Proceedings of Machine Learning Research 133:302–324, 2021 NeurIPS 2020 Competition and Demonstration Track

NeurIPS 2020 NLC2CMD Competition: Translating Natural Language to Bash Commands

Mayank Agarwal MAYANK.AGARWAL@IBM.COM
IBM Research

Tathagata Chakraborti TCHAKRA2@IBM.COM
IBM Research

Quchen Fu QUCHEN.FU@VANDERBILT.EDU
Vanderbilt University

David Gros DGROS@UCDAVIS.EDU
University of California, Davis

Xi Victoria Lin* VICTORIALIN@FB.COM
Facebook AI

Jaron Maene JARON.MAENE@GMAIL.COM
KU Leuven

Kartik Talamadupula KRTALAMAD@US.IBM.COM
IBM Research

Zhongwei Teng ZHONGWEI.TENG@VANDERBILT.EDU
Vanderbilt University

Jules White JULES.WHITE@VANDERBILT.EDU
Vanderbilt University

Editors: Hugo Jair Escalante and Katja Hofmann

Abstract

The NLC2CMD Competition hosted at NeurIPS 2020 aimed to bring the power of natural language processing to the command line. Participants were tasked with building models that can transform descriptions of command line tasks in English to their Bash syntax. This is a report on the competition with details of the task, metrics, data, attempted solutions, and lessons learned.

Keywords: Natural Language Processing, Bash, Programming, Code, Competition

1. Introduction

The command line interface (CLI) has long been an invaluable computing tool due to its expressiveness, efficiency, and extensibility. While graphical user interfaces (GUIs) struggle to keep up with the fast changing features in software development (for example, consider the time it took to move from Docker to Kubernetes in cloud platforms), CLIs provide a universal interface to almost any software feature via a combination of text-based commands and command arguments. Natural language based CLI interaction, if successful over a variety of command line tasks, has the potential to transform the way we interact with different

* Work done while the author was at Salesforce Research.

© 2021 M. Agarwal, T. Chakraborti, Q. Fu, D. Gros, X.V. Lin, J. Maene, K. Talamadupula, Z. Teng & J. White.

```
tchakra2-2:~ tathagata$ how do i compress a directory into a bz2 file
Try >> tar -cjf <archive-file> <directory>
tchakra2-2:~ tathagata$ tar -cjf file.tar.bz2 nlc2cmd-webapp/
tchakra2-2:~ tathagata$ ls
Box                 Documents           Pictures            file.tar.bz2          nlc2cmd-webapp
CLAI                Downloads           Public              get-started-python    nohup.out
Desktop             Library             bashbot-webpage     mai                   tarbot-router
tchakra2-2:~ tathagata$ extract files from archive into a directory
Try >> tar -xf <archive-file> -C <directory>
tchakra2-2:~ tathagata$ tar -xf file.tar.bz2 -C temp/
tar: could not chdir to temp/
tchakra2-2:~ tathagata$ mkdir temp
tchakra2-2:~ tathagata$ tar -xf file.tar.bz2 -C temp/
tchakra2-2:~ tathagata$ grep for all files in a directory with "port" in it, show details
Try >> grep -rv "port" <directory>
tchakra2-2:~ tathagata$ grep for all files in a directory with "port" in it, show line numbers and match case
Try >> grep -rni "port" <directory>
tchakra2-2:~ tathagata$ grep -rni "port" ./temp/nlc2cmd-webapp/
./temp/nlc2cmd-webapp//run.py:4:imports
./temp/nlc2cmd-webapp//run.py:6:from flask import Flask, request, render_template
./temp/nlc2cmd-webapp//run.py:7:import json, os
./temp/nlc2cmd-webapp//run.py:9:from ibm_watson import AssistantV2
./temp/nlc2cmd-webapp//run.py:10:from ibm_cloud_sdk_core.authenticators import IAMAuthenticator
./temp/nlc2cmd-webapp//run.py:85:    app.run(host='0.0.0.0', port=int(os.getenv('PORT', 3456)))
tchakra2-2:~ tathagata$ █
```

Figure 1: An example of a command line interface with natural language support, as described in Agarwal et al. (2020). Note that for normal Bash commands, execution proceeds as usual, while for tasks described in natural language, an `NLC2CMD` plugin intervenes (shown in green) with the translation of the described task into the corresponding command line syntax.

operating systems and cloud platforms, lowering the barriers to entry and increasing the accessibility of compute resources across the planet.

The `NLC2CMD` competition revolved around the simple use case: translating natural language (NL) descriptions of command line tasks to their corresponding command line syntax, thereby making the CLIs more accessible and more intuitive to use[1]. We compiled a large dataset consisting of the English (natural language) descriptions of tasks the users want to perform and their corresponding Bash[2] invocations (§ 4). The participants needed to build models and systems (§ 6) that generated the correct Bash command given new and unseen natural language invocations. To better measure the progress of this task and facilitate cross comparison of the submitted systems, we proposed new accuracy and energy efficiency metrics (§ 5). We also conducted a careful analysis of the winning systems and shared our suggestions for future work (§ 7).

2. Task Description

The `NLC2CMD` task aims to translate the natural language description (nlc) of a command line task to its corresponding Bash command (c). The algorithm (A) is expected to model the top-k Bash translations given the natural language description.

$$A \ : \ nlc \mapsto \{ \ p \mid p = \langle c, \delta \rangle \ \}; \qquad |A(nlc)| \leq k$$

Each prediction from the model is a set of Bash commands along with the confidence δ ($0.0 \leq \delta \leq 1.0$) in its prediction. In practice, this confidence estimation can be used to filter

1. A former position paper by the competition organizers on bringing the power of AI to the CLIs can be found in Agarwal et al. (2020).
2. https://www.gnu.org/software/bash/

out uncertain predictions, and it is factored into the competition evaluation. By default, the confidence score is set to 1.0, indicating full confidence in a model's prediction.

3. NLC2CMD: Competition Overview

The competition began in July 2020 and concluded at the end of November 2020. It was divided into three phases: Training, Validation, and Test. The Validation phase ran in October and the Test Phase ran in November. A total of 20 teams from around the world signed up for the competition by sending in a signed `Terms & Conditions` document - a pre-requisite for entry. Among these, 13 teams remained active through the end of the first phase. 9 teams went through the end of the Test Phase, and 6 of them open-sourced their solutions. The teams were allowed a maximum of 100 submissions in the first two phases, and a maximum of 10 submissions for the final phase; there were also limits on the number of daily submissions (5 for the first 2 phases, and 1 for the final phase). We hosted the competition using the EvalAI platform (Yadav et al., 2019).[3]

4. Data

4.1. NL2Bash

The NL2Bash dataset is a supervised natural language to Bash command dataset collected by Lin et al. (2018). It contains approximately 10k pairs of natural language task invocations and their corresponding command line syntax, covering over 100 commonly used Bash utilities.

4.2. Tellina Query Log

Tellina (Lin et al., 2017) is a recurrent neural network based architecture to translate natural language utterances to Bash commands. The system is also available as a web-based application to the public at large, and has been collecting data of natural language utterances from its users for quite some time now[4]. We collected nearly 1000 natural language utterances recorded from users' interactions with the Tellina system. Three programmers with prior Bash experience then annotated these collected examples: either through their own background knowledge, or by referring to the manual pages, or by searching over the Internet for guidance. Through this process, we generated multiple ground truth labels for a majority of the examples in this dataset; this was beneficial for the evaluation process since the metric used in the competition (Section 5.1.3) uses the maximum score over the predicted and the ground truth commands. Once this dataset was annotated, we filtered the data samples through the data filtering process described in section 4.4. After filtering, we were able to add around 700 examples to the NLC2CMD evaluation dataset.

4.3. NLC2CMD Data Collection Track

In addition to the main challenge of building an algorithm to translate a natural language utterance to a Bash command, we also ran a parallel data-collection track to collect natural

3. https://evalai.cloudcv.org/web/challenges/challenge-page/674/overview
4. http://tellina.rocks/

language to bash command pairs. These pairs were supplemental to the previous 2 data sources, and were collected through a web interface hosted on the competition website. Participants in this challenge were asked to submit as many pairs of English invocations and Bash commands as they wished. 21 participants from industry and academia submitted over 120 examples in this challenge. These pairs were filtered using the same process as used for the Tellina dataset, and the resulting examples (nearly 100) were part of the final phase of the challenge.

4.4. Data partitions and pipeline

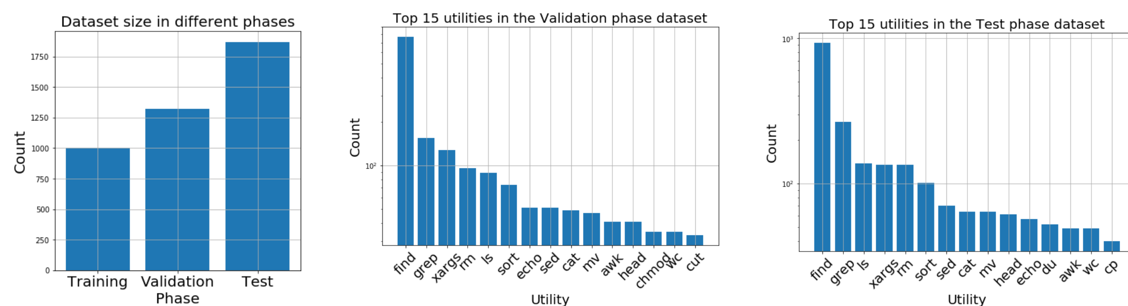
Data filtering We validate and filter the data collected through NL2Bash, Tellina Query Log, and the `NLC2CMD` Data Collection Track using the Bash parser used throughout the competition[5]. For each data sample pair of natural language invocation and its corresponding Bash command text, we parse the Bash command text through the parser to get its abstract syntax tree (AST) and then reconstruct the command text from the AST. We validate that the reconstructed command matches exactly to the original command text. This process ensures that any text which is not a valid Bash command – by either specifying a utility which is not part of the parser vocabulary, incorrect flags for the corresponding utility, or incorrect syntax altogether – is identified and omitted from the dataset. We also ensure that for a given command's text, all the utilities included in it are part of the Ubuntu 18.04 LTS command set[6]. Any command text that includes any utility that is not in the Ubuntu utility set is omitted.

Training For training, participants were provided with a filtered version of the `NL2Bash` (Lin et al., 2018) dataset, which is a supervised dataset of natural language descriptions mapped to their respective Bash commands. Participants were also provided with a version of the manual (man) pages for Ubuntu 18.04 LTS[7] to enable models to utilize information available in these man pages to learn and suggest commands unseen in the structured dataset. In addition to these two datasets, participants were free to use any freely-available dataset that they deemed fit to train their models, conditioned upon its disclosure and release to all competitors and the general public before the final stage of the competition.

Validation and Test We used a combination of data collected from sources described previously in this section to create the evaluation dataset. We create the test set for the first phase of the competition by randomly sampling 1000 data samples from a filtered version of the complete NL2Bash dataset (Lin et al., 2018) (explained in detail in the *Data filtering* paragraph above). This data split is different from that of the original NL2Bash data release; however, there may be some overlap in the data samples. The primary difference between the two datasets lies in the different choices of random seed and the number of test examples sampled.

In the subsequent phases, we add new data samples to the previous phase test set to create the test set of the following phase. 324 new data samples from the Tellina query log were added to create the the `validation` phase test set. To create the test set of the final

5. https://github.com/IBM/clai/tree/nlc2cmd/utils/bashlint
6. http://manpages.ubuntu.com/manpages/bionic/en/
7. http://manpages.ubuntu.com/manpages/bionic/en/



(a) Test dataset sizes across the 3 phases

(b) Most frequent utilities: validation phase dataset

(c) Most frequent utilities: test phase dataset

Figure 2: Dataset characteristics in the `NLC2CMD` competition

phase of the `NLC2CMD` competition, 543 additional data samples from the Tellina query log and the `NLC2CMD` challenge were added to the test set of the `validation` phase. Through the different phases of the competition, the size of the evaluation dataset grew significantly. With the addition of data from the Tellina query log and the `NLC2CMD` challenge, the evaluation data size increased from 1000 samples in the `training` phase to over 1800 in the `test` phase (see figure 2(a)).

To validate for algorithms' performances on Bash utilities not seen in the training data, we also added utilities which were not part of the NL2Bash training data in the `test` phase of the competition. While there were 101 utilities in the first phase of the competition, 27 new utilities were added through the Tellina data and the `NLC2CMD` challenge to bring the total to 128 utilities in the final phase. A distribution of the count of the top 15 utilities in the validation and test phase data samples is displayed in Figures 2(b) and 2(c) respectively.

We note that due to: (i) the random subsampling of the NL2Bash dataset to create the test set for the first phase of the competition; and (ii) the addition of data samples to this existing set to create the test set for subsequent phases – there is a possibility of algorithms overfitting by training on the publicly available NL2Bash dataset and inflating their scores. However, we decided on this format for the following reasons: a) NL2Bash is the only dataset available for the specific `NLC2CMD` task; since the other two data sources (Tellina query log, and `NLC2CMD` data challenge) weren't available right at the beginning of the competition, we had to utilize NL2Bash to create the initial test set in order to adhere to the competition schedule; and b) Adding samples to the initial test set to create the test set for the subsequent phases (instead of switching completely to the newly collected data samples) allowed us to control and prevent the underlying data distribution from shifting significantly between competition phases.

5. Metrics

We evaluated submissions to the `NLC2CMD` competition on two metrics: Accuracy, and Energy Consumption. In the following, we delve deeper into the specifics and the philosophy behind using two metrics to evaluate submissions to the competition.

5.1. Accuracy

In this section, we first discuss existing metrics used in prior work and their shortcomings in evaluating a structured prediction task such as NLC2CMD. We subsequently discuss a metric that verifies by execution. This metric fixes many shortcomings of the existing accuracy metrics, but requires considerable effort to implement. We briefly discuss the requirements to make this evaluation metric practical. Finally, we describe the metric we proposed for the competition.

5.1.1. Existing metrics

Full Command Accuracy: Accuracy metric or measuring exact match between the generated and reference sample has precedence in code generation research (Yin and Neubig, 2017; Chen et al., 2018; Fu et al., 2019). Specifically, the Full Command Accuracy metric was utilized by Lin et al. (2017, 2018) in their work on synthesizing bash commands given their natural language descriptions. The full command accuracy Acc_F^k is defined as the percentage of test instances for which a correct full command is ranked k or above in the model output. Because this metric measures the correctness of the complete command – command utility, utility flags, and the command ordering – it has a stricter notion of correctness. However, since Bash is a turing-complete language (Wikibooks, 2020), verifying the equivalence of two commands is undecidable (Lin et al., 2018); and therefore the chance of accurately measuring the correctness of the predicted command depends on how exhaustive the test set is. Theoretically, if the test set contains all the possible ways of achieving the task, this metric would accurately measure the predictors performance. However, this can be difficult, which is why the work of Lin et al. (2018) relied on human evaluations along with using this metric to measure their system performance.

BLEU score: BLEU score (Papineni et al., 2002) is a widely-used automatic machine translation evaluation metric that serves as an alternative to the otherwise time-consuming and expensive human evaluation. BLEU score is computed by comparing the n-grams of the candidate translations with the n-grams of the reference translation. We refer the reader to the work of Gatt and Krahmer (2018) for a detailed survey of evaluation methodologies in natural language generation and their characteristics.

Tran et al. (2019) study the effectiveness of using the BLEU score to evaluate the task of code migration, and conclude that the metric is not effective in reflecting the semantic accuracy of translated code. Additionally, they found a weak correlation between the BLEU score and human judgement of the semantic correctness of the translated code. Similar results were observed by Lin et al. (2017) in their study of using the BLEU score for the task of converting natural language to its corresponding bash syntax, and by Allamanis et al. (2018) in their broader work on the intersecting research areas of Machine Learning, Programming Languages, and Software Engineering.

Template Accuracy: The Full Command Accuracy matches the complete predicted command with the corresponding reference commands to measure the performance of the predictor. The full command includes the individual utilities that compose the command, along with their flags, arguments, and the ordering of the utilities in case the command is a composition of multiple utilities. While ideally it is desirable to predict the complete

command, in the normal development workflow, predicting the correct utilities and their flags is of greater importance than predicting the arguments as well. This is evidenced by the current user workflow, where the user searches manual pages, online forums, etc. for a way to accomplish their task, and then fills in their desired arguments once they've identified the appropriate utilities. Specifying the entire arguments is sometimes inconvenient for the user as well – consider specifying the full paths of files in case the user wants to move files from one location to another, when asking for only the utility is much easier in this case. To accommodate for this relative importance, Lin et al. (2018) propose the Template accuracy metric Acc_T^k. This is defined to be the percentage of test instances for which a correct command template is ranked k or above in the model output (i.e., ignoring incorrect arguments), and is also calculated via human evaluation (Lin et al., 2018).

5.1.2. Verification by Execution

Since Bash is a turing-complete language, verifying the equivalence of two commands to achieve the same task is undecidable (Churchill et al., 2019). Consider a simple case of listing all the files under a folder recursively. The 3 commands `tree /dirpath/`; `ls -R /dirpath/`; and `find /dirpath -print` all achieve the desired purpose but use different utilities. With the same utility, the commands could still differ by using different sets of flags. To account for such variation, we could execute the predicted command and the reference in a controlled shell environment and match the pre and post-execution outputs of the two commands[8]. A similar evaluation setup (verification by execution) is utilized by Lachaux et al. (2020) in their work on translation between programming languages; by Zhong et al. (2017) and Zhong et al. (2020) when mapping natural language to SQL queries; and by Guu et al. (2017) in their work on mapping natural language to logical forms in a synthetic setting.

The verification by execution setup requires a sand-boxed Bash environment where commands can run in a reproducible manner and without interference from other processes running on the machine. The test data is preferably command-specific to effectively rule out false positives. In addition, it is challenging to measure and compare the side effects[9] of the commands.

5.1.3. NLC2CMD metric

Having introduced existing accuracy metrics and their drawbacks in Section 5.1.1, we propose a metric that: a) ignores arguments in the predicted commands; b) considers the order of the utilities in case of piped commands; and c) penalizes predicting utility flags in excess of those defined in the ground truth commands.

We define $U(c)_i$ to be the ith utility in a command (`c`), and $F(u)$ to be a set of flags for a utility u. Given the predicted command C_{pred} and the reference command C_{ref}, we first define the flag score S_F^i to be:

8. This approach tends to overestimate accuracy, as a command and the reference may execute to the same results in a particular test environment while being semantically different. For example, if a directory contains only `.txt` files, `rm .` and `rm . -name ''*.txt''` will both leave an empty folder after execution. Zhong et al. (2020) mitigates this issue by testing the programs in multiple environments, thereby reducing the chance of a false positive.

9. https://en.wikipedia.org/wiki/Side_effect_(computer_science)

$$S_F^i(C_{\text{pred}}, C_{\text{ref}}) = \frac{1}{N}\Big(2 \times |F(U(C_{\text{pred}})_i) \cap F(U(C_{\text{ref}})_i)| - |F(U(C_{\text{pred}})_i) \cup F(U(C_{\text{ref}})_i)|\Big) \quad (1)$$

Here, $N = \max\big(|F(U(C_{\text{pred}})_i))|, |F(U(C_{\text{ref}})_i)|\big)$. The flag score rewards flags common to both the predicted and the reference command, but also penalizes excess predicted flags.

The normalized score for a single prediction $p = \langle C_{\text{pred}}, \delta\rangle$ is then computed as:

$$S(p) = \max_{C_{\text{ref}}} \sum_{i\in[1,T]} \frac{\delta}{T} \times \Bigg(\mathbb{I}[U(C_{\text{pred}})_i = U(C_{\text{ref}})_i] \times \frac{1}{2}\Big(1 + S_F^i\Big) - \mathbb{I}[U(C_{\text{pred}})_i \neq U(C_{\text{ref}})_i]\Bigg) \quad (2)$$

Here, $T = \max\big(|U(C_{\text{pred}})|, |U(C_{\text{ref}})|\big)$, and $\mathbb{I}[\cdot]$ is the indicator function.

Since the algorithm $A(nlc)$ predicts more than one candidate command to the natural language query nlc, the overall score is computed as:

$$Score(A(nlc)) = \begin{cases} \max_{p\in A(nlc)} S(p), & \text{if } \exists_{p\in A(nlc)} \text{ such that } S(p) > 0 \\ \frac{1}{|A(nlc)|}\sum_{p\in A(nlc)} S(p), & \text{otherwise} \end{cases} \quad (3)$$

This scoring mechanism incentivizes the precision and recall of the correct utility and its flags, weighted by the algorithm's reported confidence.[10] The reasons for choosing this metric are many: 1) It can be evaluated within the the timeline of a competition at a conference;[11] 2) Synthesizing the full command is a more difficult task and we wanted to start from a simpler option given this is the first time the competition is running; 3) In some cases it is challenging to fully specify the command arguments in the natural language; 4) A system that suggests a command with argument placeholders can be useful since once the command structure is known, the user may fill the missing argument values in an interactive manner with an AI assistant on the terminal like CLAI (Agarwal et al., 2020). We provide examples illustrating the behavior of this metric under different ground truth and predicted command variations in appendix A.

5.2. Energy Efficiency

Evaluating and mitigating the energy consumption of deep learning models has recently gained traction in the research community. Li et al. (2016) studied the energy efficiency of

10. Note that an initial version of the metric naively computed the maximum score for the top-k predictions from the algorithm. Since the metric considers both the predicted command as well as the confidence measure, it was possible to artificially clamp the metric to zero instead of its normal range of $[-1, 1]$. This could be achieved by predicting the last of the k commands with 0.0 confidence, thus ensuring a score of 0.0 for the final prediction; and clamping the overall metric minimum to 0.0 as well. This was fixed by modifying the aggregation of the individual scores of the metric by using a `max` operator only if at least one of the prediction scores is greater than 0; and an average otherwise.
11. While similar competitions have looked to implement unit tests for the test data before (Hendrycks et al., 2021), for us, the test set was not fixed apriori but only determined during the course of the NLC2CMD Challenge; this, along with the fact that the testing scheme for the competition without arguments leads to commands that are under-determined, meant that the simulation option was out of scope.

convolutional neural networks, while more recently, the work of Strubell et al. (2019) evaluated the energy consumption of developing deep learning models for NLP tasks. Strubell et al. (2019)'s findings suggested that the CO_2 emissions from training an NLP pipeline (with tuning and experimentation) is greater than the average yearly emission of an American life. In the same spirit, the work of Schwartz et al. (2019) distinguish between Red AI and Green AI. Red AI refers to the pursuit of obtaining state-of-the-art results in accuracy (or related measures) through the use of massive computational power. In contrast, Green AI refers to AI research that yields novel results without increasing computational cost, and ideally reducing it. The authors analyzed 60 randomly sampled papers from the ACL 2018[12], CVPR 2019[13], and NeurIPS 2018[14] conferences to find that a large majority of papers target accuracy or some related measure, and thus, Schwartz et al. (2019) advocate for making efficiency a research criteria along with accuracy to make AI greener. Their measures of efficiency include carbon emissions, electricity usage, among others. Since then, Lottick et al. (2019), Lacoste et al. (2019), and Henderson et al. (2020) have proposed tools to help researchers evaluate and report the energy consumption requirements of their work. Additionally, recent initiatives such as the SustainNLP[15] workshop at EMNLP 2020[16] have focused on encouraging development of efficient NLP models, and providing simpler model architectures and empirical justification of model complexity.

While the aforementioned work focuses on the energy consumption of models during the training phase, with the deployment of these models, their energy consumption in inference phase can outweigh their training cost over time. This dichotomy between the energy cost during training in machine learning research and the energy cost during inference in production systems, and the need to also measure the inference energy cost is highlighted by Henderson et al. (2020) and Bender et al. (2021). In accordance with the recommendations of Green AI and other recent work in the energy-efficient machine learning research, we measure the energy consumption of submissions to the `NLC2CMD` challenge using the `experiment-impact-tracker`[17] library by Henderson et al. (2020). The Efficiency track at the `NLC2CMD` competition was run in parallel to the Accuracy track and aimed to award the most energy-efficient model amongst the top-5 scoring (by accuracy) models.

6. Competing Solutions

Table 1 provides a snapshot of the `NLC2CMD` competition leaderboard at the end of the final (test) phase: a total of 6 teams/entries and 2 baselines made it to this stage.[18] The leaderboard captured the accuracy score, energy consumption, and latency of the entries. We also provide examples of the natural language invocation along with the ground truth command and the commands predicted by the top 3 teams in appendix B. In the following, we provide brief overviews of the techniques adopted by these entries.

12. https://acl2018.org/
13. https://cvpr2019.thecvf.com/
14. https://nips.cc/Conferences/2018
15. https://sites.google.com/view/sustainlp2020/
16. https://2020.emnlp.org/
17. https://github.com/Breakend/experiment-impact-tracker
18. Note that the Tellina was used "as is" for the competition as a baseline and was not retrained for the new metric so as to not make the competition task (running for the first time) too hard for participants.

Team Name	Accuracy Score	Energy (Joules)	Latency (sec)
Magnum	**0.532**	0.484	0.709
Hubris	**0.513**	12.037	14.870
Jb	0.499	2.604	3.142
AICore	0.489	**0.252**	0.423
AINixCLAISimple	0.429	N.A.	0.010
coinse-team	0.163	343.577	0.452
Tellina	0.138	2.969	3.242

Table 1: Final leaderboard for the `NLC2CMD` competition, showing the accuracy score for the final (test) phase, along with the energy consumed and latency for every invocation.

IR-Baseline Variation	Accuracy Score
Tf-IDF Raw	0.361
+ AInix Data	0.404
+ Prune Duplicates	0.413
+ Normalize NL	0.429
+ Adjust Conf.	0.472

Table 2: Results from simple IR baselines. Additions to the raw predictor are retained cumulatively top-to-bottom.

6.1. TF/IDF and Proposed New Baselines

Team `AINixCLAISimple` briefly explored some simple baselines on the task. Exploration of simple baselines might evaluate if the metrics or task are too easily "gameable". This team did not find a simple technique that could achieve the highest accuracy score, but did vastly outperform the original host Tellina baseline at very high efficiency.

The primary explored approach was an information retrieval (IR) approach using Tf-IDF rankings. Several variations were explored (Table 2). In all variations the NL-CMD training pairs were first indexed using the Vec4IR package (Galke et al., 2017). In the simplest variation (`Tf-IDF Raw`) the team took the evaluation utterances, queried the training, and returned the top 5 highest Tf-IDF matches each with a predicted confidence of 1.0. This approach had a score of 0.361, which is approximately 2.6x higher than the Tellina baseline.

Several additions were made. The inclusion of the AInix Archie data[19] as training data boosts the score to 0.404. Additionally, the team notes there might be duplicates of a given command in the training data (especially considering the constant arguments of the command are ignored). Because the metric takes a max, it is not reasonable to include duplicates in the five outputs. A gain of about 0.01 is achieved by pruning duplicates, instead taking the next ranked retrieved commands, up to five unique commands. A further 0.016 points can be gained by applying some string matching heuristics to normalize NL tokens which appear to be constants (file names, numbers, IP addresses, etc).

Finally, an approach was used to lower points deductions on difficult NL utterances. A logistic regression model was fit to predict the probability the prediction would result in a positive score. It used features like the Tf-IDF score, number of flags in the retrieved command, and number of utilities. These logistic regressions were used in combination with the estimated probability that all predictions will score negatively in order to potentially lower the outputted confidence to be less than 1.0. This resulted in a 0.043 point gain.

This team also explored a method of "optimally diverse picking". This was based off the observation that the metric considered the maximum score, so it might be optimal to return a diverse range of commands in the hopes at least one receives a positive score. Various approaches were explored for optimizing pick diversity, but it was found that when paired the Tf-IDF model with poorly calibrated score estimates, this approach did not help performance. A better underlying model might make the approach more useful.

19. https://github.com/DNGros/ai-nix-kernal-dataset-archie-json

The best performing model under the Tf-IDF approach achieved an accuracy score of 0.472.[20] The latency was 10ms or less, and the energy use was low enough that the measurement tool errored out. This model vastly outperforms the host neural baseline, and comes within 12% of the best performing model. On top of the efficiency and simplicity benefits, the team notes that this approach has UX advantages compared to the `Seq2Seq` models, as retrieval-based NL2Bash models are able to provide exemplar-based explanations (Gros, 2019). This can help users reason about the reliability of a given system prediction.

6.2. Transformer with Beam Search

Team `Magnum` achieved an accuracy score of 0.532 on the leader board, which is the current state-of-the-art on the `NLC2CMD` challenge. The final model is an ensemble of 5 separately-trained transformer (Vaswani et al., 2017) models consisting of 6 layers with Beam Search enabled. Team Magnum used the following key strategies in their model:

1. When Bash commands were parsed into syntax trees, the arguments (parameters) were replaced with their generic representations for tokenization. As a result, the word vocabulary size was reduced from 7,070 words to 719. The placeholder `unk` was filtered out for better usability.
2. Scores in the beam search were produced using an approximation[21] for confidence due to the high correlation observed between correct predictions and high beam scores.
3. A variety of combinations of hybrid encoders and decoders were tested. Prior research (Tang et al., 2018) indicates hybrid encoder-decoder architectures have potentially better performance in certain NLP problems. For example, RNN encoders show better performance on conditional language modeling tasks while Transformer models are better at feature extraction (Tang et al., 2018).

The results show that in the `NLC2CMD` challenge, using a Transformer as both an encoder and decoder performs best in terms of accuracy, while an RNN as the decoder can reduce training and inference time (50% less inference time). The empirical advantages of a Transformer over an RNN can be attributed to the self-attention of the Transformer that reduces locality bias and the model parallelism that allows for more efficient GPU-based computation (Vaswani et al., 2017).

6.3. Fine-tuned GPT-2 in ensemble

Team `Hubris` fine-tuned pre-trained transformer models on the `NLC2CMD` task, given their recent dominance in NLP tasks, and machine translation in particular. More specifically, the GPT-2 architecture was chosen (Radford et al., 2019), partially inspired by the fact that its larger incarnation GPT-3 had shown indications[22] of being able to perform `NLC2CMD` in a few-shot setting. Furthermore, the GPT models are pre-trained on a diverse set of sources, in contrast to models with a more specialized sequence to sequence architecture like T5 (Raffel et al., 2019), which did not have instance pre-trained on both natural language and code, at time of writing.

20. Note, due to participant user error, the "+ Adjusted Conf" model failed to upload to the evaluation server; thus the scoreboard reflects a model similar to the "+ Normalized NL" model.
21. $\text{Confidence}_i = \left(e_i^{\text{BeamScore}}\right)/2, 2 \leq i \leq 5$
22. https://beta.openai.com/?app=productivity&example=4_2_0

As the goal of the competition was template prediction, and not full command prediction, the data was pre-processed to replace concrete arguments with dummy tokens. The NL2Bash dataset was also augmented with heuristically mined data from stack-overflow questions and some minor sources like the AInix project (Gros, 2019).

To improve performance, two models of differing size and pre-training (to decrease correlation) predicted the commands in ensemble. Due to the nature of the metric, a high degree of diversity in the 5 provided output commands was greatly desirable. However, conventional beam-search generated highly correlated commands, while sample-based methods entailed too sharp a drop in the syntactic coherence of the commands. The issue was resolved by letting the two different models generate commands using a higher number of beams. The final commands were selected from the resulting set by a heuristic algorithm that tried to maximize the minimal word distance between individual commands.

As the confidences given by the GPT-2 models were not calibrated and usually overconfident, the Hubris team just gave every command confidence 1.

6.4. Multi-Step Pipelines

One popular approach considered by multiple entries and teams was the notion of multi-step pipelines, which ensembled two different models for two very distinct purposes. The key insight behind this split was the shared observation that a given instance of the `NLC2CMD` task essentially consists of two different decisions: predicting the best utility for accomplishing that task, and then figuring out the right set of flags to use. Team `jb` (JetBrains) (Litvinov et al., 2020) accomplished this by first using a classifier to predict the utility and set up a context; and then using a seq2seq transformer (Vaswani et al., 2017) with that context to predict the entire command. Team `coinse` (Kang and Yoon, 2020) took a hierarchical approach: their hierarchical decoder, inspired by prior work on the two-stage image generating StackGAN (Zhang et al., 2017), consists of a utility predictor as well as a flag predictor.

7. Discussion

A valuable outcome of a public competition is to arrive at a better understanding of the nuances of the task and find better ways to solve and measure it. This section compiles lessons learned and discussion with participants during the course of the challenge; and with the audience at the NeurIPS 2020 Competition Session and Virtual Room.

7.1. Metrics Revision

7.1.1. Suggested Alternatives for Accuracy Measurement

The following is a brief recount of the various suggestions we received during the conference for enhancing the accuracy metric in future iterations.

Semantic Match This involves making the parser more intelligent to be able to match the semantics of a task rather than the exact command and its flags. This is is not only to do with the fact that the same task can be performed using several different utilities, but also in different ways using the same utility: e.g. exec and pipes on a `find` command.

Restricted Coverage The current top performing algorithms on the NLC2CMD task are still quite low for deployment in real applications. However, the frequency of commands in the data (derived from real world sources) does follow a heavily tailed distribution. This brings into question whether one should even look for coverage. One reason we had the entire set of commands in the test set for the competition was to challenge algorithms to go beyond the NL2Bash data and include a mixture of examples as well as structured data like manual pages to help generalize (though no final participants made use of the man pane data). But a valid argument can be made in favor of reducing the coverage so that algorithms are more accurate over a more restricted and frequently used set of commands than less accurate over many more commands.

This also reduces the problem of data sparsity. Conventional ML methods might struggle on the full problem as the amount of data available for some bash commands is very low. A similar approach was taken by the text-to-SQL community where the popular benchmark datasets only include a subset of SQL grammar that covers the most practical use cases (Yu et al., 2018). In addition, Gros (2019) attempted to focus on a narrower subset of Bash commands while attempting to get usable accuracies.

IR-based Metrics We can also look at more standard scores like mean reciprocal rank (MRR) and mean average precision (MAP) to further streamline this, especially in relation to scoring the top-K response effectively. MRR has been adapted for other NL2Code tasks like CodeSearchNet (Husain et al., 2019). In case of multiple valid outputs, there are often preferences. For example, the phrase *"list my files here"* it is valid to do: `ls`, `ls -l`, `ls -la`, `ls -lta`, etc. where the later ones are likely less preferable. Going forward, it will be interesting to look at IR metrics that can deal with such situations where with multiple valid results, some are more conventional than the others. Moreover, while the current setup was designed purely on consideration of how the orchestration layer in CLAI works (c.f. Section 3.2 from Agarwal et al. (2020)), it is unclear how the confidence marked accuracy scores will play out in practice when multiple choices are presented to the end user, It might make more sense to instead penalize suggestions in sequence since that is how the user is probably going to try them out. Baking in patterns of interaction in the evaluation model would be an interesting addition to the evaluation metric.

Session Score Another interesting idea that came up in terms of measuring accuracy in the context of user interactions was to measure accuracy over multiple interactions in a session instead of averaging over several independent translations. This gives us the ability to measure how algorithms behave over time.

> **Adaptability** One particular type of evolution we can look for is how algorithms calibrate themselves based on whether they are getting their translations right or which ones the user is picking up on, and thereby adjusting their confidences or even their suggestions over time.
>
> **Fast (Re-)Training** Another measure of interest here is a "fast training" measure i.e. how easy it is for the user to add a new command or new phrasing, and be confident that the system will be able to incorporate the new knowledge quickly and/or will not mess up again in the future on the same task. Typical deep learning models might struggle with this lifelong learning as it requires care to avoid catastrophic forgetting

when adding a new task (Chen and Liu, 2018). Thus non-parametric retrieval-based models may have advantages.

Calibration of Penalties Continuing the theme of changing penalties of misclassification based on user feedback or continued misclassification, there are couple of other interesting factors at play here as well.

Statefulness of commands A command that changes the state of the machine has much more impact on being wrong as compared to those that do not. This should affect the penalty of a wrong suggestion.

Command injection and privacy concerns A curious case of building algorithms off data is the ability to inject hacks into the system by manipulating the data. GPT-3, for example, has some very severe privacy and security concerns of this type based on the data it brings up Carlini et al. (2020). For the Bash, examples of this already exist in terms of hacked aliases Zhong (2021); Shah (2020). While it is harder to evaluate this in the scope of a competition, looking out explicitly for state changes may serve as a first line of defense.

Full Text Match and Underdetermined Invocations While we focused only on templated match in this competition, the full scope of the problem is, of course, a full translation of the command. This may be a much more challenging task but it may also be more interesting from a longer-term research point of view. The template prediction also elided details of the command that are more complicated than just paths (e.g. `-type f`, or regex, some formatting options, etc). This could mean the generated command is not always useful to the user – it is not just a matter of filling in the placeholder parameters.

However, there are additional pitfalls of full text matches of commands beyond just the increased difficulty of doing it. This is to do with the fact that natural language command invocations can often be underdetermined. For example, saying *"add my username to this user group"* can be translated to the utility and flags with parameters as placeholders but the full text response to this may also be interpreted as a piped execution that first determines those parameters, in this case the username. Additionally, oftentimes the user may be referring to a generic file in which case again, the commands will be underdetermined. This is something to keep in mind in case there is intent to change the accuracy metric to a full text match later, since it does change the semantics of the task.

7.1.2. Suggested Alternatives for Energy Measurement

Initially planned as an efficiency track – most easily measured as the average time taken to make a translation – we later shifted this to an energy or power consumption upon feedback from NeurIPS reviews. While we ended up using a standard power measurement package along with a few other competitions at the conference, it has many justifiable concerns.

Latency Attacks on Power Measurement A key problem with measuring power is that an application can slow down the computation to reduce peak power consumption. Team `Magnum` found the energy track in the NLC2CMD competition could be attacked by spuriously appending sleep functions to suspend the inference process and dramatically

increase inference time. When the inference time is slowed down, the estimated attributable power draw (mWatts) – the evaluation metric of the energy track – will decrease.

Instead of measuring power, a solution would be to be shift to measuring total energy consumption instead, to take into account both time and power. The proposed energy consumption metric addresses the problems with the attack described above, but it does not capture an important consideration, which is the user experience of a model that supports user-interaction with a terminal. For example, once the inference latency falls below a certain threshold, the latency will be imperceptible to the user and further improvements will not be as beneficial. A better approach may be to measure energy with a set of non-linear profiles that incorporate user-perceptible latency in the scoring, as was done in the CLAI user study (c.f. Appendix 4.5 from Agarwal et al. (2020)).

Significance More importantly, there was a lot of debate during the competition and at NeurIPS on whether there is any point to measuring energy at all. Arguments against include the amount of energy consumed itself, and whether we should be spending effort in measuring training energy instead. In making an accuracy-efficiency trade-off, it is also necessary to take this holistic view. A wrong command could have costly side effects (like accidentally removing a folder). Saving a second of human time has a much greater CO_2 pay-off than saving the same CPU time (not to mention that human's well being). Additionally, the wrong command might potentially begin a far longer computation. This means that erring on the side of inefficient models can actually be less energy efficient in the big picture.

Finally, the efficiency track was primarily focused on power consumption and did not actually consider the latency measure in isolation. While a shift to an energy measurement will implicitly take this into account, latency is also the most user-centric metric; i.e. it is unlikely a user will use use an NLC2CMD service on the shell (versus e.g. StackExchange) if the latency is too large. Additionally, relying purely on energy measures creates indirection when trying to identify a problem like a model using excessive memory.

7.2. Other Enhancements

Communicating Explanations One aspect that we did not explore during the competition was what happens after the Bash command translation is provided to the user – i.e. explaining the translation to the user. Since users may not be experts in all Bash utilities, they may not understand how to select the most appropriate translation or the ramifications of different translations. Explanations of translations could include Bash visualizations of the AST, for example. One possible option is to employ Back-translation (Edunov et al., 2018) to convert commands back to natural language to explain the system's suggestion.

Conversational Interfaces Recent work on Text-to-SQL semantic parsing, such as the Photon (Zeng et al., 2020) system demonstration, has shown conversational code generation can help people accomplish tedious tasks using natural language. Conversational interfaces can be more approachable for average users and also allow users to provide extra information to correct translations or clarify what they want by providing more context.

8. Conclusion

This concludes a brief post-conference report on the first ever natural language on Bash competition NLC2CMD @ NeurIPS 2020. Going forward, we will look towards incorporating the feedback received at the conference, particularly in terms of revising the metric to take into account a session score with downstream effects and other interaction considerations like latency profiles and sequence of suggestions (as outlined in §7). We were also delighted to welcome on board some of the participants from the NeurIPS edition to the NLC2CMD team to help write this report and design the next iteration of the competition. More details on future iterations can be found at: `ibm.biz/nlc2cmd`.

References

Mayank Agarwal, Jorge J Barroso, Tathagata Chakraborti, Eli M Dow, Kshitij Fadnis, Borja Godoy, and Kartik Talamadupula. CLAI: A Platform for AI Skills on the Command Line. *arXiv:2002.00762*, 2020.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A Survey of Machine Learning for Big Code and Naturalness. *ACM Computing Surveys*, 2018.

Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the Dangers of Stochastic Parrots: Can Language Models Be Too Big? In *ACM FAccT*, 2021.

Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, et al. Extracting Training Data from Large Language Models. *arXiv:2012.07805*, 2020.

Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree Neural Networks for Program Translation. In *NeurIPS*, 2018.

Zhiyuan Chen and Bing Liu. Lifelong Machine Learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 2018.

Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1027–1040, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314596. URL https://doi.org/10.1145/3314221.3314596.

Sergey Edunov, Myle Ott, M. Auli, and David Grangier. Understanding Back-Translation at Scale. In *EMNLP*, 2018.

Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An End-to-end Neural Program Decompiler. *NeurIPS*, 2019.

Lukas Galke, Ahmed Saleh, and Ansgar Scherp. Word Embeddings for Practical Information Retrieval. In *INFORMATIK*, 2017.

Albert Gatt and Emiel Krahmer. Survey of the State of the Art in Natural Language Generation: Core Tasks, Applications and Evaluation. *Journal of Artificial Intelligence Research*, 2018.

David Gros. AInix: An Open Platform for Natural Language Interfaces to Shell Commands. 2019.

Kelvin Guu, Panupong Pasupat, Evan Liu, and Percy Liang. From Language to Programs: Bridging Reinforcement Learning and Maximum Marginal Likelihood. In *ACL*, 2017.

Peter Henderson, Jieru Hu, Joshua Romoff, Emma Brunskill, Dan Jurafsky, and Joelle Pineau. Towards the Systematic Reporting of the Energy and Carbon Footprints of Machine Learning. *arXiv:2002.05651*, 2020.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv:2105.09938*, 2021.

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

Sungmin Kang and Juyeon Yoon. Hierarchical Decoding of Bash Commands. Technical report, 2020.

Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised Translation of Programming Languages. *arXiv:2006.03511*, 2020.

Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. Quantifying the carbon emissions of machine learning. *arXiv preprint arXiv:1910.09700*, 2019.

Da Li, Xinbo Chen, Michela Becchi, and Ziliang Zong. Evaluating the Energy Efficiency of Deep Convolutional Neural Networks on CPUs and GPUs. In *IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom)*, 2016.

Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, Luke Zettlemoyer, and Michael D. Ernst. Program synthesis from natural language using recurrent neural networks. Technical Report UW-CSE-17-03-01, University of Washington Department of Computer Science and Engineering, 2017.

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System. In *LREC*, 2018.

Denis Litvinov, Gleb Morgachev, Artem Popov, Nikolai Korolev, and Dmitrii Orekhov. NLC2CMD Report from JB Team. Technical report, 2020.

Kadan Lottick, Silvia Susai, Sorelle A Friedler, and Jonathan P Wilson. Energy Usage Reports: Environmental Awareness as Part of Algorithmic Accountability. *arXiv:1911.08354*, 2019.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: A Method for Automatic Evaluation of Machine Translation. In *ACL*, 2002.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 2019.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *arXiv:1910.10683*, 2019.

Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. Green AI. *arXiv:1907.10597*, 2019.

Shrey Shah. Shellshock – High Voltage, 2020.

Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and Policy Considerations for Deep Learning in NLP. In *ACL*, 2019.

Gongbo Tang, Mathias Müller, Annette Rios, and Rico Sennrich. Why Self-Attention? A Targeted Evaluation of Neural Machine Translation Architectures. *arXiv:1808.08946*, 2018.

Ngoc Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien Nguyen. Does BLEU score work for code migration? In *ICPC*, 2019.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, L. Kaiser, and Illia Polosukhin. Attention is All you Need. *arXiv:1706.03762*, 2017.

Wikibooks. Bash shell scripting, 2020.

Deshraj Yadav, Rishabh Jain, Harsh Agrawal, Prithvijit Chattopadhyay, Taranjeet Singh, Akash Jain, Shiv Baran Singh, Stefan Lee, and Dhruv Batra. EvalAI: Towards Better Evaluation Systems for AI Agents. 2019.

Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. *arXiv:1704.01696*, 2017.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir R. Radev. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii, editors, *EMNLP*, 2018.

Jichuan Zeng, Xi Victoria Lin, S. Hoi, R. Socher, Caiming Xiong, Michael R. Lyu, and Irwin King. Photon: A Robust Cross-Domain Text-to-SQL System. *arXiv:2007.15280*, 2020.

Han Zhang, Tao Xu, Hongsheng Li, Shaoting Zhang, Xiaogang Wang, Xiaolei Huang, and Dimitris N Metaxas. Stackgan: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks. In *ICCV*, 2017.

Ruiqi Zhong, Tao Yu, and Dan Klein. Semantic evaluation for text-to-sql with distilled test suites. *CoRR*, abs/2010.02840, 2020. URL https://arxiv.org/abs/2010.02840.

Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating Structured Queries from Natural Language Using Reinforcement Learning. *arXiv:1709.00103*, 2017.

Weilin Zhong. Command Injection, 2021.

Appendix A. Examples of proposed metric computation

In this section we look at the behavior of the metric used to score the submissions in the `NLC2CMD` competition (section 5.1.3). In Table 3, we provide some example behaviors of the metric, under different ground truth and predicted command variations.

1. Parameters are not taken into consideration	
Ground truth	`df	grep /dev/disk0s2`
Predicted command	`df	grep diskpath`
Metric value	$1.0 \times \delta$
2. Order of flags does not matter	
Ground truth	`find . -regextype posix-egrep -regex REGEX -type f`
Predicted command	`find . -type f -regextype posix-egrep -regex REGEX`
Metric value	$1.0 \times \delta$
3. You get negative points if you translate to the wrong utility	
Ground truth	`mkdir directory`
Predicted command	`touch directory`
Metric value	$-1.0 \times \delta$
4. Order of utilities matter	
Ground truth	`df --total	tail -n 1`
Predicted command	`tail -n 1	df --total`
Metric value	$-1.0 \times \delta$
5. Predicting excessive flags gets negative penalty	
Ground truth	`find / -name linux`
Predicted command	`find / -EXdsx -name linux`
Metric value	$0.1666 \times \delta$

Table 3: Examples of the proposed metric computation and its behavior under different ground truth and predicted command variations. Here, δ is the confidence predicted along with the command by the model.

Appendix B. NLC2CMD Model Outputs

In Table 4, we present some examples from the test set – which was the data partition used to determine the final leaderboard of the competition. Along with the natural language invocations, we also present the ground truth command and the first prediction from each of the top 3 scoring teams – Magnum (section 6.2), Hubris (section 6.3), and jb (section 6.4). We note that during the evaluation, we requested top-5 predictions from each of the models, and aggregated the score over these 5 predictions. In Table 4 however, we show only the top prediction (from the 5 requested) from each of these models. From the listed examples and predictions, we can derive some interesting qualitative insights:

1. Submitted models are a) able to predict `find` commands pretty well, and b) prefer the `find` command to solve problems easily solved by other commands (see prompts A, B). This is explainable through the dominance of `find` commands in the training, validation, and test sets (see figures 2(b) and 2(c)).

2. Models do not necessarily possess an adequate understanding of the effects of utility flags (see prompts C, D). For example, for prompt C, while the models correctly predict the utility (`cp`), the intended task would not succeed without the use of the `-R` flag.

3. The ability to successfully operate on the Bash terminal goes beyond the understanding of utilities and flags. Certain files or directories hold special significance when operating on the terminal. In prompt E for example, knowing that the `etc/passwd` file holds the lists of users is critical to fulfilling the corresponding prompt. Models currently do not encode this knowledge, and therefore fail to provide a valid response to these queries.

A) Prompt	`Find all empty files starting from the current directory and delete them`
Ground truth	`find .  -type f -empty -print0 | xargs -0 /bin/rm`
Team Magnum	`find Path -type f -empty -delete`
Team Hubris	`find Path -maxdepth Quantity -type f -empty -delete`
Team jb	`find -type ARG -empty -delete`

B) Prompt	`copy "install.txt" file to home folder`
Ground truth	`cp "install.txt" "~/"`
Team Magnum	`find Path -name Regex -exec cp  File \;`
Team Hubris	`find Path -name Regex | xargs -I  cp -a --target-directory Directory --parents {}`
Team jb	`find -name ARG | xargs -I ARG cp -a --target-directory ARG --parents`

C) Prompt	`copy a folder`
Ground truth	`cp -R "string" "string"`
Team Magnum	`cp File File`
Team Hubris	`cp File File`
Team jb	`cd`

D) Prompt	`Print the hexadecimal bytes and printable characters of "Hello world"`
Ground truth	`echo Hello world | od -t x1 -t c`
Team Magnum	`echo Regex Regex Regex | od -t x1 -t c`
Team Hubris	`echo -n Regex | od -c`
Team jb	`echo -n Regex | od -c`

E) Prompt	`list the users on the system`
Ground truth	`cat etc/passwd`
Team Magnum	`w`
Team Hubris	`who | awk Program`
Team jb	`who | sed -e ARG`

F) Prompt	`list of all open files and the processes that opened them`
Ground truth	`lsof`
Team Magnum	`ps -e -f | tee File`
Team Hubris	`more File`
Team jb	`ls -l -t`

G) Prompt	`Save the user name in all capitals of the current user to variable "v"`
Ground truth	`v=$(whoami | awk 'print toupper($0)')`
Team Magnum	`whoami | awk Program`
Team Hubris	`whoami | awk Program`
Team jb	`whoami | awk`

I) Prompt	`Echo each command before running`
Ground truth	`set -x`
Team Magnum	`set -x`
Team Hubris	`set -x`

Team jb	`set -o ARG`
J) Prompt	`Change the owner of "process" to "root"`
Ground truth	`sudo chown root process`
Team Magnum	`chown Regex File`
Team Hubris	`chown Regex File english: Change the owner of all ".txt" files in directory tree "/mydir" to "root" using at most 20 files at a time`
Team jb	`chown`
K) Prompt	`kill application`
Ground truth	`ps -A | grep -i "application" | awk 'print $1' | xargs sudo kill -9`
Team Magnum	`kill $( jobs -p )`
Team Hubris	`ps -e -f | grep Regex | grep -v Regex | awk Program`
Team jb	`yes`

Table 4: Sample data points from the test dataset. The natural language prompt along with the ground truth and the first prediction (from the 5 requested) from the top 3 models of the competition are presented in the table.