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
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 256 spans)

• Maximize: $\log P([\text{Left};<\text{Mask}:0>,\text{Right};<\text{Mask}:0>;\text{Span};<\text{EOM}>])$
Inference

• $P(\cdot | [\text{Left}; <\text{Mask:0}>; \text{Right}; <\text{Mask:0}> ])$

• Generation is continued at the end
  • Until $<\text{EOM}>$ 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
  • $P(\cdot|[\text{Left}; <\text{Mask:0}>; \text{Right}; <\text{Mask:0}>])$

• Left-to-right single
  • $P(\cdot|\text{Left})$

• Left-to-right reranking
  • $P(\cdot|\text{Left})$ to generate $K$ (10) possible entries ($\text{Span1} \sim \text{SpanK}$)
  • Calculate log $P(\text{Left}; \text{SpanK}; \text{Right})$ 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
  • $N \times (N + 1) / 2$ examples for a function with $N$ non-blank lines
Infilling Lines of Code (HumanEval)

<table>
<thead>
<tr>
<th>Method</th>
<th>Pass Rate</th>
<th>Exact Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-R single</td>
<td>48.2</td>
<td>38.7</td>
</tr>
<tr>
<td>L-R reranking</td>
<td>54.9</td>
<td>44.1</td>
</tr>
<tr>
<td>CM infilling</td>
<td>69.0</td>
<td>56.3</td>
</tr>
<tr>
<td>PLBART code-cushman-001</td>
<td>41.6</td>
<td>—</td>
</tr>
<tr>
<td>code-davinci-001</td>
<td>53.1</td>
<td>42.0</td>
</tr>
<tr>
<td>code-davinci-001</td>
<td>63.0</td>
<td>56.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Pass Rate</th>
<th>Exact Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-R single</td>
<td>24.9</td>
<td>15.8</td>
</tr>
<tr>
<td>L-R reranking</td>
<td>28.2</td>
<td>17.6</td>
</tr>
<tr>
<td>CM infilling</td>
<td>38.6</td>
<td>20.6</td>
</tr>
<tr>
<td>PLBART code-cushman-001</td>
<td>13.1</td>
<td>—</td>
</tr>
<tr>
<td>code-davinci-001</td>
<td>30.8</td>
<td>17.4</td>
</tr>
<tr>
<td>code-davinci-001</td>
<td>37.8</td>
<td>19.8</td>
</tr>
</tbody>
</table>

(a) Single-line infilling.  
(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.
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

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

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

<table>
<thead>
<tr>
<th>Method</th>
<th>BLEU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ours: L-R single</td>
<td>16.05</td>
</tr>
<tr>
<td>Ours: L-R reranking</td>
<td>17.14</td>
</tr>
<tr>
<td>Ours: Causal-masked infilling</td>
<td>18.27</td>
</tr>
<tr>
<td>RoBERTa (Finetuned)</td>
<td>18.14</td>
</tr>
<tr>
<td>CodeBERT (Finetuned)</td>
<td>19.06</td>
</tr>
<tr>
<td>PLBART (Finetuned)</td>
<td>19.30</td>
</tr>
<tr>
<td>CodeT5 (Finetuned)</td>
<td>20.36</td>
</tr>
</tbody>
</table>

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

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

Counts the number of occurrences of each word in the given file.

:param filename: The name of the file to count.
:returns: 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

<table>
<thead>
<tr>
<th>Method</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left-to-right single</td>
<td>12.0</td>
</tr>
<tr>
<td>Left-to-right reranking</td>
<td>12.4</td>
</tr>
<tr>
<td>Causal-masked infilling</td>
<td>58.1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ours: Left-to-right single</td>
<td>30.8</td>
<td>30.8</td>
<td>30.8</td>
</tr>
<tr>
<td>Ours: Left-to-right reranking</td>
<td>33.3</td>
<td>33.3</td>
<td>33.3</td>
</tr>
<tr>
<td>Ours: Causal-masked infilling</td>
<td>59.2</td>
<td>59.2</td>
<td>59.2</td>
</tr>
<tr>
<td>TypeWriter (Supervised)</td>
<td>54.9</td>
<td>43.2</td>
<td>48.3</td>
</tr>
</tbody>
</table>

(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

<table>
<thead>
<tr>
<th>Method</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left-to-right single</td>
<td>18.4</td>
</tr>
<tr>
<td>Left-to-right reranking</td>
<td>23.5</td>
</tr>
<tr>
<td>Causal-masked infilling</td>
<td>30.6</td>
</tr>
</tbody>
</table>

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
```