An Investigation Into Supervision for Seq2Seq Techniques for Natural Language to Code Translation

Meheresh Yeditha

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2022

Reading Committee:
Shane Steinert-Threlkeld, Chair
Fei Xia

Program Authorized to Offer Degree:
Department of Linguistics

University of Washington

Abstract

An Investigation Into Supervision for Seq2Seq Techniques for Natural Language to Code Translation

Meheresh Yeditha

Chair of the Supervisory Committee:
Shane Steinert-Threlkeld
Department of Linguistics

This thesis examines the role of supervised data using small-scale datasets for the natural language to code task. The primary angles of inquiry are from analyzing the balance between unsupervised learning and supervised learning, as well as experimenting with several training techniques. To do so, two publicly available datasets were utilized, the CodeSearchNet task for English documentation to Python code, and the Mostly Basic Python Problems (MBPP) dataset, using the mBART seq2seq framework for running experiments. The best performing models pretrained on the full CodeSearchNet dataset, and finetuned on the MBPP dataset. Several avenues for future inquiry and effective experimentation were discovered and solidifed, including lample masking, creation of more datasets fitting the NL2C paradigm, and size and division of datasets. Finetuning is significantly more important than the pretraining phase, although both are crucial when using the seq2seq framework. Overall, this thesis solidifies the utility of seq2seq frameworks for the NL2C task, and the promise of transfer learning and inquiries for this task going forward.

TABLE OF CONTENTS

LIST OF TABLES

Chapter 1

INTRODUCTION

Eminent computer science pioneer Edsger Dijkstra once published an infamous polemic entitled "On the foolishness of 'natural language programming'" [5]. In the document, Dijkstra dismisses the burgeoning field of natural language programming, or writing machine code using natural languages like English, as impractical given the formal nature and symbols of machine code as opposed to the imprecision and broad strokes of natural language. He stated, "We would need all the intellect in the world to get the interface narrow enough to be usable, and, in view of the history of mankind, it may not be overly pessimistic to guess that to do the job well enough would require again a few thousand years" [5].

At the time, this was not a controversial statement to make. Failed attempts to create a rule-based subset of the English language powerful enough to use as a general-purpose programming language littered the computer science landscape. Unsupervised and supervised attempts to infer and learn rules through primitive neural networks and machine learning algorithms foundered in the background from lack of computing resources [16]. Despite this, several attempts were made through the 2000s to get close to this vision [16]. General-purpose languages from the 1960s like COBOL mimicked imperative English statements in their syntax, while others like Attempto Controlled English [6], which was released in the 1990s, tried a different approach by using a subset of standard English in conjunction with a set of rules as a knowledge representation and query language. Several esoteric programming languages, including Shakespeare, a language that emulates the dialogue in a Shakespearean play, were also created to this end [7]. But the lack of computing power and rule-based techniques of these approaches resulted in a failure to scale appropriately, with most of the English-appearing "code" in these languages merely syntactic sugar for higher-level program-

ming constructs [10].

However, explosive growth in deep learning in the mid-2010s has transformed the space of natural language programming from limited pipe-dream to increasingly tenable. Unsupervised techniques proved useful in creating efficient transpilers from one programming language to another, such as Facebook's Transcoder software [15]. OpenAI's GPT-2 and GPT-3 [20], enormous language models trained on unprecedented amounts of data, showed some success in translating plain-English specifications to Python code, as well as translating plain English requests into simple web applications. Common deep learning architectures like transformers and frameworks that were used to upgrade and enhance machine translation were applied to natural language programming, resulting in increasingly viable machine code and an increasingly vibrant and promising research space [10].

This thesis focuses primarily on the generation of general-purpose machine code in languages like Python, C, and Java from documentation written in English speech, a subset of natural language programming often referred to as NL2C. The best performing models in recent years have been those utilizing a very large corpus of data to approach this problem from an unsupervised language modeling, with documentation coming in the form of embedded comments or descriptive function names [20]. Other models have seen success on more limited datasets with a supervised learning perspective using seq2seq models translating to intermediate representations such as abstract syntax trees (ASTs) or graph-based program representations [24]. This thesis seeks to explore the middle ground between these two approaches, asking the following questions:

1. How well can a NL2C model perform by utilizing multiple publicly-available datasets in a transfer learning context?

2. How well does this model perform when we mix unsupervised and supervised learning techniques?

This thesis will first introduce the relevant concepts through a structured literature review

in Chapter 2. This will go over relevant concepts such as transformers, seq2seq models, transfer learning, abstract syntax trees, and other programming language-related concepts. In Chapter 3, this thesis will discuss recent advances in the field, as well as the specifics of current state of the art techniques in this subfield. In Chapter 4, the setup, tools, and methodology crucial to running these experiments, as well as the experiments performed, will be discussed in-depth. Results and discussion of these experiments will be presented in Chapter 5, wrapping up with a conclusion and discussion on future work in Chapter 6.

Chapter 2
RECENT ADVANCES

This chapter will discuss the current state of the field of natural language-to-code translation by presenting a number of crucial papers and concepts, as well as the current state of the art approach and results.

2.1 Supervised Learning with Seq2Seq

There are a number of recent papers in the area of semantic parsing, especially since the introduction of the seq2seq architecture in 2015 [19]. This framework repeatedly shows up throughout the papers in this section. The two most popular categories in the area of semantic parsing pertaining to code output are text to SQL, and text to bash. The purpose of this thesis will be to translate to general code, but it is suspected that approaches taken to model these two command/query-based languages will prove to be useful for the general task as well.

One of the first papers to tackle the text to bash task head-on is Lin et al. [24]. This paper's approach was implemented into the "Tellina" system, and is the subject of a shared task at the NeurIPS 2020 conference[1]. The input for this paper was plain English questions from Stack Exchange, which are then translated to corresponding Bash code according to each question's paired answer. First, file names and times and other entites are abstracted out. A recurrent neural network encoder-decoder model is then translated to a template program, with slots for entities identified in the first step. Finally, the slots are replaced with the entities obtained through k-nearest neighbors. The model obtained significantly higher translation accuracies than the code retrieval baseline, improving F1 score by 24.8

[1] https://nlc2cmd.us-east.mybluemix.net/

across different accuracy metrics. The system was then given to students and frequent Bash users for evaluation as a day-to-day command line assistant, indicating that a majority of participants wanted to use Tellina in the future and that even the system's partially correct responses were useful in constructing a desirable Bash command.

Another major paper by Neubig and Yin [25] takes a slightly different tack to parsing natural language descriptions into general-purpose source code (and hews closest to the task I am interested in for this thesis). Similar to Rabinovich et al. [14], this paper translates from sequences to ASTs to capture the syntax of the programming language. The data used is the Hearthstone dataset mentioned in Rabinovich et al. [14], a Django dataset comprised of lines of code augmented with a manually annotated natural language description, and the IFTTT dataset [13] that provides domain-specific languages with natural language annotations (although the approach is described as being programming language-agnostic). The approach uses a grammar model to generate the AST, which is comprised of two actions at a high level: applying a production rule to a derivation tree, and populating a terminal node. The probabilities to generate trees from this grammar model are derived using an encoder-decoder architecture on the source data. The encoder uses a bidirectional LSTM architecture to embed context like in previous papers. The decoder also uses an LSTM, but is augmented with three additional components to reflect the structure of the AST: Two action embedding matrices corresponding to embedding vectors for an action, a context vector, and an additional neural connection corresponding to passing information from an AST's parent to child. The accuracy of the system when run with the data produces substantially higher BLEU scores than previous benchmarks for Python code generation tasks, achieving scores of 75.8 on the Hearthstone dataset and 84.5 on the Django dataset.

A paper by Brockschmidt et al. [3] introduces another new decoding framework in the field of semantic parsing. This paper tackles problems in previous papers like Rabinovich et al. [14] and Neubig and Yin [25] by aiming to increase semantic relevance (i.e. consistent usage of variables). It does this by specifically focusing on the task of filling in code with holes, while also keeping in mind the applicability to the subfield of natural language to code

generation. The paper aims to construct the AST sequentially with a rule-based approach using LSTMs as in previous papers. The decoder builds on Rabinovich et al. [14] and Neubig and Yin [25] by using attribute grammars from compiler theory, making each node linked with two new vector representations for inherited and synthesized information, or attributes. This is referred to as a neural attribution grammar (NAG). Additionally, an algorithm is introduced to turn a partial AST with this additional information into a graph, adding edges between nodes to encode attribute relationships between adjacent nodes and other context, as well as a neural network that learns from this graph. The paper also treats picking appropriate production rules of a tree and variables as a classification problem. This involves restricting to expressions with booleans, arithmetics, strings, and arrays of these types. Taking a step back, this means that the paper has now posited a way to create an AST representation of a piece of text given its embeddings and a tree representing the previous status of the AST. The final tree is obtained by using a ground truth target tree, which is sparsely connected, and then obtaining representations for all the nodes in the graph using the encoder-decoder architecture. The paper also extends the decoder with an attention mechanism that represents stateful information about each node. Indeed, the paper concludes that a graph encoder architecture performs best on the task posited in the paper, achieving an accuracy of 57% when translating to the paper's posited NAG (although the large room for improvement suggests further work in this area is needed).

All of these papers describe concrete instances of supervised models obtaining strong and state-of-the-art (at the time) performance on the natural language to code task. The majority of these papers utilize core programming language data structures such as ASTs and traditional natural language processing techniques such as slot insertion and semantic parsing. Although the specific linguistic techniques utilized in these papers were not used for this thesis, some of the ideas and treatment of programming language ideas such as syntax evaluation, string insertion, and dataset treatment were directly used for the experiments contained in later sections.

2.2 Unsupervised Learning

While unsupervised learning was the predominant paradigm for the NL2C task until 2017, the emergence of transformers and the transfer learning paradigm (discussed in Section 2) dramatically transformed approaches taken for the NL2C task and its natural language analogue of machine translation (MT). First we discuss the impact on MT, then discuss the papers that have applied similar techniques to NL2C, achieving current state of the art results.

2.2.1 Machine Translation

Next, we review papers that achieve state of the art (SOTA) performance in the unsupervised translation task for natural languages. While the previous three papers used ASTs/grammars in the decoder in order to translate to code, the following approaches do not use any special decoder architecture. These include a paper that allows training on monolingual corpora for machine translation, a system for unsupervised translation known as mBART, and a system that learns translations from code to English-language documentation.

A paper by Lample et al. [9] discusses a technique to train machine translation systems on monolingual corpora. The datasets used to train this model are the WMT 2014 English-French dataset, the WMT 2016 English-German dataset, and the Multi30k dataset, which is a series of 30,000 images with annotations in English, French, and German that are translations of each other. The core technique this paper uses is an algorithm that is broken down into three major steps. The first step is to train individual language models for the source and the target languages. This is then used to create two initial translation models, one from the source to the target language and one in the other direction. The algorithm also uses back-translation [17]. Back-translation is a technique for training source to target language models on monolingual corpora. This involves creating a second "back-translation" model to translate outputs from the model in the target language back to the source language. The back-translated text is then used to improve the back-translated model.

This effectively creates a supervised learning task. The language modeling portion of the task is accomplished by denoising autoencoding, which predicts the next word by training on a corpus with words that are swapped and removed, and then asking the language model to reconstruct the original output. The model was then tested against several baselines such as word-by-word translation, word-reordering using an LSTM-based language model, oracle word reordering (best possible generation), and a model trained with cross-entropy loss using supervision. The model trained in this paper outperforms heuristic baselines on almost all tasks. Additionally, compared to the supervised approach, the researchers found that the unsupervised approach obtains the same performance as the supervised approach given three iterations of the aforementioned algorithm.

Two papers by Liu et al. [11] and Song et al. [18] introduce new systems called mBART and MASS, which apply the techniques of multilingual denoising autoencoding and masking to pretraining monolingual corpora for unsupervised learning to result in a seq to seq pre-trained model. MASS trains both the encoder and the decoder jointly. Given an unpaired source sentence and a modified version where a certain substring is masked and replaced with other symbols, MASS predicts the masked fragment by taking the remainder of the masked sequence as input to its encoder, and then taking the unmasked substring with the other tokens of the sentence masked as an input to its decoder. MASS uses 6-layer transformers in its experiments and thusly pretrains a model that can then be fine-tuned for a given task. mBART is a similar system that uses a 12-layer transformer model with denoising autoencoding after corrupting the original input and trains the encoder and decoder jointly. However, it differs from MASS in that it does not mask the input at all for decoding. In contrast to MASS, mBART also uses masking as discussed above but permutes the order of the sentences within each instance as its noise function. mBART trained on 25 languages shows comparable performance to MASS across the board in the machine translation task, while both models significantly outperform previous SOTA benchmarks like Lample et al. [9].

2.2.2 NL2C

Finally, we discuss the very recent advances made by presenting the problem of natural language to code as a text-generation task trained on very large datasets of code. These represent the current state of the art results in the field of natural language to code generation.

One of the most influential paradigm shifts in the field of NL2C is that of harnessing the power of language models, and the foremost advance in this area is the creation of the "GPT-3" model, detailed in the corresponding paper by Brown et al. [20]. One trend identified by researchers at OpenAI is that the improvement of transformer-based models on downstream benchmarks, given an increase in the training parameters and the data, tends to follow a smooth trend of improvement with scale. Thus, they test the hypothesis that in-context learning may also improve with scale by training a then-record 175 billion parameter autoregressive language model. This model architecture is generally similar to the original transformer paper by Vaswani et al. [22], but crucially is a decoder-only model instead of a seq2seq one. The largest, and best-performing, model utilized 175 billion parameters, 96 layers that were decoder-only with masked self-attention heads, 12288 units in the bottleneck layer, and a batch size of 3.2 megabytes, with a context window of 2048 tokens. The dataset that was used to train this was called "Common Crawl", an open source dataset consisting of nearly a trillion words. On top of this, the authors filtered a version of Common Crawl that was most similar to a range of high-quality corpora, performed fuzzy deduplication of the documents, and added known high quality corpora such as WebText and English-language Wikipedia. This resulted in a dataset of almost 500 billion tokens, which was tokenized using a byte-level BPE tokenizer. The model was then divided into two tasks. The first was "0-shot", or when the model is fed no task examples before being asked for an answer. The second was "few-shot", when the model is fed a handful of examples of the task before being prompted for an answer. The resulting model was then tested on a variety of downstream tasks from language modeling to question-answering to comprehension. The few-shot model performed on par with or slightly below state-of-the-art in the majority of tested tasks,

indicating that a very large language model tuned on just a handful of examples can yield very good performance.

Another paper by Chen et al. [21] directly tackles the natural language to code space using the GPT model, and achieves state of the art results on a dataset that they propose. The resulting published model is known as "Codex". The authors built on top of existing GPT models up to 12 billion parameters by using finetuning. In order to attain code translations for docstrings in a language modeling context, they sequentially ordered the docstrings and corresponding python code and evaluated the model by feeding in just docstrings to see what the model generated. The training dataset consisted of scraped files from GitHub, which contained 179 GB of Python files under 1 MB each. The authors finetuned with a 175 step linear warm up and cosine learning rate decay, and trained for a total of 100 billion tokens. Finally, they evaluated the result against a novel dataset known as HumanEval (which is also used for evaluation of the models presented in this thesis). This dataset contains a list of 164 hand-documented and coded Python code prompts, as well as accompanying code to evaluate the correctness of each of generated function against pre-written tests. The most relevant metric is known as pass@k - where k stands for the number of samples that a model generates for a prompt, pass@k is a count of the number of prompts where at least one of k samples passes the tests. The best performing model is the 12B parameter GPT-3 model, which passes 28.82% of tests for $k = 1$, 46.81% of tests for $k = 10$, and 72.31% of tests for $k = 100$.

Another approach to the problem of code generation from natural language text was taken by Li et al. [26], resulting in the creation of the "AlphaCode" model. The primary process followed the pretrain-finetuning model followed by previous papers, and also used the encoder-decoder transformer architecture for language modeling. As opposed to pretraining the model on general English-language text, AlphaCode was pretrained on GitHub code, including documentation, using a SentencePiece tokenizer. The pretraining model utilized an asymmetric architecture, with 1536 tokens for the encoder and 768 tokens for the decoder. The base model had 1 billion parameters, with a batch size of 256, while the largest model

had 41 billion parameters. Finetuning was performed on a competitive programming dataset called CodeContests, using GOLD, a specialized importance-weighting algorithm based on demonstrations, as an objective [12]. These samples are then generated from the models for each problem, and example tests given in the problem statement and clustering were used to select the 10 samples to be evaluated on hidden test cases. In the clustering stage, a new model was trained on GitHub data to generate tests given an input, which was then fed back into sample solutions for a given problem to determine how to group sample solutions. Crucially, the resulting model was measured against other human coders entering competitive programming competition through the Codeforces and Codecontests websites. Most notably, in the Codeforces competition, when allowing for 10 submissions per problem, AlphaCode ranked in the top 53.3% of all submissions in the competition in tests passed, and when allowing for an unlimited number of submissions AlphaCode ranked in the top 48.8%. In the Codecontests competition, the model was able to correctly solve up to 29.6% of tests in the test set.

These models and advances represent the most current benchmarks and best-performing models in the natural language to code space. The extent to which these models demonstrate understanding of the code, though, is the matter of extensive debate [4]. The researchers in the Codex paper especially provide a number of caveats undermining the notion of "understanding" the source code, such as how Codex is significantly more inefficient to train than a human developer, and how compared to the amount of code Codex is trained on, "even seasoned developers do not encounter anywhere near this amount of code over their careers" [21]. Another criticism of these models, as with other large language models, is the tendency of evaluate on similar or identical problems as those included in the training dataset, a potential problem cited (and addressed) in the Codex [21] and Alphacode [26] papers.

Nevertheless, these models represent the state of the art in the current NL2C space. As a result, several of the ideas posed throughout this paper, as well as several follow-ups, will directly follow from the techniques used in this section such as tokenization, masking

techniques, and training routines (see Chapter 3). Most notably, mBART will be used as the driving framework for investigations in this paper due to reproducibility and practicality reasons, as small datasets work especially well in this framework compared to those discussed in the papers in the NL2C section.

Chapter 3
ALGORITHMS AND IMPLEMENTATION

This chapter describes the experimental setup, datasets, cleaning methodology, and approach used to run the experiments described in subsequent sections. Overall, this paper approaches the problem of natural language to code by using seq2seq techniques through mBART to translate documentation to code.

3.1 Frameworks and Architecture

As discussed previously, the current paradigm for NL2C is using large language models to perform translation. This paper takes a different tack by approaching NL2C as a machine translation task, in the style of programs like Facebook's TransCoder [15]. As demonstrated in the aforementioned paper, seq2seq approaches achieve powerful results in translating programming languages, and has been scarcely explored in the NL2C task. Furthermore, in contrast to language modeling approaches to NL2C, which consume large amounts of resources and require large training times, seq2seq frameworks achieve excellent results in resource-constrained settings [11]. The most advanced seq2seq framework at the time that these experiments were begun was mBART, and therefore this is the framework that this thesis uses for all experiments.

With a handful of exceptions (see Chapter 4), this paper follows the typical mBART paradigm. This means pretraining for a language modeling objective on the source language and the target language using denoising autoencoding as discussed in Lample et al. [9], and fine tuning to achieve translation from the source language to the target language. mBART for the conditional generation task was the base model used for training. The vocabulary size for the model was 52000, with 12 encoder/decoder layers and 16 encoder/decoder attention

heads, with encoder and decoder dimensions of 4096, and with a batch size of 8. The model generates output token by token.

3.2 Dataset

```
'''
Write a function to find the maximum sum that can be formed which has no three consecutive elements present.
'''

def max_sum_of_three_consecutive(arr, n):
    sum = [0 for k in range(n)]
    if n >= 1:
        sum[0] = arr[0]
    if n >= 2:
        sum[1] = arr[0] + arr[1]
    if n > 2:
        sum[2] = max(sum[1], max(arr[1] + arr[2], arr[0] + arr[2]))
    for i in range(3, n):
        sum[i] = max(max(sum[i-1], sum[i-2] + arr[i]), arr[i] + arr[i-1] + sum[i-3])
    return sum[n-1]
```

Figure 3.1: An example docstring (enclosed in triple quotes at the top of the block), and corresponding Python code from the MBPP dataset.

For the sake of this thesis, translation between English and Python was performed. The primary datasets used were CodeSearchNet [8] and Google's Mostly Basic Python Problems [2]. CodeSearchNet is an extensive multi-programming language dataset published as part of a challenge to evaluate the state of semantic code search, or retrieving relevant code given a natural language query. This dataset was collected from publicly-available non-forked GitHub repositories, sorted by number of forks and stars. Only functions that had documentation were included as part of the final dataset. Documentation longer than one paragraph was shortened to the first paragraph, and functions that had documentation of less than three words were removed from consideration. Functions shorter than three lines were also removed, as well as unit or integration test functions. This resulted in a set of 503000 Python functions with documentation. Mostly Basic Python Problems (MBPP) is a dataset of 974 human-created problems designed to be solved by an entry-level Python programmer. Both datasets were used for pretraining and finetuning in initial experiments.

The two datasets differed significantly in form and function. The CodeSearchNet dataset had a number of class functions that called each other, and functions that served solely as getters and setters in relation to the broader class. As a result, many functions were not syntactically coherent or correct as standalone functions, and would be deemed syntactically invalid during evaluation. On the other hand, the MBPP dataset was comprised entirely of self-contained functions. To mitigate this difference, the CodeSearchNet dataset was primarily used in pretraining, to help train the model on creating a language model for the Python and English languages. The MBPP dataset was used mostly in the finetuning phase, where the one-to-one relationship between documentation and function, and the modular and self-contained nature of each sample, would prove beneficial to creating a mapping between documentation and Python. Additionally, this two-phase process was generally followed to ensure that the MBPP dataset was not "drowned out" by the much larger CodeSearchNet dataset. As expected, this setup generally yielded better results than other experimental setups.

Two repositories, `QUANTAXIS/QUANTAXIS` and `shidenggui/easytrader` were excluded from CSN due to the presence of non-Latin characters in their docstrings. For the purposes of this program, the code portion of both datasets were stripped of comments in the code (single and multi-line), and then passed into the program. Some effort was also made to remove examples and other confounding characters from the input text to make the entry text pure English (see Figure 3.1 for an example); these efforts were stopped due to inconsistency of the input data and inability to consistently obtain plain English text (see Chapter 5, Discussion).

3.3 Training and Experimental Setup

Each training example was trained using the architecture listed above. The dataset was tokenized using a Byte-Level BPE tokenizer trained on Python and English, using the full CodeSearchNet dataset, for 52000 tokens. This tokenizer is known as a sub-word tokenizer, which functions by retaining frequently-used words as single tokens, but splitting rare words

```
Construct a StructType by adding new elements to it to define the schema. The method accepts
either:

    a) A single parameter which is a StructField object.
    b) Between 2 and 4 parameters as (name, data_type, nullable (optional),
        metadata(optional). The data_type parameter may be either a String or a
        DataType object.

>>> struct1 = StructType().add(""f1"", StringType(), True).add(""f2"", StringType(), True, None)
>>> struct2 = StructType([StructField(""f1"", StringType(), True), \\
...     StructField(""f2"", StringType(), True, None)])
>>> struct1 == struct2
True
>>> struct1 = StructType().add(StructField(""f1"", StringType(), True))
>>> struct2 = StructType([StructField(""f1"", StringType(), True)])
>>> struct1 == struct2
True
>>> struct1 = StructType().add(""f1"", ""string"", True)
>>> struct2 = StructType([StructField(""f1"", StringType(), True)])
>>> struct1 == struct2
True

:param field: Either the name of the field or a StructField object
:param data_type: If present, the DataType of the StructField to create
:param nullable: Whether the field to add should be nullable (default True)
:param metadata: Any additional metadata (default None)
:return: a new updated StructType"
```

Figure 3.2: An example docstring from the CodeSearchNet dataset, problematic on multiple levels. This docstring contains a significant number of Python samples, as well as outputs. While docstrings were intended to be treated as "English", examples such as this containing Python code potentially complicated language modeling and translation. Furthermore, this sample is difficult to translate to Python, as the intended implementation is to call and reference objects and functions that are part of a library or class that the model and/or a human translator does not have access to.

into smaller meaningful subwords [1]. This tokenization strategy retains whitespace, and is able to effectively tackle issues like unknown words, and is used commonly in models with as GPT-3 [20] and mBART [11].

When finetuning on the MBPP dataset, a 75-25 train-test split was used. Unlike mBART, which masks tokens for the pretraining objective by shuffling sentences and masking tokens, the experiments carried out mask tokens and experiment with other masking techniques, but do not shuffle sentences. This is because, unlike natural languages, code does not contain a natural analogue for "sentences". As a result, each example contains just a single instance of documentation and code. When mBART-style masking is used, 35% of tokens are replaced

with a mask token with a Poisson distribution. Another masking technique used was the Lample unsupervised language masking technique [15]. This masks 15% of tokens in a given sample, drops - or excises from the input sequence - 10% of tokens, and uniformly shuffles the sequence by randomly moving each token a maximum of 3 tokens from its starting spot.

Just as with mBART, all experiments used cross-entropy loss for training. The metrics computed at evaluation included F1 score, accuracy, precision, and recall. Some experiments were conducted on measuring syntactic correctness, which involved finding the harmonic mean between whether or not a given translation compiled (1 if it did and 0 if it did not) and the F1/accuracy/precision/recall. Early stopping was also utilized, with stopping patience set to 3.

3.4 Evaluation

```
'''
    Input to this function is a string represented multiple groups for nested parentheses separated by spaces.
    For each of the group, output the deepest level of nesting of parentheses.
    E.g. (()()) has maximum two levels of nesting while ((())) has three.

    >>> parse_nested_parens('(()()) ((())) () ((())()())')
    [2, 3, 1, 3]

'''
```

Figure 3.3: An example evaluation docstring from the HumanEval dataset.

Upon creation of the model, the model was evaluated against the human-labeled dataset "HumanEval", released as part of the previously discussed "Codex" paper [21]. Samples were generated by feeding in the expected output function header into the decoder, as well as the English documentation into the encoder. pass@1 was evaluated for all models (number of problems where 1 sample was generated and passed). A high repetition penalty (5.0) and repeat n-gram size limit (8) were also imposed [23].

Chapter 4
EXPERIMENTS AND RESULTS

The primary results are summed up in the results table below. All experiments are described below the table, and the highest-performing experiments are described in detail.

4.1 Results Table

Table 1 on the following page summarizes the key findings, followed by a table summarizing each model's significance.

Experiments were run across dimensions including modality (how the code is utilized and structured, and what kind of task it is used for), balance of pretraining and finetuning data, specific document used, masking during the pretraining objective, including or excluding syntactic evaluation when choosing the best-performing model, and evaluation datasets used. In addition to the evaluation listed in the tables, a visual inspection of the output was conducted to analyze the type of errors made, which is discussed in this chapter. Overall, while the number of tests passed - the correctness metric - was generally very low, there was significant and notable variation in syntactic well-formedness, as well as the syntactic error profile between each of the models.

The best performing overall model is `M1` listed at the top of Table 1, which is a dataset pretrained on CodeSearchNet document 1. This model uses mBART default masking, which uses a Poisson distribution to mask tokens. This dataset is then finetuned on the a combination of CodeSearchNet document 1 and the Mostly Basic Python Problems (MBPP) dataset, with an 75-25 % train-eval dataset split. A similarly performing model is `M5`, which has the same configuration as the previously mentioned model but is pretrained on CodeSearchNet document 2.

Name	Model Descriptions				Evaluation	
	Finetuning Dataset	Pretraining	Evaluation	Mask Method	Compile Statistics	Tests Passed
M1	2-phase, CSN #2 + MBPP	CSN #1	25% of MBPP and CSN test	mBART	56/164	1/164
M2	2-phase, CSN #2 + MBPP with 1x oversample	CSN #1	25% of MBPP and CSN test	mBART	25/164	1/164
M3	1-phase, CSN #2	CSN #1	25% of MBPP and CSN test	mBART	23/164	0/164
M4	1-phase, 75% of MBPP	CSN #1	CSN test	mBART	15/164	0/164
M5	1-phase with CSN #2 and 75% of MBPP	CSN #1	25% of MBPP and CSN test	mBART	56/164	1/164
M6	1-phase with CSN #2 and 75% of MBPP	CSN #1	25% of MBPP and CSN test	mBART	54/164	0/164
M7	2-phase - CSN #2, then 75% of MBPP	CSN #1	25% of MBPP and CSN test, syntax eval	mBART	69/164	0/164
M8	2-phase - CSN #2, then 75% of MBPP	full CSN	25% of MBPP and CSN test	mBART	54/164	0/164
M9	2-phase - full CSN, then 75% of MBPP	full CSN	25% of MBPP and CSN test, syntax eval	mBART	75/164	0/164
M10	2-phase - full CSN, then 75% of MBPP	full CSN	25% of MBPP and CSN test	mBART	49/164	1/164
M11	2-phase - CSN #2, then 75% of MBPP	CSN #1	25% of MBPP and CSN test	lample	54/164	0/164

M12	2-phase - full CSN, then 75% of MBPP	none	25% of MBPP and CSN test	lample	56/164	0/164
M13	2-phase - full CSN, then 75% of MBPP	full CSN	25% of MBPP and CSN test	mBART	32/164	0/164
M14	2-phase - CSN #7-13, 75% of MBPP	CSN #1-6	25% of MBPP and CSN test	mBART	24/164	0/164
M15	2-phase - CSN #1-6, 75% of MBPP	CSN #1-6	25% of MBPP and CSN test	mBART	31/164	0/164
M16	2-phase - full CSN, then 75% of MBPP	CSN #1-6	25% of MBPP and CSN test	mBART	20/164	0/164
M17	none	full CSN	CSN test	mBART	0/164	0/164
M18	1-phase with 75% of MBPP	full CSN	25% of MBPP	mBART	0/164	0/164

Table 4.1: Primary Results Table. MBPP is the Mostly Basic Python Problems dataset, while CSN is the CodeSearchNet dataset. This dataset is divided further into files 1-13, and some experiments only contain some files of CodeSearchNet, in which case the specific file used is noted. Note that "syntax eval" refers to the practice of frequent checkpointing, then including syntax checking as a metric (in addition to F1) for evaluating which checkpointed model is to be chosen to represent the final model (see Section 4.5). "1x oversample" as referred to for model M2 refers to oversampling the MBPP dataset by duplicating it in the finetuning process (See Section 4.6). 2-phase finetuning refers to finetuning on two separate datasets in two separate steps, while 1-phase finetuning refers to finetuning on the union of the two datasets in one step.

4.2 Pretraining versus Finetuning

One primary mode of investigation was looking at the balance between pretraining and finetuning in terms of data size and effect. An experiment was run on pretraining on the full dataset without finetuning, `M17`, which resulted in 0 tests passing and 0 results compiling. Another experiment was run on training the model with just the finetuning objective, `M13`,

which resulted in 32/164 tests compiling and 0/164 tests passing.

Experiments were also carried out on whether or not finetuning and pretraining sets benefit from being non-overlapping - `M15` was pretrained on CSN documents 1-6 and finetuned in CSN documents 1-6 and resulted in 31/164 tests compiling and 0/164 passing. On the other hand, `M14` was pretrained on CSN 1-6 but finetuned in documents 7-13, and resulted in 24/164 tests compiling and 0/164 passing. This seems to indicate that pretraining and finetuning on the same dataset for this objective results in better performance.

4.3 Filtering Effects

When pretraining on smaller datasets, we note that these subset datasets tend to have better performance than the full dataset, which seems to be counterintuitive and contradictory to prior findings that indicate better comprehension and modeling of Python and English compared when using larger datasets instead of smaller datasets. An example of this phenomenon at work can be seen by comparing `M1` to `M10`, where the former is pretrained on CodeSearchNet document 1 and the latter is pretrained on the full CodeSearchNet dataset al.though both show comparable results in the tests passed, the partially pretrained model performs better than the fully pretrained model in terms of the number of test results that compile.

This discrepancy can be explained by looking at the generated results. For samples that pretrained on the full dataset, a majority that fail to compile print out part of the documentation prompt fed into the decoder, while samples that pretrain on a partial dataset generally do not do this. Indeed, pretraining on increasingly larger datasets seems to generally have the effect of increasing the number of plain English documentation text that is printed in the output, which is observed in the models pretrained on CodeSearchNet documents 1-6 (seen in the results table as models `M15, M16` also see Appendix A for error analysis results table).

```
def is_palindrome(string: str) -> bool:
    - make_palindrome(string) -> str:
    result = \"\"
    for i in range(len(string))[::-1]:
        if (string[i] == string[i + 1]) and
        (string[i + 2] == string[i + 3]):
            result += string[i]
    return result

```

```
def is_palindrome(string: str) -> bool:

    return True;
```

Figure 4.1: An example of two syntactically incorrect output samples between the non-syntax checking model (left. model M5) and model that included syntax checks in the evaluation stage (right, model M7). This demonstrates how the syntax checking models optimized for syntactic correctness at the cost of code that attempts to output a correct result.

4.4 Masking Experimentation

The root cause of the issue discussed in the previous section was hypothesized to be the model overfitting on initial pretraining examples, and not differentiating between samples prefixed with a python tag (`<py>`) and samples prefixed with an English tag (`<en>`). To test this hypothesis, the noising function from Lample et al. [15] was used for the pretraining objective, seen in the results table as models labeled `M11` and `M12`. This noising objective masks tokens with a 15% probability, drops tokens with a 10% probability, and shuffles the list by moving tokens a maximum of 3 tokens away from their original position. Doing so resulted in an improvement in the documentation prompt printing issue, but still had more documentation copying than models pretrained on the partial CSN dataset. The number of compiled samples was roughly the same. Additionally, using the `lample` masking method came at the cost of fewer correct generated samples. Despite this regression, the overall error profile of the `lample` samples, `M11` and `M12` (see Appendix A), also seemed to indicate a very slightly better understanding of the language than either of its analogues without using Lample masking, `M1` and `M10` (see Section 5.1.2).

4.5 Syntax Checking

One tool experimented with was that of syntax checking. More specifically, every time the model was checkpointed and saved, evaluation metrics were supplemented with a syntax checker for each generated result that would parse the generated results and determine how many samples compiled. This augmented evaluation metrics was only used while choosing the superior-performing model between two checkpointed models, and hence deciding which model was used as the final model, and was not used to update parameters of the model during training.

Two methods of doing syntax checking were experimented with, one that took the harmonic mean of the syntax compilation result (1 if it compiled, 0 if it didn't) with other metrics such as F1 score and accuracy, and one that returned 0 as the evaluation score if the sample code didn't compile, and returned the F1 score and accuracy if it did. Examples of the former technique are visible in the results table in `M7` and `M9`. While these dramatically increased the number of generated samples that compiled, the samples themselves were frequently off-base or irrelevant to the given prompt. Further examination revealed that the models that were being saved were naive ones generated in the very early stages of training. Syntax checking did not result in a significant improvement in the quality of generated samples, and was omitted from the majority of experiments conducted.

4.6 Oversampling

Since it was observed that a higher weighting of the MBPP dataset had a direct impact on increasing performance, oversampling was also included in experimentation. Specifically, this entailed oversampling the MBPP dataset by doubling its size (`M2`). By contrasting with the non-oversampled analogue (`M1`), we see noticeably reduced performance, almost entirely due to syntax errors. This trend continued for continued oversampling by larger quantities, indicating overfitting problems. The overall results are very similar to `M3`, which only includes CodeSearchNet data in training and excludes MBPP, with slightly higher

syntactic correctness and lower number of tests that passed.

4.7 Early Stopping

In order to observe underfitting and overfitting in this context, experiments were performed in order to determine the effects of running model training to completion as opposed to early stopping. These experiments found very little difference in continuing training to completion as opposed to stopping. Additionally, it was observed that more frequent checkpointing for models was directly correlated with increased compilation and correctness.

4.8 Dataset Type/Code Style

One of the most important factors that showed up repeatedly in these results is that of dataset size and dataset code style, or what task the code is designed to perform. Results consistently showed that models that more heavily weighted the MBPP dataset were strong performers. Examples of this include the difference in performance between models. Indeed, part of the regression from some models pretrained on the partial CSN dataset to models trained on the full CSN dataset may be attributable to the significantly smaller influence of the MBPP dataset, as the MBPP dataset is significantly smaller than the full CSN dataset. However, finetuning models pretrained on the full CSN dataset and finetuned on a partial CSN dataset gave very poor results, as did finetuning models pretrained on the full CSN dataset on the MBPP dataset alone (`M18`). As a result, almost all experiments used the MBPP dataset in a finetuning stage. Taken together the data seems to indicate that more CodeSearchNet data improved syntactic correctness in generated results, while more relative weighting for unique MBPP samples improved test performance.

Chapter 5

DISCUSSION

Overall, the experiments provide a window into strategies that offer promise for seq2seq techniques for the natural language to code task. This chapter will discuss the overall analysis of the experiments, as well as the effects of early stopping on the experiment results before going over the considerable effects of code style. Finally, this chapter will discuss future directions for the NL2C task, and will discuss datasets that are worth looking into for forthcoming investigations.

5.1 Training Routine Analysis

5.1.1 Dataset Cleaning and Syntax

The primary utility of the CSN dataset was in "learning" the proper syntax of each language. This succeeded in some parts, as many of the outputted code samples were syntactically correct or "almost" syntactically correct (meaning only featuring one or two syntax issues) regardless of correctness. The best performing model was able to generate syntactically correct examples in 56 out of 164 test cases when $k = 1$. However, the overall syntax error profile (see Appendix A) indicates that there is some room for improvement in "learning" the language, especially with regards to proper indentation and parenthesis matching. Using syntax correctness as part of the evaluation criteria for choosing which benchmark was the best-performing model did not increase correctness in any meaningful fashion, and instead degraded the overall quality of the code samples outputted. This experiment indicated that the best way to enhance syntax correctness would be to pretrain on more data rather than attempting to evaluate on a syntax checker in addition to correctness.

```
def is_palindrome(string: str) -> bool:

    if(string[0] == string[-1]):
        return \"\"\"!\"\"\" + str(string) +
        \"\"\"')
    l = 0
    for i in range(len(string)):
        if(string[i] != string[i-1]):
            l += 1
    if(l < len(string)):
        return False
    else:
        return True

```

```
def is_palindrome(string: str) -> bool:
    - make_palindrome(string) -> str:
    result = \"\"
    for i in range(len(string))[::-1]:
        if (string[i] == string[i + 1]) and
        (string[i + 2] == string[i + 3]):
            result += string[i]
    return result

```

Figure 5.1: An example of two syntactically incorrect output samples between the lample (left) and non-lample (right) training methods

5.1.2 Lample Masking

The lample masking technique demonstrates promise in its corruption scheme. The overall error profile of the `lample` samples, `M11` and `M12`, seems to indicate a generally better modeling ability of Python than either of its analogues without using lample masking, `M1` and `M10` (see Appendix A for more details). This is indicated by an increase in general syntax errors, and a decrease in indentation and parenthesis matching errors, for the former two models compared to the latter two models, as well as a slightly higher syntactic correctness rate. It is also worth noting that there is an increase in samples that compiled but failed tests from the latter two models to the former two models.

An example of this difference in syntax error profile can be seen in Figure 5.1. While both examples fail to compile, the left example (from `M12`) only has one notable syntax error with a malformed string, while the right sample (from `M10`) has indentation errors, misplaced characters, and string issues. The left sample also attempts to return the correct type in some cases, while the right does not, indicating potentially better translation.

While the lample model offers more promising modeling abilities upon a deeper look, it shows no overall improvement in compilation or overall correctness compared with the default mBART masking technique. The lample masking scheme could possibly benefit

from an increased amount of pretraining data to boost correctness. Similar to previous findings using masking techniques in ablation studies, these experiments suggest that a more sophisticated corruption scheme requires a larger dataset to properly model Python code.

5.1.3 The Objective Function

One important consideration for the general task of NL2C is the objective function, which is primarily focused on word (or segment) similarity. However, no major metric for evaluating the correctness of programs evaluates the similarity of a given output against a "gold standard" program as is the case with most machine translation applications. Instead, evaluation focuses on being well-formed (compiling), and passing tests designed to measure correctness. As was experimentally confirmed (see section 4.6), more frequent checkpointing directly helps with obtaining better results in the natural language to code task. This is likely attributable to the gap between the evaluation criteria of selecting the best checkpoint, which is based on the number of tests passed, and the objective function. This finding has been confirmed by other papers in this space such as AlphaCode [26], whose authors similarly concluded that "loss is a poor proxy for solve rate". On the other hand, including syntax checking in the evaluation function is not helpful for increasing overall performance of the model (see section 4.4). This is because the model ultimately optimizes for smaller, simpler functions without the essential logic for correctness. Thus, it is beneficial for future experiments to use frequent checkpointing in order to bridge the gap between a correctness-based evaluation function and a word matching-base objective function.

5.2 Data Analysis

5.2.1 Data Cleansing

Another confounding factor in modeling English and Python code comes in the form of proper docstring and Python code cleansing. Although some attempts were made to standardize the input English data from the CSN dataset by removing examples, newlines, breaks, and

other non-English characters, these attempts were mostly aborted due to lack of consistency of comments and docstrings across the repositories compiled in the CSN dataset. This may have resulted in poor language modeling, which directly affects translation quality. Future attempts to delve into seq2seq translation with small datasets would likely see better results with a more rigorous data cleansing routine.

5.2.2 Dataset Size and Balance

Another hint to the importance of dataset size and balance can be seen through the extreme performance similarity between the datasets trained with a fraction of the CSN dataset, as opposed to the full CSN dataset. Indeed, with `M10` and `M1`, although they have equivalent correctness results, the former compiles on slightly fewer samples. Mitigating any potential overfitting issues was the subject of experimentation with lample masking, as well as partial dataset training. Intermediary results training on the partial CSN dataset (such as `M16`) indicate significantly inferior results to either. This hints to there being a benefit to training on roughly equivalent number of samples in the pretraining and the finetuning stage, and an increasing benefit in superior language modeling from pretraining on an increasingly larger number of samples.

5.2.3 Code Style

Unsurprisingly, the most important factor in the quality of the translation appears to be the treatment, sequencing, and balance of the data. The inclusion of the MBPP dataset was crucial in increasing performance (see Section 4.6). One possible reason for this is the similarity in input and output style between the MBPP dataset and the CodeEval evaluation dataset. Unlike the CSN dataset, both the MBPP dataset and the CodeEval dataset were modular functions based on mathematical or logical treatment of basic data structures, such as fibonacci generation or list element deduplication. The descriptions of these tasks were a few sentences in length, and offered standardized mathematical and logical descriptions of the expected output. In contrast, the CSN dataset included large projects on GitHub such

as pandas and scikit. These libraries frequently performed class and API operations such as returning class elements and getting elements from another available interface. Additionally, the docstrings varied in quality and length as discussed earlier. This likely further stymied the utility of the CSN dataset.

It's also worth discussing the utility and purpose of such a NL2C model. While many existing models in this area are aimed to be code-completion software in the style of a language model, or as an assistant while coding in an integrated development environment, this particular NL2C use case revolves around translating Plain English into code on a function-by-function basis. This decreases the utility of libraries with many context-sensitive functions like CSN, and increases the importance of datasets like MBPP. Further attempts to locate datasets with a similar code style to the single-purpose Python functions in the MBPP dataset failed.

5.3 Future Directions and More Datasets

The contents of the experiments conducted in this paper contain several insights for future investigations. For small-scale seq2seq models, a pretraining-finetuning paradigm is optimal, which offers the best balance of unsupervised and supervised training. The lample masking technique is the most promising technique going forward, as long as large datasets are being used. On the other hand, default mBART masking is best for smaller and more focused datasets. Very frequent checkpointing (ideally 500 steps or less) is crucial to reconcile the gap between evaluation function and objective function, and further investigations into the best objective function for this task may be warranted to better optimize model training. Utilizing another training algorithm, such as GOLD [12] which was used on the AlphaCode model [26] may also offer a promising area of inquiry.

On the dataset side, perhaps the most important considerations for future investigations is the compilation of a large dataset. From the results, the finetuned/solely supervised model (`M13`) performs significantly better than the pretrained/solely unsupervised model (`M17`) in terms of compilation. Thus, finetuning seems to have a larger impact than pretraining.

Datasets for finetuning should be more carefully selected, with the code style specifically chosen for the NL2C task. This includes non-mathematical text as well as code that is not part of an internal API or calling other external class methods. This dataset, when used for fine tuning, will allow for the training of a higher quality model that better reflects the demands of the NL2C task. More data for the pretraining phase is also crucial for better language modeling, and which helps increase translation quality. Since this is specifically for the language modeling text, a broader swath of Python and documentation-specific code can be used.

More broadly speaking, much work remains to be done as to utility and overall "intelligence" of large neural networks such as the ones trained in this paper. For the NL2C task, questions such as whether or not such a model is able to capture internal structure would be a fascinating point of discussion with extensive implications. Additionally, a more systematic analysis of code generated by large NL2C models compared to human-written code could capture error types and buckets, and draw some insights into treatment of such dimensions as code structure and variables.

Chapter 6

CONCLUSION

This thesis approached the problem of natural language to code (NL2C) from a seq2seq perspective. The two questions this thesis attemped to answer were firstly to analyze the difference between unsupervised learning and supervised learning and dataset size, and secondly to understand the difference between some crucial parameters of training. This used two public datasets, the CodeSearchNet task for English documentation to Python code, and the mostly basic Python problems dataset, using the mBART seq2seq framework for running experiments. Several avenues for future inquiry were discovered and solidifed, including masking, creation of more datasets fitting the NL2C paradigm, more frequent checkpointing and analysis of the objective function. Future inquiries for this task may look more specifically into the creation of a new dataset with an appropriate code style for finetuning, and feeding more unsupervised training data into the task. Additionally, with regards to training techniques lample masking offers a promising way forward to training seq2seq models for NL2C, while inquiries into checkpointing and balancing between the objective and evaluation function are warranted.

BIBLIOGRAPHY

[1] Byte-pair encoding (bpe) tokenization - hugging face course. URL: https://huggingface.co/course/chapter6/5?fw=pt.

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[3] Marc Brockschmidt, Miltiadis Allamanis, Alexander L Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. In *International Conference on Learning Representations*, 2018.

[4] Ben Dickson. Openai codex shows the limits of large language models, Jul 2021. URL: https://venturebeat.com/business/openai-codex-shows-the-limits-of-large-language-models/.

[5] Edsger W. Dijkstra. On the foolishness of "natural language programming". In *Program Construction, International Summer Schoo*, page 51–53, Berlin, Heidelberg, 1978. Springer-Verlag.

[6] Norbert E Fuchs, Kaarel Kaljurand, and Tobias Kuhn. Attempto controlled english for knowledge representation. In *Reasoning web*, pages 104–124. Springer, 2008.

[7] Karl Hasselström and Jon Åslund. The shakespeare programming language. *SourceForge, August*, 21, 2001.

[8] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[9] Guillaume Lample, Myle Ott, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. Phrase-based & nediural unsupervised machine translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 5039–5049, 2018.

[10] Triet HM Le, Hao Chen, and Muhammad Ali Babar. Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Computing Surveys (CSUR)*, 53(3):1–38, 2020.

[11] Yinhan Liu, Jiatao Gu, Naman Goyal, Xian Li, Sergey Edunov, Marjan Ghazvininejad, Mike Lewis, and Luke Zettlemoyer. Multilingual denoising pre-training for neural machine translation. *Transactions of the Association for Computational Linguistics*, 8:726–742, 2020.

[12] Richard Yuanzhe Pang and He He. Text generation by learning from demonstrations. *arXiv preprint arXiv:2009.07839*, 2020.

[13] Chris Quirk, Raymond Mooney, and Michel Galley. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 878–888, 2015.

[14] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139–1149, 2017.

[15] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems*, 33:20601–20611, 2020.

[16] Stuart Jonathan Russell, Peter Norvig, and Ming-Wei Chang. *Section I.1, Introduction.* Pearson, 2022.

[17] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics. URL: https://aclanthology.org/P16-1162, https://doi.org/10.18653/v1/P16-1162 doi:10.18653/v1/P16-1162.

[18] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mass: Masked sequence to sequence pre-training for language generation. In *International Conference on Machine Learning*, pages 5926–5936. PMLR, 2019.

[19] Ilya Sutskever, Oriol Vinyals, Quoc V Le, Nal Kalchbrenner, Phil Blunsom, Benjamin Marie, Atsushi Fujita, Yuchen Liu, Long Zhou, Yining Wang, et al. Sequence to sequence learning with neural networks. In *NIPS*, volume 195, pages 496–527. European Language Resources Association (ELRA).

[20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, Dario Amodei. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[21] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, Dario Amodei. Evaluating large language models trained on code. *arXiv preprint*, arXiv:2107.03374, 2021.

[22] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[23] Patrick von Platen. How to generate text: Using different decoding methods for language generation with transformers, Mar 2020. URL: https://huggingface.co/blog/how-to-generate.

[24] Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, and Michael D Ernst. Program synthesis from natural language using recurrent neural networks. *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA*, Technical Report:UW–CSE–17–03–01, 2017.

[25] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, 2017.

[26] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, Oriol Vinyals. Competition-level code generation with alphacode. *arXiv preprint*, arXiv:2203.07814, 2022.

Appendix A

ADDITIONAL TABLES AND RESULTS

Model	Error Type	Count
M1	failed: invalid syntax	42
	failed: unindent does not match any outer indentation level	22
	compiled, test failed	18
	failed: expected an indented block	17
	failed: closing parenthesis ']' does not match opening parenthesis '	8
M2	failed: invalid syntax	68
	failed: expected an indented block	27
	failed: unindent does not match any outer indentation level	17
	failed: inconsistent use of tabs and spaces in indentation	9
	failed: closing parenthesis ']' does not match opening parenthesis '	8
M3	compiled, test failed	43
	failed: invalid syntax	32
	failed: inconsistent use of tabs and spaces in indentation	16
	failed: unmatched ')'	13
	failed: expected an indented block	13
M4	failed: unindent does not match any outer indentation level	84
	failed: invalid syntax	41
	failed: unexpected indent	5
	compiled, test failed	5
	failed: unexpected character after line continuation character	4

M5	failed: invalid syntax	42
	failed: unindent does not match any outer indentation level	22
	compiled, test failed	18
	failed: expected an indented block	17
	failed: closing parenthesis ']' does not match opening parenthesis '	8
M6	failed: invalid syntax	53
	failed: unindent does not match any outer indentation level	21
	compiled, test failed	15
	failed: closing parenthesis ']' does not match opening parenthesis '	8
	failed: unmatched ')'	6
M7	compiled, test failed	43
	failed: invalid syntax	32
	failed: inconsistent use of tabs and spaces in indentation	16
	failed: unmatched ')'	13
	failed: expected an indented block	13
M8	failed: invalid syntax	78
	failed: unindent does not match any outer indentation level	17
	compiled, test failed	17
	failed: unmatched ')'	5
	failed: cannot assign to operator	5
M9	failed: invalid syntax	60
	compiled, test failed	38
	failed: expected an indented block	10
	failed: unindent does not match any outer indentation level	5
	failed: unexpected character after line continuation character	5
M10	failed: invalid syntax	74
	compiled, test failed	21

	failed: unindent does not match any outer indentation level	8
	failed: expected an indented block	8
	failed: unmatched ')'	6
M11	failed: invalid syntax	55
	failed: unmatched ')'	20
	compiled, test failed	16
	failed: unindent does not match any outer indentation level	13
	failed: unmatched ']'	5
M12	failed: invalid syntax	65
	compiled, test failed	23
	failed: unindent does not match any outer indentation level	12
	failed: unmatched ')'	8
	failed: unexpected indent	5
M13	failed: invalid syntax	97
	compiled, test failed	12
	failed: unexpected character after line continuation character	11
	failed: unindent does not match any outer indentation level	8
	failed: unexpected indent	5
M15	failed: invalid syntax	97
	compiled, test failed	15
	failed: unexpected character after line continuation character	12
	failed: unmatched ')'	6
	failed: expected an indented block	5
M16	failed: invalid syntax	109
	compiled, test failed	10
	failed: unexpected character after line continuation character	9
	failed: unindent does not match any outer indentation level	7

	failed: unmatched ')'	4
M17	failed: invalid syntax	121
	failed: unindent does not match any outer indentation level	21
	failed: unmatched ')'	2
	failed: unexpected indent	2
	failed: closing parenthesis ']' does not match opening parenthesis '	2
M18	failed: invalid syntax	150
	failed: unindent does not match any outer indentation level	3
	failed: illegal target for annotation	3
	failed: unmatched ')'	2
	failed: unexpected indent	2

Table A.1: Result Error Table. Describes the most common error for each model, along with the count of each error.