Syntax-Aware Language Models for Programming Code

Mateusz Kitzol Patryk Rybak

University of Wroclaw

June 23, 2025

Agenda

Motivation

Why syntax-aware code generation with LLMs?

- **Challenge:** Training large LLMs from scratch is prohibitively expensive for most applications.
- **Opportunity:** Small transformer models can be fine-tuned effectively on code tasks using targeted techniques.
- **Approach:** This work explores **syntax-aware training methods** that guide models to better capture the structure and semantics of Python code.
- **Goal:** Evaluate whether such techniques improve generation quality in small models—and assess their potential for scaling to larger models and broader use cases.
- **Format:** The docstring–function format is a practical and widely available setup for training code generation models.

Dataset & Pre-processing

Source: CodeSearchNet

- Originally includes ∼**450,000** code functions.
- Effectively reduced to ∼**100,000** usable functions.
- Data split:
  - 80% training
  - 10% validation
  - 10% test (evaluated on **1,000** functions)

Preprocessing Pipeline Overview

Sequential Steps in the Pipeline:

1. **RemoveComments**
2. **DuplicateFilter**
3. **Pep8Formatter**
4. **SyntaxValidator**.

Goal: Ensure the dataset contains clean, unique, well-formatted, and valid code examples before training.

RemoveComments Preprocessor

Purpose:

- Cleans code by removing inline comments and docstrings.
- Simplifies input for downstream processing or model training.

How it works:

- Removes all inline comments using regular expressions.
- Removes multi-line docstrings enclosed in `'''...'''` or `"""..."""`.
- Returns the modified example with a cleaned `"code"` field.

DuplicateFilter Preprocessor

Purpose:

- Removes duplicate code examples based on their SHA-256 hash.
- Ensures the dataset contains only unique code snippets.

How it works:

- Computes a hash of the `"code"` field in each example.
- Checks whether the hash has already been seen.
- If duplicate, returns `None`; otherwise, stores the hash and returns the example.

PEP8 Formatter

What is PEP8?

- PEP8 is the official style guide for Python code.
- It promotes readability and consistency across Python codebases.

What does `autopep8` do?

- Automatically reformats Python code to comply with PEP8 standards.
- Fixes common issues such as:
  - Improper indentation
  - Line length violations
  - Inconsistent spacing
  - Unused imports

SyntaxValidator Preprocessor

Purpose:

- Filters out syntactically invalid Python code.
- Ensures only parseable code is used in training or evaluation.

How it works:

- Attempts to parse the code using Python's `ast` module.
- If parsing succeeds, the example is returned.
- If a syntax error occurs, the example is discarded (`None` is returned).

Code Normalization Example

```
# Before Preprocessing:
def bbox(self):
        """BBox"""
        return self.left, self.top, self.right, self.
            bottom

# After Preprocessing:
def bbox(self):
    return self.left, self.top, self.right, self.
        bottom
```

Listing 1: Example of Function Before and After Preprocessing

Comment: This example illustrates a common preprocessing step: removing docstrings and normalizing indentation. Such transformations help reduce variation in code style that is irrelevant for model training.

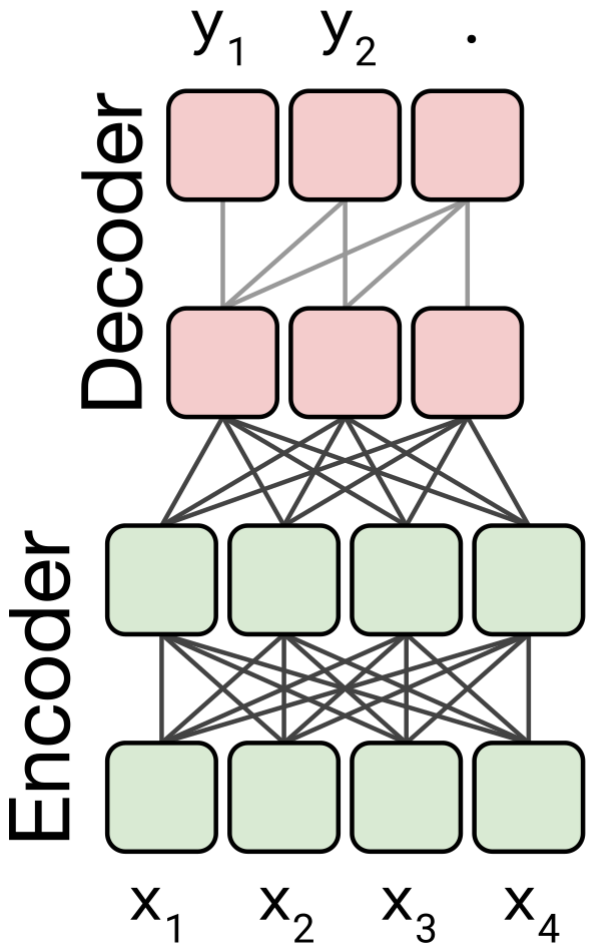
T5 – the model

Text-to-Text Transfer Transformer
But why exactly T5?

T5 – the model

But why exactly T5?

- Encoder-decoder architecture



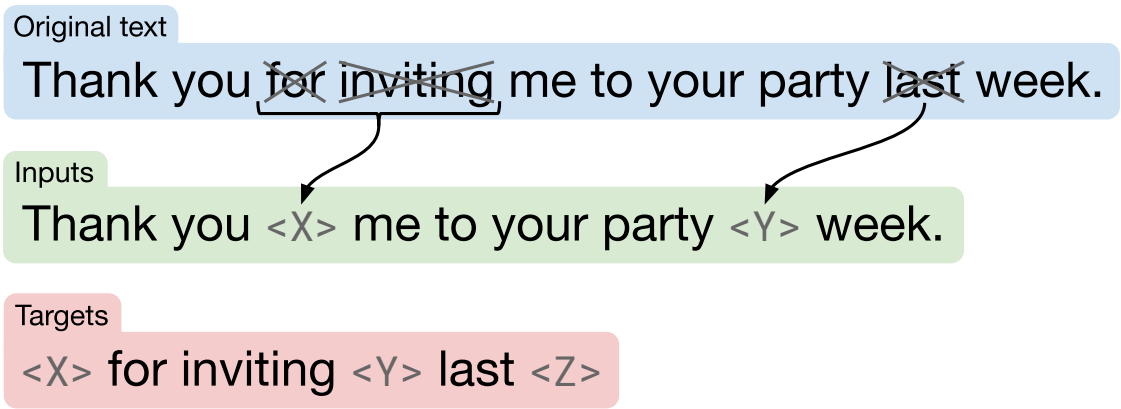
Separate volume for code modeling

Understanding the intent of docstrings

T5 – the model

But why exactly T5?

- Encoder-decoder architecture
- Denoising objective



T5 – the model

But why exactly T5?

- Encoder-decoder architecture
- Denoising objective
- Publications

T5 – the model

But why exactly T5?

- Encoder-decoder architecture
- Denoising objective
- Publications

T5 Version	Parameter Count
Small	60M
Base	220M
Large	770M
3B	2.8B
11B	11B

Evaluation Metrics — Two Perspectives

Why multiple metrics?

- **Lexical Overlap Metrics** — Judge surface-level similarity between prediction and reference code.
- **Functional Metrics** — Verify the behaviour of generated code by running it.

Together, they capture both *how* the code is written (tokens) and *what* the code does (execution).

BLEU — Precision-Oriented Lexical Metric

Core Idea: Reward n-grams in the prediction that also occur in the reference.

- **Modified n-gram Precision**: Counts matching n-grams, clipped to the maximum reference count.
- **Brevity Penalty (BP)**: Dampens scores for overly short predictions.
- **Final Score**: Geometric mean of precisions over $n = 1 \ldots 4$, scaled by BP.

Precision vs. Recall

BLEU emphasises **precision**: it asks, "Of all tokens the model produced, how many are correct?" It does *not* explicitly reward covering every reference token (recall).

ROUGE — Balancing Precision & Recall

Family of recall-aware metrics

- **ROUGE-1/2**: Overlap of unigrams and bigrams.
- **ROUGE-L**: Length of the Longest Common Subsequence (LCS).

For each variant we compute:

$$\text{Precision, Recall, F1} = \frac{2PR}{P+R}$$

Interpretation

Unlike BLEU, ROUGE explicitly captures **recall** — "How much of the reference did the model recover?" — while still reporting precision. The F1 score balances both.

HumanEval — Functional Correctness

Benchmark: 164 docstring-guided coding tasks

- Each task provides a *docstring*, function *signature*, and *hidden unit tests*.
- A prediction passes if it runs without error and satisfies **all** tests.

Precision & Recall in HumanEval

A passing solution must be **precise** (no wrong behaviour) making this metric stricter and more task-specific than lexical overlap scores.

Inference-Time Hyperparameters

- **Sampling**:
  - `True` – selects next token randomly based on predicted distribution.
  - `False` – always picks the most probable token (greedy decoding).
- **Temperature** (used only when Sampling = `True`):
  - $< 1 \rightarrow$ makes the distribution sharper, more deterministic.
  - $> 1 \rightarrow$ makes the distribution flatter, more diverse.
- **Number of Beams**:
  - Controls how many parallel sequences are explored during decoding.
  - Higher values help avoid poor early decisions.
- **Length Penalty** (when beams > 1):
  - $< 0 \rightarrow$ favors shorter outputs.
  - $> 0 \rightarrow$ favors longer completions.

Experiments

- T5 Baseline
- Finetuned T5
- Finetuned T5 + Pretokenizer
- Finetuned T5 + Pretokenizer + Segmentator
- Finetuned T5 + Pretokenizer + LogitsProcessor with Custom Loss

Baseline: T5

Objective: Assess the performance of the unmodified T5-Large model on Python code generation.

- **Model choice:** T5-Large was selected as the largest feasible variant for our hardware setup.
- **Limitation:** T5 was pretrained on general natural language—not code—resulting in poor adaptation to Python syntax and structure.
- **Observation:**
  - Predictions often contain loosely related words from the input.
  - Occasionally include Python-like tokens, but lack any coherent code logic.
- **Result:** Evaluation metrics were consistently low.
- **Conclusion:** No hyperparameter tuning was attempted.

Baseline: T5

Metric	T5 Baseline
BLEU	0.0173
Precision@1	0.1505
Precision@2	0.0207
Precision@3	0.0085
Precision@4	0.0034
Brevity Penalty	1.0000
Translation Length	10288
Reference Length	9001
Length Ratio	1.1430

Table: BLEU Metrics for T5 Baseline

Baseline T5

Metric	T5 Baseline
ROUGE-1	0.1536
ROUGE-2	0.0330
ROUGE-L	0.1402

Table: ROUGE Metrics for T5 Baseline

T5 Finetuned

Objective: Establish a strong baseline by fine-tuning T5-Large on docstring–code pairs.

- **Setup:**
  - Inputs: Docstrings.
  - Targets: Reference code snippets.
  - No syntax-aware processing or domain knowledge (yet).
  - Each example is seen 3 times.
  - Subset of the dataset used, with reference code capped at 512 tokens.
  - Inference performed using **greedy decoding**, which proved to be the most effective setting.
- **Note on fairness:**
  - Settings differ across experiments to accommodate each model's strengths.
  - Uniform configurations would be inefficient and may limit model performance.

Example: Finetuned T5 Prediction

Input: `Using the blob object, write the file to the destination path`

Reference:

```
def write_file(self, blob, dest):
    with salt.utils.files.fopen(dest, 'wb+') as fp_:
        blob.stream_data(fp_)
```

Prediction:

```
def write_blob(blob, destination):
    with open(destination, 'wb') as f:
        f.write(blob.encode('utf-8'))
```

Comment: Prediction captures the general intent but simplifies the behavior: instead of streaming binary content from a blob-like object, it assumes `blob` is a UTF-8 encodable string and writes it directly to the file.

T5 Finetuned

Metric	T5 Baseline	Finetuned T5
BLEU	0.0173	0.1782
Precision@1	0.1505	0.4992
Precision@2	0.0207	0.2390
Precision@3	0.0085	0.1413
Precision@4	0.0034	0.0894
Brevity Penalty	1.0000	0.9045
Translation Length	10288	43550
Reference Length	9001	47919
Length Ratio	1.1430	0.9088

Table: BLEU Metrics: Finetuned T5 in comparison to T5 baseline

T5 Finetuned

Metric	T5 Baseline	Finetuned T5
ROUGE-1	0.1536	0.3913
ROUGE-2	0.0330	0.1499
ROUGE-L	0.1402	0.3570

Table: ROUGE Metrics: Finetuned T5 in comparison to T5 baseline

Finetuned T5 + Pretokenizer

Objective: Improve performance on abstracted code using structural tags.

- **Setup:**
  - Inputs: Docstrings (same as previously).
  - Targets: Abstracted code snippets with tags.
  - Tags based on AST representation.
  - Each example is seen 3 times.
  - Subset of the dataset used, with reference target capped at 512 tokens.
  - Experimental best inference setting:
    - temperature - 0.1
    - beam search - 4 beams
    - length penalty - 1.3
- **Note:** Structural tags increase sequence length, forcing exclusion of longer examples and slightly reducing dataset size.

Pretokenizer: syntax-aware tags

- **Goal:** Explicitly separate code structure (syntax) from semantics.
- **Based on AST:** Uses Python's `ast` module to walk the syntax tree.
- **Visitor pattern:** Modified internal `_Unparser` for precise node handling.

Pretokenizer: syntax-aware tags

- **Goal:** Explicitly separate code structure (syntax) from semantics.
- **Based on AST:** Uses Python's `ast` module to walk the syntax tree.
- **Visitor pattern:** Modified internal `_Unparser` for precise node handling.
- **Control part:**
  - Keywords, operators, brackets, colons, indentation, etc.
  - Replaced with **abstract tags**, e.g. `[DEF]`, `[IF]`, `[BLOCK]`

Pretokenizer: syntax-aware tags

- **Goal:** Explicitly separate code structure (syntax) from semantics.
- **Based on AST:** Uses Python's `ast` module to walk the syntax tree.
- **Visitor pattern:** Modified internal `_Unparser` for precise node handling.
- **Control part:**
  - Keywords, operators, brackets, colons, indentation, etc.
  - Replaced with **abstract tags**, e.g. `[DEF]`, `[IF]`, `[BLOCK]`
- **Semantic part:**
  - Variable/function names, literals, strings, identifiers
  - Wrapped with `[SEMANTIC_START]` and `[SEMANTIC_END]`

Pretokenizer: syntax-aware tags

- **Goal:** Explicitly separate code structure (syntax) from semantics.
- **Based on AST:** Uses Python's `ast` module to walk the syntax tree.
- **Visitor pattern:** Modified internal `_Unparser` for precise node handling.
- **Control part:**
  - Keywords, operators, brackets, colons, indentation, etc.
  - Replaced with **abstract tags**, e.g. `[DEF]`, `[IF]`, `[BLOCK]`
- **Semantic part:**
  - Variable/function names, literals, strings, identifiers
  - Wrapped with `[SEMANTIC_START]` and `[SEMANTIC_END]`
- **Purpose:** In later experiment enables tokenizer to distinguish code structure from meaning.

Pretokenizer: example

Original code:

```
def add(a, b):
    return a + b
```

Pretokenized representation:

```
[DEF] [SEMANTIC_START] add [SEMANTIC_END]
[DELIMIT_1_L]
    [SEMANTIC_START] a [SEMANTIC_END] [COMMA]
    [SEMANTIC_START] b [SEMANTIC_END]
[DELIMIT_1_R] [BLOCK]
[INDENT]
    [RETURN] [SEMANTIC_START] a [SEMANTIC_END]
        [ADD] [SEMANTIC_START] b [SEMANTIC_END]
    [DEDENT]
```

Example: Finetuned T5 + Pretokenizer Prediction

Input: `Pass through to provider AssessmentSearchSession.get_assessments_by_search`

Reference:

```
def get_assessments_by_search(self, assessment_query,
    assessment_search):
    if not self._can("search"):
        raise PermissionDenied()
    return self._provider_session.get_assessments_by_search(
        assessment_query, assessment_search)
```

Prediction:

```
def get_assessments_by_search(self, assessment_query,
    assessment_search):
    if not self._can("search"):
        raise PermissionDenied()
    return self._provider_session.get_assessments_by_search(
        assessment_query, assessment_search)
```

Finetuned T5 + Pretokenizer

Metric	T5 Baseline	Finetuned T5	Pretokenizer T5
BLEU	0.0173	0.1782	0.2097
Precision@1	0.1505	0.4992	0.5321
Precision@2	0.0207	0.2390	0.2706
Precision@3	0.0085	0.1413	0.1647
Precision@4	0.0034	0.0894	0.1092
Brevity Penalty	1.0000	0.9045	0.9298
Translation Length	10288	43550	40703
Reference Length	9001	47919	43666
Length Ratio	1.1430	0.9088	0.9321

Table: BLEU Metrics: Pretokenizer T5

Finetuned T5 + Pretokenizer

Metric	T5 Baseline	Finetuned T5	Pretokenizer T5
ROUGE-1	0.1536	0.3913	0.4077
ROUGE-2	0.0330	0.1499	0.1600
ROUGE-L	0.1402	0.3570	0.3774

Table: ROUGE Metrics: Pretokenizer T5

Segmentator: Motivation and Objective

Goal: Improve the span corruption objective used in T5-like models by applying it to **syntactically coherent code segments**.

- Traditional span corruption masks arbitrary spans of tokens.
- Code has rich syntactic structure – we aim to exploit that.
- **Idea:** Replace arbitrary masking with structure-aware masking guided by Python syntax.
- **Tool:** The `segmentator` enables flexible, syntax-aware span selection.

Segmentator: Step 1 – Extracting Code Spans

Step 1: Identify Syntactic Units

- Tags inserted during pretokenization (e.g., `[DEF]`, `[RETURN]`, `[RAISE]`) define segment boundaries.
- Spans are selected using a flexible set of configuration flags (see next slide).
- Output: a list of candidate spans in the form `(start_idx, end_idx)`.

Span Extraction: Configurable Options

The `segmentator` uses flexible flags to control which spans to extract:

- **control_tags**: Extracts multi-line blocks like `[IF]`, `[FOR]`, `[DEF]`, and `[CLASS]`.
- **inline_tags**: Captures single-line constructs such as `[RETURN]`, `[DEL]`, or `[RAISE]`.
- **delimiters**: Selects spans enclosed in delimiters like parentheses or brackets.
- **lines**: Enables segmentation based on line breaks and indentation tags.
- **indented_blocks**: Groups blocks of indented code using `[INDENT]` and `[DEDENT]`.
- **semantic**: Focuses on semantically coherent units between `[SEMANTIC_START]` and `[SEMANTIC_END]`.
- **all_options**: Activates all extraction modes simultaneously.
- **strict**: Filters overlapping or nearly duplicated spans when set to `False`.

Segmentator: Step 2 – Sampling and Cleaning

Step 2: Prepare Spans for Masking

- **Clean overlapping or nested spans** to ensure non-interfering masking.
- **Sample a subset** of the remaining spans using:
  - *Min/Max masking ratio* – control how much of the input is masked.
  - *Max number of spans* – limit how many are used per sequence.
- Final result: a clean, random subset of spans ready to be masked and replaced with `<extra id X>` tokens.

Segmentator: Input/Label Formatting

Input (with masked spans):

```
Subtracts two vectors.
[DEF]...<extra id 0>
[BLOCK][INDENT][RETURN]...
<extra id 1>...[DEDENT]
```

Labels (targets for reconstruction):

```
<extra id 0>...<extra id 1>...<extra id 2>...
```

Note:

- Each <extra id i> replaces a syntactic span in the input.
- Labels concatenate these IDs and the corresponding masked spans.
- Enables more structured learning aligned with code syntax.

T5 + Pretokenizer + Segmentator

Objective: Include segmentator as additional first finetuning step

- **Setup:**
  - Training schedule: 1 epoch for the *segmentator*, followed by 2 epochs of *pretokenizer* fine-tuning.
  - The same set of hyperparameters as before turns out to be the best one
    - temperature - 0.1
    - beam search - 4 beams
    - length penalty - 1.3
- **Results:**
  - **BLEU:** 0.18 – 0.20
  - **ROUGE-L:** 0.34 – 0.38

Example: Finetuned T5 + Pretokenizer + Segmentator Prediction

Input: `Get list of telegram recipients of a hook`

```
---
serializer:  TelegramRecipientSerializer
responseMessages:
    - code:  401
      message:  Not authenticated
```

Reference:

```
def get(self, request, bot_id, id, format = None):
    return super(TelegramRecipientList, self).get(request, bot_id,
        id, format)
```

Prediction:

```
def telegram_recipients(self, request, bot_id, id, format = None):
    return super(TelegramRecipientList, self).get(request, bot_id,
        id, format)
```

BLEU-based Metrics for Different Experiments

Metric	Baseline	Finetuned	Pretokenizer	Segmentator
BLEU	0.0173	0.1782	**0.2097**	0.2017
Precision@1	0.1505	0.4992	**0.5321**	0.5171
Precision@2	0.0207	0.2390	**0.2706**	0.2552
Precision@3	0.0085	0.1413	**0.1647**	0.1517
Precision@4	0.0034	0.0894	**0.1092**	0.0978
Brevity Penalty	1.0000	0.9045	0.9298	**0.9586**
Translation Length	10288	43550	40703	**41895**
Reference Length	9001	47919	43666	43666
Length Ratio	1.1430	0.9088	0.9321	**0.9594**

Table: BLEU-based Metrics for Different Experiments

ROUGE Metrics for Different Experiments

Metric	Baseline	Finetuned	Pretokenizer	Segmentator
ROUGE-1	0.1536	0.3913	**0.4077**	0.4000
ROUGE-2	0.0330	0.1499	**0.1600**	0.1494
ROUGE-L	0.1402	0.3570	**0.3774**	0.3684

Table: ROUGE Metrics for Different Experiments

A potential bottleneck

- Previous code generation methods often reuse the same token vocabulary for both natural language input (e.g., docstrings) and code-specific control tags.
- This means that the same token can have multiple, context-dependent meanings.
- As a result, the model has to encode a wider range of dependencies using a shared embedding space.

Solution: Introduce a separate set of control tokens to disambiguate model behavior.

A potential bottleneck

- Adding new control tokens introduces new embeddings.
- Without a dedicated training objective, their interaction with original semantic tokens may cause interference.
- This can lead to a form of **knowledge degradation**.

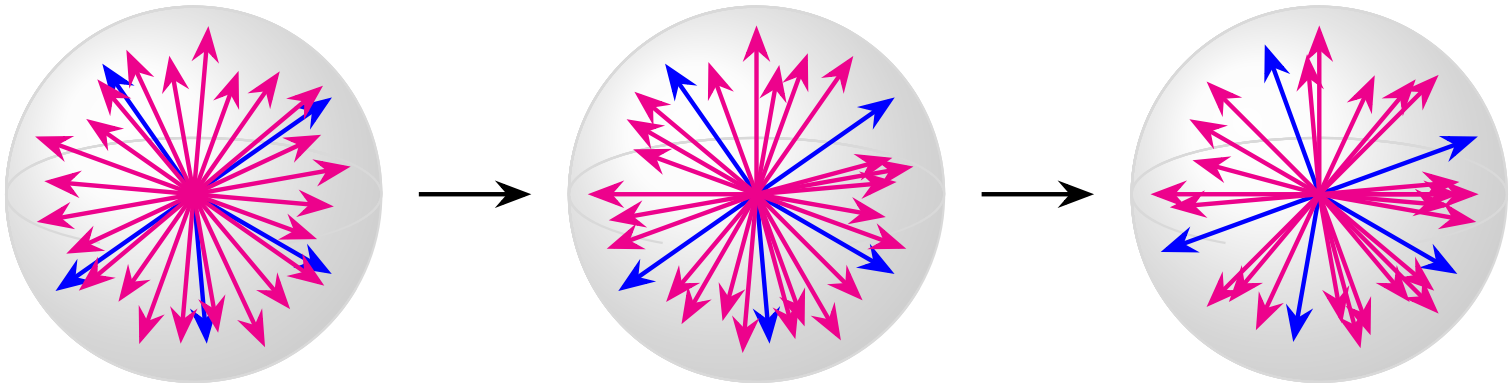


Figure: Potential knowledge degradation – pink vectors drift away from the blue ones, losing some of their learned dependencies

Finetuned T5 + Pretokenizer + LogitsProcessor with Custom Loss

Goal: Gain better performance via dodging knowledge degradation and seperating control tags.

- **Setup:**
  - Separation of tokens into **semantic** and **control** groups
  - All tokens = control tokens (for tags) + everything beyond that (kinda)
  - Custom loss function = **selective learning** and **classification**
  - Subset of the dataset used, with reference target capped at 512 tokens
  - 3 epochs
  - Control and semantic generation modes (inferance)
- **Note:** Now, tokens cover whole tags - tokenized sequence lengths have decreased

Token groups

- Let $\mathcal{C}$ be the set of control tokens, $\mathcal{C} = \{[\mathrm{IF}], [\mathrm{BLOCK}], \ldots\}$
- Let $\mathcal{S}$ be the set of semantic tokens, $\mathcal{S} = \{\mathrm{if}, \_\mathrm{Ala}, \ldots\}$

- Remove: $\mathcal{C}' = \mathcal{C} \setminus \{[\mathrm{SEMANTIC\_STOP}]\}$
- Add: $\mathcal{C}'' = \mathcal{C}' \cup \{\mathrm{custom\_eos}\}$

- Remove: $\mathcal{S}' = \mathcal{S} \setminus \{[\mathrm{SEMANTIC\_START}], \mathrm{original\_eos}\}$

Custom Loss – Selective Learning

- Mitigates semantic drift of tokens during fine-tuning.
- Differentiates between control and semantic token groups.
- Backpropagation occurs only for predictions within the correct group (kinda).

Expected Token Group	Predicted Token Group	Backpropagation?
Control	Control (mismatch)	Yes
Control	Semantic	No
Semantic	Semantic (mismatch)	Yes
Semantic	Control	No

Custom Loss – Token Sequence Example

1. [C1] [C2] [C3] ...
2. [C1] [C2] [C3] [SEMANTIC_START] [S1] [S2] ...

reversed condition

3. [C1] [C2] [C3] [SEMANTIC_START] ...
4. [C1] [SEMANTIC_START] [S1] [S2] [SEMANTIC_STOP] ...

Custom Loss – Classifier

- Selective loss slows training.
- Classifier predicts token group (semantic/control) from hidden state (last decoder layer).
- Output: binary label; trained with Binary Cross-Entropy.

$$\mathcal{L}_{\text{classifier}} = \text{BinaryCrossEntropy}(\text{prediction}, \text{target group})$$

Total Loss:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{selective}} + \alpha \cdot \mathcal{L}_{\text{classifier}}, \quad \alpha = 0.2$$

LogitsProcessor

It simply masks out scores for tokens out of a current generation mode (except for two tokens) and scales remainings.

Example: Finetuned T5 + Pretokenizer + LogitsProcessor

```
[DELIMIT_1_R]
[SEMANTIC_START] sort_ranges [SEMANTIC_END]
[DELIMIT_1_R][DELIMIT_1_R]
[SEMANTIC_START] kwargs [SEMANTIC_END]
[DELIMIT_1_R][DELIMIT_1_R][DELIMIT_1_R][DELIMIT_1_R]
[SEMANTIC_START] kwargs [SEMANTIC_END]
[DELIMIT_1_R][DELIMIT_1_R]
[SEMANTIC_START] sorted [SEMANTIC_END]
[DELIMIT_1_R]
```

```
)
sort_ranges))
kwargs))))
kwargs))
sorted)
```

Our best model + Humaneval

- **Benchmark:** HumanEval — challenging, unseen coding problems.
- **Setup:** T5 + Pretokenizer, sampling (temp = 0.1), beam size = 4, penalty = 1.3, prepared Humaneval
- **Expectation:** Likely 0 solved tasks (due to difficulty).
- **Result:** 1 task passed!
- **Implication:** Signs of semantic understanding, not just syntax mimicry.
- **Future potential:** May scale to functional correctness in more tasks.

Example: HumanEval/28 — Concatenate Strings

Task: Implement a function that concatenates a list of strings.

Expected solution: Use `" ".join(strings)`

Model's generated code:

```
def concatenate(strings):
    return " ".join([s for s in strings if s.endswith(
        " ")])
```

Commentary:

- Solution is **correct**, but **unnecessarily complex**.
- Shows understanding of:
  - Python syntax,
  - join function semantics,
  - list comprehensions.

Dataset quality

- **Meaningless docstrings**:
  - `"J'.G"`
  - `"r'\)'"`
  - `"\u5224\u65b7\u4e56\u96e2"`
- **Function diversity**:
  - Sourced from many unrelated libraries
  - Wide range of domains (e.g., data science, web, systems)
  - Inconsistent naming and formatting conventions
  - Mixed code quality

Unwanted Repetition in Generated Code

A common failure mode: excessive repetition in model outputs
Example:

```
# Prediction:
def filter_environ(self, os):
    os.environ = os.environ
    os.environ = os.environ
    os.environ = os.environ
    os.environ = os.environ
    return os.environ
```

Listing 2: Example of Finetuned T5 with Pretokenizer and Segmentator Showing Severe Repetition

Mode-Switch Tokens

- **Problem:** Special tokens like `[SEMANTIC_START]` and `[SEMANTIC_STOP]` treated like normal tokens.
- **Observation:** Only humans (not the model) knew their intended role.
- **Potential fix:**
  - Dedicated pretraining to highlight their function.
  - Auxiliary loss or tagging to enforce behavior.
- **Goal:** Help model reliably switch between control and semantic generation modes.

Curriculum Learning

- **Idea:** Start with easier tasks and gradually increase difficulty.
- **Phase 1:** Only control tokens (e.g., brackets, keywords).
- **Phase 2:** Simple semantic spans or short code snippets.
- **Phase 3:** Full function generation with mode-switching.
- **Benefit:** Allows model to build robust internal representations step by step.