Evaluating Large Language Models Trained on Code

Presented by:

Arush S. Sharma

Course Name:

ECE 696B: Trustworthy Machine Learning

Instructor:

Dr. Ravi Tandon

Affiliation: University of Arizona

Outline

Evaluating Large Language Models Trained on Code

Mark Chen [*1] Jerry Tworek [*1] Heewoo Jun [*1] Qiming Yuan [*1] Henrique Ponde de Oliveira Pinto [*1]
Jared Kaplan [*2] Harri Edwards [1] Yuri Burda [1] Nicholas Joseph [2] Greg Brockman [1] Alex Ray [1] Raul Puri [1]
Gretchen Krueger [1] Michael Petrov [1] Heidy Khlaaf [3] Girish Sastry [1] Pamela Mishkin [1] Brooke Chan [1]
Scott Gray [1] Nick Ryder [1] Mikhail Pavlov [1] Alethea Power [1] Lukasz Kaiser [1] Mohammad Bavarian [1]
Clemens Winter [1] Philippe Tillet [1] Felipe Petroski Such [1] Dave Cummings [1] Matthias Plappert [1]
Fotios Chantzis [1] Elizabeth Barnes [1] Ariel Herbert-Voss [1] William Hebgen Guss [1] Alex Nichol [1] Alex Paino [1]
Nikolas Tezak [1] Jie Tang [1] Igor Babuschkin [1] Suchir Balaji [1] Shantanu Jain [1] William Saunders [1]
Christopher Hesse [1] Andrew N. Carr [1] Jan Leike [1] Josh Achiam [1] Vedant Misra [1] Evan Morikawa [1]
Alec Radford [1] Matthew Knight [1] Miles Brundage [1] Mira Murati [1] Katie Mayer [1] Peter Welinder [1]
Bob McGrew [1] Dario Amodei [2] Sam McCandlish [2] Ilya Sutskever [1] Wojciech Zaremba [1]

[*]Equal contribution
[1]OpenAI, San Francisco, California, USA.
[2]Anthropic AI, San Francisco, California, USA. Work performed while at OpenAI.
[3]Zipline, South San Francisco, California, USA. Work performed while at OpenAI.
Correspondence to: Mark Chen <mark@openai.com>, Jerry Tworek <jt@openai.com>, Heewoo Jun <heewoo@openai.com>, Qiming Yuan <qiming@openai.com>.

Year: 2021

Introduction

- Sequence prediction models have been used for text generation and representation learning in various domains: NLP, computer vision, etc.
- GPT-3 (not initially trained for code generation) could generate simple programs from Python docstrings
- Codex, a finetuned GPT-3 model on publicly available dataset from GitHub is given the task to generate standalone python functions from the docstring

Contribution

- From the docstrings, generate the python function and evaluate the correctness of the code through unit tests

- To benchmark the model, 164 coding problems along with unit tests are created (HumanEval dataset)

- Coding problems cover language comprehension, algorithms, mathematics similar to interview question (coding round)

Examples

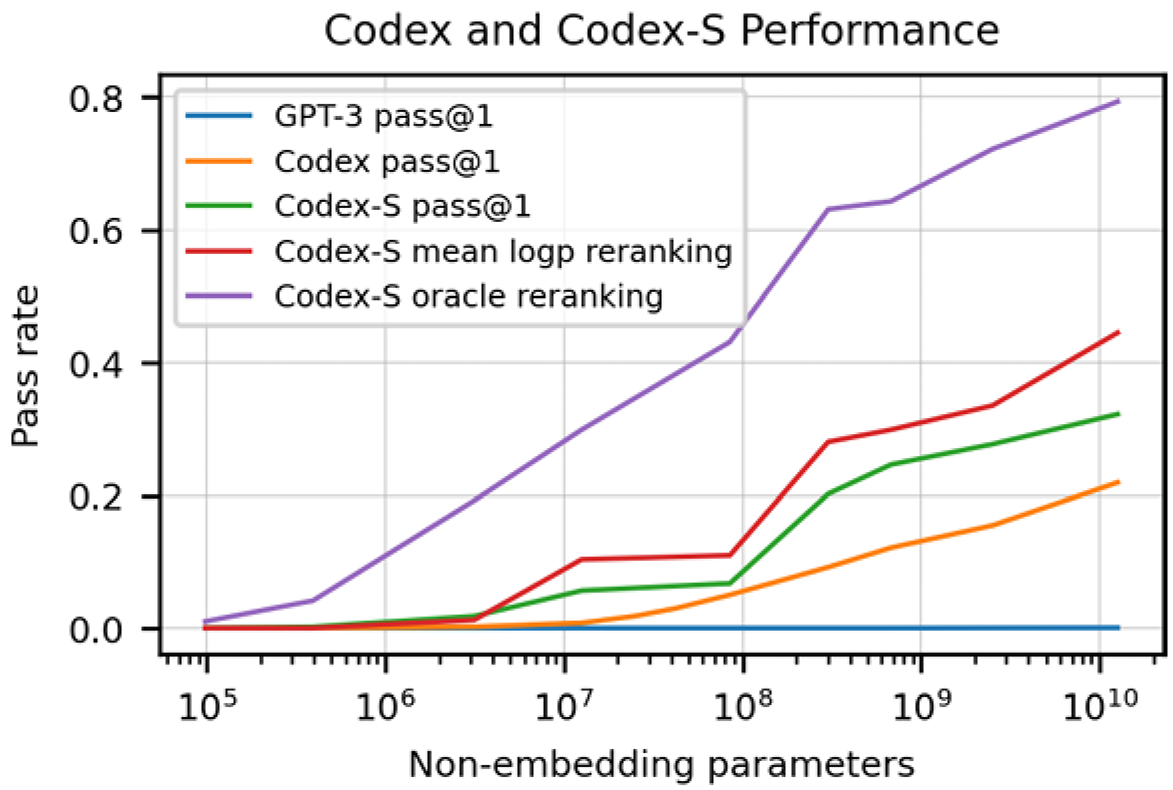
```python
def solution(lst):
    """Given a non-empty list of integers, return the sum of all of the odd elements
    that are in even positions.

    Examples
    solution([5, 8, 7, 1]) ==>12
    solution([3, 3, 3, 3, 3]) ==>9
    solution([30, 13, 24, 321]) ==>0
    """
    return sum(lst[i] for i in range(0,len(lst)) if i % 2 == 0 and lst[i] % 2 == 1)
```

```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]
```

Key Findings

During Evaluation:

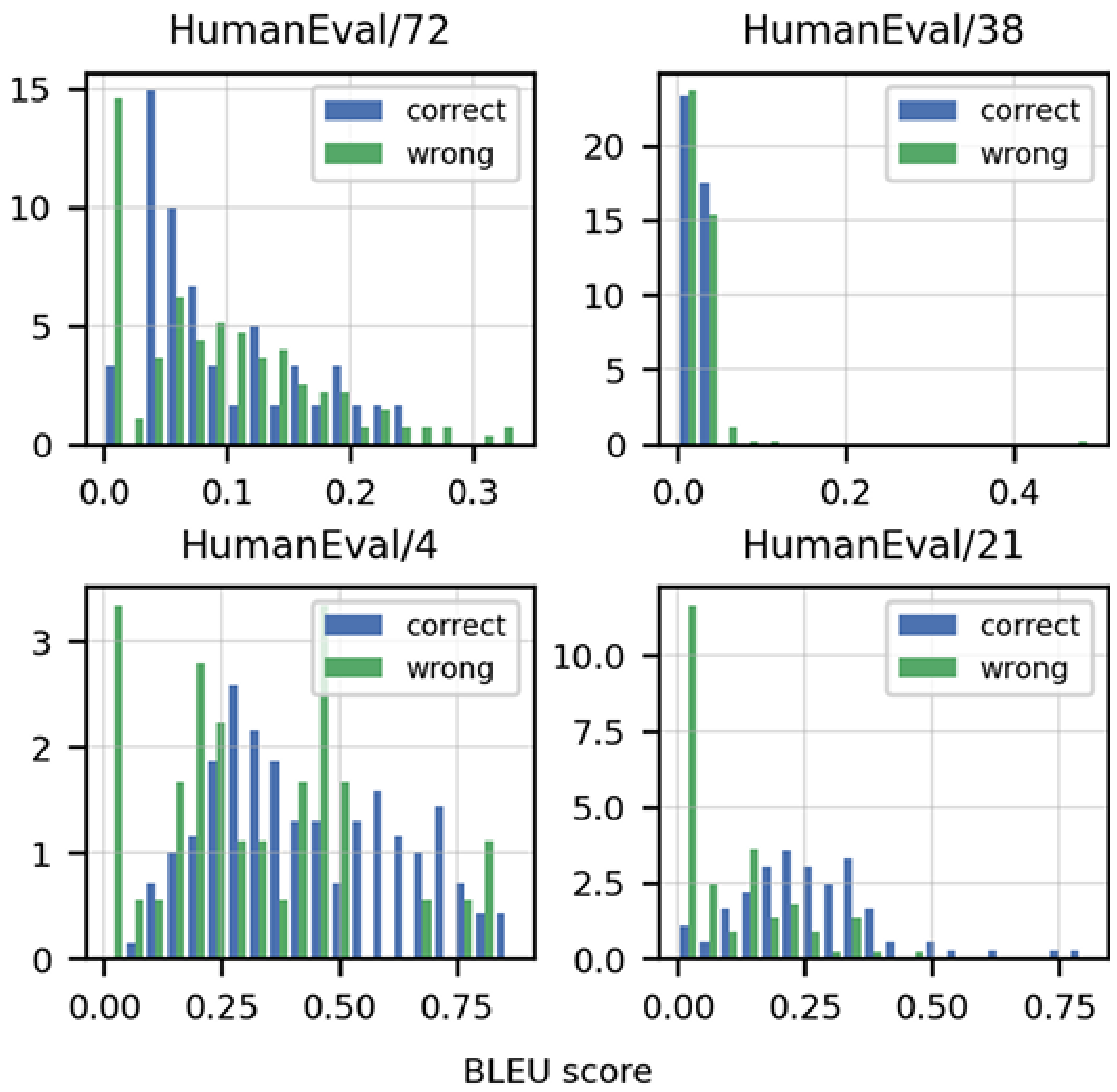
- Generate multiple samples of the code and check if any of them passes the unit tests
- A single code sample from Codex (12B parameters) solved 28.8% problems
- Codex (300M parameters) solved 13.2% problems
- GPT-J (6B parameters) solved 11.4% problems and all other GPT models achieved 0%



Evaluation Framework

- BLEU score metric is useful to compare the similarity between machine translated text and a reference text

- However, BLEU score metric is not feasible metric to test the code correctness



Evaluation Framework

- **Functional correctness** (a sample which passes all the unit tests) is a useful metric to judge the correctness of code based on docstring based generation

  - *pass@k*: k code samples are generated and if any sample passes the unit tests, the problem is considered solved

- Computing *pass@k* score: generate $n \geq k$ samples ($n = 200$, $k \leq 100$ and c is # of correct samples that passes the unit tests)

$$\text{pass}@k := \mathop{\mathbb{E}}_{\text{Problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]$$

Pass@k Examples

- 8 random samples per problem from Codex-12B

```python
def words_string(s):
    """
    You will be given a string of words separated by
        commas or spaces. Your task is
    to split the string into words and return an array
        of the words.

    For example:
    words_string("Hi, my name is John") == ["Hi", "my"
        "name", "is", "John"]
    words_string("One, two, three, four, five, six") =
        ["One", "two", "three", "four", "five", "six
    """

    ### COMPLETION 1 (CORRECT): ###
    words = []
    splitted_word = ""
    for char in s:
        if char == "␣":
            if splitted_word != "":
                words.append(splitted_word)
                splitted_word = ""
        elif char != ",":
            splitted_word += char
    if splitted_word != "":
        words.append(splitted_word)
    return words

    ### COMPLETION 2 (WRONG): ###
    return s.split()

    ### COMPLETION 3 (WRONG): ###
    words = s.split("␣")
    words = [w.strip(",") for w in words]
    words = [w.strip() for w in words]

    ### COMPLETION 4 (WRONG): ###
    # Split the string into words
    words = s.split(',')
    # Remove any whitespace around each element
    words = [w.strip() for w in words]
    # Return an array of the words in the string
    return words

    ### COMPLETION 5 (WRONG): ###
    return s.split()

    ### COMPLETION 6 (WRONG): ###
    return s.split()

    ### COMPLETION 7 (WRONG): ###
    return s.split(',')

    ### COMPLETION 8 (WRONG): ###
    return s.split(sep=None)
```

Pass@k Examples

```python
def is_prime(n):
    """Return true if a given number is prime, and
       false otherwise.
    >>> is_prime(6)
    False
    >>> is_prime(101)
    True
    >>> is_prime(11)
    True
    >>> is_prime(13441)
    True
    >>> is_prime(61)
    True
    >>> is_prime(4)
    False
    >>> is_prime(1)
    False
    """
```

```python
### COMPLETION 1 (WRONG): ###
for i in range(2, n):
    if n % i == 0:
        return False
return True

### COMPLETION 2 (WRONG): ###
for i in range(2, n):
    if n % i == 0:
        return False
return True

### COMPLETION 3 (WRONG): ###
# TODO: implement this function
pass

### COMPLETION 4 (CORRECT): ###
if n < 2: return False
if n == 2: return True
if n%2 == 0: return False
return not any(n%k == 0 for k in range(3,int(n
    **0.5)+1,2))
```

Pass@k Examples

```
### COMPLETION 5 (WRONG): ###
def helper(n):
    # check if an odd number is multiple of any
        integer up to the square root of the number
    # if this is true, n is not prime
    for i in range(2, int(n**0.5 + 1)):
        if n % i == 0:
            return False
    return True
# if helper returns true, then n is prime, else n
    is not prime
if helper(n):
    return True
else:
    return False
```

```
### COMPLETION 6 (CORRECT): ###
prime = True
if n == 1:
    return False
for i in range(2, n):
    if n % i == 0:
        prime = False
return prime
```

```
### COMPLETION 7 (WRONG): ###
for i in range(2, n):
    if n % i == 0:
        return False
return True
```

```
### COMPLETION 8 (CORRECT): ###
if n == 2 or n == 3:
    return True
if n < 2 or n % 2 == 0:
    return False
if n < 9:
    return True
if n % 3 == 0:
    return False
r = int(n**0.5)
f = 5
while f <= r:
    if n % f == 0:
        return False
    if n % (f+2) == 0:
        return False
    f += 6
return True
```

Pass@k Examples

```python
def x_or_y(n, x, y):
    """A simple program which should return the value
        of x if n is
    a prime number and should return the value of y
        otherwise.

    Examples:
    for x_or_y(7, 34, 12) == 34
    for x_or_y(15, 8, 5) == 5

    """

    ### COMPLETION 1 (WRONG): ###
    return x if n % x == 0 else y


    ### COMPLETION 2 (WRONG): ###
    if n > 1:
        return x if n%2 != 0 else y
    else:
        return "Error"


    ### COMPLETION 3 (WRONG): ###
    if n%x == 0:
        return x
    else:
        return y
```

```python
### COMPLETION 4 (WRONG): ###
return x if n % x == 0 else y


### COMPLETION 5 (WRONG): ###
if n < x:
    return n
else:
    return y


### COMPLETION 6 (WRONG): ###
for i in range(2,n-1):
    if (n % i == 0):
        return y
return x


### COMPLETION 7 (WRONG): ###
if n == 2:
    return x
elif n == 3:
    return y
else:
    return x_or_y(n-1, x, y)


### COMPLETION 8 (WRONG): ###
if n == x:
    return x
elif n == y:
    return y
else:
    return n
```

Dataset Information

HumanEval Dataset

- Each problem has a function template, docstring, function body, and several unit tests
- The dataset contains coding tasks pertaining to simple mathematics, language comprehension, algorithm, and reasoning

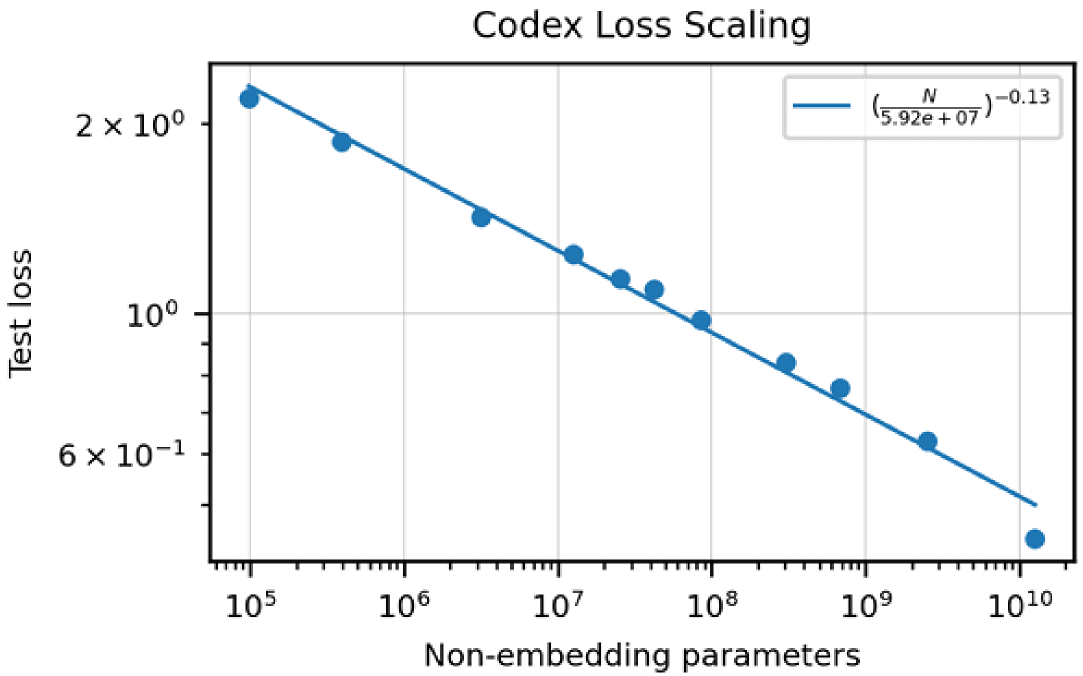
Training Dataset

- 54 million public repository on GitHub
- Each file was less than 1 MB (a total of 179 GB)
- After the preprocessing, the dataset totaled around 160 GB

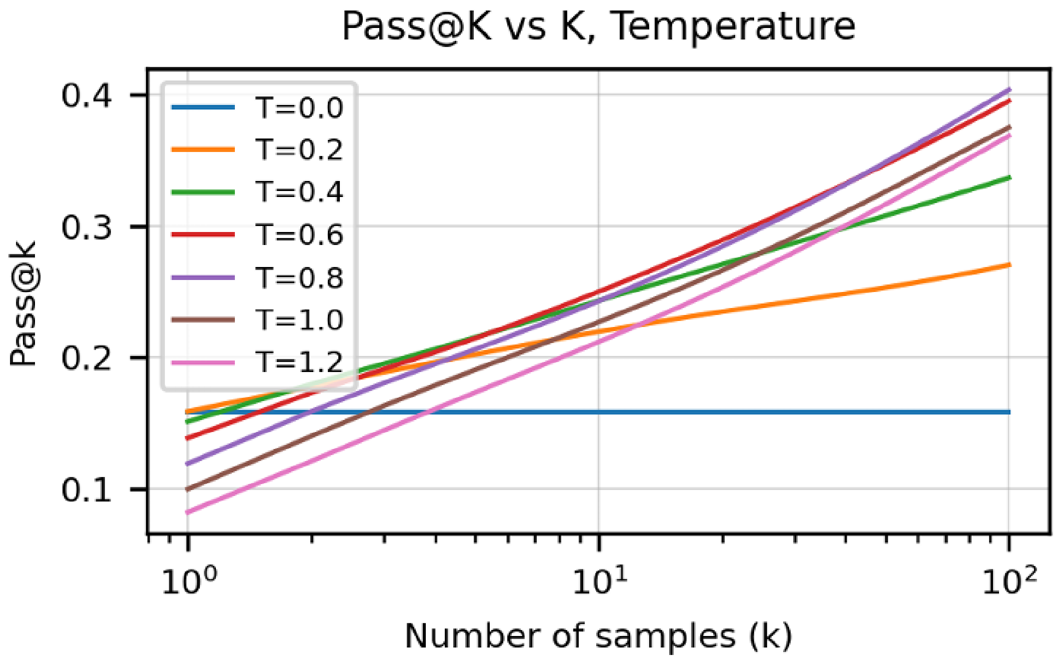
Training Parameters

- Codex is built on GPT-3 and is finetuned for the code generation
- During the training: 175 steps are used with cosine learning rate decay
- 100 billion tokens are used Adam optimizer ($\beta_1 = 0.9, \beta_2 = 0.95$)

Results

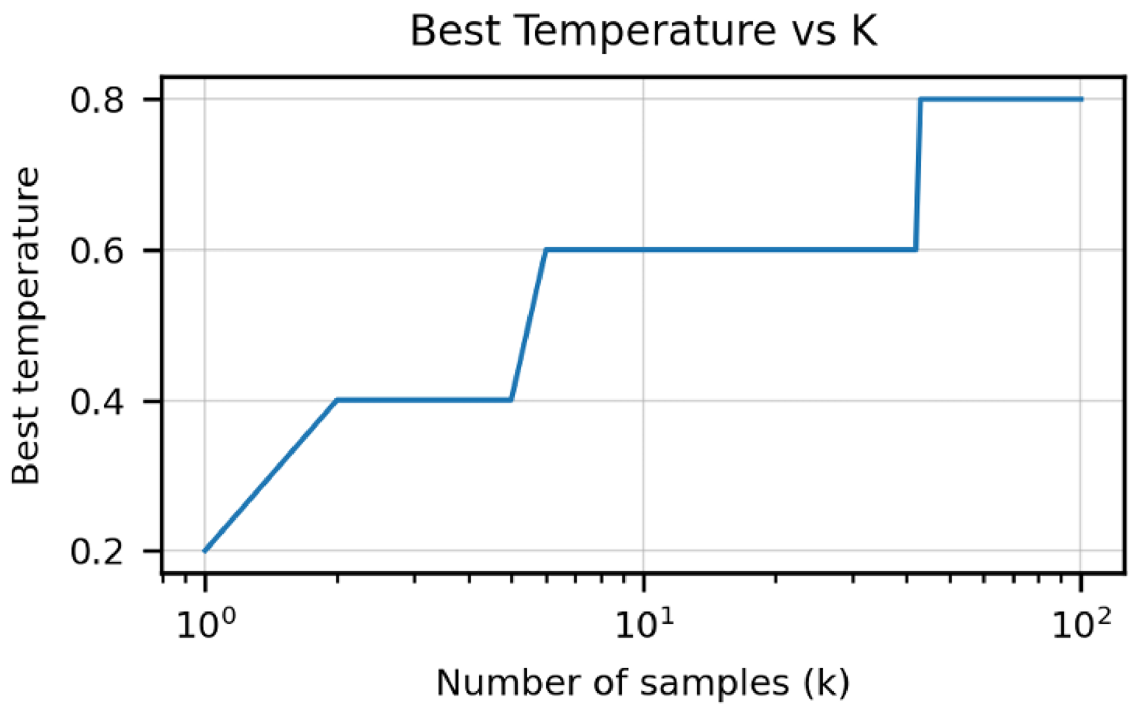


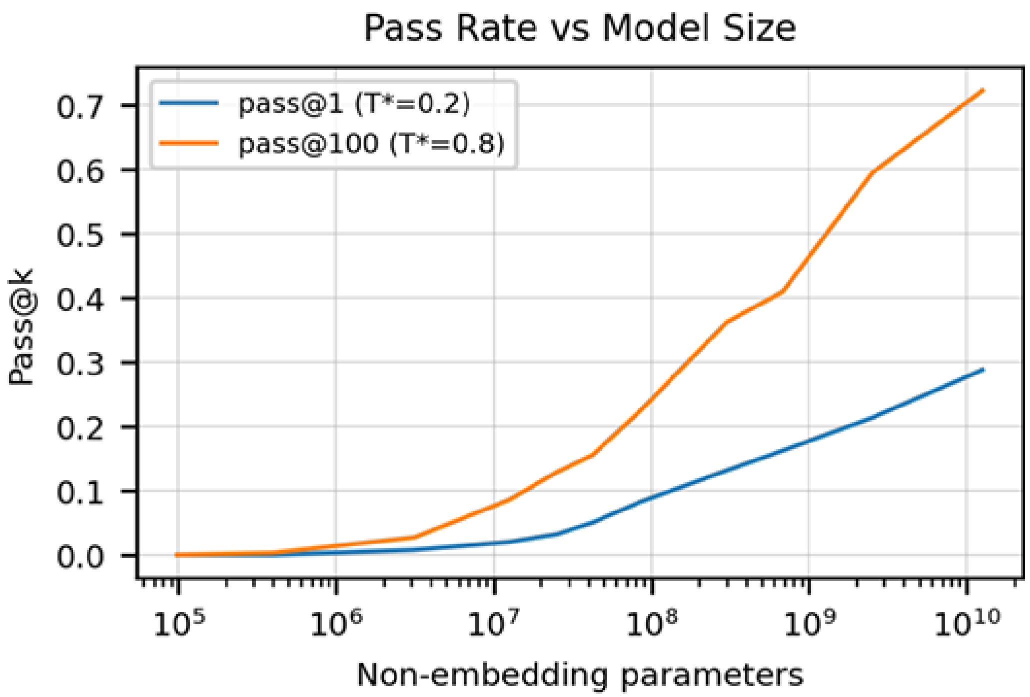
Codex Loss Scaling
$(\frac{N}{5.92e+07})^{-0.13}$
2×10^0
10^0
6×10^{-1}
Test loss
10^5
10^6
10^7
10^8
10^9
10^{10}
Non-embedding parameters



Pass@K vs K, Temperature
T=0.0
T=0.2
T=0.4
T=0.6
T=0.8
T=1.0
T=1.2
0.4
0.3
0.2
0.1
Pass@k
10^0
10^1
10^2
Number of samples (k)

Results


Best Temperature vs K
Best temperature
0.8
0.6
0.4
0.2
10^0
10^1
10^2
Number of samples (k)


Pass Rate vs Model Size
pass@1 (T*=0.2)
pass@100 (T*=0.8)
Pass@k
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
10^5
10^6
10^7
10^8
10^9
10^{10}
Non-embedding parameters

Comparative Analysis

Table 1. Codex, GPT-Neo, & TabNine evaluations for HumanEval. We find that GPT-J pass@1 is between Codex-85M and Codex-300M performance.

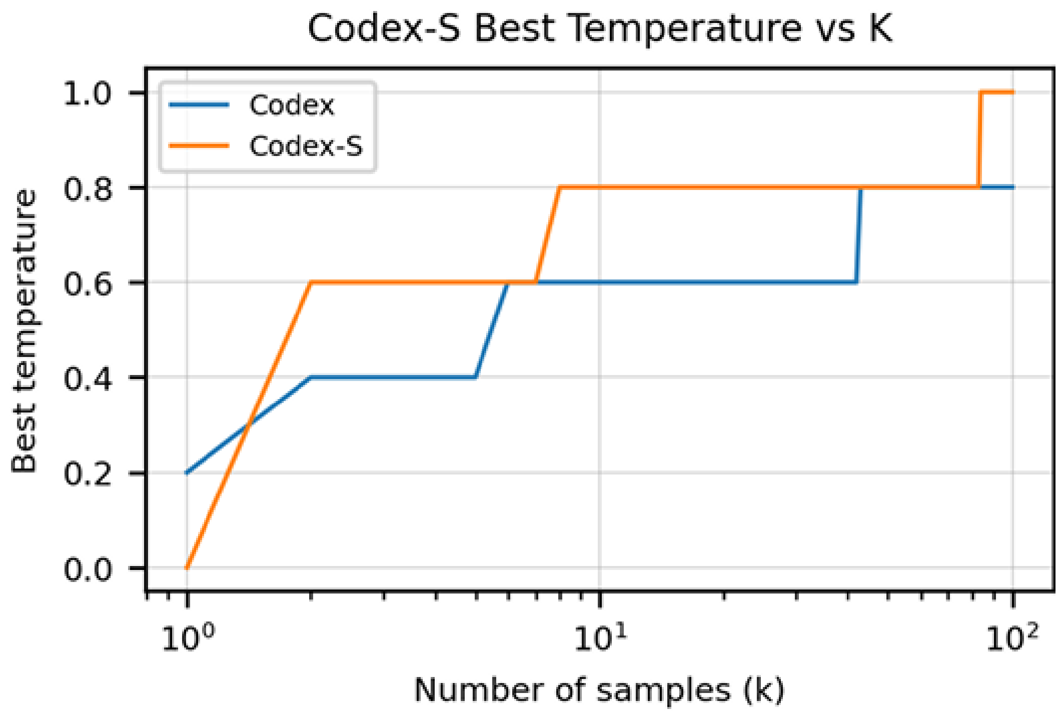
	PASS@k $k=1$	$k=10$	$k=100$
GPT-NEO 125M	0.75%	1.88%	2.97%
GPT-NEO 1.3B	4.79%	7.47%	16.30%
GPT-NEO 2.7B	6.41%	11.27%	21.37%
GPT-J 6B	11.62%	15.74%	27.74%
TABNINE	2.58%	4.35%	7.59%
CODEX-12M	2.00%	3.62%	8.58%
CODEX-25M	3.21%	7.1%	12.89%
CODEX-42M	5.06%	8.8%	15.55%
CODEX-85M	8.22%	12.81%	22.4%
CODEX-300M	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%

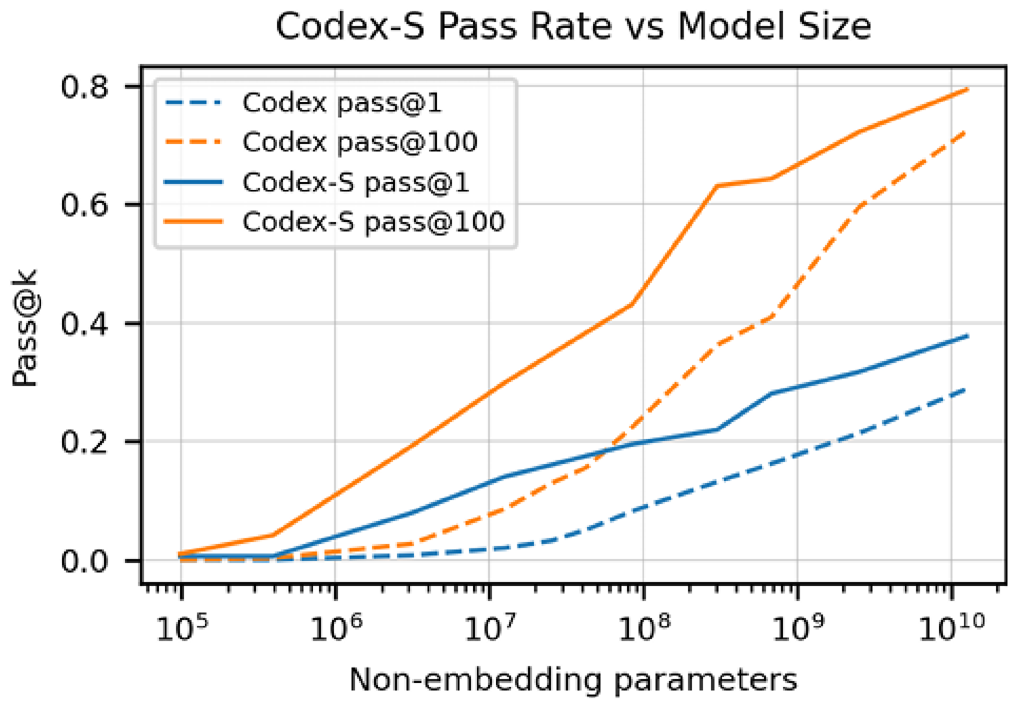
Supervised Fine Tuning

- The training dataset is constructed using the problems from competitive websites and from repositories with continuous integration (CLI)
- The authors collected 10,000 problems with problem statement, function signatures, and solutions from popular programming contest
- From CLI, 40,000 problems were collected. Overall, the authors made sure that these projects don’t contain untrusted code
- Finally, those problem samples were not included when pass@100 model fails in passing the unit tests

Supervised Fine Tuning

- The model was finetuned on the modified training dataset and named as Codex-S





Docstring Generation

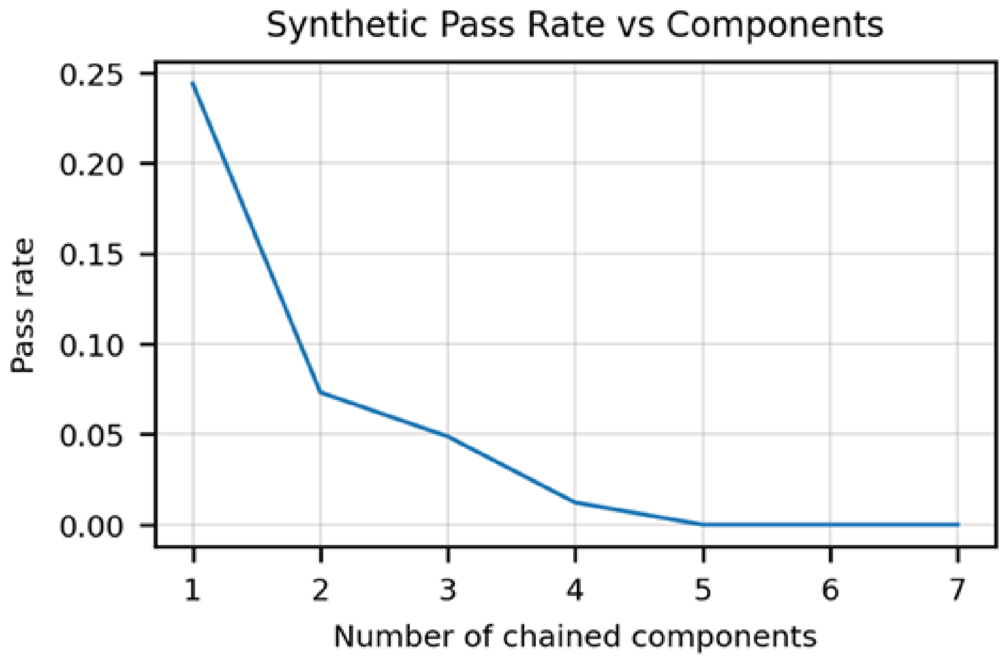
- Reverse engineering: Convert the programming code to docstring to know the intent of the program (for safety reason)
- For evaluation purposes, 10 docstring samples per code were graded by humans

MODEL	PASS@1	PASS@10
CODEX-S-12B	32.2%	59.5%
CODEX-D-12B	20.3%	46.5%

- Codex-D performs comparatively lower than Codex-S possibly due to coders devoting less time in writing good quality docstrings

Limitations

- Codex is not efficient to train since the dataset contains hundreds of millions of lines of code making it energy inefficient
- Model performance degrades as the length of docstring increases




```
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

- Codex can be prompted in ways that generate racist, denigratory, and otherwise harmful outputs as code comments

Conclusion

- The authors investigated whether it was possible to train large language models to produce functionally correct code bodies from docstrings
- By finetuning GPT-3 model (Codex) the model could solve the problems from docstring
- The performance is enhanced by producing multiple samples (k) from the model
- Finally, the authors trained a model to output the docstrings from code and found that the performance profiles of these models were similar