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

Background and Motivation

Bash can be a difficult language to learn. Many a programmer using a linux terminal for the first time has experienced the confusion that comes with the flood of commands they need to look up in order to do even the most basic tasks. 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. In this case, one set of commands that would work is:

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

It is reasonable to think that if the user is new at bash, then this would be a difficult command to understand. The user would have to understand thoroughly how to use the `head` command, and how to configure it with `-n1` and `-q` 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.

As such, it is understandable that 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.

An example of Tellina producing output that matches the intention of the natural language command:

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

Two examples of Tellina producing output that does not match 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.

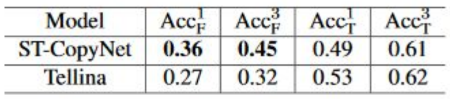
Current Direction

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. We plan on implementing different web-scraping tools to search sites such as stackoverflow (https://stackoverflow.com/) and the official bash documentation (https://tiswww.case.edu/php/chet/bash/bashref.html), using keywords that are associated with english commands to enlarge the program's training data over time. As for verifying that our associations between keywords and commands are correct and give properly formatted bash commands, both attempting to crowdsource the verification and using the already established method of paying skilled people to verify were considered; both had their pros and cons, but we ultimately concluded that attempting to crowdsource the verification would have the most long-term value. 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, 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 imagine that a more 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. Actively scraping, as opposed to scanning through massive thread dumps, would allow us to narrow our scope within a given site to relevant topics (via keywords and search phrases that are relevant to bash commands), allow us to obtain data from multiple types of files (tutorials, question threads, documentation, commented shell scripts, etc) and most importantly, expand our training data beyond any single thread dump or set of bash/natural language description associations from experts.

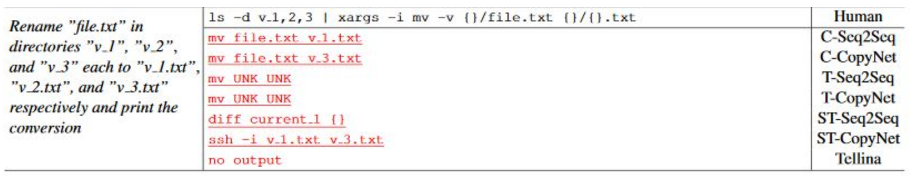
To complement the data we plan to accrue, we would need some automatic method to ensure that the gathered associations are correctly paired. As described in the motivation section, our ideal situation would be to increase our training data without any loss in quality over the expert given training data 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. As detailed in the architecture section, we plan on using jSoup, nodeJS and other tools to automate the data scraping and verification processes.

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 plan 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



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.



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.

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. Through some precursory searching, Java libraries such as jsoup (https://jsoup.org/) would be very helpful in parsing HTML scraped using built-in Java webclients. There 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, implementing captchas and browser fingerprinting specifically to reduce the strain on the site that scraping would cause. When downloading data off of the sites, we may find it necessary for our scraper to download much more data than necessary using our initial approach of downloading and subsequently parsing entire pages of html. Although learning these new frameworks like jsoup and implementing higher level concepts like web scraping may prove to be major points of difficulty in the project, we also expect them to be parts that can be quickly prototyped and upgraded over the development cycle of the project.

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 is not close to having the functionality to perfectly convey and abstract the intentions behind english phrases, which leaves us no choice but to have humans who have prior experience with the bash shell to verify that the commands and associated english phrases are correct. The paper on prior work for the development of NL2Bash[1] states that they "hired 10 Upwork freelancers who are familiar with shell scripting. They collected text–command pairs from web pages such as question-answering forums, tutorials, tech blogs, and course materials." Fundamentally, we rely on the same process of having people familiar with Bash verify commands, but we supply the data from the webpage. This abstraction and simplification makes it easier to verify the integrity and quality of the commands. Despite this, we not only have to encourage and maintain participation in our tools to verify commands, but rely on the quality of such verifications as a critical component in the workflow of our project. In order for us to effectively benchmark the quality of our additions to the Tellina data set, we would need to successfully prototype and actually have groups of people verify these commands. Taking the time to find people skilled enough to complete the task is fundamental to ensuring the quality of our finished product, but ultimately takes away from 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 would result in a more streamlined development process for everyone who interacts with the bash shell.

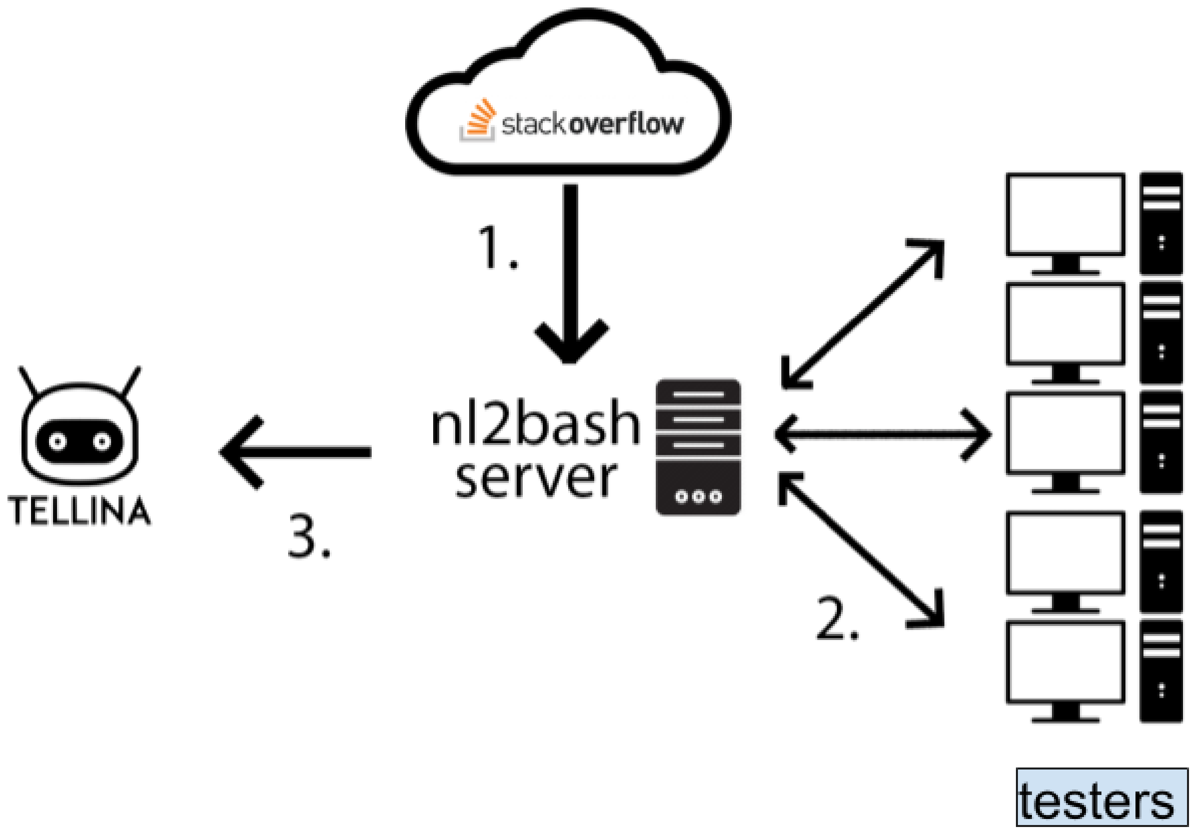
Monetarily, there are no costs in our current planned implementation for our system. As the project grows, we may consider switching to different paid services that offer more efficient web scraping that can handle more complex sites. This would significantly reduce the amount of work necessary to implement the data gathering aspect 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. At the moment, we don't have an estimate for how much data we plan to scrape and parse, and consequently, we don't have an estimate for how long it would take to verify that amount of data. Ultimately, the amount of valid data we can gather is bottlenecked by how much scraped data we can verify in a timeframe--luckily for us, this scales with the number of people we can get to verify data. The physical process of verification seems to be the largest time sink present 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 should have a proof of concept web scraper and an initial design for the verification/testing interface. By week four, we should have 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 week 4. By week five, we should have a server set up to test and run our verification system with Tellina's existing dataset, and we should 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 should focus on actually gathering data to measure our efficacy and upgrading our systems with our findings. In weeks 9-10, we will focus on upgrading and determining points of improvement for our system based on feedback from testers. On week 11, we should focus on documenting and reporting our findings. Because the two major facets of our project are so disjoint, our preliminary development schedule has two pairs working on each part of the project simultaneously. Because of our ability to rapidly prototype our systems, a good "midterm" would be to have a fully functioning prototype, and then focus on testing for the latter half of the project. We feel that despite the challenges that we've accounted for, this is an achievable goal. 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 one, two, and three 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.

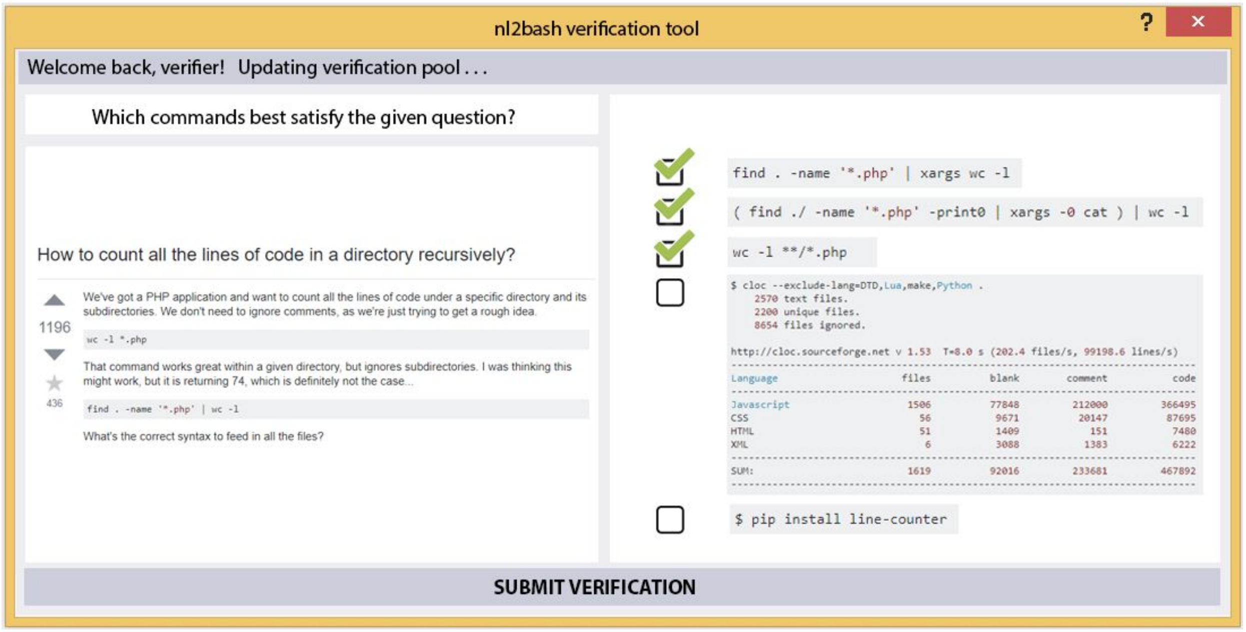
Labeled (1), the first part of our process begins with the scraping and parsing of this data. This data is then distributed to the testers as they interface with it through a client application. Each of these client applications pulls unverified questions individually, so that each tester verifies different questions and all testers can do meaningful work simultaneously.

When a tester is done verifying a question, the verification is sent back to the main server where it can be formatted. 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).

Interfaces



Welcome back, verifier! Updating verification pool . . .

Which commands best satisfy the given question?

How to count all the lines of code in a directory recursively?

1196

436

We've got a PHP application and want to count all the lines of code under a specific directory and its subdirectories. We don't need to ignore comments, as we're just trying to get a rough idea.

```
wc -l *.php
```

That command works great within a given directory, but ignores subdirectories. I was thinking this might work, but it is returning 74, which is definitely not the case...

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

What's the correct syntax to feed in all the files?

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

☑
```
( find ./ -name '*.php' -print0 | xargs -0 cat ) | wc -l
```

☑
```
wc -l **/*.php
```

☐
```
$ cloc --exclude-lang=DTD,Lua,make,Python .
    2570 text files.
    2200 unique files.
    8654 files ignored.

http://cloc.sourceforge.net v 1.53  T=8.0 s (202.4 files/s, 99198.6 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Javascript                    1506          77848         212000         366495
CSS                             56           9671          20147          87695
HTML                            51           1409            151           7480
XML                              6           3088           1383           6222
-------------------------------------------------------------------------------
SUM:                          1619          92016         233681         467892
-------------------------------------------------------------------------------
```

☐
```
$ pip install line-counter
```

The only part of our project that will be facing the testers is the verification software. Upon opening the software, it will download a new set of questions to ask the tester to verify. 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 answers from the comment section. There are checkboxes next to each of the possible valid 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, 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.

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. As for the machine itself, a simple machine with a GPU is likely sufficient for early development, but as the development data set gets larger, a larger machine, such as one hosted through Azure or AWS, may be considered. 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 machines with different specs.

Planned Technologies

With regards to web scraping, we currently plan to 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 single computer, 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.

As we get closer to implementing our individual components, a Node.js 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.

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.