# Augmented Pre-Training for Code-Aware Large Language Models in Low-Resource Environments

MATT BUCHHOLZ, University of Colorado - Boulder
DAVID BAINES, University of Colorado - Boulder

**ACM Reference Format:**
Matt Buchholz and David Baines. 2023. Augmented Pre-Training for Code-Aware Large Language Models in Low-Resource Environments. 1, 1 (December 2023), 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Since the advent of the transformer [26], large language models (LLMs) have seen widespread success on a number of natural language tasks [30]. While transformers are typically trained using a sequence of language components taken from a corpus which represents the target language (components which are then tokenized for use in model pre-training), developments such as the Bidirectional Encoder Representations from Transformers (BERT) model [5] have arisen which enable context to be considered in model pre-training to augment the syntactical and sequence information provided from the transformer alone. In other words, understanding what comes before and after a masked token is more informative than just understanding what typically follows a masked token. These BERT model architectures have been shown to work well in natural language (NL) applications in biomedicine [19], public speaking analysis [10], and even COVID-19 reaction on Twitter [18], but applying these architectures to applications which attempt to "learn" a programming language (PL) have been more difficult to implement.

CuBERT [12] and CodeBERT [7] are BERT-type models developed specifically with intention of learning a PL for use in a myriad of downstream tasks such as code search [8], code clone detection [29], defect prediction [13], program translation [4], and code generation [22]. These two precedent models only use contextual elements of the code corpora to achieve these tasks, however: CuBERT uses uni-modal PL information and CodeBERT uses bi-modal NL and PL information. As with any BERT model, both the CuBERT and CodeBERT models also depend on a large amount of training data, as well as the assumption that they are correctly inferring the underlying logic structure of the PL, called an *abstract semantic tree* (AST), from these code corpora alone. TreeBERT [11] and SynCoBERT [27] then augmented the inputs from the sequential elements of the code corpora with information from the AST of the code being learned, then the AST and code comments of the PL (respectively), which provide a sense of verification that the LLM is "learning" the PL correctly. The implicit intention of this slant towards the verifiably correct properties of the language being learned is to ensure that the information gleaned from the PL in the LLM's pre-training tasks - captured by these BERT model derivatives - enable downstream tasks such as those mentioned above more reliable, performant, and accurate.

Existing PL-specific LLMs are trained on a vast amounts of files which comprise their corpora: CuBERT and TreeBERT, for example, use "7.4 million Python files containing over 9.3 billion tokens" [12, p. 3]. Due to the

Authors' addresses: Matt Buchholz, matt.buchholz@colorado.edu, University of Colorado - Boulder; David Baines, david.baines@colorado.edu, University of Colorado - Boulder.


ACM XXXX-XXXX/2023/12-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

need of this volume of training data, training data is often skewed towards PLs with widespread adoption. Wang et al. [28], for example, use source code from nine common PLs (like Java, Python, and Ruby) for training their model. Their specific training set includes over 37 million source code files, in addition to nearly 2 million training tuples of the structure `[function body, natural language comment]`.

Though existing code LLMs largely focus on languages with widespread adoption, programming languages are constantly being introduced; some new languages gain traction, while others are passed over for existing languages. Meyerovich et al. [17] study the factors that influence the adoption of programming languages. While they find that the availability of open source libraries and the need to extend existing code are the top reasons for a programmer's language choice, they also find that the availability of tools (such as integrated development environments (IDEs) and language-supporting plugins) affect language adoption. Further, Meyerovich et al. conducted their survey and analysis in an era before the popularity of LLMs in NL and PL applications.

In this context, we theorize that the development of PL-specific, pre-trained LLMs, such as those suited for downstream tasks like those mentioned above, could help drive adoption of new PLs. To achieve this, however, the pre-training tasks used in training the LLMs must chosen such that they're learning using the underlying logic structure of the PL as efficiently as possible, since not much of that PL will be available for training. A lack of large-scale available training data is a condition we refer to as a "low-resource setting." The intention of this paper then is to explore how augmented pre-training methods can bootstrap a PL-naive LLM in these low-resource settings to maximize information gleaned from the PL while minimizing the need for data to train it. We leverage properties common for all PLs, such as an unambiguous syntax and ASTs, for these augmented training tasks in attempt to achieve this result.

## 2 RELATED WORK

### 2.1 Pre-Training of Code-Aware LLMs

Based on the NL-inspired BERT model of [5], CuBERT [12] aimed to apply the BERT model to PLs. A brief background on the corpus used in this study, which is also used by us and other subsequent BERT-based models such as TreeBERT, is as follows.

The authors of CuBERT [12] created their own training corpus, comprised of using Python files from the Github repositories found on the `bigquery-public-data` project on Google's BigQuery platform. Files which are not licensed for redistribution are removed. For fine-tuning tasks (*fine-tuning* is performed after pre-training, and ensures the LLM is "tuned" to a specific downstream task objective), the authors using the ETH Py150 corpus from [21] and make subtle changes to incorporate a validation set in addition to the training and test sets provided from the work. Duplicates between the pre-training and fine-tuning sets are removed using the method discussed in [2]. All told, the pre-training corpus consists of 7.4 million Python files with 9.3 billion tokens. Tokens are then processed to capture the non-text elements found in PLs, such as indentations and new lines. Tokens are then separated in string literals, identifiers, and other PL elements, resulting in a corpus of 16 million *unique* tokens. Finally, tokens are compressed into a "subword vocabulary" [12, p.3] - a derivation of the syntax uncovered in the "program vocabulary" of the unique tokens - which contains 50 thousand unique subword tokens which are the inputs into both the embeddings as well as the BERT model pre-training tasks.

The CuBERT study used five classification tasks and one multi-headed pointer task as fine-tuning tasks for the model's "learning" of Python: variable misuse classification, wrong binary operator classification, swapped operand classification, function-docstring mismatch classification, exception type classification, and the variable-misuse localization and repair pointer task. (More detail on these tasks can be found in Section 3.4 of [12].) Comparing CuBERT to both the BiLSTM model of [9] and a naive transformer architecture, they find that CuBERT outperforms the two benchmarks on all six fine-tuning tasks. (Table 2 in [12] provides these details; this table is reproduced below for convenience.)

| | Setting | | Misuse | Operator | Operand | Docstring | Exception |
|---|---|---|---|---|---|---|---|
| **BiLSTM** (100 epochs) | From scratch | | 76.29 % | 83.65 % | 88.07 % | 76.01 % | 52.79 % |
| | CBOW | ns | 80.33 % | **86.82 %** | 89.80 % | **89.08 %** | **67.01 %** |
| | | hs | 78.00 % | 85.85 % | **90.14 %** | 87.69 % | 60.31 % |
| | Skipgram | ns | 77.06 % | 85.14 % | 89.31 % | 83.81 % | 60.07 % |
| | | hs | **80.53 %** | 86.34 % | 89.75 % | 88.80 % | 65.06 % |
| **CuBERT** | 2 epochs | | 94.04 % | 89.90 % | 92.20 % | 97.21 % | 61.04 % |
| | 10 epochs | | 95.14 % | 92.15 % | 93.62 % | 98.08 % | 77.97 % |
| | 20 epochs | | 95.21 % | 92.46 % | 93.36 % | 98.09 % | 79.12 % |
| **Transformer** | 100 epochs | | 78.28 % | 76.55 % | 87.83 % | 91.02 % | 49.56 % |

*Table 2.* Test accuracies of fine-tuned CuBERT against BiLSTM (with and without Word2Vec embeddings) and Transformer trained from scratch on the classification tasks. “ns” and “hs” respectively refer to negative sampling and hierarchical softmax settings used for training CBOW and Skipgram models. “From scratch” refers to training with freshly initialized token embeddings, without pre-training.

The preceding setup is important for our work as it demonstrates the value of choosing a corpus that fits both your pre-trained (i.e. BERT) and fine-tuning tasks (i.e. variable misuse classification). In our work, we aim to optimize the pre-training tasks we use such that we maximize knowledge gained about our unknown PL with a minimal amount of pre-training input.

## 2.2 Augmented Pre-Training for Modeling Low-Resource Natural Languages

In the domain of natural languages, Tziafas et al. [24] show that a BERT-based language model performs much better on downstream tasks when the model is exposed to explicit syntactic pre-training. The authors also find, however, that the gains from syntatic pre-training diminish as the amount of data available for training becomes larger. In other words, the more data you have for your language, the more you can rely on traditional masked token prediction as your pre-training task such as found in BERT; thus, in a higher-resource setting, one might forego augmented pre-training like explicit syntactic tasks.

## 2.3 Augmented Pre-Training for Code LLMs

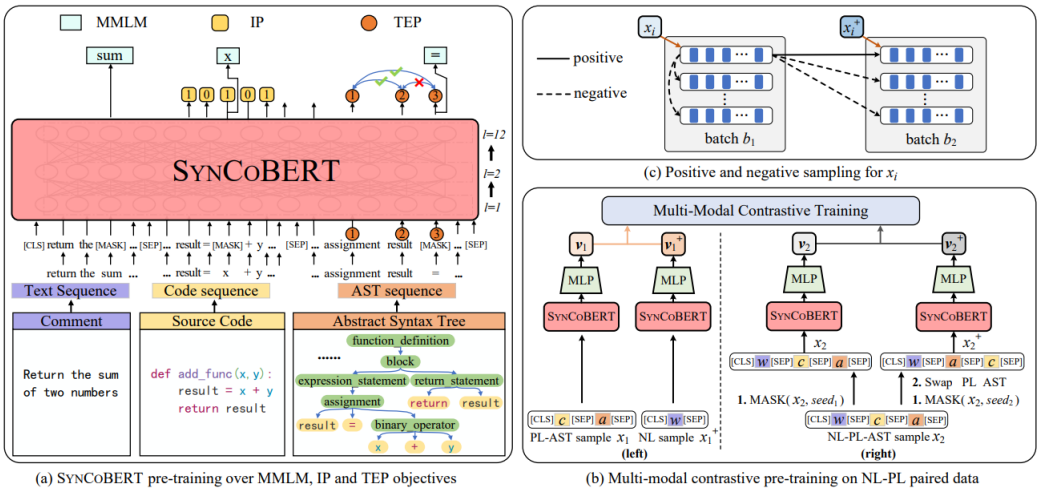
Though Tziafas et al. [24] is relatively novel for natural language LLMs, it seems that leveraging the rigid structure of programming languages is quite an active area of research for code LLMs. For example, Wang et al. [28] show how certain pre-training objectives (identifier prediction and abstract syntax tree edge prediction) improve a code LLM’s performance on downstream tasks, such as code defect detection, and translation programs from one language to another.

TreeBERT (Jiang et al, 2021) [11] introduces a syntax-oriented approach to pre-training, robust to learning the underlying logic of the language via estimating its AST. Instead of randomly masking tokens for masked language prediction, TreeBERT uses the AST of the code to weight which nodes are masked for pre-training (i.e. leaf nodes in the syntax tree are more likely to be masked; the intuition being that the leaf nodes carry more semantic content than high-level nodes like a Python `include` statement. By forcing the model to predict more semantically-informative nodes, the model ostensibly “learns” the language faster). They also mix this objective with node order prediction, a pre-training task where, given an AST, the model is asked to judge whether the AST is correct, or whether two nodes have been swapped. The authors of TreeBERT demonstrate that their model outperforms several baselines on downstream tasks, including CodeBERT (Feng et al. [7]), a code-oriented model which uses ’naive’ (not syntactically-informed) masked language modeling for pre-training.

Liu et al. (2023) introduced CodeExecutor, a model which has a pre-training regimen that includes explicit semantic information about the code [14]. In pre-training, the model is tasked with predicting the resulting program

store, given an existing program store and a line of code to execute. This is akin to an explicit training of small step operational semantics–i.e. given $< c, \sigma >$, the objective is to predict $\sigma'$; with enough examples, the model "learns" some underlying semantics of the command c.

Finally, SynCoBERT [27] is an ostensible ensemble model the combines input from code syntax (similar to CuBERT and TreeBERT), comments and NL (similar to CodeBERT), and the PL's underlying AST (similar to TreeBERT), to "learn" the underlying PL for use in downstream tasks. The authors in [27] use a multi-modal masked language model (MMLM), identifer prediction (IP), and AST edge prediction (TEP) to tune their model's learning architecture, the theory being it results in an approach that uses all available information proposed from precedent studies in its learning of the underlying PL. Below, Figure 3 from the SynCoBERT paper [27] is reproduced to show the learning architecture with these three inputs (code, comments, and ASTs) and three fine-tuning methods (MMLM, IP, and TEP) for convenience.




(a) SynCoBERT pre-training over MMLM, IP and TEP objectives

(c) Positive and negative sampling for $x_i$

(b) Multi-modal contrastive pre-training on NL-PL paired data

Figure 3: Different scenes of SynCoBERT pre-training. (a) SynCoBERT takes source code paired with comment and the corresponding AST as the input, and is pre-trained with MMLM, IP, TEP objectives. (b) Positive sampling for NL-PL paired data, (left) NL *vs* PL-AST, (right) NL-PL-AST *vs* NL-AST-PL. (c) An illustration about positive and negative pairs, including *in-batch* and *cross-batch* negative sampling.

## 2.4 Other Work In Low-Resource Code LLMs

Relatively few papers assess the feasibility of large language models applied to low-resource or domain-specific programming languages. For example, Tarassow [23] evaluates an LLMs performance on the low-resource language `hansl`; however, Tarassow relies on a human-in-the-loop methodology (an online learning technique which requires the user to give the model feedback) and a proprietary model. Tarassow doesn't, for example, fine-tune the model on hansl code (using e.g masked-token prediction).

Other work in low-resource code LLMs (e.g. Chen et al. 2022 [3], or Ahmed et al. 2022 [1]) focus on the development of multi-lingual code LLMs, and study how well these models generalize to low-resource or unseen languages, with mixed success. They do not explicitly adopt a syntax-based pre-training task, instead relying on the generalizability of the higher-resourced training languages.

## 2.5 Our Contributions

However, as far was we can tell, no research has been done to explore how well augmented pre-training scales with data available in a low-resource setting for PL-specific LLMs.

Our work differs from the above related work in that the other existing research doesn't explicitly explore the relationships between pre-training techniques and the availability of data in a low-resource setting for code LLMs.

Our research questions are therefore:

- **What types of pre-training can best augment a code LLM for a low-resource language?** In other words, what types of tasks help the model learn the underlying syntax and semantics of the language faster, given less raw data?
- **How does the amount of training data available for a language affect the relative usefulness of these augmented training methods?**

In particular, we explore the use of syntax-based pre-training tasks for these low resource settings. We choose these types of pre-training tasks because they seem most easily applicable for a low-resource programming language. In other words, even if a given language has little code available for training, such a language must have some syntactic parser for interpretation or compilation. This stands in stark contrast to natural languages, where syntactic (or dependency) parsers typically require a lot of training data, and are thus unavailable in many low-resource settings. See Vania et al. [25] for a discussion of this issue in natural languages.

While syntactic pre-training methods are likely the best suited for a low-resource setting, we also explore explicit semantic information encoded during pre-training. For this, we aim to replicate Liu et al. [15] with CodeExecutor, and whether pre-training a model on code execution traces improves downstream performance in a low-resource setting.

# 3 METHODOLOGY

We follow a similar methodology to other literature in large language models studying the effects of pre-training. We pick a few downstream tasks for evaluation which we believe are representative of 'understanding' code. We then use different pre-training methods on the same dataset to develop baseline models and our own models; we give all models the same fine-tuning on the evaluation tasks; and finally we measure their performance on those tasks. Given that the pre-training is the independent variable between the models, the model or models which perform best on the downstream tasks can be argued to have developed a better understanding of the language in pre-training.

As far as we know, no task-oriented labeled datasets exist for low-resource programming languages. Here, we again draw inspiration from Tziafas et al. [24], who simulate a low-resource setting for a natural language by training a large language model from scratch on varying amounts of a high-resource language (Dutch). In this paper, we aim to simulate a low-resource setting by training a transformer-based model on varying amounts of an high-resource language, with corresponding datasets for downstream tasks. In our study, we use Python as the language to simulate a 'low-resource' language, as there are numerous datasets which exist (see, for example, CodeXGLUE, introduced by Lu et al. [16], which introduces many different datasets with associated tasks and with different languages in the data. Python data is available for several of the tasks proposed.)

With a simulated low-resource environment, we aim to train a baseline model with various amounts of the training set. We compare this baseline to several different approaches for augmented pre-training, each applied to the same varying sizes of training data.

Our experiments aim to address a few shortcomings of the existing literature:

- **Lack of information on the relationships between available pre-training data, pre-training methods, and downstream performance.** Most existing literature on specialized pre-training for code LLMs aims, essentially, to achieve the highest downstream performance on tasks in a high-resource setting. In other words,

other research aims to develop a model which trains on as much data as possible, without exploring the relationship between amount of available training data and performance. While the authors of TreeBERT [11] do study the effect of training on various dataset sizes, none of those datasets could be seriously considered 'low resource'; the smallest of them contain hundreds of thousands of individual files, totaling millions–if not tens of millions–lines of code.

- **Addressing TreeBERT's downstream evaluation.** While the authors introducing TreeBERT demonstrate that their model performs best on some downstream tasks, downstream tasks used to evaluate the TreeBERT model (code summarization–i.e., generating function names; and code documentation–i.e., generating function header comments) could be argued to not be proper indicators of a model's understanding of the language. [1] That is, writing comments and naming functions can be seen as byproducts of the programmer's design choices, and not necessarily as products of some features inherent to the language.
- **Usefulness of downstream tasks**. Even if we concede that function naming and header-comment generation are valid measures of code 'understanding,' it's not clear those are downstream applications that are going to be most helpful for a developer in a low-resource language. If header comments and function names are going to depend largely on stylistic or design choices, a code LLM is not likely to have a lot of utility in developing applications. However, code LLM would likely be helpful in remedying issues that arise in a low-resource setting. That is, a model which can provide guidance in *writing* code could help remedy a lack of readily available tutorials, open-source projects to borrow ideas from, etc.

## 4 DATASET

To simulate a low-resource environment for Python, we choose the CodeNet dataset introduced by Puri et al. [20].

The entire dataset consists of 26,663 files, totally 1,405,073 lines of code. Of this, we reserve 20% of the dataset for validation during pre-training, and use the other 80% for pre-training. That pre-training split is also split into quarters, allowing us to train models using 25%, 50%, 75%, and 100% of the pre-train split. This allows us to study the relationship between the amount of data available, the pre-training methods chosen, and the performance of the model on downstream tasks.

We originally aimed to use the training dataset used by CuBERT (Kanade et al. 2021 [12]), and in turn, TreeBERT (Jiang et al. 2021, [12]), as TreeBERT is a model we test in this environment. However, the CodeNet dataset has several advantages for us:

- **Size.** The CodeNet dataset is already somewhat small, especially when only using the Python portion of the corpus. In contrast, the CuBERT dataset is nearly 3 Terabytes, necessitating much more downsizing for our study.
- **Accessibility.** Related to the first point above, he CodeNet dataset is much easier to download and manipulation. The CuBERT dataset, in contrast, is so large it requires the use of Google Cloud APIs and BigQuery to fetch the data.
- **Useful metadata.** For each program in the CodeNet dataset, the dataset includes metadata for the performance of the program on a dataset. This metadata includes run time, whether the program reached the correct answer, whether it timed out, how much memory it used, etc. In contrast, the CuBERT dataset is simply a scrape of open-source code from websites such as GitHub, and has no contexualizing information about the

[1] Even the metrics of evalutaing the task are suspect. Measuring BLEU scores for function header comments–that is, *n*-gram overlap with a gold-standard reference–is a bit perplexing as an evaluation metric. Even setting aside the issues with BLEU scores, in our (anecdotal) experience, different teams, projects, and individuals are going to have vastly different standards for how to do this type of documentation work. Further, much if it is extremely formulaic (for example, the copying of the parameter names into the header). So being able to replicate those types of comments is likely not a good measure of anything like code 'understanding'.

code. While we did not use this CodeNet metadata, a future study in low-resource code LLM pre-training might explore the tradeoffs in performance when using training data of varying quality.

- **Planned models to evaluate.** One of the models understudy–CodeExecutor (Liu et al. 2023 [14])–requires the use of the CodeNet dataset. Because the programs in CodeNet are run on a known dataset, Liu et al. are able to generate execution traces for the programs, which is part of their novel pre-training technique.

## 5 EXPERIMENTS

We start by replicating the pre-training experiments of TreeBERT, but with the much smaller, mono-lingual dataset mentioned above Two, the smallest datasets used to train TreeBERT are on the order of millions of lines of code. Thus, we want to evaluate its performance in a much more low-resource setting.

As a second model, we replicate the experiments of Liu et al. in CodeExecutor [14]. This model is trained first with the same methodology as the above TreeBERT replica; after that initial training, however, the model is trained to predict steps of an execution trace in the training programs.

We compare these two models to a baseline BERT model, which receives only pre-training via naive masked language modeling (i.e. the 'traditional' LLM training technique.)

Each of our models, and the baseline, is trained four times, on incremental sizes of the pre-training dataset (25%, 50%, 75%, and 100% of the training split). The entire training corpus (i.e. the 100% case) consists of 80 percent of the available Python CodeNet dataset. Each time, the model is evaluated for loss on the same-sized 'eval' set of the training data (i.e. the held out 20%), in order to make results comparable across different pre-train set sizes.

Next, each pre-trained model is fine-tuned on a downstream task, using the datasets and splits provided by the authors who suggested the tasks. Specifically, we evaluate on code completion (at the token- and line-level), and on natural-language-to-code search (given a plain English description of a program, find examples which are similar to that program from a data set). Both tasks are suggested as part of CodeXGLUE [16] and are available for code LLMs which work with Python.

In all experiments, we use the tokenizer (and underlying vocabulary) provided as part of CodeBERT [**?** ]. Inducing a vocabulary and building a tokenizer in a low-resource setting is possible, but optimizing those tools in this scenario is probably worthy of its own study. So we defer to a robust, existing tokenizer to limit potential noise from building our own tokenizer. All experiments are performed in a Google Colab notebook using a V100 GPU.

### 5.1 Pre-training

The baseline models all start with a `roberta-base model`, and are all trained with naive masked language modeling on the Python corpus, where 15% of the corpus is masked. We use the Adam optimizer and cross-entropy loss in pre-training. We train for three epochs, with a learning rate of $2 \times 10^{-5}$ and a weight decay rate of 0.01.

| Pre-Train Set Size | Epoch 1 | Epoch 2 | Epoch 3 |
|---|---|---|---|
| **25%** | 0.769 | 0.655 | 0.625 |
| **50%** | 0.651 | 0.563 | 0.527 |
| **75%** | 0.599 | 0.510 | 0.471 |
| **100%** | 0.562 | 0.480 | 0.445 |

Table 1. **Masked Language Modeling Pre-Training, Validation Loss By Epoch**. Unsurprisingly, the cross-entropy loss on the validation set decreases much faster when the model sees more data during pre-training.

For our TreeBERT replica, we also pre-train with the above settings, but use the abstract syntax tree for weighted sampling of the masked tokens. We mix this tree masked language modeling with the second TreeBERT objective, node order prediction. The loss function is defined as $\alpha \times (\text{TMLM}) + (1 - \alpha) \times (\text{NOP})$, where TMLM is the

cross-entropy loss of the masked language modeling task, and NOP is the binary cross-entropy loss of node order prediction. We use the weighting provided by the original TreeBERT authors of $\alpha = 0.75$, i.e. weight the tree masked language modeling task at three times the importance of the node order prediction task.

Given the time constraints, we weren't able to finish pre-training our TreeBERT model. We also weren't able to re-implement the proposed CodeExecutor model.

## 5.2 Code Completion

We weren't able to train models and evaluate them in the allotted time, but if we were able to, here is where we would compare the baseline model to our two models, at different sizes, and analyze the performance.

| Train Set Size | Token-level accuracy | Line-level Accuracy |
|---|---|---|
| **Baseline MLM** | | |
| **25%** | TBD | TBD |
| **50%** | TBD | TBD |
| **75%** | TBD | TBD |
| **100%** | TBD | TBD |
| **TreeBERT** | | |
| **25%** | TBD | TBD |
| **50%** | TBD | TBD |
| **75%** | TBD | TBD |
| **100%** | TBD | TBD |
| **TreeBERT + CodeExecutor** | | |
| **25%** | TBD | TBD |
| **50%** | TBD | TBD |
| **75%** | TBD | TBD |
| **100%** | TBD | TBD |

Table 2. **Code Completion Evaluation.**

## 5.3 Natural Language Code Search

Next, we fine-tune our models on natural language code search. The task is, given a natural language query, find the most relevant code from a collection of candidates. Two datasets exist for this, an adversarial set, comprised of function documentation as the query and the body of the function as the target, and a WebQueryTest set, comprised of real user queries with corresponding gold-standard programs [16].

As above, we didn't have time to pre-train, fine-tune, and evaluate our models on the second proposed task. If we had, there'd be a similar table here for natural language code search.

## 5.4 Limitations in experiment execution

Given the time constraints for a semester project, we were unable to finish executing and analyzing our experiments. Not only was the pitched project scope too broad for the amount of time available, but also we ran into several issues in replicating the experiments of the models of interesting.

For one, the TreeBERT experimental code [2] didn't work 'out of the box'. After many hours of trying to tweak the code to work, we gave up on that architecture. In hindsight, we should have re-implemented the architecture

[2]https://github.com/17385/TreeBERT

ourselves, especially given the quirks of the code base in that repository. [3] Also, for most of the code LLM pre-training code bases we investigated, their model training was tightly coupled to their dataset, which was in turn tightly coupled to their pre-processing tools[4]. For example our issues in running TreeBERT were exasperated by our use of a different dataset (CodeNet instead of CuBERT's dataset). Once more, this points to the best choice being a complete re-implementation of the architecture, allowing us to also re-implement the pre-processing tools to suit our datasets.

We also wanted to replicate the experiments of CodeExecutor [14], but in that experiment, the authors start with a model that already has explicit syntactic training ala TreeBERT. We planned to train that model starting with our re-implemented TreeBERT as a baseline, but given we were not able to get TreeBERT re-implemented, we also could not replicate CodeExecutor.

Even given the time to perform multiple iterations of pre-training and evaluation on the target datasets, much of that work would likely need to be repeated to be useful. For example, TreeBERT finds that the best balance of loss function (balancing between node order prediction and tree-based MLM) is achieved with three quarters of the weight assigned to the tree-based MLM. Such hyperparameter settings might not apply to the low-resource setting. Thus, a hyper-parameter search (retraining with different objective functions) would be needed for more conclusive results.

## 6 DISCUSSION

We've investigated how the properties of programming languages–such as an unambiguous syntax and a tight coupling of semantics and syntax–can potentially be leveraged in pre-training to get a code LLM to perform better on downstream tasks, even in a low resource setting.

While natural language LMs can also benefit from these types of pre-training, natural language likely can't leverage these types of properties to the same extent. For example, natural languages can have ambiguous syntax (e.g. 'time flies like an arrow'), and natural language utterances are often syntactically malformed. Further, a programming language has to have a syntactic parser to be useful (either as an interpreter or a compiler), meaning syntactic pre-training task data is virtually free. In contrast, for natural languages, a syntactic (or dependency) parser is often one of the first tools that has to be created in order to work in a low-resource setting, as the existence of one is not a given.

We can also contrast the coupling of syntax and semantics in programming languages with the looser coupling of those features in natural languages. In natural languages, arguments to a syntactic frame can be omitted in different domains or languages. For example, in a recipe we might write:

(1) Mix flour, yeast, and water together.
(2) Cover the dough and let rise for two hours.
(3) Bake at 350 degrees for forty minutes.
(4) Slice bread and enjoy with butter.

Before the first step, the dough does not exist; it's created as part of the mixing. Starting with the second step, the flour, yeast, and water no longer exist as individual entities. In turn, the dough ceases to be dough and becomes bread after the third step. None of these facts are conveyed through explicitation. (We, as readers, know these to be true because of the semantic frames evoked by this cooking context.)

[3] As an example, multiple functions in that repository emulate tail recursion by abusing exceptions. A comment helpfully explains a function does this by 'throwing an exception if it is its own grandparent.' In general, however, our issues in working with the code base were the usual issues in working with LLM code–poorly documented dependencies and assumptions about the run environment, the data being used, etc.
[4] Naturally these issues compound quickly when multiple pre-processing steps are needed before beginning to run the training code on the data.

As another example, some natural languages allow the omission of arguments in certain dialogue contexts. In Italian, for example, a pronoun can be explicitly stated, or it can be omitted: *Parlo italiano* and ***Io*** *parlo italiano* both mean 'I speak Italian', but only the second example makes the pronoun 'I' explicit. [5].

These natural language examples contrast with many programming languages, where the syntax and semantics are often more explicit. In the cooking example, a code parallel might be a function call `mix(flour,yeast,water)`, which *implicitly* instantiates `bread`. In the example of Italian dropping pronouns, we might view `parlo` 'speak' as a function with two arguments–a speaker and a language–but only one is passed explicitly here.

A real-world parallel in these cases could be InterSystems Cache, which has looser scoping rules and allows a programmer to infer variables across any number of stack frames, as long as that variable is not explicitly re-scoped by as a parameter or with the `new` command.[6] So, in InterSystems Cache, it's possible to write subroutines like `mix(flour,yeast,water)` which implicitly change the global state (i.e. instantiate some `bread`), or `parlo(italiano)`, which infers the speaker argument. Given just the surface forms of the code, it may be hard for a code LLM to learn these types of patterns with limited data.

In any case, it's clear that the tightness of the coupling between syntax and semantics varies across languages, and is certainly different when comparing natural languages to programming languages. This suggests that in pre-training a large language model–where learning the underlying syntax and semantics is the goal–different methods might be needed across different languages (both programming and natural).

## 6.1 Potential Future Work

In the future (once we're able to execute the experiments suggested above), we see many additional research directions.

## 6.2 CodeNet in the low-resource setting.

CodeNet [20] is not just a collection of programs–it's a collection of programs with common goals and *known datasets* and metrics on the performance of those programs (run time, whether the right answer was achieved, memory used in execution, etc.) CodeExecutor [14] leveraged the executability of those programs (to generate traces), but didn't leverage the other metadata. Investigating whether pruning the a pre-training dataset ('only use good programs') improves model performance could be an interesting direction.

## 6.3 Other syntactic or semantic representations of code.

We explore here two ways in which pre-training can be augmented with explicit syntactic or semantic information beyond what's encoded in the concrete syntax of a language. However, abstract syntax trees are not the only way to represent a syntax, and execution traces are not the only way to represent the semantics of a program. Given this, exploring how other representations–especially those that might be readily available in a low-resource setting–might help a code LLM converge faster is an interesting exploration.

## 6.4 Hyperparameter search for weighting pre-training objectives in low-resource settings.

Mixed pre-training objectives require a mixed loss fuction, which in turn means choosing weights (i.e. relative importance) of those pre-training tasks.[7] For example, Tziafas et al. [24] find that a syntax-directed pre-training regimen helps greatly when the training data is limited, but the relative effectiveness falls off when the model is

[5] Source: personal knowledge.

[6] Source: personal knowledge. More info at the official documentation site: https://docs.intersystems.com/latest/csp/docbook/DocBook.UI.Page.cls?KEY=RCOS

[7] In an alternative formulation, the tasks aren't mixed, but instead pipelined; i.e. do one pre-training task, then do the next, and so on. This is a different hyperparameter search–how to order your pre-training tasks.

exposed to more and more data. Thus, we believe the relative importance of different pre-training tasks depends on what types of information the model is able to learn directly through the data during masked language modeling, given the amount of data available.

### 6.5 Multilingualism in Low Resource Settings

Another potential avenue for increasing performance in low-resource settings could be using models which are trained primarily on higher-resource languages. In developing TreeBERT, [11] the authors found incorporating syntactic information in training helped downstream performance on a programming language which was not included in the pre-training corpus (C). This suggests their syntactic pre-training might have helped encode knowledge which was abstracted away from any particular language.

In the field of natural languages, transfer learning and multilingual models are frequently employed to boost performance in a low-resource setting. A promising avenue of this research is taking advantage of linguistic typology–i.e. understanding how different languages might share features–to pick high-resource languages best suited for the model's pre-training. As an example, [6] found that using adapters trained on one language performed best on other languages which are closely related. A parallel could be made to programming languages, but to our knowledge, no 'typology'-based study of this kind has been carried out for LMs for programming languages. [8]

## REFERENCES


[1] Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1443–1455, New York, NY, USA. Association for Computing Machinery.

[2] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code.

[3] Fuxiang Chen, Fatemeh Fard, David Lo, and Timofey Bryksin. 2022. On the transferability of pre-trained language models for low-resource programming languages.

[4] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.

[6] Fahim Faisal and Antonios Anastasopoulos. 2022. Phylogeny-inspired adaptation of multilingual models to new languages.

[7] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages.

[8] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 933–944, New York, NY, USA. Association for Computing Machinery.

[9] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780.

[10] Nanjiang Jiang and Marie-Catherine de Marneffe. 2019. Evaluating BERT for natural language inference: A case study on the CommitmentBank. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 6086–6091, Hong Kong, China. Association for Computational Linguistics.

[11] Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. 2021. Treebert: A tree-based pre-trained model for programming language. *CoRR*, abs/2105.12485.

[12] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Pre-trained contextual embedding of source code. *CoRR*, abs/2001.00059.


[8] In natural languages, typological comparisons can be made in a few ways. Naturally, phylogenetically-related languages are an easy place to find languages with similar features. For example, French shares many features with Latin because it is descended from Latin; a programming-language parallel could be in comparing to Typescript to Javascript, as Typescript can be seen as a descendant of Javascript (and it transpiles to JS). A less-studied, but perhaps interesting approach, could be in comparing (programming) languages which are not directly *related*, but which still share typological features. In natural languages, an example might be comparing Arapaho and Turkish; these languages are unrelated, but share some similarities in their tendency to agglutinate. One could even view the axes upon which natural languages are categorized–such morphosyntactic alignment, or basic constituent order–as a 'formalization' of those languages, akin to the formalisms of the statics and dynamics of a programming language. Thus we might leverage those formalizations to discover languages similar to a low-resource language, and pick languages a model could best pre-train on for transfer to that low-resource language. We leave this idea for future work, however.

[13] Jian Li, Pinjia He, Jieming Zhu, and Michael R. Lyu. 2017. Software defect prediction via convolutional neural network. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 318–328.
[14] Chenxiao Liu, Shuai Lu, Weizhu Chen, Daxin Jiang, Alexey Svyatkovskiy, Shengyu Fu, Neel Sundaresan, and Nan Duan. 2023. Code execution with pre-trained language models.
[15] Chenxiao Liu, Shuai Lu, Weizhu Chen, Daxin Jiang, Alexey Svyatkovskiy, Shengyu Fu, Neel Sundaresan, and Nan Duan. 2023. Code execution with pre-trained language models.
[16] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664.
[17] Leo A. Meyerovich and Ariel S. Rabkin. 2013. Empirical analysis of programming language adoption. *SIGPLAN Not.*, 48(10):1–18.
[18] Martin Müller, Marcel Salathé, and Per E. Kummervold. 2023. Covid-twitter-bert: A natural language processing model to analyse covid-19 content on twitter. *Frontiers in Artificial Intelligence*, 6.
[19] Yifan Peng, Shankai Yan, and Zhiyong Lu. 2019. Transfer learning in biomedical natural language processing: An evaluation of bert and elmo on ten benchmarking datasets.
[20] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, and Ulrich Finkler. 2021. Project codenet: A large-scale AI for code dataset for learning a diversity of coding tasks. *CoRR*, abs/2105.12655.
[21] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic model for code with decision trees. *SIGPLAN Not.*, 51(10):731–747.
[22] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. *CoRR*, abs/2005.08025.
[23] Artur Tarassow. 2023. The potential of llms for coding with low-resource and domain-specific programming languages.
[24] Giorgos Tziafas, Konstantinos Kogkalidis, Gijs Wijnholds, and Michael Moortgat. 2021. Improving BERT pretraining with syntactic supervision. *CoRR*, abs/2104.10516.
[25] Clara Vania, Yova Kementchedjhieva, Anders Søgaard, and Adam Lopez. 2019. A systematic comparison of methods for low-resource dependency parsing on genuinely low-resource languages. *CoRR*, abs/1909.02857.
[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention is all you need.
[27] Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation.
[28] Xin Wang, Yasheng Wang, Pingyi Zhou, Fei Mi, Meng Xiao, Yadao Wang, Li Li, Xiao Liu, Hao Wu, Jin Liu, and Xin Jiang. 2021. CLSEBERT: contrastive learning for syntax enhanced code pre-trained model. *CoRR*, abs/2108.04556.
[29] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98.
[30] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. 2023. A survey of large language models.