Oisin Doherty (oisind),
Abhinav Gottumukkala (anak4569),
Hans Jorgensen (thehans),
Lauren Martini (lmartini)
Due 4/12/18
CSE 403
Project Architecture

NL2Bash

Expanding upon existing data sets to improve translation quality

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, “Copy the first line from each text file in this directory into new_file.txt” 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:

```
head -n1 -q *.txt > new_file.txt
```

For users unfamiliar with Bash, this is a difficult command to understand. The user would have to understand thoroughly how to use the `head` command, and how to properly use the `-n1` and `-q` flags to only extract the first line from each file. They would also have to know how to redirect the output from `head` to `new_file.txt` with the `>` 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. Thus, it is expected that if the dataset were to be significantly increased in size, a significant increase in accuracy would be observed. “Accuracy” here refers to the ability of Tellina to produce a bash command that performs the task described in the natural language command. The most feasible way to go about improving Tellina is to increase the amount 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.

Current Direction

Considering the above, 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 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. Looking at the NL2Bash repository (https://github.com/TellinaTool/nl2bash), the data folder details how a large set of linux shell commands and their natural language descriptions were provided by experts. While virtually guaranteed to be high quality, their paper discussed how the training data from this approach was limited both in sheer size and request spread, and so they attempted to automate its collection. In the Tellina repository (https://github.com/TellinaTool/tellina), they explain their attempt to automate training data collection without a verification procedure; it boiled down to downloading the raw StackOverflow post dump,

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 11: Example predictions of the baseline approaches. The prediction errors are underlined.

Model	Acc_F^1	Acc_F^3	Acc_T^1	Acc_T^3
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.

parsing through its history to find relevant threads, and finding relevant pairs of bash commands and their natural language descriptions. To build on this, we have found that the most effective way to expand our training set, regardless of initial quality, is to scrape sites like StackOverflow that contain both valid bash commands and contextual clues, using similar techniques to obtain associations from threads. To complement the data we ultimately plan to gather, we need an automatic method to ensure that the gathered associations are correctly paired. As described in the motivation section, our ideal situation would be to increase our training data without any loss in quality over the expert given training data Tellina currently uses. We decided to go with our second idea, which was to try to crowdsource verifications in some manner; we considered something akin to the code verification games proposed by other groups because we couldn't see any other form of crowdsourcing drawing the number of users required to handle such a stream of automatically gathered data, but our initial design will be a flat interface where they can verify associations. Regarding specific experiments that we can use to determine the efficacy of our project, the findings presented within our first reference 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 on the side from the first reference, 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. 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.

Challenges

The primary challenges that we expect to face during our implementation of such a framework lie mainly in our inexperience with both scraping online data and the risks involved with relying on crowdsourcing for a critical component of our project.

Only one of the current members of the team has had any prior experience with scraping data from the internet. One of the most important parts to improving the data set of Tellina is the ability to gather more data with which to test with. Regardless of the quality of the data, simply having more data to train Tellina's model on will ultimately result in a higher success rate for the generated bash commands. Therefore, the success of this project relies heavily on our ability to create a client that can interface and scrape multiple different sites into a single format that we can use. Our current implementation utilizes the JSoup library (https://jsoup.org/) as our primary means of downloading and parsing the HTML of websites. As of yet, we haven't run into any issues with this approach. However, here are many issues that we could run into considering the volume of the data that we're attempting to gather. Sites may implement rate limiting, captchas, and browser fingerprinting specifically to reduce the strain on the site that scraping would cause. Although we initially expected our lack of experience with the JSoup library and scraping websites to take up a significant portion of our time, we were able to rapidly prototype this aspect of the project without issue.

Regarding data verification, the primary issue that we deal with is the inability for us to completely automate the process of command verification. Natural language processing today still has little to no ability to perfectly understand and abstract the intentions behind English phrases, which leaves us no choice but to have English-speaking humans who have prior experience Bash to verify that the commands and associated English phrases are correct. The prior developers of NL2Bash[1] state that they "hired 10 Upwork freelancers who are familiar with shell scripting [who] collected text–command pairs from web pages such as question-answering forums, tutorials, tech blogs, and course materials." Fundamentally, while we automatically supply the data from scraped web pages, we rely on the same process of having people familiar with Bash verify commands. This abstraction and simplification makes it easier to verify the integrity and quality of the commands, but it also means that we not only have to encourage and maintain participation in our verification tools (and thus design and prototype them well), but ultimately rely on their collective judgment to determine the quality of each translation. Taking the time to find people skilled enough to complete the task is fundamental to ensuring the quality of our finished product, though this effort will cost development time that could otherwise be used to make the product more efficient.

Regardless of the challenges associated with such implementations, the payoff of having a tool that has a high success rate of translating an English sentence into a Bash command would save a substantial amount of time for developers of all skill levels. In terms of this class, most of the issues that we currently face are quick to prototype, but would require significantly more development time to mature into a project that could see legitimate public use and be of any substantial help to the Tellina project. Implementing a fully-fledged data generator is far beyond the scope of this course, but we have

the potential to lay the groundwork for further iterations of this project that would develop it into a production-viable solution. We firmly believe that the completion of this project will result in a more streamlined development process for everyone who interacts with the bash shell.

Monetarily, the implementation itself does not incur any costs, since we are only using resources we already have, such as personal and CSE lab machines to host and train data. As the project grows, we may consider switching to paid services that offer more efficient web scraping on more complex sites, or more efficient, cloud-based hardware for retraining Tellina. This would significantly reduce the amount of work necessary to implement both aspects of our project, allowing us to put more time into the process of taking scraped data and formatting it according to the Tellina guidelines. For crowdsourcing, we may run into the issue of not having enough volunteers to actually test data; this would require us to pay others, most likely UW students, to encourage enough participation to verify an adequate amount of data for benchmarking. Because we don't yet have an estimate of how much data we will scrape and parse - this will ultimately be bottlenecked by how much data we can verify in a timeframe, since data itself is easy to scrape - we consequently do not have an estimate for how long it would take to verify that amount of data and thus how many verifiers we would need. Ultimately, the physical process of verification is likely to be the largest time sink in the project, as we've previously mentioned that it should be able to very quickly develop prototypes at all stages of the project, and upgrade accordingly to budget and time constraints within the scope of this class.

Project Schedule

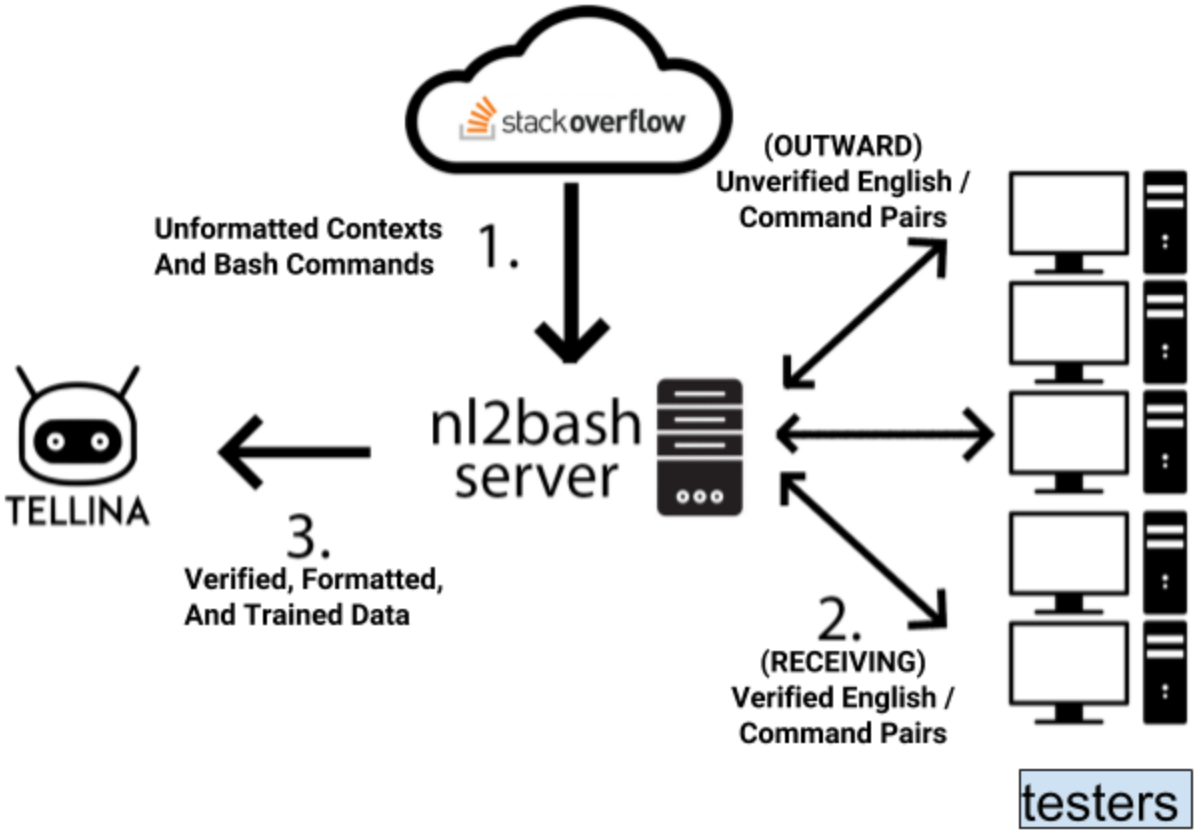
- By week three, we had planned to have a proof of concept web scraper and an initial design for the verification/testing interface. These goals have been accomplished, with the latter presented in the user manual accompanying this document.
- By week four, we plan to have a scraper that works for specialized sites like StackOverflow, as well as working prototypes for the data collection and verification parts of the system. We also intend to determine how to train and use Tellina on our own machines by the end of this week.
- By week five, we plan to have a server set up to test and run our verification system with Tellina's existing dataset, and we plan to have collected a small dataset using our own web scraper to verify and use. As soon as the server is set up, the testing phase can begin.
- In weeks 6-8, we plan to gather data for Tellina's dataset to measure our efficacy, and iteratively upgrade our systems with our findings.
- In weeks 9-10, we plan to determine points of improvement for our system based on feedback from testers, and to implement those improvements.
- Finally, on week 11, we will report and present our findings.

Because the two major facets of our project - retraining Tellina to work on new data, and gathering and verifying additional data - are largely disjoint, our preliminary development schedule has two pairs working on both parts of the project simultaneously. Because of our ability to rapidly prototype our systems, a good "midterm" would be to have a fully functioning prototype of both parts of the project.

The latter half of our schedule pertains to gathering and using additional training data in a more unified manner, as well as gathering data about the efficacy of our methods. Because of this, a good "final" project phase would be to upgrade the systems we've implemented and formally test to see how much our additional/improved dataset contributes to the success of Tellina. Regardless of how much 'good' data our project produces, if we can provide a solution to automatically gathering and verifying valid english phrases to bash commands then we have satisfied our goal of developing the groundwork necessary for further improvement beyond this class.

Architecture and Implementation

Architecture Diagram

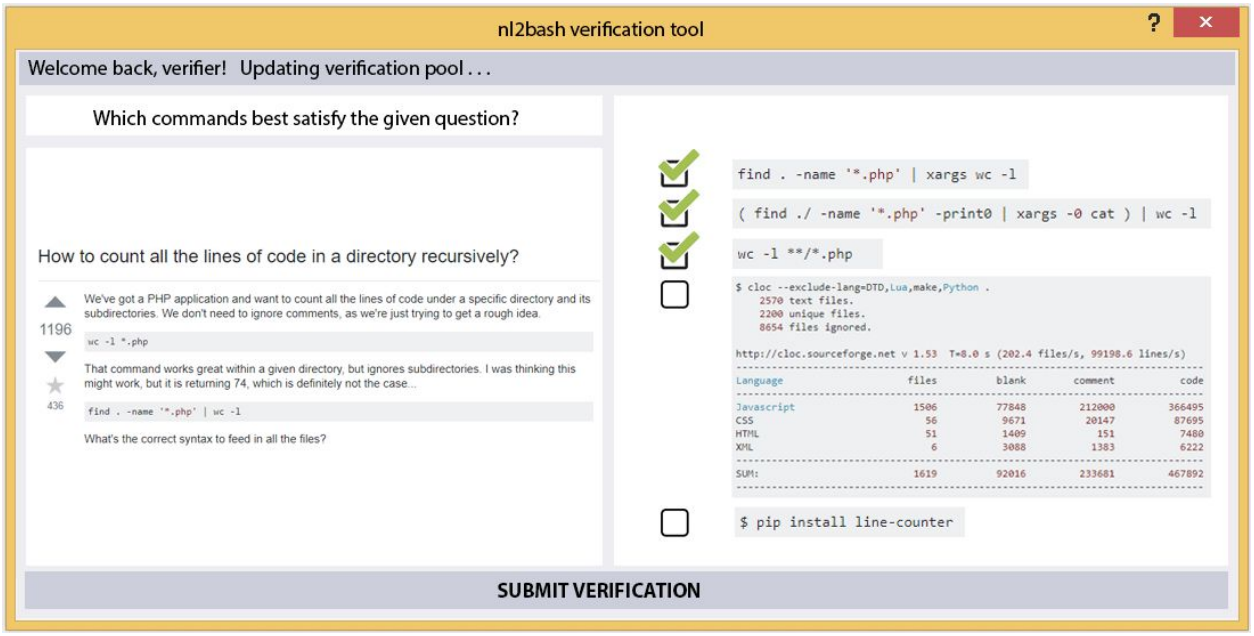


The three major components of our architecture diagram are scraping new commands, verifying them with users, and sending them to Tellina, respectively labeled (1), (2), and (3) in the above diagram. Although Tellina already has a large corpus of questions (not all of which are sufficiently verified), 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 webscraper checks for newly asked StackOverflow questions that mention the keywords “bash” and “shell” (https://stackoverflow.com/search?tab=newest&q=bash%20shell). Our scraper updates the page and parses any questions that have been asked since the last time the tool checked. This tool refreshes this page and check for new commands every thirty minutes to avoid running into issues such as Captcha that are associated with scraping large amounts of data from the web at once. 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.

User Interfaces



The only part of our project that will be facing the testers is the verification software. Upon execution, the software 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 VERIFICATION” 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 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.

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.

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 can be sufficiently done overnight with a simple Linux PC with an NVidia GPU. However, if the development data set gets too large for a PC's capabilities, or if the use of a personal computer is found to be inviable, 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.

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.