Language Models of Code

Arseniy Andreyev, Jiatong Yu

Outline

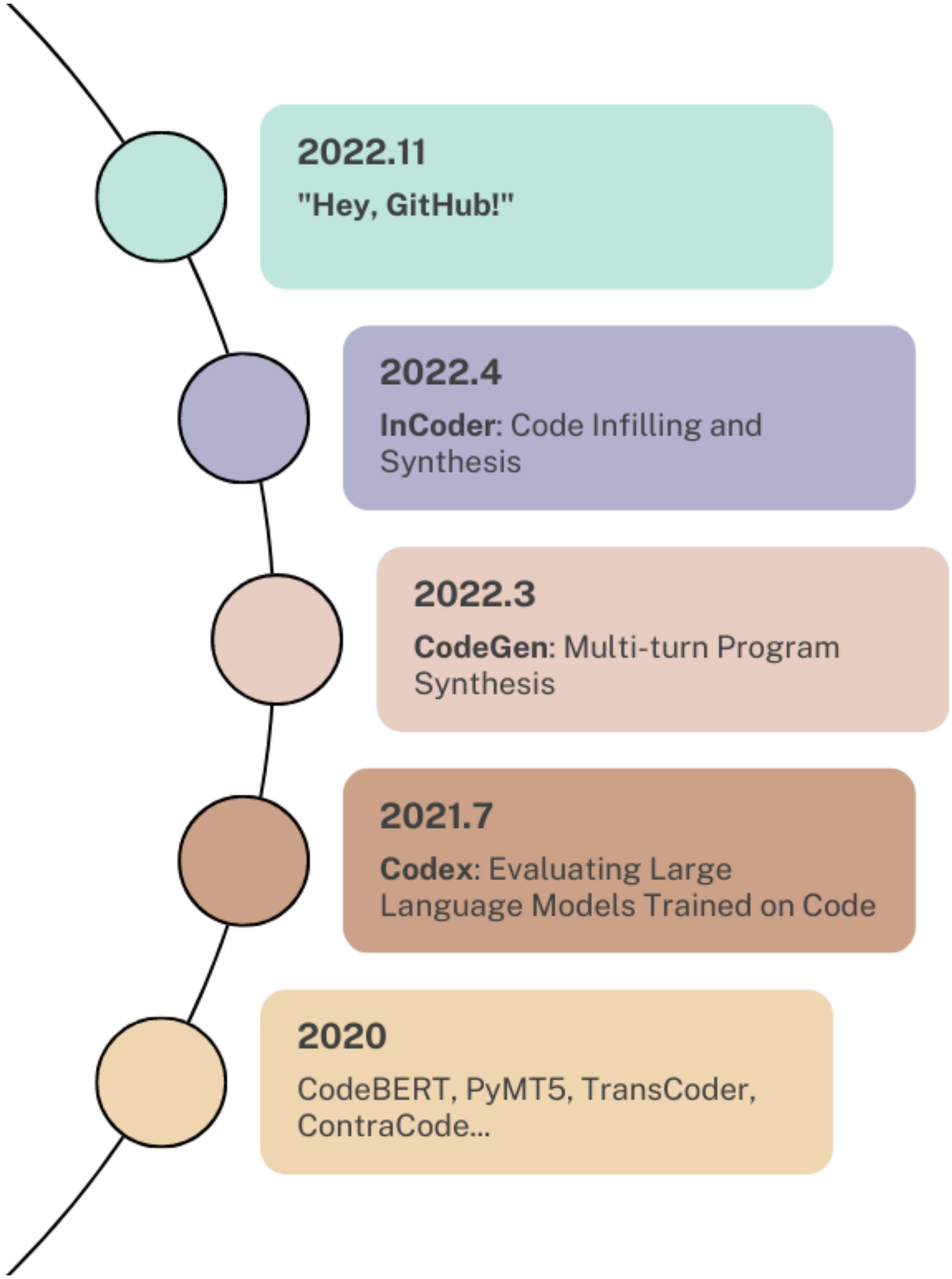
Codex

- Introduction
- Evaluation
- Methodology
- Experiments
- Discussions

InCoder

CodeGen

Codex for NLP



Outline

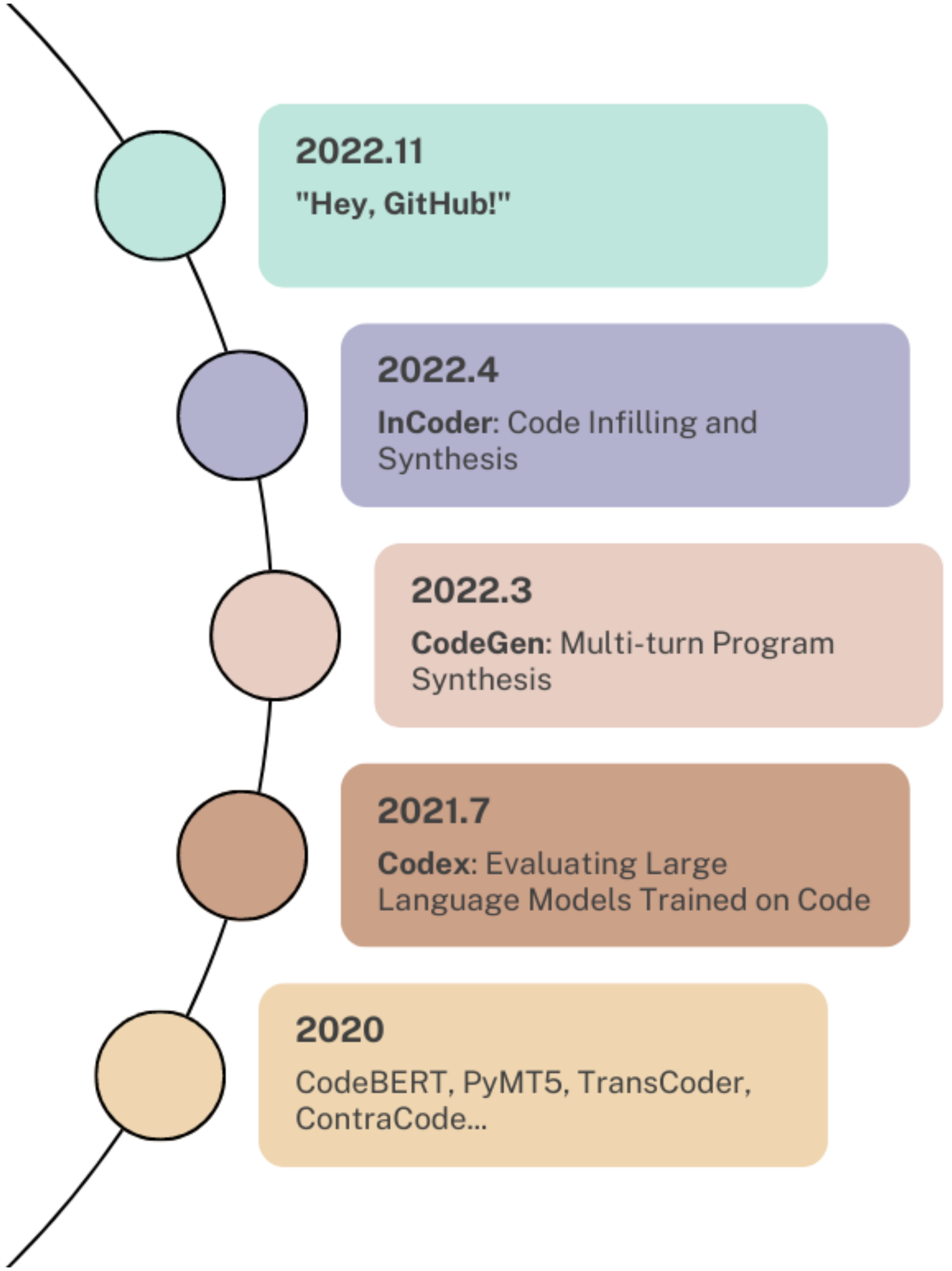
Codex

- **Introduction**
- Evaluation
- Methodology
- Experiments
- Discussions

InCoder

CodeGen

Codex for NLP



Brief overview

Brief overview

Task: generate code using LLMs

Given natural-language prompt (docstring), output the code that implements it

Brief overview

Task: generate code using LLMs
Given natural-language prompt (docstring), output the code that implements it

Model: Codex - a GPT-3, fine-tuned on code
up to 12B params

Brief overview

Task: generate code using LLMs
Given natural-language prompt (docstring), output the code that implements it

Model: Codex - a GPT-3, fine-tuned on code
up to 12B params

Training data: 160GB of Python code

Brief overview

Task: generate code using LLMs
Given natural-language prompt (docstring), output the code that implements it

Model: Codex - a GPT-3, fine-tuned on code
up to 12B params

Training data: 160GB of Python code

Evaluation: HumanEval
a novel dataset with 164 programming problems created by the authors
generate k samples from the model to see if at least one sample passes all the unit tests

Brief overview

Task: generate code using LLMs
Given natural-language prompt (docstring), output the code that implements it

Model: Codex - a GPT-3, fine-tuned on code
up to 12B params

Training data: 160GB of Python code

Evaluation: HumanEval
a novel dataset with 164 programming problems created by the authors
generate k samples from the model to see if at least one sample passes all the unit tests

Result: Codex-12B “solves” 72.3% of the problems (given 100 samples)
GPT-3 solves 0%, GPT-J solves 27.7%
if using only one sample (with lowest perplexity) we get 28.8% for Codex, 11.6% for GPT-J

Some examples

Some 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)
```

Some 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 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)
```

Some 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 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)
```

Real-life examples

```python
import time
s_t = time.time()
if self.use_amp:
    self.scaler.scale(loss).backward()
elif self.use_apex:
    with amp.scale_loss(loss, self.optimizer) as scaled_loss:
        scaled_loss.backward()
elif self.deepspeed:
    # loss gets scaled under gradient_accumulation_steps in deepspeed
    loss = self.deepspeed.backward(loss)
else:
    loss.backward()
e_t = time.time()

print("Backward time: ", e_t - s_t)
```

```python
from collections import defaultdict
asins = set()
cnt_cats = defaultdict(int)
cnt_atts = defaultdict(int)
cnt_catatts = defaultdict(lambda: defaultdict(int))
for goal in env.server.goals:
    if goal['asin'] not in asins:
        asins.add(goal['asin'])
        cnt_cats[goal['category']] += 1
        for att in goal['attributes']:
            cnt_atts[att] += 1
            cnt_catatts[goal['category']][att] += 1

cnt_cats
```

Outline

Codex

Introduction

Evaluation

Methodology

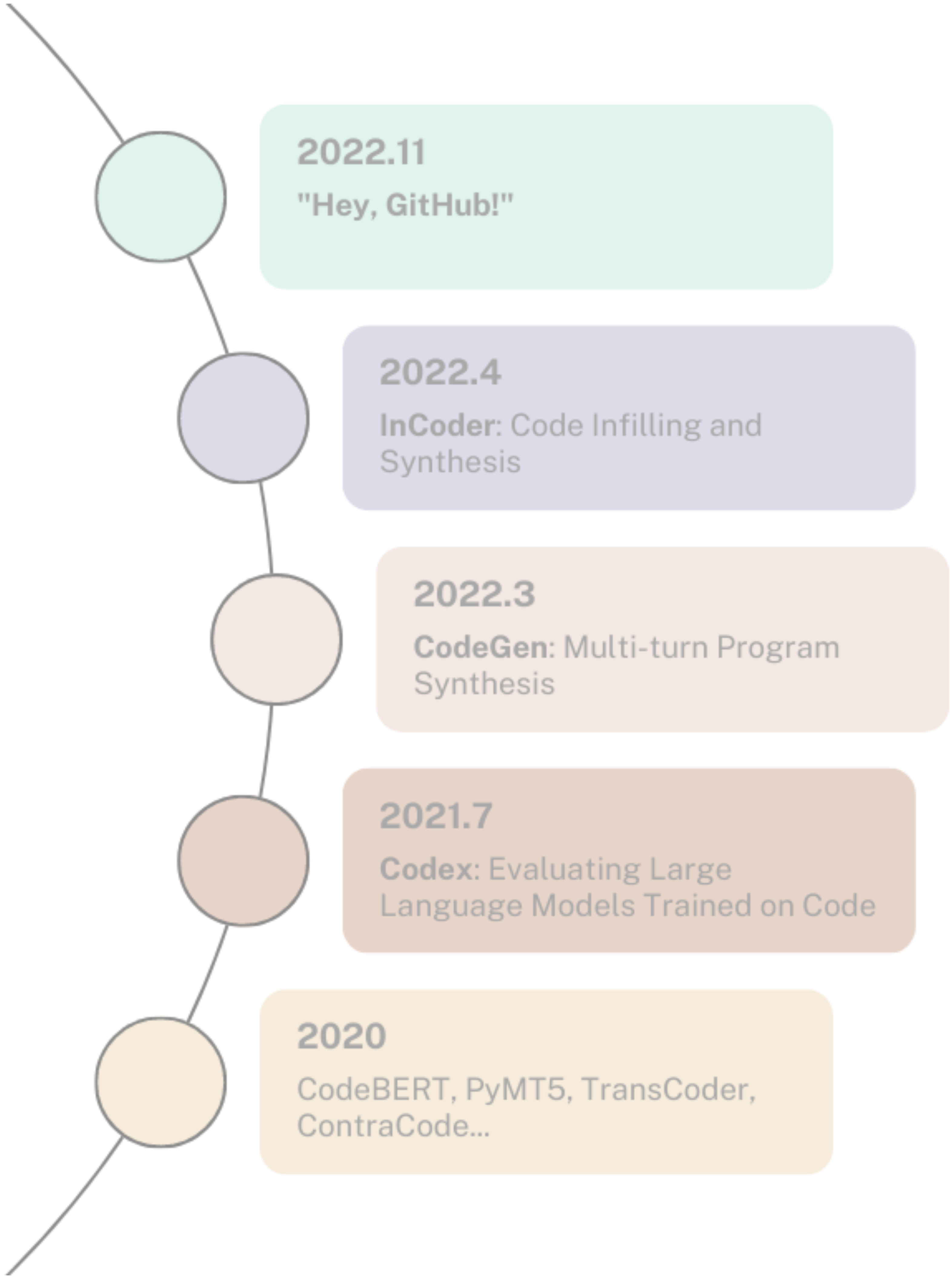
Experiments

Discussions

InCoder

CodeGen

Codex for NLP



HumanEval

164 hand-written problems

- hand-writing to avoid repeating the problems in the training data (“training data leakage”)

Evaluates language comprehension, reasoning, algorithms and simple math

HE was used in later papers (CodeGen, InCoder)

HumanEval

164 hand-written problems

- hand-writing to avoid repeating the problems in the training data ("training data leakage")

Evaluates language comprehension, reasoning, algorithms and simple math

HE was used in later papers (CodeGen, InCoder)

“Check if two words have the same characters.”

“Return median of elements in the list l.”

“sum_to_n is a function that sums numbers from 1 to n.”

“Given a non-empty list of integers lst. add the even elements that are at odd indices.”

“Return true if a given number is prime, and false otherwise.

“Return n-th Fibonacci number.”

HumanEval: Format

Format:

- function signature
- docstring with examples
- unit-tests

```
def vowels_count(s):
    """Write a function vowels_count which takes a
        string representing
    a word as input and returns the number of vowels in
         the string.
    Vowels in this case are ’a’, ’e’, ’i’, ’o’, ’u’.
        Here, ’y’ is also a
    vowel, but only when it is at the end of the given
        word.

    Example:
    >>> vowels_count("abcde")
    2
    >>> vowels_count("ACEDY")
    3
    """
```

```
def check(candidate):
     # Check some simple cases assert candidate("abcde")
     == 2, "Test 1" assert candidate("Alone") == 3, "Test
     2" assert candidate("key") == 2, "Test 3" assert
     candidate("bye") == 1, "Test 4" assert
     candidate("keY") == 2, "Test 5" assert
     candidate("bYe") == 1, "Test 6" assert
     candidate("ACEDY") == 3, "Test 7"
     # Check some edge cases that are easy to work out by
     hand. assert True, "This prints if this assert fails
     2 (also good for debugging!)"
```

HumanEval: Metric

Functional correctness:

Whether the generated code implements the correct function

I.e. passes all unit tests

This is the way humans evaluate correctness of the code

BLEU score doesn't work

optimized for the semantics of text

HumanEval: Metric

Functional correctness:

Whether the generated code implements the correct function

I.e. passes all unit tests

This is the way humans evaluate correctness of the code

BLEU score doesn't work

optimized for the semantics of text

Reference code

```
def f(a, b):
    c = a - b
    return c
```

Equivalent code

```
def f(a, b):
    summ = 0
    summ += a
    summ -= b
    return summ
```

BLEU = 66

Non-equivalent code

```
def f(a, b):
    c = a + b
    return c
```

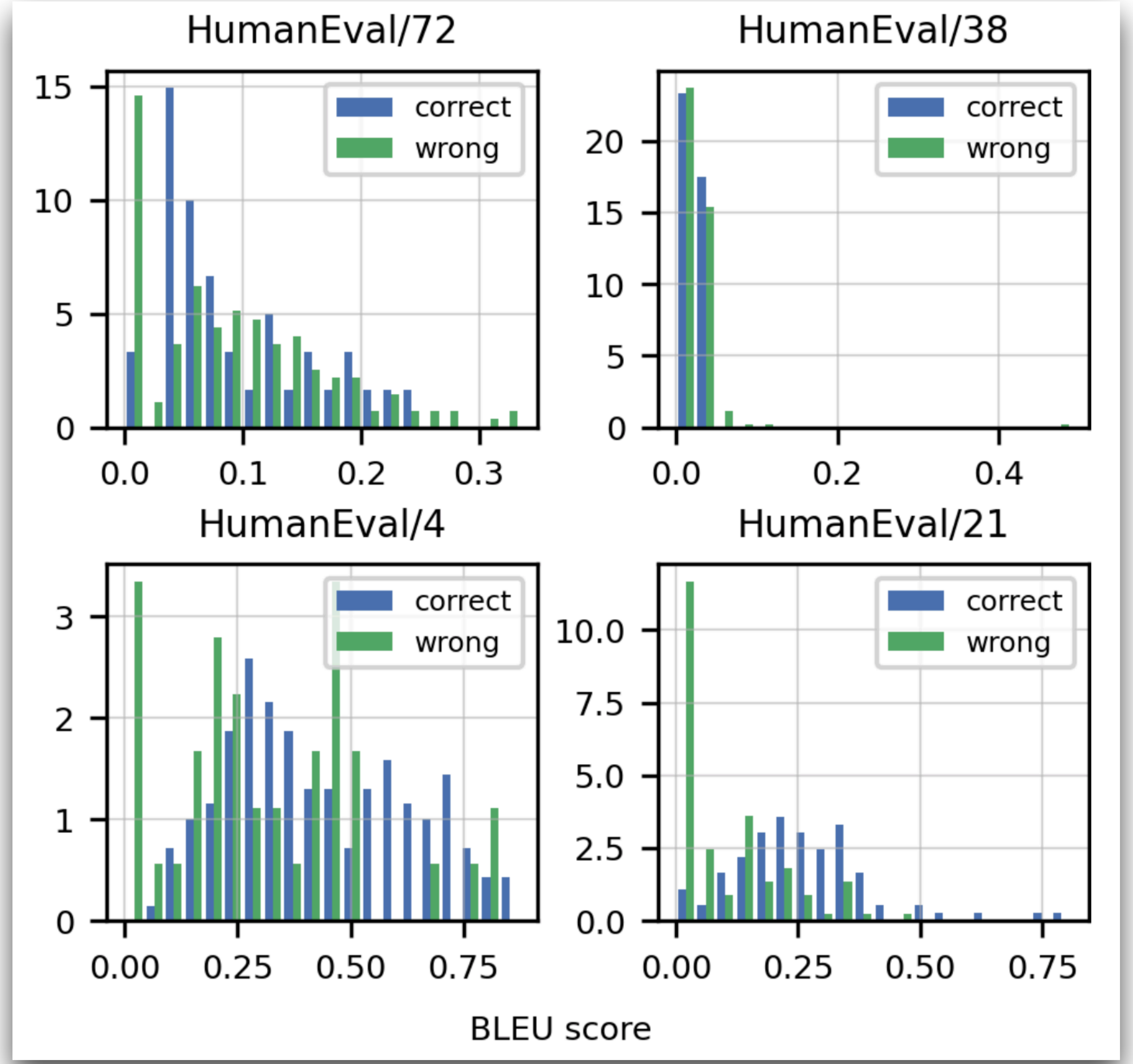
BLEU = 81

BLEU Score

BLEU score doesn't work:

- Algorithmic difference
- Variable name
- Operation orders

Optimizing for BLEU score is ***not*** equivalent to optimizing functional correctness



Q1: For evaluating code generation, why is functional correctness better than match-based metrics (e.g., BLEU)?

Existing match-based metrics are designed for comparing natural languages, which is <u>not</u> inherently <u>applicable to code</u>. In particular, when evaluating code, the aspect that matters is its <u>correct behavior</u>. One can use unit tests to check this correctness (with large likelihood).

"Perhaps the most convincing reason to evaluate functional correctness is that it is used by human developers to judge code."

In a sense, this makes evaluation of code generation more "precise" than evaluation of text generation.

Recent research (Ren et al.) showed that BLEU score doesn't capture the semantic features specific to code. Aforementioned experiment result corroborate that BLEU score and the correctness of generated code are not equivalent.

HumanEval: Sampling & pass@k

Given a prompt, generate *k* samples

a sample is generated until a stop sequence is encountered

pass@k:

having k generations per problem, a problem is “solved” if at least one generation passes all unit tests

total fraction of problems “solved” is reported

HumanEval: pass@k

Estimating pass@k:

Naively: high variance for small k

Instead:

Generate n samples ($n \geq k$)

Use the following unbiased estimator

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

where c is the #(correct samples)

HumanEval: Potential Shortcomings

Small dataset

large variance when comparing different models

Most of the tasks are "short" - could be solved in less than 10 lines of code

Data leakage

the solutions to the problems might already be present in the training data

e.g.: primality

```
### 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))
```

Outline

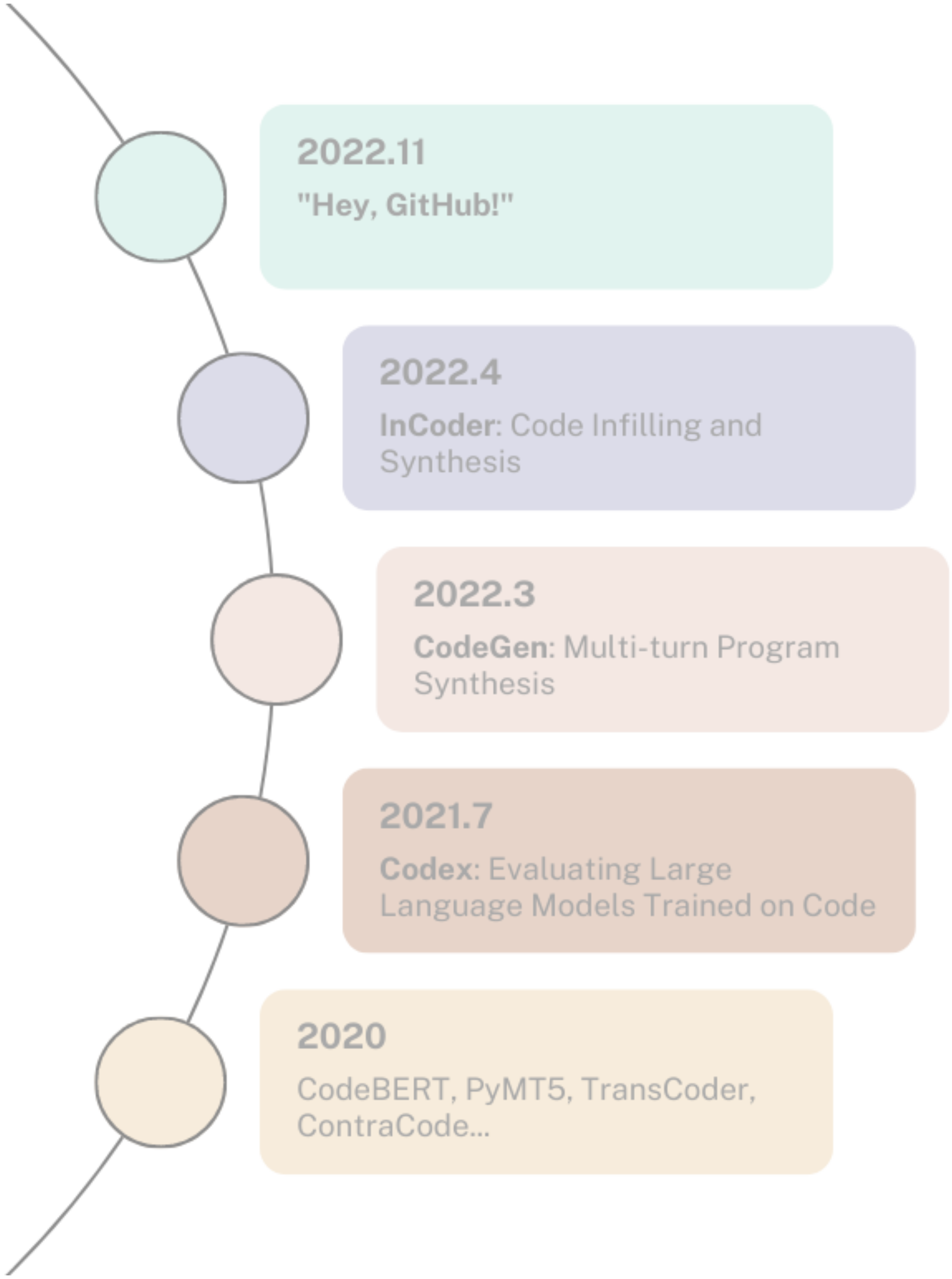
Codex

- Introduction
- Evaluation
- **Methodology**
- Experiments
- Discussions

InCoder

CodeGen

Codex for NLP



Model + training data

Model: GPT-3
    sizes: 12M - 12B

Data: Python code from Github - 159GB

Method: fine-tune on code
    use next token prediction

no difference between fine-tuning GPT-3 and training from scratch on code
    yet, faster convergence when fine-tuning

Codex-S

Motivation: potential **distribution mismatch** between GitHub files and HumanEval / APPS problems

Solution: fine-tune Codex on correct ***standalone functions***

Codex-S: Training Data

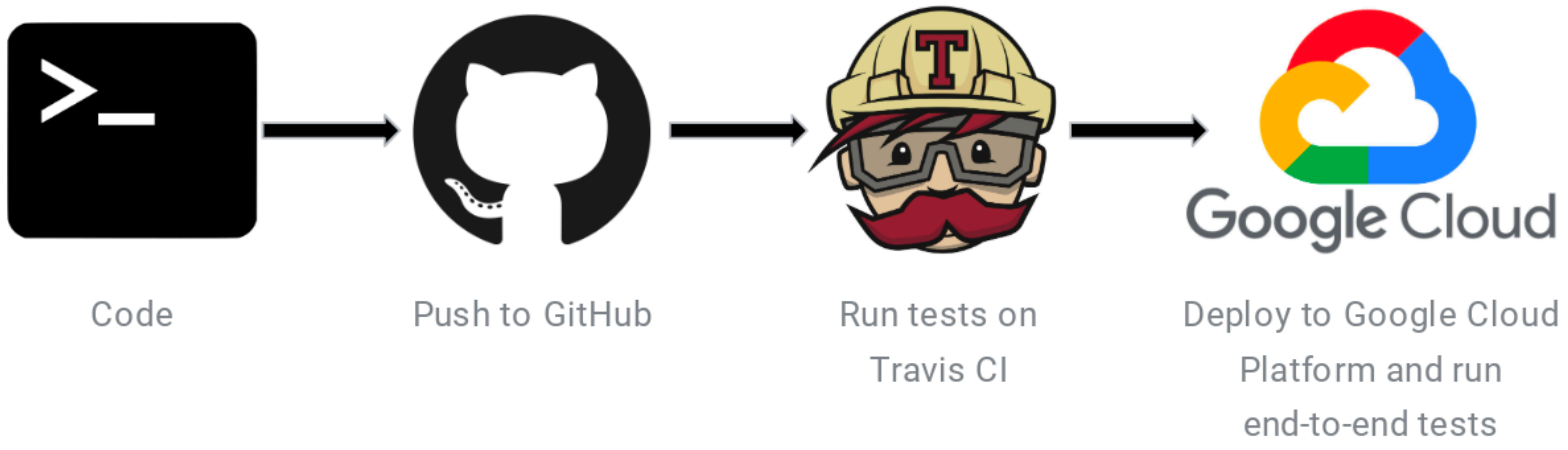
Competitive Programming (10,000 problems)

- Self-contained
- Unit test coverage
- Problem descriptions as docstrings

Continuous Integration (40,000 functions)

*"Developers regularly merge code changes into a central repository, after which **automated builds and tests** are run."*

- Open source
- Tracing test functions



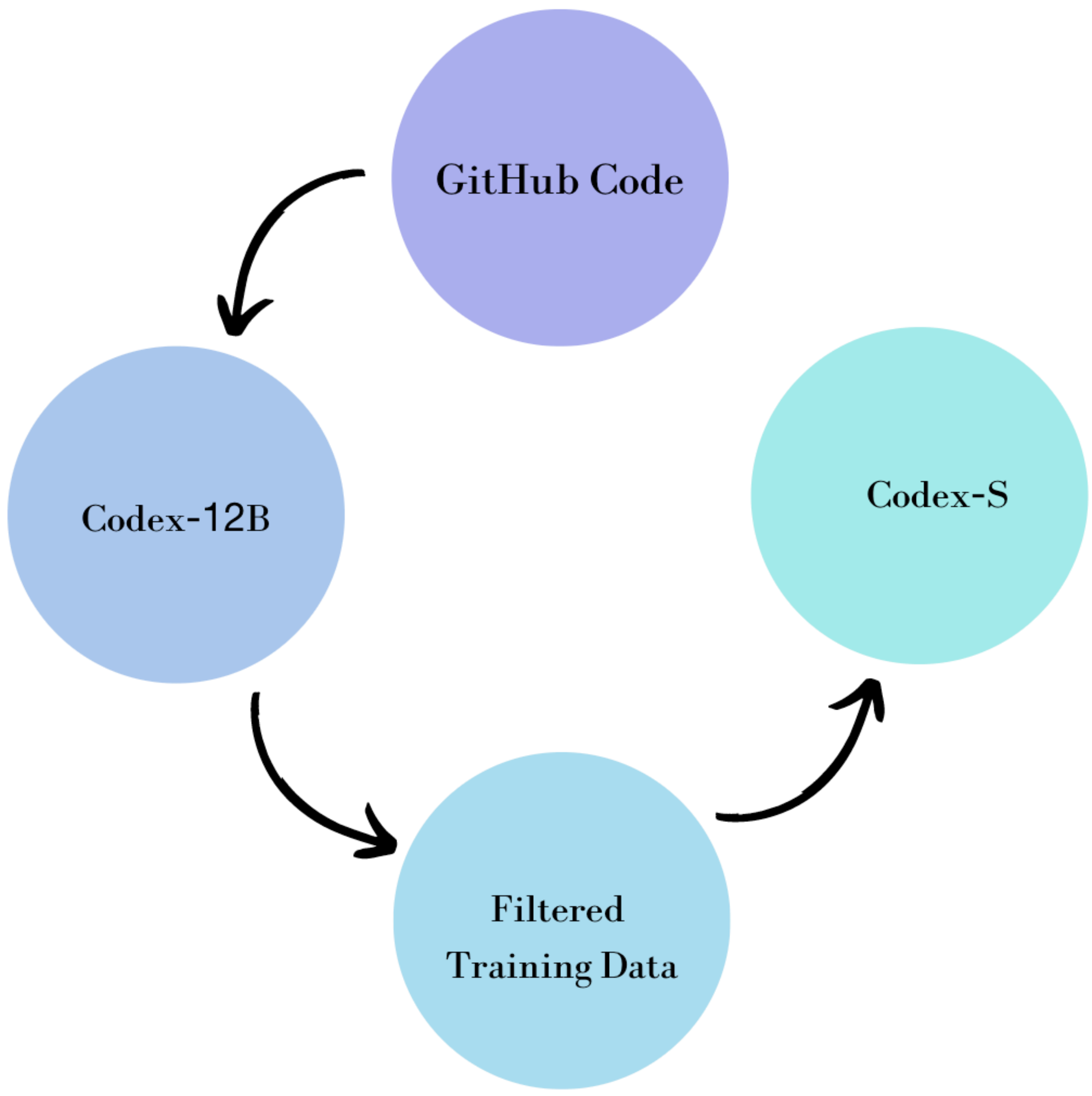
Codex-S: Filtering

Concern:

- Low-quality docstring
- Stateful functions

Solution:

Use Codex-12B to generate 100 samples per problem, discard the problem if no generation passes



Codex-S: Tuning

Training examples assembled into the same format as in *pass@k* evaluation

```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)
```

Objective: minimize negative log-likelihood of reference solution, mask out loss for prompt

Docstring Generation

docstring → code

code → docstring

Motivation: useful for safety concerns

Training Dataset: same as Codex-S dataset

Training Objective: Minimize negative log-likelihood of docstring

Evaluation: manually evaluate 10 samples per problem, creating *pass@1* and *pass@10*

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

Docstring Generation

docstring → code

code → docstring

Examples of generated docstrings:

- "I just found this function online"
- "This test is not correctly written and it's not my solution."

Outline

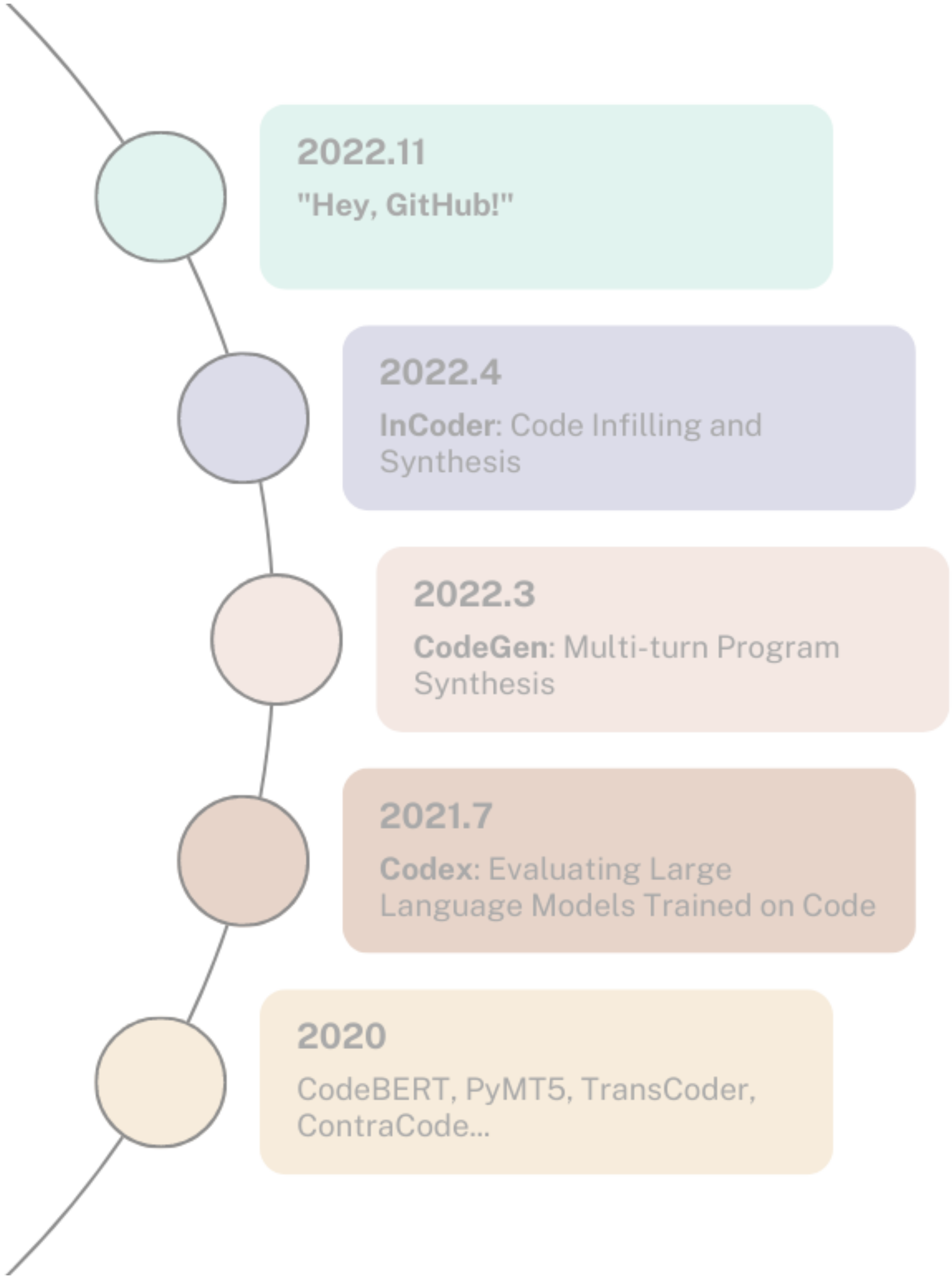
Codex

- Introduction
- Evaluation
- Methodology
- **Experiments**
- Discussions

InCoder

CodeGen

Codex for NLP



Generation Example

Codex-12B

Code generation

The following sample problem is taken from HumanEval

8 random samples are generated at temperature 0.8

Prompt

```
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
    """
```

Generated samples

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

check if n is divisible by i

Fails for n = 1 (returns True)

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

exactly the same as COMPLETION 1

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

Fails for all unit test cases

Generated samples

```
### 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))
```

handle n=1,2 cases

even numbers not prime

check if divisible by odd number up to $\sqrt{n}$

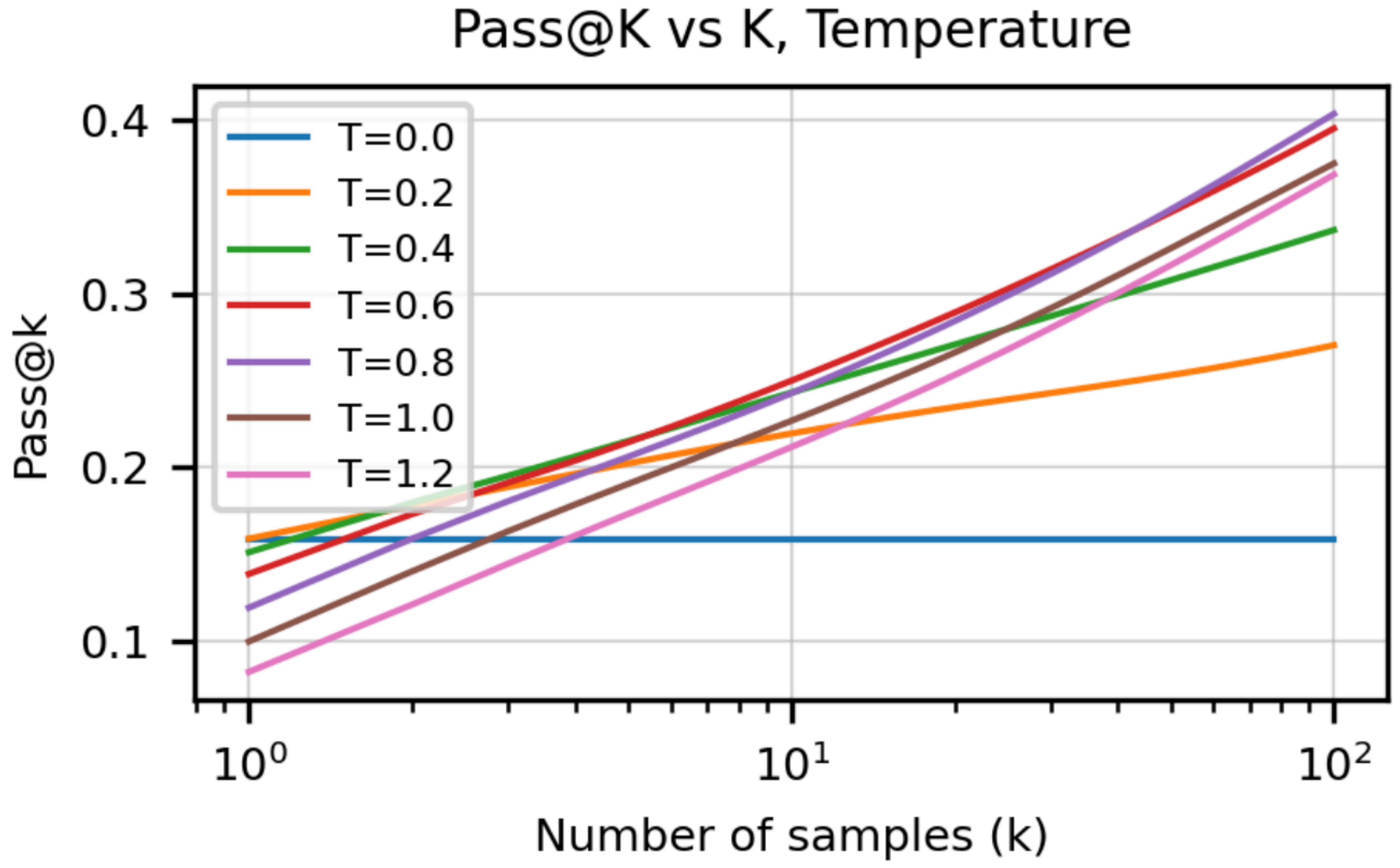
```
## COMPLETION 5 (WRONG): ###
def helper(n):
    # check if an odd number is multiple of any
      integer up to the square root of the number
    # if this is true, n is not prime
    for i in range(2, int(n**0.5 + 1)):
        if n % i == 0:
            return False
    return True
# if helper returns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

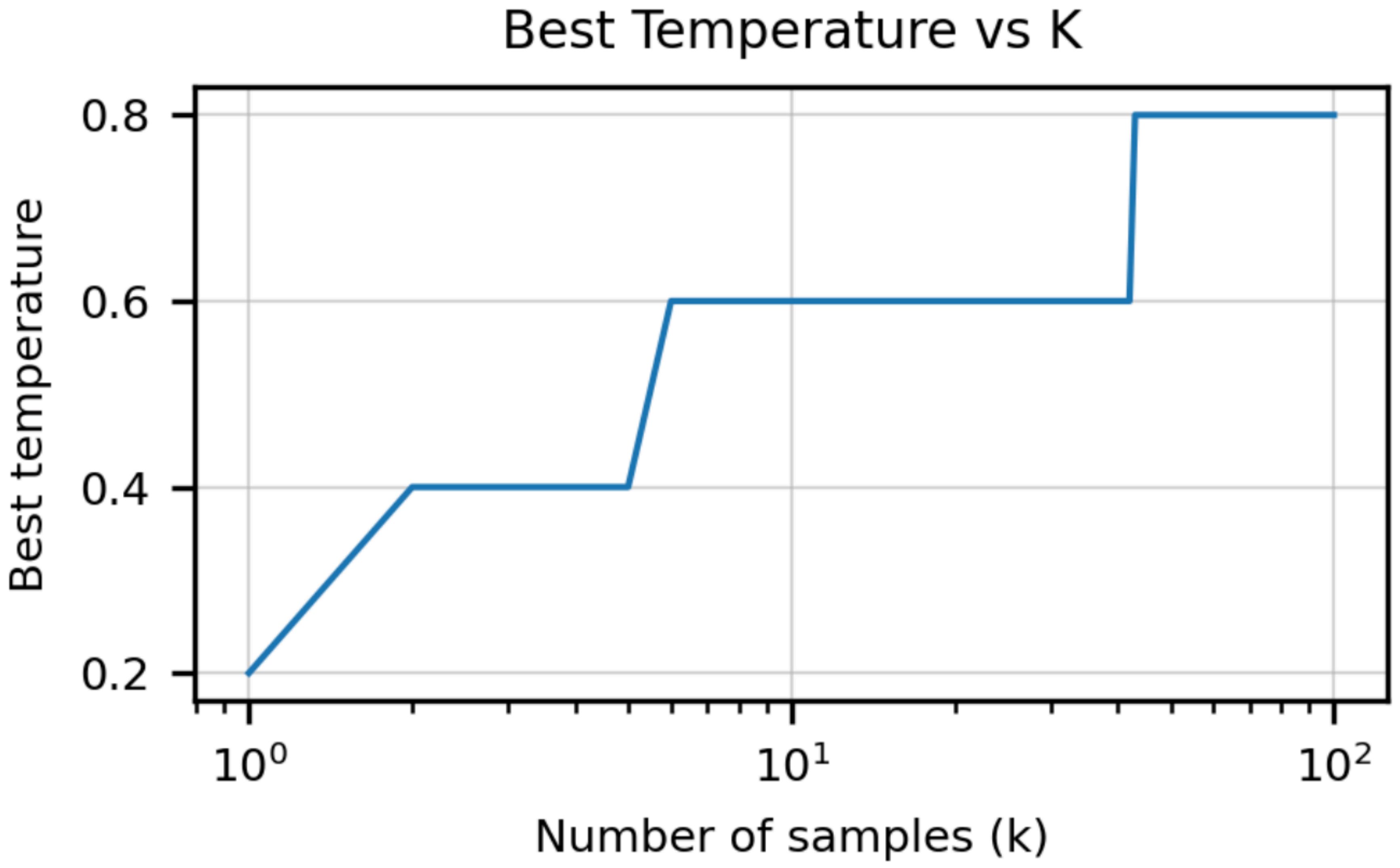
Fails for n = 1 (returns True)

Sampling Temperature

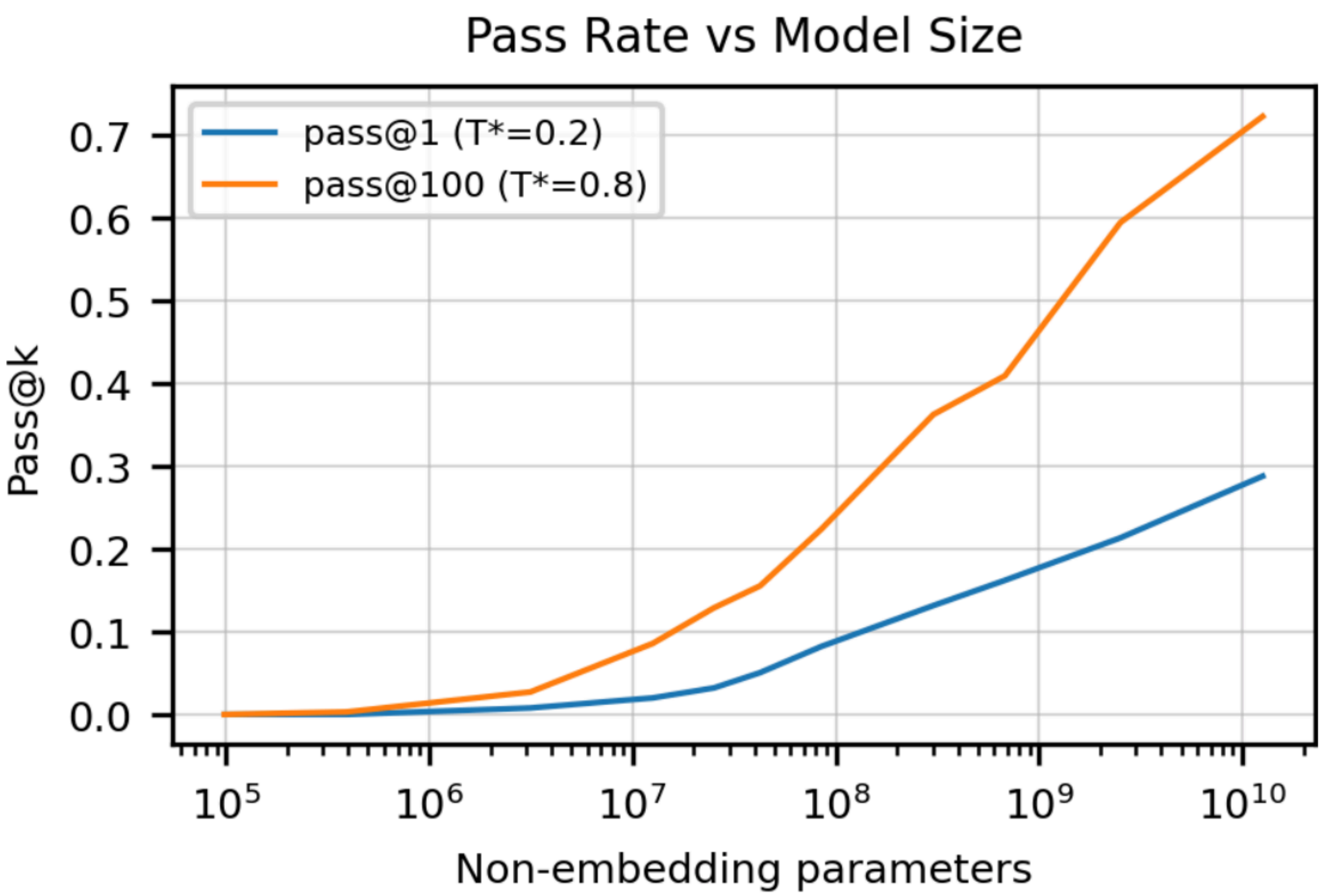
Temperature experiment on Codex-679M

larger k benefits from higher temperature
(i.e. higher diversity)



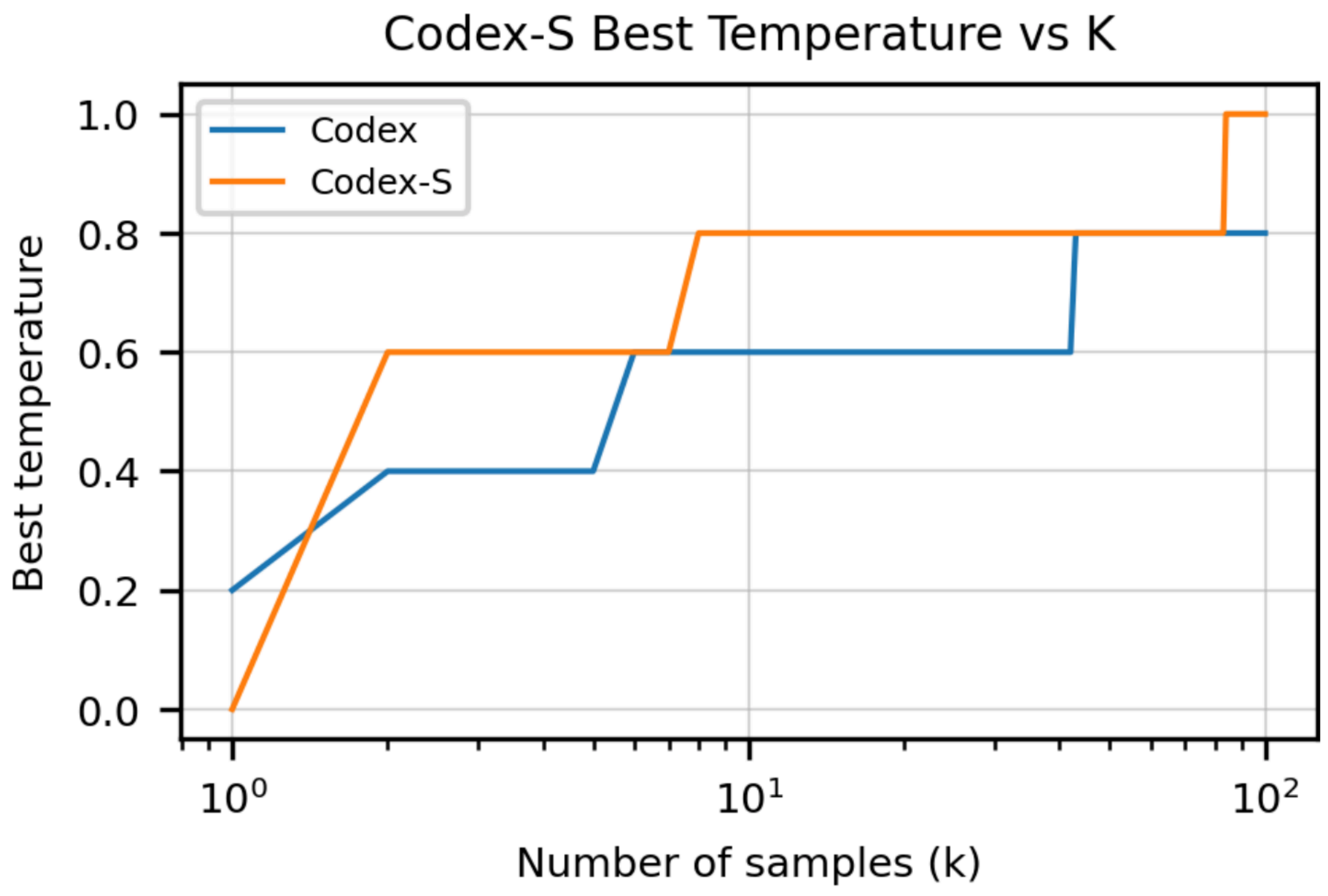


Sampling Temperature



Sampling Temperature

Codex-S prefers ***higher temperature*** compared to Codex



Q2: Why does temperature term matter for code generation and what is the relationship between temperature and *k*? What ranking heuristic works best?

Temperature

Temperature controls the variance of generated samples, and higher k *prefers higher temperature. Since pass@k rewards only whether the model generated any correct result given* k *samples, higher temperature generates more diverse solutions and more likely one of them would pass.*

Codex-S prefers higher temperature than Codex. This could be explained as Codex-S is fine-tuned on data with matching distribution, so it's more "concentrated" than Codex.

Ranking Heuristic

Mean log probability works best.

Sampling Heuristics

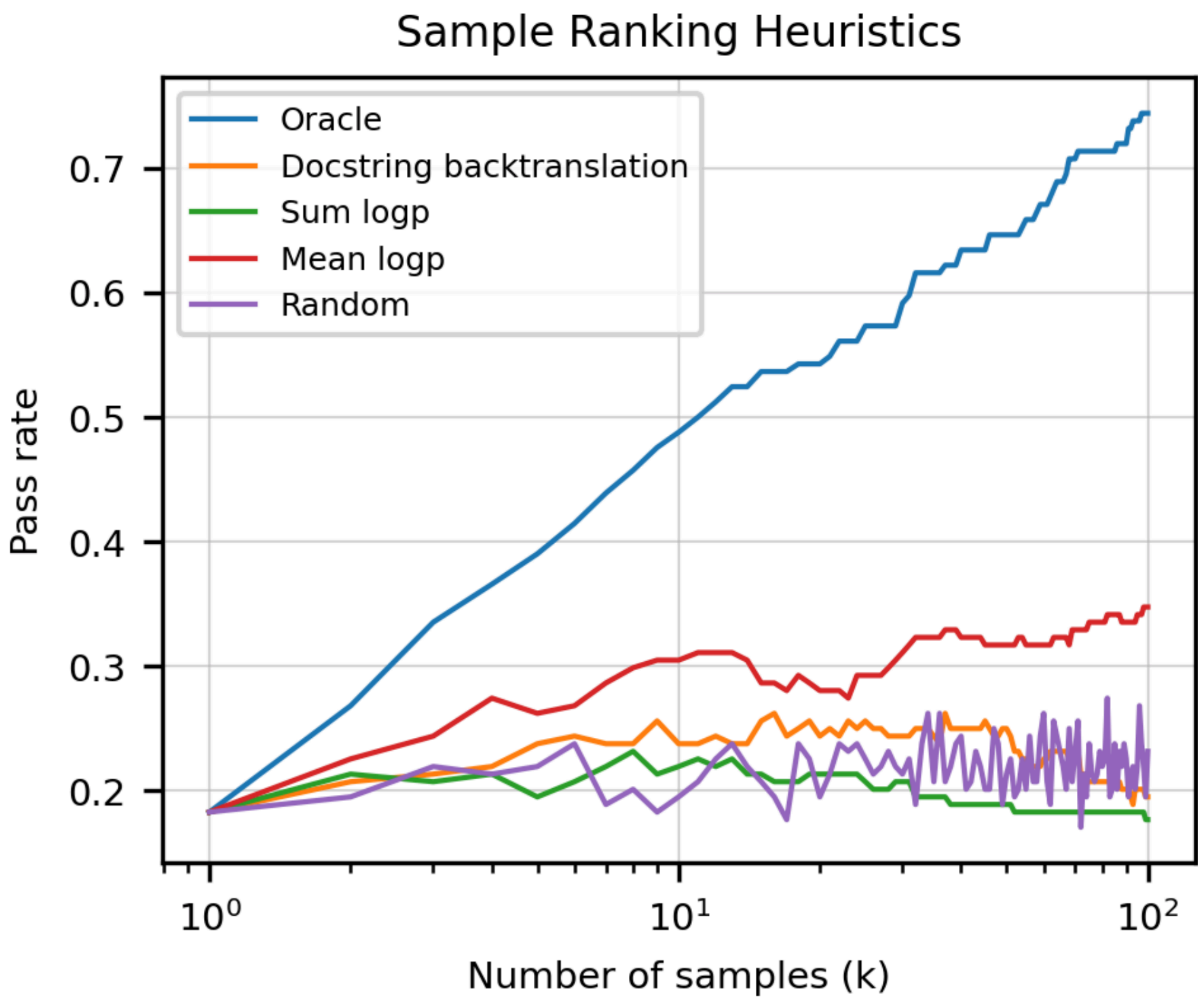
Oracle: best sample chosen as one that passes the unit tests

Back Translation: best sample maximizes P(ground truth docstring | generated sample)

Mean log probability

Sum log probability

Random



Codex-12B, temp = 0.8

Sampling Heuristics

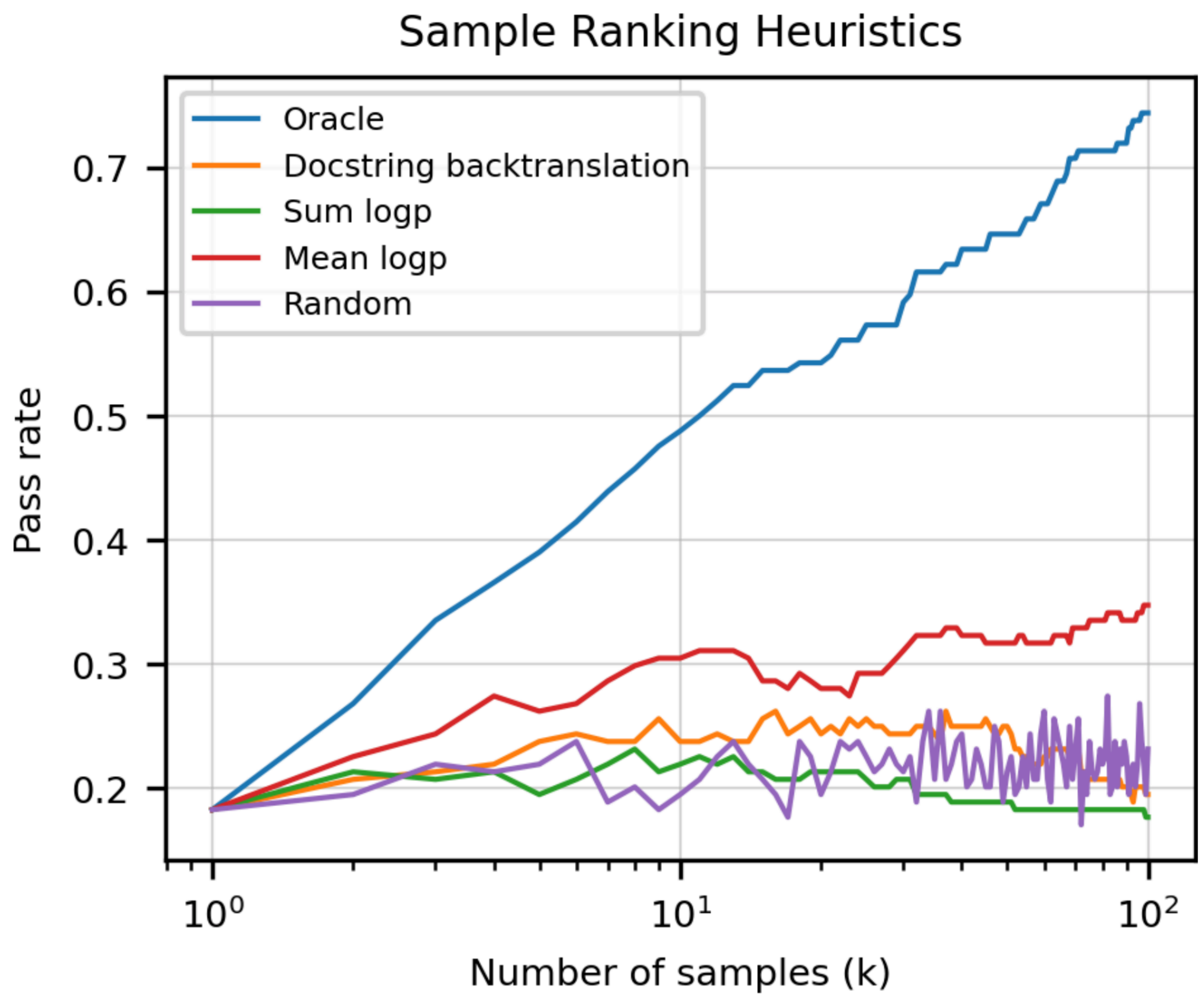
Oracle: best sample chosen as one that passes the unit tests

Back Translation: best sample maximizes P(ground truth docstring | generated sample)

Mean log probability

Sum log probability

Random



Codex-12B, temp = 0.8

Sampling Heuristics

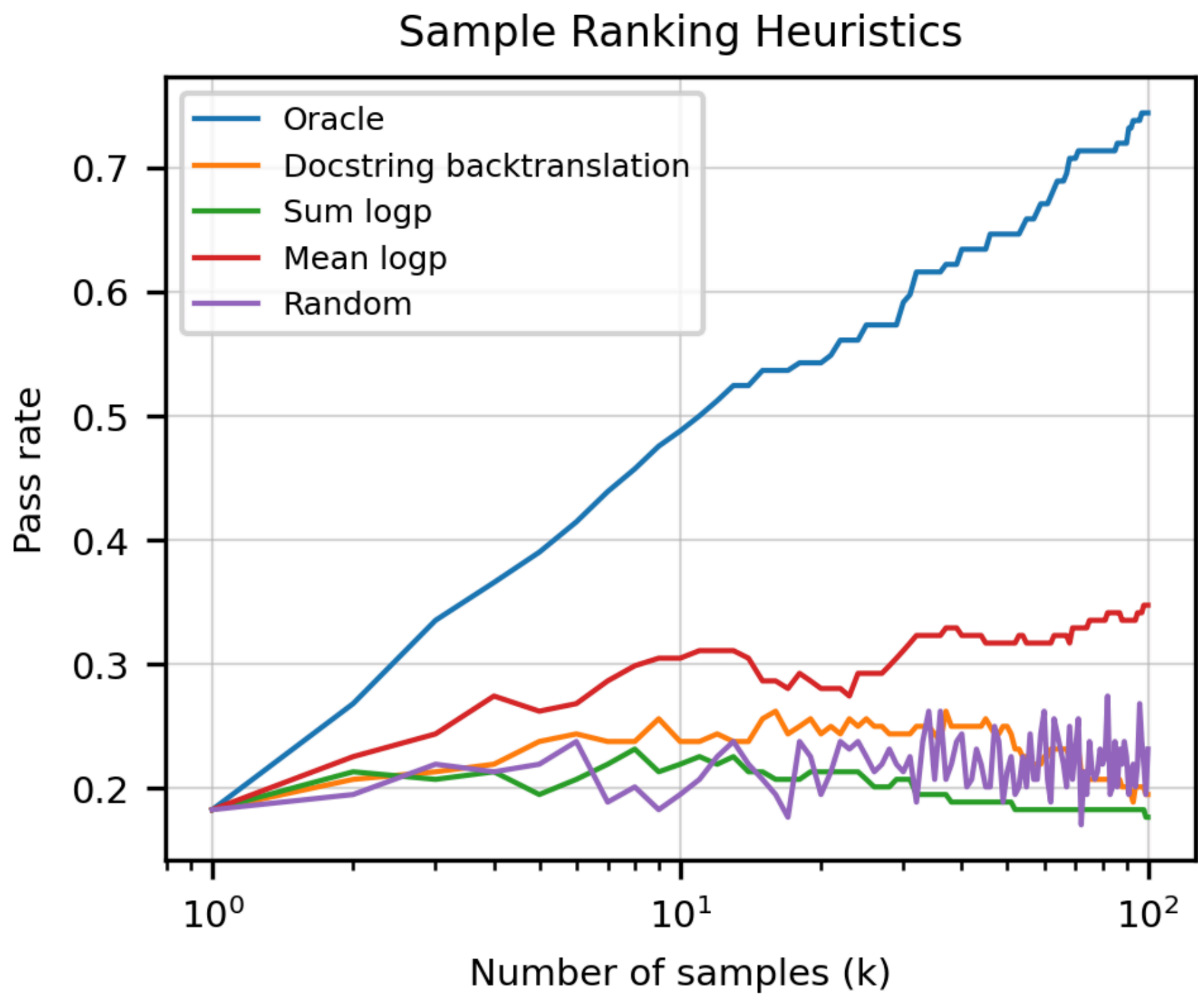
Oracle: best sample chosen as one that passes the unit tests

Back Translation: best sample maximizes P(ground truth docstring | generated sample)

Mean log probability

Sum log probability

Random



Codex-12B, temp = 0.8

Sampling Heuristics

Oracle: best sample chosen as one that passes the unit tests

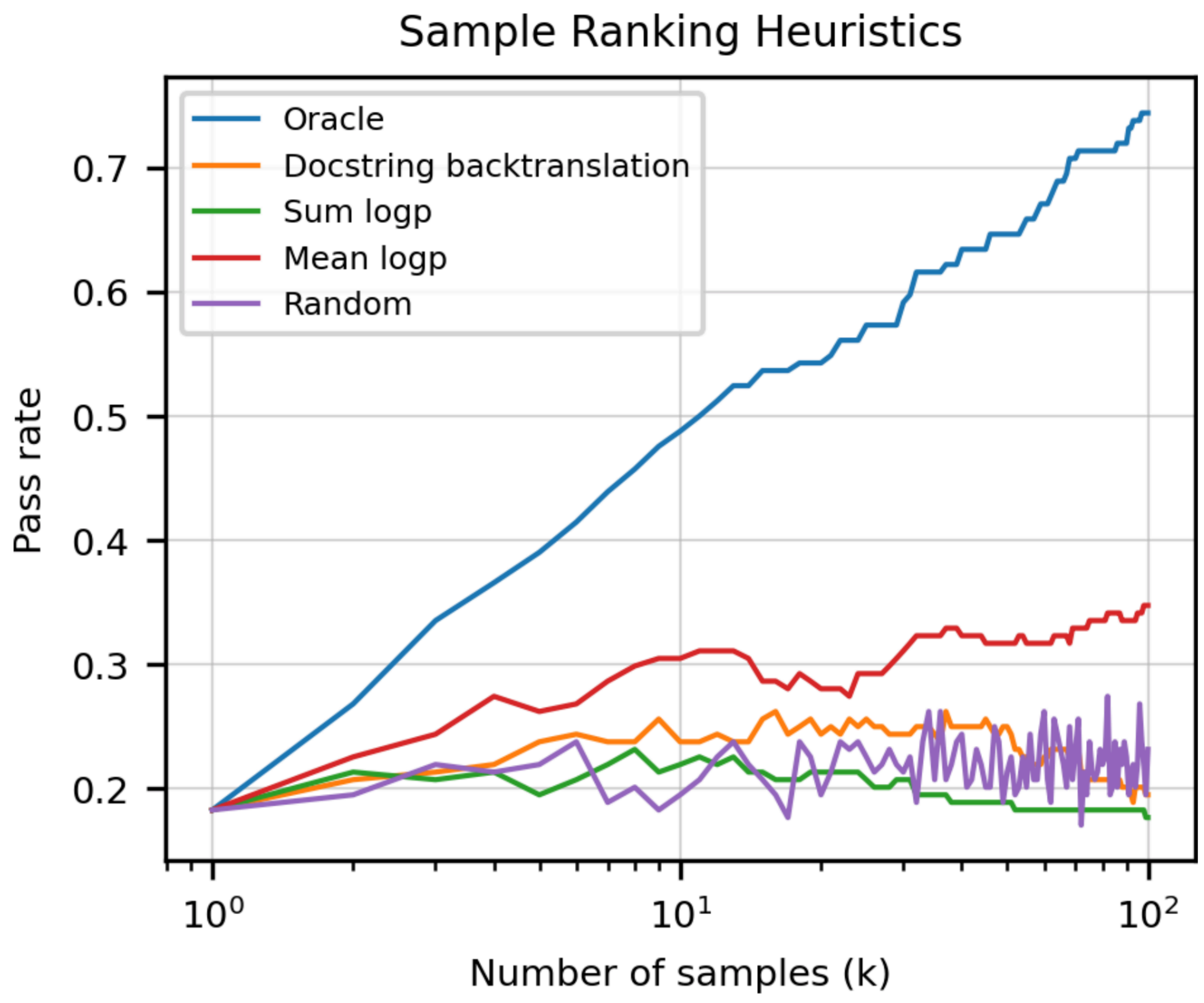
Back Translation: best sample maximizes P(ground truth docstring | generated sample)

Mean log probability

Sum log probability

Random

Codex-12B, temp = 0.8



Sampling Heuristics

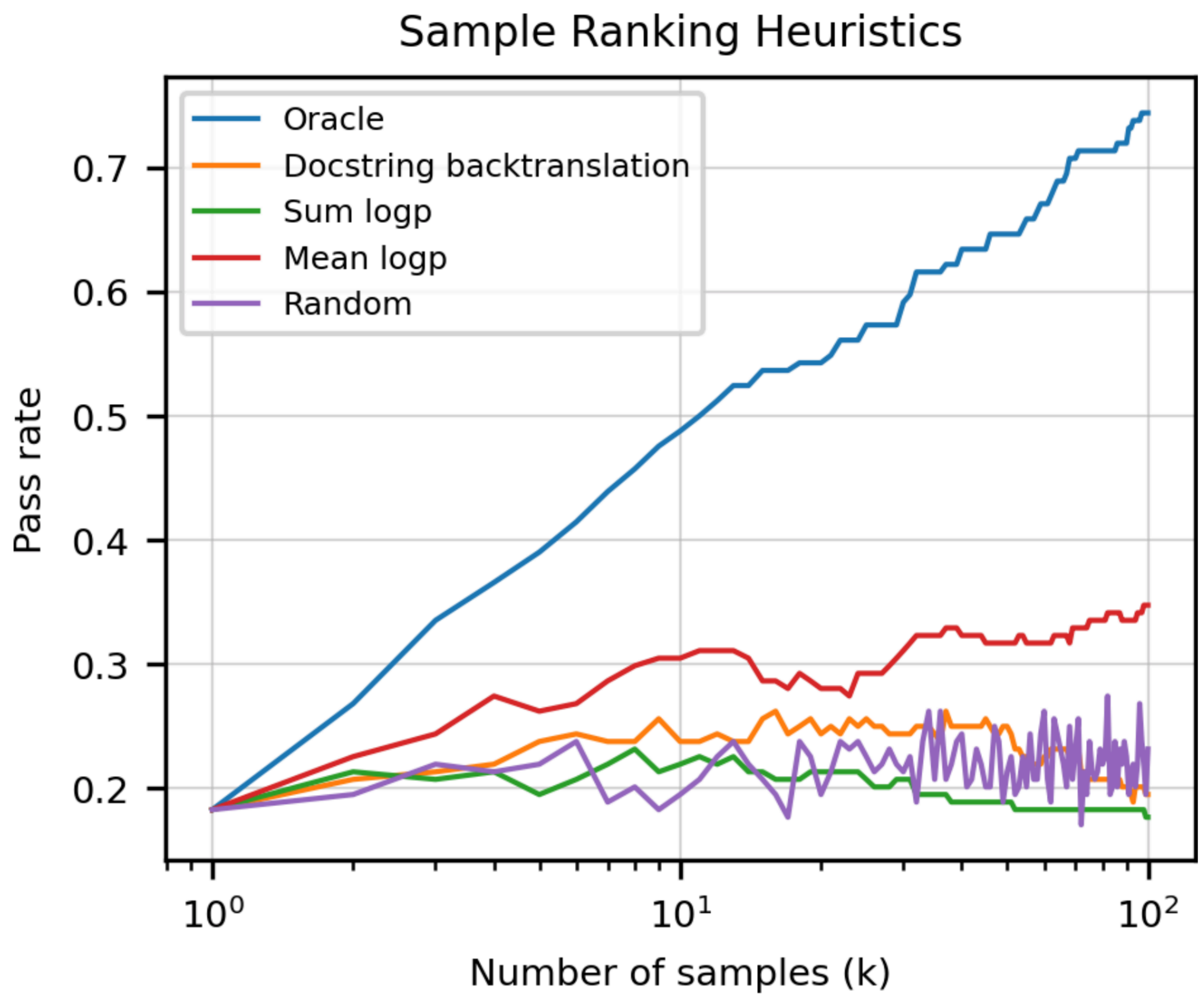
Oracle: best sample chosen as one that passes the unit tests

Back Translation: best sample maximizes P(ground truth docstring | generated sample)

Mean log probability

Sum log probability

Random



Codex-12B, temp = 0.8

Q2: Why does temperature term matter for code generation and what is the relationship between temperature and *k*? What ranking heuristic works best?

Temperature

Temperature controls the variance of generated samples, and higher k prefers higher temperature. Since pass@k rewards only whether the model generated any correct result given k samples, higher temperature generates more diverse solutions and more likely one of them would pass.

Codex-S prefers higher temperature than Codex. This could be explained as Codex-S is fine-tuned on data with matching distribution, so it's more "concentrated" than Codex.

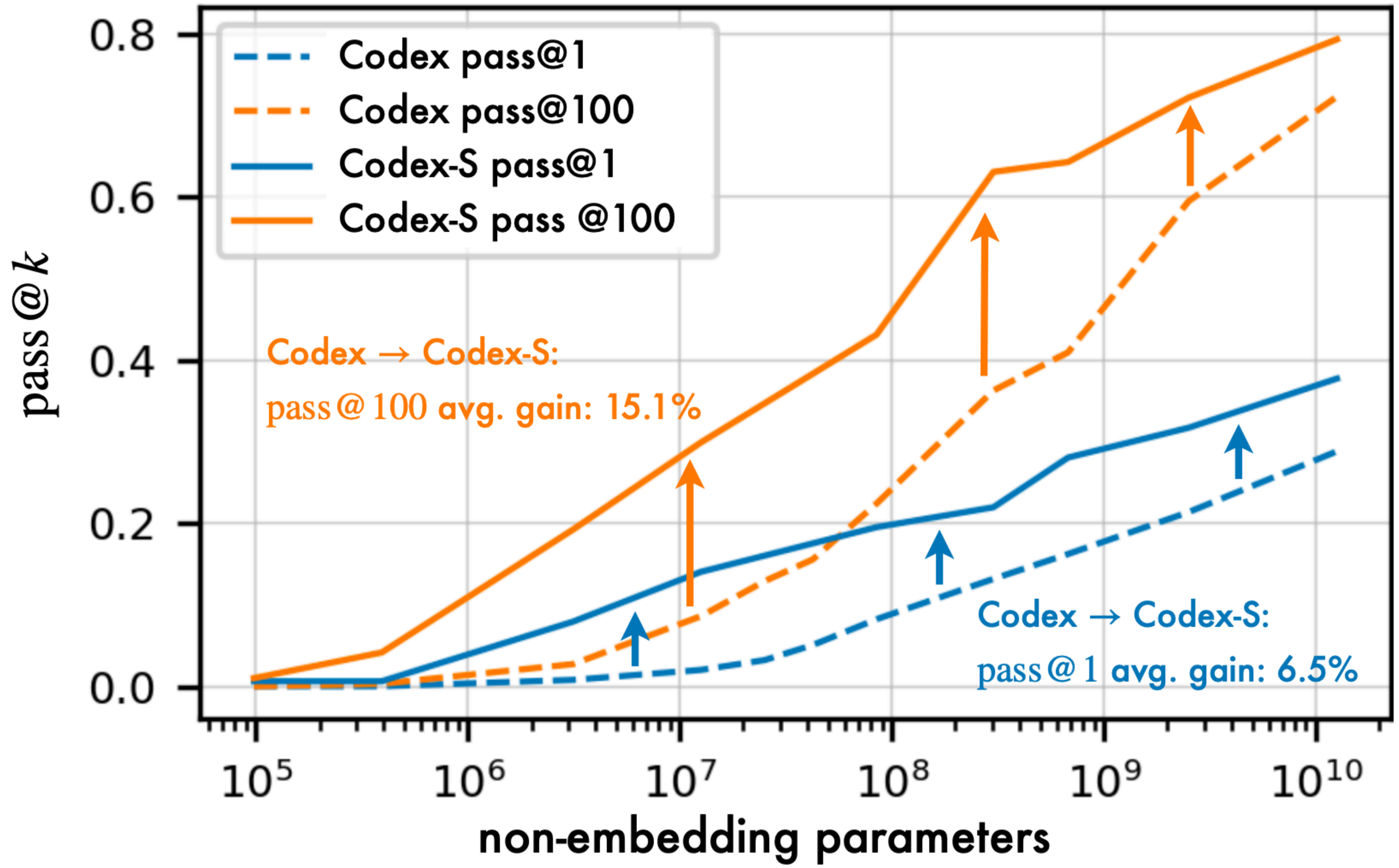
Ranking Heuristic

Mean log probability works best.

Better decoding heuristics?

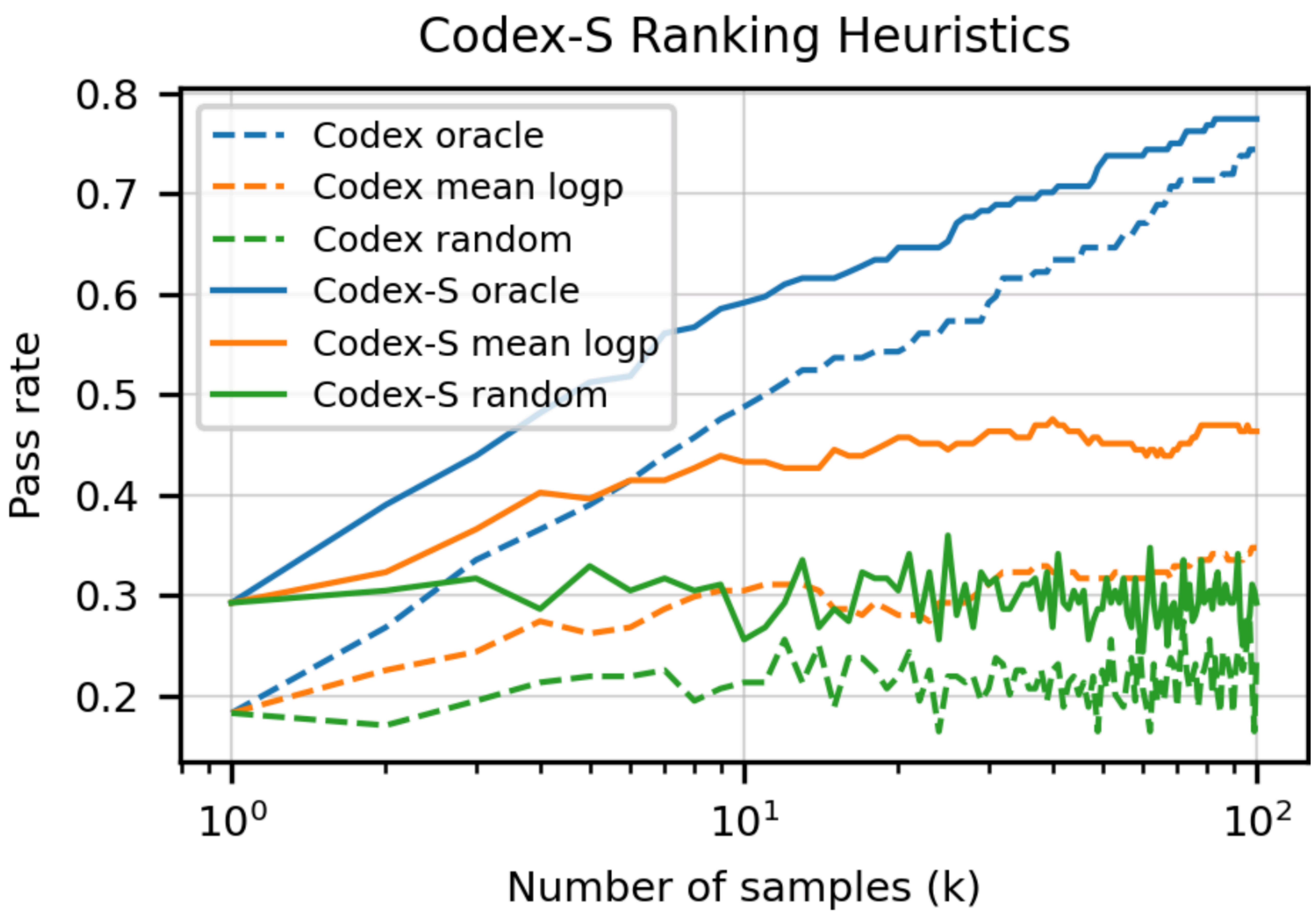
Results

Codex-S outperforms Codex on HumanEval



Results

Codex-S outperforms Codex on HumanEval



Code Model Comparison

The Pile: 8% GitHub code, along with natural language data

GPT-J and GPT-Neo: similar architecture

TabNine: Code Autocomplete as a service

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

Sources: coding websites such as Codeforces, Kattis, etc.

Difficulty Level:

1. Introductory
2. Interview
3. Competition

Distribution:

5000 training set and 5000 test set

Train set:

52% Introductory, 40% Interview, 8% Competition

Test set:

20% Introductory, 60% Interview, 20% Competition

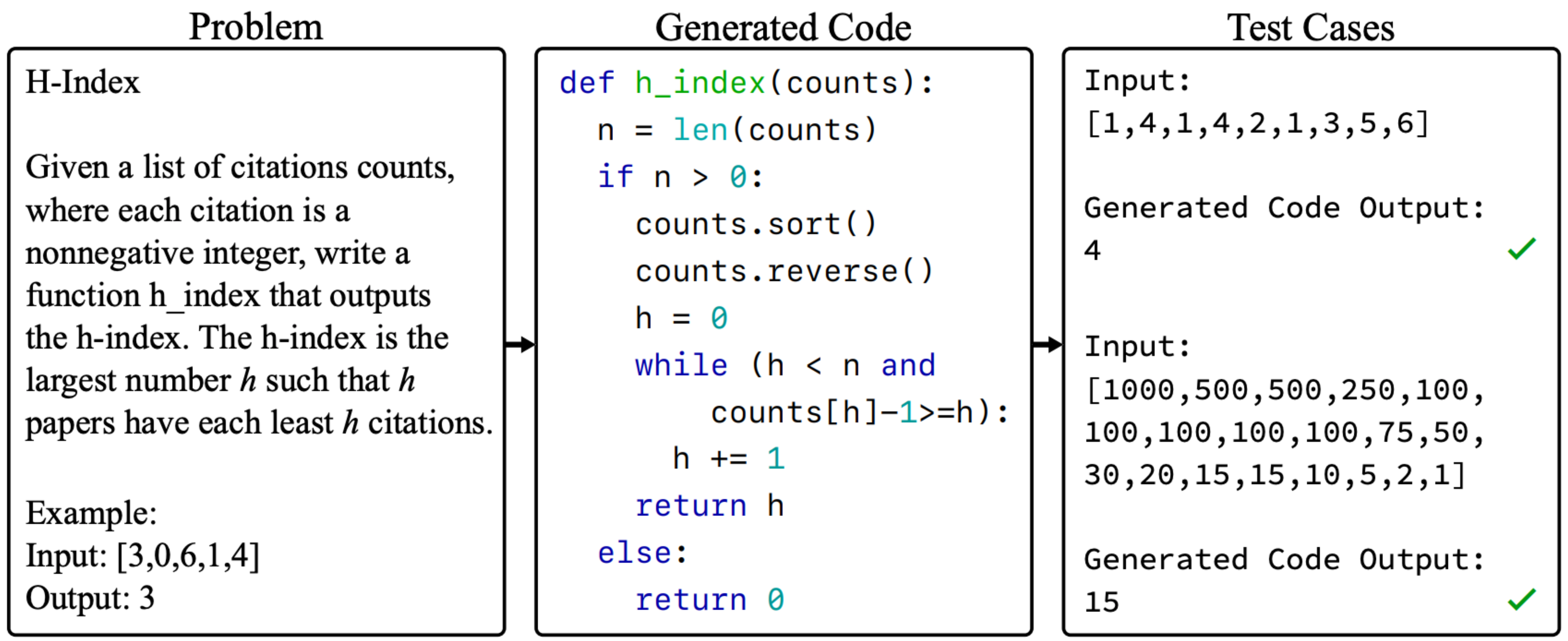


Figure 1: An example "interview"-level problem from APPS (left) along with possible generated code (middle) and two example test cases we use to evaluate the generated code (right). Our evaluation framework has test cases and 10,000 code generation problems of varying difficulty levels.

APPS

Raw Pass@k: calculated as before

Filtered Pass@k: filter out cases that doesn’t pass the 3 samples in problem description

APPS

Problem
Given is a directed graph G with N vertices and M edges. The vertices are numbered 1 to N, and the i-th edge is directed from Vertex A_i to Vertex B_i. It is guaranteed that the graph contains no self-loops or multiple edges. Determine whether there exists an induced subgraph (see Notes) of G such that the in-degree and out-degree of every vertex are both 1. If the answer is yes, show one such subgraph. Here the null graph is not considered as a subgraph.

APPS

APPS dataset	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%)

Note: passing timeouts in (parens)

Temperature 0.6 used for sampling all k in pass@k

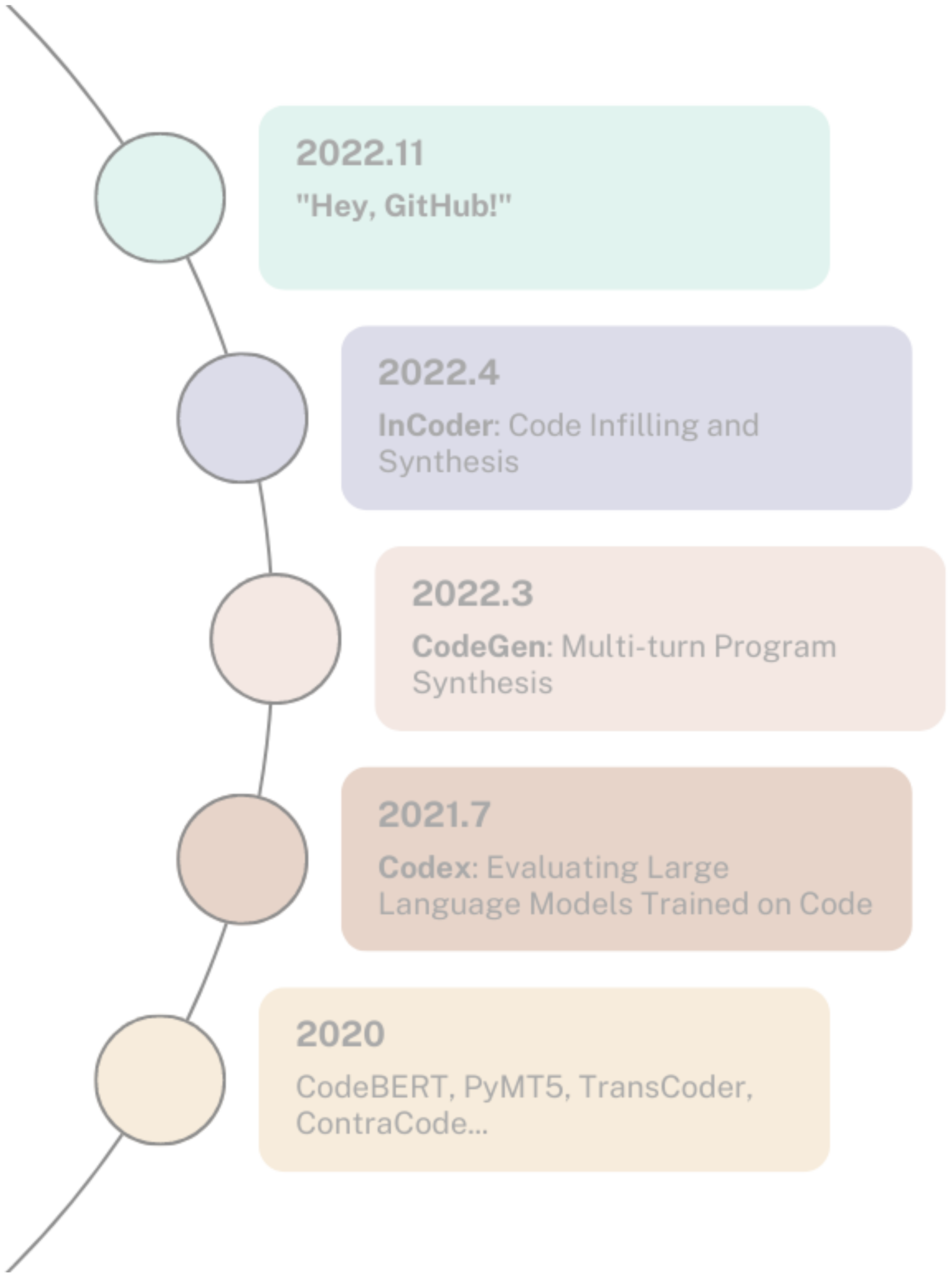
Outline

Codex

- Introduction
- Evaluation
- Code Fine-Tuning
- Experiments
- Supervised Fine-Tuning
- Docstring Generation
- **Discussions**

InCoder

CodeGen



Limitations

Degradation with length of instruction

Experiment: Compose 13 building blocks of <description, function> pair

Chained Building Blocks:

- Concatenate one-line descriptions into docstring

1. "remove all instances of the letter e from the string"

```
s = s.replace("e", "")
```

2. "replace all spaces with exclamation points in the string"

```
s = s.replace(" ", "!")
```

3. "convert the string s to lowercase"

```
s = s.lower()
```

4. "remove the first and last two characters of the string"

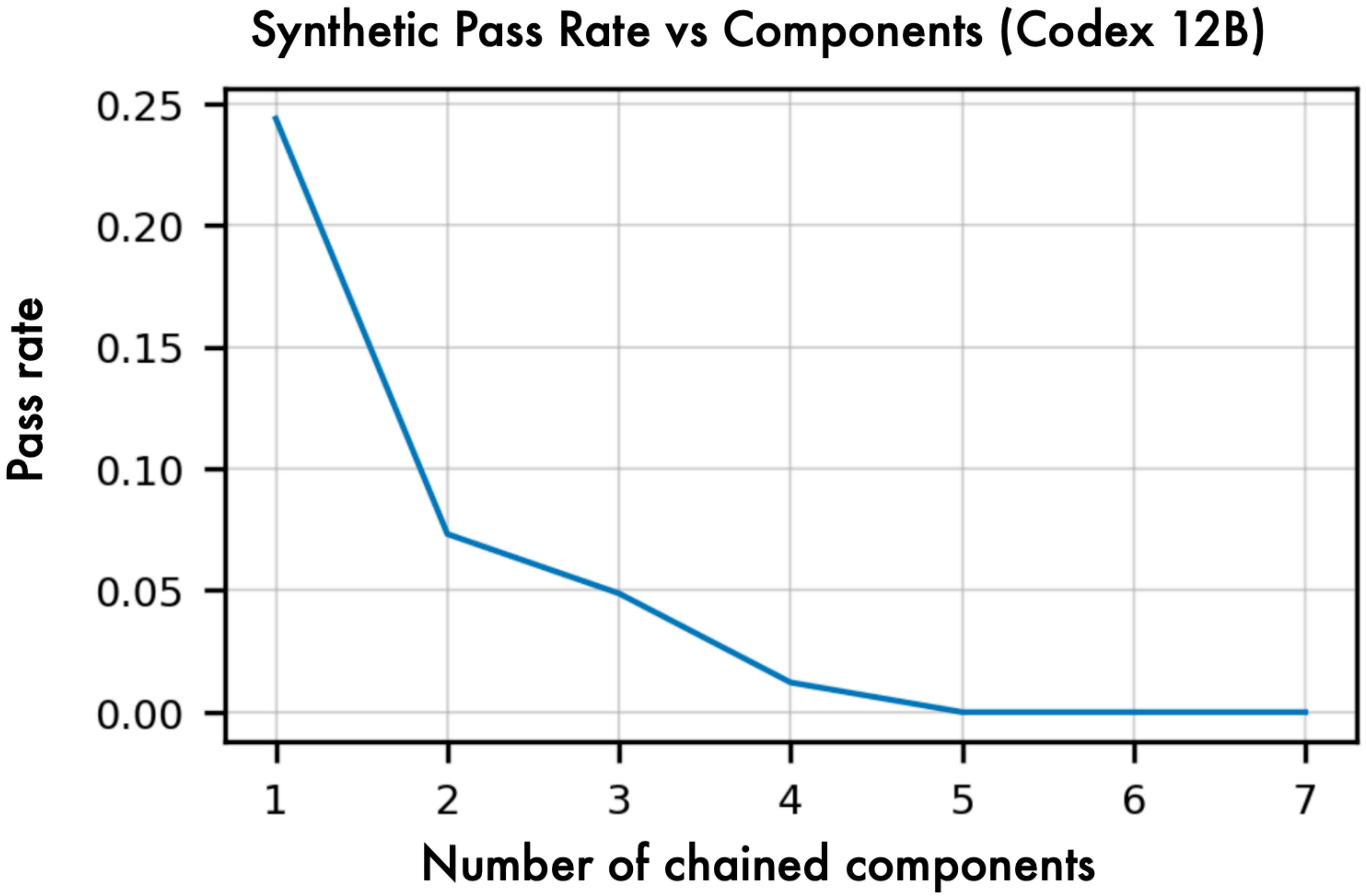
```
s = s[2:-2]
```

5. "removes all vowels from the string"

```
s = "".join(char for char in s if char not in "aeiouAEIOU")
```

Limitations

Degradation with length of instruction



Hazard Analysis

Over-Reliance: Codex may generate incorrect code that looks fine to novice programmers

Misalignment: Training distribution misalign with the intention of programmers

Bias: Biased prompting setups lead to biased generations

A model is ***intent misaligned*** *if outputs B, in a scenario where the user prefers output A and the model is both:*

(1) capable of outputting A

(2) capable of distinguishing situations where the user prefers A or B

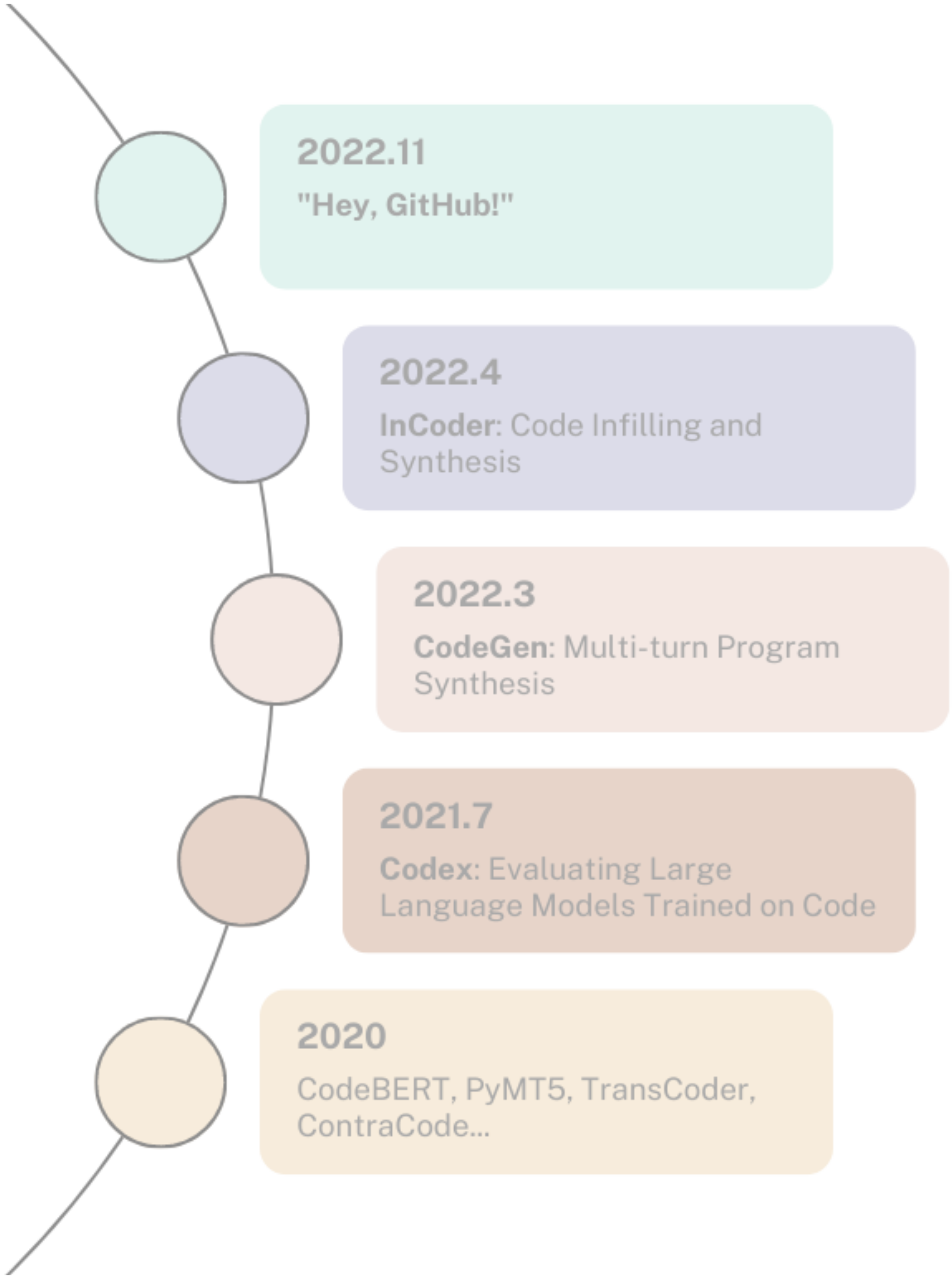
Outline

Codex

CodeGen

InCoder

Codex for NLP



CodeGen

Open-source alternative to Codex

16B params

Otherwise, very similar to Codex - albeit with a different exact dataset

Slightly outperforms Codex for all k on pass@k on HumanEval

Multi-turn evaluation provides a performance boost

Similar to chain of thought

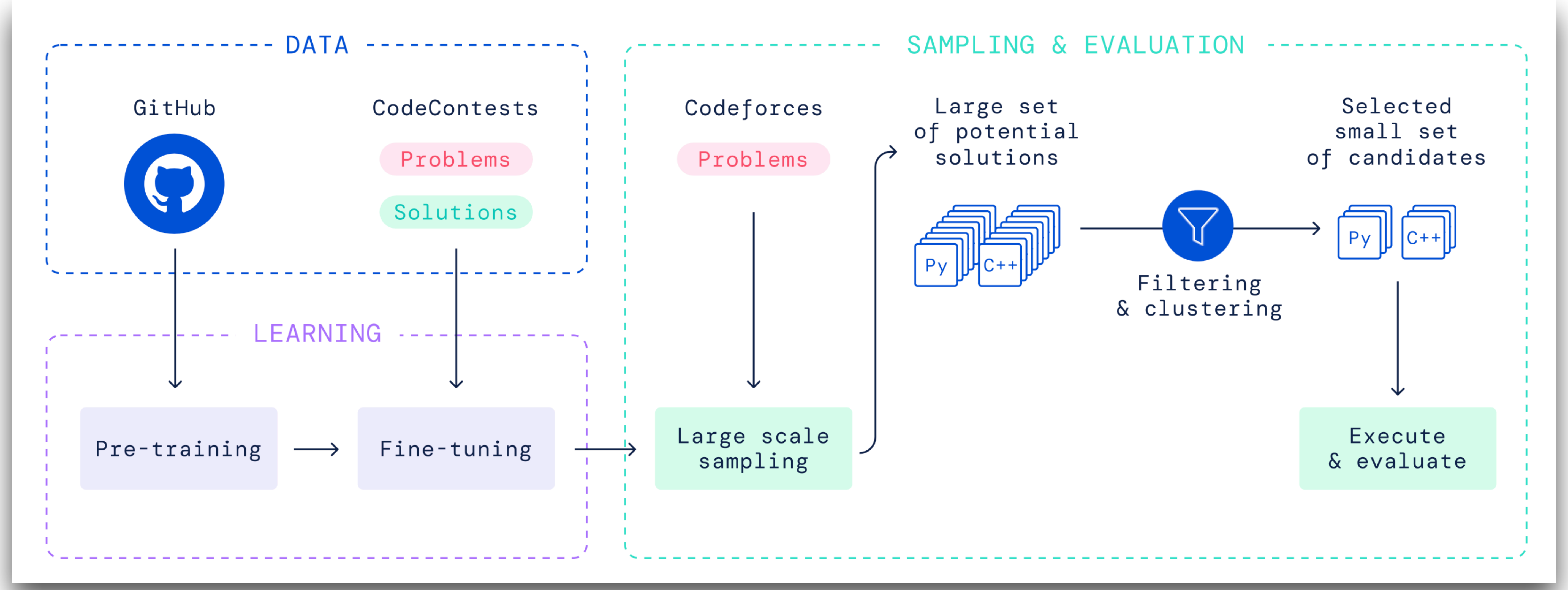
Model	pass@k [%]		
	$k=1$	$k=10$	$k=100$
GPT-NEO 350M	0.85	2.55	5.95
GPT-NEO 2.7B	6.41	11.27	21.37
GPT-J 6B	11.62	15.74	27.74
CODEX 300M	13.17	20.37	36.27
CODEX 2.5B	21.36	35.42	59.50
CODEX 12B	28.81	46.81	72.31
CODEGEN-NL 350M	2.12	4.10	7.38
CODEGEN-NL 2.7B	6.70	14.15	22.84
CODEGEN-NL 6.1B	10.43	18.36	29.85
CODEGEN-NL 16.1B	14.24	23.46	38.33
CODEGEN-MULTI 350M	6.67	10.61	16.84
CODEGEN-MULTI 2.7B	14.51	24.67	38.56
CODEGEN-MULTI 6.1B	18.16	28.71	44.85
CODEGEN-MULTI 16.1B	18.32	32.07	50.80
CODEGEN-MONO 350M	12.76	23.11	35.19
CODEGEN-MONO 2.7B	23.70	36.64	57.01
CODEGEN-MONO 6.1B	26.13	42.29	65.82
CODEGEN-MONO 16.1B	**29.28**	**49.86**	**75.00**

AlphaCode

Goal: solving competition problems

Similar approach, except:

- 41B parameters
- encoder-decoder model
- larger sampling



Avoiding data-leakage of HumanEval

- time-separated

Performed on par with a "median" human competitor

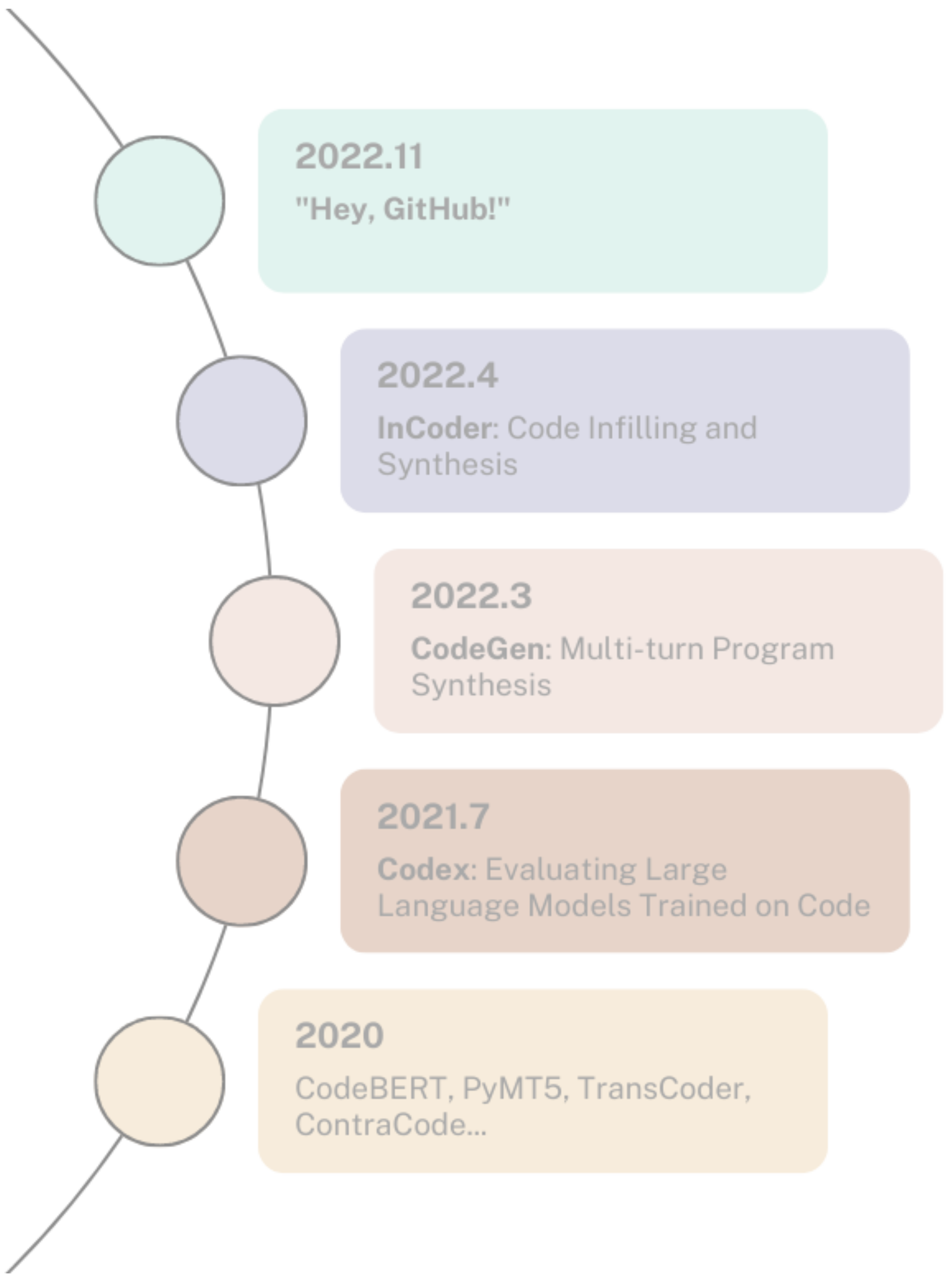
Outline

Codex

CodeGen

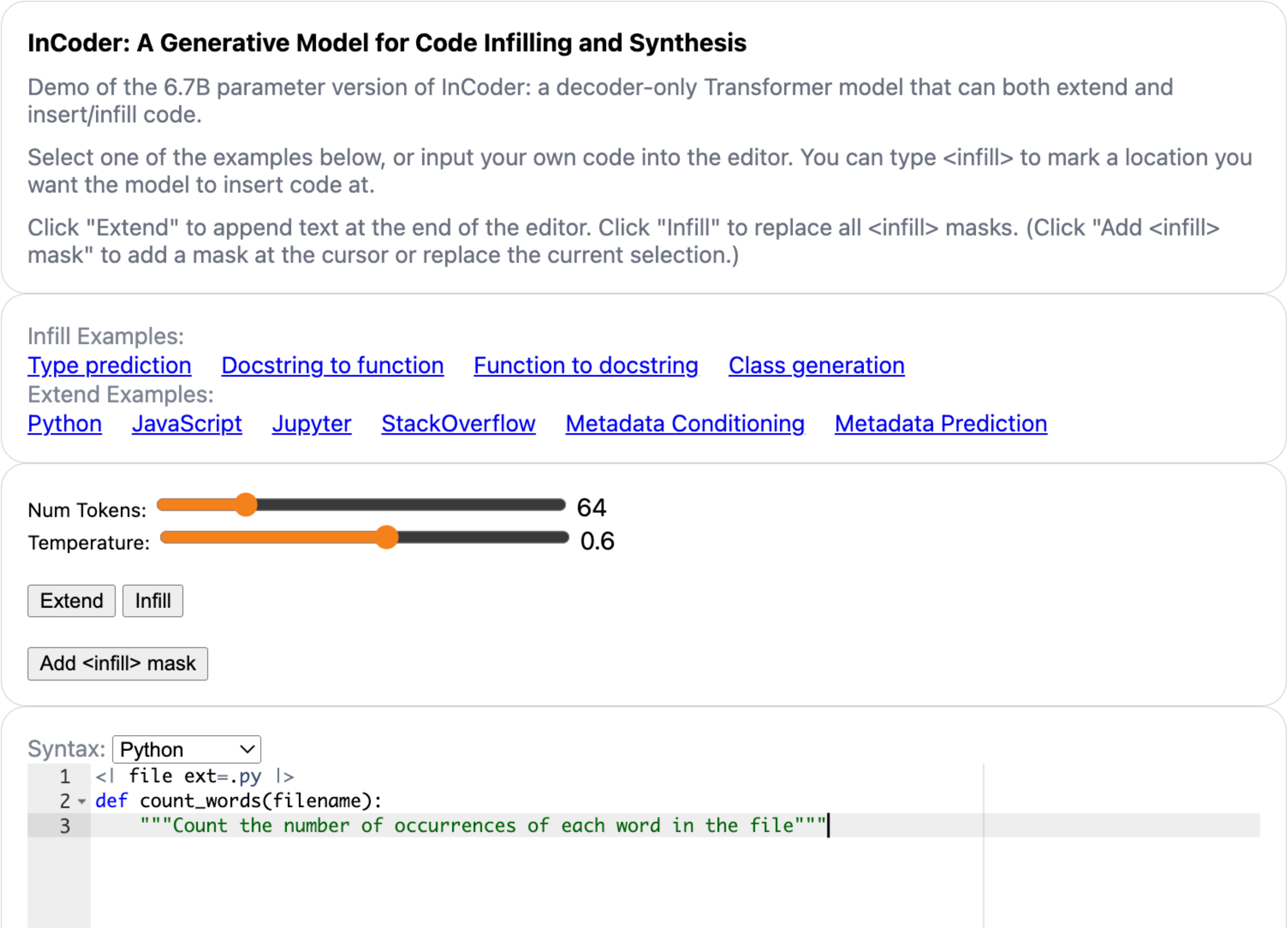
InCoder

Codex for NLP



Code Infilling and Synthesis

Motivation: supports both left-to-right generation and infilling arbitrary blocks of code



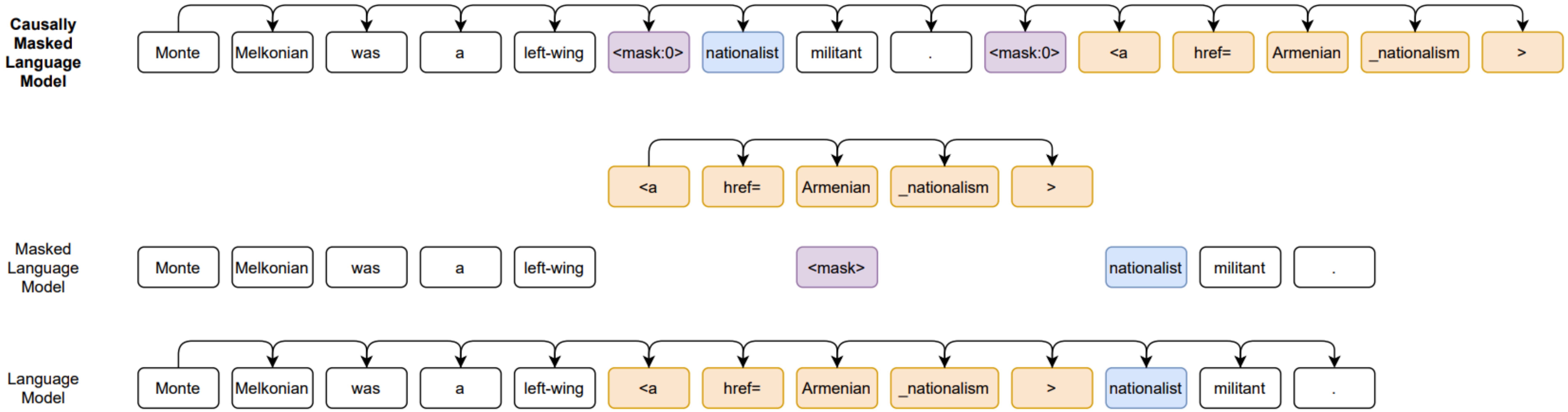
Causal Masking

Training

Given document size s, select

Number of masks: $n \sim Clamp(Poisson(1),1,16)$

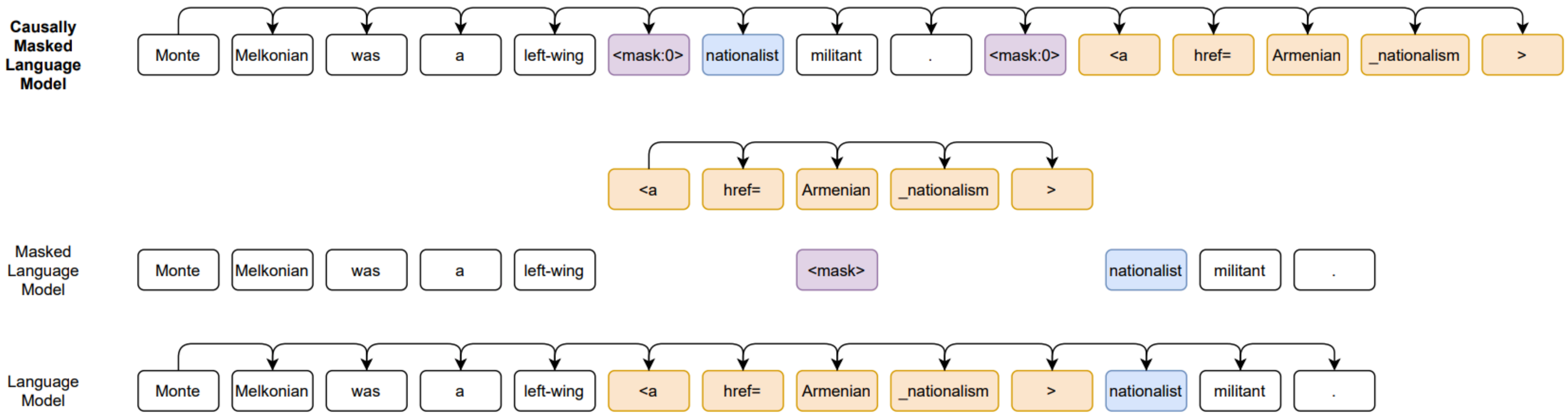
Length of each mask: $m \sim (Uniform(0,s), Uniform(0,s))$



Causal Masking

Training

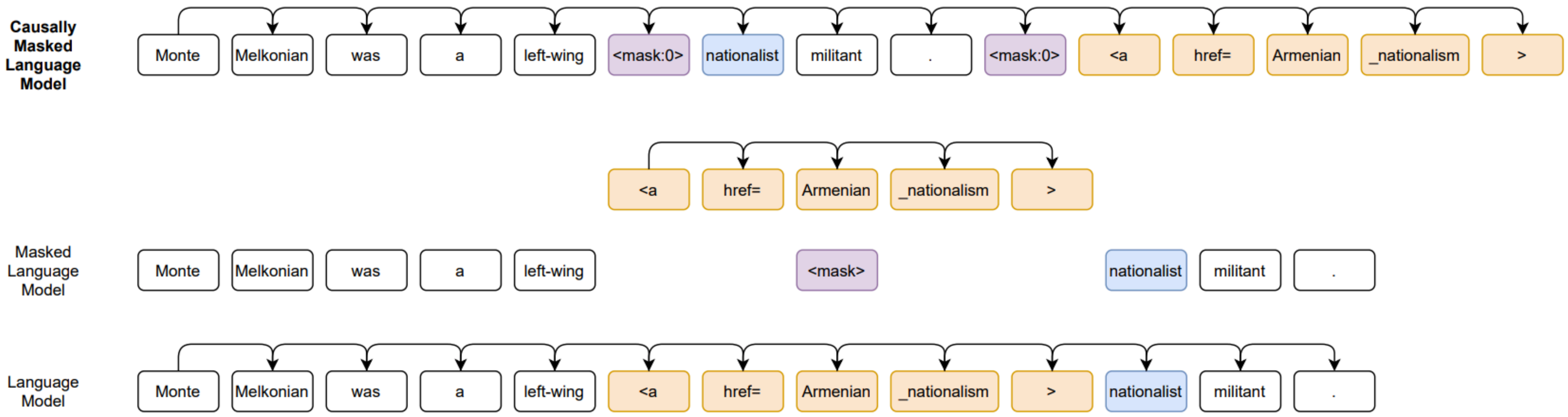
Given chosen spans, remove corresponding blocks with *mask sentinel token*
Move blocks of code to the end of file, each separated by <EOM>



Causal Masking

Training

Maximize $log\ P([left; \langle Mask\ ,0\rangle; right; \langle Mask\ 0\rangle; SPAN; \langle EOM\rangle])$



Causal Masking

Inferencing

Prompt generation through feeding the corresponding *mask sentinel token*

Left-to-right generation: Prompt generation with no right context

Infilling:

[A; <Mask:0>; C; <Mask:1>; E; <Mask:2>] → [A; <Mask:0>; C; <Mask:1>; E; <Mask:2>; <Mask:0>; B; <EOM>; <Mask:1>; D; <EOM>]

Left-to-Right

Model	Size (B)	Python Code (GB)	Other Code (GB)	Other (GB)	Code License	Infill?	HE @1	HE @10	HE @100	MBPP @1
Released										
CodeParrot [61]	1.5	50	None	None	—		4.0	8.7	17.9	—
PolyCoder [68]	2.7	16	238	None	—		5.6	9.8	17.7	—
GPT-J [63, 18]	6	6	90	730	—		11.6	15.7	27.7	—
INCODER-6.7B	6.7	52	107	57	Permissive	✓	15.2	27.8	47.0	19.4
GPT-NeoX [14]	20	6	90	730	—		15.4	25.6	41.2	—
CodeGen-Multi [46]	6.1	62	375	1200	—		18.2	28.7	44.9	—
CodeGen-Mono [46]	6.1	279	375	1200	—		26.1	42.3	65.8	—
CodeGen-Mono [46]	16.1	279	375	1200	—		29.3	49.9	75.0	—
Unreleased										
LaMDA [10, 60, 21]	137	None	None	???	—		14.0	—	47.3	14.8
AlphaCode [44]	1.1	54	660	None	—		17.1	28.2	45.3	—
Codex-12B [18]	12	180	None	>570	—		28.8	46.8	72.3	—
PaLM-Coder [21]	540	~20	~200	~4000	Permissive		36.0	—	88.4	47.0

Outline

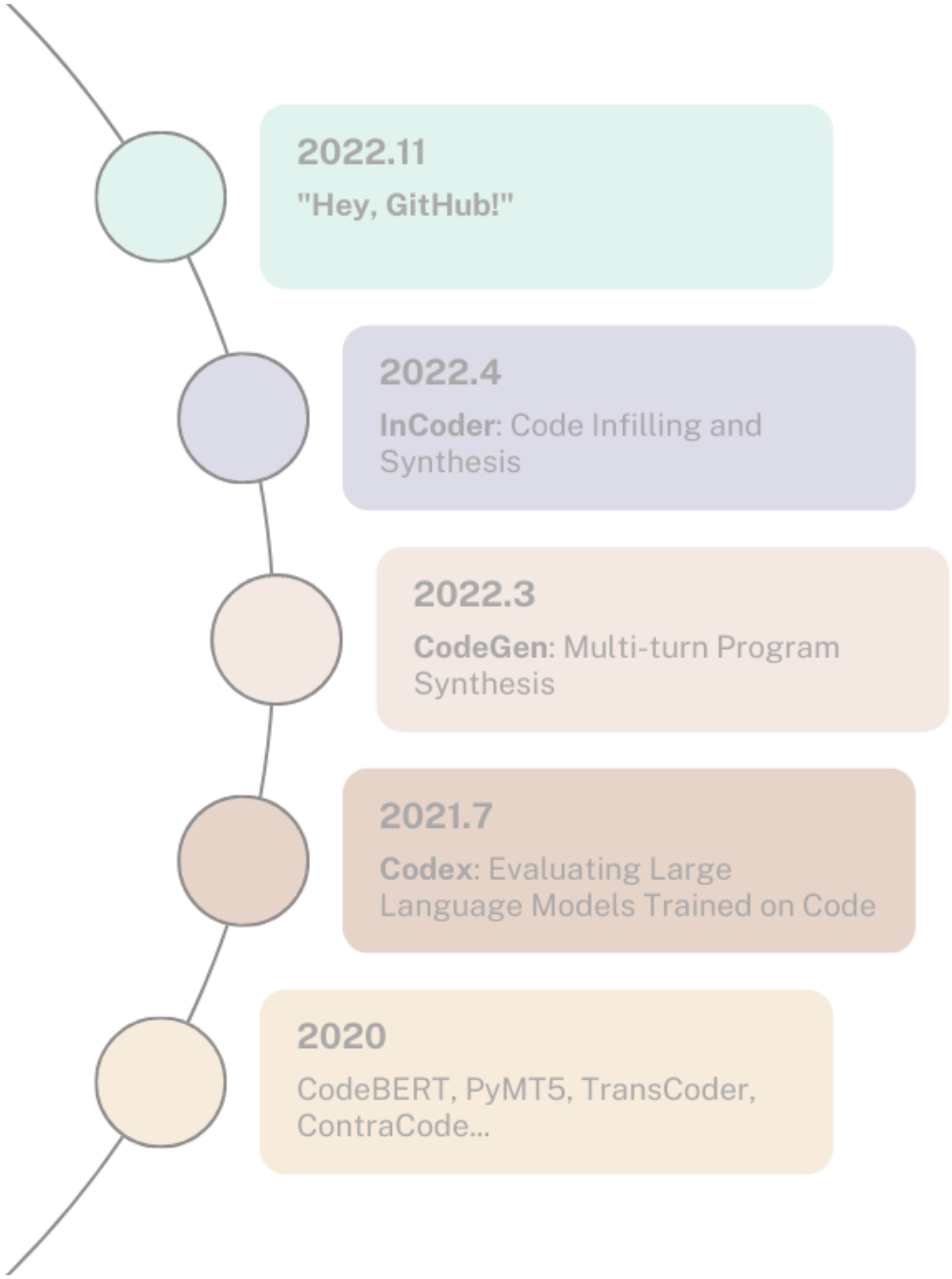
Codex

CodeGen

InCoder

CodeGen

Codex for NLP



Code Model on NLP Tasks

Highlight: Code Models (e.g. Codex) can outperform LLMs (e.g. GPT3) in NLP tasks that involve structural and logical analysis

Codex-NLP: Semantic Parsing

Task: convert an utterance *u* to a semantic meaning representation *m*, viewed either as a generation or classification problem

Dataset	Natural language	Canonical utterance	Meaning representation
SMCalFlow	Schedule Hide and Seek in the mall for Saturday night	create event called "Hide and Seek" starting next Saturday night at "mall"	`(Yield :output (CreateCommitEventWrapper :event (CreatePreflightEventWrapper :constraint (Constraint[Event] :subject (?= #(String "Hide and Seek")) :start (DateTimeConstraint :constraint (Night) :date (NextDOW :dow #(DayOfWeek "SATURDAY"))) :location (?= #(LocationKeyphrase "mall"))))))`
Overnight Cal.	which meeting has the earliest end time	meeting that has the smallest end time	`(call listValue (call superlative (call getProperty (call singleton en.meeting) (string !type)) (string min) (call ensureNumericProperty (string end_time))))`

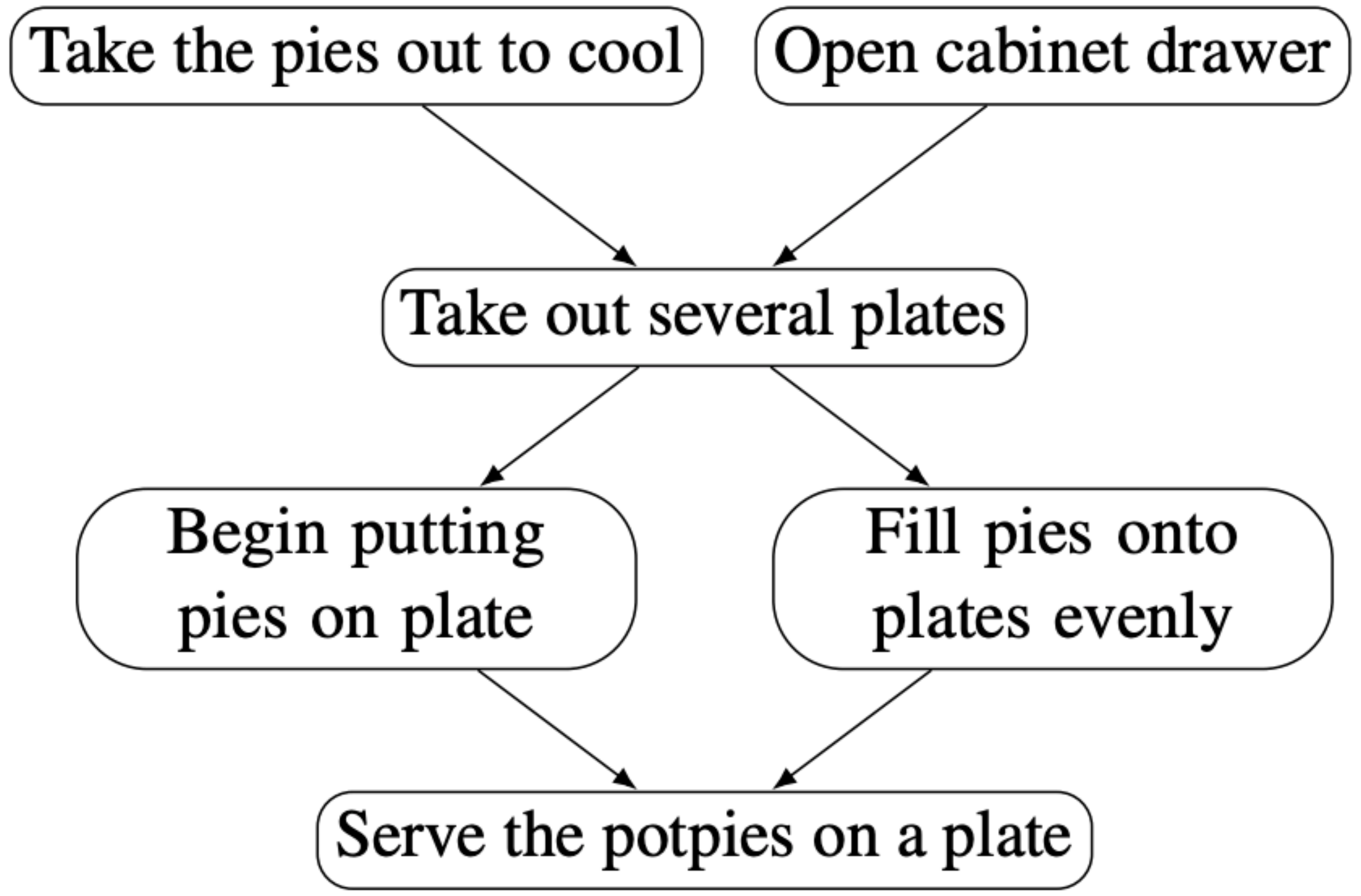
Codex-NLP: Semantic Parsing

Task: convert an utterance *u* to a semantic meaning representation *m*, viewed either as a generation or classification problem

Model	Accuracy Overnight Cal.	SMCalFlow
Davinci	0.81	0.340
Curie	0.66	0.260
Davinci Codex	0.86	0.355
Cushman Codex	0.87	0.320

Codex-NLP: Reasoning

Task: Given a event/goal T, generate a commonsense reasoning represented as a graph



(a) The script $\mathcal{G}$

```python
class Tree:

  goal = "serve the potpies on a plate"

  def __init__(self):
    # nodes
    take_pies_out_to_cool = Node()
    open_cabinet_drawer = Node()
    take_out_several_plates = Node()
    ...
    # edges
    take_pies_out_to_cool.children =
        [take_out_several_plates]
    open_cabinet_drawer.children =
        [take_out_several_plates]
    ...
```

(b) $\mathcal{G}$ converted to Python code $\mathcal{G}_c$ using our approach

Codex-NLP: Reasoning

Task: Given a event/goal T, generate a commonsense reasoning represented as a graph

	PID	Precision	Recall	F1
DAVINCI	*r1*	72.7	50.9	59.8
	r2	75.9	45.6	57.0
	r3	73.8	42.4	53.9
	avg	74.2± 1.3	46.3± 3.5	56.9± 2.4
CODEX-002	*r1*	69.0	54.5	60.9
	r2	84.6	48.1	61.3
	r3	77.6	55.7	66.2
	avg	**77.1**± 6.4	**52.8**± 3.3	**62.8**± 2.4

Codex-NLP: Proof Synthesis

Task: Given a set of facts and a hypothesis, construct a proof of how the facts conclude at the hypothesis

Method	Leaves		Steps		Intermediates		Overall
	F1	AllCorrect	F1	AllCorrect	F1	AllCorrect	AllCorrect
EntailmentWriter	86.2	43.9	40.6	28.3	67.1	34.8	27.3
EntailmentWriter (T5-11B)	**89.4**	52.9	46.6	35.3	69.1	36.9	32.1
NLProofS (ours)	**89.4 ± 0.8**	**56.0 ± 0.7**	**50.4 ± 1.9**	**38.4 ± 1.3**	**71.9 ± 1.4**	**41.3 ± 1.4**	**37.1 ± 1.5**
GPT-3 (Brown et al., 2020)	64.2 ± 2.3	15.3 ± 1.9	17.6 ± 0.6	12.3 ± 1.4	53.6 ± 1.4	22.3 ± 1.1	12.3 ± 1.4
Codex (Chen et al., 2021)	68.9 ± 3.7	19.8 ± 3.2	21.4 ± 3.0	14.6 ± 1.7	55.6 ± 2.2	23.2 ± 1.9	14.4 ± 1.4

Table B: Validation results of proof generation on EntailmentBank (Dalvi et al., 2021). Results of GPT-3 and Codex are based on prompting with 7 in-context examples randomly sampled from the training data.

Q3. As a programmer, what types of features would you like to use from a code model (describe the use cases in detail)? Do you think current code models already achieve that, or what improvements need to be done in the future?

Q3. As a programmer, what types of features would you like to use from a code model (describe the use cases in detail)? Do you think current code models already achieve that, or what improvements need to be done in the future?

Features

- Autocomplete:

Given comment and some written code to start with, the model should be able to complete a block of code that matches the intention of function, variable names, and be able to call (if needed) other functions / imported libraries

- Docstring Formatting

Given function signature and code, infill missing components in the function docstring to include variable type, description, return type, etc.

- Code Search

Given a description of intention of function, search a codebase if such function is already implemented

……

Q3. As a programmer, what types of features would you like to use from a code model (describe the use cases in detail)? Do you think current code models already achieve that, or what improvements need to be done in the future?

Discussion

Code model that can fill in blanks in code or docstring would be better than left-to-right only generation for many reasons (e.g. to call functions declared both above and below a target function). HumanEval focuses on standalone functions, which may create a potential misalignment between the model's trained goal and intention of programmers.

References

(apart from the ones on the website):

AlphaCode: https://arxiv.org/abs/2203.07814

Semantic Parsing with LMs of Code: https://arxiv.org/abs/2112.08696

Natural Language Proofs: https://arxiv.org/abs/2205.12443