Oisin Doherty (oisind),
Abhinav Gottumukkala (anak4569),
Hans Jorgensen (thehans),
Lauren Martini (lmartini)
CSE 403
Project 10: Final Report

NL2Bash

Expanding upon existing data sets to improve translation quality

Objective

We propose to increase the accuracy of the English to Bash translator, Tellina, by gathering new training data and designing a validation system to improve the quality of both new and existing data. Because Tellina is built upon machine learning, more data and higher quality data should improve overall accuracy.

Background and Motivation

Bash can be a difficult language to learn. Both new and veteran developers have experienced the confusion that comes from searching through numerous resources for Bash commands online; Even tasks that can be described simply in English, such as, “Count all the lines of php code in a directory recursively” (https://stackoverflow.com/questions/1358540/) may require the use of several different bash commands where each individual command may not be intuitive. For such a query, one command that would work is:

```
find . -name '*.php' | xargs wc -l
```

For users unfamiliar with Bash, this is a difficult command to understand. The user would have to understand how to use the `find, wc,` and `xargs` commands, and the appropriate flags to count only php files and count lines. They would also have to know how to redirect output with the pipe operator. Especially for a beginner, the amount of knowledge required to use each command may make even simple operations difficult.

In order to bridge this gap between Bash and human understanding, efforts have been made to develop tools to convert natural language into Bash commands. A converter able to translate any valid natural language command into a Bash command would be incredibly beneficial, both for those using bash and those learning it. Even Bash veterans occasionally forget a command or encounter a task that requires a command they have not used before. In those situations, a natural language to Bash converter could provide a quick solution, allowing them to complete tasks more quickly. Individuals learning bash would have an easy way to look up the commands required to complete a given task, which will greatly improve the rate at which they encounter and learn new commands.

Unfortunately, current efforts to develop natural language-to-Bash converters have not yet lived up to their full potential.[1] The main tool examined for this project is Tellina, a natural language programming model that is available at https://github.com/TellinaTool/nl2bash and as an inbrowser tool at http://kirin.cs.washington.edu:8000/.[2] Tellina provides fairly mixed results, producing correct or nearly correct bash commands for some English phrases, while producing completely incorrect commands for others. The following are examples of inputs to and outputs from Tellina:
In which Tellina produces output matching the intention of the natural language command:

```
“find all pdf files in this directory and its subdirectories →”
find . -name "*.pdf" -print
```

In which Tellina does not produce output matching the intention of the natural language command:

```
“remove the first line from each text file in this directory” →
find . -type f -exec grep California {} \; -exec rm {} \;
```

```
“view file.txt” →
rev file.txt
```

Tellina uses a neural network to learn from a dataset consisting of pairs of matching natural language and bash commands. It is expected that if the dataset were to be significantly increased in size, a significant increase in the number of bash commands correctly paired with a matching English description will occur. An English description is said to correctly match a bash command if the bash command performs the task described in the English description.

The most feasible way to go about improving Tellina is to increase the amount and quality of data available to it. The current dataset used by Tellina was originally collected by hand--workers were hired to write down natural language phrases and corresponding bash commands. However, this manual approach is slow, expensive and potentially error-prone. A data-collection method that collects, cleans and verifies data automatically would be likely be cheaper and more efficient. To our knowledge, no automated data-collection mechanism are in use for this purpose, aside from tools used to parse dumps from sites such as StackOverflow. Additionally, the existing data exists in multiple forms, from multiple sources. Some of the training data has been cleaned and manually verified, while some of it is inaccurate. Thus, another way to improve the accuracy of the commands generated by Tellina would be to develop a system that cleans and verifies the existing data automatically, in so far as that is possible. In short, this project aims to develop an automated data-collection system to increase the size of the current natural language to bash data set used with Tellina, as well as to improve the quality of existing data, with the ultimate goal of improving the accuracy of Tellina.

Approach

Existing System Architecture

The existing Tellina architecture consists of a dataset (the approximately 10,000 bash one-liners collected from StackOverflow and similar), TensorFlow neural-nets for translating English into Bash, and some extra utilities such as a Bash parser (which creates an AST) and a regex-based sentence tokenizer and an entity recognizer for Bash. The TensorFlow library from Google is used to train the data model upon which the other components rely.

For the purposes of this project, we do not intend to modify much, if any of the actual TensorFlow or evaluation code, but will be continually modifying the dataset in order to provide better, more up-to-date training of the model. Hence, the server that hosts the Tellina model will periodically re-run the TensorFlow experiments in order to generate new copies of the model to use, in order to integrate the new data as it is verified. Because the TensorFlow model would take a lot of time to train, especially as more data was added, the machine serving search results to end users will continue to serve from the old model until a new model is provided.

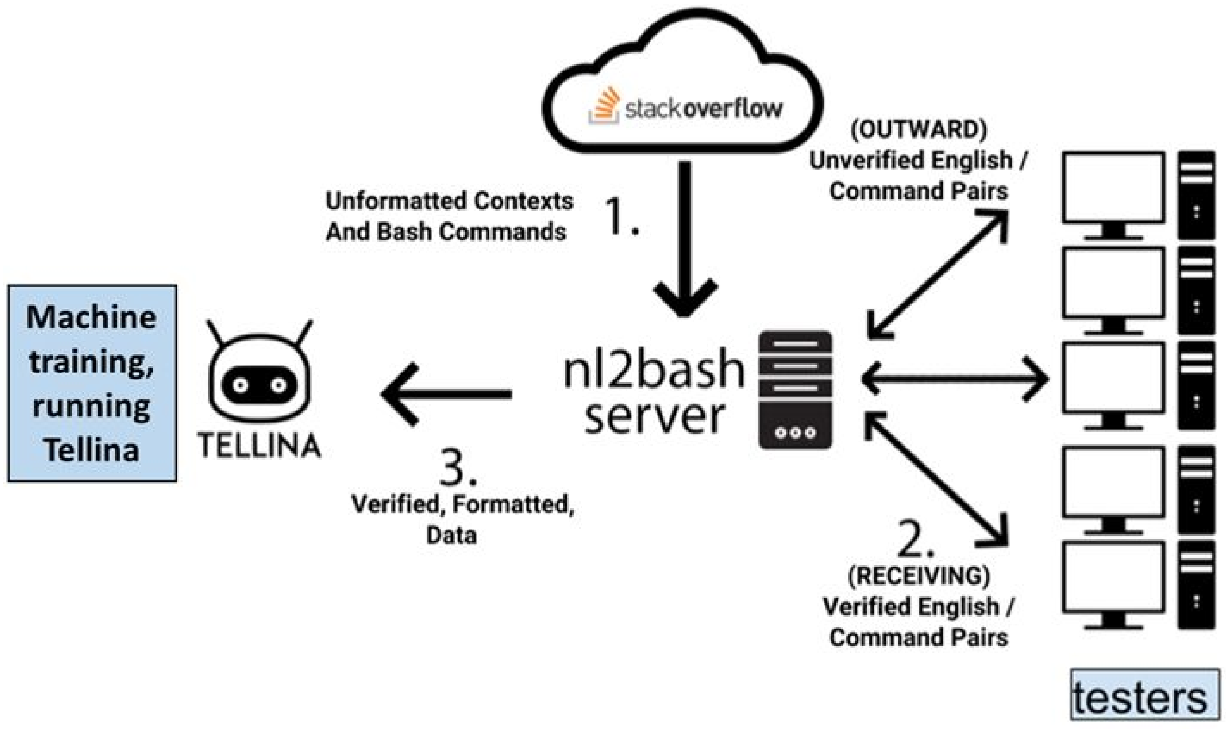
Architecture and Implementation

There are two main facets to our approach in creating a better data set for Tellina: automated data collection and data verification. Although the current Tellina Github repository contains thousands of resources scraped from StackOverflow (https://stackoverflow.com/) and the man pages for individual commands (https://tiswww.case.edu/php/chet/bash/bashref.html), we have implemented a web-scraping tool to search these sites for keywords that are associated with english commands to augment the already existing data set. The other major part of our project is the verification of both existing and the newly scraped commands that correspond to english phrases. Tellina initially verified their data by paying freelancers with experience in Bash to manually verify commands. While this approach is effective, it is not efficient, as it is expensive to hire even a few workers and each worker could only process around 50 commands per hour. As such, our primary approach to data verification comes in the form of crowdsourcing. We will consider these two phases to our project separately, below.

The TellinaTool group of projects obtained data in multiple ways. A large set of linux shell commands and their natural language descriptions were provided by experts[1]. While the majority of commands are high quality, their paper discussed how the training data from this approach was limited both in breadth of arguments and quantity of commands, and so they attempted to automate its collection. The original Tellina group attempted to parse natural language command pairs without manual verification by downloading a large quantity of StackOverflow posts, parsing through its history to find relevant threads, and finding relevant pairs of bash commands and their natural language descriptions. While we initially considered utilizing the StackOverflow archives, found at https://archive.org/details/stackexchange, that are updated every three months, there were several difficulties that lead us to develop our own web scraper. Primarily, the posts and comments are stored in two different files, Posts.xml and Comments.xml. In order to successfully generate our pairs we would have to load the contents of a 3.5GB file into memory to cross reference with our other file. In addition, the HTML contains specific classes for div elements that specify what language the code contained within is. Scraping the HTML allows us to quickly access specific information on a single page while parsing through the archives would end up being more difficult.

Ultimately, we found that the most effective way to expand our training set is to periodically scrape sites like StackOverflow that contain both valid bash commands and contextual clues, using similar techniques to obtain associations from threads. Following the approach taken by the original nl2bash paper [2], we have created an interface used for manual verification to increase the quality of our data. As described in the motivation section, our ideal outcome would ultimately be to increase both the quality and quantity of our training data without any loss in quality over the expert given training data Tellina currently uses, but understand that it will not be possible to ensure this quality without defeating this project's purpose.

Architecture Diagram


stackoverflow
Unformatted Contexts And Bash Commands
1.
(OUTWARD) Unverified English / Command Pairs
Machine training, running Tellina
TELLINA
nl2bash server
3.
Verified, Formatted, Data
2.
(RECEIVING) Verified English / Command Pairs
testers

The three major components of our architecture diagram are scraping new commands, verifying them with users, and sending them to Tellina, respectively labeled (1), (2), and (3) in the above diagram. Although Tellina already has a large corpus of questions (not all of which are sufficiently verified), our primary means of data collection comes from our server scraping different websites where questions and commands can be gathered from.

Labeled (1), the first part of our process begins with the scraping and parsing of this data. This data is intermittently scraped and formatted by our JSoup web scraper and stored in a sqlite3 database to assure that we always have unverified data ready to be supplied to testers. As testers request new english-command pairs to verify, this stored data is then distributed to the client applications as the testers interface with it through a client application. At this point, our central database and server sends formatted, but unverified data to the client applications. Each of these client applications pulls unverified questions individually, so that each tester verifies different questions and all testers can do meaningful work simultaneously. When a tester is done verifying a question, the verified commands and questions are sent back to the main server. This process of the main server sending unverified data and receiving verified data is shown in the diagram as label (2). Finally, once the verification has been agreed upon by a sufficient number of testers, this verified and formatted data is ultimately integrated into the Tellina database. At the moment, we plan to run our own Tellina server so that we have the ability to benchmark how effective our process is at improving the Tellina database. We can directly pull these numbers from the Tellina client, so the transfer of this final data set is shown as label (3).

Regarding these individual components, our central server is where much of the processing for the project lies. The server is tasked with hosting our fork of the Tellina website, storing our database of unverified commands, and interfacing with the client applications. Our current implementation for our web scraper checks for highly voted StackOverflow questions with certain tags, as detailed in the Data Collection section. Our scraper updates the page and parses any questions that have been asked since the last time the tool checked. Focusing primarily on StackOverflow allows us to both obtain data from multiple contexts (tutorials, question threads, documentation, commented shell scripts, etc) and expand our training data beyond the current Tellina database. At this point, the data is trained in our server (and should be retrained nightly to keep updated on new commands and developments), and ultimately sent to our database. The current Tellina website interfaces with a Django server in order to request the answer to user queries, so our trained data is stored in an sqlite3 database so that the existing Tellina Django server can be integrated flawlessly into our system. After verified, formatted data is sent back from testers, it is formatted and placed into this database. Ultimately, this means that our central 'server' scrapes data, supplies it to users, and places the response from the users into a database so that Tellina can interact with it.

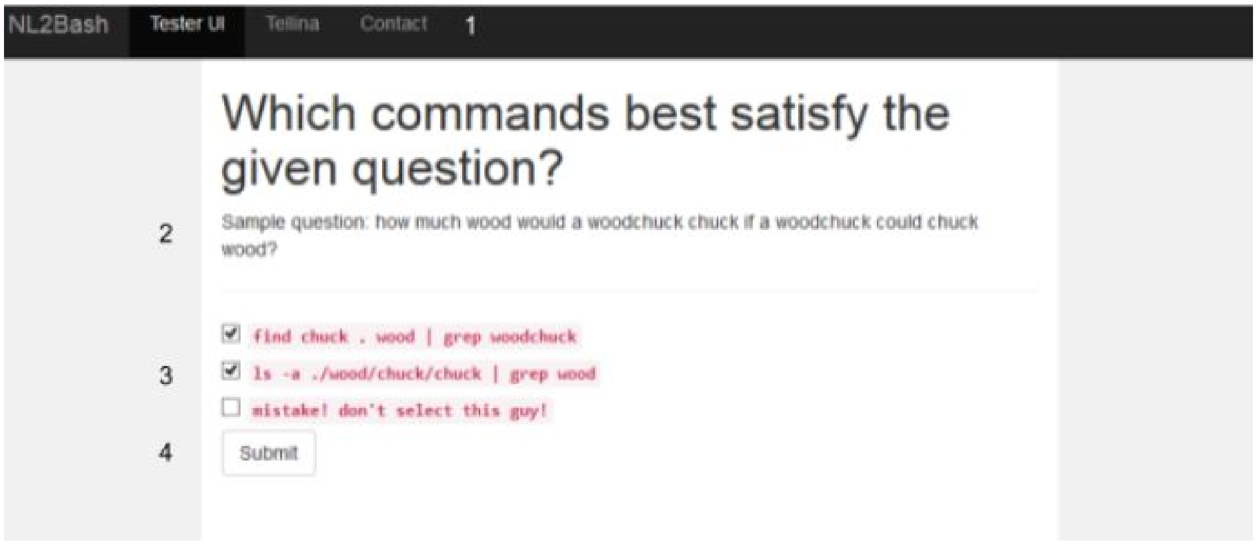
Data Collection

Our primary means of data collection is by scraping the most popular StackOverflow questions tagged with either 'shell' or 'bash' for the question being asked and commands from the top five answers. The page "https://stackoverflow.com/questions/tagged/shell+bash?page=1&sort=votes&pagesize=50" represents a search query for the previously mentioned parameters, displaying 50 links to individual questions per page. For each link present on this page, the scraper opens and scrapes each question page before moving to the next page of search results. Ultimately, the scraper will have exhausted all questions with the correct tags and shut off. For each individual page, the web scraper generates a file containing a JSON representation of the page containing the question asked by the user and an array of strings that hold the commands listed in the top five answers. There are called .verify files. These .verify files are currently output into the NL2BashWebScraper directory, which our verification server pulls from in order to serve to verifiers. This directory will be located on the server running the web scraper, and a more detailed view of that filesystem will be included once the web scraper is running on a server. When a user requests a new verification, the verification server chooses a .verify file and serves it to the verifier. The .verify files are titled with the question id which is cached in the web server to ensure that we can verify any given file a certain number of times rather than repeatedly sending it out to verifiers in lieu of other files that have not been selected.

Sorting by the most highly voted posts in our query means that our earliest scraped questions will tend to have more answers and be more thoroughly vetted by the community. One possible improvement to the scraper would be determining a heuristic where the recency of a question and the number of responses could determine whether or not we want to present that data to verifiers. Another possible improvement would be to cache the id of the questions and only scrape answers collected after a certain period instead of scraping all the questions on StackOverflow again. At the moment, there are only ~2500 questions that fit the above criteria; this means that our scraper can currently parse these questions in just over forty minutes provided that it currently takes around one second per page.

One issue that we are currently running into is that the data scraped from StackOverflow may contain code that is poorly formatted and will not run when entered into a terminal. While we looked at several ways to verify that a command is valid without having to run it, we couldn't find any tools that met our needs. Tellina includes a script filter_data.py that filters commands to retain only "grammatical and frequently used commands" amongst other utilities. All our data is run through these scripts before being used to train our model. In addition to this, we verify that each answer from StackOverflow is valid by spinning up a new virtual environment, re-imaging it to a known state, and actually running the command in the terminal. The `$?` environment variable is set to zero upon the successful execution of the previous command, and non-zero on a failure. By running each command and testing the environment variable appropriately, we can determine whether or not a script is 'valid' in the sense that it runs in a terminal without error. This approach still doesn't account for malicious commands or commands that rely upon the presence of a specific directory or file to successfully execute.

User Interfaces



Which commands best satisfy the given question?

Sample question: how much wood would a woodchuck chuck if a woodchuck could chuck wood?

☑ `find chuck . wood | grep woodchuck`
☑ `ls -a ./wood/chuck/chuck | grep wood`
☐ `mistake! don't select this guy!`

The only part of our project that will be facing the testers is the verification website. Upon execution, the website will download a new set of questions to present to the tester for verification. In the case of a StackOverflow verification, the tester is presented two major displays; one with the question asked by the StackOverflow user and one with several of the answers provided. There are checkboxes next to each of the possible valid Bash commands that the tester may check to indicate that the command sufficiently answers the supplied question. When the tester has selected the answers they feel best satisfy the question (if any), they click the "Submit" button at the bottom of the screen to pull up a new question to be verified. The tester does not have a set number of questions to answer and is free to close the program at any time. We believe that this webpage interface is simple enough to allow a wide variety of testers to verify these commands while still producing meaningful results.

The other major user interface, which appears for all end users attempting to translate English into Bash, is the Tellina web interface itself, which provides a search-engine-like GUI with a search box for inputting natural language and a sorted results list for reading candidate Bash commands. Because our project's scope involves giving the Tellina translation tool better data, rather than changing how the algorithm works, we have no plans to change this interface (http://kirin.cs.washington.edu:8000/) .

Current Technologies

With regards to web scraping, we currently use the JSoup library to scrape through several sites that would have questions related to the bash shell and answers in close proximity, like StackOverflow. This application would be hosted on a server, as detailed in our architecture diagram above, and the data generated from this program would be fed to the aforementioned verification client applications. These client applications would then send back the verified pairs to the server, where they would be formatted to work with the Tellina database.

Many of our choices in both language and frameworks come from the current Tellina project, such as our use of a Django server and sqlite3 database; we choose these technologies not because we feel that they are the optimal technologies for our project, but rather that they allow us to utilize the current code for the Tellina project without massively rewriting existing sections of their code (an undertaking well beyond the scope of this course). Our server would allow the applications to communicate and store the data on a single computer, rather than having to physically transfer data between the applications through a third party distribution tool. As shown in the architecture diagram above, the application would interface with the individual pieces to update clients with new commands to verify, and to keep scraping the web for additional question-command pairs.

Retraining the model was originally intended to be done overnight with a simple Linux PC with an NVidia GPU. However, training Tellina on the fully dataset currently takes more than 16 hours, so we will instead use a compute instance on AWS to retrain the model as new data is added. It might even be beneficial to consider a cluster environment within those services, so that the tasks of verifying, recalculating, and end user service may be distributed across multiple AWS machines with different specs and thus different costs.

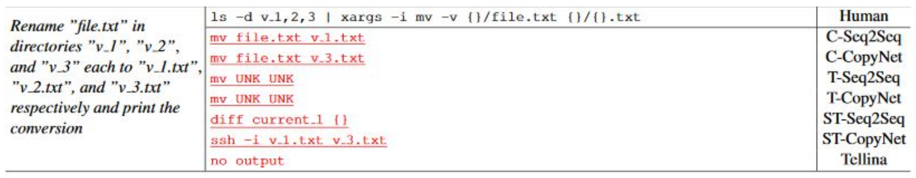
Evaluation

Regarding specific experiments that we can use to determine the efficacy of our project, the findings presented within the original paper on Tellina provide appropriate tables and graphs that we hope to emulate and ultimately compare against in our conclusion.

With regards to the program itself, we are able to directly copy the format presented within the papers and measure against the same definitions of successful commands. Tables, such as those shown as table 1 and table 2 below, detail statistics we are interested in obtaining from the program: translation accuracy to see if the scraped pairs are too difficult to use, where errors are occurring/if our heuristics are helping, and others. The Github repository that the Tellina project resides in contains a Makefile target for these evaluation tables against the current trained model, which means that, after training our new model from our improved data, we can generate tables for it and compare its metrics to those presented in the paper.

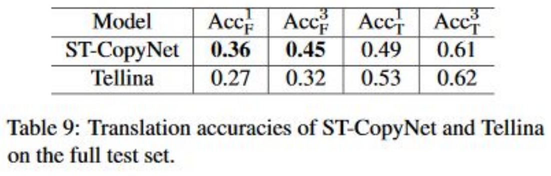
However, our best metric of success is if we can generate, on average, more successful bash commands than the current set of training data for Tellina; as such, we will have to create experiments with expert bash users to compare output from the original Tellina model and our improved model on various inputs (descriptions not found in either, descriptions found in the improved model and descriptions found in both). From this experiment, we would count the efficacy of each model; a statistically significant improvement would indicate our technique worked.

We also might conduct a small experiment/survey to find the best interface for a tester; however, this is not a focus for our project and if we do attempt this, it will be in addition to what is already detailed.



Rename "file.txt" in directories "v_1", "v_2", and "v_3" each to "v_1.txt", "v_2.txt", and "v_3.txt" respectively and print the conversion	`ls -d v_1,2,3 | xargs -i mv -v {}/file.txt {}/{}.txt`	Human
	`mv file.txt v_1.txt`	C-Seq2Seq
	`mv file.txt v_3.txt`	C-CopyNet
	`mv UNK UNK`	T-Seq2Seq
	`mv UNK UNK`	T-CopyNet
	`diff current_1 {}`	ST-Seq2Seq
	`ssh -i v_1.txt v_3.txt`	ST-CopyNet
	`no output`	Tellina

Table 1: A table from [1]. Example predictions of baseline approaches. The English description is shown on the far left, and the corresponding bash command is shown at the top of the middle column. Incorrect predictions generated by Tellina and other comparison systems are shown in red.



Model	Acc^1_F	Acc^3_F	Acc^1_T	Acc^3_T
ST-CopyNet	**0.36**	**0.45**	0.49	0.61
Tellina	0.27	0.32	0.53	0.62

Table 9: Translation accuracies of ST-CopyNet and Tellina on the full test set.

Table 2: A table from [1]. These are translation accuracies of Tellina, compared to those of the ST-CopyNet system.

Initial Results

On May 10th, 2018, we collected initial results from our data scraping and retraining, which are presented against Tellina's initial results below. To reproduce these results, follow the READMEs present in the repositories for each individual module and run the provided script from our repository. Note that we can train and evaluate our model from scratch or from the output of the TesterUI, the all.nl and all.cm files; instructions for both are in the main repository README. From these results, we can see that the Tellina model performs above or at the same level as other similar models in multiple metrics. These initial results suggest that with additional training data, our approach for data verification will further improve these metrics.

The original automatic testing accuracy of the Tellina model is shown below in Table 3.

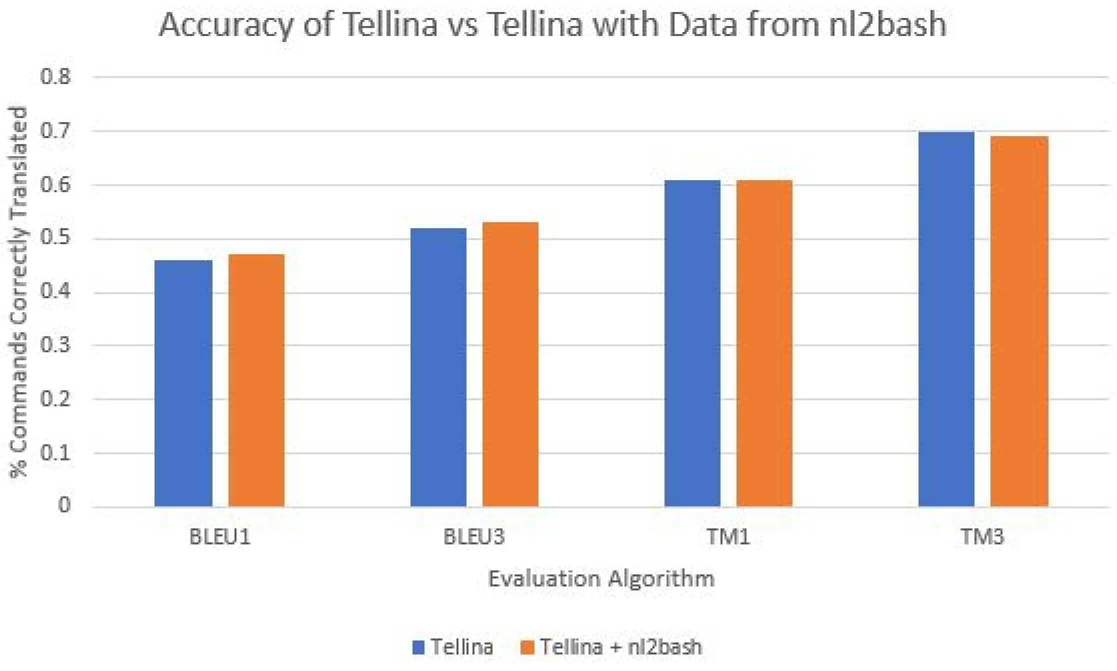
Model		Testing Accuracy w/ Given Metric on Original Dataset [1]			
		BLEU1	BLEU3	TM1	TM3
Seq2Seq	Char	49.1	56.7	0.57	0.64
	Token	36.1	43.9	0.65	0.75
	Sub-Token	46	52	0.65	0.71
CopyNet	Char	49.1	56.8	0.54	0.61
	Token	44.9	54.2	0.65	0.74
	Sub-Token	55.3	61.8	0.64	0.71
Tellina		46	52	0.51	0.70

Table 3: The table of initial results from Table 15 in [1]. Each row refers to a different combination of natural language model and input type (either Seq2Seq or CopyNet, taking in either individual characters, Bash tokens, or partial Bash tokens). Each column refers to a different translation evaluation metric - BLEU stands for Bilingual Evaluation Understudy, while TM is defined in [1]. Tellina is a normalized version of Seq2Seq taking full tokens as input.

Our process started by scraping hundreds of posts, which took a few minutes. We personally verified these pairs using our TesterUI, resulting in 46 verified pairs, which we subsequently added to our the existing all.cm and all.nl to be used by the Tellina learning module. We did not modify the original 12,000 pairs that were added to. We were able to replicate the original training process on the Amazon AWS machines, although we had to reduce the number of training epochs to approximately 2 or 3 to do so (the original model used 100 epochs). Table 4 below shows the initial training accuracies we got from running the training scripts on these machines, with the small amount of additional data from nl2bash.

Model		Testing Accuracy w/ Given Metric on Augmented Dataset			
		BLEU1	BLEU3	TM1	TM3
Seq2Seq	Char	0.35	0.41	0.39	0.44
	Token	0.37	0.45	0.63	0.72
	Sub-Token	0.47	0.53	0.61	0.68
CopyNet	Char	0.31	0.37	0.33	0.38
	Token	0.44	0.53	0.63	0.71
	Sub-Token	0.45	0.51	0.51	0.57
Tellina		0.47	0.53	0.61	0.69

Table 4: A table with initial results. The format is the same as in Table 3, with the numbers being the accuracies of the models trained on the AWS machines with additional data from the nl2bash system.



Graph 1: This plot is a visual comparison between the command prediction accuracy of the Tellina model trained in the original dataset (blue) versus trained on a dataset with both the original data and data from the nl2bash system. These values correspond to the bottom rows of the two table above.

We are able to make the comparison between the original paper's automatic evaluation results for the original model, and our own evaluation results with the augmented model because, as demonstrated in

[3], no changes had been made to the model or to the dataset since the paper was published. As seen in the table and graph above, the Tellina model performed slightly better in two of the categories after being trained on a dataset with both the original data and new data from the web scraper/tester UI system as compared to the results from the original dataset. While this is mildly promising, a significant improvement was not observed nor expected due to the small size of our initial data augmentation, and we would need to do further work, such as reducing the noise found in the data, to improve our results. Ultimately, these initial results serve more to show that we were able to reproduce the original automatic results published in [1].

Next Steps

While augmenting the pairs from the Tellina repository provided moderate results, cleaning and verifying the existing data is another approach to improving Tellina's accuracy. To do this, there are two approaches: manual verification and automatic cleaning. Manual verification could be done, using our TesterUI, by creating .verify files from the original all.cm and all.nl files. We have an experimental script that does this in our root repository, and our current instance of the website has loaded these files, allowing us to verify a working subset of the original commands. Automatic verification of the original pairs could be done through various means, such as checking bash syntax or running the commands in virtual machine instances and checking the return codes.

While our project focused on verifying and augmenting the existing Tellina repository, an interesting continuation of the project could look into utilizing new machine learning architectures that are more resilient to noise. It is intrinsically hard to separate signals from noise when dealing with natural language; as much as we can manually format incoming commands and natural language descriptions to a specific template, it's much harder to automate given our current implementation. Changing the architecture that the original NL2Bash system is built upon to a more modern architecture more suitable for NLP (a system that implements word embeddings may be an interesting starting point) could be a good next step.

References

1. Lin, X. V., Wang, C., Zettlemoyer, L., & Ernst, M. D. (2018). NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System. arXiv preprint arXiv:1802.08979.
2. Lin, X. V., Wang, C., Pang, D., Vu, K., & Ernst, M. D. (2017). *Program synthesis from natural language using recurrent neural networks* (Vol. 2). Technical Report UW-CSE-17-03-01, University of Washington Department of Computer Science and Engineering, Seattle, WA, USA.
3. Github.com. TellinaTool/nl2bash commit summary for commit 925b74beec. Retrieved May 30, 2018. URL: https://github.com/TellinaTool/nl2bash/commits/925b74beec7af0e0c0da39909c4e09b41b326a48