Language Models for Code

March 13, 2024

CS 6501: Natural Language Processing

Ganesh Nanduru
Department of Computer Science
University of Virginia
Charlottesville, VA
bae9wk@virginia.edu

Nate Kimball
Department of Computer Science
University of Virginia
Charlottesville, VA
tma5gv@virginia.edu

Alex Fetea
Department of Computer Science
University of Virginia
Charlottesville, VA
pvn5nv@virginia.edu

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Background

- Code generation/editing is a popular use of LLMs
- Github Copilot has over 1 million paid users
- Every major AI developer has released a language model for code

code llama

GitHub Copilot

OpenAI Codex

Ghostwriter

Write better code, faster, with Replit's web-based coding AI.

replit

UNIVERSITY of VIRGINIA

SCHOOL of ENGINEERING & APPLIED SCIENCE

Papers

- InCoder: A Generative Model for Code Infilling and Synthesis
- Code Llama: Open Foundation Models for Code
- Teaching Large Language Models to Self-Debug
- LEVER: Learning to Verify Language-to-Code Generation with Execution

Papers

- **InCoder: A Generative Model for Code Infilling and Synthesis**
- Code Llama: Open Foundation Models for Code
- Teaching Large Language Models to Self-Debug
- LEVER: Learning to Verify Language-to-Code Generation with Execution

InCoder: A Generative Model for Code Infilling and Synthesis (ICLR 2023)

Daniel Fried[*♡†◇] **Armen Aghajanyan**[*♡] **Jessy Lin**[♣]
Sida Wang[♡] **Eric Wallace**[♣] **Freda Shi**[△] **Ruiqi Zhong**[♣]
Wen-tau Yih[♡] **Luke Zettlemoyer**[♡†] **Mike Lewis**[♡]

Facebook AI Research[♡] University of Washington[†]
UC Berkeley[♣] TTI-Chicago[△] Carnegie Mellon University[◇]
dfried@cs.cmu.edu, {armenag,mikelewis}@fb.com

https://arxiv.org/pdf/2204.05999.pdf

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Background

- Many LLMs generate responses left-to-right
- This approach is less applicable to code development
  - Mismatched tasks: debugging, commenting, refactoring
- Current strategies
  - Encoder-only masked LMs (e.g. BERT)
  - Encoder-decoder models (BART, T5)
  - **Decoder-only** (GPT, InCoder)

Objectives

- Train an LLM that can:
    - Synthesize code from scratch
    - Edit the user’s code
    - Infill blocks of code with context on either side

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

InCoder Overview

- Causal Masking
- Infilling, docstring generation, code generation

Causal Masking

- Causal modeling:
  - Only conditions on context to the left of the generated tokens
  - Good for generating large amounts of tokens autoregressively
- Masked modeling:
  - Condition on both left and right-side context
  - Generally only synthesize up to 15% of a document
- InCoder adopts both to combine their strengths

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Causal Masking

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>
```

Causal Masking

- InCoder will mask sequences of code by marking them with a Sentinel Token and moving them to the end.

- It marks masked sequences with <Mask> and marks the end-of-sequence insertions with <EOM>

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>
```

“Sentinel Tokens”

Causal-Masked Infilling

- To leverage causal masking while also using right-side context during inference, InCoder will temporarily fill in lines by inserting sentinel tokens
- InCoder will go back after a round of generation and populate the previously masked regions
- Useful for applications like docstrings, where both the function signature (left-side context) and function implementation (right-side context) are necessary

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Maximum Likelihood Estimation

- To train, InCoder generates tokens with the objective of maximizing the log-probability of the masked document:

$$\log P([\texttt{Left};\ \texttt{<Mask:0>};\ \texttt{Right};\ \texttt{<Mask:0>};\ \texttt{Span};\ \texttt{<EOM>}])$$

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

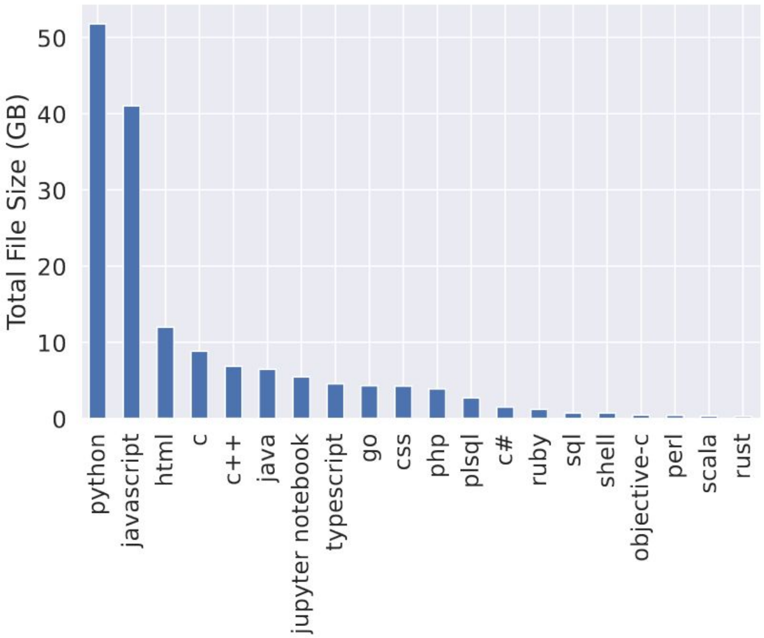
Training Data

- GitHub and GitLab open-source repositories
- Stack Overflow questions, answers, and comments
- Dataset is focused on Python code

Training Data

- 159 GB Dataset
  - 52 GB in Python
  - 57 GB from Stack Overflow
- Chart determined by file extension

A.5 CORPUS STATISTICS



UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

InCoder Models

- 1.3B and 6.7B Transformers
- 6.7B used for most evaluation purposes

Parameter	INCODER-1.3B	INCODER-6.7B
–decoder-embed-dim	2048	4096
–decoder-output-dim	2048	4096
–decoder-input-dim	2048	4096
–decoder-ffn-embed-dim	8192	16384
–decoder-layers	24	32
–decoder-normalize-before	True	True
–decoder-attention-heads	32	32
–share-decoder-input-output-embed	True	True
–decoder-learned-pos	False	False

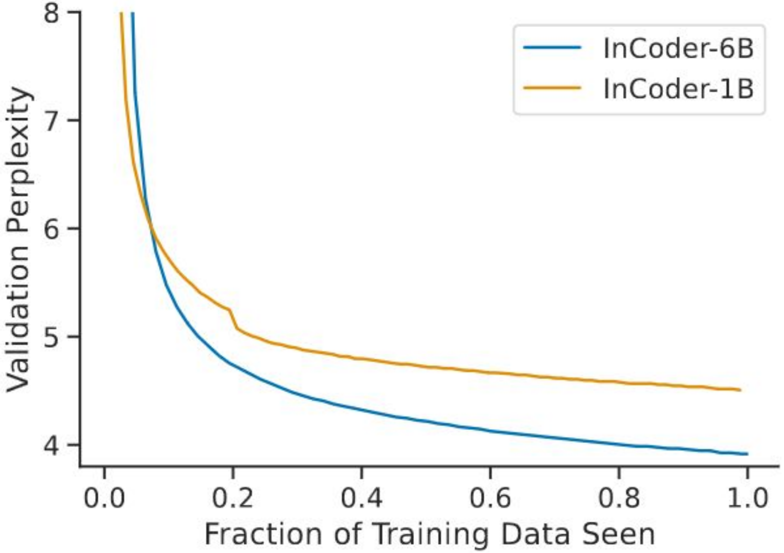
Table 6: Fairseq architecture hyperparameters for our INCODER models.

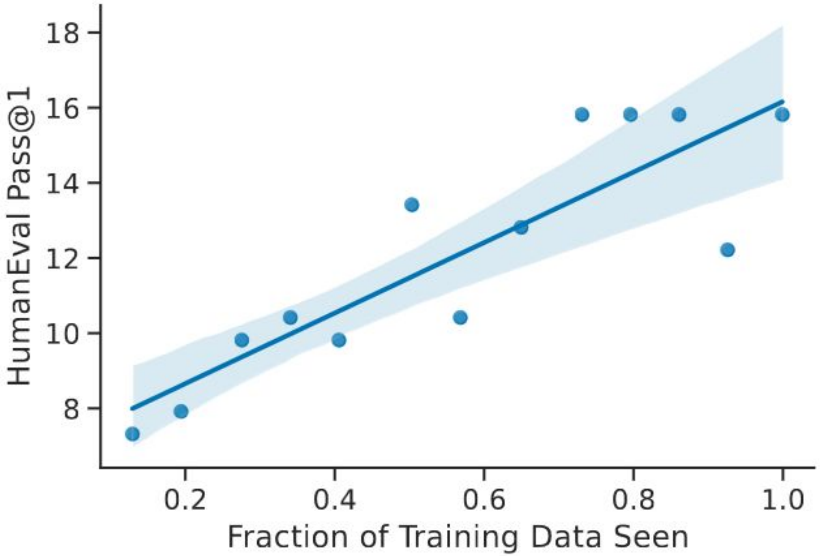
Training

- Trained on 248 V100 GPUs for 24 days
- One epoch on the training data, one pass over every document
- Implemented in PyTorch, uses its Adam optimizer
- GPU batch size of 8 and a maximum token sequence length of 2048

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Training





University of Virginia | School of Engineering & Applied Science

Inference

1. Left-to-right single: completely masks right context
2. Left-to-right reranking: masks the right context during generation, but not during selection
3. Causal-masked infilling

Inference done with nucleus sampling

Evaluation: Infilling Lines of Code

- Assessed on the HumanEval dataset
  - Includes comment descriptions of functions paired with canonical implementations
  - Includes sample function input-output pairs
- Evaluation metrics
  - Pass rate: the rate at which the function’s output matches the given input
  - Exact match: the percentage of lines identical to the canonical solution

Evaluation: Infilling Lines of Code

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

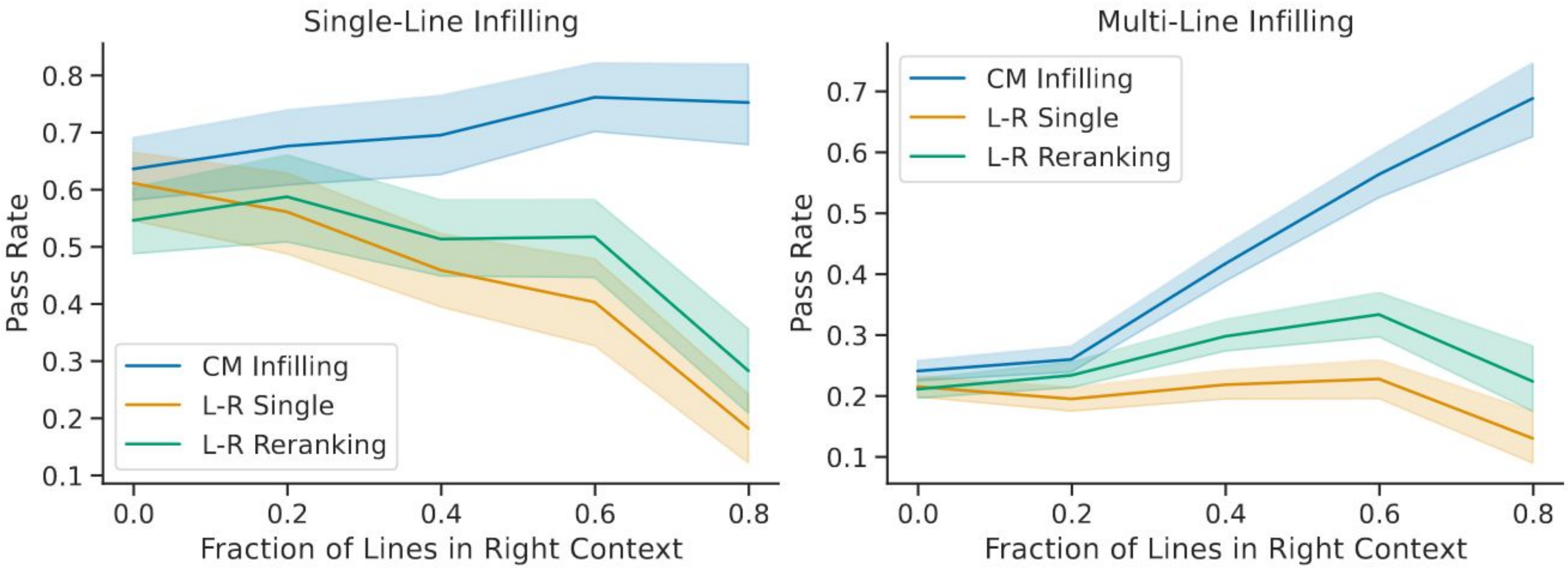
(a) Single-line infilling.

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.

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Evaluation: Infilling Lines of Code



UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Evaluation: Infilling Lines of Code

- Compared to OpenAI's code-davinci-002 proprietary API (August 2021)

Model	Inference	Pass Rate	Exact Match
INCODER-6.7B	Left-to-right single	48.2	38.7
INCODER-6.7B	Left-to-right reranking	54.9	44.1
INCODER-6.7B	Infilling	69.0	56.3
`code-davinci-002`	Left-to-right single	63.7	48.4
`code-davinci-002`	Left-to-right reranking	71.8	52.0
`code-davinci-002`	Infilling	87.4	69.6

Evaluation: Docstring Generation

- CodeXGLUE code-to-text docstring generation task
  - Uses the CodeSearchNet database, consisting of docstrings paired with corresponding code from public GitHub repositories
- Evaluation metric
  - BLEU score: how similar the LLM-generated docstring is to a set of high-quality references
    - Higher is better
- Compared against LLMs finetuned for docstring generation

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Evaluation: Docstring Generation

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

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Evaluation: Return Type Prediction

- Using CodeXGLUE again, this time isolating the return types of each function
- Second experiment done against TypeWriter, a supervised model specialized to determine input and return types for Python functions
    - Results evaluated using the Open-Source Software (OSS) dataset

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Evaluation: Return Type Prediction

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

Evaluation: Return Type Prediction

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

Conclusion

- Training a model to infill does not harm its ability to generate code left-to-right
- Causal masking is a useful tool for zero-shot performance on infilling and editing code
- Code LLMs that edit and annotate well can iteratively generate better code

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Potential Improvements and Critiques

- InCoder’s results are fairly weak
    - Did not compare well to SOTA language models for code
    - Could train for multiple passes over the data
    - Can increase dataset size and time spent training, as well as hardware
- Needs better benchmarking
    - Model was frequently compared to older versions of LLMs

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Related Work

- XL-Editor (2019)
  - Trains a language model to infill and edit natural language
  - https://arxiv.org/abs/1910.10479
- CM3 (2022)
  - Uses causal masking for left-to-right generation, but with bidirectional context
  - Strong results in zero-shot summarization and entity disambiguation
  - https://arxiv.org/abs/2201.07520

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Papers

- InCoder: A Generative Model for Code Infilling and Synthesis
- **Code Llama: Open Foundation Models for Code**
- Teaching Large Language Models to Self-Debug
- LEVER: Learning to Verify Language-to-Code Generation with Execution

Code Llama: Open Foundation Models for Code

Baptiste Rozière[†], Jonas Gehring[†], Fabian Gloeckle[†,*], Sten Sootla[†], Itai Gat, Xiaoqing Ellen Tan, Yossi Adi[◇], Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, Gabriel Synnaeve[†]

Meta AI

https://arxiv.org/pdf/2308.12950.pdf

Introduction to Code Llama

- Collection of models built upon Llama 2, specifically trained to solve programming problems
- Key features:
  - Generating code from brief descriptions
  - Filling gaps in existing code
  - Handling large inputs
- Foundational, Python, and Instruct variants

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Foundation of Code Llama: From Llama 2

- Code Llama fine-tuned from Llama 2
- Building on Llama 2 ensures understanding of natural and technical language
- Initializing the model with Llama 2 outperforms the same architecture trained on code only for a given budget

Infilling

- Improves code completion, type inference, and doc generation
- Employes causal masking, where parts of input are reordered and predicted autoregressively
- Training documents split into a prefix, a middle part and a suffix
- Formats include prefix-suffix-middle (PSM) and suffix-prefix-middle (SPM)

Long Context Fine-Tuning

- Sequence handling improved to 16,384 tokens from the initial 4,096
- Processing long sequences limited to a fine-tuning stage
- Modifies the rotation frequencies of rotary position embeddings (RoPE)
- Elevated base period from 10,000 to 1M allowing for larger sequences and ensuring model stability up to 100,000 tokens

Code Llama - Foundation Models

- Designed for IDEs to auto-complete and generate code
- Four size variants: 7B, 13B, 34B, and 70B parameters
- Infilling incorporated in 7B, 13B, and 70B models, with 34B focussed on code generation
- Trained on 500B tokens from a code-heavy dataset
- Long Context Fine-Tuning across all sizes

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Code Llama - Python

- Fine-tuned for Python to study the performance of models tailored to a single language
- Variants include 7B, 13B, 34B, and 70B parameters
- Trained on 500B tokens from the Code Llama dataset, and further specialized on 100B tokens from a Python-heavy dataset
- Optimized without infilling for the 7B to 34B models

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Code Llama - Instruct

- Designed for showing programming instructions via natural language, providing clear explanations for developers
- Available in 7B, 13B, and 34B sizes
- Models are based on Code Llama and fine-tuned with an additional about 5B tokens to better follow human instructions

Training Data and Strategy

- Trained on a near-deduplicated dataset of publicly available code
- 8% of samples data from natural language datasets related to code
- Adding natural language dataset improves the performance on Mostly Basic Programming Problems (MBPP)

Dataset	Sampling prop.	Epochs	Disk size
Code Llama (500B tokens)			
Code	85%	2.03	859 GB
Natural language related to code	8%	1.39	78 GB
Natural language	7%	0.01	3.5 TB
Code Llama - Python (additional 100B tokens)			
Python	75%	3.69	79 GB
Code	10%	0.05	859 GB
Natural language related to code	10%	0.35	78 GB
Natural language	5%	0.00	3.5 TB

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Instruction Fine Tuning

- **Proprietary Dataset:** Uses rich instruction data from Llama 2, enhancing model safety and instruction-following
- **Self-Instruct Dataset:**
  - Generated from 62,000 interview-style questions using Llama 2 70B
  - Deduplicated to 52,000 unique questions for variety
  - Code Llama 7B used for generating unit tests and ten Python solutions per question
  - First passing solution of each question included, resulting in ~14,000 problem-solution pairs
- **Rehearsal:** Training includes code dataset (6%) and natural language dataset (2%)

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

HumanEval and MBPP Benchmark Results

- Widely used description-to-code generation benchmarks
- Computed with temperature 0.8
- Zero-shot on HumanEval, 3-shot on MBPP

Model	Size	HumanEval			MBPP		
		pass@1	pass@10	pass@100	pass@1	pass@10	pass@100
code-cushman-001	12B	33.5%	-	-	45.9%	-	-
GPT-3.5 (ChatGPT)	-	48.1%	-	-	52.2%	-	-
GPT-4	-	67.0%	-	-	-	-	-
PaLM	540B	26.2%	-	-	36.8%	-	-
PaLM-Coder	540B	35.9%	-	88.4%	47.0%	-	-
PaLM 2-S	-	37.6%	-	88.4%	50.0%	-	-
StarCoder Base	15.5B	30.4%	-	-	49.0%	-	-
StarCoder Python	15.5B	33.6%	-	-	52.7%	-	-
StarCoder Prompted	15.5B	40.8%	-	-	49.5%	-	-
Llama 2	7B	12.2%	25.2%	44.4%	20.8%	41.8%	65.5%
	13B	20.1%	34.8%	61.2%	27.6%	48.1%	69.5%
	34B	22.6%	47.0%	79.5%	33.8%	56.9%	77.6%
	70B	30.5%	59.4%	87.0%	45.4%	66.2%	83.1%
Code Llama	7B	33.5%	59.6%	85.9%	41.4%	66.7%	82.5%
	13B	36.0%	69.4%	89.8%	47.0%	71.7%	87.1%
	34B	48.8%	76.8%	93.0%	55.0%	76.2%	86.6%
	70B	53.0%	84.6%	96.2%	62.4%	81.1%	91.9%
Code Llama - Instruct	7B	34.8%	64.3%	88.1%	44.4%	65.4%	76.8%
	13B	42.7%	71.6%	91.6%	49.4%	71.2%	84.1%
	34B	41.5%	77.2%	93.5%	57.0%	74.6%	85.4%
	70B	67.8%	90.3%	97.3%	62.2%	79.6%	89.2%
Unnatural Code Llama	34B	62.2%	85.2%	95.4%	61.2%	76.6%	86.7%
Code Llama - Python	7B	38.4%	70.3%	90.6%	47.6%	70.3%	84.8%
	13B	43.3%	77.4%	94.1%	49.0%	74.0%	87.6%
	34B	53.7%	82.8%	94.7%	56.2%	76.4%	88.2%
	70B	57.3%	89.3%	98.4%	65.6%	81.5%	91.9%

University of Virginia | School of Engineering & Applied Science

Performance on Multi-Language Benchmarks

- Pass@1 scores
- Computed in zero-shot

Model	Size	Multi-lingual Human-Eval						
		C++	Java	PHP	TS	C#	Bash	Average
CodeGen-Multi	16B	21.0%	22.2%	8.4%	20.1%	8.2%	0.6%	13.4%
CodeGeeX	13B	16.9%	19.1%	13.5%	10.1%	8.5%	2.8%	11.8%
code-cushman-001	12B	30.6%	31.9%	28.9%	31.3%	22.1%	11.7%	26.1%
StarCoder Base	15.5B	30.6%	28.5%	26.8%	32.2%	20.6%	11.0%	25.0%
StarCoder Python	15.5B	31.6%	30.2%	26.1%	32.3%	21.0%	10.5%	25.3%
Llama-v2	7B	6.8%	10.8%	9.9%	12.6%	6.3%	3.2%	8.3%
	13B	13.7%	15.8%	13.1%	13.2%	9.5%	3.2%	11.4%
	34B	23.6%	22.2%	19.9%	21.4%	17.1%	3.8%	18.0%
	70B	30.4%	31.7%	34.2%	15.1%	25.9%	8.9%	24.4%
Code Llama	7B	28.6%	34.2%	24.2%	33.3%	25.3%	12.0%	26.3%
	13B	39.1%	38.0%	34.2%	29.6%	27.3%	15.2%	30.6%
	34B	47.8%	45.6%	44.1%	33.3%	30.4%	17.1%	36.4%
	70B	52.8%	51.9%	50.9%	**49.1%**	**38.0%**	29.1%	45.3%
Code Llama - Instruct	7B	31.1%	30.4%	28.6%	32.7%	21.6%	10.1%	25.8%
	13B	42.2%	40.5%	32.3%	39.0%	24.0%	13.9%	32.0%
	34B	45.3%	43.7%	36.6%	40.3%	31.0%	19.6%	36.1%
	70B	53.4%	**58.2%**	**58.4%**	39.0%	36.7%	**29.7%**	**45.9%**
Code Llama - Python	7B	32.3%	35.4%	32.3%	23.9%	24.7%	16.5%	27.5%
	13B	39.1%	37.3%	33.5%	35.2%	29.8%	13.9%	31.5%
	34B	42.2%	44.9%	42.9%	34.3%	31.7%	14.6%	35.1%
	70B	**54.7%**	57.6%	53.4%	44.0%	34.8%	25.3%	45.0%

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Infilling Training Evaluation

Model	FIM	Size	HumanEval pass@1	HumanEval pass@10	HumanEval pass@100	MBPP pass@1	MBPP pass@10	MBPP pass@100	Test loss
Code Llama (w/o LCFT)	✗	7B	33.2%	43.3%	49.9%	44.8%	52.5%	57.1%	0.408
		13B	36.8%	49.2%	57.9%	48.2%	57.4%	61.6%	0.372
Code Llama (w/o LCFT)	✓	7B	33.6%	44.0%	48.8%	44.2%	51.4%	55.5%	0.407
		13B	36.2%	48.3%	54.6%	48.0%	56.8%	60.8%	0.373
Absolute gap	✗ - ✓	7B	−0.4%	−0.7%	1.1%	0.6%	1.1%	1.6%	0.001
		13B	0.7%	0.9%	3.3%	0.2%	0.6%	0.8%	−0.001

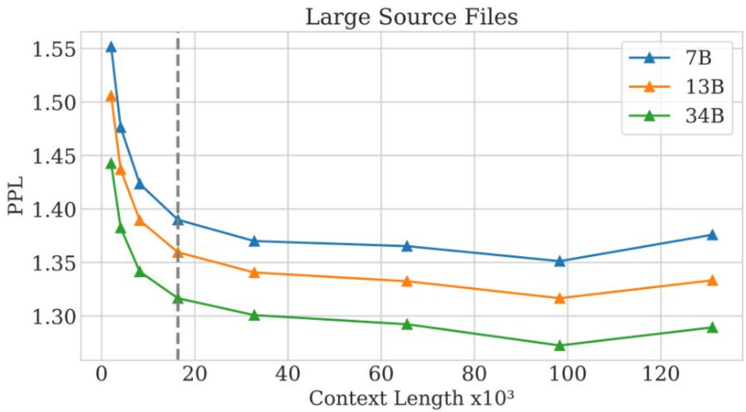
Trained with and without infilling and temperature of 0.1

Multilingual HumanEval single line infilling

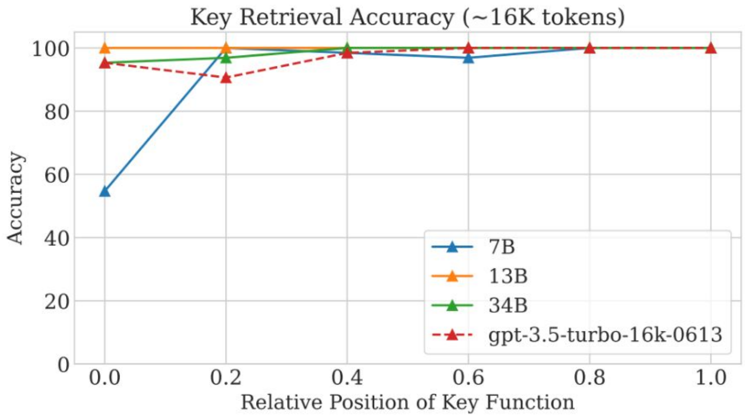
Model	Size	Python PSM	Python SPM	Java PSM	Java SPM	JavaScript PSM	JavaScript SPM
InCoder	6B		31.0%		49.0%		51.0%
SantaCoder	1.1B		44.0%		62.0%		60.0%
StarCoder	15.5B		62.0%		73.0%		74.0%
Code Llama	7B	67.6%	72.7%	74.3%	77.6%	80.2%	82.6%
	13B	**68.3%**	**74.5%**	**77.6%**	**80.0%**	**80.7%**	**85.0%**

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Long Context Fine Tuning Evaluations



(a)



(b)

Single Line Completion Performance

- Exact Match vs Bilingual Evaluation Understudy
- With and without LCFT

Model								
			EM	BLEU	EM	BLEU	EM	BLEU
Code Llama	7B	✗	36.86	60.16	47.82	69.20	46.29	67.75
Code Llama	7B	✓	**39.23**	**61.84**	**51.94**	**71.89**	**50.20**	**70.22**
Code Llama	13B	✗	37.96	61.33	50.49	69.99	49.22	69.87
Code Llama	13B	✓	**41.06**	**62.76**	**52.67**	**72.29**	**52.15**	**71.00**
Code Llama	34B	✗	42.52	63.74	54.13	72.38	52.34	71.36
Code Llama	34B	✓	**44.89**	**65.99**	**56.80**	**73.79**	**53.71**	**72.69**

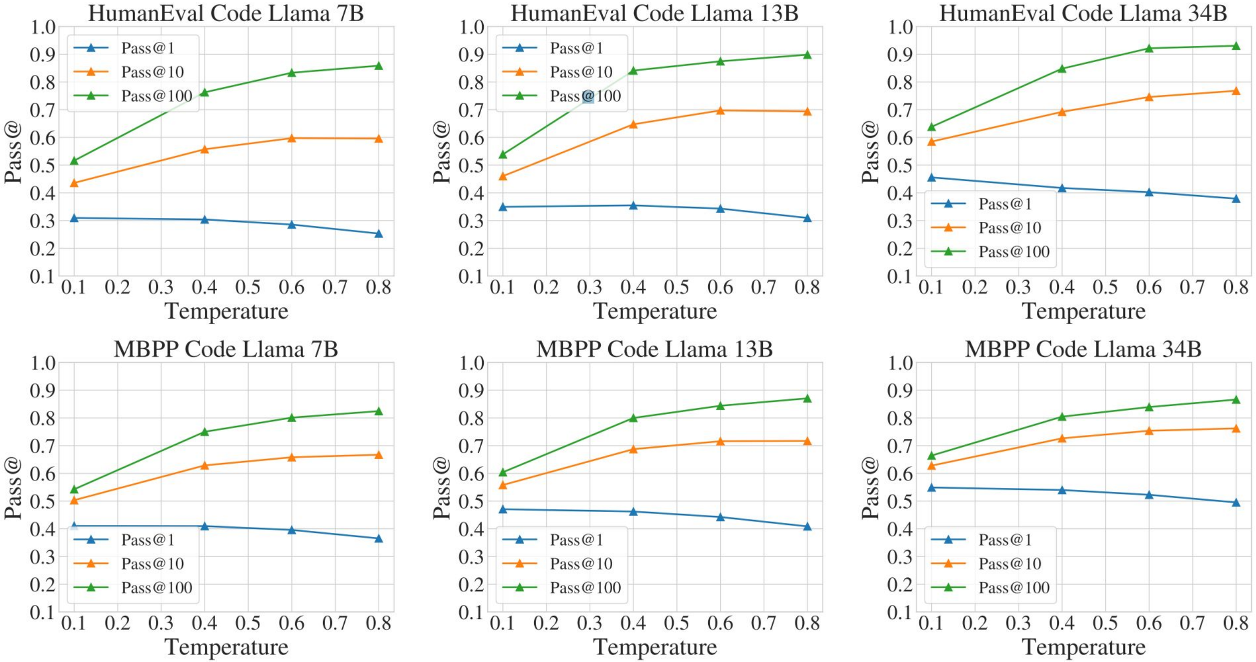
UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Impact of Self-Instruct Data

Size	SI	HumanEval	MBPP 3-shot	MBPP zero-shot
7B	✗	30.5%	43.4%	37.6%
	✓	34.8%	44.4%	37.4%
13B	✗	40.9%	46.2%	20.4%
	✓	42.7%	49.4%	40.2%

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Code Llama Performance Across Different Temperatures



UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Conclusion

- Achieved top performance in single-line code infilling
- Significant performance gains with larger models
- Strong ability in managing large code contexts

Limitations

- Limited context performance
- Unclear decision when choosing which model to use
- Performance on a variety of coding tasks

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Papers

- InCoder: A Generative Model for Code Infilling and Synthesis
- Code Llama: Open Foundation Models for Code
- **Teaching Large Language Models to Self-Debug**
- LEVER: Learning to Verify Language-to-Code Generation with Execution

Google DeepMind

Teaching Large Language Models to Self-Debug

Xinyun Chen[1] **Maxwell Lin**[2] **Nathanael Schärli**[1] **Denny Zhou**[1]
[1] Google DeepMind [2] UC Berkeley
{xinyunchen,schaerli,dennyzhou}@google.com, mxlin@berkeley.edu

https://arxiv.org/pdf/2304.05128.pdf

University of Virginia | School of Engineering & Applied Science

Background

- Language models for code
- Autoregressive nature of LLMs does not mesh with how humans code
- Prompting techniques like chain-of-thought significantly improve programming
- Recent work shows language models have potential for generating feedback messages to critique and refine their outputs

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Introduction

- Zero-shot coding is very challenging
- Instead of discarding incorrect code, investigate results, then make changes to resolve the implementation error
  - Prior methods train separate model for code repairing
- SELF-DEBUGGING teaches an LLM to debug its program via few shot demonstrations; no additional training needed
- Analogous to rubber-duck debugging, debugging without external feedback

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

SELF-DEBUGGING Framework

1. Generation step: problem description -> candidates
2. Feedback step: message concerning correctness of code determined by unit tests or by asking the model
3. Explanation step: model processes its own prediction, either by explaining it or creating an execution trace

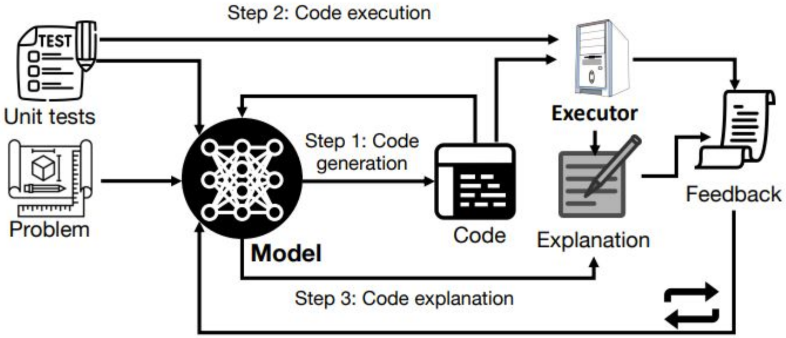


Figure 1: SELF-DEBUGGING for iterative debugging using a large language model. At each debugging step, the model first generates new code, then the code is executed and the model explains the code. The code explanation along with the execution results constitute the feedback message, based on which the model infers the code correctness and then adds this message to the feedback. The feedback message is then sent back to the model to perform more debugging steps. When unit tests are not available, the feedback can be purely based on code explanation.

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Prompting for Code Generation

- They use few shot prompting for initial code attempt
- They decode multiple samples, using majority voting on execution results to select predicted code
- When unit tests are present, they filter out programs that do not pass unit tests

Feedback

In practice, not all forms of feedback are available

- Simple feedback: sentence indicating code correctness, no explanation step
- Unit test feedback (UT): message containing execution results
- Code Explanation feedback (Expl): rubber duck debugging; the model describes the code and compares it to the problem description
- Execution trace feedback (Trace): the model explains the execution steps line-by-line

Feedback

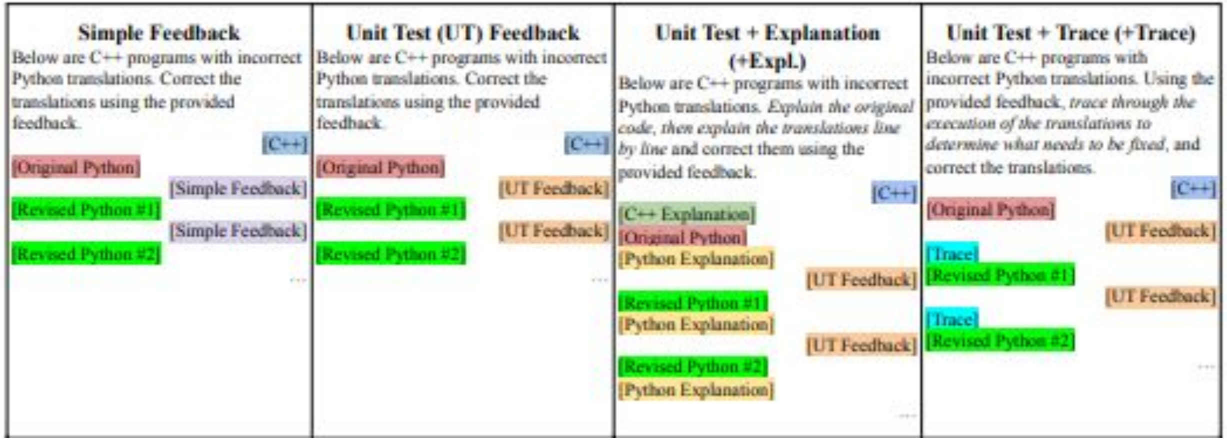


Figure 5: Examples of SELF-DEBUGGING prompts for code translation. Left-aligned blocks are model predictions, and right-aligned blocks contain the input C++ code and feedback messages based on code execution. The full prompts are in Appendix F.

Applications

- Text-to-SQL
  - Spider dataset
  - Unit tests are not available
- Code translation
  - TransCoder dataset (C++-to-Python)
  - Abundance of unit tests
- Text-to-Python
  - MBPP dataset
  - Only a subset of unit tests are presented in problem description

Step 1: Question explanation

Infer the return type of the question.

[Question]

Which customers have both "On Road" and "Shipped" as order status? List the customer names.

[Question Explanation]

"List the customer names" returns 1 column. The question returns the customer names who have both "On Road" and "Shipped" as order status. So the question returns 1 column.

Step 2: Code explanation

Summarize the return type of the execution.

[SQL]

SELECT ustomers.customer_name FROM customers JOIN orders ON customers.customer_id = orders.customer_id
WHERE orders.order_status = "On Road" OR orders.order_status = "Shipped"
Execution:
| George |

[SQL Explanation]

The execution of the SQL query above would return a table with 1 column.
The first column, "customers.customer_name" would contain the customer names.
With "customers JOIN orders", the table would contain the data about customers with orders.
With "WHERE orders.order_status = 'On Road' OR orders.order_status = 'Shipped'", the table filters the records to only include customers who have order status "On Road" or "Shipped".
So the SQL query returns a table with 1 column, the customer names who have the order status "On Road" or "Shipped".

Step 3: Self-Debugging with explanation

[Question]

[Explanation summary]

As in your explanation, the SQL query returns a table with 1 column, the customer names who have the order status "On Road" or "Shipped". The question returns 1 column, the customer names who have both "On Road" and "Shipped" as order status.

[Feedback]

So the SQL prediction above is wrong. Please fix the SQL.

[Revised SQL #1]

SELECT customers.customer_name FROM customers JOIN orders ON customers.customer_id = orders.customer_id
WHERE orders.order_status = "On Road" AND orders.order_status = "Shipped"

[Explanation summary #1]

As in your explanation, the SQL query returns an empty table, because the "WHERE" clause contains mutually exclusive conditions. The question returns 1 column, the customer names who have order status of both "On Road" and "Shipped".

[Feedback #1]

So the SQL prediction above is wrong. Please fix the SQL.

[Revised SQL #2]

SELECT customers.customer_name FROM customers JOIN orders ON customers.customer_id = orders.customer_id WHERE orders.order_status = "On Road" INTERSECT SELECT customers.customer_name FROM customers JOIN orders ON customers.customer_id = orders.customer_id WHERE orders.order_status = "Shipped"

[Explanation summary #2]

As in your explanation, the SQL query returns a table with 1 column, the customer names who have the order status "On Road" and "Shipped". The question returns 1 column, the customer names who have order status of both "On Road" and "Shipped". So the question returns 1 column.

[Feedback #2]

So the SQL prediction above is correct!

Figure 3: An example of SELF-DEBUGGING prompting for text-to-SQL generation. The SQL query, explanation and feedback are all predicted by the model. When the returned table has more than 2 rows, only the first 2 rows are included in the prompt. Database information is omitted in the figure for clarity, and we present the full prompts in Appendix E.

Experiments

They evaluate SELF-DEBUGGING against two types of code reranking baselines

- Models fine-tuned on Spider
    - T5-3B, LEVER
- Prompting-based approaches
    - MBR-Exec, Coder-Reviewer

Table 1: Comparing SELF-DEBUGGING to prior ranking techniques.

(a) Results on the Spider development set.

	Spider (Dev)
w/ training	
T5-3B + N-best Reranking	80.6
LEVER (Ni et al., 2023)	81.9
Prompting only w/o debugging	
Coder-Reviewer	74.5
MBR-Exec	75.2
SELF-DEBUGGING (this work)	
Codex	81.3
+ Expl.	**84.1**

(b) Results on MBPP dataset.

	n samples
Prior work	
MBR-Exec	63.0 ($n = 25$)
Reviewer	66.9 ($n = 25$)
LEVER	68.9 ($n = 100$)
SELF-DEBUGGING (this work)	
Codex	72.2 ($n = 10$)
Simple	73.6
UT	75.2
UT + Expl.	**75.6**

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

SELF-DEBUGGING with different feedback formats

Table 2: Results of SELF-DEBUGGING with different feedback formats.

(a) Results on the Spider development set.

Spider	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	81.3	71.1	73.2	64.7
Simple	81.3	**72.2**	73.4	**64.9**
+Expl.	**84.1**	**72.2**	**73.6**	**64.9**

(b) Results on TransCoder.

TransCoder	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	80.4	89.1	77.3	70.0
Simple	89.3	91.6	80.9	72.9
UT	91.6	**92.7**	88.8	76.4
+ Expl.	**92.5**	**92.7**	**90.4**	**76.6**
+ Trace.	87.9	92.3	89.5	73.6

(c) Results on MBPP.

MBPP	Codex	GPT-3.5	GPT-4	StarCoder
Baseline	61.4	67.6	72.8	47.2
Simple	68.2	70.8	78.8	50.6
UT	69.4	72.2	**80.6**	52.2
+ Expl.	69.8	**74.2**	80.4	52.2
+ Trace.	**70.8**	72.8	80.2	**53.2**

Ablation Studies

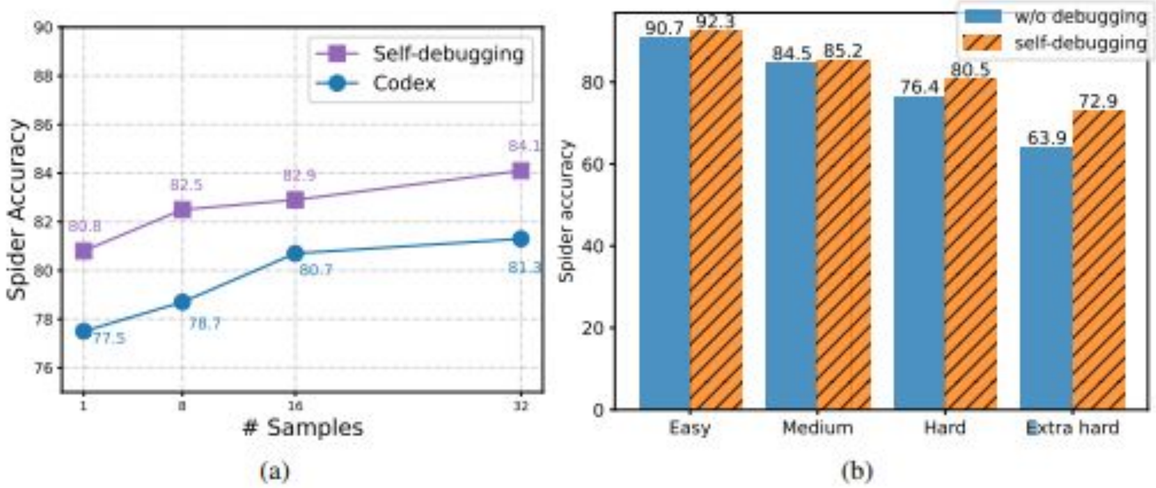


Figure 6: Ablation studies on the Spider development set with Codex. (a) Accuracies with different numbers of initial samples. (b) Breakdown accuracies on problems with different hardness levels.

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

SELF-DEBUGGING without unit test-execution

Table 3: Results of SELF-DEBUGGING without unit test execution.

(a) Results on Transcoder.

TransCoder	Codex	GPT-3.5	GPT-4
Baseline	80.4	**89.1**	77.3
Simple	83.4	**89.1**	78.2
+ Expl.	**83.9**	**89.1**	78.0
+ Trace.	**83.9**	**89.1**	**78.4**

(b) Results on MBPP

MBPP	Codex	GPT-3.5	GPT-4
Baseline	61.4	67.6	72.8
Simple	57.6	68.2	76.0
+ Expl.	64.4	68.2	76.0
+ Trace.	**66.2**	**69.2**	**76.4**

Breakdown of error types

Table 4: Breakdown on percentages of error types fixed by SELF-DEBUGGING.

(a) Breakdown on Spider with `code-davinci-002`.

Error type	%
Wrong WHERE conditions	25.7
Missing the DISTINCT keyword	17.1
Wrong JOIN clauses	14.3
Wrong number of SELECT columns	11.4
Wrong INTERSECT/UNION clauses	8.6
Wrong aggregate functions and keywords	5.8
Wrong COUNT columns	5.7
Wrong column selection	5.7
Missing nested conditions	5.7

(b) Breakdown on Transcoder with `gpt-3.5-turbo`, and MBPP with `gpt-4`.

Error type	Transcoder	MBPP
Output mismatch	61.9	69.2
Runtime errors	38.1	30.8

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Conclusion

- Achieves state-of-the-art performance across several code generation domains and notably improves sample efficiency
- Shows the importance of iteratively debugging output
- They hypothesize better code explanation ability leads to better debugging performance

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Limitations

- Depends on the code explanation ability of the model
- It is possible for the model to think the code is correct when it is not
- It is possible for the code to pass all unit tests and still be incorrect
- Results are not uniform across problem difficulties (more improvement on harder problems than easier ones)
- In practice, unit tests are not always present, and SELF-DEBUGGING brings minimal improvement when unit tests are absent

Papers

- InCoder: A Generative Model for Code Infilling and Synthesis
- Code Llama: Open Foundation Models for Code
- Teaching Large Language Models to Self-Debug
- **LEVER: Learning to Verify Language-to-Code Generation with Execution**

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

LEVER: Learning to Verify Language-to-Code Generation with Execution (ICML 2023)

Ansong Ni [1 †] **Srini Iyer** [2] **Dragomir Radev** [1] **Ves Stoyanov** [2] **Wen-tau Yih** [2] **Sida I. Wang** [2 *] **Xi Victoria Lin** [2 *]

[†]Majority of the work done during an internship at Meta AI. [*]Equal contribution [1]Yale University [2]Meta AI. Correspondence to: Ansong Ni <ansong.ni@yale.edu>, Xi Victoria Lin <victorialin@meta.com>, Sida I. Wang <sida@meta.com>.

https://arxiv.org/pdf/2302.08468.pdf

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Background

- LLMs produce the correct output more often when more samples are drawn
- By sampling at scale, the effectiveness of training a verifier to rank solutions increases
  - Verifiers assess model outputs for accuracy and consistency, providing language models with feedback to improve responses
- Verifiers have proved to be useful in helping language models choose the correct output to math problems
- Can they be expanded to solving coding problems?

Objectives

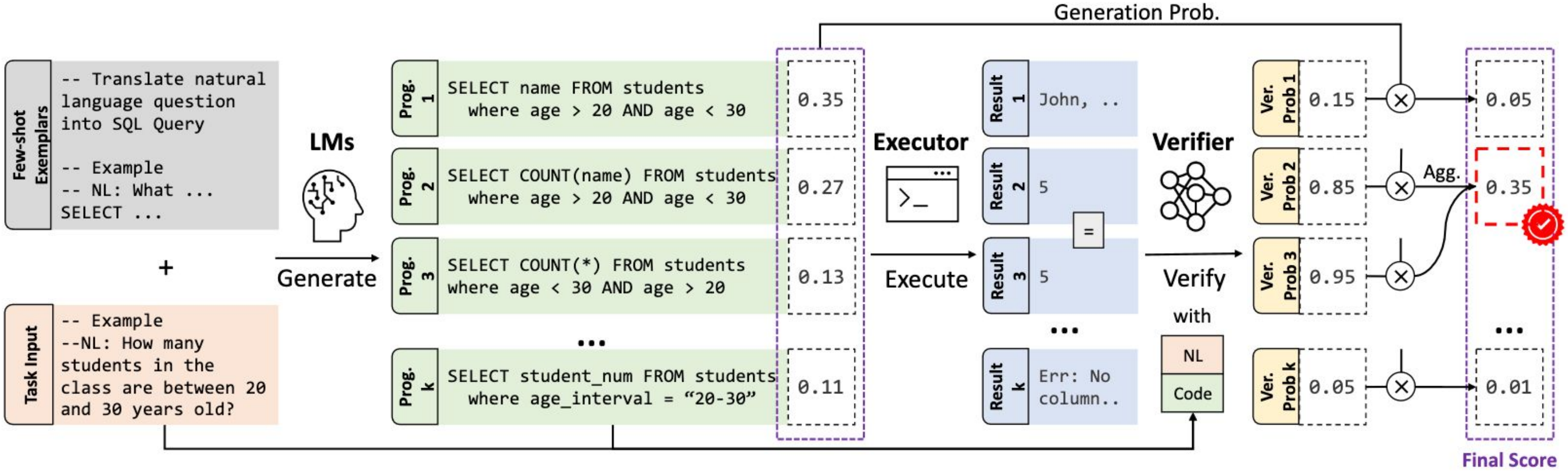
– Train a verifier to distinguish and reject incorrect code outputs
– Use the verifier to produce more correct code

University of Virginia | School of Engineering & Applied Science

LEVER Overview

- **Le**arning to **Ver**ify language-to-code generation
- Three step approach
  - Generation: create code samples from a task and few-shot exemplars
  - Execution: run the generated code
  - Verification: assess the generated code, natural language input, and execution summary and output probabilities of each code sample being correct

LEVER Process



Reranking

- Objective is to rank the code outputs by correctness and suitability to the task
- Reranking probability: Joint probability of generation and passing the verification step

$$P_R(\hat{y}, v_{=1}|x) = P_{\mathbf{LM}}(\hat{y}|x) \cdot P_\theta(v_{=1}|x, \hat{y}, \mathcal{E}(\hat{y}))$$

P_R = reranking probability

$P_{LM}(\hat{y})$ = likelihood of ŷ being generated

$P_\theta(\hat{y})$ = likelihood of ŷ producing the correct output

x = inputs (task, exemplars)

$\hat{y}$ = program being assessed

$\mathcal{E}(y)$ = execution results of y

Reranking

- Once all samples are generated and reranking probabilities are calculated, a final reranking score is given to each sample
- Final reranking score: aggregate probability of the other generated programs to have the same execution output as the program being assessed

$$R(x, \hat{y}) = P_R(\mathcal{E}(\hat{y}), v_{=1} | x) = \sum_{y \in S, \mathcal{E}(y) = \mathcal{E}(\hat{y})} P_R(y, v_{=1} | x)$$

R = final reranking score

S = all generated programs

P_R = reranking probability

x = inputs (task, exemplars)

$\hat{y}$ = program being assessed

$\mathcal{E}(y)$ = execution results of y

Training Data

- Generally for language-to-code datasets, each training data point is a triplet of natural language input, canonical code solution, and the code solution's output
- This requires supervision, since you have to annotate the programs for correctness / input-output pairs
- LEVER expands this idea by including self-generated candidate programs as canonical code solution, if their execution results match the code solution's output

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Training Data

- Spider (2018): NL to SQL queries
- WikiTQ (2015): **Wiki T**able **Q**uestions, table question answering dataset
- GSM8K (2021): **G**rade **s**chool **m**ath problems and solutions
- MBPP (2021): Python programs and test cases

	Spider	WikiTQ	GSM8k	MBPP
Domain	Table QA	Table QA	Math QA	Basic Coding
Has program	✓	✓*	✗	✓
Target	SQL	SQL	Python	Python
Data Statistics				
# Train	7,000	11,321	5,968	378
# Dev	1,032	2,831	1,448	90
# Test	-	4,336	1,312	500
Few-shot Generation Settings				
Input Format	Schema + NL	Schema + NL	NL	Assertion + NL
# Shots	8‡	8	8	3
# Samples (train / test)	20/50†	50/50	50/100	100/100
Generation Length	128	128	256	256

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Evaluation: Models

- LEVER just wants to train the verifier, not its own code generator.
- Researchers used three different code LLMs:
  - Codex (2021): A set of OpenAI code LLMs. Researchers used the code-davinci-002 model.
  - InCoder (2022): The first paper presented today, developed largely by Meta AI.
  - CodeGen (2022): A code LLM developed by Salesforce

Evaluation: Baselines

- Greedy: pick the most likely token each decoding step
- Maximum Likelihood (ML): of the code samples generated, select the one with the highest generation log-probability
- Error Pruning + ML: add a preliminary step to remove code samples with execution errors
- Error Pruning + Voting: remove code with execution errors, then majority-vote on the remaining samples

Evaluation

- Metrics
  - Execution accuracy: Percentage of examples that pass all test cases
- Fine-tuned on T5-Base model

Evaluation: Results

- Spider dataset
- Small increase from EP + ML baseline

Methods	Dev
Previous Work without Finetuning	
Rajkumar et al. (2022)	67.0
MBR-Exec (Shi et al., 2022)	75.2
Coder-Reviewer (Zhang et al., 2022)	74.5
Previous Work with Finetuning	
T5-3B (Xie et al., 2022)	71.8
PICARD (Scholak et al., 2021)	75.5
RASAT (Qi et al., 2022)	80.5
This Work with code-davinci-002	
Greedy	75.3
EP + ML	77.3
LEVER	**81.9**$_{\pm 0.1}$

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

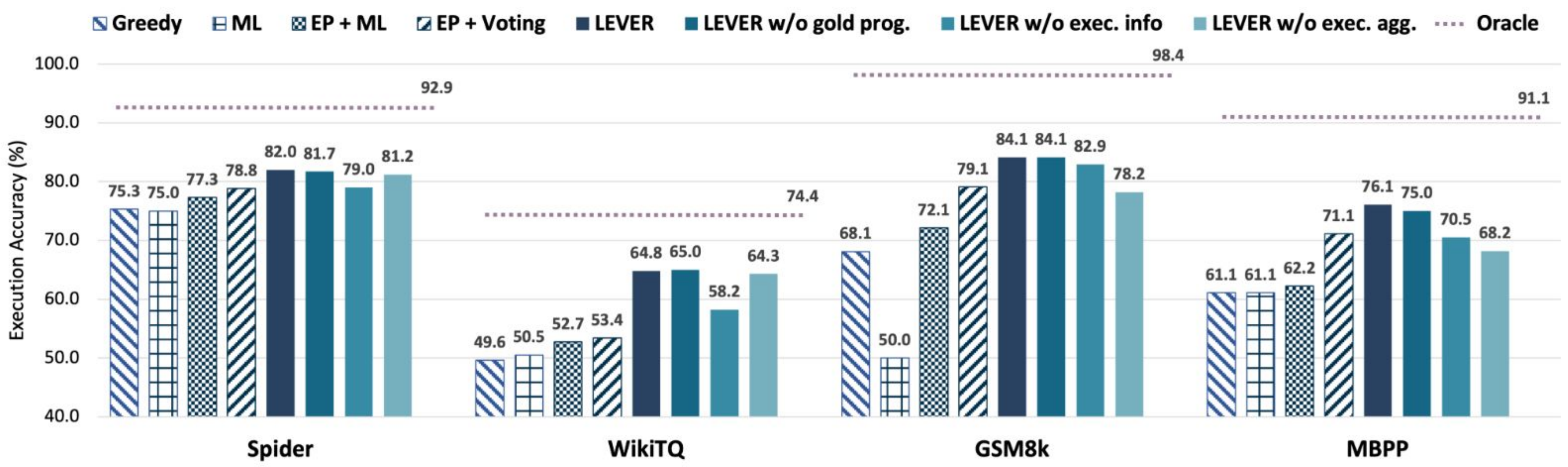
Evaluation: Results

- GSM8K
- Much more notable increase in eval accuracy
- The dataset is not a code base!

Methods	**Dev**	**Test**
Previous Work without Finetuning		
PAL (Gao et al., 2022)	-	72.0
Codex + SC† (Wang et al., 2022)	-	78.0
PoT-SC (Chen et al., 2022b)	-	80.0
Previous Work with Finetuning		
Neo-2.7B + SS (Ni et al., 2022)	20.7	19.5
Neo-1.3B + SC (Welleck et al., 2022)	-	24.2
DiVeRSe*† (Li et al., 2022b)	-	83.2
This Work with codex-davinci-002		
Greedy	68.1	67.2
EP + ML	72.1	72.6
LEVER	**84.1**$_{\pm 0.2}$	**84.5**$_{\pm 0.3}$

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Evaluation: Results



University of Virginia | School of Engineering & Applied Science

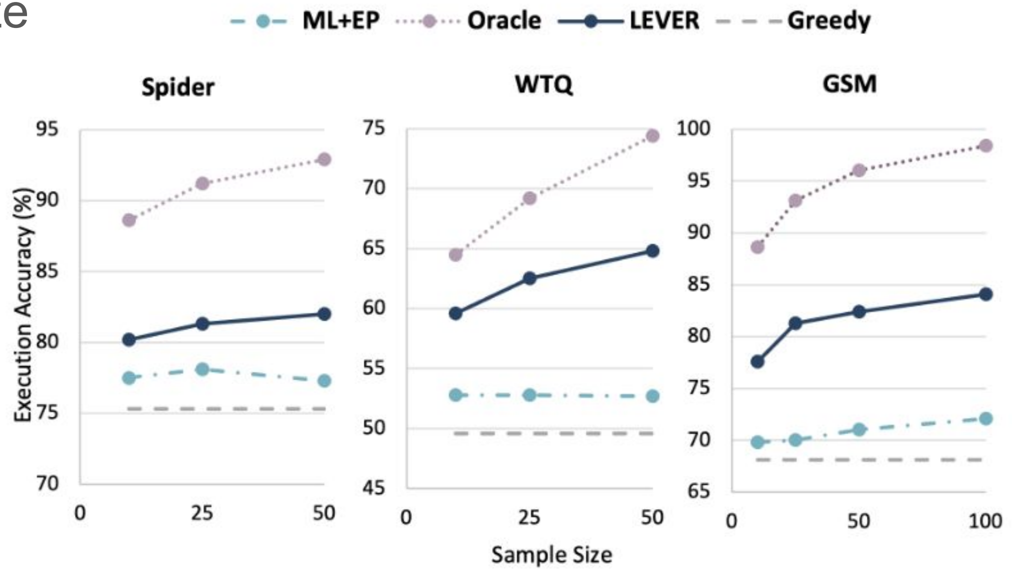
Evaluation: Results

- All baselines included
- **Oracle**: ideal performance obtained by selecting the correct program if it appears in the sample set

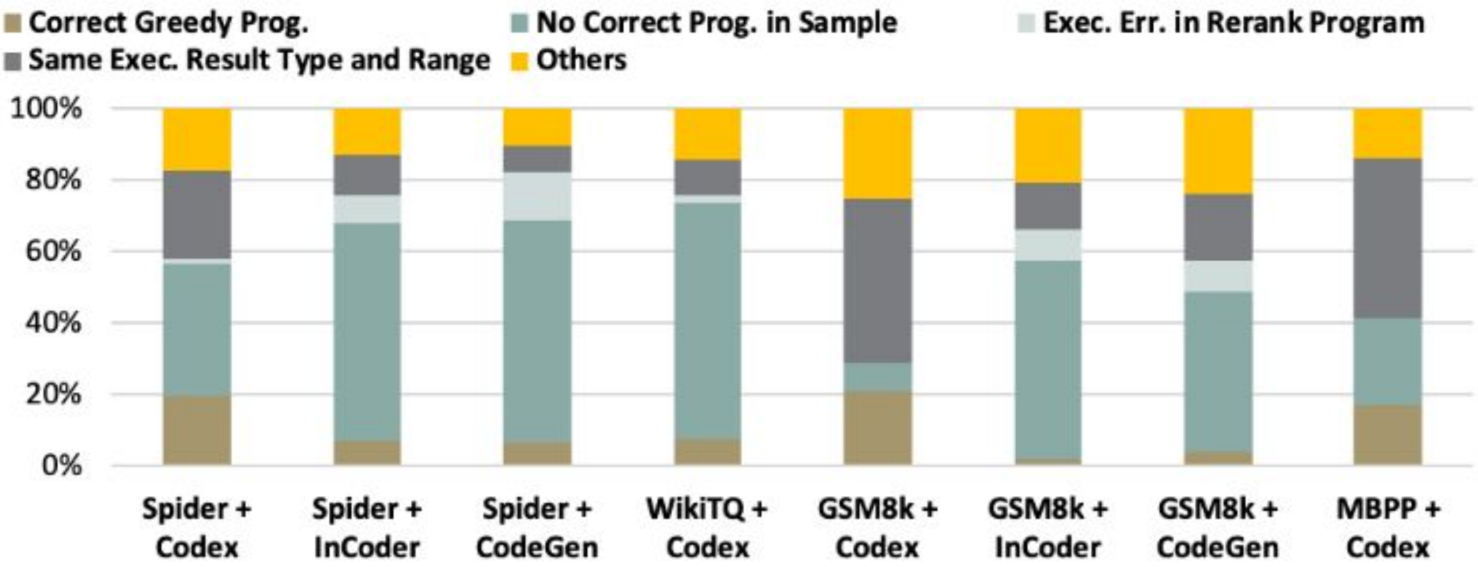
Methods	InCoder-6B		CodeGen-16B	
	Spider	GSM8k	Spider	GSM8k
Previous work:				
MBR-EXEC	38.2	-	30.6	-
Reviewer	41.5	-	31.7	-
Baselines:				
Greedy	24.1	3.1	24.6	7.1
ML	33.7	3.8	31.2	9.6
EP + ML	41.2	4.4	37.7	11.4
EP + Voting	37.4	5.9	37.1	14.2
LEVER	54.1	**11.9**	51.0	**22.1**
– gold prog.	53.4	-	**52.3**	-
– exec. info	48.5	5.6	43.0	13.4
– exec. agg.	**54.7**	10.6	51.6	18.3
Oracle	71.6	48.0	68.6	61.4

Evaluation: Results

- Ablation testing on sample size at inference time



When Does LEVER Fail?



UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Conclusion

- Using a verifier can improve the execution accuracy of code LLMs, and will almost never decrease their correctness.
- However, developing a verifier is an extra step in training.
- Verifiers can also impact inference time.

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Related Work

- AlphaCode (2022)
  - Uses majority voting based on execution results of samples
  - https://arxiv.org/abs/2203.07814
- DIVERSE (2022)
  - Meant to solve math problems with LLMs (GSM8K)
  - Trains a verifier to verify each step of an LLM's problem-solving output.
  - https://arxiv.org/abs/2206.02336

Improvements / Critiques

- More details on the time and computation costs for training and inference with verifiers
- Generalize a verifier to work with less context
- More metrics than just the execution accuracy during evaluation
- Justify why the resources and effort put into verification is worth it

Papers

- InCoder: A Generative Model for Code Infilling and Synthesis
- Code Llama: Open Foundation Models for Code
- Teaching Large Language Models to Self-Debug
- LEVER: Learning to Verify Language-to-Code Generation with Execution

Takeaways

- Language models are effective at generating functional code from task inputs
  - Even more effective when given in-context exemplar code
- Bidirectional context is helpful, even for left-to-right generation
- Long context can increase understanding of large code bases
- LLM code output can be improved by reranking the generated samples using a separate verifier
- LLMs can edit code, so they can also iteratively improve their own responses
- There is a need for annotated data for “gold examples” of code generation

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE

Questions / Comments?

UNIVERSITY of VIRGINIA | SCHOOL of ENGINEERING & APPLIED SCIENCE