InCoder: A Generative Model For Code Infilling and Synthesis

ICLR 2023

Overview

- Generative Model using bidirectional context
- Left-to-Right Generation-> Left-to-Right + Editing(Infilling, Mask)
- InCoder:
  - Type Inference
  - Docstring Generation
  - Variable Renaming
  - Complete Missing Line of Code

Causal Masking

- Code Generation either utilizes:
  - left-to-right (causal) autoregressive language modeling objective
  - Masked language modeling objective (BERT)
- Causal models
  - Only condition on context to the left
  - Can autoregressively generate entire documents
- Masked Language Models
  - Can condition both the left and right context to infill a masked region
  - Training objective is limited to generating only 15% of a document

Training

Training

Original Document

```
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
        word_counts = {}
        for line in f:
            for word in line.split():
                if word in word_counts:
                    word_counts[word] += 1
                else:
                    word_counts[word] = 1
    return word_counts
```

Masked Document

```
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
         <MASK:0> in word_counts:
                    word_counts[word] += 1
                else:
                    word_counts[word] = 1
    return word_counts
<MASK:0> word_counts = {}
         for line in f:
             for word in line.split():
                 if word <EOM>
```

- A “span k” is replaced with <Mask:k>

Training

- # of Spans = Poisson Distribution with a mean of one
  - (50% cases are single spans but count can go up to 256spans)

- Maximize: log P([Left;<Mask:0>,Right;<Mask:0>;Span;<EOM>])

Inference

- P(·|[Left;<Mask:0>;Right;<Mask:0>])
- Generation is continued at the end
  - Until <EOM> is generated or a stopping criterion is reached

Model: InCoder-6.7B

- Based on 6.7B Transformer language model (Vaswani et al. 2017)
- Focus is Python but includes 28 languages

Experiments

- Model can test for three methods

- Causal Masking Inference Procedure
  - P(·|[Left;<Mask:0>;Right;<Mask:0>])

- Left-to-right single
  - P(·|Left)

- Left-to-right reranking
  - P(·|Left) to generate K (10) possible entries (Span1~SpanK)
  - Calculate log P(Left;SpanK;Right) or another method (Chen et al.)
  - Determine candidate

Infilling Lines of Code (HumanEval)

- HumanEval dataset (Chen et al. 2021a)

- Single Line Infilling
  - Metric: Pass rate
    - The rate at which the completed function passes all of the function's input-output pairs
  - Metric: Exact Match
    - Percentage of times that the completed lines exactly match the masked lines

- Multi Line Infilling
  - More than one line
  - N x (N + 1) / 2 examples for a function with N non-blank lines

Infilling Lines of Code (HumanEval)

Method	Pass Rate	Exact Match
L-R single	48.2	38.7
L-R reranking	54.9	44.1
CM infilling	69.0	56.3
PLBART	41.6	—
code-cushman-001	53.1	42.0
code-davinci-001	63.0	56.0

(a) Single-line infilling.

Method	Pass Rate	Exact Match
L-R single	24.9	15.8
L-R reranking	28.2	17.6
CM infilling	38.6	20.6
PLBART	13.1	—
code-cushman-001	30.8	17.4
code-davinci-001	37.8	19.8

(b) Multi-line infilling.

Table 1: On our single- and multi-line code infilling benchmarks that we construct from HumanEval, our causal-masked (CM) approach obtains substantial improvements over left-to-right single candidate and left-to-right reranking baselines in both function test pass rate and exact match.

Infilling Lines of Code (HumanEval)

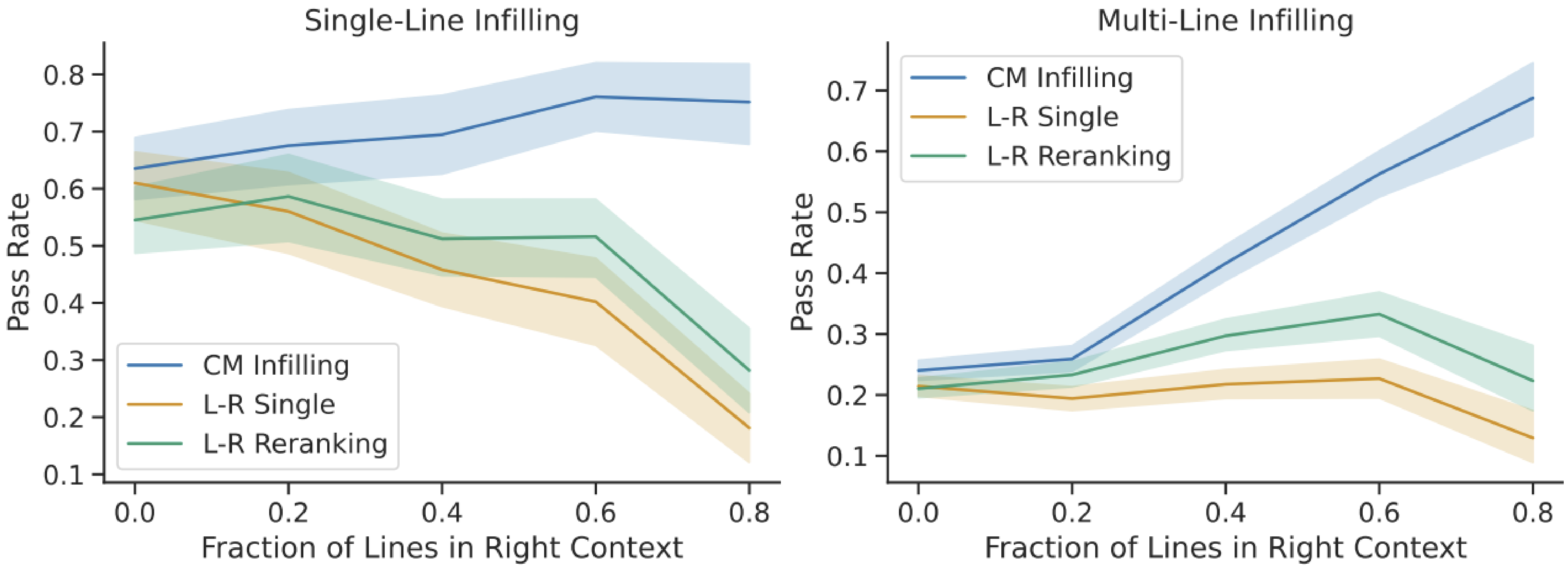

Figure 2: Infilling pass rate by the fraction of the function's lines which are provided to the right of the region that must be infilled, for single-line infilling (left) and multi-line infilling (right). Shaded regions give 95% confidence intervals, estimated using bootstrap resampling. Our causal-masked (CM) infilling method, blue, consistently outperforms both of the left-to-right (L-R) baselines, with larger gains as more right-sided context becomes available (the right side of both graphs).

Infilling Example

Original Document

```python
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
        word_counts = {}
        for line in f:
            for word in line.split():
                if word in word_counts:
                    word_counts[word] += 1
                else:
                    word_counts[word] = 1
    return word_counts
```

Multi-Region Infilling

```python
from collections import Counter

def word_count(file_name):
    """Count the number of occurrences of each word in the file."""
    words = []
    with open(file_name) as file:
        for line in file:
            words.append(line.strip())
    return Counter(words)
```

Docstring Generation (CodeXGLUE)

- CodeXGLUE code to text docstring generation task (Lu et al. 2021)
- 4-gram BLEU scores

Method	BLEU
Ours: L-R single	16.05
Ours: L-R reranking	17.14
Ours: Causal-masked infilling	18.27
RoBERTa (Finetuned)	18.14
CodeBERT (Finetuned)	19.06
PLBART (Finetuned)	19.30
CodeT5 (Finetuned)	20.36

Table 2: CodeXGLUE Python Docstring generation BLEU scores. Our model is evaluated in a zero-shot setting, with no fine-tuning for docstring generation, but it approaches the performance of pretrained code models that are fine-tuned on the task's 250K examples (bottom block).

Docstring Generation

Original Document

```
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
        word_counts = {}
        for line in f:
            for word in line.split():
                if word in word_counts:
                    word_counts[word] += 1
                else:
                    word_counts[word] = 1
    return word_counts
```

Docstring Generation

```
def count_words(filename: str) -> Dict[str, int]:
    """
    Counts the number of occurrences of each word in the given file.

    :param filename: The name of the file to count.
    :return: A dictionary mapping words to the number of occurrences.
    """
    with open(filename, 'r') as f:
        word_counts = {}
        for line in f:
            for word in line.split():
                if word in word_counts:
                    word_counts[word] += 1
                else:
                    word_counts[word] = 1
    return word_counts
```

Return Type Prediction

Method	Accuracy
Left-to-right single	12.0
Left-to-right reranking	12.4
Causal-masked infilling	**58.1**

(a) Results on the test set of the benchmark that we construct from CodeXGLUE.

Method	Precision	Recall	F1
Ours: Left-to-right single	30.8	30.8	30.8
Ours: Left-to-right reranking	33.3	33.3	33.3
Ours: Causal-masked infilling	**59.2**	**59.2**	**59.2**
TypeWriter (Supervised)	54.9	43.2	48.3

(b) Results on a subset of the TypeWriter's OSS dataset (Pradel et al., 2020). We include examples from which we were able to obtain source files, successfully extract functions and types, that have non-None return type hints, and that were not included in our model's training data.

Table 3: Results for predicting Python function return type hints on two datasets. We see substantial improvements from causal masked infilling over baseline methods using left-to-right inference.

Return Type Prediction

Original Document

```python
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
        word_counts = {}
        for line in f:
            for word in line.split():
                if word in word_counts:
                    word_counts[word] += 1
                else:
                    word_counts[word] = 1
    return word_counts
```

Type Inference

```python
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
        word_counts = {}
        for line in f:
            for word in line.split():
                if word in word_counts:
                    word_counts[word] += 1
                else:
                    word_counts[word] = 1
    return word_counts
```

Variable Renaming

Method	Accuracy
Left-to-right single	18.4
Left-to-right reranking	23.5
Causal-masked infilling	30.6

Table 4: Results on the variable renaming benchmark that we construct from CodeXGLUE. Our model benefits from using the right-sided context in selecting (L-R reranking and CM infilling) and proposing (CM infilling) variable names.

Original Document

```
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
        word_counts = {}
        for line in f:
            for word in line.split():
                if word in word_counts:
                    word_counts[word] += 1
                else:
                    word_counts[word] = 1
    return word_counts
```

Variable Name Prediction

```
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
        word_count = {}
        for line in f:
            for word in line.split():
                if word in word_count:
                    word_count[word] += 1
                else:
                    word_count[word] = 1
    return word_count
```