Oisin Doherty (oisind),
Abhinav Gottumukkala (anak4569),
Hans Jorgensen (thehans),
Lauren Martini (lmartini)
Due 5/18/2018
CSE 403
Project 9: Draft 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

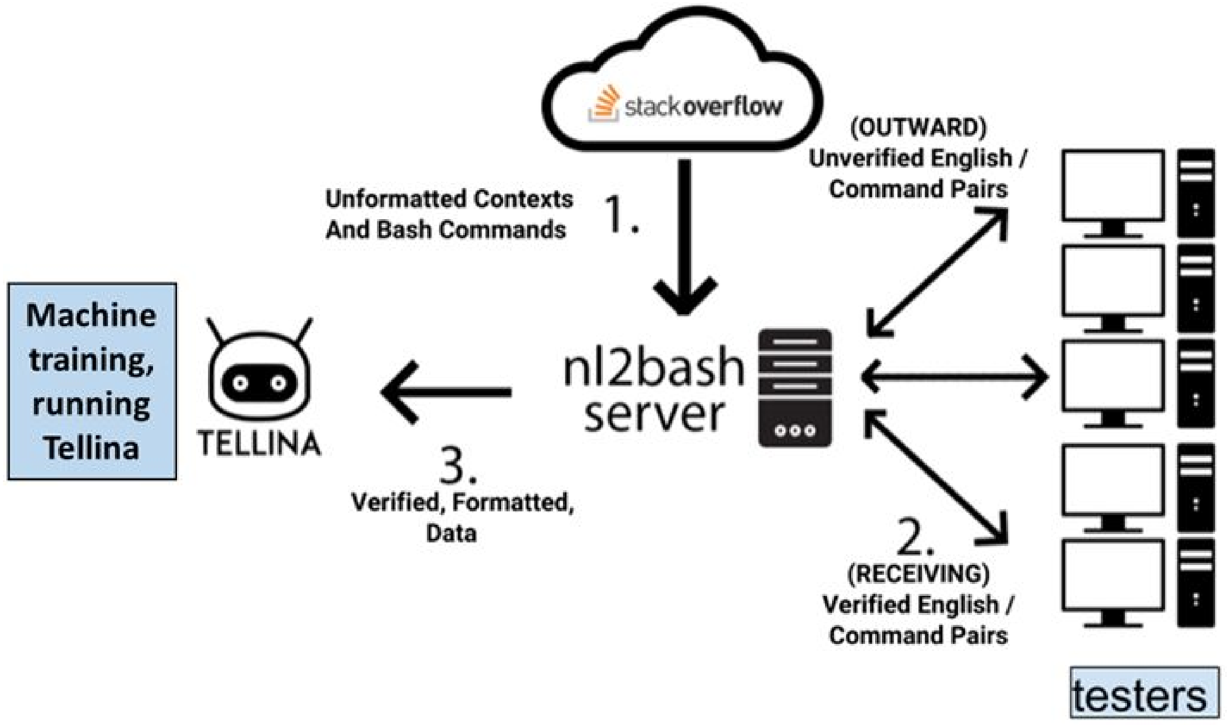
There are two main facets to our approach in creating a better data set for Tellina: automated data collection and data verification. Although the current Tellina Github repository contains thousands of resources scraped from StackOverflow (https://stackoverflow.com/) and the man pages for individual 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 guaranteed to be 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.

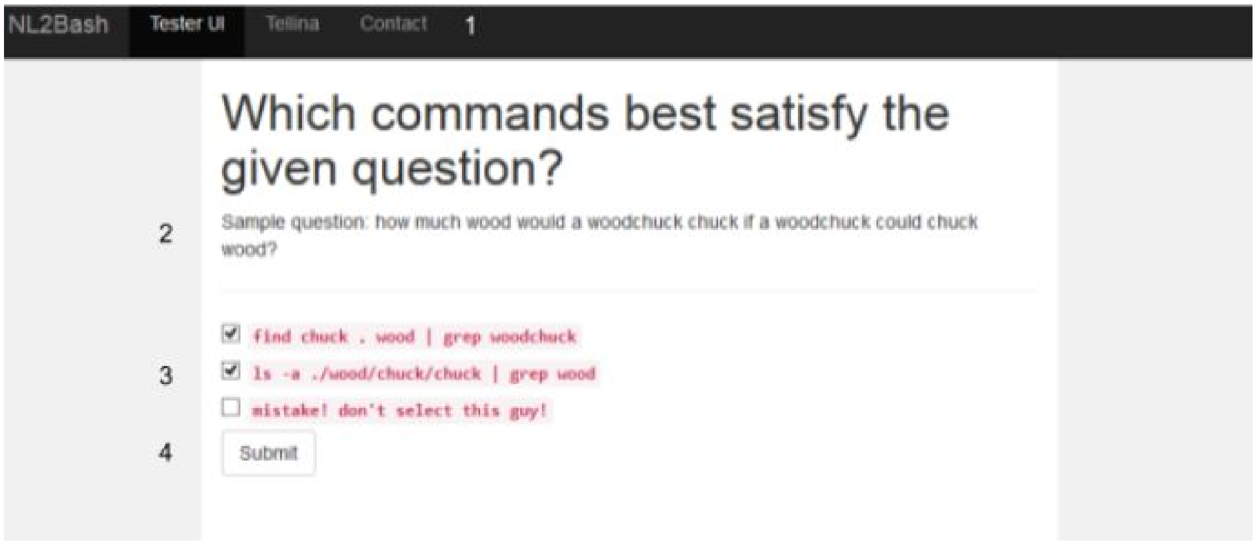
Data Collection

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.

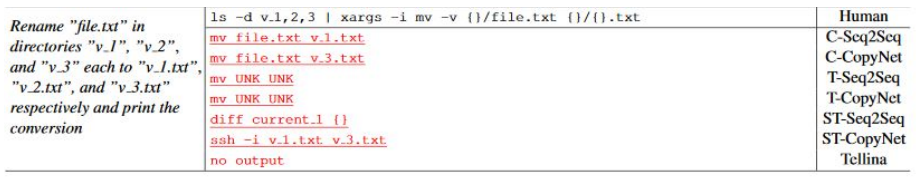
Evaluation

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.

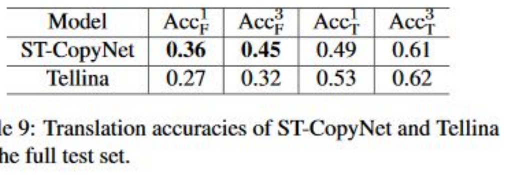
However, our best metric of success is if we can generate, on average, more successful bash commands than the current set of training data for Tellina; as such, we will have to create experiments with expert bash users to compare output from the original Tellina model and our improved model on various 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 testing accuracy of the Tellina model is shown below in Table 3.

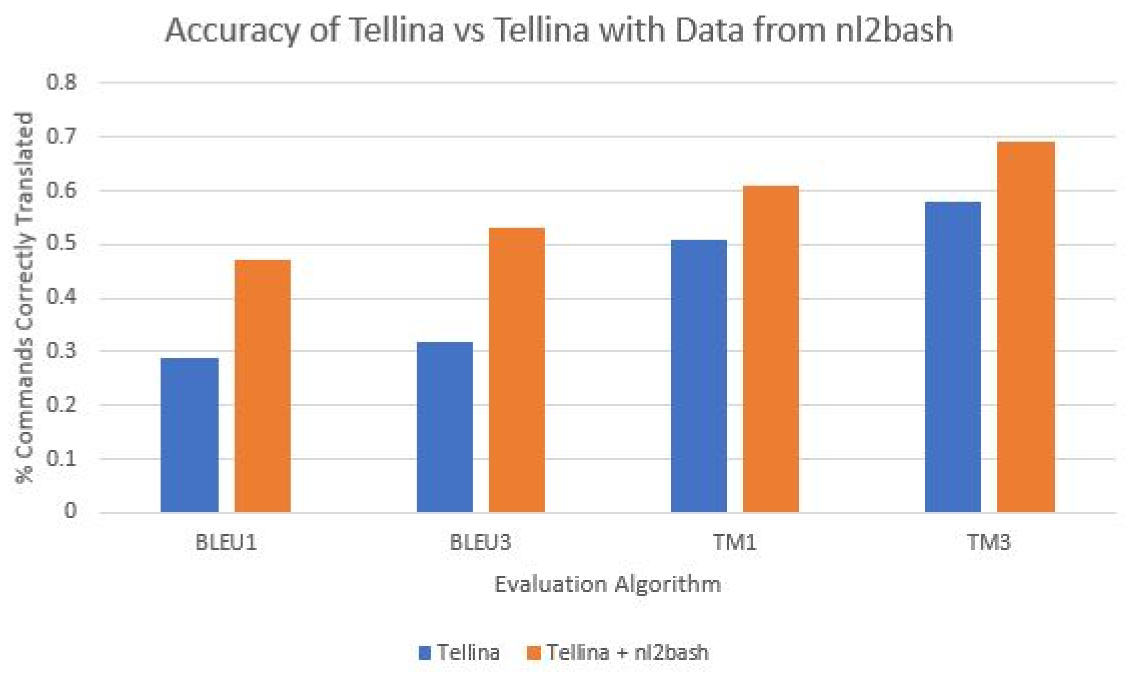
Model		Testing Accuracy w/ Given Metric on Original Dataset [1]			
		BLEU1	BLEU3	TM1	TM3
Seq2Seq	Char	0.24	0.27	0.35	0.38
	Token	0.10	0.12	0.53	0.59
	Sub-Token	0.19	0.27	0.41	0.53
CopyNet	Char	0.25	0.31	0.34	0.41
	Token	0.21	0.34	0.47	0.61
	Sub-Token	0.31	0.40	0.44	0.53
Tellina		0.29	0.32	0.51	0.58

Table 3: The table of initial results from [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.

Comparing the testing setup after the models have been generated is impractical as the original paper did not specify this with the assumption that tables generated with the same model are identical, so we compared the results of the the original model and our updated model. As seen in the table and graph above, the Tellina model performed consistently better after being trained on a dataset with both the original data and new data from the web scraper/tester UI system than on the original dataset. This is extremely promising, since it shows that even with a small number of additional commands, a significant improvement in translation accuracy can be observed.

Discussion and Lessons Learned

The primary challenges that we expected to face lied mainly in our inexperience with scraping online data, relying on crowdsourcing for a critical component of our project, and the costs incurred by training our Tellina learning model on Amazon AWS servers.

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. However, only one of the current members of the team had any prior experience with scraping data from the Internet, so we started the project anticipating several possible issues that we could encounter, such as sites implementing rate limiting, captchas, and browser fingerprinting specifically to reduce the strain on the site that scraping would cause. However, with our current implementation that utilizes the JSoup library as our primary means of downloading and parsing the HTML of websites, we were able to rapidly prototype this aspect of the project without issue, even though we feared that it would take a significant portion of our time.

We knew that we could not completely automate the process of command verification; natural language processing today still has little to no ability to perfectly understand the intentions behind English phrases, which left us no choice but to have English-speaking humans who have prior experience with Bash to verify that the commands and associated English phrases are correct. The prior developers of NL2Bash1 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." While we automatically supply the data from scraped web pages, we relied on the same process of having experts verify commands. This abstraction and simplification made it easier to verify the integrity and quality of the commands, but it also meant that we not only had 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. As of yet, we have been unsuccessful in recruiting outside help for verifying data, which means that we will have to do it ourselves, taking time that could otherwise be used to make the final product more efficient. We have, however, succeeded in the goal we had for our user interface, which was to simplify the verification process to improve the speed and lower the barrier of entry to verify data for our model.

Regardless of the challenges, 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. 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 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.

Our implementation has also incurred monetary costs, though they have been within our projections at $150. Virtually all of this money was spent on our GPU-compute machine on AWS which we used to retrain Tellina as we added additional data. We did not need to spend money on a data scraping server, because it ran just fine on a local machine, nor did we need to spend money on a server for the Tester UI, since we could just run it on UW CSE's ATTU cluster.

The remaining time and money costs we have yet to consider is that of the crowdsourcing itself, both for verification and for evaluating the final product. Because our web scraping actually turned out to be really easy, with plentiful potential NL/Bash pairs extracted, we can potentially do lots of verification to check each pair for correctness. And because we have chosen to do this verification, it does not incur any cost to us - it will just incur a large time cost as we add data to improve Tellina's output.

Another part of our project that we had difficulties with was scheduling. Our initial schedule was built under the assumption that we would use a .NET Windows Forms application to do our tester verification of the scraped results. Unfortunately, our work was delayed on this front for several reasons that became apparent as we worked on the project. In the former case, we made the decision that, in order to provide better cross-platform compatibility, we would write the Tester UI in HTML5 with django instead of .NET Windows Forms, which caused us to lose the development time spent on the .NET version. Because of this, we only made it approximately 75% of the way through our first schedule, reaching at Week 9 what we originally planned to reach at Week 6 or 7.

Our second schedule, however, did not account for another issue that arose in our project, which was that GPU-enabled machines were more difficult to obtain and configure than we thought. We originally thought that we might be able to use a personal GPU machine or one of the UW GPU machines to do the training, and that we would be able to start the training with relatively setup. However, after one or two weeks of attempting to use these machines with no success, it became apparent that our only real choice was Amazon AWS, and we discovered upon attempting to create a new GPU compute instance that each AWS account must be manually granted permission to use one, which took several days for us. However, because we decided that wide-scale distribution of our website was outside of the scope of our project, we still managed to meet most of our targets for our second schedule, which was more accurate than our first.

One of the primary draws to take this course was the ability to work on a cumulative project in an environment that would mimic what we could expect from future internships and jobs. We were given a large degree of freedom from project choice, to design, and finally implementation, all while continuously improving our project with iterative feedback. As we expected from the start, one of the greatest challenges in this course was communicating as a group. Prior project-based courses involved working with a single partner—a substantially different experience than working in a group of 4+. Unsurprisingly, it's hard to organize any number of people to work on a project at the same time, but trusting each of our group members and using GitHub alleviated a lot of the trouble and allowed for a lot of concurrent changes to be made to the project without having to consult the entire group at each step. The experience was stressful at times with some deadlines putting a lot of focus on our report and presentations rather than working on our project (one of the causes for our schedule inaccuracies), but ultimately one that we feel will improve our ability to work as a team in future employment. One of the most helpful parts of the experience was the weekly group meetings both individually and with Calvin. Individual meetings were helpful to make sure that the entire group was on the same page and avoid situations where different members had drastically different ideas about design or implementation details. Working with Calvin was also fantastic as a way to address concerns that we had overlooked and to spur discussion on interesting concepts that can be brought beyond the scope of this course. There were some parts of the course that were more helpful than others, but we are ultimately grateful to have gone through this experience, emerging as better programmers and teammates.

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.

Appendix: removed sections

Challenges

The primary challenges that we expect to face lie mainly in our inexperience with scraping online data, relying on crowdsourcing for a critical component of our project, and the costs incurred by training our Tellina learning model on Amazon AWS servers.

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.

We cannot completely automate the process of command verification; natural language processing today still has little to no ability to perfectly understand 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.” While we automatically supply the data from scraped web pages, we rely on the same process of having experts 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. The ultimate goal of our user interface is to simplify the verification process to improve the speed and lower the barrier of entry to verify data for our model.

Regardless of the challenges, 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. 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 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 will incur costs, between approximately $100-300, due to needing to use Amazon AWS to host several of the needed servers. A GPU-compute machine on AWS is currently being used to retrain Tellina with new data, which allows us to get training results relatively quickly and thus run multiple iterations as we scrape more data. AWS might also be used to host the server for data scraping, though free servers through the CSE lab might also take care of this problem.

Another possible monetary cost may come from crowdsourcing: we may run into the issue of not having enough volunteers to actually test data, which 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

Original 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 testing interface design presented in the user manual accompanying this document.

- By week four, we had planned 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 intended to determine how to train and use Tellina on our own machines by the end of this week. These goals have been partially accomplished. A working prototype of the web scraper has been developed, but the prototype for the testing interface is still in progress. Tellina has also been successfully set up on a team member's machine.
- By week five, we had planned to have a server set up to test and to run our verification system with Tellina's existing dataset, and we had planned to have collected a small dataset using our own web scraper to verify and use. These goals have been partially accomplished, as the server is still a work in progress, and we still have not been able to fully train Tellina. The training has taken much longer than expected. 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. Our initial tests should allow us to quickly check whether certain changes to our implementation positively affect our accuracy.
- 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.

Updated Schedule:

- In week 8, we plan to connect all part of the project into a pipeline that can be run automatically. That is, by the end of week 8, we plan to be able to run a single command that will start the server and start the web scraper gather data that will then be sent to the tester UI webpage. Once cleaned and verified, that data will then be automatically sent to the Tellina model, which will be retrained at some regular interval. As soon as this pipeline is working, we will focus on gathering as much data as possible for Tellina's dataset to measure our efficacy, and iteratively upgrade our systems with our findings. Our initial tests should allow us to quickly check whether certain changes to our implementation positively affect our accuracy.
- 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.

At this point, all of the goals up to week 6 of the original schedule have been completed. The server has been set up, and each project component is working in isolation. Data can be manually passed from component to component, but a pipeline has not been established yet to automatically move data through the system, from the web scraper through the tester UI to the training set for Tellina. The new schedule is slightly different from the original schedule, in that we had planned to have the pipeline complete, but getting each individual part of the project working too slightly longer than we anticipated. The only change is that automating the pipeline and beginning more regular data collection has been pushed up to week 8.

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.