Syntax-aware tokenizer for Go code style analysis

Andrii Berko[1], Vladyslav Alieksieiev[1] and Andrii Holovko[1,*]

[1]*Lviv Polytechnic National University, 12 S. Bandera str., Lviv, 79013, Ukraine*

Abstract

This study investigates how syntax-aware tokenization affects fine-tuning large language models (LLMs) for detecting Go code style violations. We introduced a syntax-aware tokenization method, deriving tokens from abstract syntax trees (ASTs) for a Llama 3.2 model, and compared its performance against the model's standard sub-word tokenization. Our experiments show that this syntax-aware approach substantially improves model performance in detecting style violations when fine-tuning is restricted to the embedding layer and classification head. However, this advantage diminishes as additional transformer layers are fine-tuned, with standard tokenization eventually outperforming the syntax-aware variant due to its better alignment with pretrained knowledge. These findings highlight an important trade-off: syntax-aware tokenization works best in scenarios requiring minimal adaptation, whereas standard tokenization provides better performance when deep fine-tuning is feasible. Future research should focus on optimizing syntax-aware methods by improving AST-to-token mapping, fine-tuning embeddings only for newly introduced tokens, and increasing their robustness for practical applications.

Keywords

Syntax-aware tokenization, code style analysis, Go programming language, multi-label classification, large language models, tokenizer design, abstract syntax trees

1. Introduction

The importance of maintaining a consistent code style across a codebase is widely recognized. This consistency is especially critical when multiple developers are working on the same project. A clear and uniform style helps developers more easily understand and review code, improves teamwork, and reduces misunderstandings. When code follows established formatting and naming conventions, identifying the program structure and logic becomes simpler, allowing developers to concentrate on functionality rather than syntax or layout. Reducing cognitive overhead not only speeds up comprehension [1], but can also minimize the likelihood of errors during development. Recent studies confirm that maintaining consistent naming, spacing, and formatting significantly enhances team collaboration and improves overall code quality [2, 3]. By adopting uniform coding standards, software teams can greatly increase the quality of their code and the efficiency of their development processes.

A consistent programming style may also serve as a behavioral fingerprint, notably improving the accuracy of plagiarism detection [4], authorship attribution [5], and even firmware analysis [6].

Many companies and open-source projects enforce specific coding standards, such as Google's style guides for Java and Python, to ensure uniformity across their codebases. The Go programming language goes a step further by including a built-in formatter, *gofmt*, which automatically formats code into a canonical style, removing debates over minor style choices. Similarly, widely adopted tools like *Prettier* for JavaScript and *Ruff* for Python automatically enforce a consistent coding style across various projects.

Code-focused large language models (LLMs), such as CodeLlama [7], StarCoder [8], and DeepSeek-Coder [9], have shown great effectiveness in software engineering tasks, including code generation and analysis. However, even in the era of LLMs, readability remains essential to software development. Atlassian, known for popular development tools like Jira, Confluence, and Bitbucket, demonstrated that LLMs could be effectively integrated into software workflows without compromising code quality [10].

MoDaST 2025: Modern Data Science Technologies Doctoral Consortium, June, 15, 2025, Lviv, Ukraine

*Corresponding author.

andrii.y.berko@lpnu.ua (A. Berko); vladyslav.i.alieksieiev@lpnu.ua (V. Alieksieiev); andrii.i.holovko@lpnu.ua (A. Holovko)

0000-0001-6756-5661 (A. Berko); 0000-0003-0712-0120 (V. Alieksieiev); 0009-0001-2380-5050 (A. Holovko)

Although the long-term influence of LLMs on software maintainability is still being explored, initial studies are encouraging. In a recent randomized controlled trial (RCT) with 151 professional developers, Borg et al. showed that AI-assisted coding tools such as GitHub Copilot enabled developers to complete programming tasks roughly 40% faster without degrading code quality or maintainability. In fact, the resulting code appeared slightly more maintainable and stylistically consistent, making it easier to hand over, extend, and support [11].

With the increasing integration of AI into software development, our research seeks to explore potential improvements in the training and adaptation pipeline for LLMs. The goal is to enable these models to assist developers not only in writing syntactically correct code but also stylistically consistent and personalized code. In earlier work, we examined the impact of dataset size on fine-tuning LLMs to classify Python code as compliant or non-compliant with the specific PEP-8 rule [12]. In the current study, we focus on another critical aspect of the LLM ecosystem: *tokenizer*.

Tokenization is an essential initial step in preparing text, including the source code, for analysis by language models. It involves breaking raw text into smaller, manageable units called tokens. While simple tokenization methods, such as splitting text by whitespace or punctuation, can be sufficient for basic tasks, they often rely on regular expressions and lack the flexibility needed for complex applications. More advanced systems, including LLMs, typically employ sub-word tokenization algorithms like Byte-Pair Encoding (BPE) [13]. BPE constructs a vocabulary by iteratively merging frequently occurring character sequences, effectively balancing vocabulary size and handling out-of-vocabulary tokens.

However, an alternative approach is to recognize that code typically follows formal rules, so capturing its syntactic structure—rather than treating it as a flat sequence of tokens—might be beneficial. Dagan et al. demonstrated that adopting domain-specific tokenizers can yield significant efficiency improvements without sacrificing performance or quality [14]. One effective method involves using abstract syntax trees (ASTs), hierarchical representations that encode the structural syntax of programming languages, thus providing a richer basis for model training or fine-tuning.

In this study, we specifically look into whether adding syntax awareness through AST-based tokenization makes it easier for a pretrained model to identify multiple style violations and whether these benefits persist when fine-tuning deeper model layers. We expand the study's scope to the Go programming language and reformulate the task as a multi-label classification problem. By utilizing ASTs to preserve the semantic and formal properties of Go code, we want to enhance the model's performance in identifying style violations.

The ultimate aim of our recent research is to better understand strategies for adapting LLMs to the task of code style analysis, contributing to the broader goal of improving software quality and reliability.

2. Related work

Research on using LLMs to support code linting and assist developers in detecting potential issues, such as memory leaks, was conducted by Holden and Kahani [15]. They trained two LLM-based classifiers on a dataset of code snippets: one classifier identified the presence of issues, while the other classified the type of issues. Their experiments demonstrated high accuracy (84.9% for issue detection and 83.6% for issue classification) and showed that the approach was significantly faster than traditional tools, highlighting the efficiency and versatility of LLMs for code linting tasks.

Han et al. [16] investigated various program representation models, which convert code snippets into numerical embeddings capturing their semantic meaning. They compared six models based on ASTs with two simpler text-based models. Evaluation across code classification, clone detection, and code search tasks revealed no single AST-based model consistently outperformed text-based models. However, specific AST-based approaches showed superior performance on particular tasks, emphasizing the importance of both textual and structural code representations.

Practical applications of AST-based methods were explored in works on AstBERT [17] and AST-T5 [18]. AstBERT integrates ASTs to effectively capture code structure and semantics. Trained on a large corpus of Java and Python code, AstBERT demonstrated strong performance in tasks like code question answering, clone detection, and refinement, underscoring the value of structural data in improving code comprehension. AST-T5 employs *structure-aware* pretraining strategies, such as AST-aware Segmentation and AST-aware Span Corruption. This model achieved improved structural coherence and integrity in code generation, significantly outperforming some larger models without complex architectural modifications or expensive analyses.

Further developments in structure-aware and grammar-augmented code generation include StructCoder [19] and SynCode [20]. StructCoder incorporates both ASTs and Data Flow Graphs (DFGs) into its encoder and introduces new auxiliary decoding tasks. This method produced a syntactically correct and semantically accurate code, achieving state-of-the-art results on various benchmarks. SynCode implements a *grammar-guided* decoding strategy that greatly reduces syntax errors in the generated code, particularly for languages with limited training data. These results highlight the benefits of grammar-aware techniques in code generation.

The value of grammar-based code representations for LLMs was explored by Liang et al. [21]. The authors examined whether incorporating grammar rules into LLMs remains beneficial, despite billion-scale models typically generating syntactically correct code. Their GrammarCoder models use grammar-based representations to significantly improve accuracy in code generation tasks. The study concluded that grammar-based representations enable LLMs to detect subtle semantic differences more effectively, confirming their continued usefulness beyond basic syntax error prevention.

3. Methodology

This section describes our methodology, covering dataset construction, tokenizer modifications, model architecture, and the experimental setup. Our primary goal is to evaluate the impact of syntax-aware tokenization on detecting style violations in Go code.

We selected the pretrained `meta-llama/Llama-3.2-1B` model as our baseline. This model was then fine-tuned on a curated dataset of Go code snippets using two distinct tokenization strategies: a *standard* approach and our proposed *syntax-aware* method.

All code for data preprocessing, model fine-tuning, and analysis is publicly available in the project's GitHub repository[1].

3.1. Dataset

Our study uses a dataset derived from the Go subset of "`The Stack v2`" [8], a comprehensive code corpus from the *BigCode Project*[2]. From over 120,000 Go code samples, we extracted 300 representative snippets for each of eight style rules defined by the *go-critic* linter's "style" group[3], resulting in a multi-label dataset. We made this dataset available on *Hugging Face* [22].

Table 1 shows eight selected style rules, their descriptions, and label indices. They generally advocate for replacing certain code constructs with alternatives that are more idiomatic or simpler. The authors of linters enable most of these rules by default and consider them non-opinionated, meaning the majority of Go developers would agree they improve code style.

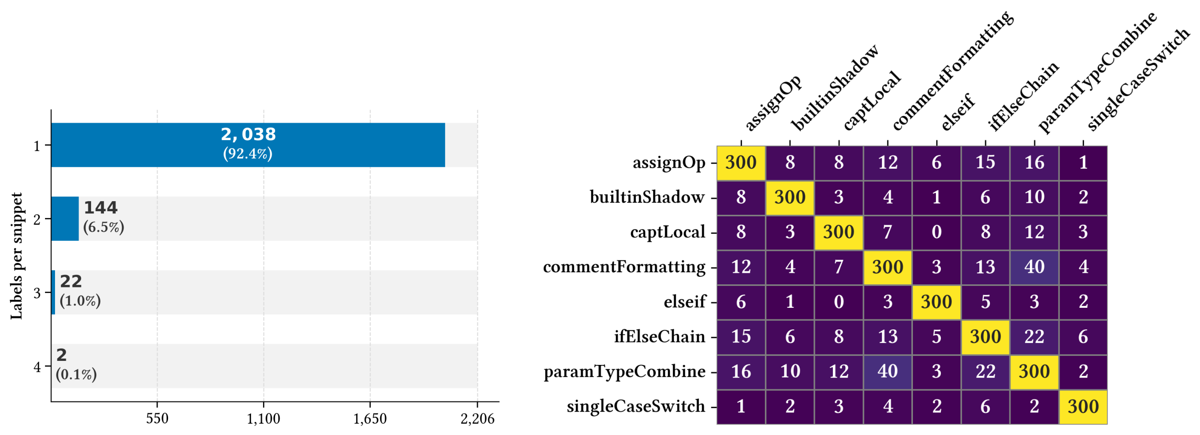
This focus on idiomatic and simple constructs aligns with broader research into code legibility. For instance, one systematic literature review identified thirteen formatting factors (including indentation, spacing, block delimiters, line length, and identifier naming conventions) whose impacts on code comprehension have been empirically studied [23]. Notably, while this review highlighted statistically significant legibility improvements for some factors, such as proper indentation, its findings for others, particularly concerning formatting layouts or identifier styles, were often divergent or inconclusive.

[1]https://github.com/aholovko/go-ast-tokenizer
[2]https://www.bigcode-project.org
[3]https://go-critic.com/overview.html#checkers-from-the-style-group

Table 1
Label definitions for *go-critic* style violations

Label	Rule	Description
0	assignOp	Simplifiable assignments via operators
1	builtinShadow	Shadowing predeclared identifiers
2	captLocal	Capitalized local variable names
3	commentFormatting	Non-idiomatic comments
4	elseif	Nested if statements replaceable by else-if
5	ifElseChain	Repeated if-else statements replaceable by switch
6	paramTypeCombine	Function parameters combinable by type
7	singleCaseSwitch	Switch statements replaceable by if

Although label sampling was uniform, most code snippets analyzed contained only one or two distinct violations, resulting in high label sparsity within our dataset. Figure 1 presents the label cardinality distribution and the co-occurrence matrix, which illustrate these inter-label relationships.



	assignOp	builtinShadow	captLocal	commentFormatting	elseif	ifElseChain	paramTypeCombine	singleCaseSwitch
assignOp	300	8	8	12	6	15	16	1
builtinShadow	8	300	3	4	1	6	10	2
captLocal	8	3	300	7	0	8	12	3
commentFormatting	12	4	7	300	3	13	40	4
elseif	6	1	0	3	300	5	3	2
ifElseChain	15	6	8	13	5	300	22	6
paramTypeCombine	16	10	12	40	3	22	300	2
singleCaseSwitch	1	2	3	4	2	6	2	300

(a) Label cardinality (b) Co-occurrence matrix

Figure 1: Label cardinality distribution and co-occurrence matrix

Label cardinality statistics confirm that the dataset has significant sparsity. Out of 2,206 code snippets, 2,038 contain exactly one style violation, 144 have two, and only 24 contain three or more (Figure 1a). This means that only 1.1% of samples contain more than two violations, leading to highly sparse multi-label target vectors where most entries are zero. Such sparsity complicates classifier learning and increases the risk of overfitting—especially when fine-tuning models with a large number of parameters. These observations motivated the use of stratified sampling and informed our selection of evaluation metrics, ensuring robustness under imbalanced label combinations.

A closer examination of label co-occurrence patterns reveals non-trivial relationships between certain style rules (Figure 1b). For instance, `paramTypeCombine` frequently appeared alongside `commentFormatting` (40 cases), `ifElseChain` (22), and `assignOp` (16). This suggests that combinable parameter declarations often co-occur with broader structural or stylistic issues. In contrast, rules like `elseif` and `singleCaseSwitch` were largely isolated, with minimal overlap, likely due to their narrower syntactic scope.

We partitioned the dataset into training, validation, and test subsets using a 70%-10%-20% split, respectively. This was achieved using the *multi-label stratified shuffle* method [24] to preserve the label distribution across all sets.

The data preparation pipeline integrated Go and Python components. A Go-based checker, leveraging the *go-critic* linter, was responsible for identifying and recording style rule violations within each code snippet. A Python wrapper orchestrated data ingestion, filtering, label assignment, stratified splitting, and the dataset's publication to the Hugging Face hub.

3.2. Syntax-aware tokenizer

The default Llama tokenizer is based on TikToken and implements Byte Pair Encoding (BPE) [25, 13]. As illustrated in Figure 2, a sample Go code snippet is tokenized into a sequence of 19 sub-word tokens, excluding the special `<begin_of_text>` marker.

```
package␣sample

func␣inc(i␣int)␣int␣{
␣␣i␣+=␣1
␣␣return␣i
}
```

```
| package | · sample | \n\n | func | · inc | (i | · int | ) |
| · int | · {\n | \ti | · += | · | 1 | \n | \treturn | · i | \n | } |
```

Figure 2: Example tokenization with the standard Llama tokenizer

To better align tokenization with Go's syntactic structure, we extended the tokenizer's vocabulary by introducing domain-specific tokens representing core language constructs. These include generic tokens for identifiers, literals, and comments, as well as specific tokens for operators, control flow keywords, and structural delimiters, as summarized in Table 2.

Table 2
Syntax-aware special tokens

Token	Description
`<IDENT>`	Identifier (variable, function)
`<LIT_INT>`, `<LIT_FLOAT>`, `<LIT_CHAR>`, `<LIT_STRING>`	Literals
`<COMMENT>`	Code comments
`<ASSIGN_OP>`, `<BINARY_OP>`	Operators
`<IF>`, `<ELSE>`, `<SWITCH>`, `<CASE>`, `<FUNC>`, etc.	Specific Go keywords
`<LBRACE>`, `<RBRACE>`, `<LPAREN>`, `<RPAREN>`, `COLON>`	Delimiters

We implemented this syntax-aware tokenization in Go, utilizing the standard `go/ast` package. Source code snippets were first parsed into abstract syntax trees, from which tokens were then extracted based on their syntactic roles. The resulting token streams were passed to the Python-based training pipeline via a C-shared library interface. Figure 3 shows an example of this tokenization applied to the sample code.

```
| <IDENT> | · sample | · | <FUNC> | · | <IDENT> | · inc | · | <IDENT> | · i | · | <IDENT> | · int | · | <IDENT> | · int | · |
| <LBRACE> | · | <IDENT> | · i | · | <ASSIGN_OP> | · | <LIT_INT> | · | 1 | · | <IDENT> | · i | · | <RBRACE> |
```

Figure 3: Sample Go code tokenized with the syntax-aware tokenizer

While our approach involves directly adding new tokens to the vocabulary and fine-tuning their embeddings, it relates to broader challenges in adapting pretrained language models to new tokenization schemes. The *Zero-Shot Tokenizer Transfer* (ZeTT) method by Minixhofer et al. [26] addresses such problems by introducing a *hypernetwork that predicts embeddings* for new tokenizers without requiring additional pretraining. Although ZeTT primarily targets natural language tasks, its core concept—decoupling the tokenizer from the pretrained model via flexible embedding projection—resonates with our objective of enhancing syntactic awareness. Our method differs by explicitly learning embeddings for new syntactic tokens during fine-tuning. However, future work could explore ZeTT-inspired strategies to potentially reduce retraining overhead or better align syntax-aware embeddings with the pretrained model's representational space.

3.3. Experiments

We evaluated the impact of *syntax-aware* versus *standard* tokenization through three fine-tuning configurations:

- **Experiment 1**: Fine-tuning the classification head only. For the syntax-aware tokenizer, token embeddings were also fine-tuned.
- **Experiment 2**: Fine-tuning the classification head and the final transformer layer (layer 15).
- **Experiment 3**: Fine-tuning the classification head and the last four transformer layers (layers 12 through 15).

Listing 1 illustrates the Llama-based architecture implemented for sequence classification, highlighting the added classification head (`score`).

```
LlamaForSequenceClassification(
  (model): LlamaModel(
    (embed_tokens): Embedding(128256, 2048)
    (layers): ModuleList(
      (0-15): 16 x LlamaDecoderLayer(
        (self_attn): LlamaAttention(
          (q_proj): Linear(in_features=2048, out_features=2048, bias=False)
          (k_proj): Linear(in_features=2048, out_features=512, bias=False)
          (v_proj): Linear(in_features=2048, out_features=512, bias=False)
          (o_proj): Linear(in_features=2048, out_features=2048, bias=False)
        )
        (mlp): LlamaMLP(
          (gate_proj): Linear(in_features=2048, out_features=8192, bias=False)
          (up_proj): Linear(in_features=2048, out_features=8192, bias=False)
          (down_proj): Linear(in_features=8192, out_features=2048, bias=False)
          (act_fn): SiLU()
        )
        (input_layernorm): LlamaRMSNorm((2048,), eps=1e-05)
        (post_attention_layernorm): LlamaRMSNorm((2048,), eps=1e-05)
      )
    )
    (norm): LlamaRMSNorm((2048,), eps=1e-05)
    (rotary_emb): LlamaRotaryEmbedding()
  )
  (score): Linear(in_features=2048, out_features=8, bias=False)
)
```

Listing 1: Llama-based architecture with a classification head for multi-label style violation detection

Model selection for each experiment was based on the highest macro-averaged F1 score achieved on the validation set during training [27]. Final performance evaluations were then conducted on the held-out test dataset.

Training and evaluation were performed on a system equipped with an NVIDIA L40S Tensor Core GPU (48 GB VRAM), 16 virtual CPUs, and 128 GB of RAM, optimized for deep learning tasks [28]. We utilized PyTorch Lightning as the training framework [29].

The loss function used was `BCEWithLogitsLoss`, which is appropriate for multi-label classification tasks. We did not apply class weights, as our data collection strategy ensured individual labels have a balanced representation across the dataset. However, the instance-level sparsity requires a careful approach to performance evaluation. In Section 3.1, we noted that most code snippets contain only one or two style violations, and none exceed four. As a result, our multi-label classification task produces very sparse output vectors, with the vast majority of labels set to zero. This sparsity can mislead standard evaluation metrics. For instance, when we measured overall accuracy, the fine-tuned model appeared to jump from a 48% baseline to 85%, suggesting excellent performance. In reality, the model had simply learned to predict all-zero labels, and the way multi-label accuracy is computed meant that this trivial "all negatives" strategy scored deceptively high. By contrast, the F1 score remained below 0.1 throughout training, revealing that the model had not learned to identify any real violations but had instead defaulted to predicting none at all. To account for this sparsity, we therefore used macro-averaged Precision, Recall, and F1 score metrics.

Each training session ran for 10 epochs with a batch size of 4 (Table 3), a configuration chosen to operate within the memory capacity of a single 48 GB NVIDIA L40S GPU without requiring gradient accumulation.

Table 3
Hyperparameters used for fine-tuning

Parameter	Value	Notes
Training epochs	10	Fixed across all experiments
Batch size	4	Empirically chosen to fit a single GPU
Learning rate	1×10^{-5}	Constant throughout training
Optimizer	AdamW	β_1=0.9, β_2=0.999, ε=10^{-8}
LR Scheduler	None	Learning rate remained fixed
Weight decay	0.01	AdamW default L2 regularization
Dropout	0.0	Llama pretraining setting

Embeddings for the newly introduced syntax-aware tokens were initialized following the default strategy of Hugging Face's `transformers` library. This strategy, based on work by Hewitt [30], involves sampling new embeddings from a multivariate normal distribution parameterized by the mean and covariance of the existing pretrained embedding matrix. Such an approach aims to align new embeddings with the established representational space, potentially improving stability and accelerating convergence. While we did not explore semantically targeted averaging for initialization (*e.g.,* combining embeddings of `if` and `else` to initialize a generic `<IF>` token), such task-specific strategies are reserved for future work.

To manage GPU memory constraints during fine-tuning, we employed several optimization techniques:

- *Mixed precision training* using "bf16-mixed" to balance computational precision with memory footprint [31];
- *Gradient checkpointing* to reduce memory by recomputing forward activations during backpropagation instead of storing them;
- *KV-cache disabling* to avoid storing large key/value tensors during the forward pass.

Finally, we deliberately decided to use full rather than parameter-efficient fine-tuning (PEFT) methods (e.g., LoRA, QLoRA). This choice ensures that all model weights, including those potentially affected by the introduction of syntax-aware tokenization, were fully updated and adapted during the training process [32].

4. Results

In this section, we present the outcomes of our experiments, comparing the performance of *standard* and *syntax-aware* tokenization across different fine-tuning configurations. All reported F1, precision, and recall scores are macro-averaged.

4.1. Experiment 1: Fine-tuning classification head

In the first experiment, where only the classification head (and embeddings for the syntax-aware tokenizer) were fine-tuned, the syntax-aware tokenizer achieved a considerably higher validation F1 score than the standard tokenizer. As shown by the training dynamics in Figures 4 and 5, the validation F1 score for the syntax-aware tokenizer reached 0.282 (precision: 0.535, recall: 0.208) by epoch 5, compared to the best value of 0.056 for the standard tokenizer in epoch 10.

The model using the standard tokenizer showed minimal reduction in training loss and did not generalize well to the validation set (validation F1 score ≤ 0.056).

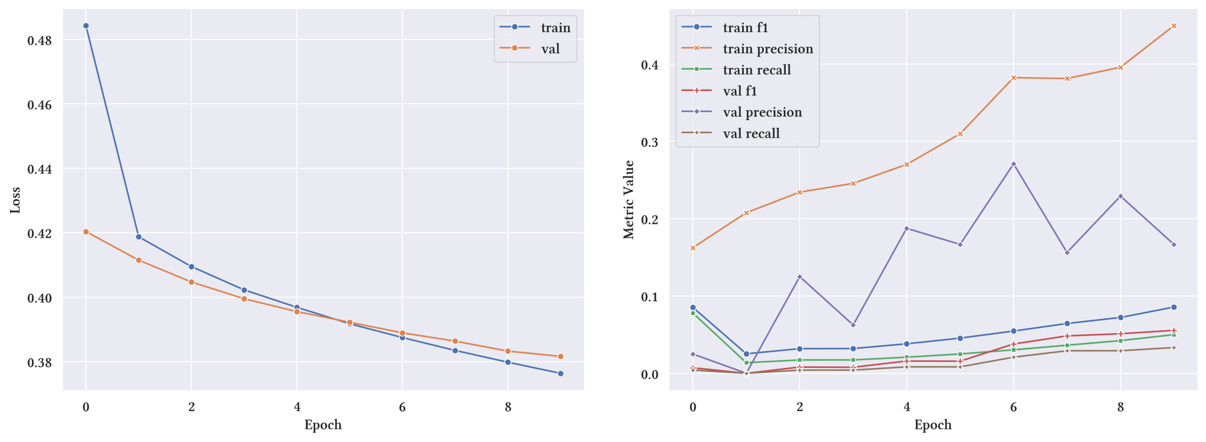


Figure 4: Fine-tuning with standard tokenizer

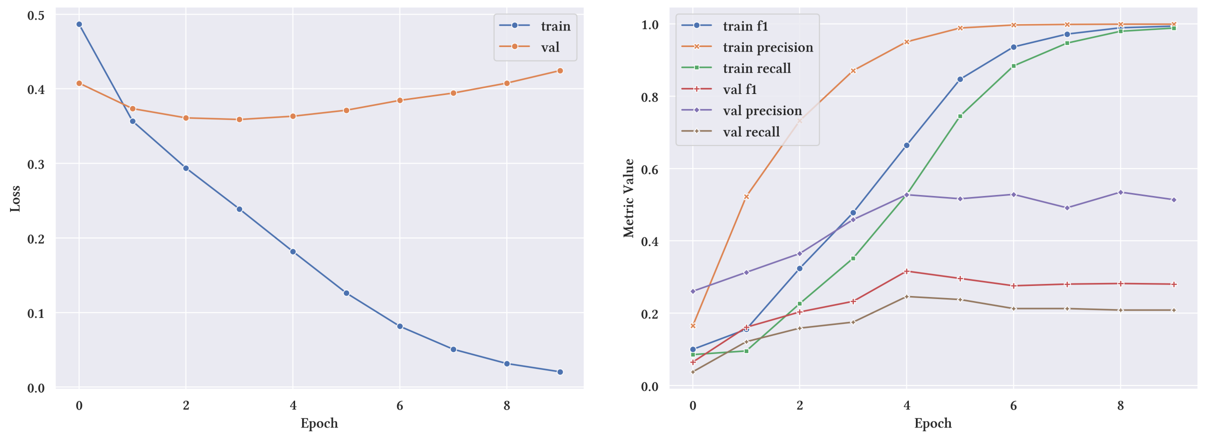
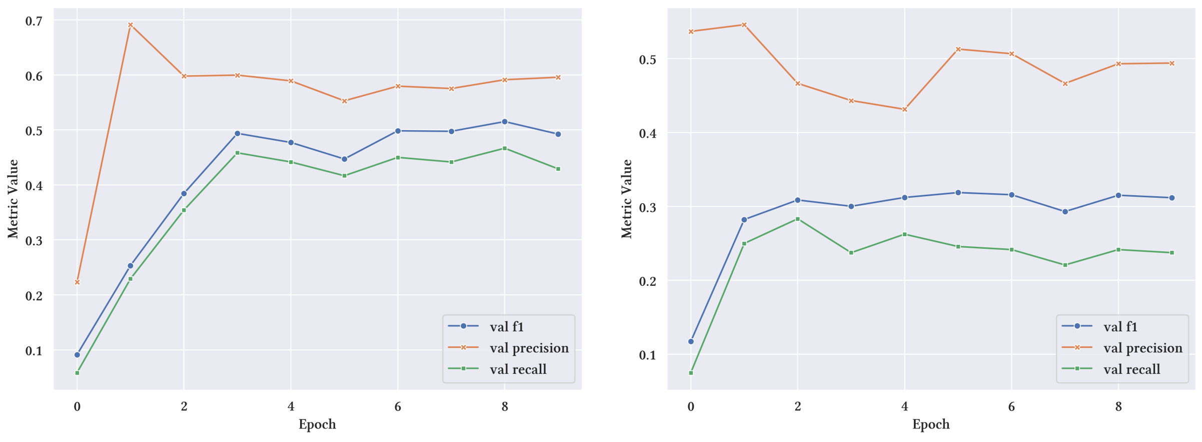


Figure 5: Fine-tuning with syntax-aware tokenizer

For the syntax-aware model, overfitting became apparent after epoch 5, with training loss decreasing towards zero while validation loss began to increase (Figure 5a). This suggests that further improvements could be achieved through early stopping, regularization techniques (*e.g.*, dropout, weight decay), or dataset augmentation. We did not apply such optimizations, as the primary objective was to compare tokenizer performance under equivalent fine-tuning setups. Nevertheless, these techniques as well as advanced loss functions tailored for multi-label scenarios [33] could be explored in future work.

4.2. Experiment 2: Fine-tuning with transformer layer 15 unfrozen

Unfreezing the final transformer layer (layer 15) alongside the classification head led to different outcomes for the two tokenizers, as shown in Figure 6. The standard tokenizer's performance improved substantially, achieving a validation F1 score of 0.515. The syntax-aware model also improved but achieved a comparatively lower validation F1 score of 0.318.

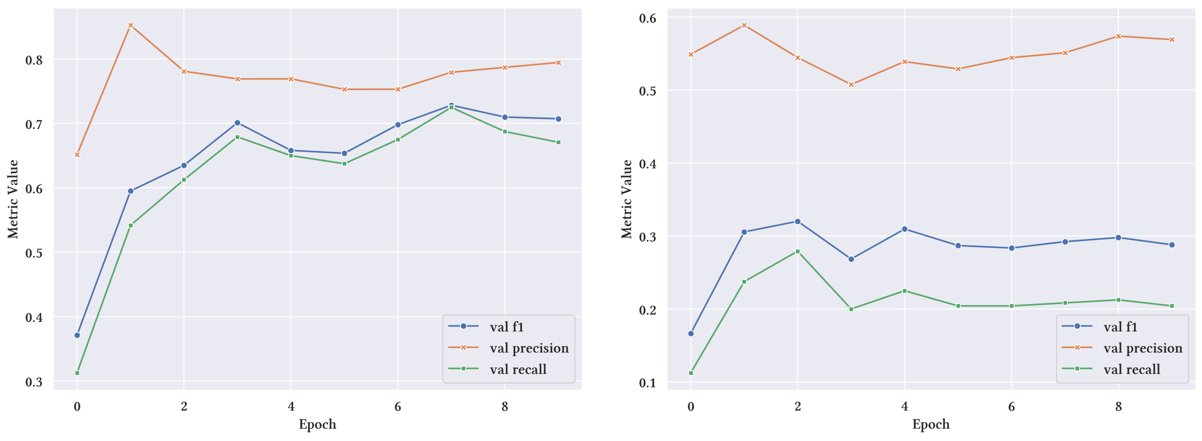


(a) standard (b) syntax-aware

Figure 6: Fine-tuning with layer 15 unfrozen

4.3. Experiment 3: Fine-tuning with layers 12–15 unfrozen

With deeper fine-tuning involving layers 12 through 15 (Figure 7), the standard tokenizer showed further improvement, reaching a validation F1 score of 0.728. The syntax-aware model's validation F1 score was 0.320 in this configuration, indicating no significant improvement over Experiment 2 for this tokenizer.



(a) standard (b) syntax-aware

Figure 7: Fine-tuning with layers 12–15 unfrozen

4.4. Evaluation and resource comparison

Table 4 summarizes the performance on the test set and resource consumption for all experimental configurations.

In Experiment 1 (classification head fine-tuning only), the syntax-aware model (B) achieved a test set F1 score of 0.290, substantially outperforming the standard model (A), which scored 0.066. This improved performance came with a higher number of trainable parameters (262M vs. 16.4K) and greater GPU memory usage (37.5 GB vs. 8.2 GB).

As more transformer layers were unfrozen for fine-tuning, the standard tokenizer's performance exceeded that of the syntax-aware variant. In Experiment 2 (layer 15 unfrozen), the standard model achieved F1 score of 0.470, compared to 0.340 for the syntax-aware model. In Experiment 3 (layers 12–15 unfrozen), this gap widened, with the standard model reaching F1 score of 0.718 versus 0.313 for

Table 4
Test set performance and resource usage (A = standard; B = syntax-aware)

Metric	Experiment 1		Experiment 2		Experiment 3	
	A	B	A	B	A	B
F1	0.066	**0.290**	**0.470**	0.340	**0.718**	0.313
Loss	0.393	0.348	0.556	0.594	0.395	0.533
Precision	0.246	0.495	0.550	0.505	0.765	0.410
Recall	0.039	0.219	0.421	0.262	0.706	0.279
Trainable Params	16.4K	262M	60.8M	323M	243M	506M
GPU Memory	8.2 GB	37.5 GB	10.3 GB	38.8 GB	19.9 GB	43.6 GB

the syntax-aware model. These trends were generally consistent across precision and recall metrics on the test set. The syntax-aware model consistently required more trainable parameters and GPU memory across all experimental setups.

5. Ablation study

We performed an ablation study to separately assess the impact of syntax-aware tokenization and embedding adaptability on model performance. All experiments used the Llama 3.2-1B architecture, with fine-tuning limited to the classification head and input embeddings. This setup allowed us to clearly evaluate the individual and combined effects of tokenization strategy and embedding training.

Table 5
Ablation study results on the validation set

ID	Tokenizer	Embeddings Trainable	F1	Precision	Recall
A1	Standard	No	0.056	0.167	0.033
A2	Standard	Yes	0.215	0.547	0.142
B1	Syntax-aware	No	0.070	0.231	0.456
B2	Syntax-aware	Yes	0.282	0.535	0.208

The main insights from this ablation study are:

- **Impact of training embeddings (A1 vs. A2)**: Making the embeddings trainable significantly boosted performance for the standard tokenizer. The F1 score increased nearly fourfold (from 0.056 to 0.215), precision more than tripled (from 0.167 to 0.547), and recall improved over fourfold (from 0.033 to 0.142). This highlights the importance of fine-tuning embeddings for effective learning, especially in sparse multi-label classification tasks.
- **Syntax-aware tokenization (A1 vs. B1)**: Introducing syntax-aware tokens while keeping embeddings frozen resulted in a moderate F1 score improvement (from 0.056 to 0.070). Recall increased substantially from 0.033 to 0.456—the highest recall value observed in this ablation study. Precision also improved, rising from 0.167 to 0.231. These results suggest that syntax-aware tokens, even without fine-tuning embeddings, enhance the model's ability to identify violations.
- **Combined strategy (A2 vs. B2)**: The combination of syntax-aware tokenization and trainable embeddings (B2) yielded the highest overall performance in this study, achieving the best F1 score (0.282). Compared to the standard tokenizer with trainable embeddings (A2), this configuration showed comparable precision (0.535 vs. 0.547 for A2) and notably improved recall (0.208 vs. 0.142 for A2). This suggests that syntax-aware tokenization and embeddings fine-tuning are complementary.
- **Impact of training syntax-aware embeddings (B1 vs. B2)**: Making the embeddings trainable for the syntax-aware tokenizer significantly improved performance. Precision increased from

0.231 to 0.535. While recall decreased (from 0.456 to 0.208), the substantial gain in precision led to a much higher F1 score (0.282 for B2 vs. 0.070 for B1). This shift indicates that training the embeddings helped the model become more discerning, reducing the tendency to over-predict violations that was apparent with frozen syntax-aware embeddings.

In summary, key takeaways from this ablation study include:

- Syntax-aware tokenization notably boosts recall due to its targeted representation;
- Embedding adaptation significantly improves precision by refining token semantics for the task;
- Integrating both approaches (B2) provides the most balanced and effective model.

These findings validate our strategic choice of combining syntax-aware tokenization with embeddings adaptation, demonstrating its effectiveness when fine-tuning for style violation detection is restricted to the classification head and input embeddings.

6. Discussion

This study aimed to investigate the impact of syntax-aware tokenization on detecting style violations in Go code, comparing it against standard sub-word tokenization across various fine-tuning depths. Our experiments revealed a nuanced relationship: the syntax-aware tokenizer notably outperformed the standard tokenizer when fine-tuning was limited to the classification head and the embedding layer (Experiment 1). In this scenario, the syntax-aware tokenizer achieved a test set F1 score of 0.290, significantly exceeding the standard tokenizer's score of 0.066. However, it is important to recognize that the syntax-aware approach fine-tuned the *entire* embedding layer, updating many more parameters than the standard tokenizer, which fine-tuned only the classification head. This larger parameter space, combined with explicitly encoded syntactic information, likely contributed to its initial strong performance.

However, this initial advantage diminished as more transformer layers were unfrozen during fine-tuning (Experiments 2 and 3). The standard tokenizer demonstrated continuous performance improvements with deeper fine-tuning, ultimately achieving an F1 score of 0.718. In contrast, the syntax-aware tokenizer's performance plateaued at approximately 0.31–0.34. This shift suggests that the standard tokenizer is more effective at leveraging the increased capacity gained from additional unfrozen layers by adapting its well-established pretrained parameters. Conversely, the syntax-aware tokenizer faced several challenges. Its extensive set of syntax-based embeddings required more training data or better optimization strategies. Additionally, the altered sequence structures resulting from syntax-aware tokenization might have limited the model's ability to utilize its pretrained knowledge effectively. The available dataset may also have been insufficient to fully train the new embeddings and adapt the model appropriately.

These contrasting outcomes present practical implications for choosing tokenization strategies in code analysis tasks, such as code linting. Syntax-aware tokenization is particularly beneficial when fine-tuning is focused primarily on the embedding and classification layers, with minimal adjustments to deeper transformer layers, as demonstrated in Experiment 1. In this context, syntactic information provides a clear and valuable signal to the model. However, when resources allow deeper fine-tuning of additional layers, standard tokenization generally offers greater performance by effectively leveraging extensive pretrained knowledge. Thus, the key challenge for syntax-aware methods in deeper fine-tuning scenarios involves efficiently training syntactic token embeddings and effectively adapting the model to the new input structures.

The current study has several limitations, which also point to opportunities for future research. Firstly, the design of our syntax-aware tokenizer—specifically the mapping from abstract syntax tree to tokens—was relatively basic and could benefit from more advanced or customized AST-based features. Secondly, our AST-based approach requires fully syntactically correct code, limiting its applicability to incomplete or erroneous code snippets, which are common in real-world scenarios. Thirdly, certain experimental choices, made to ensure a fair comparison between tokenizers, may have constrained the syntax-aware tokenizer's potential. The observed overfitting in Experiment 1 indicates that standard regularization methods, such as early stopping, dropout, or weight decay, might enhance performance if applied. Moreover, specialized loss functions designed for sparse multi-label classification, like those proposed by Zhang and Wu [33], could offer additional improvements.

Future research should therefore address these limitations and pursue further improvements. One key direction involves refining the AST-to-token mapping process to generate more effective syntax-aware tokens. Applying regularization techniques, implementing early stopping, and utilizing specialized loss functions represent essential next steps for optimizing the fine-tuning process. Integrating new syntactic embeddings with the model's pretrained knowledge is another important area. Future studies could explore more parameter-efficient strategies, such as selectively training embeddings for newly introduced syntactic tokens instead of retraining the entire embedding layer. Techniques such as embedding projection or alignment, inspired by advances like Zero-Shot Tokenizer Transfer [26], could help bridge this gap and reduce retraining efforts. Finally, improving the robustness of syntax-aware tokenization for real-world scenarios involving syntactically incorrect code is essential. This could include developing error-tolerant parsing techniques, leveraging partial ASTs, or creating hybrid approaches that dynamically combine AST-based tokens with traditional sub-word units, greatly enhancing the applicability and effectiveness of syntax-aware tokenization in diverse programming environments.

7. Conclusion

This study demonstrates that syntax-aware tokenization can substantially enhance language model performance in detecting coding style violations when fine-tuning is limited to the classification head and input embeddings. Under such conditions, syntax-enriched tokens offer valuable structural information that accelerates learning by making Go-specific syntactic patterns more explicit to the model.

However, this initial advantage diminishes as more layers of the pretrained model are unfrozen for fine-tuning. Standard tokenization, by better leveraging extensively pretrained knowledge across a larger number of adaptable layers, tends to achieve better performance in deeper fine-tuning configurations. This reveals a crucial trade-off: the immediate effectiveness of explicit syntactic information, which particularly benefits lightweight adaptation, versus the broader adaptive capacity of established pretrained models when more extensive training is available.

Therefore, the choice of tokenization for code analysis depends on available resources and desired fine-tuning depth. Syntax-aware methods are a good starting point for efficient model adaptation in specific cases. However, future work needs to improve their scalability and how they work with more broadly trained models. Key areas for development include better embedding alignment and stronger tokenization models, aiming for code language systems that are both syntax-aware and highly adaptable.

Declaration on Generative AI

During the preparation of this work, the authors used GPT-4o in order to: Paraphrase and reword, Improve writing style, Grammar and spelling check. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] P. Oliveira, R. Gheyi, J. Costa, M. Ribeiro, Assessing Python Style Guides: An Eye-Tracking Study with Novice Developers, in: Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software,

SBC, Porto Alegre, RS, Brasil, 2024, pp. 136–146. URL: https://sol.sbc.org.br/index.php/sbes/article/view/30356. doi:10.5753/sbes.2024.3325.

[2] T. Kanoutas, T. Karanikiotis, A. L. Symeonidis, Enhancing Code Readability through Automated Consistent Formatting, Electronics 13 (2024). URL: https://www.mdpi.com/2079-9292/13/11/2073. doi:10.3390/electronics13112073.

[3] W. Zou, J. Xuan, X. Xie, Z. Chen, B. Xu, How does code style inconsistency affect pull request integration? An exploratory study on 117 GitHub projects, Empirical Software Engineering 24 (2019) 3871–3903. URL: https://doi.org/10.1007/s10664-019-09720-x. doi:10.1007/s10664-019-09720-x.

[4] O. Karnalim, G. Kurniawati, Programming Style On Source Code Plagiarism And Collusion Detection, International Journal of Computing 19 (2020) 27–38. URL: https://computingonline.net/computing/article/view/1690. doi:10.47839/ijc.19.1.1690.

[5] I. Khomytska, V. Teslyuk, I. Bazylevych, I. Shylinska, Approach For Minimization Of Phoneme Groups In Authorship Attribution, International Journal of Computing 19 (2020) 55–62. URL: https://computingonline.net/computing/article/view/1693. doi:10.47839/ijc.19.1.1693.

[6] V. Yatskiv, N. Yatskiv, J. Su, A. Sachenko, Z. Hu, The Use of Modified Correction Code Based on Residue Number System in WSN, in: Proceedings of the 2013 IEEE 7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), Institute of Electrical and Electronics Engineers, Berlin, Germany, 2013, pp. 513–516. URL: http://ieeexplore.ieee.org/document/6662738/. doi:10.1109/IDAACS.2013.6662738.

[7] B. Rozière, J. Gehring, F. Gloeckle, et al., Code Llama: Open Foundation Models for Code, 2023. URL: https://arxiv.org/abs/2308.12950. doi:10.48550/ARXIV.2308.12950.

[8] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, H. de Vries, StarCoder 2 and The Stack v2: The Next Generation, 2024. URL: https://arxiv.org/abs/2402.19173. doi:10.48550/ARXIV.2402.19173.

[9] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, W. Liang, DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence, 2024. URL: http://arxiv.org/abs/2401.14196. doi:10.48550/ARXIV.2401.14196.

[10] W. Takerngsaksiri, M. Fu, C. Tantithamthavorn, J. Pasuksmit, K. Chen, M. Wu, Code Readability in the Age of Large Language Models: An Industrial Case Study from Atlassian, 2025. URL: http://arxiv.org/abs/2501.11264. doi:10.48550/arXiv.2501.11264.

[11] M. Borg, D. Hewett, D. Graham, N. Couderc, E. Söderberg, L. Church, D. Farley, Does Co-Development with AI Assistants Lead to More Maintainable Code? A Registered Report, 2024. URL: https://arxiv.org/abs/2408.10758. doi:10.48550/ARXIV.2408.10758.

[12] A. Holovko, V. Alieksieiev, Fine-Tuning Large Language Models for Code-Style Analysis: The Significance of Dataset Size, IJC (2025) 141–147. URL: https://computingonline.net/computing/article/view/3885. doi:10.47839/ijc.24.1.3885.

[13] R. Sennrich, B. Haddow, A. Birch, Neural Machine Translation of Rare Words with Subword Units, in: K. Erk, N. A. Smith (Eds.), Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Association for Computational Linguistics, Berlin, Germany, 2016, pp. 1715–1725. URL: https://aclanthology.org/P16-1162/. doi:10.18653/v1/P16-1162.

[14] G. Dagan, G. Synnaeve, B. Rozière, Getting the most out of your tokenizer for pre-training and domain adaptation, 2024. URL: https://arxiv.org/abs/2402.01035. doi:10.48550/ARXIV.2402.01035.

[15] D. Holden, N. Kahani, Code Linting using Language Models, 2024. URL: https://arxiv.org/abs/2406.19508. doi:10.48550/ARXIV.2406.19508.

[16] S. Han, D. Wang, W. Li, X. Lu, A Comparison of Code Embeddings and Beyond, 2021. URL: https://arxiv.org/abs/2109.07173. doi:10.48550/ARXIV.2109.07173.

[17] R. Liang, T. Zhang, Y. Lu, Y. Liu, Z. Huang, X. Chen, AstBERT: Enabling Language Model for Financial Code Understanding with Abstract Syntax Trees, in: Proceedings of the Fourth Workshop on Financial Technology and Natural Language Processing (FinNLP), Association for Computational Linguistics, Abu Dhabi, UAE (Hybrid), 2022, pp. 10–17. URL: https://aclanthology.org/2022.finnlp-1.2. doi:10.18653/v1/2022.finnlp-1.2.

[18] L. Gong, M. Elhoushi, A. Cheung, AST-T5: Structure-Aware Pretraining for Code Generation and Understanding, 2024. URL: https://arxiv.org/abs/2401.03003. doi:10.48550/ARXIV.2401.03003.

[19] S. Tipirneni, M. Zhu, C. K. Reddy, StructCoder: Structure-Aware Transformer for Code Generation, ACM Transactions on Knowledge Discovery from Data 18 (2024) 1–20. URL: https://dl.acm.org/doi/10.1145/3636430. doi:10.1145/3636430.

[20] S. Ugare, T. Suresh, H. Kang, S. Misailovic, G. Singh, SynCode: LLM Generation with Grammar Augmentation, 2024. URL: https://arxiv.org/abs/2403.01632. doi:10.48550/ARXIV.2403.01632.

[21] Q. Liang, Z. Zhang, Z. Sun, Z. Lin, Q. Luo, Y. Xiao, Y. Chen, Y. Zhang, H. Zhang, L. Zhang, B. Chen, Y. Xiong, Grammar-Based Code Representation: Is It a Worthy Pursuit for LLMs?, 2025. URL: https://arxiv.org/abs/2503.05507. doi:10.48550/ARXIV.2503.05507.

[22] A. Holovko, go-critic-style, 2025. URL: https://huggingface.co/datasets/aholovko/go-critic-style. doi:10.57967/HF/5304.

[23] D. Oliveira, R. Santos, F. Madeiral, H. Masuhara, F. Castor, A systematic literature review on the impact of formatting elements on code legibility, Journal of Systems and Software 203 (2023) 111728. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121223001231. doi:10.1016/j.jss.2023.111728.

[24] K. Sechidis, G. Tsoumakas, I. Vlahavas, On the Stratification of Multi-label Data, in: D. Gunopulos, T. Hofmann, D. Malerba, M. Vazirgiannis (Eds.), Machine Learning and Knowledge Discovery in Databases, volume 6913, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 145–158. URL: http://link.springer.com/10.1007/978-3-642-23808-6_10. doi:10.1007/978-3-642-23808-6_10.

[25] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan, A. Goyal, et al., The Llama 3 Herd of Models, CoRR abs/2407.21783 (2024). URL: https://doi.org/10.48550/arXiv.2407.21783. doi:10.48550/ARXIV.2407.21783.

[26] B. Minixhofer, E. M. Ponti, I. Vulić, Zero-Shot Tokenizer Transfer, 2024. URL: https://arxiv.org/abs/2405.07883. doi:10.48550/ARXIV.2405.07883.

[27] S. Raschka, Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning, 2020. URL: http://arxiv.org/abs/1811.12808. doi:10.48550/arXiv.1811.12808.

[28] NVIDIA, NVIDIA L40 GPU Accelerator, 2023. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/datasheets/L-40/product-brief-L40.pdf.

[29] W. Falcon, T. Z. L. Team, PyTorch Lightning, 2025. URL: https://zenodo.org/doi/10.5281/zenodo.3530844. doi:10.5281/ZENODO.3530844.

[30] J. Hewitt, Initializing New Word Embeddings for Pretrained Language Models, 2021. URL: https://nlp.stanford.edu/~johnhew/vocab-expansion.html.

[31] NVIDIA, Train with Mixed Precision, 2023. URL: https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html.

[32] R. Shuttleworth, J. Andreas, A. Torralba, P. Sharma, LoRA vs Full Fine-tuning: An Illusion of Equivalence, 2024. URL: http://arxiv.org/abs/2410.21228. doi:10.48550/arXiv.2410.21228.

[33] Y. Zhang, Y. Cheng, X. Huang, F. Wen, R. Feng, Y. Li, Y. Guo, Simple and Robust Loss Design for Multi-Label Learning with Missing Labels, 2021. URL: http://arxiv.org/abs/2112.07368. doi:10.48550/arXiv.2112.07368.