plot these two metrics as a function of model size. Performance appears to scale smoothly as a sigmoid in log-parameters.

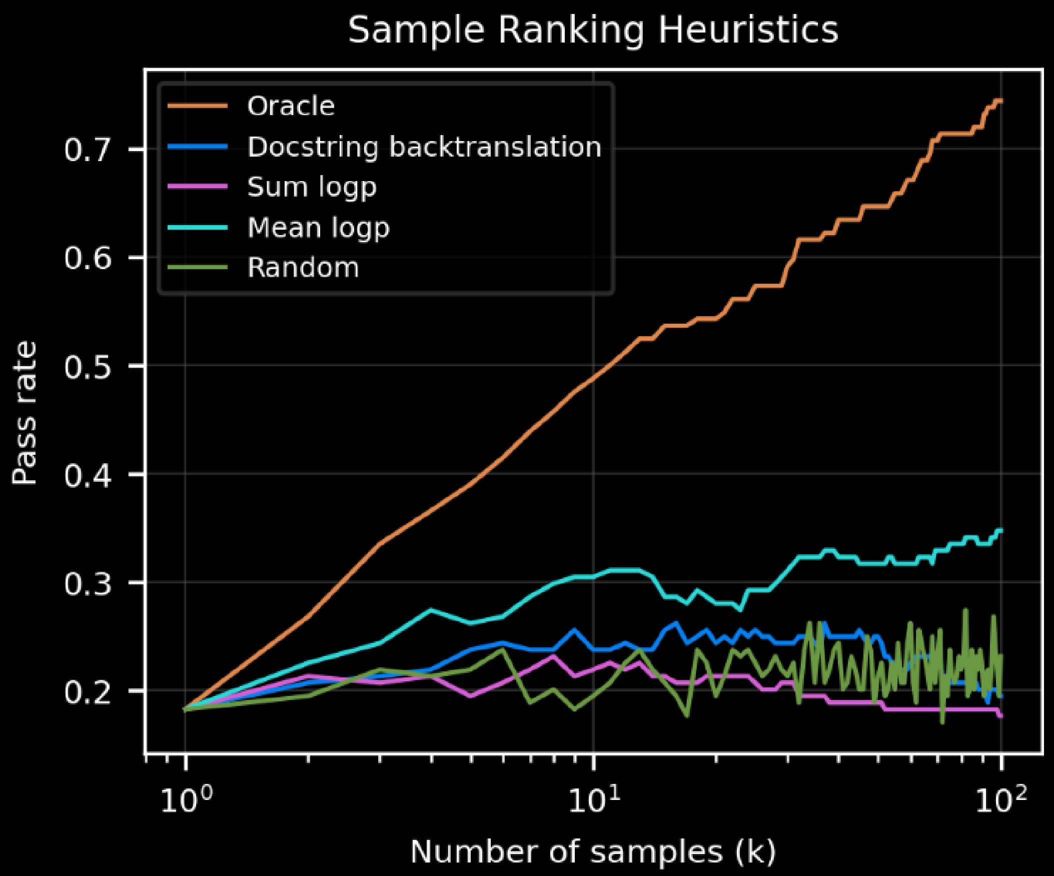


Figure 7. Model performance in the setting where we can generate multiple samples, but only evaluate one. We can do better than randomly selecting a sample by choosing the solution with the highest mean log-probability (red) or with the highest back-translation score (orange) described in Sec. 5. The blue line represents the theoretical best performance obtained using an oracle with prior knowledge of the unit tests.

Related Approaches

Two models in the same vein as Codex:

GPT-Neo (Black et al., 2021)

GPT-J-6B (Wang et al., 2021)

Both are trained on The Pile (8% of which is sourced from GitHub)

GPT-J-6B appears to produce qualitatively reasonable code (Woolf, 2021)

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

Temperatures

GPT-Neo: 0.2, 0.4, 0.8

GPT-J-6B: 0.2, 0.8

Tabnine: 0.4, 0.8

x20 fewer parameters than GPT-J-6B

Codex-12B goes considerably beyond the performance of prior models

APPS - (competitive problem solving) dataset

- 5000 training, 5000 test/evaluation
- Each example includes unit tests
- Competitive problems is full program although core is one function
- Metrics used for evaluation:
  - Full correctness/partial correct as well (coding competition test cases only some may pass)'
  - Timeouts may happen.

APPS - Results

Evaluating Large Language Models Trained on Code

Table 2. Finetuned GPT-Neo numbers from the APPS paper referenced above. For Codex-12B, the number of passing programs that timeout on some test is in the bracket. We used temperature 0.6 for sampling to cover all k in pass@k, so raw pass@1 results could be improved with lower temperature.

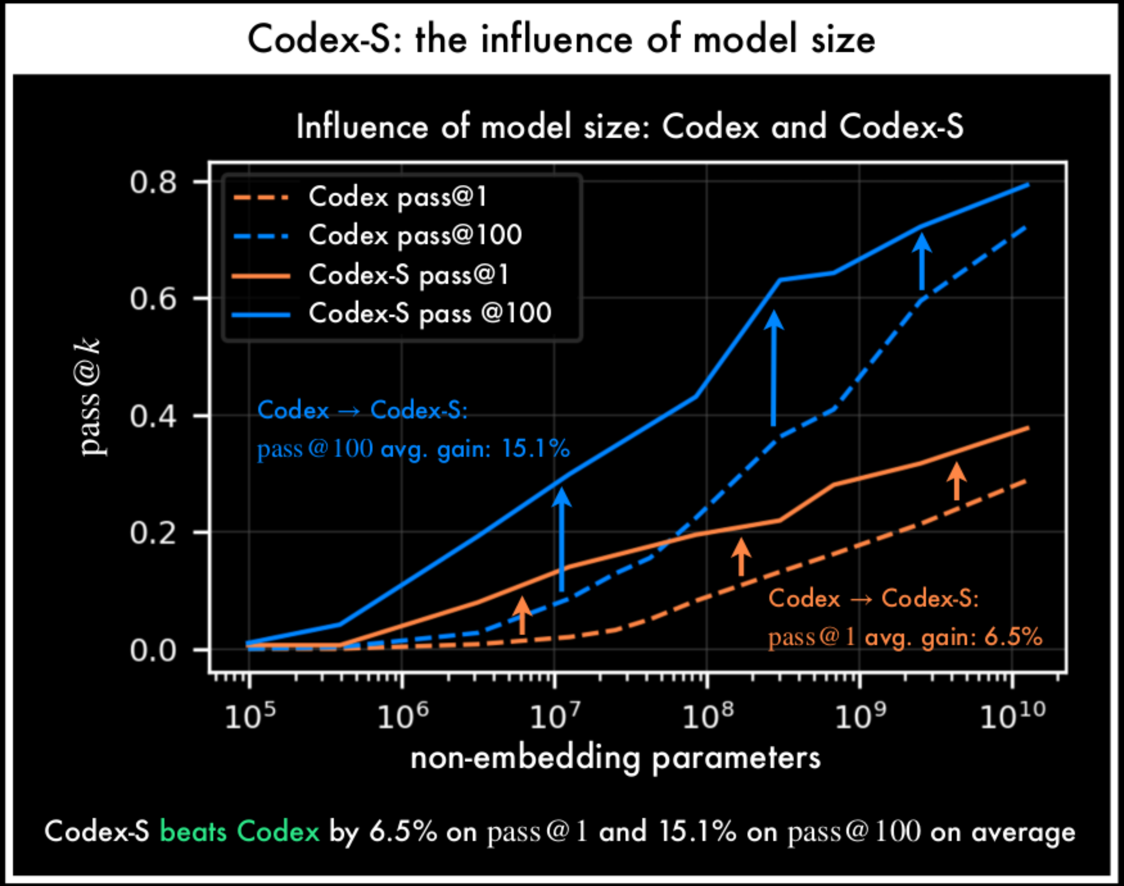
	Introductory	Interview	Competition
GPT-Neo 2.7B raw pass@1	3.90%	0.57%	0.00%
GPT-Neo 2.7B raw pass@5	5.50%	0.80%	0.00%
1-shot Codex raw pass@1	4.14% (4.33%)	0.14% (0.30%)	0.02% (0.03%)
1-shot Codex raw pass@5	9.65% (10.05%)	0.51% (1.02%)	0.09% (0.16%)
1-shot Codex raw pass@100	20.20% (21.57%)	2.04% (3.99%)	1.05% (1.73%)
1-shot Codex raw pass@1000	25.02% (27.77%)	3.70% (7.94%)	3.23% (5.85%)
1-shot Codex filtered pass@1	22.78% (25.10%)	2.64% (5.78%)	3.04% (5.25%)
1-shot Codex filtered pass@5	24.52% (27.15%)	3.23% (7.13%)	3.08% (5.53%)

Codex-S supervised fine-tuning

- Use functional code from competitive programming website, continuous integration website.
- 10000 from competitive websites, 40000 from CI websites.
- Filter those that works well.
- The problems are collected in the same format problem in docstring, solutions.
- Codex-S performed better, but the pass@k rates for k>1 required higher temperatures than codex. (Obviously due to codex-S having more narrower/crisper definitions).

Codex-S results

Supervised Fine-tuning: Results



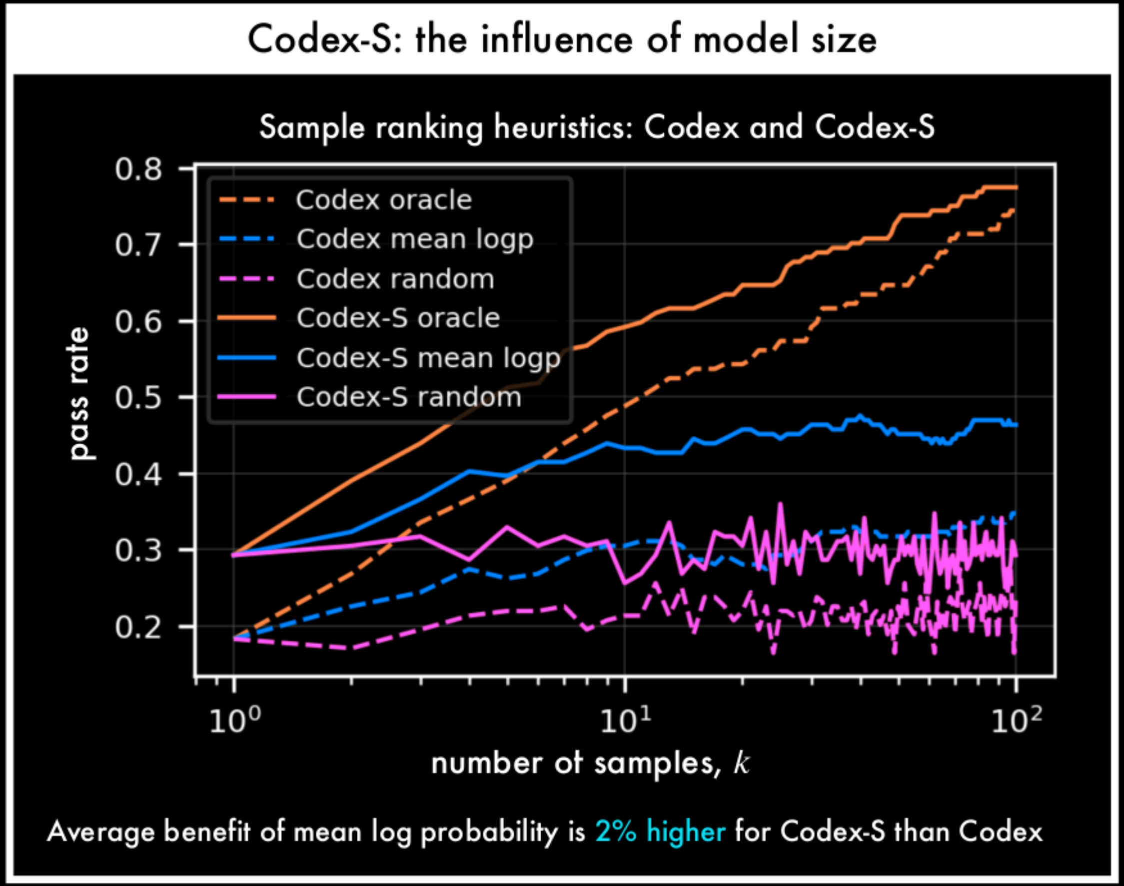
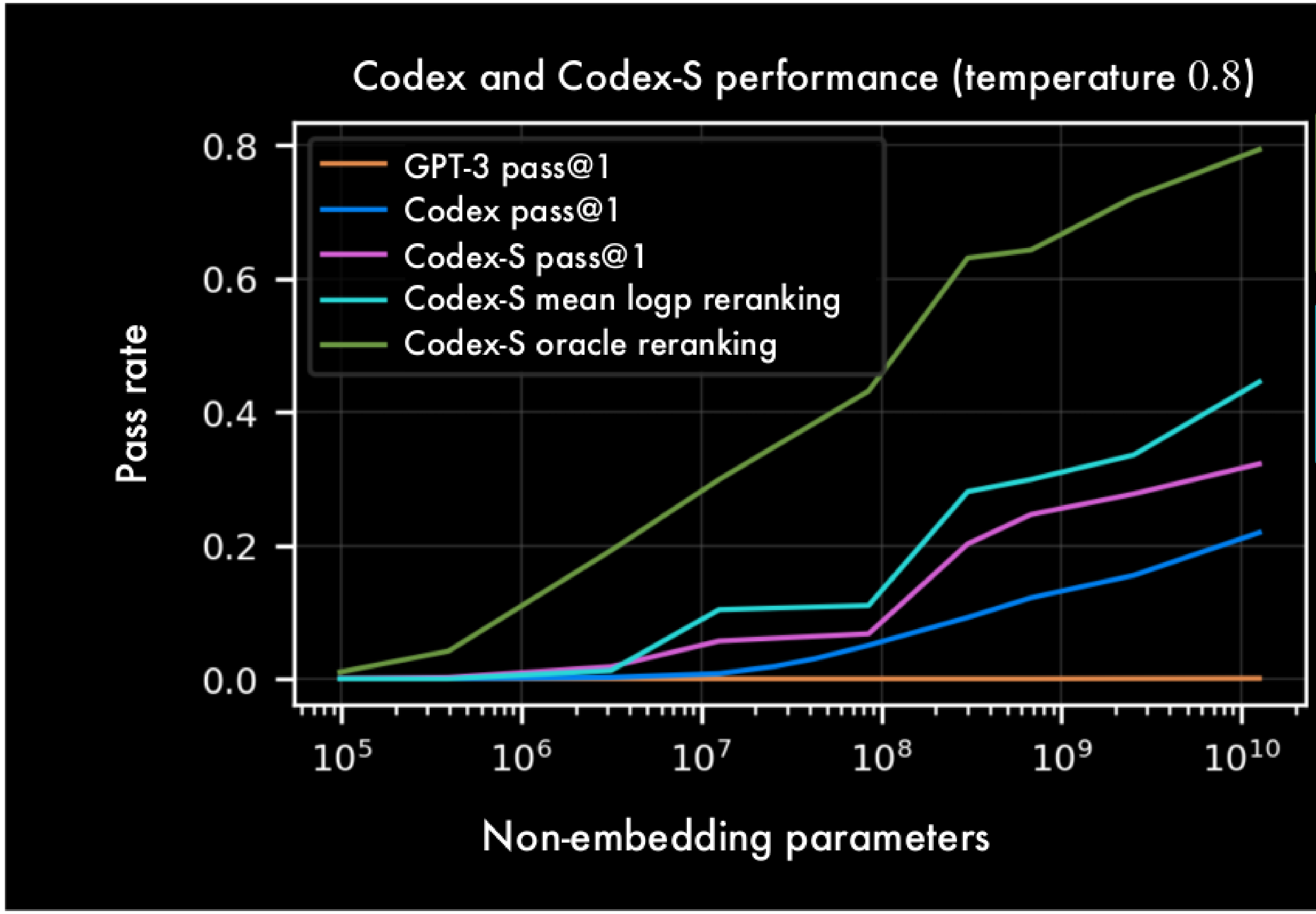


Image credits/References:

Codex-S results

Comparing training strategies on different model sizes on HumanEval



Codex-S solves 77.5% by selecting the sample among 100 passing the unit tests

Codex-S solves 44.5% by reranking 100 samples via mean logp

Limitations – more components ie. functions

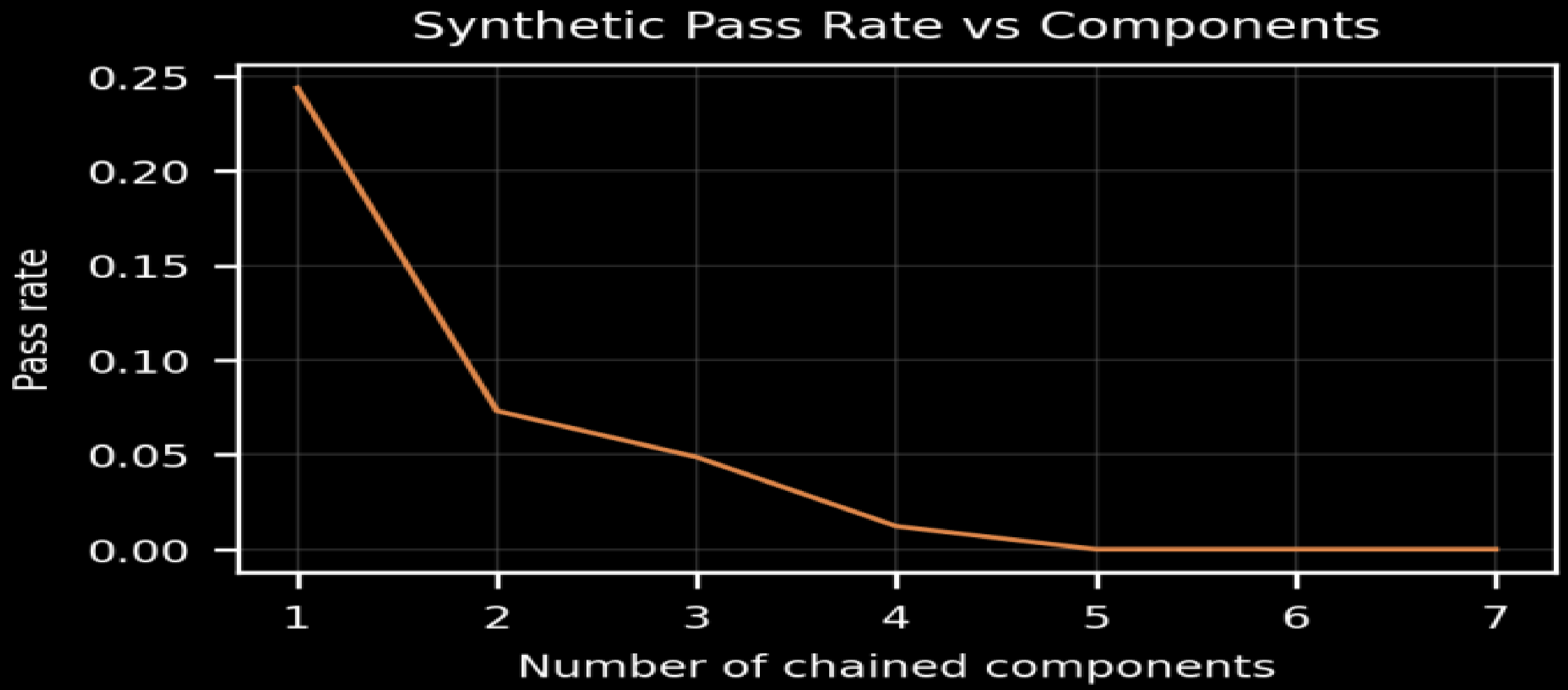


Figure 11. Pass rates of Codex-12B samples against the number of chained components in the synthetically generated docstring. With each additional component, pass rate drops by roughly a factor of 2-3.

Limitations

- Sums of math – binding attributes to objects

```
def do_work(x, y, z, w):
    """ Add 3 to y, then subtract 4
    from both x and w. Return the
    product of the four numbers. """
    t = y + 3
    u = x - 4
    v = z * w
    return v
```

Limitations

- Over-reliance - has subtle bugs
- Mistakes in training code generate bad code
- Security issues
- For developers
  - Reduce cost of producing s/w
  - Can focus on value add stuff (say design docs)
- Security implications – huge !
- IP issues – who owns the code if same code generated by same s/w in two companies ?

Future work

- Paper lists
  - Quantify economic value
  - Documentation/testing practice
  - Impact of code generation tools metrics (time saved etc.)
  - More language grammars insights (?)
- My observations
  - Where is the reasoning here ?
  - No wonder having billions of parameters returns better results due to huge amounts of compressed data inside the model.
  - Dynamic code generation library for simple functions …, testing etc. Are more complex.. More work !