A Transformer-based Approach for Translating Natural Language to Bash Commands

Abstract—Deep learning models have been used to provide more accurate translations between different natural languages, such as English and German. The success of these efforts has generated interest in applying deep learning to translate natural language into programming language instructions. This paper explores the translation of natural language into Bash Commands, which developers commonly use to accomplish command-line tasks in a terminal. Using this approach, a terminal takes a command as a sentence in plain English and translates it into the corresponding string of Bash Commands. The automation of generating Bash Commands within terminals can provide novice programmers with a more friendly environment and may save the time they look for examples online. The paper analyzes the performance of several architectures on this translation problem using the data from the NL2CMD competition at the NeurIPS 2020 conference. The architectural insights in this paper improved the current state-of-the-art accuracy on this translation task from 13.8% to 53.2%.

Index Terms—Semantic parsing, Bash Commands Generation, Transformer, Natural Language Processing, Human-Computer Interaction

I. INTRODUCTION

Translating natural language into source code for software or scripts can help developers find ways of accomplishing tasks in languages they are not familiar with, similar to how help forums like Stack Overflow are used today. As early as 1966, Sammet [1] envisioned a future of automated code generation where people program in their native language. While generating software templates from configuration files are common practice today, the research in translating natural language into code are still in relatively early stage.

Past research mainly focused on scripting languages or small code snippets. Various datasets have been created to aid research on generating code from natural languages. Examples of such datasets include WikiSQL for SQL [2], CoNaLa for Python [3], and NL2Bash for Bash [4].

This paper focuses on the task of translating natural language into Commands in the Bash scripting language. Translating natural language into Bash Commands is an example of semantic parsing, which means natural language is translated into logical forms that can be executed [5]. For example, the phrase “how do I compress a directory into a bz2 file” can be translated to the Bash command: `tar -cjf FILE_NAME`.

Translating natural language to Bash is part of the development of Command-Line Artificial Intelligence (CLAI) tools, such as IBM’s CLAI tooling [6]. IBM’s approach integrates AI models into terminals as one of the tools, which allows programmers to interact with terminals more intuitively. For example, programmers can type natural language directly into terminals to get suggestions of Bash Commands that could accomplish the task, thereby reducing the amount of time they spend looking for examples and documentation online, as well as minimizing the time they spend switching between different interfaces (e.g., terminal vs. web search bar).

To increase programmer productivity, the Bash Commands suggested by a tool should be both syntactically and semantically correct. If the suggestions are not syntactically correct and cannot execute, programmers may simply ignore them since they distract from the task at hand. Moreover, if translations are not semantically correct, programmers may execute Bash Commands that do not accomplish the goal that they want to achieve, or worse, have negative impacts on the system (such as deleting important files or directories).

In the near term, natural language to Bash Commands translation is unlikely to replace discussion groups or help forums. They can, however, provide a quick reference mechanism that may improve on-demand code suggestions and popups generated by integrated development environments (IDEs). This type of AI-based approach complements other prior work, such as SOFix [7], which can fix bugs in code by mining postings in Stack Overflow.

This paper provides the following contributions to the study of translating natural language into Bash Commands:

1) It describes an architecture that improves the state-of-the-art performance on translating national language to Bash Commands from 13.8% to 53.2%.
2) It presents results demonstrating that the Transformer model [8] is the current best-performing architecture on this problem,
3) It shows how parameter values can be modified to shrink vocabulary size by 90% and improve accuracy, and
4) It explores the use of Beam Search to provide multiple potential translations and adapt the beam score to create heuristic weights that improved the accuracy by 1.2%.

The remainder of this paper is organized as follows: Section II summarizes recent development in semantic parsing and explains how competitions have been an effective way to encourage advances in deep learning models; Section III summarizes the challenges for translating natural language to Bash Commands; Section IV analyzes the performance of different model structures and training techniques; Section V discusses different metrics and error analysis for the state-of-the-art (SOTA) model; and Section VI presents concluding remarks and outlines our future work.
II. BACKGROUND AND RELATED WORK ON TRANSLATION MODELS

The Natural Language to Command (NL2CMD) competition [9] was launched at the NeurIPS 2020 conference to foster improvements in natural language to Bash Commands translation. The competition challenged teams to build models that could transform descriptions of command-line tasks in English to their Bash syntax. NL2CMD is an updated challenge based on the NL2Bash [4] dataset, in which the NL2Bash is used as the public training dataset and hidden validation/test data are provided by IBM [9].

Various architectures have been explored on different tasks of program synthesis from natural language. For example, Lin et al. [10] achieved state-of-the-art generation of shell scripts using Recurrent Neural Networks (RNNs) [11]. Likewise, Zeng et al. [12] utilized the Bert [13]-based encoder and a pointer-generator [14] decoder to generate SQL code from text. Moreover, ValueNet [15] (Transformer encoder + LSTM decoder with pointer networks [16]) was the first Text-to-SQL system incorporating values. In addition, Xu et al. [17] improved upon the TranX [18] transition-based neural semantic parser to translate natural language into general programming languages, such as Python.

The best results in prior work on the problem of translating natural language to Bash Commands were produced by Telltina [10]. Telltina used the Gated Recurrent Unit Network (GRU) [19], which is an RNN that achieved 13.8% accuracy on the NL2CMD metrics proposed by IBM [9]. The Telltina [10] paper produced the NL2Bash [4] dataset and new semantic parsing methods that set the baseline for mapping English sentences to Bash Commands.

Machine translation has seen great progress with the introduction of attention mechanisms [20], which improve translation quality and reduce training time via better support for parallelism. For example, Google Translate has replaced the original Google Neural Machine Translation System (GNMT) [21] with a hybrid model whose encoder uses the Transformer model [20] based solely on attention mechanisms. This model has higher translation quality, is more stable in training, and exhibits lower latency [8].

Transformer models generally have better accuracy and faster training times [8] than RNNs [11] on machine translation tasks. Prior research on machine translation has primarily investigated a single architecture, Gated Recurrent Unit Network (GRU), for translating natural language to Bash Commands. This paper enhances prior research by exploring the performance of several architectures on the NLC2CMD dataset.

Our experiments with applying Transformer models to the natural language to Bash task show that they outperform other approaches, such as RNNs (18.4% improvement) and Bidirectional RNN (BRNN, up to 4.4% improvement) [22]. Analyzing how model structural choices and prediction strategies affect model performance in natural language to command translation task [9] is therefore a key contribution of this paper.

Since the energy and accuracy metrics for model evaluation were specifically designed for the NL2CMD competition, potential improvements for the metrics are also discussed.

III. KEY RESEARCH CHALLENGES

This section summarizes key challenges that we encountered when translating natural language to Bash Commands and describes general obstacles the machine translation community is currently facing. Translating between two vastly different languages is hard, which is the case when translating natural language to code. Natural language is ambiguous and versatile, whereas code aims for explicit and deterministic expressions.

A. Challenge 1: Translating from an ambiguous language to precise Bash Commands is hard

Translating human language into code is inherently hard. One reason is that human language is ambiguous by nature. As the famous Winograd test [23] puts it, in the sentence “The trophy would not fit in the brown suitcase because it was too big”, it can either mean trophy or suitcase. While a human may be able to decide which one is correct, computers have a harder time since understanding this sentence requires “the use of knowledge and commonsense reasoning” [24].

There are two general types of ambiguities [25]:

- **Genuine ambiguities**, where a sentence really can have two different meanings to an intelligent listener. An example of genuine ambiguity is “merge file A with B in folder C”. This sentence has at least 2 interpretations: “merge file A with B if B is in folder C” or “merge file A with B and put the result in folder C”.
- **Computer ambiguities**, where the meaning is entirely clear to a listener, but a computer detects more than one meaning.” A compute ambiguity can occur when multiple parse trees exist for a natural language sentence, such that when the tree is flattened the order of words for input can be undefined.

Both types of ambiguities can affect the performance of translation from natural language to Bash Commands.

B. Challenge 2: The natural language to Bash translation task is usually a many-to-many mapping

Translation tasks are usually many-to-many mappings, which means there can be multiple correct translations for the same sentence. Moreover, even the sentence itself can have multiple methods of expression. As the size of the dictionary grows, there will be more possible translations for the same input. The process of creating the target sentences requires significant human effort.

Natural language is inherently flexible and Bash Commands can have functional overlap between different utilities. For example, when translating natural language to Bash the phrases find the word "foo" in file "bar" and search in "bar" for "foo" have the same meaning. Similarly, both grep -w foo bar and cat bar | grep -w foo are valid translations.
C. Challenge 3: Developing a single metric for scoring translations is hard

Translation between natural languages can use the bilingual evaluation understudy (BLEU) score as a generally accepted metric [26]. There is no standard way, however, to evaluate generated code quality. Several methods have been proposed to address this problem.

For example, RUBY [27] is used to evaluate SMT-based (Statistical Machine Translation [28]) models of code migration (e.g., translating source code between different programming languages). RUBY has shown higher correlation coefficients than BLEU does with the semantic correctness of generated source code.

In previous work by Lin et al. [4] on translating Natural languages to Bash, BLEU was used as an approximate evaluation for Bash Commands. The author, however, showed that BLEU is not an effective metric to evaluate a formal language for Bash Commands. BLEU was used as an approximate evaluation for Bash Commands. The author, however, showed that BLEU is not an effective metric to evaluate a formal language like Bash since it may not agree with manual evaluation [4]. Instead, Lin et al. [4] proposed a Bash evaluation metric called Token Match (TM) that measures the token overlap between the target Bash Command and the reference. However, experiments have shown that TM is not discriminating enough [4].

Our experience indicates that using BLEU as the evaluation metric for generating code incurs the following drawbacks:

- **BLEU encourages long sentences of translated outputs** since it contains a brevity penalty that punishes output that is shorter than the reference target. This factor was originally designed to encourage long predictions of natural language since shorter sentences have the advantages of fewer n-grams to match. However, brevity in code may actually be beneficial as long as the code is structurally and semantically correct.

- **BLEU can not give different weights to keywords since it takes the average precision of n-gram pairs.** Sentences in natural languages, such as English, can handle missing words better compared to programming languages, due to the inherent redundancy in natural language. BLEU was originally designed for evaluating natural language so it does not score Bash Commands with missing words effectively. Redundancy allows natural language to preserve meaning to a few missing words since each word has a relatively small impact on the sentence. Bash Command utility keywords (e.g., `ls`, `sed`, etc.) have a much higher impact on the sentence semantics than parameters and thus should be given a larger weight.

These fundamental differences in brevity between natural language and code may be caused by their design principles. As M. Boot [29] pointed out, redundancies in natural language are superfluous information in pure linguistics. On the other hand, as C. Hoare [30] pointed out, one key principle for programming language design is “utmost simplicity.”

D. Challenge 4: Paired English and Bash Commands data is not easily accessible

Machine translation models require many training examples. Data are often scarce, however, especially for supervised learning that requires paired data (i.e., data with labels). The Transformer is a supervised learning model for machine translation. Collecting training examples for this model is hard, however, because it requires an understanding of both the source and target languages. Without a large number of training examples, therefore, it may be hard for the model to generalize beyond the small samples in the training set.

Translating natural language to Bash Commands provides a unique challenge in which there are both a large number of English sentences and Bash Commands, but paired data (i.e., English sentences with the corresponding Bash Commands) are not easily accessible. For paired sources, such as coding help forums like Stack Overflow, the question is usually a detailed description of the command and should be summarized succinctly by humans. Creating new data pairs from scratch is another potential solution, but writing Bash Commands requires considerable coding skills and is thus hard to crowd-source.

IV. APPROACHES FOR IMPROVING NATURAL LANGUAGE TO BASH TRANSLATION

This section investigates specific research questions and provides empirically-grounded answers to them that can help address these challenges. Building a natural language to Bash translation model involves three phases: (1) pre-processing of the training data, (2) selecting the best model architecture for the task, and (3) devising an effective approach for determining which of many possible translations should be presented to the user. This section explores key research questions in each of these areas and provides information that should aid other researchers in developing more accurate methods of translating natural language to Bash Commands.

A. Experimentation Approach

For our investigation we utilized the NLC2CMD dataset, which contains 10,347 pairs of English sentences and their corresponding Bash Commands. Of the 10,347 pairs of data, 29 had grammar issues and were therefore excluded. The size of this public dataset was relatively small in the natural language processing research field and the goal for data processing was to create a small word vocabulary and utilize as much data as possible.

We narrowed our investigation of these three steps in the pipeline for translating natural language to Bash Commands, as shown in Figure 1. Research questions related to each step are explored below to assess their effects on accuracy and energy consumption. The key narrowing assumptions we made for each step are listed below.

1) **Data pre-processing** – The data was randomly shuffled and split 80/20 for local training and testing. Pairs of invocations and Bash Commands were tokenized by

\[1\text{It was not feasible to address all the challenges mentioned above, e.g. Challenge III-D requires recruitment of a significant number of programmers.}\]
English and Bash tokenizers respectively, before being fed into multiple Transformer models.

2) **Model architecture selection** – We tested Multiple model structures (RNN, BRNN, Transformer) and the hyperparameters of the best performing model on the local test set were recorded. Models were then trained with these hyperparameters on all local data. Finally, the models with the best performance on the hidden validation dataset were picked for an ensemble, which combines the results of different models and chooses the one with the highest confidence score for each prediction.

3) **Prediction strategy** – As discussed in Challenge 2 in Section III-B multiple translations may be possible. Moreover, as noted in Challenge 1 in Section III-A, ambiguity in the natural language can make precise translation hard. A key question is thus how to determine the final prediction that should be selected and shown to the user. We explored the use of ensemble models for selecting the final translation shown to the user.

### B. Research Questions

Our study explored the research questions described below. **Which deep learning architectures perform best when translating natural language to Bash Commands?**

Since there is relatively little literature published on translating natural language to Bash Commands, an important concern is identifying which architectures published in other domains perform best. In particular, Sequence-to-Sequence [31] models have been studied extensively in the context of translations, so we explored their performance on this particular task. These models consist of two main components: an encoder and a decoder. The encoder turns the inputs into vectors and the decoder reverses the process. We compared different combinations of encoder-decoder layers, including RNN, BRNN, and Transformer, to translate natural language to Bash Commands.

Chen et al. [32] discovered that Transformer quality gains stemmed mostly from the Transformer encoder and that RNN decoders often have faster inference times. We therefore mixed and tested different combinations of encoder and decoder types. Table I summarizes the performance comparison (measured in seconds) between different model structures.

<table>
<thead>
<tr>
<th>Encoder</th>
<th>Decoder</th>
<th>Accuracy</th>
<th>Train</th>
<th>Inference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformer</td>
<td>Transformer</td>
<td>0.522</td>
<td>1625</td>
<td>0.126</td>
</tr>
<tr>
<td>Transformer</td>
<td>RNN</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RNN</td>
<td>Transformer</td>
<td>0.486</td>
<td>1490</td>
<td>0.116</td>
</tr>
<tr>
<td>RNN</td>
<td>RNN</td>
<td>0.336</td>
<td>1151*</td>
<td>0.069</td>
</tr>
<tr>
<td>BRNN</td>
<td>Transformer</td>
<td>0.495</td>
<td>1411</td>
<td>0.120</td>
</tr>
<tr>
<td>BRNN</td>
<td>RNN</td>
<td>0.476</td>
<td>1218</td>
<td>0.065*</td>
</tr>
</tbody>
</table>

The results shown in Table I indicate that in this particular case, using the Transformer as both an encoder and decoder has the best accuracy. Likewise, the model with an RNN as the decoder can reduce inference time by 50%.

The NLC2CMD competition brought together a widely diverse set of initial approaches. The research community has not yet gathered behind a smaller set of architectural approaches for this problem. Although we cannot rigorously compare each architecture to others, analyzing the results from the NLC2CMD competition provides insights into (1) strategies for handling training data, (2) which models tended to perform better, and (3) performance issues of specific models.

To provide a high-level perspective on how model architecture impacts performance, we analyzed the architectures of the top-performing teams in the NLC2CMD competition. Table II shows the Top 4 teams and the baseline model on the NLC2CMD Challenge leaderboard [9]. The Transformer architecture discussed in Section IV-C was produced from an analysis of Team Magnum’s architecture, which won the accuracy competition.

AICore [9] won the energy track by having the least energy consumption with a two-stage prediction design consisting of two 2-layer Transformers. The first model predicted the template and the second model filled in the arguments. We suspect their small energy consumption is due to smaller model size (in contrast, the Magnum teams’ model consisted of 6 layers). However, the gain in less energy consumption also came with a cost of lower accuracy (4.3% decrease).

Team Hubris [9] adopted a fine-tuned ensemble GPT-2 as the language model and achieved second place in accuracy. GPT-2 models are large (usually more than 5 GB) and power-hungry. It is therefore challenging to apply them as a background program running continuously in a terminal to suggest translations of Bash Commands. Another problem with GPT-2 ensembles is their inference time (774M params) was prohibitive for real-world deployment, which requires fast response time and low energy consumption to run continuously in the background. Considerable effort is needed to compress and deploy GPT-2 ensembles to compete with other solutions.

Team Jb [9] augmented the training data using backtranslation [34] and created 78,000 augmented training samples. They...
also used the manual pages of Linux Bash Commands [35] to concatenate utilities with corresponding flags and generated an additional 200,000 new samples. Similar to Team AICore, they also used a two-stage model consisting a classifier for utility prediction and a transformer for command generation. Interestingly, a large number of additional training samples was insufficient to overcome the architectural improvements of other teams.

The results shown in Table II provide several key insights:
- Transformer models were the most popular choice. In this task, two-stage models performed worse than a single-stage-and-larger model.
- GPT-2 approaches achieved near state-of-the-art accuracy, but produced much larger models compared to Transformers and had much longer inference times.
- Data augmentation improved accuracy (Team Jb is 1% more accurate than Team AICore) but had less impact than the model structure in this task (with the caveat that the two teams had similar—but not identical—model).

The experiments in the remainder of the paper utilize Transformer models since they were the best-performing architecture in the NLC2CMD competition.

**How do Bash command parameters affect the performance of natural language to Bash translation?**

As discussed in Section III-D, obtaining training data of paired English and Bash Commands is hard. Without sufficient training data, the model may not be able to learn the entire vocabulary that it must translate to or from. Finding ways of reducing vocabulary size is thus essential to develop more accurate models.

Bash Commands typically consist of three terms: (1) utilities that specify the main goals of the command (e.g., `ls`), (2) flags that provide metadata regarding command execution (e.g., `-verbose`), and (3) parameters that specify directories, strings, or other values that the command should operate on. Each utility has a bounded number of flags that can be passed to it. In contrast, parameters have a much larger range of values. Training examples for translating natural language to Bash Commands provide values for the parameters, which can vary significantly between translated examples of the same command.

We hypothesized that including the actual parameter values (such as `ls /var/www` and `ls /etc`) from the training examples would vastly increase the overall vocabulary size and decrease model accuracy. Our rationale for this hypothesis was that there were few limited paired examples of natural language and Bash Commands and translation models typically perform worse with large vocabulary sizes and limited training data.

To test this hypothesis, we used the English and Bash tokenizers from the Tellina model [10] with our modification. As shown in Figure 2, Bash tokens can be categorized as utilities, flags, and parameters (i.e., arguments, such as a specific path). The English tokenizer decapitalized all the letters and replaced parameters with generic forms. The Bash tokenizer parsed Commands into syntax trees with each element labeled as utility, flag, or parameter.

Our accuracy metric focused mainly on the structure and syntactic correctness of the Bash command. We therefore replaced all the parameters in Bash with their corresponding generic representations. For example, a folder path like `tmp/bin` is replaced with `PATH`. By applying this transformation, the Bash vocabulary size was reduced from 8,184 to 776 tokens, and the accuracy of the Transformer models we tested increased by 1.3%. As shown in Table III, we achieved accuracy and performance increases across all architectures, especially for the ones with less accuracy.

**How accurately do beam scores reflect the quality of Bash translations?**

As discussed in Section III-B, a natural language sentence may have multiple possible Bash meanings that could be viable translations. To better utilize the Bash suggestion ability of the translation model, users were provided with five Bash translations for each English sentence, ranked by confidence from high to low. One way to produce multiple translations is to enable Beam Search [36], which creates a tree structure exploration space and evaluates the probability of words at each step. This scoring mechanism thus allowed for five predictions for each invocation (English sentence) and expected confidence weights at each position.

We explored the impact of Beam Search on translation accuracy. When Beam Search is enabled, the Transformer model produced a negative beam score for each corresponding prediction. We observed a strong correlation between a high score and a correct prediction by mapping the exponential of the beam score to the range 0-1. Figure 3 shows the confidence score distribution contrasted with correct and incorrect

<table>
<thead>
<tr>
<th>Team</th>
<th>Model</th>
<th>Data Augment</th>
<th>Accuracy</th>
<th>Power</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magnum</td>
<td>Transformer</td>
<td>No</td>
<td>0.832*</td>
<td>682.3</td>
<td>0.709</td>
</tr>
<tr>
<td>Hubris</td>
<td>GPT-2</td>
<td>No</td>
<td>0.513</td>
<td>809.6</td>
<td>14.87</td>
</tr>
<tr>
<td>Jb</td>
<td>Classifier+Transformer</td>
<td>Yes</td>
<td>0.499</td>
<td>826.9</td>
<td>3.142</td>
</tr>
<tr>
<td>AICore</td>
<td>Two-stage Transformer</td>
<td>No</td>
<td>0.489</td>
<td>506.9*</td>
<td>0.423</td>
</tr>
<tr>
<td>Tellina [10]</td>
<td>BRNN (GRU)</td>
<td>No</td>
<td>0.138</td>
<td>916.1</td>
<td>3.242</td>
</tr>
</tbody>
</table>
predictions. We also found the translation quality of the same sentence between Beam Searches tended to have a strong correlation, which means Beam Search alone is not a sufficient solution for five Bash translations.

When the first prediction in our tests got a negative score, only 9.2% of the time did the other four predictions get any positive score. Based on our observations, we hypothesized that when the first result from Beam Search led to a wrong prediction, the rest of the predictions were also likely incorrect. A better solution would thus involve ensembling different models to decrease the bias for the five predictions.

**How to balance between exploration gains and minimize punishment in natural language to Bash translation?**

In any deep learning model, the metric used to measure the “accuracy” of predictions is critical. As described in Section III-C, it is hard to capture the accuracy of translating natural language to Bash Commands in a single metric. A key question is therefore how to communicate the expected “quality” or “accuracy” of a translation to a user to ensure they see a wide range of possible translations, but also understand which translations the model is more confident about.

For each English sentence, five translations to Bash Commands are produced to better assist users by providing multiple choices. The metric for evaluating the final (i.e., all five) accuracy can be summarized as the following [9]: If any of the predictions have a positive score, take the max among the five, otherwise take the average:

\[
\text{Score}(A(nlc)) = \begin{cases} 
\max_{p \in A(nlc)} S(p) & \text{if } \exists_{p \in A(nlc)} S(p) > 0, \\
\frac{1}{|A(nlc)|} \sum_{p \in A(nlc)} S(p) & \text{otherwise,}
\end{cases}
\]  

(1)

(2)

Considering the scoring mechanism that every predicted result contributed to model performance evenly, we hypothesized that using 1.0 as the first confidence weight and using Beam Search to produce the other four prediction weights would improve model performance. The evaluation score increased accordingly. In contrast, the improvement vanishes if completely neglecting other predictions when the first prediction is wrong. To achieve the balance between exploration and minimize punishment, we capped our confidence score empirically with the following equation:

\[
\text{Confidence}_i = \left(\frac{\text{BeamScore}_i}{2}\right), 2 \leq i \leq 5
\]  

(3)

**What are ensemble guidelines for improving natural language to Bash translation?**

Ensembling has been widely studied as a method of improving the accuracy of machine learning tasks. There are three types of ensemble: bagging, boosting, and stacking. The advantage of an ensemble approach can be statistical, computational, and representational [37]. We sought to determine how effective ensemble would be in translating natural language to Bash Commands and if any general guidelines could be found based on results from processing the NL2CMD dataset.

The key insight behind ensembling is that multiple models can often produce a better overall prediction. One model may learn better translations of file operations with the `ls` command. Conversely, another model may learn better translations of how to copy files or search for text within a file. The goal is to find a group of models that provide better overall translation of sentences into Bash Commands when they are combined.

Past research [38] has shown that a checkpoint ensemble (CE) is an effective ensemble approach compared to minimum validation (MV) model selection and last k smoothers (LKS), though not as good as random initialization ensembles (RIE). CE is particularly suitable for deep learning models since they have long training times. By evaluating the validation scores, CE combines different checkpoints during a single model’s training process [38].

Our experiments, however, did not detect significant improvement from applying ensembling to translate natural language to Bash Commands. As shown in Table IV, we
ensembled different numbers of checkpoints and found the improvement from ensemble was random and minimal. We also observed the same pattern when ensembling models of different architectures.

<table>
<thead>
<tr>
<th></th>
<th>Top1</th>
<th>Top2</th>
<th>Top3</th>
<th>Top4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>0.529</td>
<td>0.524</td>
<td>0.523</td>
<td>0.522</td>
</tr>
<tr>
<td>Ensemble</td>
<td>0.529</td>
<td>0.528</td>
<td>0.525</td>
<td>0.525</td>
</tr>
</tbody>
</table>

Guarding against corner cases when translating natural language to Bash Commands.

Corner cases are rare—but damaging—inputs to a system that result in unpredictable outputs. They are particularly damaging in a translation system since past research has shown that a Neural Machine Translation (NMT) model may lose user trust due to incorrect translations [39]. Machine translation hallucination is the phenomenon when a model produces strange “translations” for certain inputs [8].

Our study found that the RNN model from past work [4] exhibited hallucinations when it encountered certain corner cases. For example, when given the input of “List all files in the current directory that start with b”, the system produced a reasonable translation `find . -name ‘b’ -prune -or -print`. However, with the input “List all files in the current directory that start with a’, the system produces `find . -name [__SP__UNK] ...` with the word `[__SP__UNK]` repeating over 13 times. We suspect that since “a” is an article in English, the model treats it as an unfinished sentence. Moreover, since the model is recurrent it produces a random number of unknown words `[__SP__UNK]` after the first appearance.

Other studies [39] have shown that the Transformer model hallucinates significantly less than the canonical (Recurrent) model. Since we replaced the parameters with generic representations in the data pre-processing step, the aforementioned cases were the same in our model and did not pose an edge case. Our model may still produce “unk” as an out-of-vocabulary word, however, so we chose to filter out the word “unk” from our model’s output to provide a better user experience and guard against translation hallucination.

C. Summary of the Highest Performing Architecture

We tested several different data processing, architectural, and post-processing strategies, as discussed above. We now describe the best-performing model that we tested on the NLC2CMD competition data. Although this model will likely be improved by subsequent work, it provides a starting point for researchers focusing on natural language to Bash Command translation. In particular, our results show that the Transformer model is a robust foundation for future research in this area.

Our Transformer model pipeline was built from the following six steps shown in Figure 4 and described below:

1) **Parsers and filters** – The paired raw data first go through different parsers that convert English sentences and Bash commands into syntax trees (data that cannot be parsed are removed).

2) **Flatten and pre-process** – The syntax trees are flattened and the parameters are replaced with their generic representations.

3) **Tokenizer** – The flattened sentence pairs are tokenized and dictionaries are created for English sentences and Bash commands.

4) **Transformer models** – Tokenized sentences are fed into Transformer models and Beam Searches are enabled to produce multiple translations.

5) **Ensemble** – The best-performing models on the validation dataset are chosen to create an ensemble.

6) **Post-process** – The translations produced by the ensemble model are post-processed by removing the placeholder word `unk`.

D. Parsing and Tokenization

Bash Commands can be complex and nested, as shown in Figure 5. This structure helps explain why programmers may find it hard to create—or even comprehend—Bash Commands, thereby motivating the need for a customizable parser. We built our parser atop the Tellina [10] parser that was developed based on Bashlex [40] in prior work. This parser can parse a Bash Command into an abstract syntax tree (AST) that consists of utility nodes, each of which may contain multiple corresponding flags and parameters. During the tokenization stage, utilities and flags are kept “as is” and parameters are categorized and replaced with `_NUMBER`, `_PATH`, `_NUMBER`, `_PATH`, `_NUMBER`, `_PATH`,
Natural language sentences are pre-processed by filtering out the stop words. The remaining words are then decapitalized and lemmatized to create a relatively smaller dictionary mapping. For example, the sentence Add '.avi' extension to all files/directories with '.mkv' extension under '/volume1/uploads' directory tree becomes add _FILE extens to all file and directory with _FILE extens under _FILE directory tree.

E. Model Details

The model with the highest accuracy used a Transformer as both the encoder and the decoder, as shown in Figure 6. The encoder and decoder each consisted of six layers. The model was trained for 2,500 steps and used an ensemble of the four top-performing single models.

The first positional weight was set to 1.0 and the rest of the weights were set to the exponential of beam scores capped by 0.5. We focused on training an efficient and robust model that can be deployed easily. The need to modify the network structure was therefore relatively low. Instead of FairSeq [41] (which allows users to modify the low-level network structure), we chose OpenNMT [33], which is an open-source neural sequence learning framework to implement our Transformer model.

We found that the Transformer model is sensitive to learning rate and larger batch sizes will produce better results. The detailed training hyperparameters are available on our GitHub repository (blinded). Likewise, the guiding principle behind our tuning strategy is derived from Popel et al. [42].

Our model achieved 53.2% accuracy on the hidden test dataset for the NLC2CMD competition and had the top performance in both inference time and energy consumption. We addressed challenge III-A by masking out specific parameters. To limit ambiguity the dataset also restricted the natural language description to a single sentence and the Bash command to a single line [4].

The NLC2CMD competition provided us with an accuracy metric and an energy metric (discussed in Section III-C), thereby addressing Challenge III-B. When the dataset was collected, the same Bash command was paired with many English descriptions to increase language diversity [4], thereby addressing Challenge III-D.

We trained our model on 2 Nvidia 2080 Ti Graphic cards with 64GB memory. Figure 7 visualizes the global attention between a sample English sentence and a Bash command [43].

V. METRICS AND ERROR ANALYSIS

This section discusses different metrics and error analysis for the model. Section V-A describes the accuracy metric and proposes an improved energy metric. Section V-B then analyses the distribution of different error types.

A. Metrics

Accuracy. The ideal metric for an evaluation would check if the predicted Bash Command produces the same result as the reference answer. That metric is not practical, however, since establishing a simulated environment for 10K variant situations is beyond the scope of this paper. Instead, our scoring mechanism specifically checks for structural and syntactic correctness that “incentivizes precision and recall of the correct utility and its flags, weighted by the reported confidence” [9].
The metric first defines two terms: Flag score $S_{F}^{p}$ and Utility score $S_{U}^{p}$.

As shown in Equation 4 [9], Flag score is defined as twice the union of reference flags and predicted flags number minus the intersection, scaled by the max number of either reference flags or predicted flags. The range of flag score is between -1 and 1.

$$S_{F}^{p}(F_{\text{pred}}, F_{\text{ref}}) = \frac{1}{N} \left( 2 \times |F_{\text{pred}} \cap F_{\text{ref}}| - |F_{\text{pred}} \cup F_{\text{ref}}| \right)$$  \hspace{1cm} (4)

As shown in Equation 5 [9], Utility score is defined as the number of correct reference utilities scaled by capping flag score between 0 and 1, minus the number of wrong utilities, scaled by the max number of either reference utilities or predicted utilities. By summing all the utility scores within a predicted command, the range of normalized utility score is between -1 and 1.

$$S_{U} = \sum_{i \in [1, T]} \frac{1}{T} \times \left( U_{\text{pred}} = U_{\text{ref}} \times \frac{1}{2} \left( 1 + S_{F}^{p} \right) - \left| U_{\text{pred}} \neq U_{\text{ref}} \right| \right)$$  \hspace{1cm} (5)

The metric (normalized utility score) for evaluating a single prediction can be summarized as the following [6]:

1) Parameters are not taken into consideration
2) The order of flags affects the score
3) Wrong utility will result in negative points
4) The order of utilities does not affect the score
5) Predicting excessive flags will result in a penalty

**Energy.** The measurement and reporting of energy consumption of natural language programming (NPL) models is a relatively new phenomenon [44] [45]. As Henderson et al. [46] pointed out, part of the reason stems from the complexities of collecting the result. Indeed, according to Appendix B of Henderson et al. [46], out of 100 NeurIPS papers from the 2019 proceedings, only 1 measured energy consumption in some way, whereas 45 measured runtime performance.

To address this gap, the NeurIPS 2020 conference recommended “energy” as a more direct way of measuring environmental impact. We found the current energy metric used by the NL2CMD competition was not ideal, however, since it used estimated attributable power drawn (mWatts) to compute scores. This metric disproportionately punished models with less inference time.

For example, the GPT-2 model with an inference time of 14.87 seconds should have consumed a huge amount of energy (considering the model size). On the leaderboard in Table II the power metric is even less than the baseline, which is a much smaller model (GRU) and the inference is 3.24 seconds. Moreover, energy $mWh$ can be easily affected by trivially extending inference time. For example, by simply sleeping 3 seconds after each batch, the performance of a test submission can be improved from 682 to 88 on the leaderboard. A potential fix would be to measure the total energy consumed instead of the power since it punishes both bigger model size and longer inference.

**B. Error Analysis**

Previous research [4] listed sparse training data, utility errors, and flag errors as the top three causes of wrong predictions. Since sparse training data is a subjective metric, we decided to only analyze the incorrect utility and flag predictions. We used a separate, independently created testing dataset of 1,867 samples (previous work manually analyzed 100 samples from the dev dataset collected the same way as training dataset), and evaluated the accuracy results in more detail.

As shown in Figure 8, more than two-thirds of all errors are utility errors, which means the variety of flags is less significant than having enough data for each utility. As shown in Figure 9, among the top six incorrectly predicted utilities, `ls` and `grep` are the most frequently confused with `find`. This confusion is expected since the three utilities’ functionality have large overlaps and are among the most frequently used Bash Commands. By manually examining the incorrect predictions, we also found that these three utilities appear in many piped commands, which helps explain the large proportion they comprised in all the wrongly predicted utilities.

**VI. CONCLUDING REMARKS**

This paper evaluated various deep learning approaches to translating natural language into Commands in the Bash scripting language. Our results showed that Transformer-based models considerably outperform the RNN/LSTM-based models in the English-to-Bash translation task. Word vocabulary size and whether to post-process translations directly affected the results.

Our work is an initial step in building an automated system for translating natural language to Bash Commands. We improved the accuracy of this task from 13.8% to 53.2%. Our model is being integrated into IBM’s CLAI [6] skill catalog and to help hasten the adoption of intelligent terminals in production systems.

The following is a summary of the lessons learned from our study of translating natural language to Bash Commands:

- Models that use Transformers as both the encoder and decoder can achieve state-of-the-art accuracy in the natural language to Bash command translation task.
- Reducing word vocabulary size and tuning the positional weights empirically can improve accuracy scores when translating natural language to Bash Commands.
The RNN-based decoder model had lower accuracy, but reduced training time and inference time by roughly 50% compared to Transformers in the translation of natural language to Bash Commands.

A properly designed accuracy metric can measure the quality of a Bash command without the need to establish a simulated environment.

A properly designed energy metric can be an alternative to inference time as the metric to reflect the environmental impact of NLP models.

Our future work focusing on improving the current metrics and our model’s usability. A useful feature would be to translate the generated Bash back to natural language, thereby providing users with insight into why the model may have made translation errors. Acceptable inference time is also important due to the trade-off between accuracy and inference time, so a good metric should consider both for the best user experience. While longer inference time allows more accurate models—and thus enhances adoption—the inference time should have a limit, e.g., research by Nielsen et al. [47] shows that waiting times of more than 1.0 second leads to users feeling disengaged.

Our current model produces a generic representation instead of the actual values for arguments. We are therefore developing a better solution that adds a copying mechanism [48] to achieve a similar result for Bash as ValueNet [15] for SQL generation. Conversational Bash generators [49] are also a promising area to explore since they allow users to provide extra information to correct incorrect translations or clarify what they want. We are also exploring model deployment (either online or offline) since the energy consumption and environmental impact of our model will be magnified as more users adopt this type of command-line AI approach.

REFERENCES