Oisin Doherty, Abhinav Gottumukkala and Lauren Martini

Due 4/5/18

CSE 403

Project Proposal

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. 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, 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 continually enlargen the program's training data. 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 continuously scrape sites like StackOverflow that contain both valid bash commands and contextual clues, using similar techniques to obtain associations from threads. Continually scraping, as opposed to scanning through massive thread dumps, would narrow our scope within any particular site to relevant topics, 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.

In addition to continuously obtaining data, 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. The first type of data verification that we considered was to just continue with the method that Tellina had already used: expand the paying of experts to a system where they are given assignments (sets of verifications) and paid upon completion of them. Our second idea 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. The first method, if implemented with some required accountability for payments, would guarantee quality in the incoming training data and increase the chance that verifications are completed; the downside is that it would be costly to pay so many people. The second method would aim to achieve the same, but with less cost; however, it would have fewer guarantees of correctness.

When building our final solution, we would have to experiment with each option described in this section (utility of scraping various types of pages, contracted vs crowdsourced verifications). 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. We plan to directly copy the format presented within the papers and measure against the same definitions of successful commands. Refer to the reference for punitive tables and graphs for our final results. We know our project will have been successful if we can generate, on average, more successful bash commands than the current set of training data for Tellina.

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.

By week three we should have a proof of concept web scraper and an initial design for the testing interface. By week four we should have have a scraper that works for specialized sites like stackoverflow. By week five we should have working prototypes for both parts of our system. In weeks 6-8 we should focus on upgrading and determining points of improvement for our system. In weeks 9-10 we should focus on actually gathering data to measure our efficacy and upgrading our systems with our findings. 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 honestly be having a fully functioning prototype. We feel that despite the challenges that we've accounted for, this is an achievable goal. Because of this, a good "final" would be upgrading the systems we've implemented and formally testing to see how much our additional testing 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.

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.