Session #15: Self-Supervised Coding Models

Tuesday, October 18

CSCI 601.771: Self-supervised Statistical Models

Johns Hopkins University

Your AI pair programmer

GitHub Copilot uses the OpenAI Codex to suggest code and entire functions in real-time, right from your editor.

https://github.caom/features/copilot

sentiments.ts | write_sql.go | parse_expenses.py | addresses.rb

```typescript
#!/usr/bin/env ts-node

import { fetch } from "fetch-h2";

// Determine whether the sentiment of text is positive
// Use a web service
async function isPositive(text: string): Promise<boolean> {
  const response = await fetch(`http://text-processing.com/api/sentiment/`, {
    method: "POST",
    body: `text=${text}`,
    headers: {
      "Content-Type": "application/x-www-form-urlencoded",
    },
  });
  const json = await response.json();
  return json.label === "pos";
}
```

Copilot

Replay

https://github.caom/features/copilot

Amazon CodeWhisperer

Build applications faster with the ML-powered coding companion

https://aws.amazon.com/codewhisperer/

Python

```python
import boto3
from botocore.exceptions import ClientError


# Function to upload a file to an S3 bucket
def upload_file(file_name, bucket, object_name=None):
    """Upload a file to an S3 bucket

    :param file_name: File to upload
    :param bucket: Bucket to upload to
    :param object_name: S3 object name. If not specified then file_name is used
    :return: True if file was uploaded, else False
    """

    # If S3 object_name was not specified, use file_name
    if object_name is None:
        object_name = file_name

    # Upload the file
    s3_client = boto3.client('s3')
    try:
        response = s3_client.upload_file(file_name, bucket, object_name)
    except ClientError as e:
        logging.error(e)
        return False
    return True
```

Amazon CodeWhisperer

Google AI Blog

The latest from Google Research

ML-Enhanced Code Completion Improves Developer Productivity

Tuesday, July 26, 2022

Posted by Maxim Tabachnyk, Staff Software Engineer and Stoyan Nikolov, Senior Engineering Manager, Google Research

salesforce

AI Coding with CodeRL: Toward Mastering Program Synthesis with Deep Reinforcement Learning

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]

Ayo, Fadil, Yongrui

- Stakeholders 😭

Introduction

- Large language models: powerful!
  - GPT3
    - Could generate simple programs from Python docstrings
    - Exciting: not explicitly trained on code generation
  - ➞ hypothesis: a specialized GPT would excel at coding tasks ➞ Codex

- Method: inspired by real-world programming
  - Real-word: iterations, bug fixes
  - Approximation: generating many samples from our model, select one that passes all unit tests
  - Further evaluation: what if only one sample is generated?

Models & GPT

- GPT: Baseline to compare with Codex
  - GPT-12B: solve no problems when single sample generated
  - GPT-J: solve 11.4% of problems
- Codex = GPT + fine-tuning
  - Solve 28.8% of problems with single sample generated
- Codex-S = Codex + supervised fined-tuning
  - Solve 77.5% of problems with at least one correct solution
  - Solve 37.7% of problems with single sample generated
- Codex-D: generate docstrings from code

Codex-S

- Motivation: some code unrelated to synthesizing functions from docstrings
- Supervised fine-tuned on correctly implemented standalone functions
- Data collected from
  - Competitive programming website
    - problem statements ➞ docstrings
    - example unit tests
  - Repositories with continuous integration
    - input/output for functions ➞ unit tests
    - no need to know algorithms and data structures
      - complement the puzzle nature of coding competition
      - broaden the distribution of tasks

Codex-D

- Motivation: describe the intent behind generated code
- But not easy
  - Leave out important details
  - Over conditioned on the function name
  - Developers devote less time to writing docstrings

Samples

```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]
```

prompts

Generated sample

```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)
```

prompts

Generated sample

Evaluation

- Pass@k evaluates
  - Functional correctness
  - k: Number of code samples generated per problem
  - Pass: Any sample that passes the unit tests.
  - Total fraction of problems solved is reported.
- Downside: Causes high variance

Evaluation

- Pass@k evaluates
  - Functional correctness
  - k: Number of code samples generated per problem
  - Pass: Any sample that passes the unit tests.
  - Total fraction of problems solved is reported.
- Downside: Causes high variance

```
def pass_at_k(n, c, k):
    """
    :param n: total number of samples
    :param c: number of correct samples
    :param k: k in pass@$k$
    """
    if n - c < k: return 1.0
    return 1.0 - np.prod(1.0 - k /
        np.arange(n - c + 1, n + 1))
```

Figure 3. A numerically stable script for calculating an unbiased estimate of pass@k.

- Let's reduce variance:
  - Generate n >= k samples per task
  - n= 200
  - k<= 100
  - c: Count the number of correct samples, c <= n
  - Calculate unbiased estimator

Evaluation Metric

- BLEU Score vs. Pass @ k
  - Match based metric vs Function based metric
  - Sequential vs Tree structure
  - Ambiguity in NLP vs Unique semantics in Code
    - BLEU score has problems getting semantics that are code-specific

Evaluation Metric

- BLEU Score vs. Pass @ k
  - Match based metric vs Function based metric
  - Sequential vs Tree structure
  - Ambiguity in NLP vs Unique semantics in Code
    - BLEU score has problems getting semantics that are code-specific

Conclusion: BLEU score may not indicate improved rates of functional correctness in practice.

Datasets

- HumanEval
  - Dataset of 164 handwritten programming problems
  - Each problem includes a
    - Function signature
    - Docstring
    - Body
    - Unit tests, 7.7 tests per problem
  - Programming tasks in the HumanEval dataset assess
    - Language comprehension
    - Reasoning
    - Algorithms
    - Simple mathematics.

pass@k in practice

t.py

t.py > ...

```python
import os
os.environ["HF_ALLOW_CODE_EVAL"] = "1"

from evaluate import load
code_eval = load("code_eval")
test_cases = ["assert add(2,3)==5"]

candidates = [["def add(a, b): return a+b"]]
pass_at_k, results = code_eval.compute(references=test_cases, predictions=candidates, k=[1])
print(pass_at_k)

candidates = [["def add(a,b): return a*b"]]
pass_at_k, results = code_eval.compute(references=test_cases, predictions=candidates, k=[1])
print(pass_at_k)

candidates = [["def add(a, b): return a+b", "def add(a,b): return a*b"]]
pass_at_k, results = code_eval.compute(references=test_cases, predictions=candidates, k=[1, 2])
print(pass_at_k)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER

```
fadil@fadil-Book-13-RZ09-0357:~/Desktop/test$ /bin/python3 /home/fadil/Desktop/test/t.py
{'pass@1': 1.0}
{'pass@1': 0.0}
{'pass@1': 0.5, 'pass@2': 1.0}
fadil@fadil-Book-13-RZ09-0357:~/Desktop/test$
```

```python
def pass_at_k(n, c, k):
    """
    :param n: total number of samples
    :param c: number of correct samples
    :param k: k in pass@$k$
    """
    if n - c < k: return 1.0
    return 1.0 - np.prod(1.0 - k /
        np.arange(n - c + 1, n + 1))
```

Figure 3. A numerically stable script for calculating an unbiased estimate of pass@k.

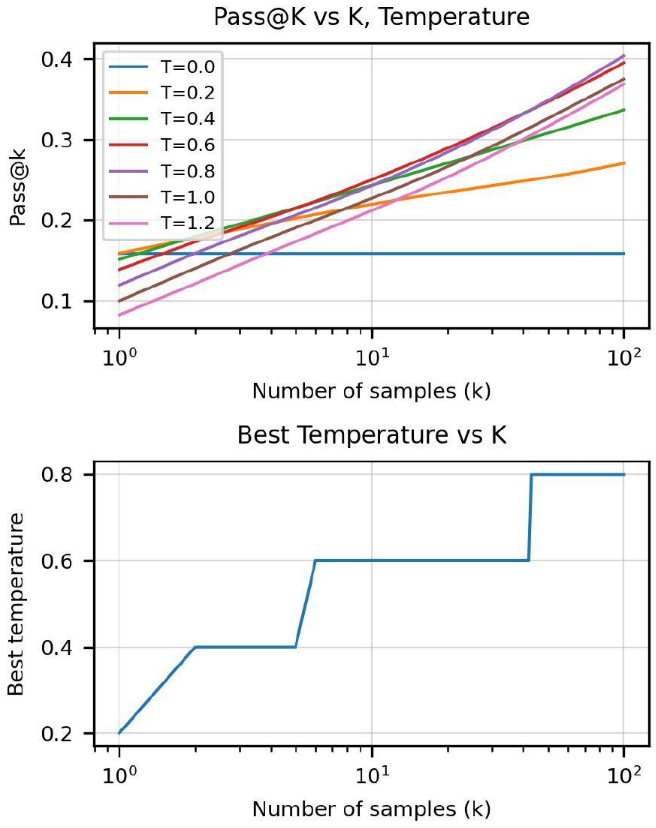
Results: Temp vs k

In sequence generating models, for vocabulary of size N (number of words, parts of words, any other kind of token), one predicts the next token from distribution of the form:

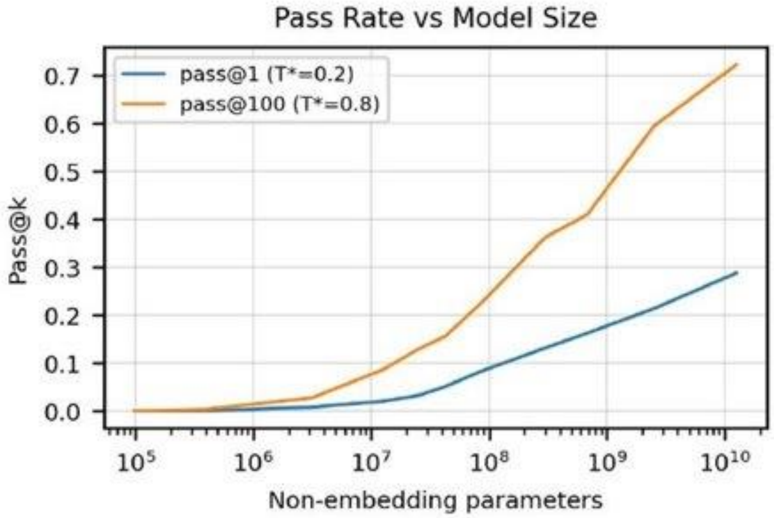
$$\mathrm{softmax}(x_i/T) \quad i = 1, \dots N,$$

Here T is the **temperature**. The output of the softmax is the probability that the next token will be the i-th word in the vocabulary.

The temperature determines how greedy the generative model is.

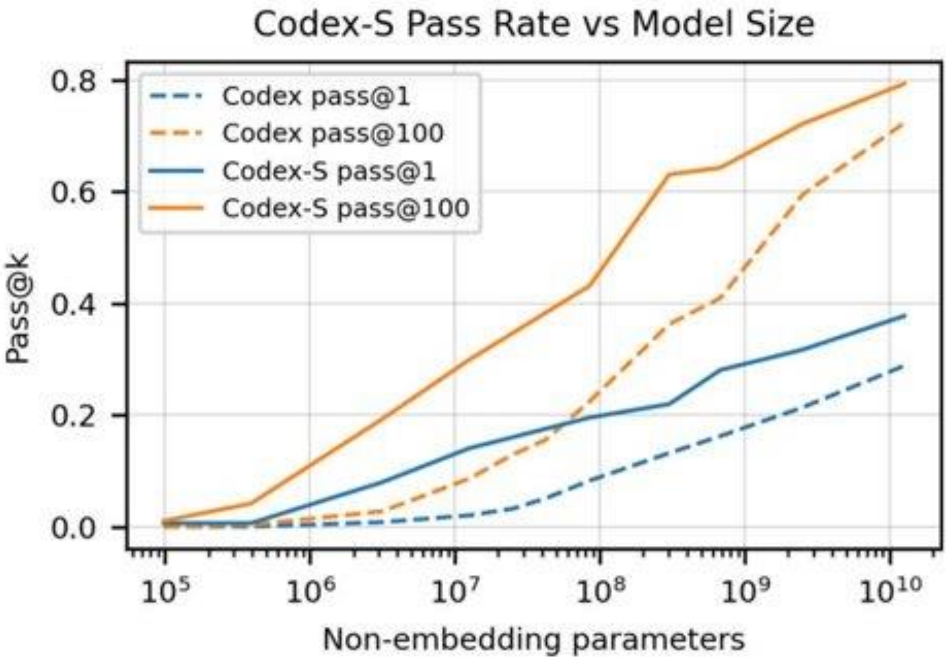


Codex and Codex-S Comparison on HumanEval



	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%

Codex and Codex-S Comparison on HumanEval



Datasets

- APPS
  - APPS dataset was used to measure the coding challenge competence of language models.
  - Collected from open source materials
  - 10,000 coding problems
  - 5000 training problems
    - Each with a set of unit tests and, for the training data, a set of correct solutions.
  - 5000 testing problems
    - Each with a set of unit tests and, for the training data, a set of correct solutions.

APPS Results

	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%)

Limitations

- Not sample efficient to train
- Invoke undefined/out-of-scope functio
- Performance degradation as docstring
- Have difficulty with binding operations

```python
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
```

```python
def add_elements(arr, k):
    """
    Given a non-empty array of integers arr and an
        integer k, return
    the sum of the first k element that has at most two
         digits.

    Example:

        Input: arr = [111,21,3,4000,5,6,7,8,9], k = 4
        Output: 24 # sum of 21 + 3

    Constraints:
        1. 1 <= len(arr) <= 100
        2. 1 <= k <= len(arr)
    """

    ### COMPLETION 1 (WRONG): ###
    pass


    ### COMPLETION 2 (WRONG): ###
    for i in range(0, k):
        if len(str(arr[i])) < 3:
            k = k - 1
            continue
        else:
            return arr[i]


    ### COMPLETION 3 (WRONG): ###
    sum_ = 0
    i = 0
    while k > 0 and i < len(arr):
        if len(str(arr[i])) <= 2:
            sum_ += arr[i]
            k -= 1
        i += 1
    return sum_
```

Thank You!

Empiricist

- Examples from CoPilot (https://github.com/features/copilot).
- Examples from a public model, CodeGen.
- Examples from GTP-2 (https://beta.openai.com/playground).
- Test and evaluation of the codes generated by GPT-2
- (https://colab.research.google.com/drive/1R6rWcqseTKGWglBiPyIlutwBx4xEQuCl#scrollTo=hiNfrqWr8Vl1)