

Deep Learning for Code Generation using Snippet Level Parallel Data

Aneesh Jain

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Application

Chandan K Reddy, Chair
Ismini Lourentzou
Muhammad Ali Gulzar

December 7, 2022
Blacksburg, Virginia

Keywords: Deep Learning, Code Dataset, Code Translation, Software Development,
Compilation, Reinforcement Learning

Copyright 2022, Aneesh Jain

Deep Learning for Code Generation using Snippet Level Parallel Data

Aneesh Jain

(ABSTRACT)

In the last few years, interest in the application of deep learning methods for software engineering tasks has surged. A variety of different approaches like transformer based methods, statistical machine translation models, models inspired from natural language settings have been proposed and shown to be effective at tasks like code summarization, code synthesis and code translation. Multiple benchmark data sets have also been released but all suffer from one limitation or the other. Some data sets only support a select few programming languages while others support only certain tasks. These limitations restrict researchers' ability to be able to perform thorough analyses of their proposed methods. In this work we aim to alleviate some of the limitations faced by researchers who work in the paradigm of deep learning applications for software engineering tasks. We introduce a large, parallel, multi-lingual programming language data set that supports tasks like code summarization, code translation, code synthesis and code search in 7 different languages. We provide benchmark results for the current state of the art models on all these tasks and we also explore some limitations of current evaluation metrics for code related tasks. We provide a detailed analysis of the compilability of code generated by deep learning models because that is a better measure of ascertaining usability of code as opposed to scores like BLEU and CodeBLEU. Motivated by our findings about compilability, we also propose a reinforcement learning based method that incorporates code compilability and syntax level feedback as rewards and we demonstrate it's effectiveness in generating code that has less syntax errors as compared to baselines. In addition, we also develop a web portal that hosts the models we have trained for code translation. The portal allows translation between 42 possible language pairs and also allows users to check compilability of the generated code. The intent of this website is to give researchers and other audiences a chance to interact with and probe our work in a user-friendly way, without requiring them to write their own code to load and inference the models.

Deep Learning for Code Generation using Snippet Level Parallel Data

Aneesh Jain

(GENERAL AUDIENCE ABSTRACT)

Deep neural networks have now become ubiquitous and find their applications in almost every technology and service we use today. In recent years, researchers have also started applying neural network based methods to problems in the software engineering domain. Software engineering by its nature requires a lot of documentation, and creating this natural language documentation automatically using programs as input to the neural networks has been one of their first applications in this domain. Other applications include translating code between programming languages and searching for code using natural language as one does on websites like stackoverflow. All of these tasks now have the potential to be powered by deep neural networks. It is common knowledge that neural networks are data hungry and in this work we present a large data set containing codes in multiple programming languages like Java, C++, Python, C#, Javascript, PHP and C. Our data set is intended to foster more research in automating software engineering tasks using neural networks. We provide an analysis of performance of multiple state of the art models using our data set in terms of compilability, which measures the number of syntax errors in the code, as well as other metrics. In addition, we propose our own deep neural network based model for code translation, which uses feedback from programming language compilers in order to reduce the number of syntax errors in the generated code. We also develop and present a website where some of our code translation models have been hosted. The website allows users to interact with our work in an easy manner without any knowledge of deep learning and get a sense of how these technologies are being applied for software engineering tasks.

Acknowledgments

I would like to thank my advisor, Dr. Chandan K. Reddy for his continued support and guidance throughout the pursuit of my Master's degree. His advice has been crucial in helping me accomplish my objectives and I sincerely appreciate the time and energy he has invested with me. I would like to also thank Dr. Ismini Lourentzou, who supported me early on in my journey with her wise words and feedback and helped me gain clarity about the kinds of problems I wanted to pursue. I would like to thank Dr. Muhammad Ali Gulzar as well for his valuable feedback on my work.

I would also like to thank all the wonderful collaborators I got the opportunity to work with in my lab. The process of brain storming ideas with them really helped me broaden my horizons and the way in which I think about problems. I value all the interactions I had with my peers. They have helped in my intellectual growth tremendously.

I would like to thank my close friends Sheeban Wasi, Akshay Ghai and Abhiraj Kulaar. They brought levity to my journey through this program with their friendship and support at times when the journey was stressful. They always encouraged me to pursue excellence in my chosen field and I am grateful to them for holding me to a higher standard.

Finally, I would like to thank my parents Arun Kumar Jain and Neena Singla, and my sisters Nikita Jain and Malvika Jain. Without their love and unwavering support, I would not have been able to move to a new country to pursue my dreams. I have been able to successfully complete this degree only because they supported me at times when I needed their support the most.

Contents

List of Figures	ix
List of Tables	xii
1 Introduction	1
1.1 AI for Software Engineering	1
1.2 Software Engineering Tasks	3
1.3 Existing Methods	5
1.4 Datasets for Software Engineering Tasks	6
1.5 Definitions	9
1.6 Main Contributions	10
2 <i>XLCoST</i> Dataset Construction	13
2.1 Data Source	13
2.2 Data Collection	15
2.3 Data Alignment	16
2.4 Data Splitting	21
2.5 Data Characteristics	23
3 Experiments with <i>XLCoST</i>	25
3.1 Task Setting	25

3.1.1	Generation Tasks	25
3.1.2	Retrieval Tasks	26
3.2	Evaluation Metrics	28
3.3	Baselines	29
3.4	Results and Discussion	30
3.4.1	Effect of Sequence-to-Sequence Pretraining	30
3.4.2	Effect of Pretraining on Specific Languages	30
3.4.3	Performance on Low-Resource Languages	30
3.4.4	Comparison between BLEU and CodeBLEU	31
3.4.5	Effect of Transfer Learning from Snippet-level Training	34
4	Analysis of Compilation Rate	35
4.1	Compilation Rate for Generative Tasks	35
4.2	Top Compilation Errors in Generated Programs	37
4.3	Ablation studies with out-of-distribution (OOD) data	37
4.3.1	Comparison against author baselines	38
4.3.2	Snippet-Transfer for OOD data	38
4.3.3	Ablation study with LeetCode fine-tuning	40
4.4	Qualitative analysis of generated hypotheses	40
4.5	Challenges with evaluating execution based metrics	41
5	Reinforcement Learning to Improve Translation Quality	48
5.1	Background	48
5.2	RL4CNCT	48
5.2.1	Problem Formulation	49
5.2.2	Reward Function	51
5.2.3	Loss Function	53

5.3	Dataset	54
5.4	Baseline	55
5.5	Results	55
5.6	Qualitative Analysis	55
6	Public demo website	59
6.1	The Django framework	59
6.2	Overall site design	60
6.3	Functional details	60
6.4	Challenges and learnings	62
7	Conclusion	67
	Bibliography	69

List of Figures

2.1	A breakdown of the problem sets included in the data and their organization on the GeeksForGeeks portal. Only three sub-categories per category is shown here for purposes of brevity.	14
2.2	An illustration of the data and the tasks. The first column is the Problem Description; each cell in the second column is a Comment; each cell from the third column is a code Snippet. The combination of all the code snippets in a column is a Program (truncated due to space limitation). The arrows show the input and output data for each task. Solid lines are for generation tasks and dashed lines are for retrieval tasks. Note that the Program Synthesis task uses both Problem Description and Comments as input.	24
4.1	Sample LeetCode raw metadata. The metadata contains problem description, example tests and constraints.	45
4.2	A single example from the processed test case file. Each line item in the file corresponds to in problem ID and contains it's corresponding test cases parsed from the metadata.	46

5.1	An overview of the RL4CNCT method. Reading from left to right, we begin with source code sequence x , a policy pre-trained model and a reference pre-trained model. Both models begin with the same set of weights. The policy model generates a translated code sequence $\hat{y}$. a) The translated code sequence is passed through the compiler to determine the compilation reward signal, +1 for successful compilation, -1 for unsuccessful compilation. b) Abstract Syntax Tree match score is calculated between the generated sequence $\hat{y}$ and the ground truth sequence y . c) Data Flow Graph match score is calculated between the generated sequence $\hat{y}$ and the ground truth sequence y . d) KL-Divergence between the output distribution of the actor model (policy) and the critic model (reference) is minimised so that the policy does not stray too far away from the reference. Signals from a), b), c) and d) are combined into a reward and used in the PPO loss function to fine-tune the pre-trained policy model.	49
6.1	Home page of the Translation demo website. This page is displayed when a user first visits the website.	60
6.2	Drop down menus to choose source and target languages. The user can select between 42 language pairs to translate based on the source and target language choice.	61
6.3	Loading source code through pre-existing examples. Clicking on any of the buttons marked red loads a pre-set example code in the source language selected. Users can also type their own input code into the source code text box.	62
6.4	Clicking the translate button activates the model which generates code in selected target language. The generated translation is displayed in the translated code text box.	63
6.5	The generated target language code. The figure shows the Python output generated by the model when C++ is selected as the source language and an example code is provided as input.	64
6.6	A detailed workflow of demo website that shows how the back end works when the TRANSLATE button is clicked. The input code string is parsed from the front-end text box and relayed to the model using an AJAX request. The model processes the input and the generated output is relayed back to the front-end target code text box.	65

6.7	After the translated code is generated, the user can also check of the generated code compiles by clicking on the compile button. The target language compiler is automatically selected, which then checks for syntax errors in the generated code. Errors, if they exist, are displayed in compilation output text box. If no errors exist, a success message is displayed.	66
6.8	The compilation output text box indicates that the model generated translation compiled successfully.	66

List of Tables

1.1	Comparison of <i>XLCoST</i> against other parallel code datasets in terms of alignment, supported tasks, size and supported languages (Py - Python, JS - JavaScript). Column “Size” refers to the number of parallel data pairs. *This number is for single programs, not pairs.	9
1.2	An overview of the tasks. All the tasks have pairwise data at both snippet-level and program-level in 7 programming languages, C++, Java, Python, C#, Javascript, PHP, and C. The tasks can be divided into two categories, generation and retrieval. The generation tasks include Code Translation, Code Summarization and Code Synthesis; the retrieval tasks include NL (natural language) Code Search and XL (Cross-Lingual) Code Search.	11
2.1	Example of the parallel alignment of the code data in four languages (C++, Java, Python and PHP). The programs given here check if a given number is divisible by 3 or not.	17
2.2	Example of the parallel alignment of the code data in four languages (C#, Javascript, PHP and C). The programs given here aim to find the LCM of two given numbers.	18
2.3	Example of the parallel alignment of the code data in all seven languages. The Programs given here aim to find two elements whose sum is closest to zero.	19
2.4	Number of pairwise code-code data in training, validation, and testing splits for each language-pair. The upper triangle (in bold font) shows the number of parallel code snippets, and the lower triangle shows the number of parallel programs. This data is used for the Code Translation and XL Code Search tasks. (Py is short for Python. JS is short for Javascript.)	22

2.5	Number of pairwise code-text data in each language. “Snippet” means snippet-comment pairs, and “Program” means program-description (problem description) pairs. This data is used for Code Summarization (Code-to-Text), Code Synthesis (Text-to-Code) and NL Code Search tasks.	22
2.6	The train-valid-test split and basic statistics of <i>XLCoST</i> data. SN - Snippets; PR - Program.	23
3.1	From top to bottom, the table contains results for Code Translation, Code Synthesis, Code Summarization, and Code Search at the snippet-level and program-level. CodeBLEU scores are reported for Code Generation tasks (Translation and Synthesis). For Translation, the language column on the left represents the source language and the row on the top represents the target language. BLEU scores are reported for Summarization and MRR for Search.	32
3.2	BLEU scores for two tasks (Translation and Code Synthesis) separated by a double horizontal rule. Above the double rule are presented BLEU scores of the Translation task for the 42 programming language pairs in the <i>XLCoST</i> dataset. Below the double rule are the BLEU scores for the Code Synthesis task. Column headers represent target languages. For Translation, row headers represent source languages.	33
3.3	Transfer learning from Snippet-Level training for Program Translation task on low resource language C. ST - Snippet Transfer.	34
4.1	Compilation rate of the generated programs in Program Synthesis task. . . .	36
4.2	Compilation rate of the generated programs in Program Translation task. For Python (Py) the score without the parenthesis is calculated using all the generated hypotheses, including the singular “\n” token. The score within the parenthesis is calculated excluding all the hypotheses where only the singular “\n” token is generated.	36
4.3	Top compilation errors for Code Translation in each target language (Javascript not included).	37
4.4	Leetcode data statistics.	38
4.5	A comparison of compilation rate for Java-C# (LeetCode) translation between the author’s fine-tuned models, CoST fine-tuned models and <i>XLCoST</i> fine-tuned models.	38

4.6	Program Translation compilation rate for OOD Leetcode data. ST stands for Snippet Transfer . Wavy-lines represent the best performing PLBART models. Bold values represent the best performing CodeT5 models.	39
4.7	Comparison of compilation rate on fine-tuning with LeetCode data, fine-tuning with <i>XLCoST</i> data and fine-tuning with both data sets. LC stands for LeetCode . Wavy-lines represent best performing PLBART models. Bold values represent best performing CodeT5 models.	41
4.8	Qualitative example for C++-Python translation using PLBART. i) PLBART+LC generates code with extra “_ :” tokens. ii) PLBART+ <i>XLCoST</i> generates the whole function again inside main with a return outside of any function definition. iii) PLBART + <i>XLCoST</i> + LC generates code without compilation errors.	42
4.9	Qualitative example for C++-Java translation using CodeT5. i) CodeT5+LC generates code with multiple class definitions of the same name. ii) CodeT5+ <i>XLCoST</i> generates code that used undefined variables in the function call. iii) CodeT5 + <i>XLCoST</i> + LC generates code without compilation errors.	43
4.10	Table showing a sample C++ program from the LeetCode data and the generated Java and Python translations using the PLBART model.	47
5.1	<i>XLCoST</i> dataset statistics after filtering for compilable programs.	54
5.2	Comparison of CodeT5 model trained on <i>XLCoST</i> without RL and the RL4CNCT model. Bold values represent the best performing models. C++, Python and C# are languages with larger number of data points as compared to C.	56
5.3	Qualitative comparison for Java-C++ translation between CodeT5 and RL4CNCT. i) CodeT5 generates a non-standard data type subset without defining it or importing it. RL4CNCT generates the correct datatype. ii) CodeT5 generetes the wrong tokens to return out of a function, RL4CNCT is able to generate the correct C++ syntax.	57
5.4	Qualitative comparison for C++-Python translation between CodeT5 and RL4CNCT. i) CodeT5 generates an extra closing parenthesis. ii) CodeT5 generates the wrong syntax for function declaration, RL4CNCT is able to generate the correct Python syntax using def keyword.	58

Chapter 1

Introduction

The development of the Transformer model [1] fundamentally changed the landscape of neural machine translation and the world of NLP and Deep Learning. The transformer architecture became the giant upon whose shoulders models like BERT [2], BART [3], RoBERTa [4] and GPT-3 [5] came to stand and pushed the state of the art for Natural Language Processing.

A similar landscape shift is now being observed in the programming language domain using these models and applying them to the software engineering paradigm. With the advent of open-source code repository websites like Github, a huge amount of source code data has become available which in turn has sparked an interest in the research community towards leveraging this data to solve a wide variety of software engineering tasks [6, 7].

1.1 AI for Software Engineering

With all of the research effort that is being put into this field, it is delightful to see how this research is beginning to reach its intended consumers. Most notably, a tool that has made a stellar breakthrough from the world of ML research to the common consciousness of the software development community is Copilot. Open AI in partnership with Github launched Copilot which is advertised as an NLP based “AI pair programmer” that can generate full code on the fly based on natural language prompts or code completions from incomplete pieces of code. It is capable of generating multiple candidate solutions for the same prompt that developers can choose from based on their needs. Since its launch the scientific community has actively probed and engaged with it to ascertain its usefulness to software engineers in terms of productivity. Ziegler et al. [8] investigated ways of connecting the productivity

benefit of GitHub Copilot to usage measurements from developer activity and concluded that neural code completion systems have the potential to hugely improve developer productivity since they are able to incorporate contextual information about the developers current activities and can generate responses tailored to them. They also concluded that even if developers don't fully accept the suggestions provided by Copilot, the multiple suggestions serve as good starting points to start building solutions on their own. This may be hard to quantify directly in terms of productivity or the amount of time saved, but in general the option to have good starting points has been received very positively by developers.

To better understand the developers' response to the application modernization process of translating source code from one language to another, Weisz et al. [9] spoke with 11 software developers. A common theme that emerged throughout their interviews was that the initial framework or scaffolding that the Neural Machine Translation (NMT) model provides would be a good starting place that the developers could then probe, correct and modify according to their requirements.

In a similar setting, Vaithilingam et al. [10] present a user study with 24 participants on the usability of GitHub Copilot. They found that 19 out of 24 participants expressed a strong preference to use Copilot for their day-to-day programming tasks even though the users reported a variety of difficulties they experienced while using it. One of the criticisms that a user provided was that, since Copilot provides a whole block of code as a solution to the posed problem it sometimes becomes difficult for the user to understand the generated code. The authors of the study suggest that a possible solution to this problem can be providing inline comments in the generated code in order for the users to be able to understand it better. For a harder problem, one of the users found that breaking the problem down into simpler sub tasks and writing prompts for the sub tasks helped Copilot generate better solutions as compared to generating the solution for the full complex task. Even with these criticisms, the authors observed that the participants had an overwhelming preference towards using Copilot since it provided a good starting point to approach the programming task.

Barenkamp et al. [11] presents a systematic meta-review of all the steps in the Software Development Life-Cycle (SDLC) in which AI can potentially be applied. During the project planning phase, AI based tools can be used to estimate project durations and costs while trying to optimize for both [12]. Ascertaining developer bandwidth and allocating tasks based on workload and project deadlines is yet another area in which AI tools can provide assistance [13]. They have discussed many tasks like semantic code search, code translation, creating code documentation etc. which can be aided by AI during the implementation phase of the SDLC. At the testing stage, automated bug detection as well as code repair tools can be employed [14, 15, 16].

In general, deep learning based programming assistance within the SDLC seems to have been received positively by the software engineering community. However much more research still needs to be carried out to address all the concerns that engineers have with fully trusting these systems.

1.2 Software Engineering Tasks

In the previous section we discussed at a high level how AI can be applied to a variety of software engineering tasks. We looked at studies and surveys that have been conducted to evaluate the usefulness of these tools to software engineers as well. In the following section we will expand upon some of the software engineering tasks that this work deals with in particular. These tasks include code translation, code summarization, code synthesis and code search.

Code Translation (Code-to-Code): Code Translation is the task of converting source code from one programming language to another. The issue of code translation becomes most pertinent in scenarios where code bases written in obsolete or deprecated languages need to be modernised. The task requires employing personnel who possess expertise in multiple coding languages and even then it is a time consuming process to perform. Efficient and accurate automated code translation becomes highly valuable in such scenarios. Nguyen et al. [17] present a phrase based statistical machine translation model and demonstrate its capabilities by migrating 2 Java projects into C#. Aggarwal et al. [18] present a statistical machine translation model to convert Python 2 code to Python 3. These are some of the very first works proposed in the application of machine translation for programming languages.

Code Summarization (Code-to-Text): Generating a readable summary that describes the functionality of a program is known as source code summarization [19]. This task is usually performed manually by programmers during the process of development when they add descriptive comments for pieces of code they write or at the end of the development effort when documentation is created. Regardless of when this exercise is performed, it remains an important part of the software development process in order to ensure that people who are unfamiliar with a code base are able to get up to speed quickly. Given that a person with significant familiarity with the code base and experience is required to write up such documentation, the value of being able to generate it automatically can be profound. Wan et al. [20] introduces an actor-critic based reinforcement learning approach for the code summarization task which leverages the abstract syntax tree structure in addition to the code sequences as input. Iyer et al. [21] present an LSTM based neural network called Code-

NN which is capable of generating summaries for C# and SQL code. Ahmad et al. [22] leverage a Transformer [1] to generate summaries for Java and Python codes.

Code Synthesis (Text-to-Code): Code Synthesis refers to the task of generating code using natural language descriptions of functionality as input. We already saw in the previous section, how multiple user studies suggest that even seasoned programmers welcome the prospect of being able to use initial machine suggested code as scaffolding to build more complex logic that fits their needs. In our estimation, this task is especially useful for amateur programmers or people who are just starting out with learning how to program. Having limited knowledge and context about programming languages, beginners might not know how to go about writing code to accomplish a certain task. However, if they can provide a natural language prompt that explains their intent and code can be automatically generated, it would help close that knowledge gap very quickly and this could be used a great educational tool. Zhu et al. [23] demonstrate a Generative Adversarial Network based approach to capture natural language utterances and generate code that is consistent with the utterance. Sun et al. [24] present a Convolutional Neural Network based approach that generates a program by predicting the grammar rules of the programming language.

Code Search: Code search usually refers to using a retrieval framework to rank order and retrieve the most relevant pieces of code from a repository based on a given input query. The input query is usually expressed in natural language that tries to capture the objective that the user wants to accomplish using the code. Some existing portals¹ try to solve this problem, however, the existence of large code repositories like GitHub clearly demonstrate the need for having an effective search methodology that specifically caters to code. Stolee et al. [25] conducted a survey with undergraduate students and concluded that among the survey participants who programmed daily or weekly, 85% searched for code at least weekly. Stolee et al. [25] reported various reasons why students search for code including reusing it, modifying it or just using it to get ideas for their own implementations. Gu et al. [26] introduced the first deep learning based method for semantic code search. Their method jointly embedded natural language and programming language in a high dimensional vector space and minimised the distance between code snippets and their corresponding natural language descriptions. Wan et al. [27] present a multi-modal attention network that leverages the sequential tokens of code, it's abstract syntax tree and the control flow graph to create a comprehensive multi-modal representation of the code. They demonstrate their multi-modal approach to be highly effective at the code retrieval task.

¹<https://stackoverflow.com>

1.3 Existing Methods

In the last 2 to 3 years, research interest in solving these tasks has peaked. Researchers have proposed a myriad of deep learning based approaches. Introduced by Feng et al. [28], CodeBERT is a bimodal pre-trained model for programming language and natural language. It is based on transformer style architecture trained using a hybrid objective function including masked language modelling and replaced token detection. The hybrid objective function allows leveraging of both unimodal as well as bimodal data, where bimodal data provides tokens for model training and unimodal data helps learn better generators. The authors show that CodeBERT is effective in code search and code to text generation tasks. GraphCodeBERT [29], instead of using syntax level structure of code in the form of AST's, uses semantic level information in the form of data flow graphs. This captures the information of where the value comes from between variables while avoiding the complexity and deep heirarchy of AST's. The authors evaluate the method on 4 downstream tasks of code search, code refinement, translation and clone detection and show that it achieves state of the art performance on all 4 tasks. Code transformer [30] uses language agnostic features computed from the source code and its abstract syntax tree to accomplish the task of code summarization by predicting the method name, given a code snippet. The authors argue that leveraging both context information computed from the source code as a sequence of tokens and structural features obtained from the AST leads to a much more powerful representation of the code when compared to using only one of them. They also show that their language agnostic approach enables them to train multilingual models and performance on all languages is greatly enhanced by a training model on many programming languages.

DOBF [31] proposes a code specific objective function to better pre-train models designed to be fine tuned for code generation tasks. The authors argue that objectives like masked language modelling are easier to optimise for code generation tasks, since programming languages by nature contain multiple instances of tokens at various locations. It will be easy for the model to just copy a variable's name from another occurrence if it occurs more than once in a function. A deobfuscation objective however, masks all occurences of a given variable, thereby making it a much harder problem to solve for the model by significantly amplifying the training signal.

In code2seq [32], the authors try to improve on performance of models adopted from neural machine translation for the purposes of code summarisation, documentation and retrieval. They propose a code2seq approach that leverages the syntactic structure of programming languages using abstract syntax trees to better encode source code. Code2vec [33] represents code snippets as continuous distributed vectors. Once a snippet can be represented as a vector embedding, these embeddings can be used to predict semantic properties of the

snippet. The embeddings are learned by leveraging paths in the abstract syntax tree of the code snippet and this approach is tested by using the embeddings to predict suitable function names for the code snippet.

BERT [2] and GPT [5] are encoder-only or decoder-only style architectures. The authors argue that this arrangement is suboptimal and present CodeT5 [34] a combined encoder-decoder Transformer model that has been pre-trained to make greater use of the code semantics indicated by the developer-assigned identifiers. In PLBART, Ahmad et al. [35] present a sequence-to-sequence model pre-trained on a large corpus of Java and Python functions using a denoising objective. The authors argue that their approach allows the model to reason about language syntax and semantics and therefore generate coherent code.

Most of the works we have reviewed so far employ some sort of encoder-decoder framework where the decoder relies on the ground truth to predict the next word in the sequence. Wan et al. [20] argue that this setting is suboptimal since at test time the model is expected to predict the sequence without the help of the ground truth which can lead to exposure bias. Wan et al. [20] propose an actor-critic based reinforcement learning(RL) framework for the code summarization task. In their method, instead of maximising the likelihood of the next word in the sequence given the current generated sequence and ground truth, they use the actor to generate the sequence and employ the critic to directly optimise the BLEU score of the generated sequence. Le et al. [36] introduce an actor-critic based RL framework for the code synthesis task. Le et al. [36] incorporate test case based execution level feedback in their reinforcement learning algorithm to encourage functional correctness in the generated code sequences. They employ their previously released model, CodeT5 [34], with modifications to the pre-training objective as well as pre-training data, as a stochastic policy for code generation and use another classifier model as a critic to predict the probability of the generated sequence passing test cases. Feedback from the critic model as well as from actual execution of test cases is incorporated into the reward function to train the actor.

1.4 Datasets for Software Engineering Tasks

None of the innovation in model architectures and algorithms discussed in the previous section has been happening decoupled from data sources. A model is only as good as the data it is trained on. A variety of benchmark data sets have now become available to the community to help researchers solve these complex problems. Introduced in 2021 by researchers from Microsoft Research Asia, Developer Division, and Bing, CodeXGLUE [37] has now become a popular benchmark dataset in the community to evaluate code intelligence models. CodeXGLUE which stands for General Language Understanding Evaluation benchmark

for Code, consists of 14 constituent datasets that are each capable of supporting a variety of different task and programming languages. These tasks include code-to-code tasks like clone detection, defect detection, cloze test, code completion, code repair, and code-to-code translation; text-to-code tasks like natural language code search, text-to-code generation; code-to-text tasks like code summarization and text-to-text tasks like document summarization. CodeXGLUE has quickly become the benchmark data set of choice among researchers, with a large number of researchers choosing to evaluate their models using this data set.

In 2019, GitHub and Microsoft Research-Cambridge released CodeSearchNet [38] in collaboration, in an effort to encourage research into the domain of semantic search for code. CodeSearchNet consists of 2 million docstring-function pairs scraped from publicly available, open-source, non-fork GitHub repositories that span over languages like Python, Javascript, Ruby, Go, Java, and PHP. The idea is to use docstrings which are top-level function or method comments as queries to retrieve the relevant code. This dataset is now widely used as an evaluation benchmark for code search tasks. They also started the CodeSearchNet challenge in which researchers who used their data set could compete with each other and attempt to outperform each other.

Introduced by Puri et al. [39], Project CodeNet released a data set curated from two online judge websites AIZU² and AtCoder³ containing 14 million code samples, each of which is an intended solution to one of 4000 coding problems. CodeNet contains code samples written in over 50 different languages, although C++, C, Python, and Java end up forming a significant majority of the proportion of problems. An interesting feature of the CodeNet dataset is that in addition to code it also provides very rich metadata associated with each code sample which includes code size, memory footprint, CPU run time and acceptance and error statuses. When possible they also attempt to provide problem descriptions and input and output pairs. The rich information available in the CodeNet opens up the possibility for researchers to conduct a whole host of different experiments with it.

Ahmad et al. [40] proposed the AVATAR corpus containing solutions for 8,475 problems in Java and Python collected from competitive programming sites, online platforms, and open source repositories. Specifically, these sources include AtCoder, Google Code Jam, Codeforces, and online platforms like GeeksforGeeks, LeetCode, Project Euler. AVATAR specifically targets the code translation between Java and Python, presenting parallel code data at the program level as opposed to the usual function level. Zhu et al. [41] introduced CoST, a code translation data set containing parallel data from 7 commonly used programming languages. The data in CoST is parallel at the snippet as well as the program level, snippets referring to small self-contained chunks of code within a program which makes it

²<https://onlinejudge.u-aizu.ac.jp>

³<https://atcoder.jp>

very effective for code translation tasks. Table 1.1 summarises these data sets and presents some of their main characteristics and differences.

The domain of deep learning for software engineering has certainly benefited from these benchmarks and data sets, however they do not come without drawbacks of their own. CodeXGLUE [37] supports 10 different tasks and includes a huge number of data points, however each task only supports certain languages. Translation for example only supports Java and C#, Natural Language Code Search only supports Python, Text to code generation only supports Java. CodeSearchNet [38] is indeed a large data set supporting many languages however only the single task of semantic code search. CodeNet is similarly large and multilingual but consists of user submitted code for online code challenges. Since the codes are contributed by different users all around the world without a unified format, style or structure, they are of high variance even for the same problem. Because of this, the alignment quality of code is low in CodeNet. AVATAR also supports translation between Java and Python only. Whenever researchers who use these data sets release their benchmarks and comparisons, they usually have to contend with these limitations of specific languages and tasks as well. Popular works like CodeT5 [34] and PLBART [35] only release benchmark models and scores for Java to C# and C# to Java translation since they use the CodeXGLUE data set. This severely limits researchers from being able to perform thorough evaluations and presenting comparative analyses across a wide variety of settings. We will attempt to tackle some of these challenges through the work presented in this thesis.

The intent of this work is to address some of the challenges we discussed related to the availability of data and also challenges with respect to the performance evaluation of code generation models. In this work we curated a new benchmark data set named *XLCoST* which stands for **Cross-Lingual Code Snippet** data set. This data set contains fine-grained parallel data from 7 popular programming languages. In addition to programming languages, the data set also contains problem descriptions and code comments in English. This allows fine grained alignment at both the program and snippet level. We will formally define what we mean by a program and a snippet in the subsequent sections. One of the major challenges we have observed with data sets that are currently available is that they either only support certain languages or certain tasks. *XLCoST* aims to alleviate this limitation by providing support for 10 different code related tasks in all 7 languages. It is our hope that this data set can become a solid benchmark that more research can evaluate itself against without being hampered by the availability of limited languages. In addition to the data set, we also present a reinforcement learning based method to improve the quality of one specific software engineering task, code translation. We leverage the data set we have collected and employ it in combination with reinforcement learning method to see if gains in generating more syntactically correct code can be achieved.

Table 1.1: Comparison of *XLCoST* against other parallel code datasets in terms of alignment, supported tasks, size and supported languages (Py - Python, JS - JavaScript). Column “Size” refers to the number of parallel data pairs. *This number is for single programs, not pairs.

Dataset	Alignment	Task	Labelling	Size	Languages
CodeNet	Program	Multiple	Solutions to the same problem	13.9M*	55 programming languages
AVATAR	Program	Translation	Solutions to the same problem	57,414	Java, Py
CodeXGLUE	Method	Multiple	Matching function names	11,800	Java, C#
CoST	Snippet	Translation	Matching code comments	132,046	C++, Java, Py, C#, JS, PHP, C
XLCoST	Snippet	Multiple	Matching code comments	1,002,296	C++, Java, Py, C#, JS, PHP, C, English

1.5 Definitions

We have discussed at length the challenges and limitations that machine learning researchers face when working on problems pertaining to the coding domain. Before going into the details of our proposed solutions, empirical analysis and our contributions, we would like to formally establish some definitions that will be repeatedly used throughout this work for consistency.⁴

Problems: A problem is best thought of as a question that deals with concepts of implementing a data structure or an algorithm. These are designed for tutoring and coding interview preparation and can be widely found on popular tutoring websites^{5,6}.

Programs: Whenever we talk about a program we refer to the full code solution of a given problem in a specific programming language. Each problem in this data set may contain up to 7 programs (one for each language). The programs for the same problem share similar logic and structure.

Snippets: The code between two consecutive comments in a program is termed as a snippet (code before the first comment and after the last comment are also included). On an average, each program contains 8.81 snippets.

Description: Each problem also has a short natural language description, for example, “Maximum Consecutive Increasing Path Length in Binary Tree.”

Comments: The code comments describing snippets in each program in this data set are referred to as comments in this work. The programs are well commented and each program has an average of about 9 comments.

Compilability: Throughout this work, whenever we talk about compilability, we refer to the qualitative aspect of a piece of code being able to compile successfully without throwing any errors when it is put through its language specific compiler.

⁴This section, with modifications is adapted from our arXived work: Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, Chandan K. Reddy “XLCoST: A Benchmark Dataset for Cross-lingual Code Intelligence”.

⁵<https://leetcode.com>

⁶<https://www.geeksforgeeks.org>

1.6 Main Contributions

This work presents *XLCoST* [42], a machine learning benchmark data set that contains fine-grained, parallel data in 7 commonly used programming languages (C++, Java, Python, C#, Javascript, PHP, C), and natural language (English). The data is parallel across 7 languages, at both code snippet level and program level. This means that, given a program in one language, the data set contains the same program in up to 6 other programming languages. Each program is divided into several code snippets, and programs in all the languages are aligned at the snippet level. Moreover, each of the snippets is accompanied with a comment, and the comment for a particular snippet is the same across all the languages. Table 1.1 presents a comparative analysis of *XLCoST* in terms of the number of available parallel data samples against other widely used parallel code datasets. We see that when compared against *XLCoST*, specifically for availability of aligned parallel data, most of these data sets only support selective languages. *XLCoST* however, contains around 1 million parallel snippets and 123K parallel programs in total, which is significantly larger than many available parallel code data sets. We believe that this data set is a valuable asset for the research community and can potentially benefit a number of code-related research problems.

To further facilitate the development and evaluation of models with a focus on source code, we also introduce 10 different cross-lingual tasks. These tasks can be divided into two categories: Generation and Retrieval. The generation tasks include Code Translation (Code-to-Code), Code Summarization (Code-to-Text), and Code Synthesis (Text-to-Code); the retrieval tasks include NL (Natural Language) Code Search and XL (Cross-Lingual) Code Search. Each task is at both snippet and program level. To evaluate how challenging the tasks are with the proposed dataset, we run experiments on all the 10 tasks with a number of state-of-the-art baseline models. We also conduct an empirical study to understand how the model design relates with the performance on different tasks with *XLCoST* data set. The data set, as well as the code for our experiments can be found in our GitHub repository⁷. We argue for the importance of using better metrics in order to evaluate the effectiveness of deep learning methods in generating usable code. To that end we define a metric called “Compilation Rate” of code, present a study of the compilability of the code generated using the methods discussed in this work and show comparisons how different methods vary significantly in their ability to generate actually usable code. We curate another parallel data set from the popular interview preparation website, LeetCode and to empirically demonstrate the value provided by *XLCoST* in generating better quality code for out of distribution data.

Based on the results of our analysis of compilability of model outputs, we identify some weaknesses and propose a reinforcement learning based model to potentially improve upon

⁷<https://github.com/reddy-lab-code-research/XLCoST>

Table 1.2: An overview of the tasks. All the tasks have pairwise data at both snippet-level and program-level in 7 programming languages, C++, Java, Python, C#, Javascript, PHP, and C. The tasks can be divided into two categories, generation and retrieval. The generation tasks include Code Translation, Code Summarization and Code Synthesis; the retrieval tasks include NL (natural language) Code Search and XL (Cross-Lingual) Code Search.

Category		Task	Data	Description
Generation	Code Translation (Code-to-Code)	Snippet Translation	872K/47K/83K	Translate code snippet across programming languages
		Program Translation	106K/6K/11K	Translate program across programming languages
	Code Summarization (Code-to-Text)	Snippet Summarization	446K/22K/41K	Generate comment for given code snippet
		Program Summarization	50K/3K/5K	Generate problem description for given program
	Code Synthesis (Text-to-Code)	Snippet Synthesis	446K/22K/41K	Generate code snippet giving comment
		Program Synthesis	50K/3K/5K	Generate program giving problem description and comments
Retrieval	NL Code Search	Comment-to-Snippet Search	446K/22K/41K	Retrieve code snippet for given comment
		Problem-to-Program Search	50K/3K/5K	Retrieve program for given problem description
	XL Code Search	Snippet-to-Snippet Search	872K/47K/83K	Retrieve code snippets in other languages for given snippet
		Program-to-Program Search	106K/6K/11K	Retrieve programs in other languages for given snippet

these short comings. We incorporate compiler feedback, as well as semantic and syntax level information in the form of abstract syntax trees and data flow graphs in our reinforcement learning reward function to encourage the model to make fewer syntactic errors. We present the details of our method and empirical results of our experiments as well as discuss possible future directions we can take with this approach.

Finally, we also develop and present a web portal that hosts the code translation models we developed, in order to enable to community to interact with our work. The portal allows users to enter their own code or use example code to generate translations in the 7 languages supported by *XLCoST*, as well as to evaluate whether the generated translation compiles or not.

The remainder of this document is organised in the following manner: In Chapter 2, we dive into the details of how *XLCoST* was curated, where the data was sourced from, how it was processed and cleaned and it’s fundamental characteristics. In Chapter 3, we describe in detail all of the different tasks that *XLCoST* supports, we talk about the evaluation metrics employed to evaluate each task and present a comparative analysis of the performance of multiple baselines on each of the tasks using *XLCoST*. In Chapter 4, we present a study of the quality of code generated using some of the baseline methods discussed in Chapter 3 from the perspective of it’s compilability. We focus on the translation task and evaluate if

using snippet-level aligned data can help generate code with less frequent compilation errors. In Chapter 5, we present our reinforcement learning method to improve the code generation quality. We describe the details of the model design, the experiments we performed and the results we obtain. In Chapter 6, we present the web portal we developed to host some of the models we developed in this work, we discuss the technical details of its implementation, its functionalities and some of the challenges we faced during its development. Finally, in Chapter 7, we provide our concluding thoughts.

Chapter 2

XLCoST Dataset Construction

In this chapter we will discuss in great detail, the full process of how we created the *XLCoST* [42] data set. We will discuss details about the data source in section 2.1 and how the data is collected in section 2.2. We discuss the processing and alignment in section 2.3. In section 2.4, we talk about how the train/test/val splits are created and finally, we will highlight some qualitative and quantitative characteristics of the data in section 2.5.¹

2.1 Data Source

The data has been sourced from different subpages of the extremely popular programming education portal GeeksForGeeks². There are a few major reasons why we chose to use GeeksForGeeks as our data source.

Firstly, it houses thousands of data structures and algorithm problems along with solutions in up to 7 different programming languages - C++, Java, Python, C#, Javascript, PHP, and C. A majority of the problems on this site fall under two categories - Data Structures and Algorithms. These two categories have different sub-categories within them. For example, the Data Structures page has the hyperlinked sub-categories of Array, Linked Lists, Stack, Queue, etc. which when clicked direct the user to all the problems relating to that specific sub-category and their corresponding solutions in different programming languages. The same goes for the Algorithms page, where examples of some of the sub-categories are Greedy

¹*This chapter, with modifications is adapted from our arXived work: Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, Chandan K. Reddy “XLCoST: A Benchmark Dataset for Cross-lingual Code Intelligence”.*

²<https://www.geeksforgeeks.org>

algorithms, Divide and Conquer, Graph Algorithms, and so on. Figure 2.1 shows how the problems are organized.

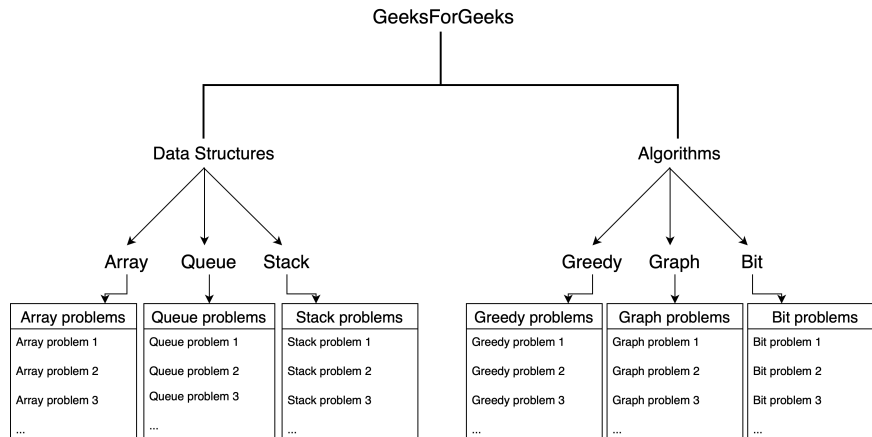


Figure 2.1: A breakdown of the problem sets included in the data and their organization on the GeeksForGeeks portal. Only three sub-categories per category is shown here for purposes of brevity.

Secondly, the solution programs for the same problem follow the same structure, down to the variable names. Although submissions to the GeeksforGeeks portal are user generated content, the content that ultimately gets published on the website is heavily moderated. Users need to follow a strict set of guidelines³ against which the submission is evaluated and only then published. The detailed guidelines clearly state that all submitted solutions must follow the same program structure and variable names. The example quotes: “Variables used in the code, program structure, and output should match with the existing code; Variable names and output of newly added code (for example Java) should be the same as existing code (For example C++).” are directly from the guidelines.

This strict moderation results in the programs being semantically consistent across the different languages. In most cases, the programs for the same problem share the same set of comments in the same order. The consistency between code-comment pairs across languages indicates that they are parallel at the code block level which, helps us break each program into smaller code blocks or ‘Snippets’. This is where the fine-grained alignment at the snippet-level and at the program-level in *XLCoST* comes from which can further be used for a multitude of code-related downstream tasks.

³<https://write.geeksforgeeks.org/how-to-write/>

2.2 Data Collection

For scraping the data, we use python scripts and some external libraries, the most important of them being BeautifulSoup⁴ and Selenium⁵. BeautifulSoup4 is a python library that facilitates the acquisition of data from HTML and XML files. Selenium is a python package that, in our case, is used to automate web browser functions to navigate through the GeeksForGeeks page directories. Each page in GeeksForGeeks that houses a problem and its solution has a uniform HTML page structure. This structure allows us to extract specifically targeted sections which are needed for our dataset using BeautifulSoup4.

Using the directory structure of the GeekForGeeks website, the ability of Selenium to navigate through these pages and the utility of BeautifulSoup4 for content extraction from these pages, data is extracted from the website with practically no manual intervention. Every problem page can have one or more solutions to the same problem in different languages. For example, if there is a problem statement “Given a number n , print n -th Fibonacci Number”, there can be different logics to solve the same problem. One solution may use recursion-based logic, another can utilize dynamic programming and yet another can use a space complexity optimized method to solve the same problem. Each logic has code in different languages and code pertaining to each logic resides in separate sections of the page which can be identified via their HTML tags. We extract every possible problem statement and solution from the above-mentioned two categories and do not put any filter on what category, type, difficulty, etc. the solutions belong to at collection time.

After collecting the data, we first remove duplicate problems, as some problems might be presented in multiple subcategories. Then we extract problem description and solution programs in each available language from the page. Each program is sliced into code snippets by splitting at the comments, after which the comments and docstrings are removed from the programs. Any personal information such as the name of the code’s contributor, was removed from both the comments and the codes at this time. Eventually, we get 4 types of information from one page: 1) Problem Description; 2) Parallel programs in different languages; 3) Code Snippets; 4) Code Comments.

The IP policies and regulations for GeeksForGeeks are carefully followed and we confirm that no data privacy policy has been violated when collecting the data.

⁴<https://www.crummy.com/software/BeautifulSoup/>

⁵<https://www.selenium.dev/>

2.3 Data Alignment

The snippet-level alignment of the data is performed by matching comments in the solution programs (for the same problem) across different languages. As we mentioned earlier, Geeks-ForGeeks programs follow a standard template, because of which the comments in different language programs (for the same problem) align parallelly in most cases. This yields parallel snippets that have the same functionality across different languages.

Misalignment detection: In some cases, the comments in different solution programs are not aligned. The misalignment can come from different numbers of comments, and the differences in the comment content. This is usually due to some solution program not strictly following the guidelines and templates. For solution programs with the same number of comments, we evaluate the alignment by calculating the average similarity score of each pair of comments in the two programs (using Python *difflib.SequenceMatcher*⁶). If the average score is below a certain threshold (80% in our case), it would be categorized as misalignment and manual checking would be needed. Solution programs with different number of comments were automatically categorized as misaligned and marked for manual checking. The rationale behind selecting 80% as the threshold for comment misalignment is that since SequenceMatcher is a non-contextual similarity metric, a 80% matching score means that a significant portion of two given comments are exactly the same, which is a strong indicator that two code snippets (from different languages) are parallel to each other. Moreover, no matter whether a problem meets the 80% threshold or not, we review each of them to verify the alignment quality.

Out of the 11,265 problems, around 64% fell below the 80% threshold and needed manual alignment. Some of the misalignment cases are false-negatives, and we skipped those after reviewing them. For the rest of the misalignment cases, we grouped them into different categories by the difference in numbers of comments across different languages.

Manual checking and aligning: Based on the differences in number of comments, the misaligned programs were split into the following categories:

Category 0: The programs have the same number of comments. The type of misalignment is usually only due to different wording in the comments and can be easily fixed.

Category k : The difference in number of comments is k . When $k < 3$, extra comments needed to be discarded in some cases and code from these comments was moved to appropriate snippets to preserve the alignment with other languages. In some cases, there were also missing comments which had to be added along with the moving of the appropriate code block as in the previous case. When $k \geq 3$, the programs will be discarded. Tables 2.1, 2.2,

⁶<https://docs.python.org/3/library/difflib.html>

and 2.3 show examples of what the data looks like after this alignment has been performed.

Table 2.1: Example of the parallel alignment of the code data in four languages (C++, Java, Python and PHP). The programs given here check if a given number is divisible by 3 or not.

C++	Java	Python	PHP
<i>/* C++ program to find if a number is divisible by 3 or not */</i>	<i>/* Java program to find if a number is divisible by 3 or not */</i>	<i>''' Python program to find if a number is divisible by 3 or not '''</i>	<i>/* PHP program to find if a number is divisible by 3 or not */</i>
<code>#include<bits/stdc++.h> using namespace std;</code>	<code>class IsDivisible {</code>		<code><?php</code>
<i>/* Function to find that number divisible by 3 or not */</i>	<i>/* Function to find that number divisible by 3 or not */</i>	<i>''' Function to find that number divisible by 3 or not '''</i>	<i>/* Function to find that number divisible by 3 or not */</i>
<code>int check(string str) {</code>	<code>static boolean check(String str) {</code>	<code>def check(num) :</code>	<code>function check(\$str) {</code>
<i>/* Compute sum of digits */</i>	<i>/* Compute sum of digits */</i>	<i>''' Compute sum of digits '''</i>	<i>/* Compute sum of digits */</i>
<code>int n = str.length(); int digitSum = 0; for (int i=0; i<n; i++) digitSum += (str[i]-'0');</code>	<code>int n = str.length(); int digitSum = 0; for (int i=0; i<n; i++) digitSum += (str.charAt (i)-'0');</code>	<code>digitSum = 0 while num > 0 : rem = num % 10 digitSum = digitSum + rem num = num / 10</code>	<code>\$n = strlen(\$str); \$digitSum = 0; for (\$i = 0; \$i < \$n; \$i++) \$digitSum += (substr (\$str, \$i) - '0');</code>
<i>/* Check if sum of digits is divisible by 3 */</i>	<i>/* Check if sum of digits is divisible by 3 */</i>	<i>''' Check if sum of digits is divisible by 3 '''</i>	<i>/* Check if sum of digits is divisible by 3 */</i>
<code>return (digitSum % 3 == 0) ; }</code>	<code>return (digitSum % 3 == 0) ; }</code>	<code>return (digitSum % 3 == 0)</code>	<code>return (\$digitSum % 3 == 0); }</code>
<i>/* Driver code */</i>	<i>/* main function */</i>	<i>''' main function '''</i>	<i>/* Driver code */</i>
<code>int main() { string str = "1332"; check(str)? cout << "Yes" : cout << "No "; return 0; }</code>	<code>public static void main (String[] args) { String str = "1332"; if(check(str)) System.out.println("" Yes"); else System.out.println("No "); } }</code>	<code>num = 1332 if(check(num)) : print "Yes" else : print "No"</code>	<code>\$str = "1332"; \$x = check(\$str) ? "Yes" : "No "; echo(\$x); ></code>

Table 2.2: Example of the parallel alignment of the code data in four languages (C#, Javascript, PHP and C). The programs given here aim to find the LCM of two given numbers.

C#	JavaScript	PHP	C
<pre>/* C# program to find LCM of two numbers */ using System; class GFG {</pre>	<pre>/* Javascript program to find LCM of two numbers */</pre>	<pre>/* PHP program to find LCM of two numbers */ <?php</pre>	<pre>/*C program to find LCM of two numbers*/ #include <stdio.h></pre>
<pre>/* Recursive method to return gcd of a and b */ static int gcd(int a, int b) { if (a == 0) return b; return gcd(b % a, a); }</pre>	<pre>/* Recursive function to return gcd of a and b */ function gcd(a, b) { if (b == 0) return a; return gcd(b, a % b); }</pre>	<pre>/* Recursive function to return gcd of a and b */ function gcd(\$a, \$b) { if (\$a == 0) return \$b; return gcd(\$b % \$a, \$a) ; }</pre>	<pre>/* Recursive function to return gcd of a and b */ int gcd(int a, int b) { if (a == 0) return b; return gcd(b % a, a); }</pre>
<pre>/* method to return LCM of two numbers */ static int lcm(int a, int b) { return (a / gcd(a, b)) * b; }</pre>	<pre>/* Function to return LCM of two numbers */ function lcm(a, b) { return (a / gcd(a, b)) * b; }</pre>	<pre>/* Function to return LCM of two numbers */ function lcm(\$a, \$b) { return (\$a / gcd(\$a, \$b)) * \$b; }</pre>	<pre>/* Function to return LCM of two numbers */ int lcm(int a, int b) { return (a / gcd(a, b)) * b; }</pre>
<pre>/* Driver method */ public static void Main() { int a = 15, b = 20; Console.WriteLine("LCM of " + a + " and " + b + " is " + lcm(a, b)); }</pre>	<pre>/* Driver program to test above function */ let a = 15, b = 20; document.write("LCM of " + a + " and " + b + " is " + lcm(a, b));</pre>	<pre>/* Driver Code */ \$a = 15; \$b = 20; echo "LCM of ",\$a, " and " ,\$b, " is ", lcm(\$a, \$b); ?></pre>	<pre>/* Driver program to test above function */ int main() { int a = 15, b = 20; printf("LCM of %d and % d is %d ", a, b, lcm(a, b)); return 0; }</pre>

Table 2.3: Example of the parallel alignment of the code data in all seven languages. The Programs given here aim to find two elements whose sum is closest to zero.

C++	Java	Python	C#
<pre> /* C++ code to find Two elements whose sum is closest to zero */ # include <bits/stdc++.h> # include <stdlib.h> # include <math.h> using namespace std; void minAbsSumPair(int arr [], int arr_size) { int inv_count = 0; int l, r, min_sum, sum, min_l, min_r; </pre>	<pre> /* Java code to find Two elements whose sum is closest to zero */ import java.util.*; import java.lang.*; class Main { static void minAbsSumPair(int arr[], int arr_size) { int inv_count = 0; int l, r, min_sum, sum, min_l, min_r; </pre>	<pre> ''' Python3 code to find Two elements whose sum is closest to zero ''' def minAbsSumPair(arr, arr_size): inv_count = 0 </pre>	<pre> /* C# code to find Two elements whose sum is closest to zero */ using System; class GFG { static void minAbsSumPair(int []arr, int arr_size) { int l, r, min_sum, sum, min_l, min_r; </pre>
<pre> /* Array should have at least two elements */ if (arr_size < 2) { Console.Write("Invalid Input"); return; } </pre>	<pre> /* Array should have at least two elements */ if(arr_size < 2) { document.write("Invalid Input"); return; } </pre>	<pre> ''' Array should have at least two elements ''' if arr_size < 2: print("Invalid Input") return </pre>	<pre> /* Array should have at least two elements */ if (arr_size < 2) { Console.Write("Invalid Input"); return; } </pre>
<pre> /* Initialization of values */ min_l = 0; min_r = 1; min_sum = arr[0] + arr[1]; for(l = 0; l < arr_size - 1; l++) { for(r = l + 1; r < arr_size; r++) { sum = arr[l] + arr[r]; if(abs(min_sum) > abs(sum)) { min_sum = sum; min_l = l; min_r = r; }}}} </pre>	<pre> /* Initialization of values */ min_l = 0; min_r = 1; min_sum = arr[0] + arr[1]; for(l = 0; l < arr_size - 1; l++) { for(r = l+1; r < arr_size; r++) { sum = arr[l] + arr[r]; if(Math.abs(min_sum) > Math.abs(sum)) { min_sum = sum; min_l = l; min_r = r; }}}} </pre>	<pre> ''' Initialization of values ''' min_l = 0 min_r = 1 min_sum = arr[0] + arr [1] for l in range (0, arr_size - 1): for r in range (l + 1, arr_size): sum = arr[l] + arr[r] if abs(min_sum) > abs(sum) : min_sum = sum min_l = l min_r = r </pre>	<pre> /* Initialization of values */ min_l = 0; min_r = 1; min_sum = arr[0] + arr[1]; for (l = 0; l < arr_size - 1; l++) { for (r = l+1; r < arr_size; r++) { sum = arr[l] + arr[r]; if (Math.Abs(min_sum) > Math.Abs(sum)) { min_sum = sum; min_l = l; min_r = r; }}}} </pre>
<pre> /* Driver Code */ int main() { int arr[] = {1, 60, -10, 70, -80, 85}; minAbsSumPair(arr, 6); return 0; } </pre>	<pre> /* main function */ public static void main (String[] args) { int arr[] = {1, 60, -10, 70, -80, 85}; minAbsSumPair(arr, 6); } } </pre>	<pre> ''' Driver program to test above function ''' arr = [1, 60, -10, 70, -80, 85] minAbsSumPair(arr, 6); </pre>	<pre> /* main function */ public static void Main () { int []arr = {1, 60, -10, 70, -80, 85}; minAbsSumPair(arr, 6); } } </pre>

JavaScript	PHP	C
<pre> /* JavaScript code to find Two elements whose sum is closest to zero */ function minAbsSumPair(arr, arr_size) { var inv_count = 0; var l, r, min_sum, sum, min_l, min_r; </pre>	<pre> /* PHP program to find the Two elements whose sum is closest to zero */ function minAbsSumPair(\$arr, \$arr_size) { \$inv_count = 0; </pre>	<pre> /* C code to find Two elements whose sum is closest to zero */ # include <stdio.h> # include <stdlib.h> # include <math.h> void minAbsSumPair(int arr[], int arr_size) { int inv_count = 0; int l, r, min_sum, sum, min_l, min_r; </pre>
<pre> /* Array should have at least two elements */ if(arr_size < 2) { document.write("Invalid Input "); return; } </pre>	<pre> /* Array should have at least two elements */ if(\$arr_size < 2) { echo "Invalid Input"; return; } </pre>	<pre> /* Array should have at least two elements */ if(arr_size < 2) { printf("Invalid Input"); return; } </pre>
<pre> /* Initialization of values */ min_l = 0; min_r = 1; min_sum = arr[0] + arr[1]; for(l = 0; l < arr_size - 1; l++) { for(r = l + 1; r < arr_size; r++) { sum = arr[l] + arr[r]; if(Math.abs(min_sum) > Math.abs(sum)) { min_sum = sum; min_l = l; min_r = r; } } } </pre>	<pre> /* Initialization of values */ \$min_l = 0; \$min_r = 1; \$min_sum = \$arr[0] + \$arr[1]; for(\$l = 0; \$l < \$arr_size - 1; \$l++) { for(\$r = \$l+1; \$r < \$arr_size ; \$r++) { \$sum = \$arr[\$l] + \$arr[\$r]; if(abs(\$min_sum) > abs(\$sum)) { \$min_sum = \$sum; \$min_l = \$l; \$min_r = \$r; } } } </pre>	<pre> /* Initialization of values */ min_l = 0; min_r = 1; min_sum = arr[0] + arr[1]; for(l = 0; l < arr_size - 1; l++) { for(r = l+1; r < arr_size; r++) { sum = arr[l] + arr[r]; if(abs(min_sum) > abs(sum)) { min_sum = sum; min_l = l; min_r = r; } } } </pre>
<pre> /* Driver Code */ arr = new Array(1, 60, -10, 70, -80, 85); minAbsSumPair(arr, 6); </pre>	<pre> /* Driver Code */ \$arr = array(1, 60, -10, 70, -80, 85); minAbsSumPair(\$arr, 6); ?> </pre>	<pre> /* Driver program to test above function */ int main() { int arr[] = {1, 60, -10, 70, -80, 85}; minAbsSumPair(arr, 6); getchar(); return 0; } </pre>

2.4 Data Splitting

A natural way to generate train-validation-test sets from the data is to split at the problem level. Since the parallel programs are within each problem, splitting the data at problem level can avoid data leakage. However, during the data processing, we noticed that some problems are very similar. For example, “Check if a large number is divisible by 3 or not” and “Check whether a large number is divisible by 53 or not”. If one problem goes to the training set and the other goes to the test set, it can lead to potential data leakage and bias. To address this concern, we first cluster all the similar problems into groups, and make the split at the group-level. In this way, we can ensure that similar problems go to the same split. For clustering, we first calculate the similarity score (using Python *difflib.SequenceMatcher*) between every two pairs of problem descriptions, and then group all the problems using various similarity score thresholds (60%-80%) based on length of the descriptions.

There is also the additional challenge that the number of programs in different languages are imbalanced. Only 5.1% of the problems have C programs, and it is 31.5% for PHP programs. Random splitting at the problem level can exacerbate this problem, resulting in very small test/validation data set for C and PHP. Moreover, only a small number of problems have programs in all the 7 languages. It is beneficial to use these problems for evaluation, as they can provide a fair comparison across all the languages. Satisfying these two constraints, we take the following steps to create the data split:

1. Out of all the problems that have programs (solutions) in all 7 languages, we randomly sample the test and validation sets for C. We start out with creating the splits for C in particular since it represents the smallest proportion of the dataset.
2. Next, we first remove all the problems that have C programs. Out of all the problems that have programs in the remaining 6 languages (excluding C), we randomly sample the partial test and validation sets for PHP, so that the combined problems from this step and the previous one can be used as final test and validation sets of the PHP programs.
3. Finally, we remove all the problems that have C programs or PHP programs. Since the remaining 5 languages have approximately same number of programs, we randomly sample the partial test and validation sets and use them for all the 5 languages. The final test and validation set for each of the 5 languages is the combination of these problems and problems from the previous two steps. This allows us to maintain a split ratio of approximately 85-5-10 (train-val-test) for all the 7 languages.

Our splitting strategy provides a balanced split across languages and ensures there is no overlap between the test, validation or training sets across all languages. Tables [2.4](#) and [2.5](#) summarize the data statistics after the splits have been created using our strategy.

Table 2.4: Number of pairwise code-code data in training, validation, and testing splits for each language-pair. The upper triangle (in bold font) shows the number of parallel code snippets, and the lower triangle shows the number of parallel programs. This data is used for the Code Translation and XL Code Search tasks. (**Py** is short for Python. **JS** is short for Javascript.)

Lang		C++	Java	Py	C#	JS	PHP	C
C++	train	–	89040	80100	85662	69507	17811	3386
	val	–	4419	3913	4408	3808	923	352
	test	–	8059	7228	7922	6965	1647	222
Java	train	9450	–	77759	87065	69341	17853	2996
	val	490	–	3938	4437	3826	929	353
	test	901	–	7259	8011	7005	1672	238
Py	train	9139	8991	–	75843	67219	17616	2478
	val	468	471	–	3922	3750	923	311
	test	878	882	–	7215	6861	1655	203
C#	train	9187	9301	8826	–	68093	17873	2958
	val	488	491	470	–	3826	928	352
	test	890	898	877	–	6961	1668	238
JS	train	8482	8470	8182	8367	–	17117	1875
	val	472	475	459	475	–	921	309
	test	878	881	864	877	–	1617	200
PHP	train	3056	3068	3003	3071	2971	–	856
	val	157	158	153	158	157	–	271
	test	303	307	304	307	302	–	183
C	train	402	409	380	394	308	170	–
	val	59	59	59	59	59	55	–
	test	45	49	48	49	49	43	–

Table 2.5: Number of pairwise code-text data in each language. “Snippet” means snippet-comment pairs, and “Program” means program-description (problem description) pairs. This data is used for Code Summarization (Code-to-Text), Code Synthesis (Text-to-Code) and NL Code Search tasks.

		C++	Java	Py	C#	JS	PHP	C	Total
Snippet	train	93847	91089	81207	87583	70649	18027	3763	446165
	valid	4432	4460	3946	4436	3829	930	350	22383
	test	8118	8154	7293	8013	7033	1682	250	40543
Program	train	9797	9623	9263	9345	8590	3087	463	50168
	valid	492	494	472	491	475	158	60	2642
	test	909	911	887	899	886	308	51	4851

2.5 Data Characteristics

The final data set that we get after all the pre-processing, alignment, and splitting consists of 11,265 programming problems. Table 2.6 summarizes the quantitative characteristics of the final data set. As shown in Table 2.6, there are 57,661 unique programs. Each program consists of 8.81 snippets on average, which results in 509,091 snippets.

Table 2.6: The train-valid-test split and basic statistics of *XLCoST* data. SN - Snippets; PR - Program.

Split	Snippet-level								Program-level							
	C++	Java	Py	C#	JS	PHP	C	Total	C++	Java	Py	C#	JS	PHP	C	Total
train	93847	91089	81207	87583	70649	18027	3763	446165	9797	9623	9263	9345	8590	3087	463	50168
valid	4432	4460	3946	4436	3829	930	350	22383	492	494	472	491	475	158	60	2642
test	8118	8154	7293	8013	7033	1682	250	40543	909	911	887	899	886	308	51	4851
total	106397	103703	92446	100032	81511	20639	4363	509091	11198	11028	10622	10735	9951	3553	574	57661
Stats	C++	Java	Py	C#	JS	PHP	C	Avg.	C++	Java	Py	C#	JS	PHP	C	Avg.
#lines/code	3.41	3.71	2.41	3.82	3.23	4	4.05	3.37	32.45	34.93	20.54	35.64	26.47	23.23	31.5	29.71
#tokens/code	21.52	24.1	21.63	23.06	22.52	28.14	25.37	22.83	205	227.1	188.5	215.3	184.6	163.5	198	202
#tokens/text	8.25	8.14	7.97	8.23	7.96	8.45	9.67	8.15	10.68	10.67	10.75	10.7	10.87	9.91	8.19	10.66
#SN/PR	–	–	–	–	–	–	–	–	9.52	9.42	8.51	9.33	8.2	5.81	7.77	8.81

In addition to the quantitative characteristics, the following are the most important qualitative characteristics of the data set.

Multilingual: The data set contains parallel data in 8 languages (7 commonly used programming languages and English).

Parallel: The data set contains 4 types of parallel data, snippet-to-snippet, program-to-program, snippet-to-comment, program-to-problem (and comments) which further enables 10 different tasks.

Finely-aligned: The data is parallel at both snippet level and program level. To the best of our knowledge, this data set is the finest-aligned among parallel code data sets.

Large: It is the largest parallel data set for source code in terms of both size and number of languages.

Simple: Each program in this data set is standalone without dependency on other programs. It ensures that the complexity of the tasks is controllable.

All these characteristics enable the data set to be employed for solving a variety of software engineering tasks. These tasks include natural language program search, program summarization, program synthesis and program cross lingual program search as well. All these tasks can be accomplished both at the snippet level and at the full program level. Figure 2.2 serves to summarize how each component of the data set can be leveraged for these tasks.

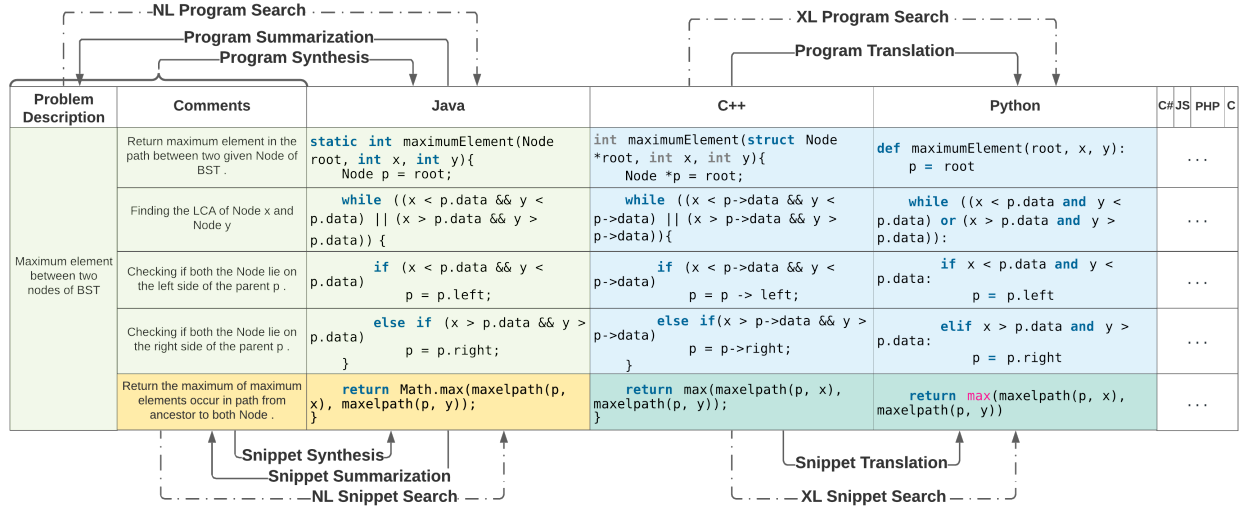


Figure 2.2: An illustration of the data and the tasks. The first column is the Problem Description; each cell in the second column is a Comment; each cell from the third column is a code Snippet. The combination of all the code snippets in a column is a Program (truncated due to space limitation). The arrows show the input and output data for each task. Solid lines are for generation tasks and dashed lines are for retrieval tasks. Note that the Program Synthesis task uses both Problem Description and Comments as input.

Chapter 3

Experiments with *XLCoST*

After obtaining the cleaned and processed data set, we perform several empirical analyses to ascertain the performance of the data set on each of the 10 tasks presented in Table 1.2. In section 3.1 of this chapter, we review the task settings. In section 3.2, we describe the evaluation metrics used for each task. In section 3.3, we describe the baseline models used and finally in section 3.4, we discuss the results that we obtained from all the different experiments performed.¹

3.1 Task Setting

The tasks we evaluate can be divided into two categories, generation and retrieval. The generation tasks include Code Translation, Code Summarization, and Code Synthesis; the retrieval tasks include NL (natural language) Code Search and XL (Cross-Lingual) Code Search. All the tasks are at both snippet-level and program-level. Figure 2.2 shows the input and output data for each of the tasks.

3.1.1 Generation Tasks

Code Translation (Code-to-Code)

Code Translation is the problem of converting source code from one programming language to another. Efficient and accurate code translation is valuable in scenarios like legacy code migration, software platform adaptation, etc. In recent years, deep learning based methods

¹ *This chapter, with modifications is adapted from our arXived work: Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, Chandan K. Reddy “XLCoST: A Benchmark Dataset for Cross-lingual Code Intelligence”.*

have been employed to address this problem. However, due to lack of high-quality pairwise data, translation between many programming language pairs remains unexplored.

The proposed *XLCoST* data set provides parallel data in 7 common programming languages, which supports the translation of 42 language pairs. Moreover, the data is parallel at both snippet and program level. The fine-grained alignment is particularly beneficial for translation tasks. The Code Translation task includes two tasks, Snippet Translation and Program Translation.

Code Summarization (Code-to-Text)

The objective of Code Summarization task is to generate natural language descriptions of the code that is given as input. We perform this task under two settings, generating snippet level summary by leveraging the comment-snippet pairings, and generating problem level summary using the problem description and program code pairings. The most valuable applications of this task include increasing the comprehensibility of uncommented or unfamiliar code to first time viewers and making it easier to collaborate as well as educate.

Code Synthesis (Text-to-Code)

The Code Synthesis task focuses on generating source code from text inputs. It includes Snippet Synthesis and Program Synthesis. We use the comment of each code snippet as input to generate the code snippet for the Snippet Synthesis task, since they are of similar length (on average 8 tokens for comments and 23 tokens for snippets). However, programs are usually much longer (Avg. 202 tokens) than problem descriptions (Avg. 11 tokens). To generate programs, it is necessary that the input text is detailed and informative. Therefore, we use a combination of problem description and step-by-step comments as input to generate the entire program. Since the programs in *XLCoST* are well commented (on average 9 comments/snippets per program, shown in Table 2.6), we can ensure that the models have enough information to synthesize the whole program.

3.1.2 Retrieval Tasks

NL Code Search

The immense popularity of code/knowledge sharing platforms ² is testament to the fact that searching for relevant code and problem solutions is a genuine challenge software developers face. Although these forums provide a solution to this problem, it can still be very hard to find the exact solution to a problem without putting in the right keywords which usually

²<https://www.stackoverflow.com/>

map to the post descriptions rather than the code itself. Direct NL to Code search is a task that could potentially bridge this gap, especially for beginner programmers who might not be familiar with the keywords associated with their queries in the first place. To that end, we present two NL (Natural Language) Code Search tasks:

Comment-to-Snippet Search: For Comment-to-Snippet Search, the natural language comments associated with each snippet are used as queries and the expected output is a ranked list of code snippets that perform the task described in the comment. In this ranked list, we are looking for the exact match of the code snippet associated with the NL comment query to be ranked the highest. This is a language specific task i.e., we look for candidate solutions of the NL comment within the same language. As Figure 2.2 shows, we already have the the comment to code snippet pairings in the dataset for all languages. We create a separate dataset for each of the 7 languages and use **MRR** (Mean Reciprocal Rank) score for evaluating this task.

Description-to-Program Search: The Description-to-Program Search task is similar to Comment-to-Snippet Search task, the only difference being that instead of comments, the NL queries are full program descriptions and the candidate solutions are also complete programs instead of snippets. Here, we again create a different dataset for each language, containing the NL description and program code pairs and use **MRR** score as the evaluation metric.

XL Code Search

XL (Cross-lingual) Code Search is the task of retrieving code that performs similar functions in multiple other languages given a piece of code in one particular language. Natural Language code search has been extensively explored in recent years but, to the best of our knowledge, using code as queries to search for similarly functioning code in multiple other languages is a task that requires more enquiry. Kim et al. [43] introduce a method for statically identifying code fragments that could be semantically related to user input code. Its applications can be similar to code translation which include legacy code migration and it could also help programmers learning new languages transition from one language to another more easily.

Snippet-to-Snippet Search: Snippet-to-Snippet search uses a code snippet in a given language as an input query and the expected output is a ranked list of code snippets that perform the same function but in other languages. In contrast to the NL Code Search setting, the candidate list in this case can contain multiple correct answers corresponding to the different languages in the data set.

For this task we create 7 different data sets, one for each language where the chosen language is the query language and all other languages form the candidate solutions. Take the data set for C++ as an example which contains entries like “1057-C++-1/1057-C#-

1”. This basically represents the datapoint where the first snippet of problem ID 1057 in C++ is the query and the corresponding answer snippet is in C#. However this is not the only correct pairing, the data set contains all the possible correct pairings which include {1057-C#-1, 1057-C-1, 1057-Python-1, 1057-Javascript-1, 1057-PHP-1, 1057-Java-1}. Any of these solutions when present the output candidate list is considered as a correctly chosen candidate. It should however also be noted that all queries do not have exhaustively all other languages as candidate solutions. To account for multiple correct answers we use a modified **MRR** (Mean Reciprocal Rank) for evaluation. We formally define the modifications we made in section 3.2.

Program-to-Program Search: The Program-to-Program Search is setup in the exact the same way as the Snippet-to-Snippet Search, the only difference being that the both the query code and the solution candidate is the full program instead of a snippet. The data format and evaluation strategies remain the same as Snippet-to-Snippet Search.

3.2 Evaluation Metrics

We use BLEU [44] score to evaluate code-to-text generation tasks; BLEU and CodeBLEU [45] to evaluate code-to-code and text-to code generation tasks; MRR to evaluate retrieval tasks.

- **BLEU:** BLEU score has been the defacto evaluation metric used to evaluate natural language translation tasks. Given an input code sample, BLEU score evaluates the n -gram overlap between the generated and the ground-truth target text and code.
- **CodeBLEU:** BLEU measures the similarity between the generated translation and a set of reference texts. However, programming languages have more rigorous syntax and semantics as compared to natural languages. A minor change in the code sequence, such as addition or removal of a bracket, may not affect the BLEU score by much, but it can potentially alter the structure and functionality of the code substantially. CodeBLEU is designed for automatic evaluation of code synthesis. It takes into consideration the Abstract Syntax Tree (AST) matching and Data Flow Graph (DFG) matching, which measure the syntax and semantics of the code, respectively making it a better evaluation metric for code related tasks.
- **Mean Reciprocal Rank (MRR):** The reciprocal rank is defined as the inverse of the rank of the first correct candidate for a given query. MRR is the mean of the reciprocal rank for all the queries in the test set. In order to evaluate our XL Code Search task, we modified the traditional definition of the **MRR** metric in order to account for the possibility of multiple correct candidate solutions. We modify it in the following way: Given a query q_i from the set of queries $Q = \{q_1, q_2 \dots q_m\}$, candidate set $C_i =$

$\{c_{i1}, c_{i2} \dots c_{in}\}$ corresponding to q_i and the answer set $A_i = \{a_{i1}, a_{i2}, \dots a_{ik}\}$ where $k \in [1, 6]$, r_{ij} is the reciprocal rank of the j^{th} candidate c_{ij} for $c_{ij} \in A_i$.

$$MRR_{q_i} = \frac{1}{k} \sum_{\substack{j=1 \\ c_{ij} \in A_i}}^n r_{ij} \quad (3.1)$$

$$MRR_Q = \frac{1}{m} \sum_{i=1}^m MRR_{q_i} \quad (3.2)$$

- **Compilation Rate:** Although CodeBLEU has been proposed as improvement over BLEU score to evaluate the generation quality, compilation and execution are the true north stars when it comes to programming languages. If generated code can be successfully compiled and executed, that is the strongest signal for the quality of generation. In keeping with that spirit, we also evaluate the compilation rate of programs. The compilation failure/success is determined by passing the generated sequences to the language specific compilers. If no errors are reported by the compiler, it is considered a successful compilation. The compilation rate is formally defined as:

$$\text{Compilation rate} = \frac{\# \text{ of generated test set programs that compile}}{\text{total } \# \text{ of programs in test set}} \times 100 \quad (3.3)$$

3.3 Baselines

We use CodeBERT, PLBART, CodeT5 for all the generation tasks; RoBERTa and CodeBERT for retrieval tasks.

Naive Copy [37] directly copies the input source code as the translation output. This baseline shows how similar two programming languages are.

RoBERTa [4] is a robustly optimized version of BERT pretrained on huge natural language corpora. We use it only for retrieval tasks.

CodeBERT [28] uses the BERT [46] architecture pretrained on CodeSearchNet [38] data. CodeSearchNet contains data from 6 programming languages (Java, Python, Javascript, PHP, Ruby, Go). We use the encoder-only version for retrieval tasks and encoder-decoder version (the decoder is randomly initialized) for generation tasks.

PLBART [35] is initialized with mBART [47] and further pretrained on a large-collection of Java and Python functions and natural language descriptions from Github and StackOverflow with denoising auto-encoding objective.

CodeT5 [34] employs T5 [48] architecture and is pretrained on corpora of 8 programming languages (Java, Python, C#, Javascript, PHP, C, Ruby, Go) with identifier-aware objective.

3.4 Results and Discussion

We now present and discuss the results we obtained for each of the tasks evaluated with the hope that it can provide some insights on how to design a model for the challenges introduced in this data set.

Table 3.1, shows the performance of baseline models for Code Translation, Code Synthesis, Code Summarization, and Code Search tasks. Moving from the top to bottom. For code generation tasks (Translation and Synthesis) we report the CodeBLEU scores. For the Summarization task we report the BLEU scores. In Table 3.2, we also report the BLEU scores for the generation tasks, Translation and Synthesis.

3.4.1 Effect of Sequence-to-Sequence Pretraining

In Table 3.1, on an average, CodeBERT performs significantly worse than PLBART and CodeT5 on almost all the generation tasks (refer to the first three sections of the table). Different from PLBART and CodeT5, which are both encoder-decoder models pretrained with sequence-to-sequence objectives, only the encoder in CodeBERT is pretrained, and the decoder weights are randomly initialized for sequence-to-sequence tasks. Experimental results show that encoder-decoder architecture and sequence-to-sequence pretraining are better aligned with generation tasks and thus can potentially achieve superior performance.

3.4.2 Effect of Pretraining on Specific Languages

CodeBERT is pretrained on CodeSearchNet, which contains data from 6 programming languages, Java, Python, Javascript, PHP, Ruby, and Go. PLBART is pretrained on Java and Python from GitHub data. CodeT5 is trained on the 6 languages from CodeSearchNet and additional C and C#. In Table 3.1, CodeT5 consistently outperforms the other two models for almost all generation tasks. When the source or target language is C, CodeT5 outperforms the other two by a wide margin. Pre-training on specific languages can potentially benefit the generation tasks with these languages as either input or output.

3.4.3 Performance on Low-Resource Languages

In Table 3.1, most models performs significantly worse on C compared to other languages, both when C is source or target language, in almost all the tasks (except for Code Search). As shown in Table 2.6, C has the least number of samples for all the tasks. It shows that tasks in low-resource languages are potentially more challenging.

3.4.4 Comparison between BLEU and CodeBLEU

In section 3.2, we described some of the limitations of BLEU score. Due to the fact that BLEU only measures the n -gram matching and ignores code syntax and semantics, it can be observed from our translation results (see Table 3.2) that BLEU scores clearly over estimate the model performance. Almost all BLEU score values can be found to be greater than CodeBLEU scores presented in Table 3.1. The effect is more pronounced for the program level tasks than for the snippet level tasks. This shows that maintaining long term structure in full programs, which are much longer than snippets, is harder.

However, from our Code Synthesis results (see Table 3.2), we observe that the BLEU scores are lower than CodeBLEU scores. This is because the Code Synthesis results are much lower (in absolute value) than Translation results to begin with, and taking into account AST and DFG matching increases the overall scores.

Table 3.1: From top to bottom, the table contains results for Code Translation, Code Synthesis, Code Summarization, and Code Search at the snippet-level and program-level. CodeBLEU scores are reported for Code Generation tasks (Translation and Synthesis). For Translation, the language column on the left represents the source language and the row on the top represents the target language. BLEU scores are reported for Summarization and MRR for Search.

CodeBLEU	Model	Snippet-level							Program-level						
		C++	Java	Py	C#	JS	PHP	C	C++	Java	Py	C#	JS	PHP	C
C++	Naive Copy	–	64.56	34.79	63.19	53.16	42.56	84.2	–	57.36	17.68	58.02	53.16	18.97	75.91
	CodeBERT	–	84.94	74.55	84.99	82.79	68.56	45.46	–	74.73	24.96	76.35	72.95	50.4	21.84
	PLBART	–	83.85	74.89	84.57	83.19	68.62	83.95	–	75.26	70.13	78.01	61.85	67.01	72.59
	CodeT5	–	86.35	76.28	85.85	84.31	69.87	90.45	–	80.03	71.56	81.73	79.48	70.44	85.67
Java	Naive Copy	70.85	–	35	78.43	57.81	42.49	69.74	64.25	–	39.87	72.68	57.81	42.51	62.48
	CodeBERT	87.27	–	58.39	92.26	84.63	67.26	39.94	79.36	–	8.51	84.43	76.02	51.42	21.22
	PLBART	87.31	–	58.3	90.78	85.42	67.44	72.47	81.41	–	66.29	83.34	80.14	67.12	63.37
	CodeT5	88.26	–	74.59	92.56	86.22	69.02	82.78	84.26	–	69.57	87.79	80.67	69.44	78.78
Python	Naive Copy	39.22	31.89	–	31.79	38.34	36.02	37.79	37.47	29.78	–	27.59	38.42	35.48	35.66
	CodeBERT	80.46	58.5	–	54.72	57.38	65.14	10.7	68.87	28.22	–	17.8	23.65	49.3	18.32
	PLBART	80.15	74.15	–	73.5	73.2	66.12	62.15	74.38	67.8	–	66.03	69.3	64.85	29.05
	CodeT5	81.56	78.61	–	78.89	77.76	67.54	68.67	78.85	73.15	–	73.35	71.8	67.5	56.35
C#	Naive Copy	69.78	78.71	34.77	–	57.85	42.53	66.73	64	73.63	40.09	–	57.79	42.96	60.87
	CodeBERT	86.96	90.15	56.92	–	84.38	67.18	40.43	78.52	82.25	10.82	–	75.46	51.76	21.63
	PLBART	84.98	6.27	69.82	–	85.02	67.3	75.74	80.17	81.37	67.02	–	79.81	67.12	57.6
	CodeT5	88.06	91.69	73.85	–	85.95	68.97	81.09	83.59	85.7	69.52	–	80.5	69.63	77.35
JS	Naive Copy	60.82	59.25	38.84	64.27	–	41.56	55.84	53.81	51.77	42.31	54.86	–	42.11	49.04
	CodeBERT	84.38	84.42	52.57	84.74	–	66.66	33.29	75.43	72.33	9.19	75.47	–	52.08	19.79
	PLBART	84.45	84.9	69.29	85.05	–	67.09	72.65	80.19	76.96	64.18	78.51	–	67.24	67.7
	CodeT5	85.06	85.48	73.15	85.96	–	68.42	80.49	82.14	79.91	68.42	81.77	–	68.76	74.57
PHP	Naive Copy	36.33	35.61	24.62	36.67	35.55	–	35.95	34.62	31.33	25.68	32.81	32.26	–	33.45
	CodeBERT	82.58	81.57	69.29	80.96	79.94	–	28.45	50.13	46.81	16.92	49.75	48.12	–	22.19
	PLBART	83.87	81.66	71.17	78	82.94	–	57.39	79.4	72.77	61.26	74.16	44.26	–	56.23
	CodeT5	86.33	85.12	73.22	84.56	83.56	–	79.3	85.55	82.09	72.26	83.79	81.72	–	65.86
C	Naive Copy	83.93	65.46	38.49	63.05	55.55	41.85	–	78.4	59.41	20.2	59.83	53.54	19.75	–
	CodeBERT	45.84	39.69	13.55	39.71	29.85	38.88	–	21.7	21.27	21.1	19.5	15.64	31.71	–
	PLBART	82.53	72.35	49.16	75.78	75.05	60.86	–	78.42	13.45	5.53	45.15	31.47	25.17	–
	CodeT5	90.26	81.81	63.81	83.05	79.73	66.32	–	88.17	76.12	56.32	80.2	76.5	64.28	–
CodeBLEU	Model	C++	Java	Py	C#	JS	PHP	C	C++	Java	Py	C#	JS	PHP	C
Code Synthesis	CodeBERT	22.7	25.53	12.26	23.44	23.87	36.47	10.63	26.51	31.14	24.5	33.37	29.09	39.84	18.08
	PLBART	34.89	32.23	4.62	29.36	29.63	37.56	22.88	44.09	41.55	33.77	40.7	38.33	43.01	6.72
	CodeT5	35.48	33.51	21.1	30.64	29.99	36.37	21.93	45.18	42.73	35.02	43.6	38.66	45.02	34.88
BLEU	Model	C++	Java	Py	C#	JS	PHP	C	C++	Java	Py	C#	JS	PHP	C
Code Summarization	CodeBERT	14.4	13.13	3.96	14.07	11.81	11.25	5.84	7.68	5.47	2.04	7.58	7.67	7.5	6.64
	PLBART	14.77	13.76	8	14.37	10.93	9.07	7.5	7.65	6.35	4.86	9.23	6.78	6.03	4.14
	CodeT5	17.36	16.69	10.76	17.44	14.34	13.42	6.63	9.62	8.82	6.32	7.75	8.23	10.5	12.84
MRR	Model	C++	Java	Py	C#	JS	PHP	C	C++	Java	Py	C#	JS	PHP	C
NL Code Search	RoBERTa	25.77	25.85	27.08	25.64	26.78	33.47	36.14	51.47	50.4	48.98	52.24	50.05	62.01	56.34
	CodeBERT	29.77	29.41	30.94	29.08	31.2	38.75	41.56	59.13	56.07	57.97	56.65	54.37	65.13	47.13
XL Code Search	RoBERTa	41.73	41.25	36.16	41.18	43.17	41.17	37.1	48.28	47.66	46.11	46.4	47.6	43.76	40.15
	CodeBERT	42.11	41.71	36.98	41.52	43.41	41.09	37.87	48.71	48.33	47.24	47.96	47.66	44.02	40.43

Table 3.2: BLEU scores for two tasks (Translation and Code Synthesis) separated by a double horizontal rule. Above the double rule are presented BLEU scores of the Translation task for the 42 programming language pairs in the *XLCoST* dataset. Below the double rule are the BLEU scores for the Code Synthesis task. Column headers represent target languages. For Translation, row headers represent source languages.

BLEU	Model	Snippet-level							Program-level						
		C++	Java	Py	C#	JS	PHP	C	C++	Java	Py	C#	JS	PHP	C
C++	Naive Copy	–	64.57	37.29	65.89	59.73	37.44	84.44	–	64.47	34.48	65.98	58.09	38.13	84
	CodeBERT	–	85.03	79.72	85.64	84.61	87.18	44.48	–	80.09	15.43	81.24	78.14	50.68	11.7
	PLBART	–	84.02	80.12	84.86	85.34	87.31	85.26	–	81.23	77.5	83.96	69.6	83.94	77.94
	CodeT5	–	86.45	81.73	86.55	86.24	89.84	91.82	–	84.29	78.69	85.69	83.75	90.88	93.64
Java	Naive Copy	64.48	–	34.04	76.95	60.42	35.02	65.67	64.88	–	31.28	78.09	58.23	34.35	65.48
	CodeBERT	88.18	–	61.45	92.64	86.42	84.57	38.47	83.24	–	2.51	88.31	81.16	52.7	12.27
	PLBART	87.69	–	55.05	91.63	87.17	84.92	71.43	85.83	–	73.16	89	84.62	84.18	66.1
	CodeT5	89.2	–	79.37	92.9	87.95	88.12	84.29	87.67	–	76.92	90.5	85.38	88.86	85.64
Python	Naive Copy	37.36	34	–	35.06	42.64	22.05	38.5	34.43	30.81	–	31.93	39.57	21.22	35.68
	CodeBERT	82.02	57.17	–	53.94	57.64	80.32	9.01	72.56	25.24	–	8.39	17.97	48.46	4.5
	PLBART	81.17	69.17	–	69.29	68	82.27	57.8	78.4	73.1	–	71.14	73.73	79.56	27.07
	CodeT5	83.07	76.61	–	78.59	78.16	85.14	70.64	81.8	77.9	–	78.21	76.25	85	51.89
C#	Naive Copy	65.71	77.11	34.99	–	61.13	35.25	66.63	66.13	78.04	32.2	–	59.09	36.08	66.39
	CodeBERT	87.76	89.99	59.63	–	86.13	84.41	39.19	82.48	86.86	4.68	–	80.8	53.39	11.48
	PLBART	87.03	4.97	70.85	–	86.88	84.66	76.83	84.77	87	73.94	–	84.55	84.2	60.27
	CodeT5	88.81	91.67	77.64	–	87.75	88.01	83.54	87.18	89.28	76.81	–	85	89.25	83.72
JS	Naive Copy	59.84	60.36	42.63	61.23	–	33.05	56.07	57.99	57.26	39.58	58.54	–	34	53.8
	CodeBERT	84.97	84.1	54.19	84.69	–	83.37	31.68	79.54	77.97	3.36	80.73	–	54.02	11.04
	PLBART	84.75	84.44	70.99	84.85	–	84.19	73.57	84.24	82.46	70.68	83.86	–	84.4	69.46
	CodeT5	85.74	85.22	77.46	85.96	–	86.91	82.03	85.72	84.14	75.46	85.44	–	87.5	77.92
PHP	Naive Copy	37.39	35.03	22.08	35.2	33.17	–	36.62	38.15	34.32	21.36	36.08	34.44	–	37.53
	CodeBERT	84.56	82.35	74.68	81.93	82.82	–	27.94	52.19	50.94	10.87	54.61	53.13	–	8.42
	PLBART	84.81	82.13	76.42	78.86	85.43	–	53.62	84.56	78.61	68.9	80.26	44.71	–	56.16
	CodeT5	88.59	85.88	79.47	85.67	86.46	–	83.05	89.96	86.95	80.29	87.77	87.01	–	68.74
C	Naive Copy	84.34	65.65	38.47	66.64	56.19	36.67	–	83.99	65.29	35.94	66.4	54.52	37.53	–
	CodeBERT	45.55	38.79	9.83	39.09	26.85	27.03	–	15.51	17.77	6.01	14.92	13.06	8.53	–
	PLBART	83.01	72.21	44.76	76.26	78.8	72.37	–	84.88	10.65	4.02	38.53	18.6	0.2	–
	CodeT5	91.76	82.12	65.89	84.06	82.16	82.82	–	93.15	83.08	54.6	85.39	82.42	78.7	–
Code Synthesis	CodeBERT	17.19	18.78	7.82	18.58	19.53	22.54	5.16	21.39	27.16	19.58	30.83	25.53	29.6	8.85
	PLBART	24.01	28.12	1.31	26.61	17.27	20.16	12.9	39.94	41.01	29.92	38.92	35.95	35.91	3.53
	CodeT5	28.52	29.65	12.87	28.16	21.54	18.7	12.47	41.53	42.37	31.9	42.14	36.31	39.97	28.64

3.4.5 Effect of Transfer Learning from Snippet-level Training

From Table 3.1, first section, we noticed that models perform significantly better at snippet-level than program-level on most language pairs in the translation task. This is because 1) Snippets are much shorter than programs. As shown in Table 2.6, the average length of snippets is 1/7 of the programs. 2) Snippet data is much more than program data. As shown in Table 1.2, the amount of pairwise snippet data is 8 times of program data. Motivated by this, we employ transfer learning from snippet-level training to improve the Program Translation performance on low-resource language C. Table 3.3 shows the performance of each model with and without the transfer learning. For example, CodeBERT is trained only on program data; “CodeBERT + ST” (ST is short for Snippet Transfer) model is first trained on the snippet data, and then on program data. All the models’ performances improve by a wide margin on all the language pairs after snippet-level transfer learning, both when C is the source or target language.

Table 3.3: Transfer learning from Snippet-Level training for Program Translation task on low resource language C. ST - Snippet Transfer.

Model	C-C++	C-Java	C-Py	C-C#	C-JS	C-PHP	C++-C	Java-C	Py-C	C#-C	JS-C	PHP-C
CodeBERT	21.67	21.27	21.1	19.48	15.68	31.71	21.87	21.27	18.32	21.57	19.79	22.19
CodeBERT+ST	38.85	37.55	19.79	33.52	27.1	37.61	31.99	30.52	24.07	34.16	29.67	28.35
PLBART	78.42	13.43	5.53	45.14	31.42	25.17	72.61	63.4	29.01	57.6	67.71	56.15
PLBART+ST	81.1	70.78	44.26	72.68	73.27	60.71	79.72	77.3	47.48	74.09	72.6	64.64
CodeT5	88.17	76.15	56.3	80.2	76.42	64.28	85.67	78.76	56.44	77.38	74.56	65.8
CodeT5+ST	89.06	79.04	62.61	80.53	78.59	68.31	88.96	82.08	60.97	80.93	79.58	77.58

Chapter 4

Analysis of Compilation Rate

In section 3.2, we argued how the gold standard for judging the quality of the generated hypothesis when it comes to code is always its compilability and its ability to execute without runtime errors. Evaluating executability of the generated hypothesis is much more complicated and we discuss the challenges faced in section 4.5 of this chapter. We evaluate the compilability of the generated hypothesis for both within-distribution and out-of-distribution data and we also show the effectiveness of *XLCoST* in being able to improve the compilation rate using some ablation studies. We present a detailed study of the compilation rate of generated hypotheses in the following sections of this chapter.¹

4.1 Compilation Rate for Generative Tasks

First, in Tables 4.1 and 4.2 we report the compilation rate of the hypotheses generated by Synthesis and Translation respectively using the *XLCoST* test sets. We do not report compilation statistics for Javascript since it is not a compiled language.

On average, CodeT5 generates the most compilable code for Program Synthesis as well as Translation. CodeBERT generates the least compilable code for translation and PLBART performs poorly for Synthesis. For both Synthesis as well as Translation we again see that the lowest resource language, C results in the poorest performance with respect to compilation. Something to note in Table 4.2, is the Python compilation rate for CodeBERT. We observed that 50-60% of the hypotheses generated by CodeBERT with Python as the target language, were the singular “\n” token. When passed through the compiler this is returned as successfully compilable code. This artificially over-estimates the compilation

¹ *This chapter, with modifications is adapted from our arXived work: Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, Chandan K. Reddy “XLCoST: A Benchmark Dataset for Cross-lingual Code Intelligence”.*

rate for Python hypotheses generated by CodeBERT which is presented without parenthesis in Table 4.2. The actual compilation rate, after removing all the hypotheses which only contained the singular “\n” is presented Table 4.2 without the parenthesis. We choose to present both these scores in order to maintain consistency of the metric proposed in equation 3.3. The compilation rate in the parenthesis should be considered the true score for CodeBERT Python translation. We call this out so that it may not be erroneously inferred that CodeBERT generates more compilable code with Python as the target language.

Table 4.1: Compilation rate of the generated programs in Program Synthesis task.

Program-level-synthesis							
	C++	Java	Python	C#	PHP	C	Avg
CodeBERT	51.4	46.51	83.43	36.78	27.16	0	40.88
PLBART	33.55	7.84	84.86	5.77	75.72	1.89	34.94
CodeT5	60.86	59.61	92.04	54.84	68.69	35.85	61.98

Table 4.2: Compilation rate of the generated programs in Program Translation task. For Python (Py) the score without the parenthesis is calculated using all the generated hypotheses, including the singular “\n” token. The score within the parenthesis is calculated excluding all the hypotheses where only the singular “\n” token is generated.

Program-level-translation								
	Model	C++	Java	Py	C#	PHP	C	Avg
C++	CodeBERT	–	40.28	(87.83)94.05	38.02	11.69	0	36.80
	PLBART	–	47.12	89.21	60.44	82.47	25.53	60.95
	CodeT5	–	64.28	88.88	66.7	71.43	48.94	68.04
Java	CodeBERT	47.01	–	(85.91)94.52	57.19	13.46	0	42.43
	PLBART	64.5	–	87.72	67.32	61.86	24	61.08
	CodeT5	67.86	–	88.05	73.42	75.32	36	68.13
Python	CodeBERT	29.74	73.57	–	3.75	12.86	0	32.47
	PLBART	35.9	23.79	–	10.47	86.5	6.12	26.55
	CodeT5	64.43	62.28	–	53.91	73.95	32.65	57.44
C#	CodeBERT	47.58	55.01	(85.92)93.61	–	13.14	0	41.86
	PLBART	57.14	45.75	86	–	61.86	26	55.35
	CodeT5	66.7	68.85	87.98	–	74.04	36	66.21
JS	CodeBERT	38.98	35.96	(86.41)94.74	37.9	12.38	0	43.99
	PLBART	62.81	36.29	85.57	44.59	65.15	24	63.68
	CodeT5	65.7	66.15	88.14	67.45	72.31	40	79.95
PHP	CodeBERT	10.71	8.01	(31.74)72.35	6.41	–	0	19.49
	PLBART	52.6	21.47	73.31	48.4	–	0	39.15
	CodeT5	67.86	67.63	91.32	65.38	–	25	63.48
C	CodeBERT	0	0	(0)0	0	0	–	0
	PLBART	36.17	0	6.12	8	0	–	30.58
	CodeT5	74.47	54	59.18	46	27.27	–	52.18

4.2 Top Compilation Errors in Generated Programs

Table 4.3 shows the top compilation error types from compiling the generated programs from the Program Translation task. We aggregated the results of generated programs from all the baselines by the target language, because 1) the top error types of each baseline are very similar and 2) presenting detailed error stats for the 42 different language pairs and 3 models will take a lot of space. From this table, we can see that the top error types are mostly syntactic errors, such as bracket mismatch (C++, PHP, C), indentation mismatch (Python), missing “,” (Java). This indicates that the models need improvement in capturing the structure of the programs.

Table 4.3: Top compilation errors for Code Translation in each target language (Javascript not included).

Language	Top-3 Compilation Errors in Each Target Language		
C++	expected ‘}’ at end of input	stray ‘#’ in program	‘define’ does not name a type
Java	‘;’ expected	not a statement	unclosed character literal
Python	SyntaxError: invalid syntax	SyntaxError: unexpected EOF while parsing	IndentationError: expected an indented block
C#	Too many characters in character literal	Unexpected symbol ‘end-of-file’	Newline in constant
PHP	Syntax error, unexpected ‘}’, expecting EOF..	Syntax error, unexpected ‘)’..	Syntax error, unexpected EOF on line 1
C	expected declaration or statement at end of input	expected ‘=’, ‘,’, ‘;’, ‘asm’ ... before ‘)’ token	expected statement before ‘)’ token

4.3 Ablation studies with out-of-distribution (OOD) data

In addition to evaluating the effectiveness of *XLCoST* using its own test sets, we would also like to see how much *XLCoST* helps in improving translation quality on out-of-distribution data. The greatest challenge in conducting effective ablation studies with multiple languages is the lack of availability of baselines to compare against. We discussed in Chapter 1, how most benchmark data sets only support certain languages and as a result, other researchers who release their models for similar tasks, only do so for those limited languages. For example the authors of both CodeT5 [34] and PLBART [35] only release Java-C#(and vice-versa) translation models fine-tuned on the CodeXGLUE [37] data set. These models are fine-tuned on 10,300 Java-C# program pairs.

In order to design our ablation experiments, we obtain another code data set gathered from the popular interview preparation website called LeetCode to measure OOD generalization. This data set also has code solutions for data structures and algorithms based problems in C++, Java and Python. The data set statistics are presented in Table 4.4.

In addition to LeetCode data, we also leverage the CoST [41] data set as a baseline to compare performance against *XLCoST* since CoST enables us to compare more language

pairs rather than just Java-C#. It is important to note that although CoST is also similarly aligned at the snippet level like *XLCoST*, it is much smaller in size. We refer the reader to Zhu et al. [41] for a more detailed breakdown of CoST.

Table 4.4: Leetcode data statistics.

Language	Number of Programs
C++	557
Java	1018
Python	1775

4.3.1 Comparison against author baselines

First we take the author provided fine-tuned models, models fine-tuned on CoST and models fine-tuned on *XLCoST*, and use these to generate Java to C# translations of the programs in the LeetCode data. None of the models have been trained on any of the LeetCode data. The results are presented in Table 4.5 and we can see that none of the C# hypotheses generated by the author provided fine-tuned models compile. Only 0.29% of the hypotheses generated by PLBART fine-tuned on CoST compile and 1.28% of the hypotheses generated by CodeT5 fine-tuned on CoST compile. *XLCoST* fine-tune models are able to generate comparatively more compilable code on OOD data, with PLBART+*XLCoST* giving a compilation rate of almost 30%.

Table 4.5: A comparison of compilation rate for Java-C# (LeetCode) translation between the author’s fine-tuned models, CoST fine-tuned models and *XLCoST* fine-tuned models.

Program-level-translation	
Model	Java-C#
PLBART+CodeXGLUE	0
PLBART+CoST	0.29
PLBART+XLCoST	28.88
CodeT5+CodeXGLUE	0
CodeT5+CoST	1.28
CodeT5+XLCoST	1.38

4.3.2 Snippet-Transfer for OOD data

In Chapter 3, section 3.4.5, we saw how the Snippet-Transfer training technique was effective in improving performance for low resource languages like C. Encouraged by those results, we

would also like to see if Snippet Transfer helps in generating more compilable hypotheses for the OOD LeetCode data. Details of Snippet-Transfer training can be found in section 3.4.5. In this ablation study we take the CoST, *XLCoST* and Snippet-Transfer fine-tuned models and generate Translation hypotheses in C++, Java, Python, C#, PHP and C, using the LeetCode programs as the source languages. We compare the compilation rate between hypotheses generated by CodeT5 and PLBART models (see Table 4.6). Wavy-lines represent the best performing PLBART models and bold values represent the best performing CodeT5 models.

We see that barring a few language pairs like C++-Java and Java-Python for PLBART; Python-C++ and Python-C# for CodeT5, both models exhibit improvement in compilation rate when fine-tuned on *XLCoST* instead of CoST. PLBART trained using Snippet-Transfer performs poorer than PLBART+*XLCoST* except for C++-C#, C++-PHP and Java-Python where improvements can be observed. CodeT5 responds much better to Snippet-Transfer training and improvements can be observed for almost all language pairs other than where Python is the source language. When Java is the source language, Snippet-Transfer results in significant performance gains with CodeT5. Similar improvements can also be observed for C++-Java, C++-C# and C++-PHP translation using CodeT5. The results are not as consistent with Python as the source language for both PLBART and CodeT5.

Table 4.6: Program Translation compilation rate for OOD Leetcode data. ST stands for **S**nippet **T**ransfer. Wavy-lines represent the best performing PLBART models. Bold values represent the best performing CodeT5 models.

Program-level-translation							
	Model	C++	Java	Py	C#	PHP	C
C++	PLBART+CoST	–	<u>0.18</u>	1.44	0.54	0.90	0
	PLBART+XLCoST	–	0	<u>66.07</u>	15.08	11.13	<u>0.36</u>
	PLBART+ST	–	0	22.26	<u>17.23</u>	<u>21.72</u>	0.17
	CodeT5+CoST	–	1.80	44.34	0.18	0.18	0
	CodeT5+XLCoST	–	2.51	68.76	6.64	0.54	2.69
	CodeT5+ST	–	7.54	26.92	14.18	43.62	3.41
Java	PLBART+CoST	1.67	–	12.28	0.29	1.18	0
	PLBART+XLCoST	<u>4.91</u>	–	0.88	<u>28.88</u>	<u>25.25</u>	<u>8.74</u>
	PLBART+ST	0.78	–	<u>66.50</u>	16.80	24.36	2.94
	CodeT5+CoST	1.18	–	50.00	1.28	0.20	0
	CodeT5+XLCoST	1.67	–	59.33	1.38	20.63	1.38
	CodeT5+ST	10.60	–	83.98	10.31	27.40	1.17
Python	PLBART+CoST	0.17	3.10	–	0	1.35	0
	PLBART+XLCoST	<u>9.46</u>	<u>5.41</u>	–	<u>8.11</u>	<u>10.65</u>	<u>14.99</u>
	PLBART+ST	0	0.22	–	1.91	10.30	0.28
	CodeT5+CoST	1.69	2.93	–	3.15	0.85	0
	CodeT5+XLCoST	1.18	41.97	–	1.24	10.70	3.49
	CodeT5+ST	0.67	4.33	–	2.80	19.88	0.60

4.3.3 Ablation study with LeetCode fine-tuning

In this experiment we take the LeetCode data presented in Table 4.4 and divide the programs for each language into train/test/val splits using an 80-10-10 ratio. We would like to investigate if training on LeetCode data improves compilation rate of the test set and also investigate the impact of *XLCoST* fine-tuning along with LeetCode data. We use PLBART and CodeT5 again in three different settings; models trained only on LeetCode data, models trained only on *XLCoST* data and models trained on both *XLCoST* and LeetCode data. For all three settings, the hypotheses are generated using the LeetCode test set of the corresponding language. Results from this experiment are presented in Table 4.7. When C++ and Java are source languages, we see the both PLBART and CodeT5 generate the most compilable hypotheses when they are trained on *XLCoST* data first and then LeetCode data, except for the single case of Java-C++ translation using PLBART. In almost all cases training on both *XLCoST* and LeetCode performs better than just training on LeetCode, we do acknowledge however that the standalone LeetCode training sets are much smaller than the *XLCoST* training sets. It would be interesting to investigate if this effect persists even when both the LeetCode training set and *XLCoST* training set sizes are similar. It is promising to see however that *XLCoST* is able to provide significant performance gains on OOD data when only a small number of training examples are available. In certain cases like C++-Python using both PLBART and CodeT5, C++-Java using CodeT5, only training on *XLCoST* is able to outperform only training on LeetCode, while in other cases like C++-Java and Java-Python using PLBART, Java-C++ and Java-Python using CodeT5, LeetCode training performs better. For Python-C++ and Python-Java translation as well, CodeT5+*XLCoST*+LC results in significant performance gains. Our ablation studies do give us a positive signal that *XLCoST* data can help generate more compilable code when it comes to code translation.

4.4 Qualitative analysis of generated hypotheses

Tables 4.8 and 4.9 show some qualitative examples from our ablation studies with PLBART and CodeT5 respectively. In Table 4.8, we have a C++ program as the source and we can see the Python translations generated by PLBART in the three different settings evaluated in section 4.3.3. We can observe that PLBART+LC generates code with additional “_ :” tokens in the beginning which throw a compilation error. PLBART+*XLCoST* generates the logic of the whole function again inside an if statement which throws a `return outside function` error. PLBART+*XLCoST*+LC generates code without any compilation errors.

In Table 4.9, we have a C++ program as the source and we can see the Java translations generated by CodeT5 in the three different settings evaluated in section 4.3.3. It can be

Table 4.7: Comparison of compilation rate on fine-tuning with LeetCode data, fine-tuning with *XLCoST* data and fine-tuning with both data sets. LC stands for **LeetCode**. Wavy-lines represent best performing PLBART models. Bold values represent best performing CodeT5 models.

		Program-level-translation		
	Model	C++	Java	Py
C++	PLBART+LC	–	4.87	1.81
	PLBART+XLCoST	–	0	74.54
	PLBART+XLCoST+LC	–	<u>58.53</u>	<u>92.72</u>
	CodeT5+LC	–	0	67.27
	CodeT5+XLCoST	–	4.87	74.54
	CodeT5+XLCoST+LC	–	39.02	92.72
Java	PLBART+LC	0	–	26.00
	PLBART+XLCoST	<u>4.87</u>	–	1.00
	PLBART+XLCoST+LC	0	–	<u>64.00</u>
	CodeT5+LC	31.70	–	90.00
	CodeT5+XLCoST	0	–	58.00
	CodeT5+XLCoST+LC	48.78	–	95.00
Python	PLBART+LC	0	<u>81.00</u>	–
	PLBART+XLCoST	<u>29.09</u>	8.00	–
	PLBART+XLCoST+LC	0	0	–
	CodeT5+LC	30.90	12.00	–
	CodeT5+XLCoST	1.81	36.00	–
	CodeT5+XLCoST+LC	47.27	35.00	–

observed that CodeT5+LC generates multiple class definitions with the same class name which throws an error. CodeT5+*XLCoST* generates better quality code, however before the function call in `main`, variables `N` and `K` are not defined and given values which throws a compilation error. CodeT5+*XLCoST*+LC generates code that does not throw any compilation errors, however a more careful look reveals that the generated code does not follow the logic in the source C++ program. The generated hypothesis simply uses a for loop to append values to a list and returns it's size. Although the generated code does not have any compilation errors, it does not follow the logic that it is supposed to. This highlights the importance of including execution based metrics in the analysis as well, that can check for program correctness. Evaluating program correctness using execution based metrics is not a trivial task however and we discuss the challenges we faced in accomplishing this task in section 4.5.

4.5 Challenges with evaluating execution based metrics

In the previous sections, we have discussed the evaluation of the code translation task using compilability as a metric. A natural question that arises is that in addition to evaluating

Table 4.8: Qualitative example for C++-Python translation using PLBART. i) PLBART+LC generates code with extra “-:” tokens. ii) PLBART+*XLCoST* generates the whole function again inside `main` with a return outside of any function definition. iii) PLBART + *XLCoST* + LC generates code without compilation errors.

Source C++	Translated Python		
	PLBART + LC	PLBART + <i>XLCoST</i>	PLBART + <i>XLCoST</i> +LC
<pre> #include <stdlib.h> #include <math.h> #include <stdio.h> #include <vector> #include <string> #include <unordered_map> #include <iostream> using namespace std ; class Solution { public : int maxOperations (vector < int > & nums , int k) { unordered_map < int , int > lookup ; int result = 0 ; for (const auto & num : nums) { if (lookup . count (k - num) && lookup [k - num]) { -- lookup [k - num] ; ++ result ; } else { ++ lookup [num] ; } } return result ; } }; int main () { } </pre>	<pre> - : def maxOperations (self , nums , k) : lookup = { } result = 0 for num in nums : if num and lookup [k - num] : lookup [k - num] += 1 result += 1 else : lookup [num] -= 1 return result </pre>	<pre> def maxOperations (nums , k) : lookup = dict () result = 0 for num in nums : if (k - num) and lookup [k - num] : lookup [k - num] -= 1 result += 1 else : lookup [num] = 1 return result if __name__ == '__main__' : nums = [] for num in nums : if (k - num) and lookup [k - num] : lookup [k - num] += 1 result += 1 else : lookup [num] = 1 return result </pre>	<pre> class Solution (object) : def maxOperations (self , nums , k) : lookup = { } result = 0 for num in nums : if k - num in lookup and lookup [k - num] : lookup [k - num] -= 1 result += 1 else : lookup [num] += 1 return result </pre>

compilability, why not execute the programs and evaluate for correctness of output as well? The first challenge we face when it comes to evaluation is the availability of test cases. The correctness of problems relating to Data Structures and Algorithms is usually examined through pre-defined test cases. These are input-output pairs specific to each problem such that when the input is provided, the code must generate the corresponding pre-defined output in order to be deemed correct based on the implemented logic.

Table 4.9: Qualitative example for C++-Java translation using CodeT5. i) CodeT5+LC generates code with multiple class definitions of the same name. ii) CodeT5+*XLCoST* generates code that used undefined variables in the function call. iii) CodeT5 + *XLCoST* + LC generates code without compilation errors.

Source C++	Translated Java		
	CodeT5 + LC	CodeT5 + <i>XLCoST</i>	CodeT5 + <i>XLCoST</i> +LC
<pre>#include <stdlib.h> #include <math.h> #include <stdio.h> #include <vector> #include <string> #include <unordered_map> #include <iostream> using namespace std ; class Solution { public : int kthGrammar (int N , int K) { return bitCount (K - 1) % 2 ; } private : int bitCount (int n) { int count = 0 ; for (; n ; n &= n - 1) { ++ count ; } return count ; } }; int main () { }</pre>	<pre>public static class Solution1 { public int kthGrammar (int N , int K) { return bitCount (N - 1) % 2 ; } public static class Solution1 { public int kthGrammar (int N , int K) { return bitCount (K - 1) % 2 ; } } public static class Solution1 { public int kthGrammar (int N , int K) { return bitCount (N - 1) % 2 ; } } } }</pre>	<pre>import java . util . * ; class GFG { static class Solution { int kthGrammar (int N , int K) { return bitCount (K - 1) % 2 ; } } private int bitCount (int n) { int count = 0 ; for (; n != 0 ; n &= n - 1) { ++ count ; } return count ; } } public static void main (String [] args) { Solution s = new Solution () ; s . kthGrammar (N , K) ; }</pre>	<pre>import java . util . ArrayList ; import java . util . List ; import java . util . List ; @ SuppressWarnings (" unchecked") public class P1344 { public static void main (String args []) { } public static class Solution1 { public int kthGrammar (int N , int K) { List < Integer > list = new ArrayList < > () ; for (int i = 0 ; i < N ; i ++) { list . add (i) ; } return list . size () ; } } } }</pre>

On the LeetCode website², two to three (on average) test cases are always provided as examples to the user as part of the problem description in order to help them formulate their solution. The main test cases that the online judge uses to evaluate the solution are hidden. In order to solve the challenge of availability of test cases, we scraped these example test cases with the problem description as text. An example of the scraped text is presented in Figure 4.1. Lines 7 - 22 in Figure 4.1, are of significant interest to us since they contain the example test cases. We parse these test cases from the metadata using some regular expressions and obtain a well structured JSON file containing test cases for each problem,

²<https://leetcode.com>

as shown in Figure 4.2.

However, using these test cases and executing the model generated programs is not as straight forward as it seems. There are multiple steps involved. First the test case needs to be placed in the language specific syntax after the function and class definitions that contain the solution code. Then a call to the function must also be placed with the correct variables as input arguments. The function may or may not be contained within a class which will determine how it needs to be called. There are significant syntactic variations in the generated hypotheses which makes it very hard to design and implement an automated test execution pipeline. Refer to Table 4.10 for an example. We have a LeetCode source program in the C++ language and its corresponding generated translations in Java and Python using the PLBART model. Notice how in the Source C++ program, the target function, `totalFruit`, is contained inside the `Solution` class but it is not explicitly called at the end of the program in the `main` function. Both the Java and Python translations however do contain the function calls at the end with the corresponding accessory code. In order to develop an effective test execution pipeline, all these edge cases need to be considered for all the different languages keeping their syntactical peculiarities in mind. We need to setup the test variables with their values, determine whether the function has already been called or not and if it has not been called, setup the function call appropriately based on whether it is a part of a class or not. These are non-trivial challenges to tackle for multiple languages which hindered us from being able to develop an effective test execution pipeline. This certainly remains as a piece of future work that we would like to tackle.

```
1 You are visiting a farm that has a single row of fruit trees arranged from left to right.
   The trees are represented by an integer array fruits where fruits[i] is the type of
   fruit the ith tree produces.
2 You want to collect as much fruit as possible. However, the owner has some strict rules
   that you must follow:
3 You only have two baskets, and each basket can only hold a single type of fruit. There is
   no limit on the amount of fruit each basket can hold.
4 Starting from any tree of your choice, you must pick exactly one fruit from every tree (
   including the start tree) while moving to the right. The picked fruits must fit in
   one of your baskets.
5 Once you reach a tree with fruit that cannot fit in your baskets, you must stop.
6 Given the integer array fruits, return the maximum number of fruits you can pick.
7 Example 1:
8     Input: fruits = [1,2,1]
9     Output: 3
10    Explanation: We can pick from all 3.
11 Example 2:
12    Input: fruits = [0,1,2,2]
13    Output: 3
14    Explanation: We can pick from trees [1,2,2]. If we had started at the first tree, we
        would only pick from trees [0,1].
15 Example 3:
16    Input: fruits = [1,2,3,2,2]
17    Output: 4
18    Explanation: We can pick from trees [2,3,2,2].
19 If we had started at the first tree, we would only pick from trees [1,2].
20 Example 4:
21    Input: fruits = [3,3,3,1,2,1,1,2,3,3,4]
22    Output: 5
23    Explanation: We can pick from trees [1,2,1,1,2].
24 Constraints:
25     1 <= fruits.length <= 105
26     0 <= fruits[i] < fruits.length
```

Figure 4.1: Sample LeetCode raw metadata. The metadata contains problem description, example tests and constraints.

```
1 {  
2   "pid": "904",  
3   "tests": [  
4     {"inputs": "fruits = [1,2,1]",  
5      "output": "3"},  
6  
7     {"inputs": "fruits = [0,1,2,2]",  
8      "output": "3"},  
9  
10    {"inputs": "fruits = [1,2,3,2,2]",  
11     "output": "4"},  
12  
13    {"inputs": "fruits = [3,3,3,1,2,1,1,2,3,3,4]",  
14     "output": "5"},  
15  ]  
}
```

Figure 4.2: A single example from the processed test case file. Each line item in the file corresponds to in problem ID and contains it's corresponding test cases parsed from the metadata.

Table 4.10: Table showing a sample C++ program from the LeetCode data and the generated Java and Python translations using the PLBART model.

Source C++	Translated Java	Translated Python
<pre> #include <stdlib.h> #include <math.h> #include <stdio.h> #include <vector> #include <string> #include <unordered_map> #include <iostream> using namespace std ; class Solution { public : int totalFruit (vector < int > & tree) { unordered_map < int , int > count ; int result = 0 ; for (int i = 0 , j = 0 ; j < tree . size () ; ++ j) { ++ count [tree [j]] ; while (count . size () > 2) { -- count [tree [i]] ; if (count [tree [i]] == 0) { count . erase (tree [i]) ; } ++ i ; } result = max (result , j - i + 1) ; } return result ; } }; int main () { } </pre>	<pre> import . util . * ; class Solution { static int totalFruit (int [] tree) { HashMap < Integer , Integer > count = new HashMap < Integer , Integer > () ; int result = 0 ; for (int i = 0 , j = 0 ; j < tree . length ; ++ j) { if (count . containsKey (tree [j])) { count . put (tree [j] , count . get (tree [j]) + 1) ; } else { count . put (tree [j] , 1) ; } while (count . size () > 2) { -- count . get (tree [i]) ; if (count . get (tree [i]) == 0) { count . remove (tree [i]) ; } ++ i ; } result = Math . max (result , j - i + 1) ; } return result ; } } class GFG { public static void main (String [] args) { Solution s = new Solution () ; s . totalFruit (s . tree) ; } } </pre>	<pre> from collections import defaultdict def totalFruit (tree) : count = defaultdict (int) result = 0 for i in range (len (tree)) : count [tree [i]] += 1 while (len (count) > 2) : count [tree [i]] -= 1 if (count [tree [i]] == 0) : count.pop (tree [i]) i += 1 result = max (result , j - i + 1) return result if __name__ == '__main__' : print (totalFruit (tree)) </pre>

Chapter 5

Reinforcement Learning to Improve Translation Quality

In this chapter we introduce a deep reinforcement learning model to try to improve the quality of code generated with the translation task. The models that we have used so far in this work use the conventional next-token prediction objective, which is heavily influenced by natural language processing (NLP). However, the model optimization process ignores the functional correctness and compilability of the generated codes. We would like to employ the compilability feedback signals as the external source of knowledge into the model optimization based on the Deep Reinforcement Learning (DeepRL) framework. More specifically, we follow these steps: (1) consider the pre-trained code Language Model (LM) of each language-pair as an actor network; (2) synthetically sample sequences of generated codes for each language-pair by actor model; (3) define a reward based on the compilability signals of the generated code samples as well as the abstract syntax tree (AST) and data flow graph (DFG) information ; (4) utilize DeepRL framework to optimize the pre-trained models from the compilability-aware reward. Our hypothesis is that incorporating these signals as feedback into the optimization process will lead the model to generate code that has fewer syntactical errors.

5.1 Background

5.2 RL4CNCT

We propose the model RL4CNCT: **R**einforcement **L**earning for **C**ompilable **N**eural **C**ode **T**ranslation. The key idea of RL4CNCT is to provide a framework for the code translation model fine-tuning based on reinforcement learning (RL) that takes advantage of compiler

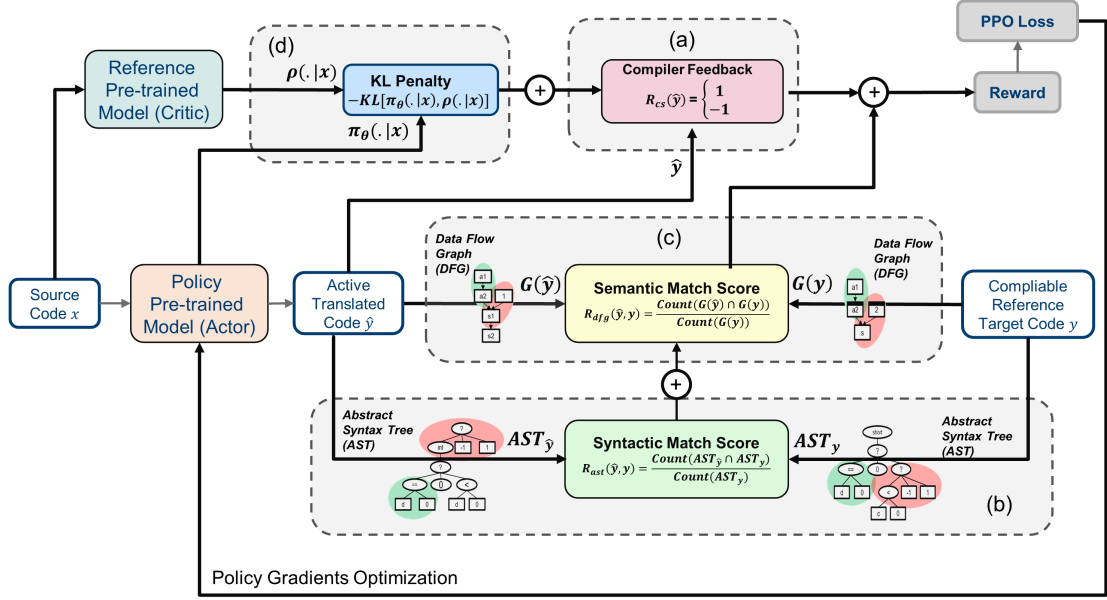


Figure 5.1: An overview of the RL4CNCT method. Reading from left to right, we begin with source code sequence x , a policy pre-trained model and a reference pre-trained model. Both models begin with the same set of weights. The policy model generates a translated code sequence $\hat{y}$. a) The translated code sequence is passed through the compiler to determine the compilation reward signal, +1 for successful compilation, -1 for unsuccessful compilation. b) Abstract Syntax Tree match score is calculated between the generated sequence $\hat{y}$ and the ground truth sequence y . c) Data Flow Graph match score is calculated between the generated sequence $\hat{y}$ and the ground truth sequence y . d) KL-Divergence between the output distribution of the actor model (policy) and the critic model (reference) is minimised so that the policy does not stray too far away from the reference. Signals from a), b), c) and d) are combined into a reward and used in the PPO loss function to fine-tune the pre-trained policy model.

feedback as extra knowledge for model optimization, as a result, enhancing translated codes in terms of compilability and functional correctness. Fig. 5.1 shows the general structure of our proposed RL4CNCT model with the proximal policy optimization (PPO) approach taking reward $\mathcal{R}$. As shown in Fig. 5.1, the total reward is composed of four elements: (i) compiler feedback; (ii) syntactic match score; (iii) semantic match score; and (iv) KL-divergence penalty. We provide further details about each of these components in the subsections below.

5.2.1 Problem Formulation

The code generation procedure can be formulated as a sequential discrete finite-horizon Markov Decision Process (MDP) with the use of reinforcement learning (RL) in which an agent interacts with the compiler over discrete horizon T which is equivalent to the maximum length in the generated code tokens. The RL4CNCT is formulated as follows:

State $\mathcal{S}$: The state of environment at each time-step, denoted as $s_t = (\hat{y}_{<t}, x)$, $s_t \in \mathcal{S}$, is determined by the source code x , as well as the set of translated tokens before that time-step $\hat{y}_{<t}$.

Action $\mathcal{A}$: The code model decides the action at each time-step, denoted as $a_t = \hat{y}_t$, $a_t \in \mathcal{A}$, which is equivalent to the predicted translated token at time-step t .

Policy $\pi_\theta(a_t|s_t)$: The stochastic policy network parameterized by the downstream code translation model that predicts the next token conditioned on the previously generated tokens and the source code, so, $\pi_\theta(\hat{y}_t|\hat{y}_{<t}, x) : \mathcal{S} \rightarrow \Delta(\mathcal{A})$ where $\Delta(\mathcal{A})$ denotes the probability distribution over all actions (target vocabulary) that policy network produces to predict the next action $\hat{y}_t$ by *top-k* sampling from it. We initialize the policy with the pre-trained reference code model ρ , i.e., $\pi_\theta^0(\cdot) = \rho$.

Reward $\mathcal{R}$: The reward $\mathcal{R}(\hat{y}, x, y)$ will be obtained at the end of the code translation episode (i.e., the *< endoftokens >* token) based on the translated code compilability and its alignment with compilable codes. The reward function $\mathcal{R}(\cdot)$ is composed of different components which are explained in more detail in Section 5.2.2.

Advantage $\hat{A}_\pi^t$: The advantage estimator at time-step t is defined as follows.

$$\begin{aligned}\hat{A}_\pi^t &= \delta_t + \gamma \delta_{t+1} + \dots + \gamma^{T-t+1} \delta_{T-1}, \\ \delta_t &= r_t - V_\pi(\hat{y}_{<t}) + \gamma V_\pi(\hat{y}_{<t+1}),\end{aligned}\tag{5.1}$$

where γ is the discount rate; r_t is the reward at time-step t ; and $V_\pi(s_t)$ is the state value function at time-step t which can be approximated by a linear function on top of decoder hidden states.

Objective: The objective of RL4CNCT is to find a policy that maximizes the reward of translated codes sampled from the policy.

$$\max_{\theta} \mathbb{E}_{x \sim \mathcal{X}, \hat{y} \sim \pi_\theta(\cdot|x)} [\mathcal{R}(\hat{y}, x, y)],\tag{5.2}$$

where $\mathcal{X}$ is the training set of source codes; $\pi_\theta(\cdot)$ is the policy network; and $\mathcal{R}(\cdot)$ is the reward function. In order to reduce the variability of predictions, we formulate the objective function as a maximization of the advantage instead of reward as shown in Eq. (5.3).

$$\max_{\theta} \mathbb{E}_{x \sim \mathcal{X}, \hat{y} \sim \pi_\theta(\cdot|x)} \left[\sum_{t=0}^T \hat{A}_\pi^t((\hat{y}_{<t}, x), \hat{y}_t) \right],\tag{5.3}$$

We adopt the policy gradient method to estimate the gradient of non-differentiable reward-based objectives in Eqs. (5.2) and (5.3). Therefore, updating policy parameters for a given

source code x can be derived as:

$$\max_{\theta} \mathcal{L}_{\theta}^{PG} = \max_{\theta} \mathbb{E}_{\hat{y} \sim \pi_{\theta}} \left[\sum_{t=0}^T \left(\log \pi_{\theta}(\hat{y}_t | \hat{y}_{<t}) \hat{A}_{\pi}^t \right) \right], \quad (5.4)$$

$$\text{where } \nabla_{\theta} \mathcal{L}_{\theta}^{PG} = \mathbb{E}_{\hat{y} \sim \pi_{\theta}} \left[\sum_{t=1}^T \left(\nabla_{\theta} \log \pi_{\theta}(\hat{y}_t | \hat{y}_{<t}) \hat{A}_{\pi}^t \right) \right], \quad (5.5)$$

where $\nabla_{\theta} \mathcal{L}_{\theta}^{PG}$ refers to the estimated gradient of objective function based on the policy parameterized by θ . In order to further reduce the variations and avoid changing the policy too much at each iteration, the objective function in Eq. (5.4) will be reformulated as shown in Eq. (5.6), called the conservative policy iteration.

$$\begin{aligned} \mathcal{L}_{\theta}^{PG} &= \mathbb{E}_{\hat{y} \sim \pi_{\theta}} \left[\sum_{t=0}^T \left(\frac{\log \pi_{\theta}(\hat{y}_t | \hat{y}_{<t})}{\log \pi_{\theta_{old}}(\hat{y}_t | \hat{y}_{<t})} \hat{A}_{\pi}^t \right) \right] \\ &= \mathbb{E}_{\hat{y} \sim \pi_{\theta}} \left[\sum_{t=0}^T \left(c_{\pi}^t(\theta) \hat{A}_{\pi}^t \right) \right], \end{aligned} \quad (5.6)$$

where θ_{old} is the policy parameters before the update; and $c_{\pi}^t(\theta)$ is the ratio of log-probabilities from new and old policies.

5.2.2 Reward Function

Figure 5.1 illustrates that the reward of RL4CNCT is composed of four different components which are designed to guide and control actions simultaneously towards generating more compilable codes. These components are designed due to (1) the sparsity of code compilation signal which is only received at the end of code translation episode; and (2) the high chance of policy divergence from the pre-trained code models. (5.7) shows the combination of these different reward terms in the final reward vector $\mathcal{R}(\hat{y}, x, y) \in \mathbb{R}^T$ with T as the translation episode length.

$$\begin{aligned} \mathcal{R}(\hat{y}, x, y) &= \{r_t : t = 1, \dots, T\}, \\ r_t &= 1(cond) \left[R_{cs}(\hat{y}) + R_{ast}(\hat{y}, y) + R_{dfg}(\hat{y}, y) \right. \\ &\quad \left. - \beta R_{kl}(x, \hat{y}_{<t}) \right] + 1(\neg cond) [-\beta R_{kl}(x, \hat{y}_{<t})], \\ cond &= (\hat{y}_t == \langle endoftokens \rangle) \end{aligned} \quad (5.7)$$

where r_t is the total reward at time-step t ; $R_{cs}(\cdot)$, $R_{ast}(\cdot)$, and $R_{dfg}(\cdot)$ are the compiler signal, syntactic match score, and the semantic match score reward terms that will be received at

the end of the translation episode where $\hat{y}_t == \langle \text{endoftokens} \rangle$. The $R_{kl}(x, \hat{y}_{<t})$ is a kl-divergence penalty between the reference pre-trained model and the active policy network which is imposed to reward at each time-step to control actions. β is also a linear combination weight for the divergence penalty to balance the combination of different reward terms.

For each source code x , we sample multiple translated codes in the target language based on the current policy network, $\hat{y} \sim \pi_\theta(.|x)$. Then, we pass these sampled translated codes $\hat{y}$ to a compiler and determine the binary reward based on the compilation signal as shown in Eq. (5.8). This reward term is designed to guide actions towards producing better compilable codes.

$$R_{cs}(\hat{y}) = \begin{cases} +1, & \text{if } \hat{y} \text{ passed compilation test} \\ -1, & \text{otherwise} \end{cases} \quad (5.8)$$

Syntactic Matching Score

Since the compiler signal alone is too sparse, we also need to add additional information to better control and guide the syntactic structure of policy samples. To do so, we define a syntactic matching score $R_{ast}(\hat{y}, y)$ between the generated hypothesis $\hat{y} \sim \pi_\theta(.|x)$ and a compilable ground-truth code y in the target language. The goal is to maximize this matching score for better compilability. We use the abstract syntax tree (AST) to find a tree representation of the abstract syntax structure. Then, we compare the AST sub-trees extracted from the hypothesis and reference target ASTs, respectively, and calculate the syntactic match score as a percentage of matched AST sub-trees.

$$R_{ast}(\hat{y}, y) = \text{Count}(AST_{\hat{y}} \cap AST_y) / \text{Count}(AST_y) \quad (5.9)$$

where $\text{Count}(AST_{\hat{y}} \cap AST_y)$ is the number of matched AST sub-trees between the hypothesis $\hat{y}$ and reference y ; and $\text{Count}(AST_y)$ is the total number of reference AST sub-trees. This score can assess the syntactic quality of code since the differences between ASTs can be impacted by grammatical problems such as token missing and data type errors.

Semantic Matching Score

For better compilability, we need to also take into account the semantic matching between hypothesis $\hat{y}$ and the compilable reference y , in addition to their syntactic matching. In programming languages, the semantic of a code is closely related to the dependencies of its variables. As a result, in order to construct a semantic matching score, we make use of the data-flow graph (DFG), a graph representation of code in which the nodes stand in for variables and the edges for the sources of each variable's values. We denote DFG of a code Y as $\mathcal{G}(Y) = (V; E)$ where $V = \{v_1, \dots, v_m\}$ is the set of variables, and $e_{i,j} = \langle v_i, v_j \rangle$ is the

$i \rightarrow j$ edge showing that the value of j -th variable originates from the i -th variable. Then, we calculate the semantic match score as a percentage of matched data-flows in DFGs.

$$R_{dfg}(\hat{y}, y) = \text{Count}(\mathcal{G}(\hat{y}) \cap \mathcal{G}(y)) / \text{Count}(\mathcal{G}(y)) \quad (5.10)$$

where $\text{Count}(\mathcal{G}(\hat{y}) \cap \mathcal{G}(y))$ represents the number of matched DFG data-flows between hypothesis $\hat{y}$ and compilable reference y ; and $\text{Count}(\mathcal{G}(y))$ represents the total number of reference DFG data-flows. Maximizing this score can guide and control policy to generate codes which are more aligned with compilable target codes in terms of variable relations, thus, enhancing the semantic quality of the translated code.

KL-Divergence Constraint

We incorporate a negative kl-divergence penalty $KL(\pi||\rho)$ into the reward to prevent the active policy π from deviating too much from the pre-trained code model ρ . The kl-penalty at each time-step t can be approximated as:

$$\begin{aligned} R_{kl}(x, \hat{y}_{<t}) &= KL(\pi||\rho) \approx \log \frac{\pi(\cdot|x, \hat{y}_{<t})}{\rho(\cdot|x, \hat{y}_{<t})} \\ &= \log(\pi(\cdot|x, \hat{y}_{<t})) - \log(\rho(\cdot|x, \hat{y}_{<t})) \end{aligned} \quad (5.11)$$

where $\log(\pi(\cdot|x, \hat{y}_{<t}))$ and $\log(\rho(\cdot|x, \hat{y}_{<t}))$ are the log-probabilities obtained from the active policy π and pre-trained model ρ at time-step t given source code x and the previously predicted tokens $\hat{y}_{<t}$. This reward term can control actions and play the role of entropy bonus in controlling exploration and exploitation where greater β in Eq. (5.7) will provide less exploration and more exploitation.

5.2.3 Loss Function

We employ proximal policy optimization (PPO) [49] and define the loss function of RL4CNCT as follows.

$$\mathcal{L}_{\theta} = -\mathcal{L}_{\theta}^{PG} + \alpha \mathcal{L}_{\theta}^{VF} \quad (5.12)$$

$$\mathcal{L}_{\theta}^{PG} = \mathbb{E}_{\hat{y} \sim \pi_{\theta}} \left[\sum_{t=0}^T \min \left(c_{\pi}^t(\theta) \hat{A}_{\pi}^t, \text{clip} \left(c_{\pi}^t(\theta), 1 - \epsilon, 1 + \epsilon \right) \hat{A}_{\pi}^t \right) \right] \quad (5.13)$$

$$\mathcal{L}_{\theta}^{VF} = \mathbb{E}_{\hat{y} \sim \pi_{\theta}} \left[\sum_{t=0}^T \left(V_{\pi}(\hat{y}_{<t}) - V_t^{trg} \right)^2 \right] \quad (5.14)$$

Table 5.1: *XLCoST* dataset statistics after filtering for compilable programs.

Lang		C++	Java	Py	C#	PHP	C
C++	train	–	–	–	–	–	–
	val	–	–	–	–	–	–
	test	–	–	–	–	–	–
Java	train	5251	–	–	–	–	–
	val	266	–	–	–	–	–
	test	520	–	–	–	–	–
Python	train	5325	4811	–	–	–	–
	val	266	250	–	–	–	–
	test	529	496	–	–	–	–
C#	train	5464	5081	5027	–	–	–
	val	279	262	259	–	–	–
	test	553	513	529	–	–	–
PHP	train	1758	1595	1785	1764	–	–
	val	91	85	93	92	–	–
	test	192	176	201	197	–	–
C	train	233	191	169	204	92	–
	val	34	34	32	34	33	–
	test	28	28	24	29	26	–

where the loss function $\mathcal{L}_\theta$ is the linear combination of surrogate policy objective function $\mathcal{L}_\theta^{PG}$ and the value function squared error term $\mathcal{L}_\theta^{VF}$. Therefore, minimizing the loss function will lead to the maximization of the surrogate advantage policy objective as well as the minimization of value error. In Eqs. (5.12)-(5.14), ϵ is the policy ratio clip range, and α is the linear combination weight between different loss terms.

5.3 Dataset

For our experiments we will be using the *XLCoST* dataset again. Since the RL framework necessarily utilizes compilability as feedback, we filter the original *XLCoST* data and only keep programs which compile without throwing syntax errors. In addition to the framework requiring the input code to be compilable, our hypothesis is that using ground truth data that is free of compilation errors will provide a stronger signal to RL4CNCT for correcting syntactic errors than using the full data, which could also contain flawed programs. Table 5.1 shows statistics of the compilable filtered dataset for all languages (except JavaScript which is excluded due to the lack of compilation).

5.4 Baseline

We employ the CodeT5 [34] model for baseline comparisons which we have observed to be the best performing model in all our previous experiments. The baseline CodeT5 models are fine-tuned on the unfiltered *XLCoST* data for the translation task. The RL4CNCT method employs the CodeT5 models fine-tuned on the unfiltered *XLCoST* data in the previous step, and further trains them on the filtered *XLCoST* data using the reinforcement learning objective. We compare the performance between these two settings based on compilability of the hypotheses generated by the filtered test sets demonstrated in Table 5.1.

5.5 Results

The results of our experiments are presented in Table 5.2. We perform our experiments for a subset of languages available in the *XLCoST* data set, stratified by low resource and high resource languages. C is the lowest resource language in the data set with the fewest number of data points which is reduced even further after filtering. C++, Python and C# represent languages that have a higher number of data points.

In Table 5.2, we can observe that RL4CNCT significantly helps in increasing the compilability of generated code for high resource languages. The improvement when Python is the target language is especially pronounced and we conjecture that this is because of the easy nature of the Python programming language itself. C++ and C# have more complicated syntactic elements, they are strongly typed languages as compared to Python and so the model may have to work harder to make improvements.

For the low resource language C, we see only minute improvements in performance as compared to CodeT5. Only 1-2 additional translations generated by RL4CNCT compile, but since the test set for C is already small, the percentage improvement looks higher. This shows us the importance of having a large enough training data size in order to be able to see meaningful performance gain.

5.6 Qualitative Analysis

In Tables 5.3 and 5.4, we present some examples of the programs for which compilation is improved by RL4CNCT. Table 5.3 shows the C++ translations generated by both CodeT5 and RL4CNCT when Java is used as the source language. We can observe in the first example that CodeT5 generates a non-standard data type called `subset` which takes in a pair of integers. The use of the non-standard data-type without importing it or defining it causes a compilation error. Moreover it can be observed that the source Java program

Table 5.2: Comparison of CodeT5 model trained on *XLCoST* without RL and the RL4CNCT model. Bold values represent the best performing models. C++, Python and C# are languages with larger number of data points as compared to C.

	Model	High Resource			Low Resource	
		C++	Py	C#	PHP	C
C++	CodeT5	–	74.10	53.80	84.80	46.40
	RL4CNCT	–	83.70	58.90	87.50	46.40
Java	CodeT5	59.00	56.00	69.70	84.00	42.80
	RL4CNCT	64.00	90.70	72.70	88.00	57.10
Python	CodeT5	53.30	–	56.80	78.10	8.30
	RL4CNCT	58.60	–	47.20	78.10	12.50
C#	CodeT5	60.30	55.00	–	83.20	41.30
	RL4CNCT	64.70	83.30	–	87.30	55.10
PHP	CodeT5	68.20	80.00	75.10	–	38.40
	RL4CNCT	72.90	90.50	75.60	–	42.30
C	CodeT5	71.40	70.80	62.00	61.50	–
	RL4CNCT	75.00	75.00	65.50	76.90	–

uses a regular array containing singular integers rather than pairs. RL4CNCT generates the C++ translation using the correct `vector` data-type. In the second example for Java-C++ translation we observe that CodeT5 generates a return statement to exit out of the function using syntax that is not consistent with C++. It tries to use syntax that resembles Java more closely, which causes the compilation error. RL4CNCT however is able to generate the correct C++ syntax to return a value from the function.

Table 5.4 shows the Python translations generated by both CodeT5 and RL4CNCT when C++ is used as the source language. In the first example we observe that CodeT5 generates an extra closing parenthesis which raises a compilation error for `unmatched ‘)’`. RL4CNCT is able to generate the correct number of required parentheses. In the second example we observe that the syntax generated by CodeT5 for function definition is wrong. Function definitions in Python are required to be preceeded with the `def` keyword and RL4CNCT is able to generate the correct function definition as opposed to CodeT5 which uses `func`.

Table 5.3: Qualitative comparison for Java-C++ translation between CodeT5 and RL4CNCT. i) CodeT5 generates a non-standard data type `subset` without defining it or importing it. RL4CNCT generates the correct datatype. ii) CodeT5 generates the wrong tokens to return out of a function, RL4CNCT is able to generate the correct C++ syntax.

Source Java	Translated C++	
	CodeT5	RL4CNCT
<pre>import java . util . * ; class GFG { static void printVector (ArrayList < Integer > arr) { if (arr . size () != 1) { for (int i = 0 ; i < arr . size () ; i ++) { System . out . print (arr . get (i) + " ") ; } System . out . println () ; } } static void findWays (ArrayList < Integer > arr , int i , int n) { if (n == 0) printVector (arr) ; for (int j = i ; j <= n ; j ++) { arr . add (j) ; findWays(arr, j , n-j); arr.remove(arr.size()-1); } } }</pre>	<pre>#include <bits/stdc++.h> using namespace std ; void printVector (subset < int , int > arr) { if (arr . size () != 1) { for (int i = 0 ; i < arr . size () ; i ++) { cout << arr [i] << "_" ; } cout << endl ; } } void findWays (vector < int > & arr , int i , int n) { if (n == 0) printVector (arr) ; for (int j = i ; j <= n ; j ++) { arr . push_back (j) ; findWays (arr , j , n - j) ; arr . pop_back () ; } }</pre>	<pre>#include <bits/stdc++.h> using namespace std ; void printVector (vector < int > arr) { if (arr . size () != 1) { for (int i = 0 ; i < arr . size () ; i ++) { cout << arr [i] << "_" ; } cout << endl ; } } void findWays (vector < int > arr , int i , int n) { if (n == 0) printVector (arr) ; for (int j = i ; j <= n ; j ++) { arr . push_back (j) ; findWays (arr , j , n - j) ; arr . pop_back () ; } }</pre>
<pre>import java . util . * ; import java . lang . * ; class GFG { static int minimumIncrement (int arr [] , int N) { if (N % 2 != 0) { System . out . println ("-1 ") ; System . exit (0) ; } int cntEven = 0 ; int cntOdd = 0 ; for (int i = 0 ; i < N ; i ++) { if (arr [i] % 2 == 0) { cntEven += 1 ; } } cntOdd = N - cntEven ; return Math . abs (cntEven - cntOdd) / 2 ; } }</pre>	<pre>#include <bits/stdc++.h> using namespace std ; int minimumIncrement (int arr [] , int N) { if (N % 2 != 0) { cout << "-1" ; c exit (0) ; } int cntEven = 0 ; int cntOdd = 0 ; for (int i = 0 ; i < N ; i ++) { if (arr [i] % 2 == 0) { cntEven += 1 ; } } cntOdd = N - cntEven ; return abs (cntEven - cntOdd) / 2 ; }</pre>	<pre>#include <bits/stdc++.h> using namespace std ; int minimumIncrement (int arr [] , int N) { if (N % 2 != 0) { cout << "-1" << endl ; return 0 ; } int cntEven = 0 ; int cntOdd = 0 ; for (int i = 0 ; i < N ; i ++) { if (arr [i] % 2 == 0) { cntEven += 1 ; } } cntOdd = N - cntEven ; return abs (cntEven - cntOdd) / 2 ; }</pre>

Table 5.4: Qualitative comparison for C++-Python translation between CodeT5 and RL4CNCT. i) CodeT5 generates an extra closing parenthesis. ii) CodeT5 generates the wrong syntax for function declaration, RL4CNCT is able to generate the correct Python syntax using `def` keyword.

Source C++	Translated Python	
	CodeT5	RL4CNCT
<pre>#include <bits/stdc++.h> using namespace std ; int func (int N , int P) { int sumUptoN = (N * (N + 1) / 2) ; int sumOfMultiplesOfP ; if (N < P) { return sumUptoN ; } else if ((N / P) == 1) { return sumUptoN - P + 1 ; } sumOfMultiplesOfP = ((N / P) * (2 * P + (N / P - 1) * P)) / 2 ; return (sumUptoN + func (N / P , P) - sumOfMultiplesOfP) ; } int main () { int N = 10 , P = 5 ; cout << func (N , P) << " " ; return 0 ; }</pre>	<pre>def func (N , P) : sumUptoN = (N * (N + 1) // 2) sumOfMultiplesOfP = 0 if (N < P) : return sumUptoN elif ((N / P) == 1) : return sumUptoN - P + 1 sumOfMultiplesOfP = ((N / P) * (2 * P + (N / P - 1) * P)) // 2 return (sumUptoN + func (N // P , P) - sumOfMultiplesOfP) N = 10 P = 5 print (func (N , P))</pre>	<pre>def func (N , P) : sumUptoN = int (N * (N + 1) / 2) sumOfMultiplesOfP = 0 if (N < P) : return sumUptoN elif ((N / P) == 1) : return sumUptoN - P + 1 sumOfMultiplesOfP = ((N / P) * (2 * P + (N / P - 1) * P)) // 2 return (sumUptoN + func (N // P , P) - sumOfMultiplesOfP) if __name__ == '__main__' : N = 10 P = 5 print (func (N , P))</pre>
<pre>#include <bits/stdc++.h> using namespace std ; int findWays (int N , int dp []) { if (N == 0) { return 1 ; } if (dp [N] != -1) { return dp [N] ; } int cnt = 0 ; for (int i = 1 ; i <= 6 ; i ++) { if (N - i >= 0) { cnt = cnt + findWays (N - i , dp) ; } } return dp [N] = cnt ; } int main () { int N = 4 ; int dp [N + 1] ; memset (dp , -1 , sizeof (dp)) ; cout << findWays (N , dp) ; return 0 ; }</pre>	<pre>func findWays (N , dp) : if (N == 0) : return 1 if (dp [N] != -1) : return dp [N] cnt = 0 for i in range (1 , 7) : if (N - i >= 0) : cnt = cnt + findWays (N - i , dp) dp [N] = cnt return dp [N] N = 4 dp = [-1] * (N + 1) print (findWays (N , dp))</pre>	<pre>def findWays (N , dp) : if (N == 0) : return 1 if (dp [N] != -1) : return dp [N] cnt = 0 for i in range (1 , 7) : if (N - i >= 0) : cnt = cnt + findWays (N - i , dp) dp [N] = cnt return dp [N] N = 4 dp = [-1] * (N + 1) print (findWays (N , dp))</pre>

Chapter 6

Public demo website

In order to enable the research community to engage with our work, we also developed a website¹. The website demonstrates the translation capabilities of models trained using *XLC_oST*. In this chapter we describe details of the technology stack we used to develop the website, we describe how the website functions, we describe some of the challenges we faced during the development.

6.1 The Django framework

We employ the Django² framework to build our website. There are a couple of reasons why we chose to go with Django as our framework of choice. The first reason is that Django takes care of a lot of the complications associated with traditional web frameworks like React and NodeJS which have a significant learning curve associated with them. Django however, is a Python based web framework that uses HTML and CSS which makes it easy to use and fast to deploy. Django is open-source and is very well maintained and supported by the community. All of our models have been developed using PyTorch and the accessory code used to derive inferences from the models is also written in Python. Since Django is a Python based web framework it felt like a natural choice to go with it because it would play well with all of the we already had in place to support our models.

¹<http://code.cs.vt.edu>

²<https://www.djangoproject.com>

6.2 Overall site design

Figure 6.1 shows what the demo website looks like when a user first navigates to it. The website layout is fairly simple to make it user friendly. There are two drop-downs to select the source and target languages. The source language drop down selects the coding language that the user wants to translate *from*. The target language drop down selects the coding language that the user wants to translate *to*. Below each of the drop downs is the corresponding text box for the source and target languages. Underneath the source language text box are buttons for loading examples. Underneath the target language box is another text box for the compilation output. There are 3 other buttons labelled as per the functions they perform: TRANSLATE, CLEAR and COMPILE.

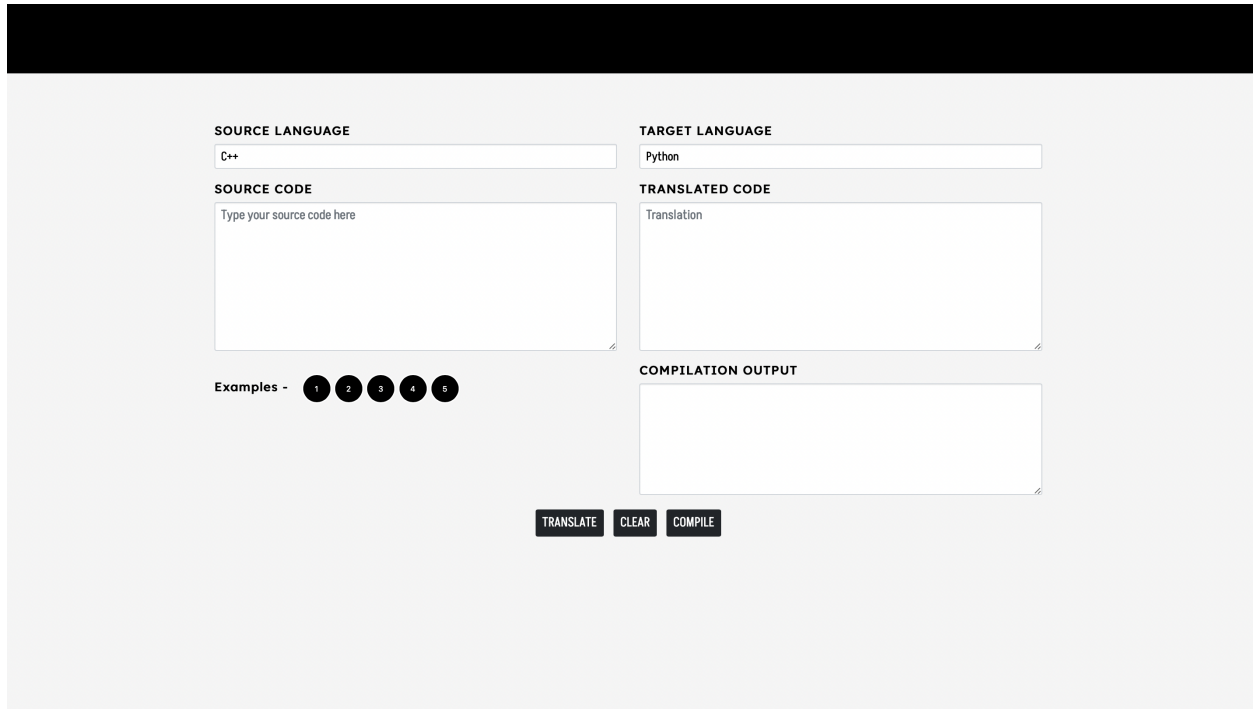
The image shows a web interface for a translation demo. At the top, there is a solid black horizontal bar. Below this, the interface is divided into two main columns. The left column has a 'SOURCE LANGUAGE' dropdown menu with 'C++' selected, followed by a 'SOURCE CODE' text area with the placeholder text 'Type your source code here'. Below the text area is a row of five circular buttons labeled 'Examples - 1', '2', '3', '4', and '5'. The right column has a 'TARGET LANGUAGE' dropdown menu with 'Python' selected, followed by a 'TRANSLATED CODE' text area with the placeholder text 'Translation'. Below this is a 'COMPILATION OUTPUT' text area. At the bottom center, there are three buttons: 'TRANSLATE', 'CLEAR', and 'COMPILE'.

Figure 6.1: Home page of the Translation demo website. This page is displayed when a user first visits the website.

6.3 Functional details

With the layout of the website in mind let's go through the functioning of the website using examples and figures. After landing on the home page, the user selects the source and target languages from the drop downs. This selection of language pairs decides which model will

be loaded in the back end to fulfil the users request. For example if the user selects C++ as the source language and Python as the target language, the model that will be loaded in the back end will be corresponding to the C++-Python translation. How this looks at the front-end is shown in Figure 6.2.

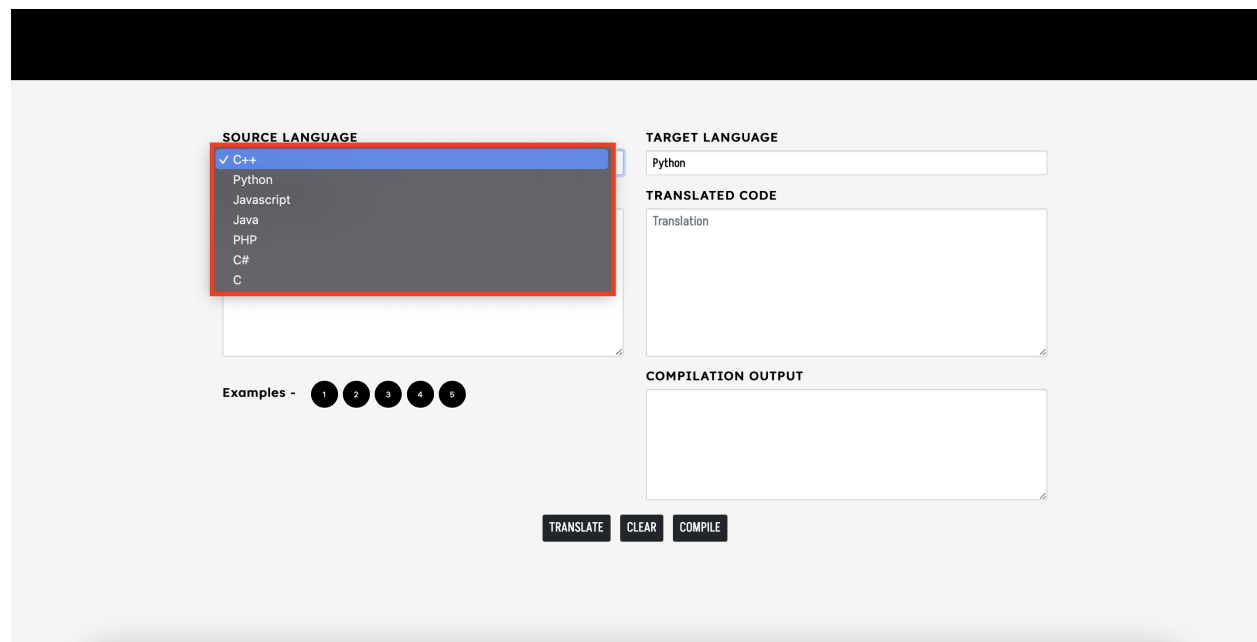
The screenshot shows a web application interface for language translation. At the top, there is a black header bar. Below it, the interface is divided into two main columns. The left column has a 'SOURCE LANGUAGE' section with a dropdown menu currently showing 'C++' selected. Other options visible in the dropdown are Python, Javascript, Java, PHP, C#, and C. Below the dropdown is a large text area for 'SOURCE CODE'. At the bottom of the left column, there are five circular buttons labeled 'Examples - 1', '2', '3', '4', and '5'. The right column has a 'TARGET LANGUAGE' section with a dropdown menu showing 'Python'. Below this is a 'TRANSLATED CODE' section with a text area labeled 'Translation'. At the bottom of the right column is a 'COMPILATION OUTPUT' section with a text area. At the very bottom of the interface, there are three buttons: 'TRANSLATE', 'CLEAR', and 'COMPILE'.

Figure 6.2: Drop down menus to choose source and target languages. The user can select between 42 language pairs to translate based on the source and target language choice.

Once the selection has been made, the user can either enter their own code in the text box on the right labelled “SOURCE CODE” or click on one of the example buttons on the bottom to load a pre-selected piece of code as shown in Figure 6.3.

After entering the desired code into the source box, the user can click the TRANSLATE button. The model takes a few seconds to generate the translation and the final output is displayed in the box in the right labelled “TARGET CODE”. This is highlighted in Figures 6.4 and 6.5.

Once the TRANSLATE button has been clicked, a sequence of events takes place at the back end. These are presented in Figure 6.6, which captures the detailed back end workflow. It can be observed from the figure that when the user clicks the TRANSLATE button, an AJAX request is made to the back end model prediction function contained in the `predict.py` file. An AJAX request, also known as Asynchronous Javascript and XML, is a special kind of request that can asynchronously exchange data between the front and the back end without needing to reload the whole page. This is specifically useful for our case since we can load the HTML page when the user first logs in and then asynchronously submit

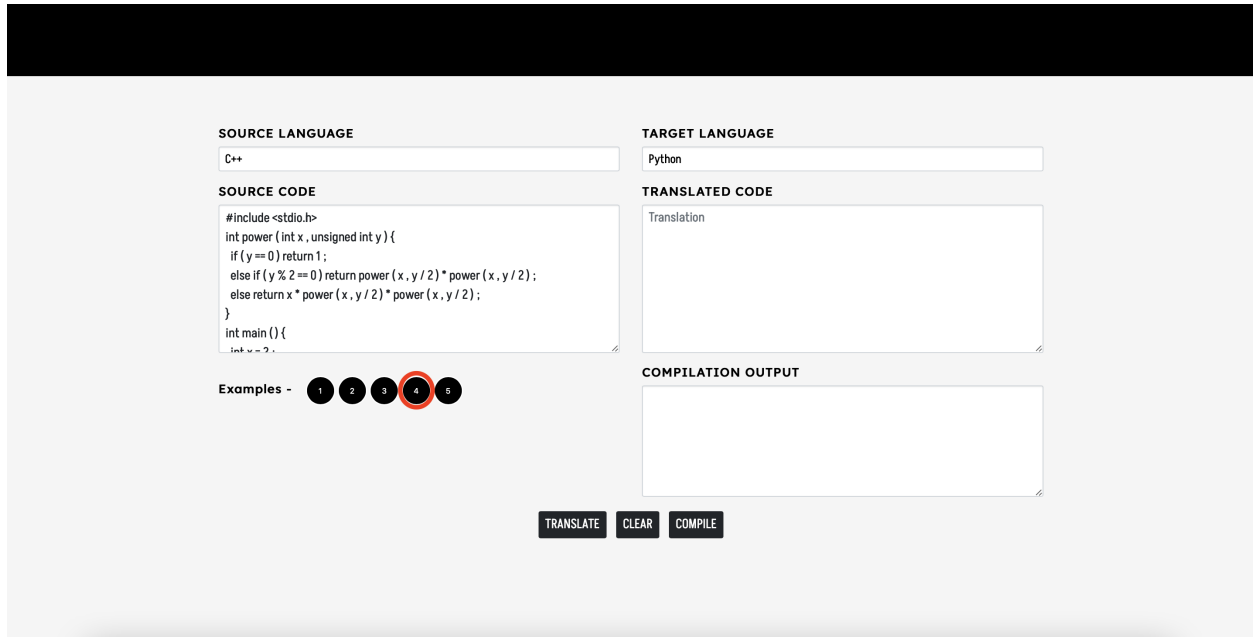


Figure 6.3: Loading source code through pre-existing examples. Clicking on any of the buttons marked red loads a pre-set example code in the source language selected. Users can also type their own input code into the source code text box.

requests to the model for translation on button click. Once the model generates a response it is again relayed to the front-end and.

In addition to generating the translation, the website also allows the user to check if the generated code compiles or not. Once the translation has been generated, the user can click on the “COMPILE” button to compile the code. The “COMPILATION OUTPUT” text box displays a success message if the code successfully compiles or displays the compilation error. The language specific compiler is chosen at the back end based on the target language selected by the user in the beginning. Figures 6.7 and 6.8 present what the compilation looks like at the front end.

6.4 Challenges and learnings

There were multiple challenges we faced during the development of this demo website. Initially we were unsure of what kind of frame work would be the best to be able to implement all our requirements. We finally ended up using Django which worked very well for us. We did not have any experience with deploying and inferencing models on web applications. It took a lot of trial and error to be able to learn how to effectively serve the model keeping

SOURCE LANGUAGE
C++

TARGET LANGUAGE
Python

SOURCE CODE

```
#include <stdio.h>
int power (int x, unsigned int y) {
    if (y == 0) return 1;
    else if (y % 2 == 0) return power (x, y / 2) * power (x, y / 2);
    else return x * power (x, y / 2) * power (x, y / 2);
}

int main () {
    int x = 2;
```

TRANSLATED CODE

Translation

COMPILATION OUTPUT

Examples - 1 2 3 4 5

TRANSLATE **CLEAR** **COMPILE**

Figure 6.4: Clicking the translate button activates the model which generates code in selected target language. The generated translation is displayed in the translated code text box.

resource constraints like computing power in mind. Since deep learning models are so large, initialising them and loading their weights repeatedly takes a lot of processing time. This can get annoying for the user since they may just end up waiting for the website to load while the model initialises. In order to get around this, we had to learn how to cache the model, so that repeated instantiations do not take place and the website performs well. We also had to learn how to build the pipeline that will capture text from the user, pre-process it, provide it to the model, capture the model output and then finally relay it back to the front end. Before getting into developing and deploying this website, we did not have a lot of experience with web development. Through the process of developing this website we have learned a great detail about how to design and develop a pipeline that can deploy deep learning models effectively. We were able to incorporate all the different components of our research and present them with the help of this website. Being able to conduct deep learning research, development and then finally productionizing the work has enabled us to bring this work full circle and it has indeed been an invaluable experience.

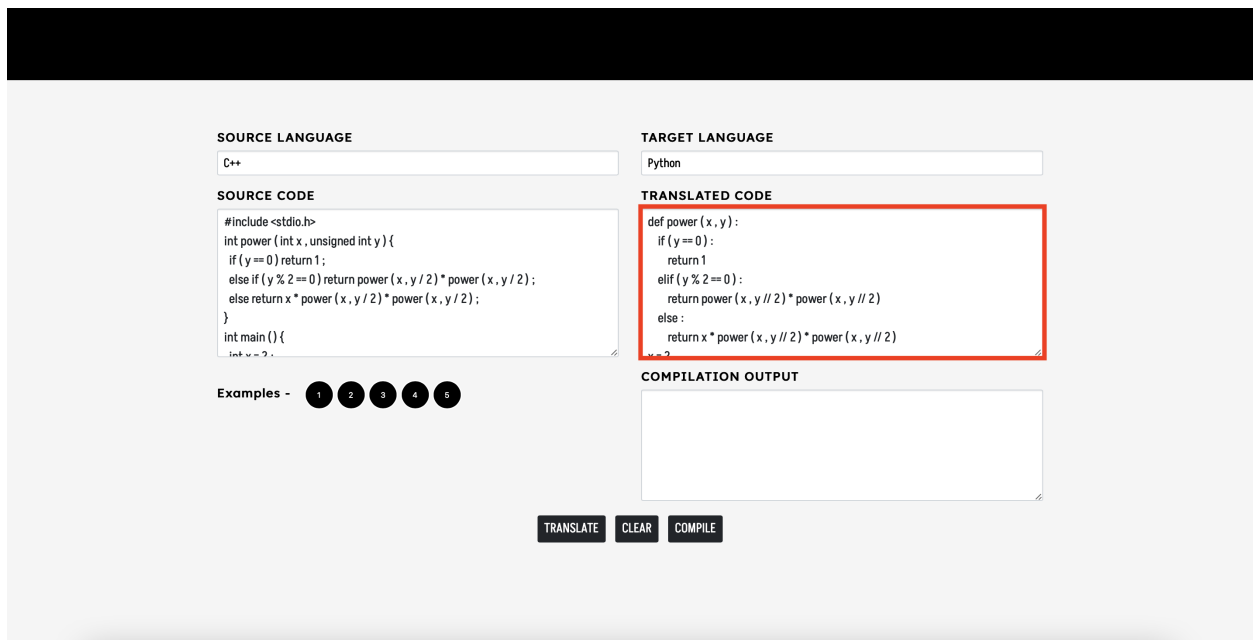


Figure 6.5: The generated target language code. The figure shows the Python output generated by the model when C++ is selected as the source language and an example code is provided as input.

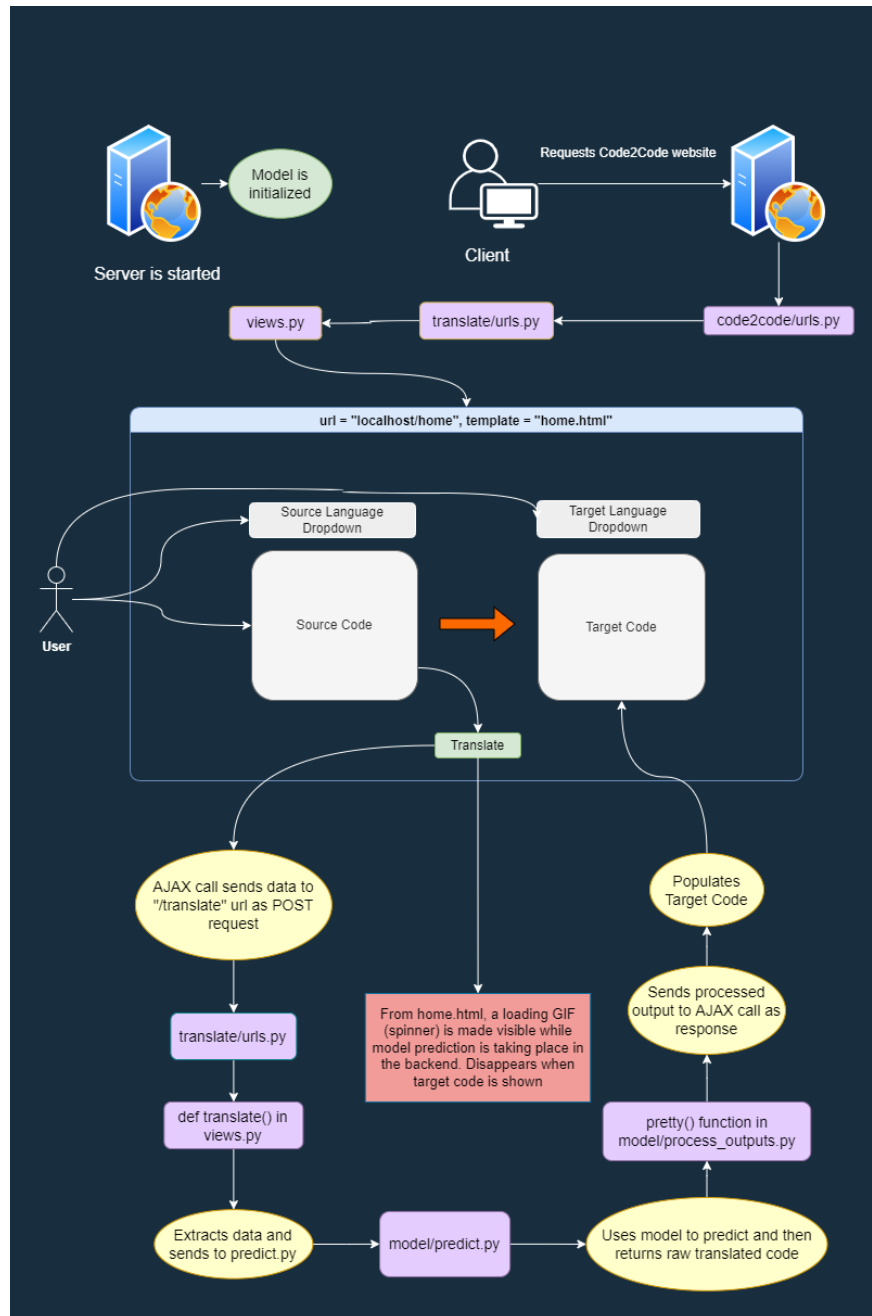


Figure 6.6: A detailed workflow of demo website that shows how the back end works when the TRANSLATE button is clicked. The input code string is parsed from the front-end text box and relayed to the model using an AJAX request. The model processes the input and the generated output is relayed back to the front-end target code text box.

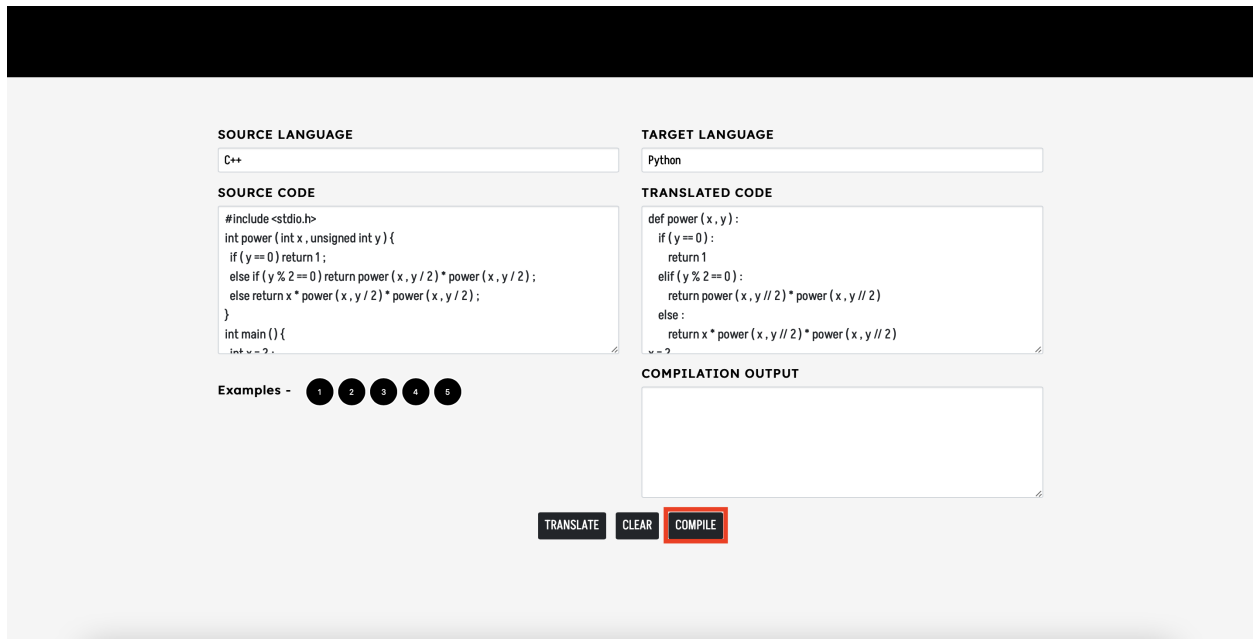


Figure 6.7: After the translated code is generated, the user can also check of the generated code compiles by clicking on the compile button. The target language compiler is automatically selected, which then checks for syntax errors in the generated code. Errors, if they exist, are displayed in compilation output text box. If no errors exist, a success message is displayed.

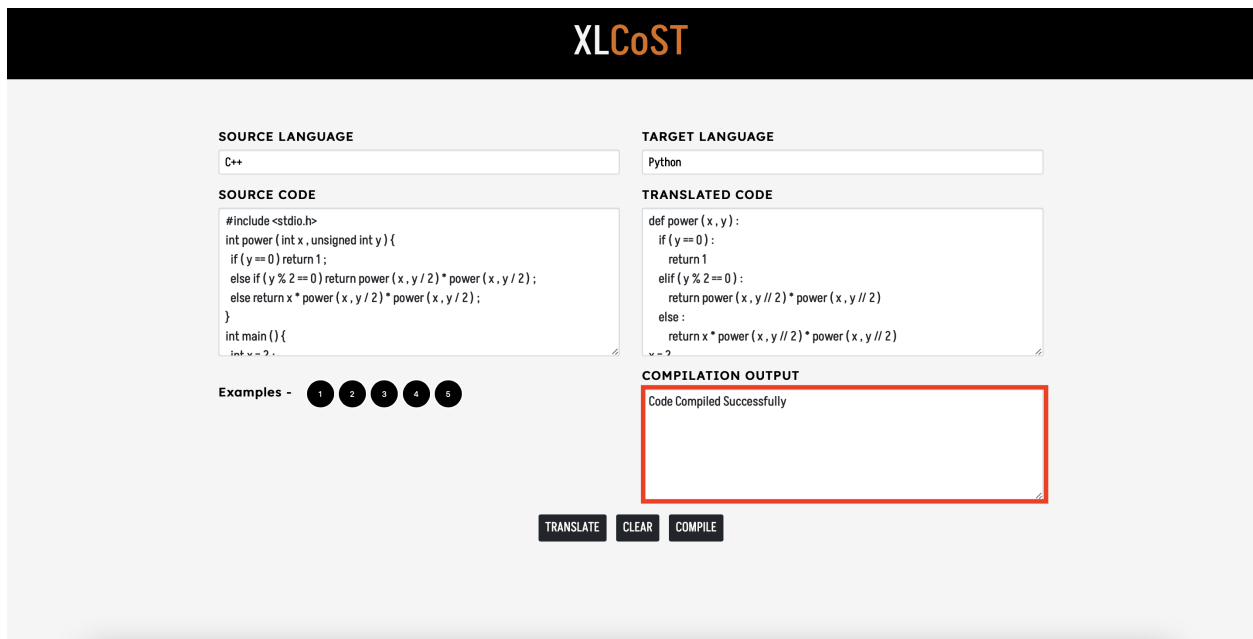


Figure 6.8: The compilation output text box indicates that the model generated translation compiled successfully.

Chapter 7

Conclusion

In this work, we sought to contribute to the state of the art of deep learning for software engineering tasks by providing to the community more effective tools in the form of data sets and evaluation metrics. Additionally we also sought to improve the quality of code generated using deep learning methods in terms of its usability. To that end, we curated a code dataset named *XLCoST* which stands for Cross-Lingual Code Snippet data set. The data set contains finely-aligned parallel data at the snippet and program levels in order to facilitate more deep learning research in the code domain. We showed potential applications of this data set for multiple different tasks including Code Translation, Code Summarization, Code Synthesis and Code Search. We evaluated the performance of this data set on multiple baselines presented insights that could be gleaned for future research.

In addition to the usual metrics used for evaluating code related tasks like BLEU and CodeBLEU, we talked about the importance of compilability of code generated using deep learning methods for tasks like translation and code generation. We presented a study of the compilability of generated hypotheses using our data set. Additionally we showed how using the *XLCoST* data set can lead to improvements in the compilability of out-of-distribution data using ablation studies.

Motivated by our study of compilation we also developed a deep reinforcement learning based approach to improve the quality of code translation. We incorporated compiler feedback as a measure of correctness in the reward function, as well as signals from the abstract syntax trees and data flow graphs to guide the model towards generating syntactically correct code. We presented the results of experiments which showed that our method was able to outperform the baseline.

Finally, we also developed a public portal where the models we developed for Code Translation have been deployed, in order to give the research community the opportunity to be

able to directly interact with our work.

We acknowledge that compilability on its own is limited in being able to capture the generation quality and execution based metrics that evaluate logical correctness are very much needed. Even if generated code compiles correctly it may not necessarily mean that it performs its intended function correctly. We attempted to tackle this problem and discussed the many challenges we faced in finding an effective solution to them for the current scope of work. Creating a pipeline that supports execution based metrics that would provide richer insights into the quality of generation remains as a future work to be explored. In the past two years many new data sets have been presented that also include test cases from evaluating model performance on various software engineering tasks. APPS [50], HumanEval [51] and MBXP [52] are some of the most recently released among them. Leveraging these data sets in our model development workflow and using the signal from execution of test cases remains a promising future work direction that we would very much like to explore. Just as we incorporated compiler feedback in our reinforcement learning model, we would also like to include feedback from the execution of test cases using some of the new data sets that have been introduced recently. This will provide a much stronger feedback signal to the reinforcement learning model. In addition we would also like to incorporate test case base evaluation of model performance in our experiments to ascertain the true usability of code generated using deep learning methods.

Bibliography

- [1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” Advances in neural information processing systems, vol. 30, 2017.
- [2] J. D. M.-W. C. Kenton and L. K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” in Proceedings of NAACL-HLT, 2019, pp. 4171–4186.
- [3] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, “Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension,” in Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, 2020, pp. 7871–7880.
- [4] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “Roberta: A robustly optimized bert pretraining approach,” arXiv preprint arXiv:1907.11692, 2019.
- [5] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, “Language models are few-shot learners,” in Advances in Neural Information Processing Systems, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: <https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfc4967418bfb8ac142f64a-Paper.pdf>
- [6] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, “A survey of machine learning for big code and naturalness,” ACM Computing Surveys (CSUR), vol. 51, no. 4, pp. 1–37, 2018.

- [7] Y. Yang, X. Xia, D. Lo, and J. Grundy, “A survey on deep learning for software engineering,” ACM Computing Surveys (CSUR).
- [8] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, “Productivity assessment of neural code completion,” in Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming, 2022, pp. 21–29.
- [9] J. D. Weisz, M. Muller, S. Houde, J. Richards, S. I. Ross, F. Martinez, M. Agarwal, and K. Talamadupula, “Perfection not required? human-ai partnerships in code translation,” in 26th International Conference on Intelligent User Interfaces, 2021, pp. 402–412.
- [10] P. Vaithilingam, T. Zhang, and E. L. Glassman, “Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models,” in CHI Conference on Human Factors in Computing Systems Extended Abstracts, 2022, pp. 1–7.
- [11] M. Barenkamp, J. Rebstadt, and O. Thomas, “Applications of ai in classical software engineering,” AI Perspectives, vol. 2, no. 1, pp. 1–15, 2020.
- [12] F. Chicano, F. Luna, A. J. Nebro, and E. Alba, “Using multi-objective metaheuristics to solve the software project scheduling problem,” in Proceedings of the 13th annual conference on Genetic and evolutionary computation, 2011, pp. 1915–1922.
- [13] C. Stylianou and A. S. Andreou, “Investigating the impact of developer productivity, task interdependence type and communication overhead in a multi-objective optimization approach for software project planning,” Advances in Engineering Software, vol. 98, pp. 79–96, 2016.
- [14] L. Perreault, S. Berardinelli, C. Izurieta, and J. Sheppard, “Using classifiers for software defect detection,” in 26th International Conference on Software Engineering and Data Engineering, 2017, pp. 2–4.
- [15] Y. Li, S. Wang, and T. N. Nguyen, “Dlfix: Context-based code transformation learning for automated program repair,” in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 602–614.
- [16] T. Durieux and M. Monperrus, “Dynamoth: dynamic code synthesis for automatic program repair,” in Proceedings of the 11th International Workshop on Automation of Software Test, 2016, pp. 85–91.

- [17] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, “Lexical statistical machine translation for language migration,” in Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, 2013, pp. 651–654.
- [18] K. Aggarwal, M. Salameh, and A. Hindle, “Using machine translation for converting python 2 to python 3 code,” PeerJ PrePrints, Tech. Rep., 2015.
- [19] P. W. McBurney and C. McMillan, “Automatic documentation generation via source code summarization of method context,” in Proceedings of the 22nd International Conference on Program Comprehension, 2014, pp. 279–290.
- [20] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, “Improving automatic source code summarization via deep reinforcement learning,” in Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, 2018, pp. 397–407.
- [21] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, “Summarizing source code using a neural attention model,” in Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2016, pp. 2073–2083.
- [22] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, “A transformer-based approach for source code summarization,” arXiv preprint arXiv:2005.00653, 2020.
- [23] Y. Zhu, Y. Zhang, H. Yang, and F. Wang, “Gancoder: An automatic natural language-to-programming language translation approach based on gan,” in CCF International Conference on Natural Language Processing and Chinese Computing. Springer, 2019, pp. 529–539.
- [24] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, and L. Zhang, “A grammar-based structural cnn decoder for code generation,” in Proceedings of the AAAI conference on artificial intelligence, vol. 33, no. 01, 2019, pp. 7055–7062.
- [25] K. T. Stolee, S. Elbaum, and D. Dobos, “Solving the search for source code,” ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 23, no. 3, pp. 1–45, 2014.
- [26] X. Gu, H. Zhang, and S. Kim, “Deep code search,” in 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). IEEE, 2018, pp. 933–944.
- [27] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. Yu, “Multi-modal attention network learning for semantic source code retrieval,” in 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2019, pp. 13–25.

- [28] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang et al., “Codebert: A pre-trained model for programming and natural languages,” arXiv preprint arXiv:2002.08155, 2020.
- [29] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu et al., “Graphcodebert: Pre-training code representations with data flow,” arXiv preprint arXiv:2009.08366, 2020.
- [30] D. Zügner, T. Kirschstein, M. Catasta, J. Leskovec, and S. Günnemann, “Language-agnostic representation learning of source code from structure and context,” in International Conference on Learning Representations (ICLR), 2021.
- [31] B. Roziere, M.-A. Lachaux, M. Szafraniec, and G. Lample, “Dobf: A deobfuscation pre-training objective for programming languages,” arXiv preprint arXiv:2102.07492, 2021.
- [32] U. Alon, S. Brody, O. Levy, and E. Yahav, “code2seq: Generating sequences from structured representations of code,” in International Conference on Learning Representations, 2019. [Online]. Available: <https://openreview.net/forum?id=H1gKY09tX>
- [33] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “Code2vec: Learning distributed representations of code,” Proc. ACM Program. Lang., vol. 3, no. POPL, pp. 40:1–40:29, Jan. 2019. [Online]. Available: <http://doi.acm.org/10.1145/3290353>
- [34] W. Yue, W. Weishi, J. Shafiq, and C. H. Steven, “Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” in Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, 2021.
- [35] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, “Unified pre-training for program understanding and generation,” in Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Online: Association for Computational Linguistics, Jun. 2021, pp. 2655–2668. [Online]. Available: <https://aclanthology.org/2021.naacl-main.211>
- [36] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. Hoi, “Coderl: Mastering code generation through pretrained models and deep reinforcement learning,” arXiv preprint arXiv:2207.01780, 2022.
- [37] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou,

- N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, “Codexglue: A machine learning benchmark dataset for code understanding and generation,” CoRR, vol. abs/2102.04664, 2021.
- [38] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “Code-searchnet challenge: Evaluating the state of semantic code search,” arXiv preprint arXiv:1909.09436, 2019.
- [39] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker et al., “Project codenet: A large-scale ai for code dataset for learning a diversity of coding tasks,” arXiv preprint arXiv:2105.12655, vol. 1035, 2021.
- [40] W. U. Ahmad, M. G. R. Tushar, S. Chakraborty, and K.-W. Chang, “Avatar: A parallel corpus for java-python program translation,” arXiv preprint arXiv:2108.11590, 2021.
- [41] M. Zhu, K. Suresh, and C. K. Reddy, “Multilingual code snippets training for program translation,” 2022.
- [42] M. Zhu, A. Jain, K. Suresh, R. Ravindran, S. Tipirneni, and C. K. Reddy, “Xlcost: A benchmark dataset for cross-lingual code intelligence,” arXiv preprint arXiv:2206.08474, 2022.
- [43] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. L. Traon, “Facoy: a code-to-code search engine,” in Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 946–957.
- [44] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “Bleu: a method for automatic evaluation of machine translation,” in Proceedings of the 40th annual meeting of the Association for Computational Linguistics, 2002, pp. 311–318.
- [45] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, “Codebleu: a method for automatic evaluation of code synthesis,” arXiv preprint arXiv:2009.10297, 2020.
- [46] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” in Proc. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2019.
- [47] Y. Liu, J. Gu, N. Goyal, X. Li, S. Edunov, M. Ghazvininejad, M. Lewis, and L. Zettlemoyer, “Multilingual denoising pre-training for neural machine translation,”

- Transactions of the Association for Computational Linguistics, vol. 8, pp. 726–742, 2020.
- [48] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, “Exploring the limits of transfer learning with a unified text-to-text transformer,” Journal of Machine Learning Research, vol. 21, no. 140, pp. 1–67, 2020.
- [49] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” 2017. [Online]. Available: <https://arxiv.org/abs/1707.06347>
- [50] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song et al., “Measuring coding challenge competence with apps,” arXiv preprint arXiv:2105.09938, 2021.
- [51] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman et al., “Evaluating large language models trained on code,” arXiv preprint arXiv:2107.03374, 2021.
- [52] B. Athiwaratkun, S. K. Gouda, Z. Wang, X. Li, Y. Tian, M. Tan, W. U. Ahmad, S. Wang, Q. Sun, M. Shang et al., “Multi-lingual evaluation of code generation models,” arXiv preprint arXiv:2210.14868, 2022.