Improving Text-to-Code Generation with Curriculum Learning

Candidate Number: XLYR4[1]

MSc Data Science and Machine Learning

Supervisors: Efstathia Christopoulou, Ignacio Iacobacci, Mark Herbster

September 2023

[1]**Disclaimer:** This report is submitted as part requirement for the MSc Data Science and Machine Learning at University College London. It is substantially the result of my own work except where explicitly indicated in the text. The report will be distributed to the internal and external examiners, but thereafter may not be copied or distributed except with permission from the author.

Abstract

This thesis investigates the effectiveness of Curriculum Learning (CL) strategies to enhance text-to-code generation. By progressively introducing complex programming concepts during training, we evaluate 13 diverse CL strategies based on various criteria. Our findings demonstrate that CL consistently improves text-to-code generation, with the best strategy achieving an impressive 4.71% improvement in the CodeBLEU metric. Moreover, our approach showcases a remarkable 22.86% reduction in the number of non-executable samples compared to the baseline model. This research contributes to more efficient and accurate automated code synthesis, leveraging CL to optimize text-to-code generation systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Text-to-code generation involves the process of designing and constructing an executable computer program that meets a specific problem description. This area of research has drawn considerable attention due to its profound implications in the software industry, leading to improved productivity and increased accessibility in programming-related careers and educational endeavors. As software systems become more complex, there's a growing need for efficient methods to generate code from human-readable descriptions. This is valuable for enhancing developer productivity and enabling collaboration between programmers and domain experts.

Recent advances in natural language processing and large language model have led to the development of coding-assistant models like Github Copliot [Chen et al., 2021], CodeGen [Nijkamp et al., 2023], StartCoder [Li et al., 2023], and StarChat [Tunstall et al., 2023]. These models represent the evolving landscape of text-to-code generation and reflect the growing interest in intuitive code synthesis interfaces. Their exploration presents exciting opportunities for further innovation in text-to-code generation.

Despite the tremendous potential of this area, code generation tasks still challenges. Code generation is inherently different compared with natural language generation as it necessitates adherence to strict syntax and semantic rules. Furthermore, code generation is bounded by the structural constraints of the code itself, requiring that the generated code remains valid and functional during compilation and execution.

To address these challenges and improve the text-to-code generation process, this thesis employs the curriculum learning method [Bengio et al., 2009], which mimics human learn-

ing by reordering training examples. We propose to organize samples with various different difficulty metrics ranging from simple length to introducing gradually more complex programming concepts to the model, during training. This methodology enables the model to begin with simple code segments and gradually advance to more intricate and comprehensive code programs, ultimately enhancing the quality and precision of the generated codes.

The main objective of this thesis is to explore the application of different curriculum learning strategies in code generation and compare their effectiveness. Through experimental comparisons of multiple curriculum learning methods, we evaluate the strengths and weaknesses of different strategies in improving code generation capabilities and assess their applicability.

1.2 Thesis Contributions

In this research, we make significant contributions in text-to-code generation through an extensive exploration of Curriculum Learning (CL) strategies, assessing their effectiveness from four distinct perspectives: ordering data based on properties of the natural language description, based on the properties of the code, based on the capabilities of an external model to generate code (given a natural language description) and finally a combination of the above. A total of 13 diverse CL models are examined to evaluate their impact on text-to-code generation performance.

For our investigation, we consider a limited resources setting, where models can only be trained for a short amount of time. Throughout the thesis, all CL strategies outperform the baseline model in terms of the CodeBLEU metric, with the best strategy achieving an impressive improvement of 4.71%. Furthermore, to gain comprehensive insights, we select several well-performing models and allow them to be trained further. Subsequently, we evaluate the quality of their generated codes. Impressively, the best model exhibits a significant 22.86% reduction in the number of non-executable samples compared to the baseline model.

Our findings provide compelling evidence that curriculum learning is a powerful technique for enhancing the quality and accuracy of text-to-code generation. By thoughtfully designing the training process to progressively introduce more challenging tasks, we facilitate effective learning and improved generalization, culminating in higher-quality code synthesis.

Overall, this research contributes to the advancement of curriculum learning strategies in code generation tasks, emphasizing their potential to enhance the quality of code synthesis. It establishes a strong foundation for further investigations in training Code Language Models with curriculum learning, promoting the development of more efficient and intelligent automated code generation systems.

Chapter 2

Background

2.1 Large Language Models

With the emergence of ChatGPT [OpenAI, 2023], the awareness and familiarity of Large Language Models (LLMs) have grown steadily. This technological advancement has brought LLMs into the spotlight, capturing the attention and interest of a broader audience. The capabilities demonstrated by ChatGPT, such as its natural language understanding, generation, and interaction, have contributed to a deeper understanding of the potential applications and implications of LLMs. As a result, these models have become a subject of discussion and exploration in various domains, including academia, industry, and popular culture. The advent of ChatGPT marks a significant milestone in the ongoing evolution of LLMs, propelling them into the forefront of technological innovation and public discourse.

In this section, we will introduce a selection of well-known models: Transformer [Vaswani et al., 2017], BERT [Devlin et al., 2019] and GPT [Radford and Narasimhan, 2018].

Transformer, a revolutionary architecture in the field of natural language processing, was first introduced by Vaswani et al. [2017] in 2017. One of the fundamental structures of Transformer is the encoder-decoder framework. The encoder is responsible for encoding the input sequence into a set of representations that capture the semantic and syntactic information of the text. The decoder, on the other hand, utilizes these representations to generate the output sequence one token at a time. The important self-attention mechanism, embedded in both encoder and decoder, allows the model to consider the importance of different words and their dependencies within a sequence. The multi-head attention mechanism facilitates the consideration of different aspects of context by performing the

attention function in parallel.

$$\text{MultiHead}\,(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$

$$\textbf{where}\ \text{head}_\text{i} = \text{softmax}\left(\frac{(QW_i^Q)(KW_i^K)^T}{\sqrt{d_{model}}}\right)(VW_i^V)$$

where Q is the query, K and V is the key-value pairs. (Shown in Figure 2.1) The position-wise feed-forward neural networks $\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$ is applied to capture the nonlinear relationships. Moreover, the positional encodings ensure the model can differentiate between token positions within a sequence. The comprehensive illustration of the Transformer's structure is presented in Figure 2.1.

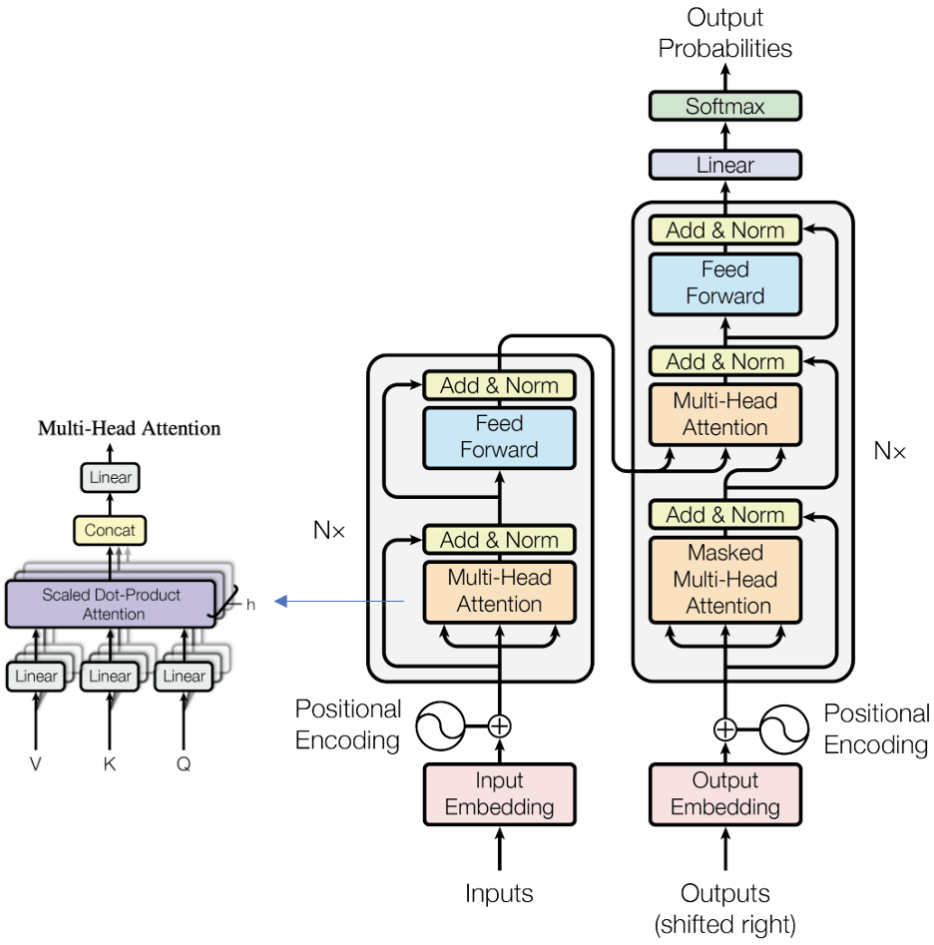

Figure 2.1: (**left**) The details of Multi-Head Attention. (**right**) The architecture of Transformer. Provided by Vaswani et al. [2017].

BERT (Bidirectional Encoder Representations from Transformers) [Devlin et al., 2019], based on the encoder structure of the Transformer model, employs a Masked Language Model (MLM) technique to conduct pretraining on extensive unlabeled text data. This approach involves conditioning the model on both left and right context within all layers, thereby facilitating the creation of deep bidirectional representations. In this task, the MLM randomly conceals some input tokens, aiming to predict the original masked word by considering its context. In addition, Jacob introduces a "next sentence prediction" task, enhancing sentence relationship understanding by jointly pretraining text-pair representations. Figure 2.2 shows the pretraining procedure of BERT model. These innovations enable BERT's remarkable performance across diverse natural language understanding tasks.

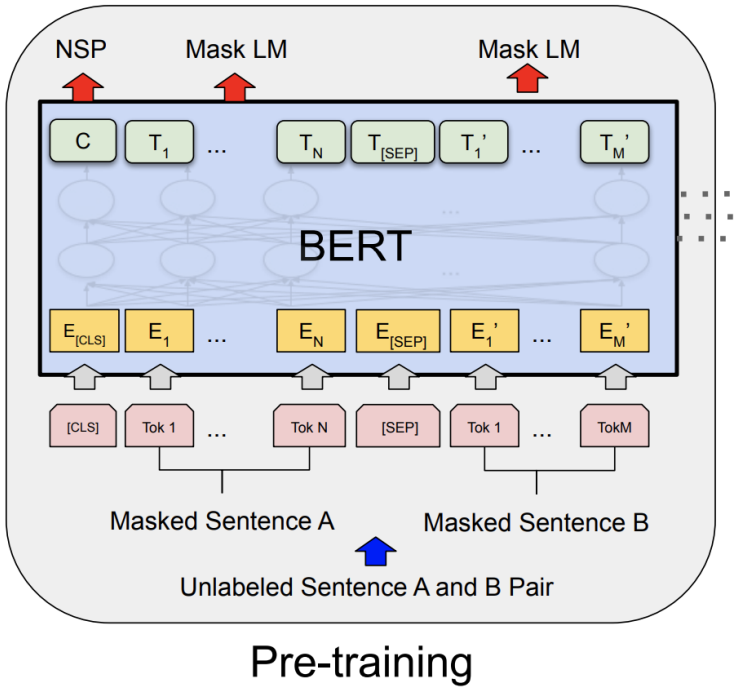

Figure 2.2: The pretraining architecture of BERT model, provided by Devlin et al. [2019].

GPT (Generative Pre-trained Transformer) [Radford and Narasimhan, 2018] selected the core decoder structure of the Transformer architecture and implemented an auto-regressive language modeling methodology in its initial training, using a sizable unlabeled textual corpus to optimize the likelihood:

$$L(u_1, ..., u_n) = \sum_i \log P(u_i | u_{i-k}, ..., u_{i-1}; \Theta)$$

where k is the size of the context window, and P is the conditional probability of the token u_i given the previous tokens in the context window by the model with parameters Θ [Radford and Narasimhan, 2018]. This approach involves iteratively generating successive terms to simulate the process of text generation. Concurrently, the model enhances its memory structure to manage extended contextual dependencies within the text, thus GPT has the capacity to generate coherent and fluent text.

2.2 Code Language Models

Code Language Models (CLMs) have shown remarkable potential in addressing the intricate relationship between Natural Language (NL) and Programming Languages (PL). Leveraging the capabilities of large-scale pretraining and subsequent fine-tuning, CLMs have found utility in a diverse range of code-related tasks, including code completion, retrieval, summarization, translation, and generation. These models exhibit a profound understanding of programming language syntax, semantics, and even coding conventions. Such advancements in the field have led to the development of sophisticated coding assistants, such as CodeX [Chen et al., 2021], StartCoder [Li et al., 2023], and StarChat [Tunstall et al., 2023], aimed at enhancing developer productivity and expanding access to coding expertise. The implications of these developments extend across various industries, from automating routine coding activities to aiding in intricate software development researchers.

Within this section, we will introduce a comprehensive investigation into multiple CLMs, which classified into three primary categories: encoder-only models, decoder-only models, and encoder-decoder models. Moreover, this research is attentive to the limitations posed by finite resources. To enhance the comprehensibility of these considerations, Figure 2.3 presents a visual exposition, depicting the number of parameters across the different CLMs.

2.2.1 Encoder-only models

Encoder-only models are a class of neural network architectures comprising solely an encoder component, designed to process input data and extract meaningful representations from the input text. Due to their inherent characteristics, these models have demonstrated superior performance in code search and code understanding tasks such as code retrieval.

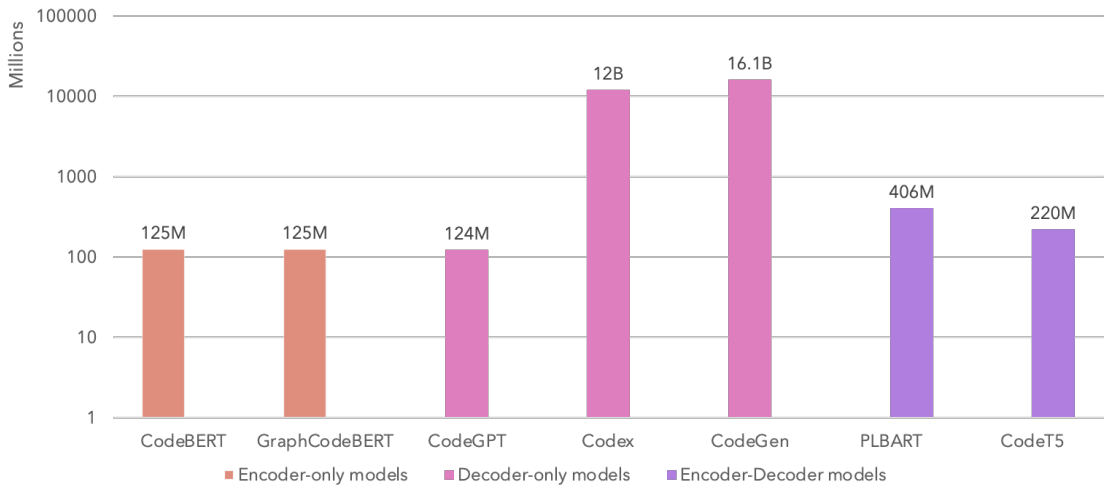

Figure 2.3: A visual representation of the sizes of different CLMs.

CodeBERT [Feng et al., 2020] represents a pioneering effort in the field of code language models. This model leverages the effective masked language modeling approach of BERT [Devlin et al., 2019] to derive comprehensive code-specific representations, which achieve high-quality contextual embeddings for source code. Feng et al. [2020] trained their model using the replaced token detection task, where the objective is to identify credible alternatives selected from generators, to learn the NL-PL representation.

GraphCodeBERT, proposed by Guo et al. [2021], extends the CodeBERT model [Feng et al., 2020] by incorporating data flow information extracted from the inherent structure of code. Data flow represents the relationship between variables, indicating 'where-the-value-comes-from', which enhances the model's efficiency. In addition to employing the masked language modeling task, Guo et al. [2021] introduced two structure-aware pretraining tasks. The first task involves predicting code structure edges, while the second task aligns representations between the source code and code structure. To efficiently incorporate the code structure, Guo et al. [2021] also implement the model with a graph-guided masked attention function, which enables the model to effectively leverage the data flow information and enhance the understanding of code semantics during pretraining.

2.2.2 Decoder-only models

Decoder-only models are only uses the decoder component of the Transformer architecture, which generate an output sequence based solely on some initial input, context vectors, or tokens, without taking any additional input sequence into account. Due to their intrinsic

characteristics, these models have showcased superior proficiency in code translation and program synthesis tasks such as code completion. Moreover, as illustrated in Figure 2.3, it becomes evident that the efficient decoder-only models, Codex [Chen et al., 2021] and CodeGen [Nijkamp et al., 2023], exhibit significantly larger sizes in comparison to both encoder-only and encoder-decoder models. This implies that the process of training and fine-tuning a substantial decoder-only model demands substantial computational resources.

CodeGPT [Lu et al., 2021] is a decoder-only model, following the auto-regressive, left-to-right language modeling paradigm, where it predicts the likelihood of a token based on the preceding tokens. Its left-to-right nature makes it particularly suitable for tasks involving program generation, such as code completion. However, due to the fact that code is typically not written in a single left-to-right pass, effectively incorporating context that appears "after" the generation location presents a non-trivial challenge. (this paragraph can be removed if it is not suitable)

Codex provided by Chen et al. [2021] is a GPT-based model which fine-tuned on the open-source code from Github. To mitigate the impact of unrelated code segments (in addition to the standalone functions) in GitHub repositories, which self-contained training samples with well-crafted problem statements are curated from competitive programming platforms for supervised fine-tuning. In addition, by tracing inputs and outputs for functions to collect the data and generate unit tests for training, expanding task diversity. To control for quality, Codex-12B [Chen et al., 2021] is used to generate numerous problem-specific samples and filter the training problem if there are no samples pass the unit tests. The whole loop improves the Codex's proficiency in code function generation across multiple programming languages, thereby facilitating rapid prototyping and enhancing development efficiency. Codex still remains competitive in program synthesis nowadays.

CodeGen [Nijkamp et al., 2023] takes the form of autoregressive transformers, utilizing a language modeling approach with next-token prediction as the central learning objective. These models are trained on a diverse dataset derived from both natural language and programming language sources from GitHub repositories. Training proceeds in a sequential manner across distinct datasets, including THEPILE, which predominantly comprises English text; BIGQUERY, encompassing six distinct programming languages; and BIGPYTHON, housing a substantial volume of data in the Python programming language.

Nijkamp et al. [2023] has introduced the concept of prompt perplexity as a proxy for

assessing the model's comprehension of user intent. This metric is computed as

$$\text{PPL} = \exp\left(-\frac{1}{m}\sum_{i=1}^{n}\text{logProb}_i\right)$$

where m is the total number of tokens of all prompts $\{p_i\}_{i=1}^{n}$, and Prob_i is the conditional probability of the prompt p_i given the concatenation $\{p_1, s_1, ..., p_{i-1}, s_{i-1}\}$ by the model, s_i is the sub-programs generated by the model with corresponding p_i [Nijkamp et al., 2023].

The combination of a higher prompt perplexity value and pass rate in program synthesis for multi-turn specifications, as compared to single-turn specifications, underscores the model's enhanced capability in understanding a user's complex requirement into multiple sequential steps. This improved understanding facilitates more effective program synthesis by the model. The aforementioned findings highlight the model's potential to grasp intricate user instructions and subsequently generate programs with heightened proficiency.

2.2.3 Encoder-Decoder models

An encoder-decoder model initially employs an encoder to encode an input sequence, followed by the utilization of a left-to-right language model (i.e. a decoder) to decode an output sequence based on the encoded input sequence. These pretrained models prove valuable in various sequence-to-sequence tasks and demonstrate excellent performance in code understanding like code defect detection and clone detection. They also perform well in conditional generation, such as code translation and text-to-code generation.

PLBART [Ahmad et al., 2021] is introduced as an extension of the encoder-decoder model BART [Lewis et al., 2019], possessing the capability to address a wide range of program and language understanding and generation tasks. Ahmad et al. [2021] adopts the denoising sequence reconstruction, as proposed by Lewis et al. [2019], where the model learns to reconstruct the original sequence from a corrupted input sequence, including three noise settings: token masking, token deletion and token infilling. By leveraging this technique, PLBART effectively acquires crucial program aspects, including program syntax, style, and logical flow.

CodeT5 [Wang et al., 2021], undergoes pretraining using the expansive CodeSearchNet dataset [Husain et al., 2020], encompassing both unimodal (PL-only) and bimodal (NL-PL) data across six programming languages: Ruby, JavaScript, Go, Python, Java, and PHP. Moreover, supplementary data from publicly available code repositories on Github

contributes data pertaining to C and C#. This model is introduced as a unified pretrained encoder-decoder model rooted in the T5 framework [Raffel et al., 2020b] that effectively harnesses the semantic information conveyed through developer-assigned identifiers.

The pretraining methodology is underscored by the application of the masked span prediction technique [Raffel et al., 2020a], wherein the input sequence undergoes random masking with multiple masks, and the output sequence comprises the masked contents in the correct order. Meanwhile, [Wang et al., 2021] proposed a new identifier-aware pretraining task that enable the model to differentiate identifier tokens and recover them when masked. Additionally, user-written comments are leveraged through a bimodal dual generation task to achieve improved NL-PL pairs to learn cross-modal alignment. Figure 2.4 illustrates the visual insights into the pretraining tasks. These innovative tasks enable the model to proficiently capture semantic information from code, with a specific emphasis on the crucial token type information that possess substantial code semantics.

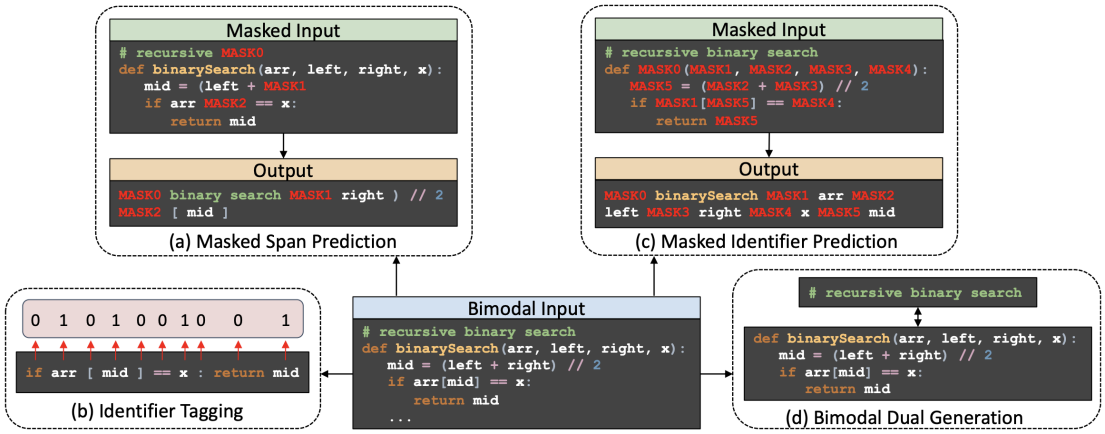

Figure 2.4: Pretraining tasks of CodeT5: sequential training progression and bimodal dual generation. This figure is proposed by Wang et al. [2021].

2.3 Evaluation Metrics

In the domain of Natural Language Processing (NLP), suitable evaluation metrics play a foundational role, exerting significant influence over development, advancement, and practicality. These metrics offer standardized measures to assess the effectiveness of models, enabling objective comparisons and supporting iterative enhancements. Linking theoretical expertise and real-world applicability, they ensure a smooth transition of progress

into practical benefits. These metrics guide decision-making and foster coherent communication, thereby aligning the evolution of NLP with practical goals through quantifiable and relatable advancements. In the following section, we will introduce two important evaluation metrics for text-to-code generation.

2.3.1 BLEU

The BLEU metric, initially proposed by Papineni et al. [2002], serves as an automatic tool for evaluating the quality of machine-generated text, with a specific focus on the domain of machine translation. This metric operates within a range of 0 to 1 and undertakes an evaluation of similarity, with heightened scores indicating an augmented degree of likeness between a machine-generated translation and one or multiple reference human translations. Furthermore, the metric's scoring magnitude elevates with an increased count of reference translations per sentence.

The essence of the BLEU metric can be distilled into two essential components: modified n-gram precision and sentence brevity penalty.

Modified N-gram Precision

The main task of implementing BLEU involves comparing n-grams, which are contiguous sequences of n words, between the machine-generated output and reference translations. This comparison does not consider word positions, focusing solely on matches. More matches indicate a better candidate translation quality. This modified n-gram precision is calculated by collecting the counts of candidate n-gram and their corresponding highest reference, clipping the candidate counts by their reference maximum values, summing them, and then dividing by the total number of candidate n-grams. This aspect of BLEU measures the extent to which the candidate translation employs words that match those in the references, reflecting both the adequacy of translation using similar words and the fluency demonstrated by longer n-gram matches. To address multi-reference test sets, Papineni et al. [2002] computes the n-gram precision p_n using the following approach:

$$p_n = \frac{\sum_{C \in \{candidates\}} \sum_{\text{n-gram} \in C} Count_{clip}(\text{n-gram})}{\sum_{C' \in \{candidates\}} \sum_{\text{n-gram}' \in C'} Count(\text{n-gram}')}$$

Sentence Brevity Penalty

BLEU considers sentence length to address issues like penalizing spurious words and rewarding appropriate word usage. A brevity penalty is introduced to adjust the metric when the candidate translation's length differs from the reference translations' lengths. This penalty aims to align candidate length with reference length ranges, enhancing fairness. The goal is to make the brevity penalty 1.0 when the candidate's length matches any reference translation's length.

BLEU metric calculation

To calculate BLEU score, Papineni et al. [2002] takes geometrical average of modified n-gram precision scores p_n of test corpus and multiply the exponential of of brevity penalty factor.

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

$$\textbf{where BP} = \begin{cases} 1 & \text{if } c > r \\ e^{1-r/c} & \text{if } c \leq r \end{cases}$$

where r is the effective reference length of test corpus and c is the candidate translation length.

2.3.2 CodeBLEU

In the field of code synthesis, as the BLEU metric lacks consideration for grammatical and logical accuracy, Ren et al. [2020] introduced a supplementary automated evaluation metric, named CodeBLEU. This metric retains the strengths of BLEU's n-gram matching while enhancing the evaluation by incorporating code syntax using abstract syntax trees (ASTs) and code semantics through data-flow analysis. This augmentation aims to capture crucial syntactic and semantic attributes of codes, enhancing the assessment of code quality beyond mere n-gram precision.

To emphasize keyword relevance, utilize the structural attributes of trees, and account for semantic logic, this metric is defined as a weighted combination of four components (shown in Figure 2.5): BLEU, weighted BLEU, syntactic AST match, and semantic data-flow match [Ren et al., 2020]. This composite approach enables us to comprehensively assess the quality of code synthesis, incorporating both linguistic and structural aspects.

The CodeBLEU metric is computed as [Ren et al., 2020]:

$$\text{CodeBLEU} = \alpha \cdot \text{BLEU} + \beta \cdot \text{Weighted\_BLEU} + \gamma \cdot \text{AST\_Match} + \delta \cdot \text{Data-flow\_Match}$$

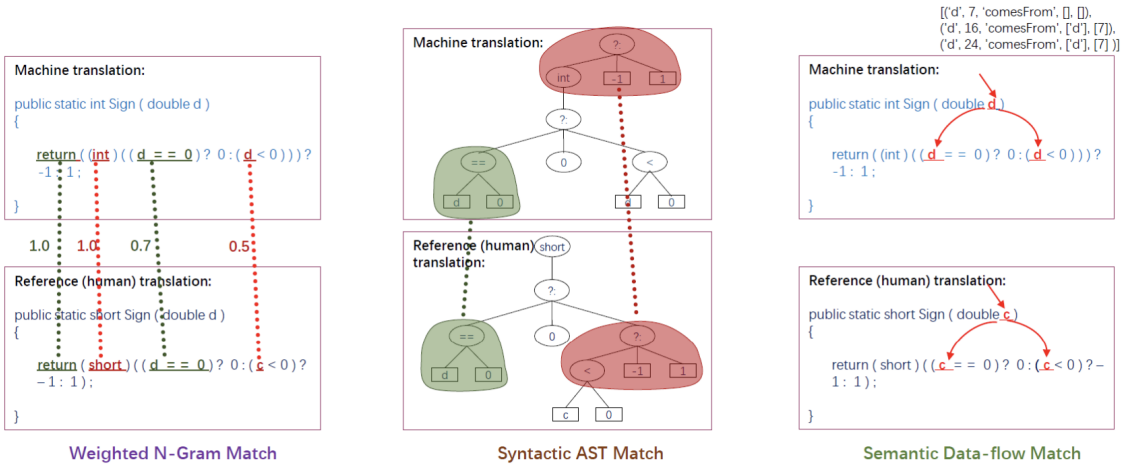

Figure 2.5: Four components of CodeBLEU metric, proposed by Ren et al. [2020]

Weighted BLEU

Illustrated in Figure 2.5, the n-gram precision allocates varying weights to distinct n-grams, emphasizing the significance of programming language keywords. Thus, keywords hold augmented importance through higher assigned weights. Subsequently, the weighted n-gram match precision p_n is computed as follows [Ren et al., 2020]:

$$p_n = \frac{\sum_{C \in \text{Candidates}} \sum_{i=1}^{l} \mu_n^i \cdot \text{Count}_{clip}(C(i, i+n))}{\sum_{C' \in \text{Candidates}} \sum_{i=1}^{l} \mu_n^i \cdot \text{Count}_{clip}(C'(i, i+n))}$$

where μ_n^i is the weights of varying keywords or n-grams, and $\text{Count}_{clip}(C(i, i+n))$ is the count of co-occurring n-grams in both a candidate code and a reference code set, starting from position i.

Similar to the original BLEU, the weighted BLEU score is computed as:

$$\text{Weighted\_BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

Syntactic AST Match

Ren et al. [2020] introduces a scoring mechanism to assess code quality through a syntactic aspect by aligning tree structures, such as Abstract Syntax Trees (ASTs). ASTs serve as tree-based representations of the abstract syntactic structure of programming languages, with each node signifying a construct present in the source code. Consequently, dissimilarities between ASTs can capture grammatical errors like missing tokens and data type inconsistencies. As depicted in Figure 2.5, Ren et al. [2020] extracts sub-trees from both candidate and reference ASTs and computes accuracy using the following formula:

$$\text{AST\_Match} = \frac{\text{Count}_{clip}(\text{T}_{cand})}{\text{Count}(\text{T}_{ref})}$$

where $\text{Count}_{clip}(\text{T}_{cand})$ denotes the count of candidate sub-trees that align with the reference, and $\text{Count}(\text{T}_{ref})$ is the cumulative count of reference sub-trees.

Semantic Data-flow Match

In the context of programming languages, the semantics of source code are intricately tied to the dependency relationships among variables. Despite code exhibiting high similarity, its semantic interpretation can diverge significantly. To address this challenge, Ren et al. [2020] adopted the concept of data-flow [Guo et al., 2021] to represent source code as a graph. In this graph, nodes correspond to variables extracted from the leaves of the AST, while edges depict the origin of each variable's value. This semantic graph effectively quantifies the interplay of variables' values and sources. Hence, such a graph serves as a mechanism to evaluate the semantic alignment between the candidate and reference code, as depicted in Figure 2.5. Similar to the AST match score, the semantic data-flow match score is computed as:

$$\text{Data-flow\_Match} = \frac{\text{Count}_{clip}(\text{DF}_{cand})}{\text{Count}(\text{DF}_{ref})}$$

2.4 Curriculum Learning

The origins of Curriculum Learning (CL) can be traced back to Elman [1993], who introduced the concept of curriculum learning and emphasized the significance of commencing training neural networks with small inputs. Following that, Bengio et al. [2009] provided a theoretical definition of what is a '**curriculum**'.

Definition 1 *[Bengio et al., 2009] Let z denote a random variable that represents an instance for the learning process, $P(z)$ signifies the intended training distribution from which the learner aims to grasp a pertinent function. Here, $0 \leq W_\lambda(z) \leq 1$ denotes the weight assigned to instance z during the curriculum sequence at step λ, where $0 \leq \lambda \leq 1$, and $W_1(z) = 1$. Consequently, the associated training distribution at step λ can be expressed as*

$$Q_\lambda(z) \propto W_\lambda(z)P(z) \quad \forall z$$

such that $\int Q_\lambda(z)dz = 1$, then we have

$$Q_1(z) = P(z) \quad \forall z$$

*We call the corresponding sequence of distributions $Q_\lambda(z)$ is a **curriculum** if the entropy of these distributions increases*

$$H(Q_\lambda) < H(Q_{\lambda+\epsilon}) \quad \forall \epsilon > 0$$

and $W\lambda(z)$ is monotonically increasing in λ, i.e. $W_{\lambda+\epsilon}(z) \geq W_\lambda(z)\ \forall z,\ \forall \epsilon > 0$.

With Bengio et al. [2009]'s pioneering application of CL to deep neural networks, this approach became widely recognized and employed across fields such as natural language processing and computer vision, in a wide range of tasks. As research advances, two distinct levels of curriculum learning application have emerged. At the data level, the framework comprises two essential components: the curriculum criterion, also known as the difficulty metric, and the curriculum scheduler. Definition 1 provides a broad definition of the curriculum criterion, employed to rank examples according to their complexity from easy to challenging during training. Subsequently, a selection methodology determines the instances to be employed for current-phase training. The scheduler, in turn, takes on the responsibility of determining optimal moments for curriculum updates, yielding the most comprehensive performance enhancement. This approach has been predominantly followed across numerous studies [Bengio et al., 2009, Christopoulou et al., 2022, Platanios et al., 2019, Zhang et al., 2018, Gong et al., 2021, Chang et al., 2021]. On an alternative trajectory, numerous curriculum learning investigations pivot toward the model level to augment modeling capacity. This involves activating additional units [Morerio et al., 2017], introducing neural units [Karras et al., 2018], or refining convolutional filters [Sinha et al., 2020] during training. This approach dispenses with the need for a distinct difficulty metric

function. Instead, it hinges on the presence of a model capacity curriculum, delineating how the model's architecture or parameters should be adjusted to accommodate the entirety of training data.

Algorithm 1 General Curriculum Learning Algorithm [Soviany et al., 2022]

1: M - a model
2: E - a training dataset
3: P - performance measure
4: C - a curriculum criterion or difficulty metric
5: S - curriculum scheduler
6: n - number of epochs
7: l - curriculum level: {data level, model level}
8: **for** each $t \in [1, n]$ **do**
9: $\quad p \leftarrow P(M)$
10: $\quad$ **if** $S(t, p) = true$ **then**
11: $\quad\quad M, E \leftarrow C(l, M, E)$
12: $\quad$ **end if**
13: $\quad E^* \leftarrow select(E)$ **if** l is data level
14: $\quad M \leftarrow train(M, E^*)$
15: **end for**

Algorithm 1 provides a generic framework for curriculum learning, integrating with the conventional training loop utilized in machine learning model training. In this paradigm, step 9 involves the computation of the current performance score p, a metric potentially used by the scheduler S to ascertain the optimal timing for implementing the curriculum. The subsequent steps, 10 to 12, encompass the essence of curriculum learning. Depending on the value of l, the curriculum criterion transforms either the dataset E by arranging it in ascending order of difficulty or the model M by enhancing its modeling capacity. It is noteworthy that curriculum can be applied on both data and model levels. Moving forward to step 13, the training loop engages in the selection of a mini-batch E^* at the data level, accomplished through techniques like batching, weighting, or sampling. This mini-batch is then utilized to update the model M at step 14 [Soviany et al., 2022].

Chapter 3

Literature Review

3.0.1 Code Language Models

Following the achievements of large language models in natural language processing, the past few years have seen a range of research in code language models, getting the state-of-the-art outcomes. As an example, CodeT5 [Wang et al., 2021] introduced a pre-training task that incorporates awareness of identifiers and a bimodal dual generation task within a unified pre-trained encoder-decoder Transformer architecture [Raffel et al., 2020b]. This framework has demonstrated superior performance [Zhu et al., 2022] in the text-to-code generation task.

The text-to-code generation tasks, aiming to automated generation of executable source code from provided natural language text prompts [Le et al., 2022], hold significance due to their far-reaching implications within the software industry. These tasks contribute to enhancing productivity and fostering greater accessibility in programming-oriented professions. Recent research in code language models has given rise to a new wave of coding-assistant models, capable of automating the text-to-code conversion process. Among these models, CoPilot (GitHub Copilot) emerged as a pioneering coding companion, leveraging GPT-based language models to offer code suggestions and auto-completions directly within integrated development environments (IDEs). Similarly, Codex [Chen et al., 2021] demonstrates proficiency in generating code functions across various programming languages, thereby facilitating rapid prototyping and enhancing development efficiency. StartCoder and StarChat explore conversational interfaces for code generation, with StartCoder [Li et al., 2023] engaging in dialogues to refine code instructions and StarChat [Tunstall et al., 2023] offering interactive code generator based on the StartCoder within chat-based inter-

actions.

The Scaling Laws [Kaplan et al., 2020] has demonstrated predictable power-law relationships between test loss and various parameters, such as model size, the quantity of training labels, and computational budget. Notably, recent advancements in model performance are rooted in the expansion of model parameters and training data, leading to exponential growth in model complexity and training costs. In this thesis, we adopt a novel approach—leveraging curriculum learning in the code domain—to address this challenge, aiming to expedite convergence and improve performance in a resource-efficient manner.

3.0.2 Curriculum Learning in NLP

The foundational concept [Elman, 1993] of Curriculum Learning (CL) derives its inspiration from the human learning process, which typically involves initially mastering simpler tasks and gradually transitioning to more complex ones, thereby progressively scaling up in difficulty as the learner's proficiency improves. Subsequently, Bengio et al. [2009] extended this concept to deep neural networks, employing a gradual progression from easy to difficult examples to train layers in a greedy manner [Hinton et al., 2006]. Their findings demonstrated that CL has the potential to accelerate convergence and enhance performance across numerous tasks.

In essence, the CL framework encompasses two core components: a difficulty metric that orders training examples in ascending order of complexity and a scheduler responsible for identifying opportune instances for selecting samples to update the model.

In the realm of NLP, the formulation of difficulty metrics typically tends to be intuitive to human comprehension and leverages domain-specific characteristics rooted in linguistic information. For instance, metrics encompassing parameters such as sentence length, occurrences of coordinating elements, or the presence of uncommon words [Platanios et al., 2019, Zhang et al., 2018, Kocmi and Bojar, 2017] within a natural language text and utilized to pre-determine the complexity of training samples before embarking on the actual training process. These pre-computed indicators of difficulty contribute to the construction of a well-structured curriculum. Concurrently, there are automatic metrics grounded in model-based approaches, as exemplified by the computation of training loss at a particular epoch [Gan et al., 2021]. Within our curriculum criteria, we adopt a strategy akin to human intuition, deploying familiar metrics such as text or code length in the context of a large code language model. The main point of our approach involves a preliminary sorting of samples in an offline manner, prior to their training to the model. This initial ordering

is retained consistently throughout the subsequent processes.

On the other hand, the scheduler incorporates diverse strategies, encompassing practices such as weighting, and batching with sampling. Weighting is commonly employed in conjunction with loss metrics, where it interacts with the model by applying weighted factors to assign greater importance to accurately labeled instances and reduces significance to incorrectly labeled ones [Huang and Du, 2019]. The batching scheduler can be implemented with diverse metrics, involving the partitioning of the training dataset into discrete subsets. Subsequently, some pieces of samples are drawn from these subsets based on predetermined sampling criteria. Kocmi and Bojar [2017] introduced a design wherein each example is encountered once within a single epoch, ensuring balanced exposure. Platanios et al. [2019] evaluated the model's competence, which reflects its learning progress, and select training examples that possess difficulty scores lower than the current competence level throughout the training process. Zhang et al. [2018] adopted a probabilistic sampling technique to allocate weights to data. Xu et al. [2020] uniformly divided the training dataset into distinct buckets, showcasing the efficacy of curriculum learning during the fine-tuning phase exclusively.

With demonstrated success in NLP, curriculum learning has exhibited its potential to boost convergence and enhance model performance, our work strives to extend this efficacy to the code domain. Leveraging the successful methodologies and the promising results observed in NLP, our strategy is directed towards investigating how curriculum learning can be effectively employed to enhance the capabilities of models in tasks related to code, such as text-to-code generation. Our aim is to capitalize on the benefits witnessed in NLP and ascertain how similar principles can be adapted and leveraged for code-based challenges.

Chapter 4

Methodology

A curriculum framework comprises two important components: the curriculum criterion (difficulty metric) and the scheduler. The curriculum criterion serves the purpose of evaluating training samples and ranking them based on their levels of difficulty, while the scheduler dictates the timing and selection of instances for each training step. In our experiments, we employ 9 distinct criteria derived from properties of text, code, and generated code outputs from an external model. Additionally, we combine some difficulty metrics based on the performance of individual criteria.

4.1 Curriculum Criteria

As previously mentioned, our approach involves employing an offline sorting method for arranging examples, requiring the need for difficulty metrics to establish a pre-training ranking of data. In the context of the text-to-code generation task, we have the opportunity to incorporate human-intuitive metrics from two distinct perspectives: the natural language aspect and the programming language facet.

4.1.1 Curriculum Criteria of Natural Language Descriptions

Building upon the curriculum criteria as applied in the field of natural language processing, prior research [Kocmi and Bojar, 2017, Platanios et al., 2019, Zhang et al., 2018] have substantiated the viability of employing metrics such as sentence length and the presence of rare words. Consequently, we extend this approach called **text length** and **rarity words** to the text-to-code generation task, exploring the applicability of these metrics in

our domain.

Furthermore, we introduce another complexity metric, namely **text perplexity** evaluated by the pretraining model. Perplexity (PPL) is defined as the exponential average negative log-likelihood of a given sequence. For a tokenized sequence denoted as $X = (x_1, ...x_n)$, the perplexity of X is calculated using the formula,

$$\text{PPL}(X) = \exp\left(-\frac{1}{n}\sum_{i=1}^{n}\log p_\theta(x_i|x_{<i})\right)$$

Here, $\log p_\theta(x_i|x_{<i})$ represents the log-likelihood of the ith token conditioned on the preceding tokens $x_{<i}$, as determined by the pretraining model [HuggingFace]. Perplexity essentially serves as an assessment of the model's capacity to predict tokens uniformly within a specified corpus. In our context, we utilize this metric to assess which type of training data aligns more closely with the distribution of the pretraining model. Data with lower perplexity values are easier to train for the model, indicating a stronger alignment with the model's distribution.

4.1.2 Curriculum Criteria of Programming Languages

For the programming language aspect, we adopt a similar approach to that used in natural language processing, incorporating intuitive methods based on the structural characteristics of code. These methods encompass factors such as **code length** and the **number of code lines**, which offer a obvious reflection of the complexity within a given code program.

Subsequently, our attention shifts towards the content and inherent structure of the code, specifically delving into aspects like the **ratio of alphanumeric** and the **number of variables**.

Ratio of Alphanumeric

In Python programming, the alphanumerical ratio within a code program serves as an informative gauge of the equilibrium between letters and numbers. A higher alphanumerical ratio suggests a prevalence of alphabetic characters in the code, potentially implying the usage of more descriptive identifiers. This practice can significantly enhance the overall readability and comprehension of the code. As such, this metric can be harnessed to evaluate the style of identifier naming and the overall readability of the code. We employ the packages '**.isalpha()**' and '**.isalnum()**' to differentiate digits and letters from other special

characters.

$$\text{Ratio} = \frac{number\ of\ isalpha()}{number\ of\ isalnum()}$$

A higher alphanumeric ratio indicates more amenable for curriculum learning.

Number of Variables

As demonstrated by Guo et al. [2021], the incorporation of data-flow in code language models highlights the significance of variables in code-related tasks. In this criterion, we quantify the distinct count of variables in each program by utilizing the abstract syntax tree, as depicted in Figure 4.1. In this representation, each leaf node symbolizes a variable within the formal language of the source code. Programs with fewer variables are considered as data that is easier to train.

```
if a = b
  then
    return "equal"
  else
    return a + " not equal to " + b
```

Number of variables: 2

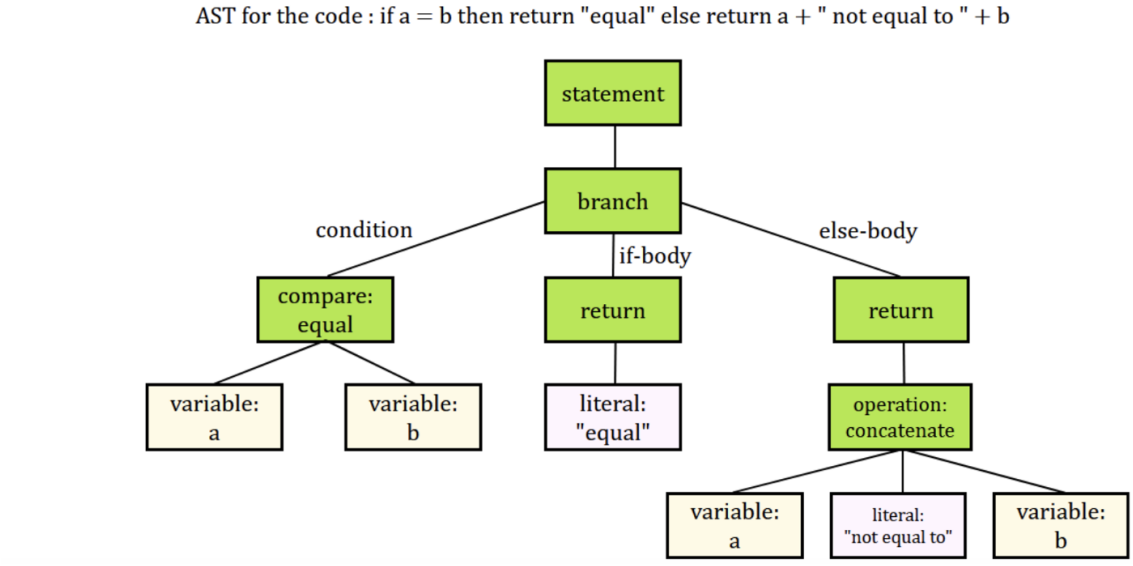

Figure 4.1: Number of variables from an abstract syntax tree and its corresponding source code. (the example of the AST from [Diaz et al.])

4.1.3 Teacher-Student CL

Teacher-student curriculum learning is a distinctive approach within curriculum methods that divides the training process into two segments. In this method, the student model is dedicated to mastering the original task, while an external model, referred to as the teacher, assesses the difficulty of training examples. Building on the foundation laid by Zhang et al. [2018], we adopt a similar strategy by employing a simple auxiliary model to generate difficulty metric scores for each training sample.

Generated Code Perplexity

Similar to the computation of text perplexity described in Section 4.2.1, we apply the same formula to calculate the perplexity of generated codes. A lower perplexity value for the generated code signifies its proximity to the distribution of the training data, indicating higher fluency and alignment with the programming language structure. Conversely, a higher perplexity value suggests potential incoherence and unnaturalness in the generated code, making it challenging to trained by the model.

Generated CodeBLEU Scores

the CodeBLEU score serves as a pivotal metric for assessing the quality of text-to-code generation. When constructing the curriculum criterion, reliance on this score imparts a high level of credibility. With this understanding, we introduce an auxiliary model to compute this metric for each training data point.

$$\text{CodeBLEU} = \alpha \cdot \text{BLEU} + \beta \cdot \text{BLEU}_{weighted} + \gamma \cdot \text{Match}_{AST} + \delta \cdot \text{Match}_{dataflow}$$ [Ren et al., 2020]

Given that CodeBLEU measures the similarity between the target and the prediction, a higher score indicates a higher likelihood of achieving improved outcomes. Notably distinct from other curriculum strategies, the generated CodeBLEU scores are arranged in descending order, reflecting their ranking from higher to lower scores.

4.2 Curriculum Scheduler

4.2.1 Scheduler for Individual Criterion

Generally, as Algorithm 1, let d_i denote an instance within the training dataset D. In the initial step, the objective is to attribute each d_i with a complexity score $c_i \in C$, signifying its intricacy with respect to the model as evaluated by diverse difficulty metrics. In the next step, we employ the Annealing scheduler as proposed by Xu et al. [2020]. Utilizing the scores c_i, the samples are re-organized and allocated into k difficulty-level buckets $\{C_j : j = 1, ..., K\}$, ordered from the simplest to the hardest, with possibly varying sizes. Subsequently, we adopt a random selection, choosing 1/(K+1) percentage of samples from all preceding buckets according to the following:

$$S_1 = C_1$$

$$S_2 = C_2 + \frac{1}{K+1} \cdot S_1$$

$$S_3 = C_3 + \frac{1}{K+1} \cdot S_1 + \frac{1}{K+1} \cdot S_2$$

$$\cdots$$

$$S_K = C_K + \frac{1}{K+1} \cdot S_1 + \cdots + \frac{1}{K+1} \cdot S_{K-1}$$

The final curriculum is structured as $\{S_j : j = 1, ..., K\}$, with each S_j represents a distinct subset. Subsequently, every S_j subset is subjected to shuffling and trained for a single epoch. In continuation of the training process, an additional stage denoted as S_{K+1} is introduced. This stage involves random selection of examples from the complete training dataset until the point of model convergence.

4.2.2 Scheduler for Combination of Criteria

Continuing the structure of the Annealing scheduler for individual criterion, we have also formulated a similar scheduler for combinations of two distinct difficulty metrics. As outlined in Algorithm 2, designated as m_1 and m_2, two distinct strategies are arranged by ranking and subsequently partitioned into separate buckets denoted as C_1 and C_2. Importantly, it's worth noting that the sizes of these two groups are not required to be equal. Following that, C_1 is established as the primary curriculum, and random samples are selected from R, which is assembled using an alternative curriculum C_2 as represented in

section 4.2.1. The combination of C_1 buckets and the samples from R is then utilized to establish the stages S for the purpose of sequential training.

Algorithm 2 Combination Curriculum Criteria Algorithm

m_1: the main curriculum
m_2: the second curriculum, which selecting random samples by this metric
C_1: assess the training samples by m_1 and split into K_1 buckets $\{C_{1,i} : i = 1, ..., K_1\}$
C_2: assess the training samples by m_2 and split into K_2 buckets $\{C_{2,j} : j = 1, ..., K_2\}$
Note: K_1 and K_2 can be different
$S_1 = C_{1,1}$
$R_1 = C_{2,1}$, stages of m_2 for random selections.
for $i \in [2, min(K_1, K_2)]$ **do**
    $r = \frac{1}{K_2+1} \cdot R_1 + \cdots \frac{1}{K_2+1} \cdot R_{i-1}$ random selection from previous buckets
    $R_i = C_{2,i} + r$
    $S_i = C_{1,i} + r$
end for
if $K_1 > K_2$ **then**
    for $i \in [K_2 + 1, K_1]$ **do**
        $S_i = C_{1,i}$
    end for
end if
if $K_1 < K_2$ **then**
    for $i \in [K_1 + 1, K_2]$ **do**
        $R = \frac{1}{K_2+1} \cdot R_1 + \cdots \frac{1}{K_2+1} \cdot R_{i-1}$
        $R_i = C_{2,i} + R$
        $S_i = R$
    end for
end if
$S = \{S_i : i = 1, ..., max(K_1, K_2)\}$

Chapter 5

Experimental Results and Discussion

5.1 Datasets and Experiment Setup

5.1.1 Datasets

We choose the XLCoST dataset provided by Zhu et al. [2022]. This dataset comprises thousands of program-level samples from seven distinct programming languages, namely C, C++, C#, Java, JavaScript, PHP, and Python. Notably, the solution programs for the same problem exhibit uniform structure, even down to the variable names. This semantic consistency across languages proves advantageous for tasks such as code translation and other language-related operations.

To ensure effective code generation, the input text must be detailed and informative. Zhu et al. [2022] adopted a combination of problem descriptions and step-by-step comments as the input for generating the entire program. The programs are well-commented, with an average of 9 comments per data, providing substantial context and guidance for the models during the synthesis process, an example is shown in Figure 5.1 and Table 5.1 displays some statistical results about the training data.

average number of comments for descriptions	9 comments
average number of lines of codes	20 lines
minimum number of lines of codes	2 lines
maximum number of lines of codes	127 lines

Table 5.1: Statistics analysis for each data in the training dataset.

In the experiments, we use the Python program-level subset extracted from the XLCoST dataset. This subset encompasses a total of 9263 training samples, accompanied by 472

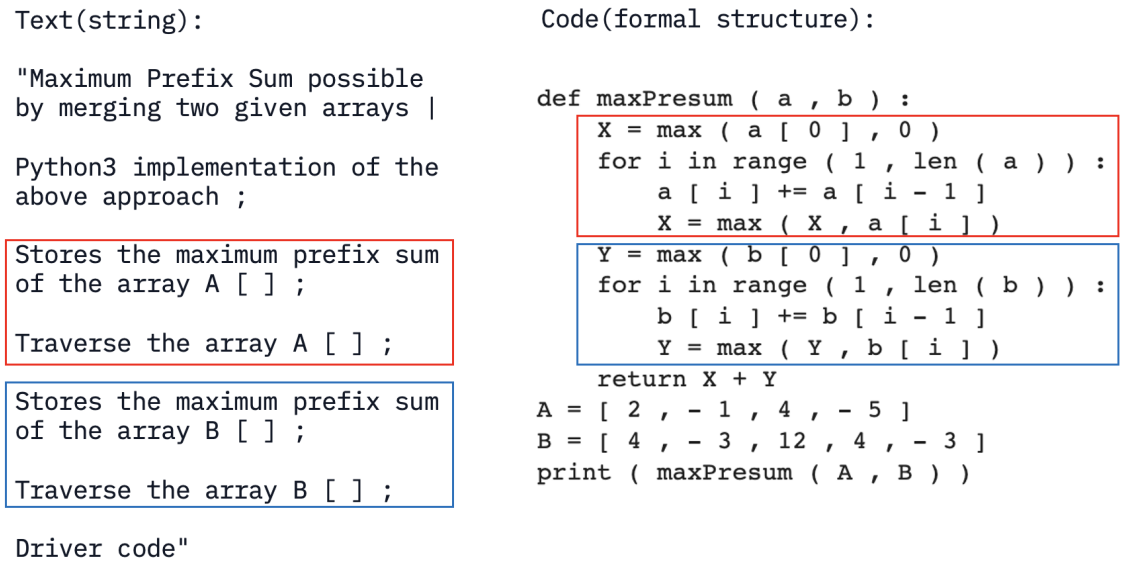

Figure 5.1: Data example. The problem description and important comments with the corresponding snippets.

validation samples and 887 samples allocated for testing purposes. However, this selected subset also presents certain issues, as some of the code samples within it do not conform to the correct code structure, as depicted in Figure 5.2. As a result, these code samples cannot

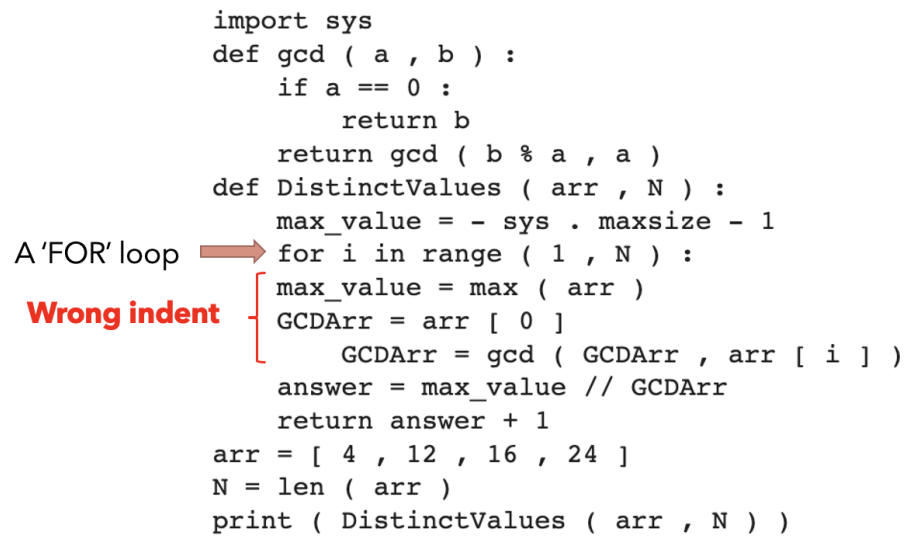

Figure 5.2: The wrong code example. The 10th sample of the training set.

be adequately represented using an abstract syntax tree. In order to make sure the models are not trained on incorrect code samples, a filtering step was applied to the dataset to get

rid of problematic samples from both the training and validation datasets while retaining the complete test data. Subsequently, a new dataset has been constructed to be employed in the following experiments. Table 5.2 presents the number of program-level samples in Python between the original and the filtered dataset.

	Train	Valid	Test	Total
XLCoST dataset	9263	472	887	10622
Filtered dataset	8485	433	887	9805

Table 5.2: The train-valid-test split of Python-Program-Level dataset

5.1.2 Experiment Setup

We closely replicate the hyper-parameter settings applied in Zhu et al. [2022]'s work, with specific configurations (shown in the Table 5.3), including a batch size of 16 and a maximum tokenizer length of 400 for the both encoder and the decoder, and set the evaluation step as 50. However, we utilize the Large CodeT5 model, which encompasses 770 million parameters, enabling enhanced convergence rates and efficiency. Furthermore, to generate the code, we set the maximum length to 1024, which can completely cover the length of the most of programs in training data. The distribution of code length for training data is shown in Figure 5.3. To optimize the generation and achieve greater stability in outcomes, we implement beam search for code generation. We experiment with different beam sizes, specifically setting the number of beams to 3 and 5 in comparison to the baseline model. Despite the similar CodeBLEU scores, we observe that the code generated with beams set to 5 exhibits higher error rate (70 and 84 respectively). Consequently, we opt to maintain a beam size of 3 for optimal results.

model name	CodeT5-large-ntp-py
batch size	16
encoder tokenizer length	400
decoder tokenizer length	400
evaluation step	50
maximum length of model generating	1024
number of beams	3

Table 5.3: Hyper-parameters setting.

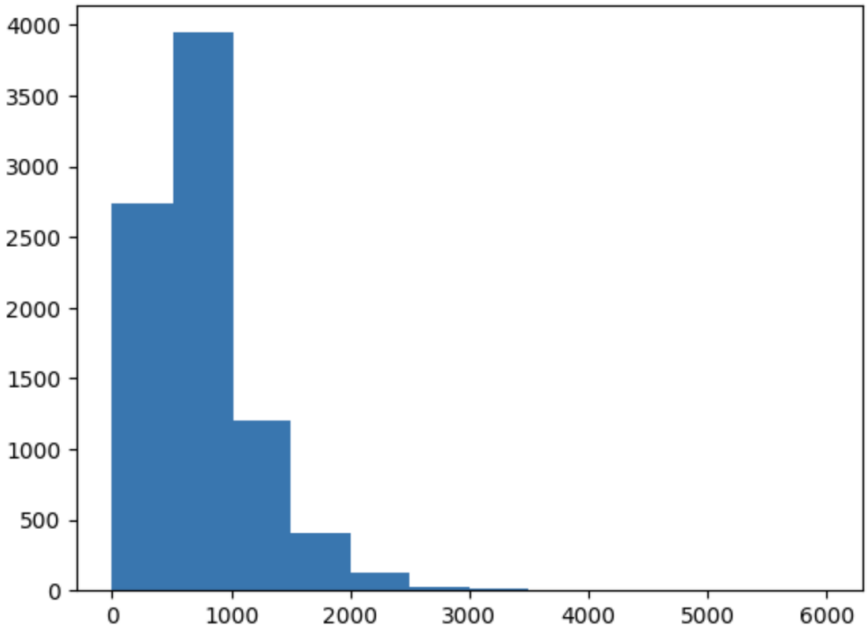

Figure 5.3: Distribution of code length for training data.

5.2 Baseline Model

In our experiment, we employ the pretraining model 'CodeT5-large-ntp-py' [Le et al., 2022], building upon the foundational architecture and techniques of CodeT5 [Wang et al., 2021]. This model involves an expansion of the Python pretraining dataset. The size of this dataset is approximately ten times larger than the one used to train the original CodeT5 model. Furthermore, to enhance the program synthesis capability, Le et al. [2022] introduced the application of next-token prediction (NTP) during the pretraining process of CodeT5. It's noteworthy to mention that, in contrast to the broader language support of CodeT5, the scope of this approach is limited to just six programming languages as defined in CodeSearchNet [Husain et al., 2020].

Due to limited computational resources, we conduct model training for a duration of 800 steps in order to compare and evaluate the effects of various difficulty metrics. The selection of this specific training steps was determined by the necessity to ensure that all curriculum strategies comprehensively utilize the entire training dataset at least once during the training process. Subsequently, we proceed to continue training the baseline model, increasing the number of steps, and then observed that as the model further trains and converges, both BLEU and CodeBLEU scores exhibit noticeable improvements, as demonstrated in Table 5.4. Therefore, a subset of models underwent extended training to

achieve enhanced and more stable results. This additional training aimed to assess the efficacy of curriculum learning and to conduct a comprehensive analysis of the quality of generated code.

model	BLEU	CodeBLEU
800steps baseline	31.33	33.94
1500steps baseline	33.88	36.38

Table 5.4: Results of baseline models for different training steps

5.3 Analysis of Results

5.3.1 Individual Curriculum Criteria

difficulty metric	BLEU	CodeBLEU	increasing rate of CodeBLEU
baseline	31.33	33.95	
text length	**32.71**	35.01	3.12%
rarity words	32.68	34.84	2.62%
text perplexity	32.35	**35.28**	**3.92%**
number of code lines	32.16	35.09	3.36%
code length	31.18	34.51	1.65%
alphanumerical ratio	32.05	34.8	2.5%
number of variables	32.31	35.13	3.48%
generated code perplexity	32.07	34.73	2.3%
generated CodeBLEU	32.40	35.24	3.8%

Table 5.5: Experimental results for different curriculum criteria

We primarily conducted a comparison of the BLEU and CodeBLEU scores across all curriculum criteria. The results were categorized into three groups based on the attributes of the text, code, and generated code from an auxiliary model, as presented in Table 5.5. It is evident that the models employing curriculum learning do exhibit more or less improvement compared with the baseline. The top-performing model, labeled as '**text perplexity**', achieved an impressive **3.92%** increase in CodeBLEU score. As anticipated, the strategies involving '**number of variables**' and '**number of variables**' also yielded significant enhancements, **3.8%** and **3.48%** respectively, given their focus on inherent

code attributes. It is not surprising that criteria centered around text yielded better BLEU scores compared to other models, given their emphasis on natural language. However, metrics derived from the auxiliary model did not yield exceptionally higher results, as initially expected. This could be attributed to the constraint of using only the pretraining 'CodeT5-base' model due to limited computational resources, instead of the same pretraining model employed in the principal task.

Furthermore, it's important to acknowledge that the effectiveness of a difficulty metric can be influenced by several factors, such as the number of buckets, the size of each bucket, and the number of training passes for each bucket. Given the complexity of finding the optimal settings for each strategy, this thesis focuses on presenting a range of potential curriculum criteria to illustrate the substantial benefits of curriculum learning in enhancing the quality of text-to-code generation.

5.3.2 Combination of Curriculum Criteria

Subsequently, we moved forward with the selection of the top three performing metrics, adhering to the initial bucket arrangement, in order to assess their combined performance.

difficulty metric	BLEU	CodeBLEU	increasing rate of CodeBLEU
baseline	31.33	33.95	
text perplexity with number of variables	32.27	34.75	2.36%
text perplexity	32.35	35.28	3.92%
number of variables with text perplexity	32.01	**35.55**	**4.71%**
number of variables	32.31	35.13	3.48%
generated CodeBLEU with number of variables	31.93	34.96	2.97%
generated CodeBLEU	32.40	35.24	3.8%
generated CodeBLEU with text perplexity	32.37	**35.29**	**3.95%**

Table 5.6: Experimental results for combination curriculum strategies

As indicated in Table 5.6, we have highlighted the combination methods that exhibit improvements when compared to two individual criteria. While certain metric combinations do yield additional improvements in the CodeBLEU score, it necessitates further experimentation to substantiate this observation. Moreover, the table underscores the importance of selecting the primary curriculum, which significantly influences the outcomes of the combined strategies.

5.4 Analysis of Extended Training Models

For a more comprehensive analysis, we extended the training of the models that incorporate the top three most effective criteria and the best-performing combination model. This extended training procedure entailed the complete shuffling of the entire training dataset following the curriculum learning epochs, and continued until reaching a training step count of 1500. The objective of this extension was to enhance the convergence of the model. In addition to the previously utilized evaluation metrics, BLEU and CodeBLEU, we introduced a distinct metric called '**number of non-executable samples**' to quantify the count of generated programs that fail to execute. The results of this analysis are depicted in Table 5.7.

difficulty metric	BLEU	CodeBLEU	increasing rate of CodeBLEU	number of non-executable samples	reduced rate of non-executable samples
baseline	33.88	36.38		70	
generated CodeBLEU	34.31	36.65	0.74%	**58**	**17.14%**
number of variables	34.16	36.92	1.46%	62	11.43%
text perplexity	34.47	**37.04**	**1.81%**	68	2.86%
number of variables with text perplexity	34.09	36.88	1.37%	67	4.29%

Table 5.7: CodeBLEU and the number of non-executable samples for outperforming models

As the number of training steps increases, the CodeBLEU scores exhibit a noticeable improvement in comparison to models trained exclusively during the curriculum learning epochs. However, the rate of increase has diminished over time. This reduction can be attributed to the fact that models subjected to curriculum learning require a certain number of steps (approximately 200) to adapt their distribution when transitioning to regular random training steps. Given this observation, it can be inferred that even if we were to continue training the models, the efficacy of curriculum learning is unlikely to deteriorate compared to the current results in Table 5.7.

5.4.1 Non-Executable samples

In terms of the metric indicating the number of non-executable samples, even the top-performing model, denoted as '**generated CodeBLEU**,' still exhibits a notable count of errors. This observation prompted us to conduct an in-depth analysis of the generated programs to further understand the underlying issues. The result is shown in Figure 5.4.

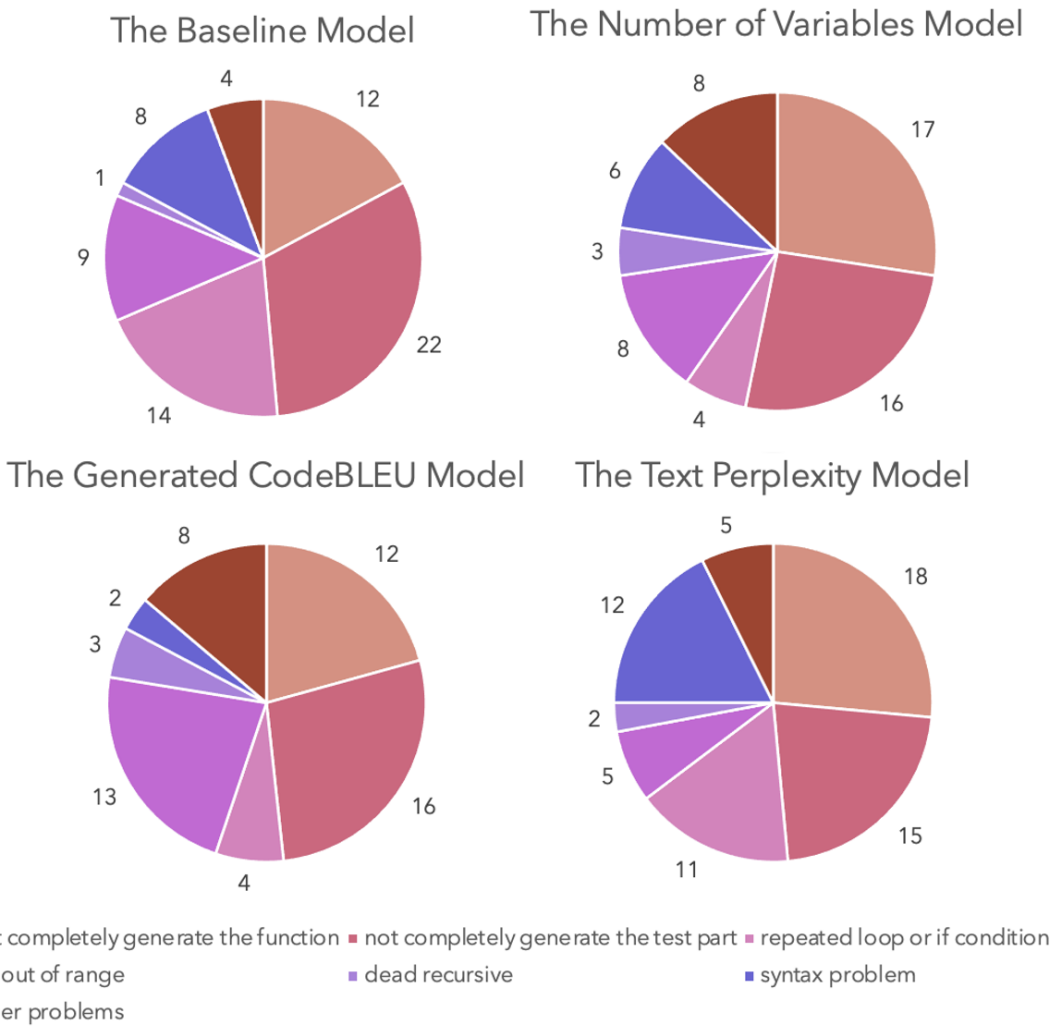

Figure 5.4: The distribution of non-executable samples.

At the outset, errors were categorized into seven distinct types for comprehensive analysis. These categories encompass instances where the generated code: incompletely forms a function, incompletely generates the test section, fails due to repeated loops or 'if' conditions, results in a list being out of range, exceeds time limits, displays syntax issues, and involves other miscellaneous problems. Notably, the "run out of time" category includes

occurrences of dead recursion and dead loops. Meanwhile, the 'syntax problem' category encompasses issues such as invalid or undefined names, incorrect indentation, and function parameter-related errors. All remaining issues, including mathematical or space issues, were categorized as 'other problems'.

Analysis

Figure 5.4 illustrates that the baseline model (located in the upper left quadrant) predominantly yields non-executable programs due to incomplete code generation. Notably, incompletely generating the function is largely attributed to model limitations, as our approach involves truncating programs that exceed the maximum length. This truncation hinders the model's ability to generate excessively long code. This limitation becomes more pronounced in the '**number of variables**' and '**text perplexity**' models (shown in upper and lower right in Figure 5.4), where the aforementioned ability further diminishes. Conversely, the "**number of variables**" model and the '**generated CodeBLEU**' model (lower left in Figure 5.4) significantly mitigate issues related to generate the repeated loops. Furthermore, they also exhibit improvements in generating complete test parts. Notably, the '**generated CodeBLEU**' model outperforms others in addressing syntax problems, albeit showing weaker performance in handling list-related issues. These two curriculum criteria, which emphasize the code's structural aspects, markedly enhance their performance in this metric. As for the '**text perplexity**' model, it demonstrates a bit improvement in addressing list problems but encounters a higher incidence of syntax problems. This discrepancy can be attributed to the curriculum strategy's focus on perplexity within input texts, rather than code structure.

Combination of Generated CodeBLEU and the Number of Variables Models

The substantial improvements demonstrated by the '**generated CodeBLEU**' and '**number of variables**' models motivate us to further investigate their combined impact. To achieve this, we employ the combination scheduler that merges the strengths of these two successful curriculum criteria. By harnessing the benefits between their respective strategies, we seek to amplify their performance gains and potentially achieve an even higher level of proficiency in the text-to-code generation task.

As depicted in Figure 5.5, our combined model showcases remarkable progress, registering a mere 54 non-executable samples. This reflects a substantial reduction of **22.86%** in the non-executable rate, when contrasted with the baseline model. The main reason of

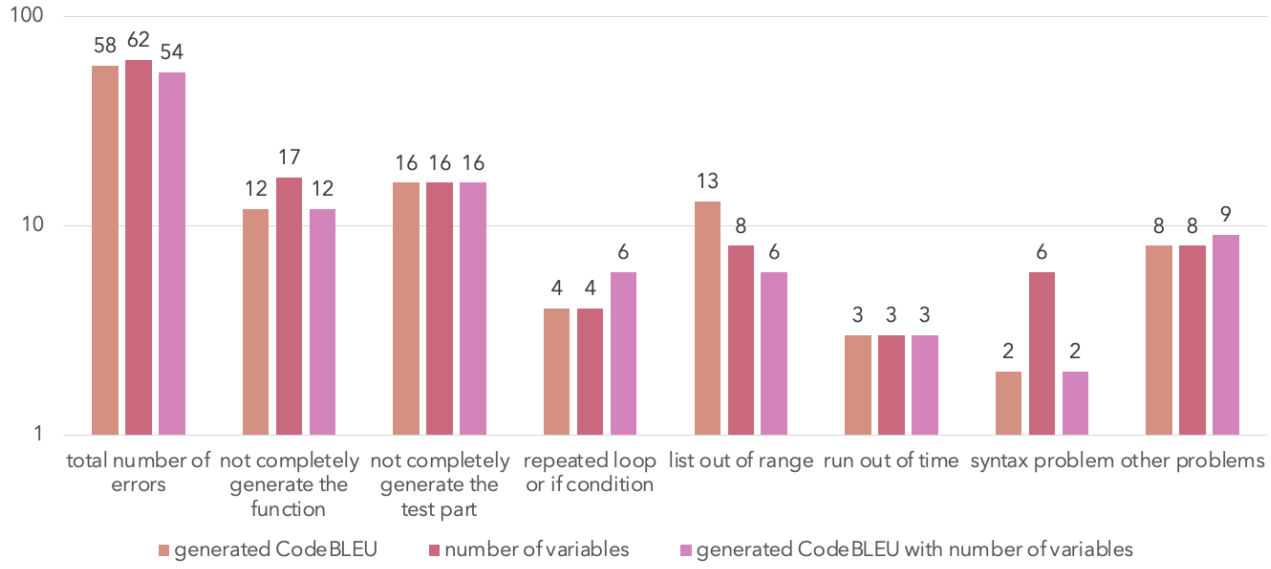

Figure 5.5: The performance of the generated CodeBLEU and the number of variables models and their combination.

this achievement is we integrate the strengths of two distinct curriculum criteria, resulting in a model that inherits the advantages of both the 'number of variables' and 'generated CodeBLEU' models. Notably, the combination model exhibits enhanced proficiency in handling list-related challenges, a skill attributed to the 'number of variables' curriculum strategy. Simultaneously, it excels in generating lengthy code sequences and optimizing syntax issues, as evidenced by the prowess of the 'generated CodeBLEU' model.

This outcome underscores the power of judicious combination methods in harnessing the potential of curriculum learning to a greater extent. By strategically merging effective criteria, we can harness their complementary strengths and achieve performance levels that exceed what individual strategies could offer. This observation reinforces the notion that a thoughtful approach to combining distinct curriculum criteria can yield further enhancements in the realm of curriculum learning.

Chapter 6

Conclusion

Drawing inspiration from successful applications in natural language processing, this study investigates the impact of curriculum learning within the code domain. Through a exploration of diverse difficulty metrics and strategic scheduling, our work aim to improve the convergence and performance of models engaged in text-to-code generation.

We systematically evaluate the effects of various curriculum criteria on code generation tasks. These criteria encompass a spectrum of attributes, from text-based metrics to structural elements inherent in programming languages. By adopting a principled approach and leveraging insights from both natural language and programming, we construct a robust curriculum framework. This approach effectively guide the training process, ensuring that models encounter progressively challenging examples in a strategic manner.

Our analysis unveil a series of compelling findings. Notably, the integration of curriculum learning consistently lead to improvements across different metrics. Models trained with curriculum learning exhibit enhanced performance, particularly evident in the CodeBLEU scores. While some difficulty metrics showcase more substantial gains than others, each curriculum strategy contribute to the refinement of code generation capabilities.

Furthermore, our exploration venture into the intricate realm of combining curriculum criteria. This endeavor demonstrate that the judicious fusion of diverse strategies can yield synergistic enhancements. By blending attributes from separate curriculum criteria, we achieve models that outperformed individual strategies, showcasing the potential for further advancements through thoughtful combinations.

In conclusion, our study underscores the transformative impact of curriculum learning on text-to-code generation tasks. By tailoring curriculum criteria to the specific demands of the code domain and strategically scheduling training samples, we achieve significant

improvements in model convergence and performance. As the field of code language models continues to evolve, the integration of curriculum learning promises to be a pivotal strategy for advancing the frontiers of automated code generation.

Limitation

Our study has several limitations that offer insights into potential areas for improvement. Due to resource constraints, we could only conduct preliminary explorations of different curriculum strategies for text-to-code generation, without thoroughly optimizing bucketing strategies. The choice of an external model for generating difficulty metrics might not have been the most optimal for our task, and the impact of these metrics could be more comprehensive with fine-tuned models. We only conducted single-run experiments for some criteria, which limited the robustness of our conclusions, and the effectiveness of each curriculum criterion warrants further validation through repeated experiments. While our study focused on text-to-code generation, the generalizability of these strategies to other code-related tasks requires further investigation.

Bibliography

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation, 2021. 2.2.3

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 41–48, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605585161. doi: 10.1145/1553374.1553380. URL https://doi.org/10.1145/1553374.1553380. 1.1, 2.4, 1, 2.4, 3.0.2

Ernie Chang, Hui-Syuan Yeh, and Vera Demberg. Does the order of training samples matter? improving neural data-to-text generation with curriculum learning, 2021. 2.4

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. 1.1, 2.2, 2.2.2, 3.0.1

Fenia Christopoulou, Gerasimos Lampouras, and Ignacio Iacobacci. Training dynamics for curriculum learning: A study on monolingual and cross-lingual NLU. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 2595–

2611, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.167. URL https://aclanthology.org/2022.emnlp-main.167. 2.4

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423. (document), 2.1, 2.1, 2.2, 2.2.1

Caupolican Diaz, Kyra Thompson, and Steven Swiniarski. Abstract syntax tree. URL https://www.codecademy.com/resources/docs/general/developer-tools/abstract-syntax-tree. (document), 4.1

Jeffrey L. Elman. Learning and development in neural networks: the importance of starting small. *Cognition*, 48(1):71–99, 1993. ISSN 0010-0277. doi: https://doi.org/10.1016/0010-0277(93)90058-4. URL https://www.sciencedirect.com/science/article/pii/0010027793900584. 2.4, 3.0.2

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020. 2.2.1

Zifa Gan, Hongfei Xu, and Hongying Zan. Self-supervised curriculum learning for spelling error correction. In *Conference on Empirical Methods in Natural Language Processing*, 2021. URL https://api.semanticscholar.org/CorpusID:243865335. 3.0.2

Yantao Gong, Cao Liu, Jiazhen Yuan, Fan Yang, Xunliang Cai, Guanglu Wan, Jiansong Chen, Ruiyao Niu, and Houfeng Wang. Density-based dynamic curriculum learning for intent detection. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, CIKM '21, page 3034–3037, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384469. doi: 10.1145/3459637.3482082. URL https://doi.org/10.1145/3459637.3482082. 2.4

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin

Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow, 2021. 2.2.1, 2.3.2, 4.1.2

Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, jul 2006. ISSN 0899-7667. doi: 10.1162/neco.2006.18.7.1527. URL https://doi.org/10.1162/neco.2006.18.7.1527. 3.0.2

Yuyun Huang and Jinhua Du. Self-attention enhanced CNNs and collaborative curriculum learning for distantly supervised relation extraction. 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 389–398, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1037. URL https://aclanthology.org/D19-1037. 3.0.2

HuggingFace. Perplexity of fixed-length models. URL https://huggingface.co/docs/transformers/perplexity. 4.1.1

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2020. 2.2.3, 5.2

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020. 3.0.1

Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation, 2018. 2.4

Tom Kocmi and Ondřej Bojar. Curriculum learning and minibatch bucketing in neural machine translation. In *Proceedings of the International Conference Recent Advances in Natural Language Processing, RANLP 2017*, pages 379–386, Varna, Bulgaria, September 2017. INCOMA Ltd. doi: 10.26615/978-954-452-049-6_050. URL https://doi.org/10.26615/978-954-452-049-6_050. 3.0.2, 4.1.1

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *NeurIPS*, 2022. 3.0.1, 5.2

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension, 2019. 2.2.3

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023. 1.1, 2.2, 3.0.1

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin 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. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021. 2.2.2

Pietro Morerio, Jacopo Cavazza, Riccardo Volpi, Rene Vidal, and Vittorio Murino. Curriculum dropout, 2017. 2.4

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis, 2023. 1.1, 2.2.2

OpenAI. Gpt-4 technical report, 2023. 2.1

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics. doi:

10.3115/1073083.1073135. URL https://aclanthology.org/P02-1040. 2.3.1, 2.3.1, 2.3.1

Emmanouil Antonios Platanios, Otilia Stretcu, Graham Neubig, Barnabas Poczos, and Tom M. Mitchell. Competence-based curriculum learning for neural machine translation, 2019. 2.4, 3.0.2, 4.1.1

Alec Radford and Karthik Narasimhan. Improving language understanding by generative pre-training, 2018. URL https://api.semanticscholar.org/CorpusID:49313245. 2.1, 2.1

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140): 1–67, 2020a. URL http://jmlr.org/papers/v21/20-074.html. 2.2.3

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(1), jan 2020b. ISSN 1532-4435. 2.2.3, 3.0.1

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis, 2020. (document), 2.3.2, 2.5, 2.3.2, 2.3.2, 2.3.2, 4.1.3

Samarth Sinha, Animesh Garg, and Hugo Larochelle. Curriculum by smoothing. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546. 2.4

Petru Soviany, Radu Tudor Ionescu, Paolo Rota, and Nicu Sebe. Curriculum learning: A survey, 2022. 1, 2.4

Lewis Tunstall, Nathan Lambert, Nazneen Rajani, Edward Beeching, Teven Le Scao, Leandro von Werra, Sheon Han, Philipp Schmid, and Alexander Rush. Creating a coding assistant with starcoder. *Hugging Face Blog*, 2023. https://huggingface.co/blog/starchat. 1.1, 2.2, 3.0.1

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon,

U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf. (document), 2.1, 2.1

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021. (document), 2.2.3, 2.4, 3.0.1, 5.2

Benfeng Xu, Licheng Zhang, Zhendong Mao, Quan Wang, Hongtao Xie, and Yongdong Zhang. Curriculum learning for natural language understanding. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6095–6104, Online, July 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.542. URL https://aclanthology.org/2020.acl-main.542. 3.0.2, 4.2.1

Xuan Zhang, Gaurav Kumar, Huda Khayrallah, Kenton Murray, Jeremy Gwinnup, Marianna J Martindale, Paul McNamee, Kevin Duh, and Marine Carpuat. An empirical exploration of curriculum learning for neural machine translation, 2018. 2.4, 3.0.2, 4.1.1, 4.1.3

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K. Reddy. Xlcost: A benchmark dataset for cross-lingual code intelligence, 2022. 3.0.1, 5.1.1, 5.1.2

Appendix A

Other appendices, e.g. code listing