Analysis and Prediction of Parameterized Commands in Implicit and Explicit Shell Workflows

Vraj Patel
The University of North Carolina at Chapel Hill
patelvap@cs.unc.edu

***Abstract* - The command line serves as a barrier to many novice users wishing to enter the technology sphere as its user interface has not changed over time. One method of easing novice users into the command line is to provide an interface that can predict the user's next command in a workflow to save time and correct errors which is in contrast to a scripted interface. This paper proposes a new algorithm for predicting commands whose novelty is the use of both implicit and explicit workflows, and a flexible definition for the length of an implicit workflow. The algorithm uses sliding windows, frequencies, removing sub-repetitions, and command graphs to predict a user's next command in a given implicit or explicit workflow. In testing, our algorithm can reach accuracies as high as 70% when predicting the fifth command of a command sequence in an implicit workflow and 40% when predicting the last command of an explicit workflow. Gaps in accuracy decrease between users of varying levels of experience when histories are sampled and testing our algorithm in a co-learning context drops accuracy significantly due to per-user shell history variability.**

Keywords - command prediction, Bash shell, CLI, user experience, sliding windows, decomposed pipes, command graphs, implicit workflow, explicit workflow

I. Introduction

A programmer can be considered to have a multitude of tools and configurations in their arsenal. These tools have evolved to increase productivity and ease of use through their user interfaces. For example, consider the text editor which has evolved from line editors such as ed, which is cumbersome, to vi and emacs, which have a strict learning curve, to today's editors of VS Code and IntelliJ, which take away many of the pains of the programmer such as syntax checking.

We focus on the command line interface (CLI). The CLI is a tool used to enter commands in a text-based interface. The shell is an example of a CLI that programmers use to interact with the operating system and some of its variants include Csh, Bash, and Zsh. It is a tool in the programmer's arsenal that has not had significant changes to its user interface and has remained the de facto tool for the programmer to interact with the operating system [6]. In some cases, shell commands have been replaced by Graphical User Interfaces (GUI) applications that provide an interface for direct manipulation [6]. Examples include the Activity Monitor for macOS which serves as a substitute for the `pkill` command, for killing a process, as well as the top command, for seeing a view of processes, which essentially provide editing rather than command line interfaces to manipulate OS state [14]. That lack of evolution begets a lack of accessibility when learning the shell and thus poses a daunting challenge for novice users who wish to take advantage of the capabilities of the shell [6]. The wide breadth of applications the shell provides also begets difficulty with users who are unfamiliar with or have forgotten certain shell commands. GUI applications attempt to replace the shell but are no substitute for the breadth of applications it provides [5].

One example of the breadth of applications the shell provides is the ability to create implicit and explicit workflows. We define workflows in the scope of this paper as a sequence of commands to perform a task. Workflows can be as simple as performing two commands in succession by changing a directory and listing files to a more complex task such

as formatting a file with `awk` or `sed`. Workflows in the shell can also be implicit or explicit. They can be implicit when the user simply enters a sequence of commands in succession or explicit through the use of pipes chaining multiple commands together into one unified command to enter into the shell prompt. Furthermore, shell scripting languages, such as Bash or Zsh, are Turing complete and provide the capabilities of looping and conditionals allowing the user to further create complex implicit or explicit workflows [13].

One such example of a GUI application that captures shell functionality is the CyVerse Discovery Environment (Discovery) which does so through direct manipulation [5] [6]. This GUI allows users to select shell programs (called applications) and manually configure their options. However, this application falls short in the amount of time it takes to create these workflows and is limited in that it does not support the scripting functionalities of the shell such as loops and conditionals.

With the assumption that the shell is still relevant and will not undergo a significant evolution, it is a reasonable assumption that any tool made to make the shell more accessible should be an addition to the shell itself. Two explored avenues for making the shell more accessible are a scripted approach and a predictive approach.

An example of a scripted approach that aims to assist novice users is the SuperShell. The SuperShell is a superset of the Bash shell that can be used to invoke canned scripts written by programmers who have tried to anticipate certain hypothesized problems [1]. These scripts are registered with the SuperShell and invoked when errors occur. The scripts, in anticipating problems, attempt to correct erroneous commands entered by presenting the user with options that may rectify their issues. However, the SuperShell is limited in its breadth of assistance as it relies on predefined canned scripts.

Scripted approaches, as a whole, have a limitation as their functionality is limited by the extent of the scripts. In contrast, a data-driven predictive approach is dynamic as it can adapt based on user command histories. An example of a predictive approach outside the shell is demonstrated in a graphical program called VisComplete. VisComplete is a GUI application that attempts to ease the learning curve of creating scientific workflows by offering predictions to specific users based on the collective history of all users. These scientific workflows consist of modules where each module is a processing step in the scientific workflow. VisComplete aggregates user workflows and presents potential workflows to the user that they can iteratively refine which will lead to a refined prediction [8]. We discuss VisComplete in more detail in Section V, relation to related works.

There are also a few methods of implementing a predictive approach for the shell. One method by Davison and Hirsh is to use decision trees to predict the next command stub. A command stub is defined as the program name of a command. For example, the command stub of the command `swipl -s file.pl` is `swipl`. The features of the decision tree are the two previous command stubs which are used to predict the third command stub. This method achieves an accuracy of 38% in predicting the user's next command stub [10] [11]. Another method by Durant and Smith is to use decision tables to predict the next command. These decision tables also use the previous two commands to predict the third command and are faster to create than the decision tree. This method achieves an accuracy of 40% in predicting the user's next command [9]. The final method known to us is by Korvemaker and Greiner who use a Mixture of Experts model to predict the user's next full command. The Mixture of Experts model contains four individual models. Two of these models use a linked list for each command where each node contains a distribution of potential next commands. The third model uses a frequency table of the last 100 commands to predict the distribution of the next command and the fourth model is used only to predict the first command of the session [2]. It is important to note that these methods only are in the realm of predicting implicit workflows and fall short in analyzing and predicting explicit workflows such as pipes.

This paper will present an alternative predictive approach to the models discussed above. We aim to differ from previous works by defining a workflow to be of a variable length instead of just two commands, using a graph data structure, and replacing command line arguments. To do so, we chose to implement a variable sized sliding window parser to use previous 1 through 4 commands to make a prediction as well as a graph data structure that stores sequences of commands that represent workflows. These workflows can either be implicit or explicit. In the case of the explicit workflows of pipes, we decompose them to a sequence of commands that is the equivalent implicit workflow. Our

graph data structure is keyed and traversed by command stub and a prediction is made by taking the highest frequency full commands. Our variable sized sliding window parser is motivated by how previous works only look at the two previous commands to predict a third and thus uses a rigid definition of a workflow. Arguably, exploiting explicit workflows and using a more flexible definition of an implicit workflow and its length can yield better predictions. This method can achieve accuracies as high as 70% depending on how many previous commands are used to predict the next full command and the type of user.

The rest of this paper is structured as follows: Section II describes the data source behind the research in this paper; Section III describes the parsing and predictive methods of the prediction algorithm; Section IV describes the results, describing how a prediction is determined to be accurate and the results of the various experiments; Section V describes related works in greater detail and compares them to the command prediction algorithm and results described in this paper; Section VI has the contributions; Section VII describes avenues for future work including using this algorithm as a plugin for the shell. We also include all of our results in the appendix in table and graph form along with some noteworthy observations.

II. Data Source

As we are using a data-driven approach to predicting shell commands, we require a set of data that is disambiguated by type of user to determine how well our predictive algorithm works based on the user. Fortunately, we were able to use a command line data set collected by Saul Greenberg at the University of Calgary. The Greenberg data set contains shell history data collected from 168 users of the UNIX Csh over 4 months. The 168 users fell into four groups. Novice programmers who were students in an introductory Pascal course that had little to no exposure to UNIX-like shell; experienced programmers who were senior computer science undergraduates expected to have a fair knowledge of the UNIX environment, computer scientists, composed of faculty, graduates, and researchers, and non-programmers, who predominantly performed word-processing and document preparation tasks. The Greenberg dataset notably includes the start and end times for a login session and the full line entered by the user. We use these two data points in the research behind this paper. There are approximately 302,000 command lines from the 168 users with 62,000 distinct command lines translating to an average of 467 per user as well as an average of 89 distinct command stubs per user [3] [4].

In analyzing this data, Greenberg noticed the following trends in shell usage behavior. First, a substantial portion of each user's previous actions is repeated. Second, new command lines are regularly composed despite many actions being repeated. Third, users exhibit considerable recency in the commands they enter. Fourth, many commands entered are unique as a significant number of recurring commands are not covered by the last few items. And fifth, when a user uses the history to recall a command line, they typically recall the same command lines repeatedly [3] [4]. We find these trends notable as knowing users regularly repeat commands and exhibit recency potentially signifies that if we can isolate these repetitions, we can predict them.

III: Parsing and Predicting

A. Parsing Data

1. Nested Session Lists

Each of the users in the Greenberg data set had an associated file with their user type and with commands in sequential order segmented by session start and end time. We parse these commands in a three-stage parsing method. In the first stage, we individually parse these files by session into lists grouped by the type of user. For example, given 3 computer scientist users each with 5 sessions logged and 3 experienced users each with 3 sessions logged, we would parse the data into a first nested list of 15 lists of commands for computer scientists and a second nested list of 9 lists of commands for experienced programmers. In this example, each list is a session. These sessions, however, do not explicitly segment workflows, with the exception of pipes which we count as individual workflows, and a given session could contain many different workflows. These sessions are not segmented explicitly due to how the shell records history but workflows still exist implicitly in cases such as: editing a program, compiling, testing, and pushing to remote; starting

a local database and web server application for local development; or formatting data for printing. Shell workflows are also moving targets and can be interrupted by external events such as Slack messages [2]. For example, if a user is taking steps in initializing a new Node project but receives a message asking to push commits to remote for another project, the user must interrupt their workflow in initializing the Node project to push the requested commits which would subsequently be reflected in the user's history. Thus, identifying implicit workflows is a challenging task.

2. Sliding Command Windows

As neither the shell nor our data set explicitly records implicit workflows, we instead elect to divide our parsed sessions into sequences of commands that could potentially represent workflows. To do this, we implement our second stage of parsing by applying the variable sized sliding window idea upon each session, where the size of the window is 2 through 5, inclusive. We chose to use a sliding window parser with this varied window size as we aimed to obtain all implicit workflows with the session data. We assume no workflow, implicit or explicit, will consist of more than 5 commands. We show a small example of applying the sliding window parser below in figure 1. Command line arguments are replaced in this figure with numbers preceding with `$` which will be explained later in this subsection. For the sake of brevity, we only show the sliding window parsing method for a session of 4 commands.

Before variable sliding window parsing:

```
[ cd $0, ls -lAh, vim $1.c, make ]
```

After variable sliding window parsing:

```
[
      [ cd $0, ls -lAh ],
      [ ls -lAh, vim $1.c ],
      [ vim $1.c, make ],
      [ cd $0, ls -lAh, vim $1.c ],
      [ ls -lAh, vim $1.c, make ],
      [ cd $0, ls -lAh, vim $1.c, make ]
]
```

Figure 1: Sliding window parsing example.

As shown, a session of 4 commands results in 3 sequences of 2 commands, 2 sequences of 3 commands, and 1 sequence of 4 commands. The sliding window parsing method is meant to exhaustively extract all possible workflows from a user and a collective of users' history to use in prediction. We can intuitively determine that the 6 command chains each are a standalone workflow. The first is changing a directory and listing its files, the second is listing files and opening one for editing, the third is opening a file for editing and then compiling and so on. We determined that having repeated commands within our graph data structure is acceptable as it allows us to get an exhaustive list of all workflows in a session.

```
1 Parse_Commands_Into_Subsets(parsed_list: lst[str], subset_size: int)
2   result = [] # empty list
3
4   for i in range(0, len(parsed_list)
5     if (i + subset_size) < len(parsed_list)
6       result.append(parsed_list[i:i+subset_size])
7
8   return result
9
10 Parse_Commands_Sliding_Window(parsed_list: lst[str])
11   # parsed_list is a list representing a session of sequential commands
13
14   result = []
15
16   for i in range(2, 5)
17     subset = Parse_Commands_Into_Subsets(parsed_list, i)
18
19     result.append(subset)
20
21   return result
```

Figure 2: Sliding Window Algorithm Pseudocode.

3. Replacing Arguments

The sliding window parsing method creates what we call command sequences where each sequence is chronologically ordered and of the same length as our window, 2 to 5. We further parse these command sequences in our third stage by replacing file arguments in the format of `${arg num}` where "`arg num`" is a number corresponding to a unique argument. We do not replace the file extension, program flags, or the actual program itself. For example, given the following command sequence, `[ls -la documents/, cd documents/, cd working_dir/]`, we would replace the arguments to result in `[ls -la $0, cd $0, cd $1]`. This approach of normalizing commands provided more data for predicting commands as it makes new workflows to be predicted from previous workflows that differed only in arguments used. Pseudocode for this algorithm is shown below in figure 3.

```
1 Replace_Arguments(command_set: list[str])
2   arg_dict ← {}
3   arg_counter ← 0
4
5   for command in command_set
6     for word in command
7       if word is type(file_name)
8         if arg_dict.contains(file_name)
9           updated_word ← arg_dict[file_name]
10        else
11          arg_dict[file_name] ← "$" + arg_counter
12          arg_counter += 1
13          updated_word ← arg_dict[file_name]
14
15        command[word] = updated_word
```

Figure 3: Pseudocode for Replacing Arguments

B. Graph Construction

With the data parsed by our three-stage parsing method, we were then faced with the issue of how to store the data. We opted to take a similar route to VisComplete, which models the history of explicit workflows as graph objects, and store these command chains in a graph data structure [8]. We further elected to use a graph data structure as it could represent the sequential nature of command line data, where one command will succeed another command, in a directed acyclic fashion. In this data structure we will store all the potential workflows we extracted from the three-stage parsing method. The top-level of this graph is a dictionary of graph nodes that are keyed by the program stub, such as `vim`, the first argument in a command line, indicating the program to be invoked. Individual graph nodes contain the following fields: commands, a dictionary with keys of full shell commands (e.g. `vim $0`) and a count of how many times the command appeared in the history used for prediction; frequency, the number of times the command stub has appeared in the history; and children, a dictionary of graph nodes that contains the command stub and commands that immediately succeed the current graph node's commands in a command chain. We show pseudocode of a GraphNode object's fields below in figure 4. As nodes are keyed by command stubs, two different command sequences can appear in the same

```
1 class GraphNode:
2    command_stub: String.
3    frequency: Integer
4
5    # Map with key of full shell command and value of
6    # frequency of full shell command occurring in command
7    # set
8    commands: Map<String, Integer>
9
10   # Map with key command stub and value of corresponding
11   # graph node. The command stub field of the graph node
12   # equal to the key command stub
13   children: Map<String, GraphNode>
```

Figure 4: Graph Node Class Fields.

graph path as long as the command stubs of the sequences are the same. For example, the two command sequences in figure 5, while different, will appear in the same path as the stubs are the same. The `ls` child node following the `cd` top-level parent node will have two entries in its commands dictionary field and the `grep` child node following the `ls` parent node will similarly have two dictionary entries in its commands dictionary field.

Each command sequence obtained from the sliding window parsing method is added to the graph. The first command in a sequence will correspond either be a new node in the top-level dictionary or an increment to the frequency of an existing node if the command stub has been seen. The subsequent command in the sequence will be placed in the children dictionary field of the graph node and the next command in the children dictionary of the previous child node. If a child node does not exist, then one will be created. Otherwise, we simply increment the frequency of occurrence. This algorithm is shown in pseudocode in figure 6. Figure 8 below shows a truncated example of one of the top-level graphs as a result of this graph construction algorithm shown in figure 6. The top-level `ls` node has a frequency of 5 and contains 2 children graph nodes, `cat` and `vim`, which each have one child node which, in turn, each have a child node of their own with the `vim` node having a chain of 3 children. The list of command sequences in figure 7 would produce the graph in figure 8.

```
1. [cd $0, ls -lAh, grep -rnw $1]
2. [cd $0, ls -la, grep -r $1]
```

Figure 5: Command Sets Occupying the Same Path.

We were also presented with the issue of repeated command chains such as `[cd, ls, cd, ls, cd]` which is a false workflow chain in that it is composed of repeated executions of the true workflow `[cd, ls]`. This is apparent in many of the users such as one computer scientist whose top 3 most common command chains of length 5 were `ls`

and `cd $0` alternating repeatedly. When encountering repeated subsequences within the command chain, we truncate the command chain and give the command an increased frequency in our graph data structure. For example, the previous repeated command chain of `[ls, ls, ls]` would translate to `[ls (3)]`. Similarly, a command chain such as `[ls, cd, ls, cd, make]` where the repeated group is a sequence of two commands would translate to `[ls (2), cd (2), make]`. In these examples, the values in the parenthesis represent the value by which to increment the frequency in the graph data structure. The increment value is 1 for commands without a number in parenthesis.

```
1 Construct_Graph(command_sets: list[list[str]]):
2    command_dict ← {} # empty map
3
4    for command_set in command_sets:
5      command_stub ← command_set[0].split(" ")[0]
6      first_command ← command_set[0]
7
8      if command_dict[command_stub] is None
9        node ← GraphNode(command_stub←command_stub, frequency←1)
10       command_dict[command_stub] = node
11     else
12       node ← command_dict[command_stub]
13       node.frequency += 1
14
15     if node.commands[first_command] is None
16       node.commands[first_command] ← 1
17     else
18       node.commands[first_command] += 1
19
20     for command in command_set[1:] # index 0 taken care of above
21       command_stub ← command.split(" ")[0]
22
23       if node.children[command_stub] is None
24         child_node ← GraphNode(command_stub←command_stub, frequency←1)
25         node.children[command_stub] ← child_node
26       else
27         child_node ← node.children[command_stub]
28         child_node.frequency += 1
29
30       if child_node.commands[command] is None
31         child_node.commands[command] ← 1
32       else
33         child_node.commands[command += 1
34
35
36       node ← child_node
37
38   return command_dict
```

Figure 6: Graph Construction Pseudocode.

```
[
   [ls -l, vim $0],
   [ls -lAh, vim $0],
   [ls -la, vim $0, git add ., git commit -a, git push origin main],
   [ls -la, cat $0 | less],
   [ls -lAh, cat $0 | less, vim $0, make run],
   [ls -lAh, cat $0 | less, vim $0]
]
```

Figure 7: Sample Command Sequence.

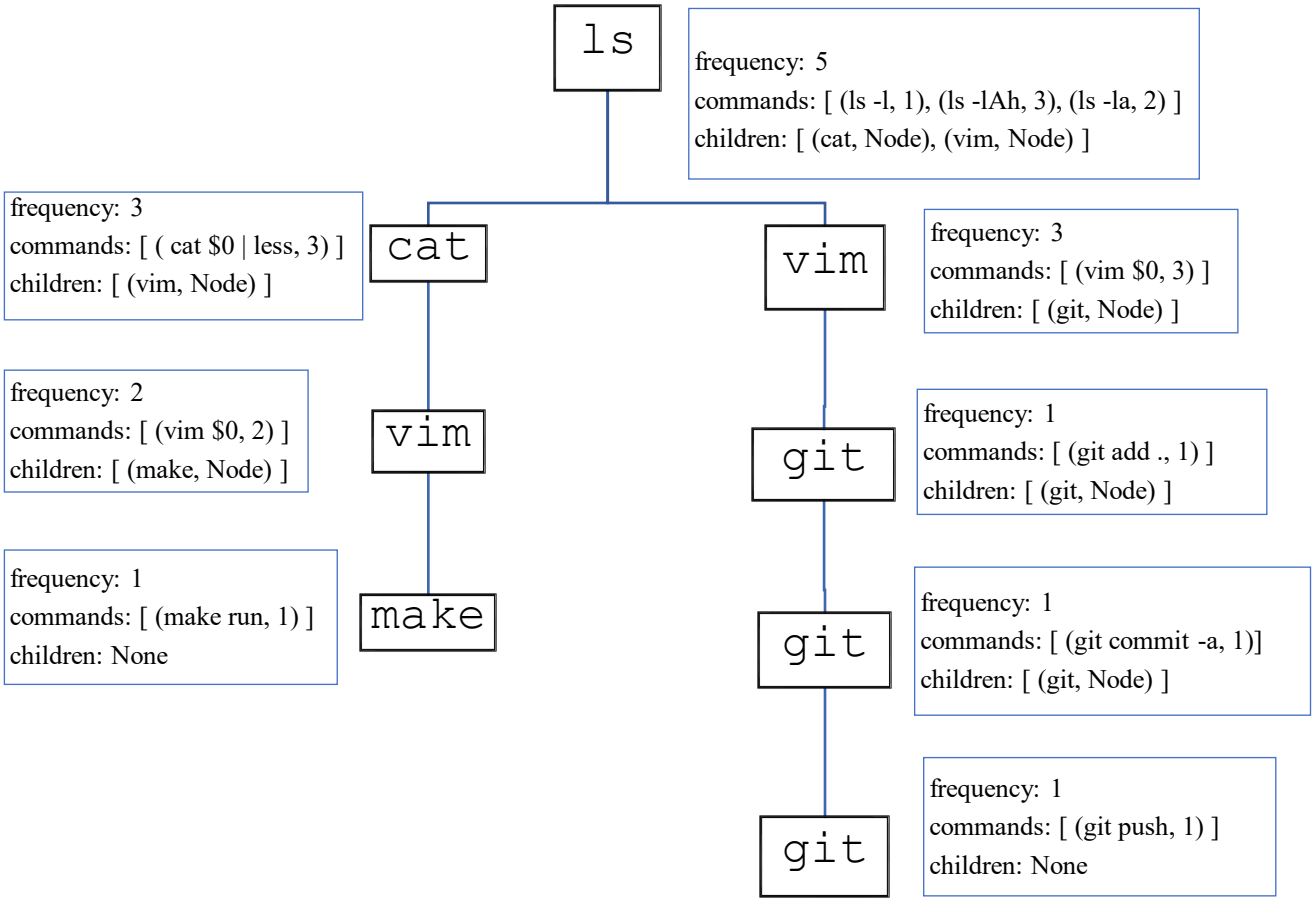


Figure 8: Graph Created from Command Sequence in Figure 7.

C. Obtaining a Prediction

With the data structure populated, we now move to consider how to predict the next command. We split the command chains resulting from the three stages of parsing randomly into an 80/20 train/test split. We use the train set to construct the graph data structure and the test set to determine whether our method can accurately predict the next command. We define accurately predicting the next command if our algorithm can accurately predict the N^{th} command in a command chain from the 1 to N-1^{th} commands. To traverse the graph with the 1^{st} to N-1^{th} commands, we use only the command stubs. The 1^{st} command stub gives the key for which graph node to enter from the aforementioned top-level dictionary. The 2^{nd} command stub gives the key for which child graph node to enter from the 1^{st} command stub's dictionary field of command nodes. We repeat this previous step until we reach the child node corresponding to the N-1^{th} command stub. If at any point a node for a particular command stub does not exist, we simply return None as no prediction can occur without a path in the graph. This algorithm is shown in pseudocode below in figure 9.

```
1 Get_Prediction(commands: list[str], graph: map, num_predictions: int)
2    command_stub ← commands[0].split(" ")[0]
3
4    # Traverse graph with given commands
5    if graph[command_stub] is None
6       return None
7    else
8       node ← graph[command_stub]
9
10   for command in commands[1:]
11      command_stub ← command.split(" ")[0]
12
13      if node.children[command_stub] is not None
14         node ← node.children[command_stub]
15      else
16         return None
17
18   # At last node in graph, now get prediction list
19   child_nodes ← node.children.values()
20   commands ← child_node.commands for child_node in child_nodes
21
22   # sort commands by descending frequency
23   # note: command.value() is the frequency of the command appearing
24   commands ← sorted(commands.items(), key = command.value())[:num_predictions]
```

Figure 9: Get Prediction Pseudocode.

As the child node corresponding to the N-1th command contains a dictionary of child nodes with values of frequency, we take all command-frequency pairs from the commands dictionary field of the graph node, sort them in descending order of frequency, and return the top 3, 5, and 15 predictions to determine both if we made an accurate prediction and how accurate our top predictions are.

Let us walk through the following command chain using the graph in figure 8 as an example.

`[ls -lAh, cat $0 | less, vim $0]`

The first command stub of `ls` tells us to traverse the ls node. The second command stub of `cat` tells us to enter the `cat` child node of the `ls` node. The third command stub of `vim` tells us to enter the `vim` child node of the `cat` node. At this point, we are ready to predict the next command. Since `make` is the only child of the `vim` node, we can take the top N full commands of the highest frequencies from the `make` node's commands dictionary field and return the top 3, 5, and 15 predictions.

D. Expanding and Predicting Pipes

UNIX pipes are unique in that they allow the user to explicitly segment their workflow by chaining multiple command stubs into a single command. We can decompose pipes into command sequences with redirection, replace file arguments, construct a graph, and run our predictive algorithm on these sequences to see how well predictions fare on segmented workflows.

Given a UNIX command with pipes, `nroff -me $0.nr | colcrt | page`, we can expand it to the following sequence of 3 commands: `[nroff -me $0.nr > $1, colcrt < $1 > $1, page < $1]`. In this example, `$1` represents the temporary file used in redirection as an intermediate for a pipe in our replacement algorithm. Pseudocode for this algorithm is shown below in figure 10.

```
1 Expand_Pipe_Command(pipe_command: str)
2    expanded_commands ← []
3    input_redirection ← "< input_temp"
4    output_redirection ← "> output_temp"
5
6    split_command ← pipe_command.split('|')
7
8    for i in range(len(split_command))
9      if i is 0
10       command ← split_command[i] + output_redirection
11     else if i+1 is len(split_command)
12       command ← split_command[i] + input_redirection
13     else
14       command ← split_command[i] + input_redirection + output_redirection
15
16     expanded_commands.append(command)
17
18   return expanded_commands
```

Figure 10: Expanding Pipe Pseudocode

IV: Results

This section will describe the results of our predictive algorithm on all the commands in the Greenberg data set both segmented by the user as well as all four types of users consolidated, decomposed pipes, as well as co-learning where we create our graph data structure with all but one of each type of user and attempt to predict the commands of the held-out user with all the other users. For decomposed pipes and co-learning, we do not aggregate the four types of users as different types of users use a disjunct sequence of commands pertaining to their experience with the shell [3].

A. Determining Accuracy

We use fuzzy matching to determine if a particular prediction is accurate. Fuzzy matching is a technique used to determine how close two strings are via the Levenshtein Distance metric, also known as the Minimum Edit distance [7] [12]. If a predicted command and the actual command receive a fuzzy match score of at least 85, we count that prediction as accurate. We chose a score of 85 to be optimal considering the relatively short length of shell commands and quasi-infinite variance of shell commands as mentioned before. In practice, this results in predicted commands being off by a few characters.

Our accuracy function outputs the following fields: the proportion of predictions that were correct in the top 3, 5, and 15 predictions, the proportion of predictions that were incorrect in the top 3 predictions, the proportion of predictions that resulted in a None prediction meaning that the path did not exist in the graph data structure, the proportion of predictions that were correct in the first prediction, as well as the average relative frequencies of predictions in the graph for successful and failing predictions. We define average relative frequency as the average frequency of all the nodes a command sequence in the testing set traversed divided by the size of the train set. Furthermore, command sequences in the test set that resulted in a None prediction were not counted as valid predictions. This means that a valid prediction must consist in either a correct or incorrect prediction and that the proportion of correct in 3 predictions and proportion of incorrect in 3 predictions will be equal to 1.

We show an example of a successful prediction for a window size of 4 below in figure 11. In this case, the first three commands in the command sequence field are used to traverse the graph data structure, the fourth command is the expected command, and the results are the list of predictions generated by the algorithm. The numbers in the second value of the tuples are the value of the frequency field in the Graph Node.

We show an example of a failing prediction for a window size of 4 below in figure 12. In this case, the expected command is not predicted due to it either not being present at that point of the graph or not having a high enough

frequency to make it relevant. In this case, the expected command is in the graph but its frequency of 3 is far too low to be relevant compared to the frequencies of the predicted commands.

```
'Command Sequence': [
       ('ver -Pplsin $0', 150),
       ('w', 33),
       ('fg', 15),
       ('ver -Pplsin $0', 3)],

'Expected':
       ('ver -Pplsin $0', 3),

'Results': [
       ('lpr -Pplsin $0', 258),
       ('ver -Pplsin $0', 150)]
```

Figure 11: Example of successful prediction.

```
'Command Sequence': [
       ('cp $0.ml $1', 10713),
       ('e', 1761),
       ('e $1', 318),
       ('page $0', 3)],

'Expected':
       ('page $0', 3),

'Results': [
       ('e $0', 657),
       ('e $0.l', 651),
       ('e', 300),
       ('make $0', 213),
       ('cd $0', 150)]}
```

Figure 12: Example of failing prediction.

B. All Commands

The following section describes notable results of the predictive algorithm described in the previous section when testing the entire Greenberg dataset with each type of user as well as all users aggregated together. The full results will be in appendix A in tables and graphs with some insights but we include notable results below.

For predicting the second command from the first, we see accuracies of 36%, 38%, 39%, 61%, and 43% within the top 3 predicted results for computer scientists, experienced programmers, non-programmers, novice programmers, and all programmers combined, respectively. These accuracies increase to 42%, 46%, 46%, 69%, 51% for the top 5 predicted results.

For predicting the third command from the first two, we see accuracies of 52%, 56%, 55%, 71%, and 57% within the top 3 predicted results for computer scientists, experienced programmers, non-programmers, novice programmers, and all programmers combined, respectively. These accuracies increase to 60%, 65%, 64%, 78%, 65% for the top 5 predicted results.

For predicting the fourth command from the first three, we see accuracies of 68%, 71%, 65%, 78%, and 69% within the top 3 predicted results for computer scientists, experienced programmers, non-programmers, novice programmers, and all programmers combined, respectively. These accuracies increase to 76%, 78%, 74%, 84%, 76% for the top 5 predicted results.

For predicting the fifth command from the first four, we see accuracies of 71%, 74%, 69%, 80%, and 70% within the top 3 predicted results for computer scientists, experienced programmers, non-programmers, novice programmers, and

all programmers combined, respectively. These accuracies increase to 75%, 78%, 75%, 84%, 74% for the top 5 predicted results.

For using the aggregated user data to predict the Nth command from the first N-1 commands, we see accuracies of 42%, 45%, 45%, 64%, and 75% within the top 3 predicted results for computer scientists, experienced programmers, non-programmers, novice programmers, and all programmers combined, respectively. These accuracies increase to 48%, 52%, 53%, 70%, 80% within the top 5 predicted results.

From using all the data, we see we reach our highest accuracy when predicting the fifth command from the first four which is a slight improvement over our accuracy metrics when predicting the fourth command from the first three. As such, we can assume that when using histories that are so expansive, using the previous three to four commands to predict the user's next command is ideal and gives the highest chance of being correct. This numerical observation is consistent with the data graphed in Appendix A subsection 6 which show a peak in correct in 3 and correct and 5 proportions for all four types of users and the users aggregated. Interestingly, all types of users as well as all users aggregated show a peak in accuracy in the correct in 15 metric when predicting the fourth command from the first 3 indicating that the graph data structure contains the correct command but without the frequency required for it to be prominent enough such that it appears in the top 3 or top 5. We also observe that novice programmers tended to have the highest accuracies for metrics measuring correctness out of the four types of users as well as all users consolidated. The inverse is true for computer scientists who tended to have the lowest accuracies for measuring correctness with the exception of predicting the 4^{th} and 5^{th} commands which non-programmers had the lowest accuracies. One potential reason for novice users tending to have the highest accuracies is their limited set of commands due to inexperience with the shell. Likewise, computer scientists had the highest variability in shell commands which would lead to lower accuracies [3][4].

C. Sampled Data

The Greenberg data set contains 302,000 commands collected over several months. This is unlikely to be the case in practice as available shell histories are likely to be significantly smaller. As such we randomly sampled 10% of the collective session lists of the four types of users and ran our predictive algorithm. The full results will be in appendix B in tables and graphs with some insights but we include notable results below.

For predicting the second command from the first we see accuracies of 43% in the top 3 predicted results and 51% in the top 5. For predicting the third command from the first two we see accuracies of 59% in the top 3 predicted results and 67% in the top 5. For predicting the fourth command from the first three we see accuracies of 72% in the top 3 predicted results and 80% in the top 5. For predicting the fifth command from the first four we see accuracies of 85% in the top 3 predicted results and 87% in the top 5. For predicting the N^{th} command from the first N-1 commands we see accuracies of 64% for the top 3 predicted results and 71% in the top 5.

For our sampled data, see that computer scientists, experienced programmers, and non-programmers tend to have accuracies hover around the 50% mark except when predicting the second command from the first. This intuitively makes sense as the sampling may have unintentionally changed the distribution of workflows from the non-sampled data. Novice programmers are the exception to this observation as their accuracy proportion is always above 65% and is 60% for predicting the second command from the first which is a category the other three types of users struggled to have high results in. We also notice that the aggregated programmer data appears to have the highest prediction accuracy in the top 3 predictions when predicting the fourth and fifth commands from the first three and first four respectively. This observation is not consistent in the unsampled data and also not consistent with aggregated user data. Additionally, it appears that there is a significant increase in the proportion of None predictions when predicting the fifth command from the first four. This is not seen with the aggregated user data graphed where there is only a slight increase in the proportion of None predictions accompanied by a peak in accuracy for correct predictions. The full data in table and graph form can be seen in Appendix B.

D. Decomposed Pipes

Predicting pipes through the expansion method outlined in the previous section yielded the following results. For computer scientists, we were able to accurately predict the Nth command from the first N-1th commands 40% of the time in the top 3 predictions and 41% in the top 5. For experienced programmers, we were able to accurately predict the Nth command from the first N-1th commands 23% of the time in the top 3 predictions and 26% in the top 5. For non-

programmers, we were able to accurately predict the Nth command from the first N-1th commands 24% of the time in the top 3 predictions and 26% in the top 5. For novice programmers, we were able to accurately predict the Nth command from the first N-1th commands 26% of the time in the top 3 predictions and 26% in the top 5.

We believe that computer scientists had the highest accuracy as they are the most experienced in using the shell and are more likely to explicitly segment workflows in the form of pipes compared to the other three types of users. Interestingly Non-programmers had an astonishingly high correct in 15 proportion compared to the other types of users. A potential reason for this is that non-programmers may have only used a predefined set of explicit workflows in the form of piped commands while those who are experienced with or learning how to use the shell may be encouraged to be more creative with what commands they chain together. Similarly, these explicit workflows are defined by the user and thus are prone to their idiosyncrasies in accomplishing a task. With that intuition, it makes reasonable sense to assume that is the reason for the low accuracy results of decomposing pipes.

The full results are in Appendix C in tables and graphs.

E. Co-learning

In the co-learning context, we use all but one of each type of user to predict the held-out user. As such our graph data structure consists of commands from all but one user and we test accuracies with only the held-out user. The full results will be in appendix D in tables and graphs with some insights but we include some results below.

For predicting the second command from the first, we see accuracies of 35%, 29%, 23%, and 41% in the top 3 predictions for computer scientists, experienced programmers, non-programmers, and novice programmers, respectively. These accuracies increase to 41%, 33%, 34%, and 43% for the top 5 predicted results.

For predicting the third command from the first two, we see accuracies of 34%, 29%, 26%, and 33% in the top 3 predictions for computer scientists, experienced programmers, non-programmers, and novice programmers, respectively. These accuracies increase to 40%, 35%, 31%, and 35% for the top 5 predicted results.

For predicting the fourth command from the first three, we see accuracies of 30%, 21%, 26%, and 25% in the top 3 predictions for computer scientists, experienced programmers, non-programmers, and novice programmers, respectively. These accuracies increase to 36%, 35%, 30%, and 45% for the top 5 predicted results.

For predicting the fifth command from the first four, we see accuracies of 26%, 31%, 27%, and 17% in the top 3 predictions for computer scientists, experienced programmers, non-programmers, and novice programmers, respectively. These accuracies increase to 32%, 32%, 31%, and 17% for the top 5 predicted results.

We see a notable drop in accuracies when compared to the results from both all commands and sampled commands. This may be due to how one shell user may not use the same workflows or commands as other users making it notably more difficult to predict shell workflows for a user without their history. Additionally, we see a large increase in the proportion of none predictions for all four types of users when predicting the fifth command from the first four, potentially due to the held-out user using command chains of length 5 not typically used by other users in their group. Furthermore, we see that all metrics besides proportion of None predictions remain relatively stable despite the command number attempting to predict. This does not hold true for the novice programmers which has a large jump in inaccuracy when predicting the fifth command from the first four.

The full results are in Appendix D in tables and graphs.

V: Relation to Related Work

A. GUI Applications

The CyVerse Discovery Environment (Discovery) can be viewed as a graphical alternative to the shell. Discovery allows users to select shell programs, called applications, and configure parameters corresponding to options for the program. For example, for the `grep` program in the following image in figure 13, the user can manually enter the various options of `grep` such as the input and output file names, and text to match as well as select additional true or false options such as a case insensitive search, inverse matching, and returning only the count.

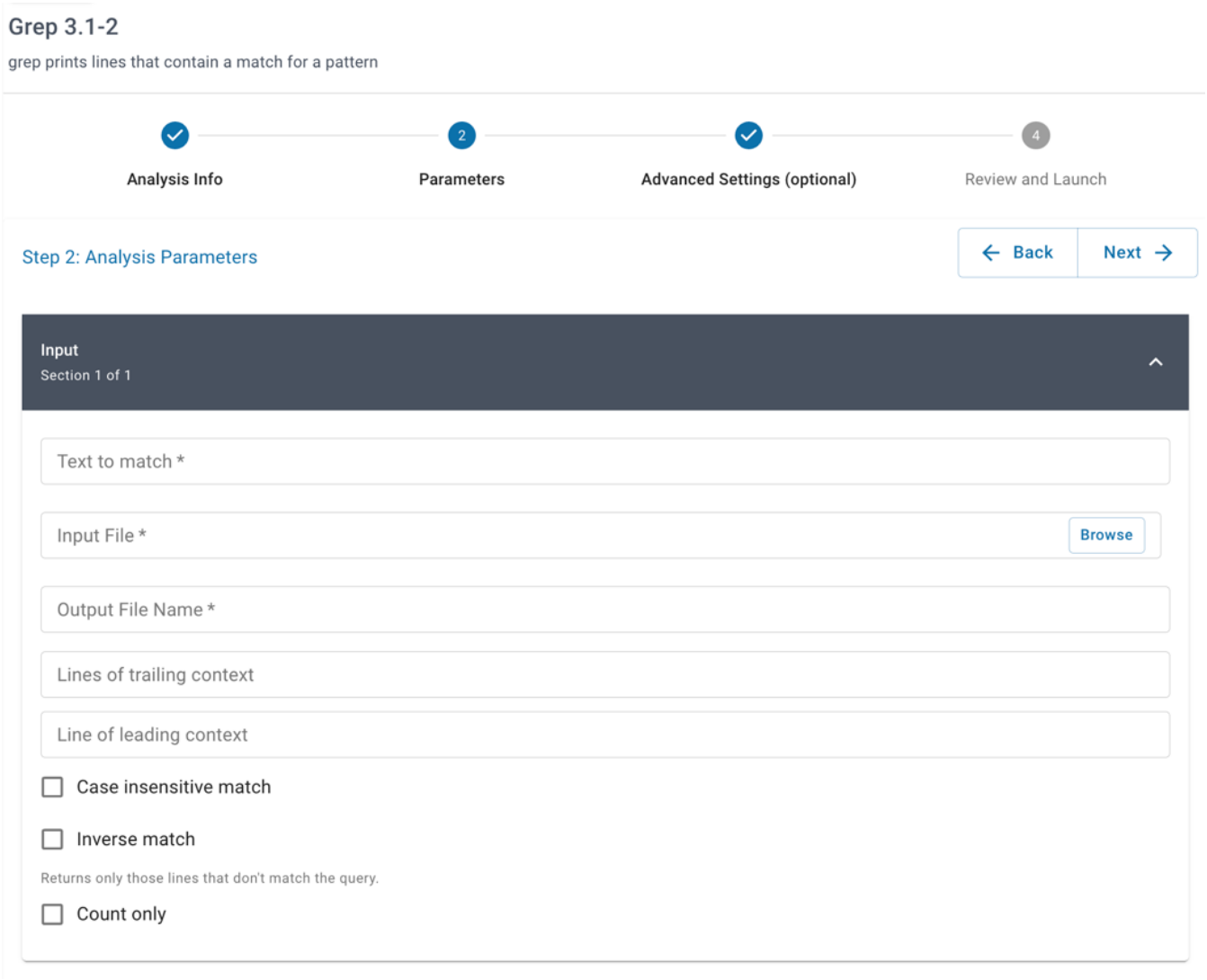


Figure 13: Application Configuration Menu in Discovery.

Discovery also allows users to chain applications together into creating workflows that would be the equivalent of UNIX pipes. These applications are executed sequentially with one program's output being used as the subsequent program's input as shown below in figure 14.

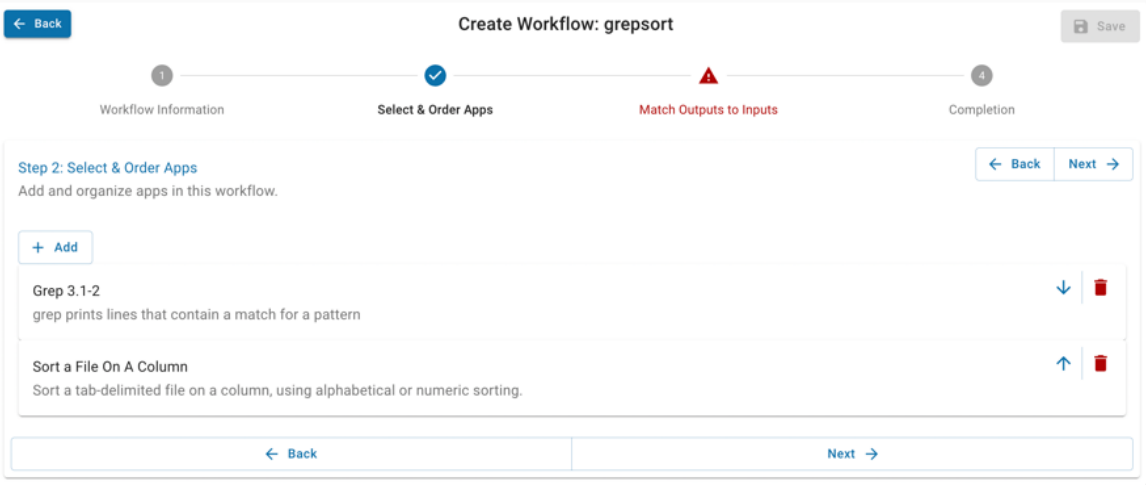


Figure 14: Chaining Applications into Workflows in Discovery UI

Users can define workflows as combinations of applications and enter parameters for each application in a unified interface. For example, the configuration in the image below in figure 15 is equivalent to the following Bash command: `cat A_1_abundance.tsv | grep 'enst00000391'| sort -n -k 2`. As we see a translation from GUI workflows to shell commands, we can conclude that the work described in this paper is applicable to GUIs as well.

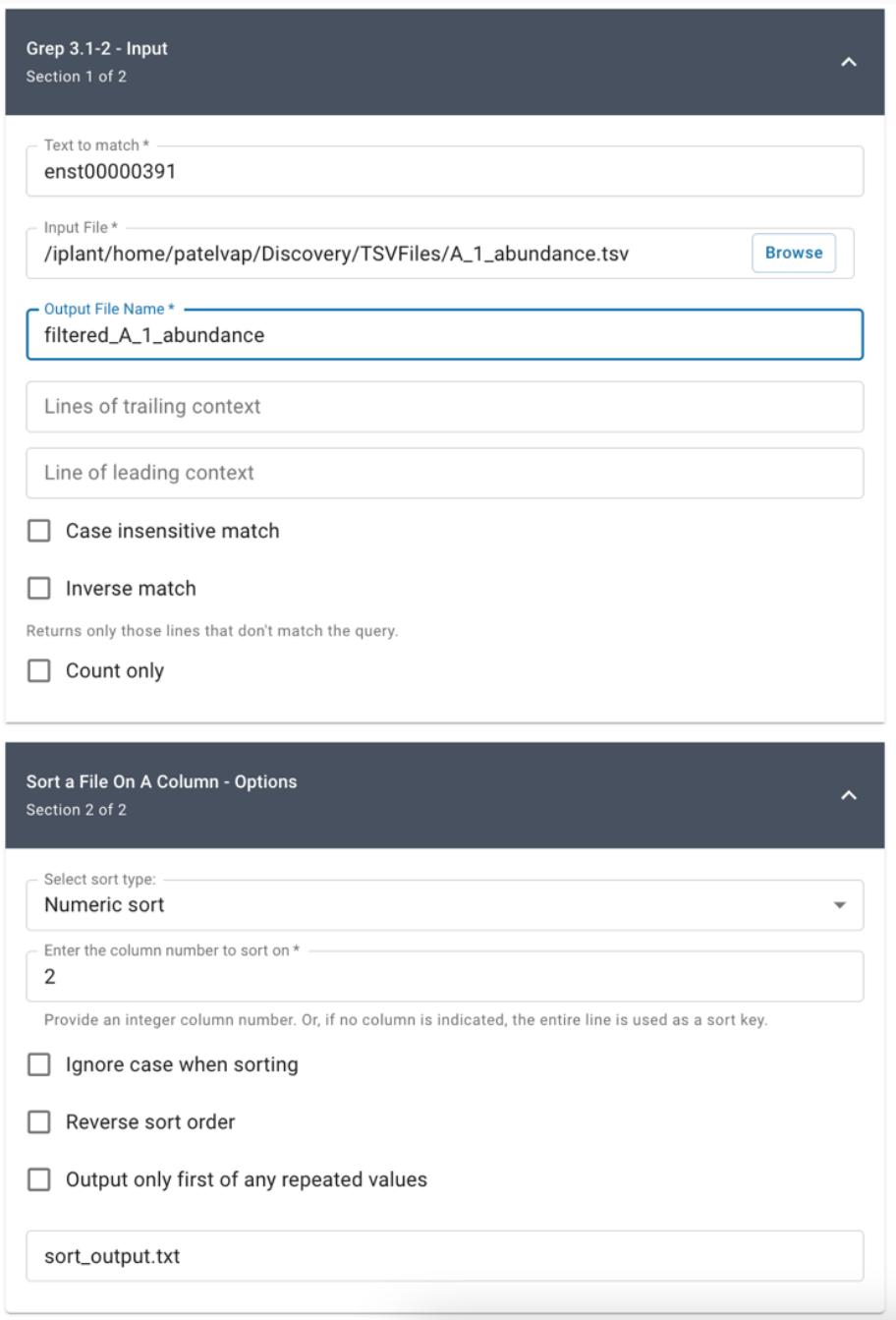


Figure 15: Workflow Configuration Menu in Discovery

However, the Discovery interface has shortcomings as an alternative to the shell. For one, application workflows take time to configure. A user has to manually select applications to be in a workflow and then manually enter their desired parameters. It is quicker and easier for an experienced user to use the one-line shell command with pipes shown above. Secondly, Discovery lacks programming constructs offered by the shell such as loops and conditionals which allow

users to create scripts to repeat tasks instead of manually repeating them. While the user interface is far friendlier than the shell, novice users still need to know what each application and its parameters do to use them effectively and combine them into workflows. In short, this GUI falls short in easing the learning curve for novice users in learning the shell as while it has a more appealing user interface, it lacks the functionality to bridge the knowledge gap without the user searching through documentation of the large amount of applications Discovery offers.

B. SuperShell

SuperShell is a superset of the Bash Shell that aims to assist novice users through canned scripts written by programmers who are trying to anticipate problems a user would have when learning the Bash Shell. In other words, SuperShell is an example of a scripted approach towards making it easier to use the shell. SuperShell has several novel features including integrating shell history with versioning implicitly; recording detailed command history with the user's full command, associated file (reading/writing), file contents, standard input, standard output, standard error, and timestamp; searching detailed command history; and locating file versions that introduced an error and helping the user go back to a previous version of the file due to erroneous edits. SuperShell introduces a script that abstracts many of these features called `shelp`. This script suggests reasons for errors and possible solutions to errors and the script is either run or the user is prompted to run the script when standard error is detected [1].

To aid the `shelp` script, SuperShell defines and continually updates the following environment variables with the shell history: LAST_MODIFIED, LAST_COMPILED, LAST_EXECUTED, LAST_FILENAME, LAST_STDERR, LAST_STDOUT, and LAST_COMMAND. These variables all refer to the file name the user is working with except for the last three [1].

The `shelp` script uses predefined "SuperShell Rules" in an attempt to identify the cause of an error and provide a possible solution. SuperShell will run all predefined rule scripts that are a collection of many if-statements which check if environment variables meet certain conditions. If any of these suggestions are true then the associated suggestions are stored in a temporary suggestions file which is then presented to the user [1].

As mentioned before, SuperShell is a scripted approach to helping novice users become accustomed to the shell. With that, SuperShell does fall short in its reliance on canned scripts to help users. The programmer of SuperShell must define these scripts in anticipation of user behavior which implies that the scope of user behavior and capacity of error recovery is finite. As a learning tool for novice users, the SuperShell is particularly useful with predefined anticipated workflows for the users where the programmer can predefine scripts that can help users within their working scope. Furthermore, the SuperShell poses little utility to the experienced user in assisting them with their errors as they are more likely to know the cause of their errors. Additionally, SuperShell cannot intelligently adapt to a user's workflow as the scripts `shelp` uses are predefined and must be continually modified to match new workflows.

C. VisComplete

VisComplete is a graphical application that allows users to create pipelines to process scientific data. These pipelines are analogous to shell workflows but their commands are less standard than the generic commands provided by the shell. As a result, they tend to be so complex that novice users tend to spend a disproportionate amount of time constructing pipelines compared to experienced users.

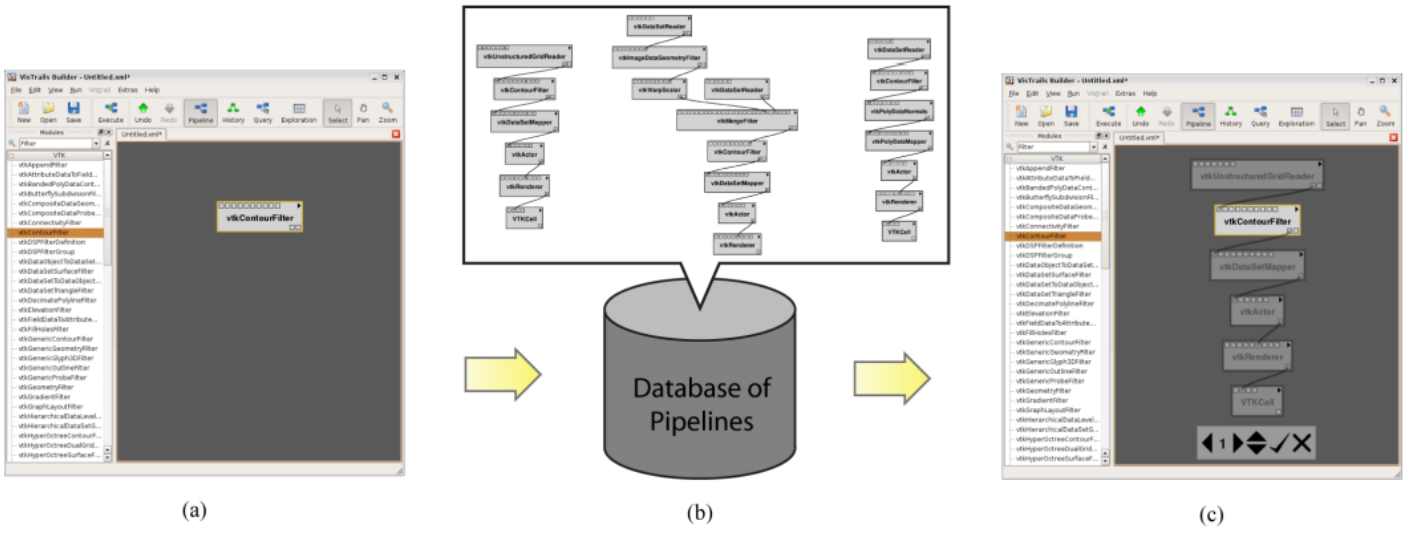


Figure 16: VisComplete Presenting Predictions.

VisComplete pipelines are represented as graph objects internally and the application provides autocomplete suggestions based on the history of the user and the aggregated history of other users. These autocomplete suggestions are presented to the user in a non-obtrusive fashion from which the user can choose pipeline stages they want in their workflow which in turn refines VisComplete's prediction to present newer, refined autocomplete suggestions to the user. This is shown below in figures 9 and 10 where figure 16 shows the user interface for presenting a prediction and figure 17 shows how predictions are chosen based on the path with the larger frequency in the database. In other words, the algorithm will iteratively refine itself per user addition and gradually give fewer suggestions based on pipeline maturity [8].

Figure 17: VisComplete Determining Predictions based on Frequency of Paths in Database.

VisComplete stores pipeline history in a database of pipelines and uses a scoring system to judge recommendations. These histories are stored in a database of pipelines to provide suggestions (figure 16). The full graphs are not stored in the pipeline database, but rather the paths are along with their frequency of occurrence (figure 17). These scores are computed by paths, not subgraphs, in the upstream path. Predictions are presented to the user based on a confidence metric shown by the following equations in figures 11-13:

$$c(v) = \frac{\sum_{P \in upstream(v)} count(v \mid P)}{\sum_{P \in upstream(v)} count(P)}$$

Figure 18: Confidence of a Single Vertex c(v).

$$c(G) = \prod_{v \in G} c(v)$$

Figure 19: Confidence of a Graph G.

$$c(p_{i+1}) = c(p_i) \cdot c(v)$$

Figure 20: Confidence of New Prediction.

Figure 18 is an equation that shows how VisComplete determines the confidence of a single vertex as a measure of how likely that vertex is given the upstream paths. This is computed by the sum of the counts of the vertex v following a given path P divided by the sum of the counts of a path P. Figure 19 shows the confidence for a whole graph G is the product of the confidences of each of its vertices. Figure 20 shows how predictions, and therefore confidences, are computed iteratively as a new prediction is the product of the confidence of the previous prediction and the confidence of the new vertex [8].

The VisComplete algorithm serves as the basis of the presented command prediction algorithm for the shell. As such, it is important to note the key differences between VisComplete and the shell. For one, VisComplete predicts both upstream and downstream nodes in the graph while we predict only downstream commands, providing an auto complete facility at the workflow level rather than the token level. Furthermore, predicting command N+2 in the context of the shell is far more difficult than predicting command N+1 as we can potentially accurately predict command N+1 from the previous N commands in the history if it is part of a shell workflow but the list of potential N+2 commands is exponentially larger due to our graph data structure. Shell commands also have a large variance as there is a finite set of programs for the vast majority of users but a quasi-infinite set of arguments through command line options and program arguments. This is in stark contrast to nodes in VisComplete which do not contain parameters so each node is an individual topic unit. Additionally, workflows in VisComplete are always explicit since the user creates them and each node is explicitly known to the predictive system unlike in the shell where the predictive system has to infer workflows. Furthermore, as previously mentioned, predicting the next Bash command is a moving target as external events, such as the user receiving a Slack message, can add commands that do not belong to the logical workflow making workflow inference more difficult [2]. An example of this can be when a user is attempting to set up a new project directory but receives a message so they are prompted to change directories to complete their new task. This means that the underlying command distribution is not stationary.

As previously mentioned, the shell also doesn't directly segment workflows, unless the user does so explicitly, as VisComplete does with its graph system of creating pipelines. This means that there is not necessarily a distinct start and end of a workflow when viewing the history of a Bash session. Pipes, however, are a feature of the shell that do segment workflow but that is a result of the user's actions, not the action of the shell. The user can have workflows without having explicit workflows such as: editing a file, compiling, testing, pushing to remote version control; starting a local database and web server application; formatting a LaTeX document for printing. Thus, only a limited number of explicit workflows are created, especially by novice users.

D. Decision Trees to Predict UNIX Command Stubs

One of the earliest works in the area of command prediction in the shell was by Davison and Hirsh (D&H). The authors gathered UNIX history data similarly to Saul Greenberg from 75 students and 2 faculty at Rutgers University. On average, each student gave 2184 commands with an average command length of 3.77. The researchers used a C4.5 decision tree as their predictive model and determined accuracy in an online fashion where they construct the tree on the previous N command stubs and tested if the predicted command is equal to the current command stub due to the sequential nature of the data. They trained the decision tree with 2 features, the previous 2 command stubs, and achieved a 38% accuracy score in predicting the next command stub [10] [11]. This figure of 38% is from the top prediction of the decision tree while this paper explores returning the top N predictions where N is 3, 5, or 15.

The researchers also created a prototype shell that incorporated their predictive algorithm called `ilash` or Interactive Learning Apprentice SHell which is an extension to the UNIX shell `tcsh`. This extension would present the predicted command stub in the prompt string and allow the user to insert the command stub with a keystroke. However, the decision tree was not self-updated with the user's history through continual use and had to be updated typically once a day per the researcher's recommendation [10] [11].

The decision tree method falls short in that it only trains on and predicts command stubs, i.e., the program being executed. For example, the stub for `grep -rnw "match_string"` is `grep`. The method presented in this paper does not have such a limitation and predicts full commands with arguments replaced but the program name, flags, and the number of arguments are maintained. Intuitively speaking, predicting the full command, with options and arguments, would yield both greater ease of use for the novice user and increased productivity for the advanced user.

E. Decision Tables to Predict UNIX Commands

Durant and Smith aimed to build off of D&H's work in command prediction, particularly by applying boosting to decision trees to get better results. Instead of the C4.5 decision tree, they elected to use the J48 decision tree from the Weka system, which is a variation of the C4.5 decision tree. They used the same features as D&H, namely the 2 previous commands. With a boosted decision tree, the researchers were able to achieve an accuracy of 42%. However, the decision trees could not be generated for histories with more than 2,000 commands as it was too computationally intensive. As a compromise, the authors created a decision table using the Weka system, also using the previous 2 commands as features. Decision tables are a simpler, less compute-intensive algorithm than decision trees that resulted in a small decrease of accuracy to 40%. An example of a decision table entry is shown below where the first two entries are the previous two commands and the third is the command that would be predicted in this approach [9] [10] [11].

cd workingdir/	vim file.c	gcc file.c

What is notable about Durant and Smith's additions to D&H's research is that they were able to achieve these accuracies with a relatively small history set of 100 to 1,000 commands. However, they were not able to achieve a significant accuracy boost over D&H's original results potentially showing that decision trees and decision tables may not be the most optimal approach in predicting UNIX commands with high accuracy compared to the method described in this paper [9] [10] [11].

F. Predicting UNIX Commands with a Mixture of Experts Model

Korvemaker and Greiner also aimed to extend the works of D&H using the Greenberg dataset by incorporating the following: predicting complete commands and including the additional features of the previous command's error code, time of day, and the day of the week. However, adding these additional features resulted in deteriorated performance. This resulted in them modifying the work of D&H by using a Mixture of Experts Model that resulted in 47.9% accuracy in predicting the next command [2].

The authors also compared the results of the Mixture of Experts model to a simpler approach of populating a conditional probability table (figure 21) referring to this approach as AUR. This table contains three columns, Command_t, potential Command_{t+1}s, and the probability of the potential Command_{t+1}s following Command_t. With this simple approach, they were able to achieve a 47.4% accuracy which is very close to the results of the Mixture of Experts Model [2].

Command_t	Command_{t+1}	Prob
vi cw.tex	latex cw.tex	1.0
latex cw.tex	dvips cw.dvi dvips cw.tex	0.8 0.2
dvips cw.dvi	gv cw.ps	1.0
dvips cw.tex	dvips cw.dvi	1.0
gv cw.ps	vi cw.tex wall	0.8 0.2

Figure 21: Conditional Probability Table. Probability of Command_{t+1} following Command_t.

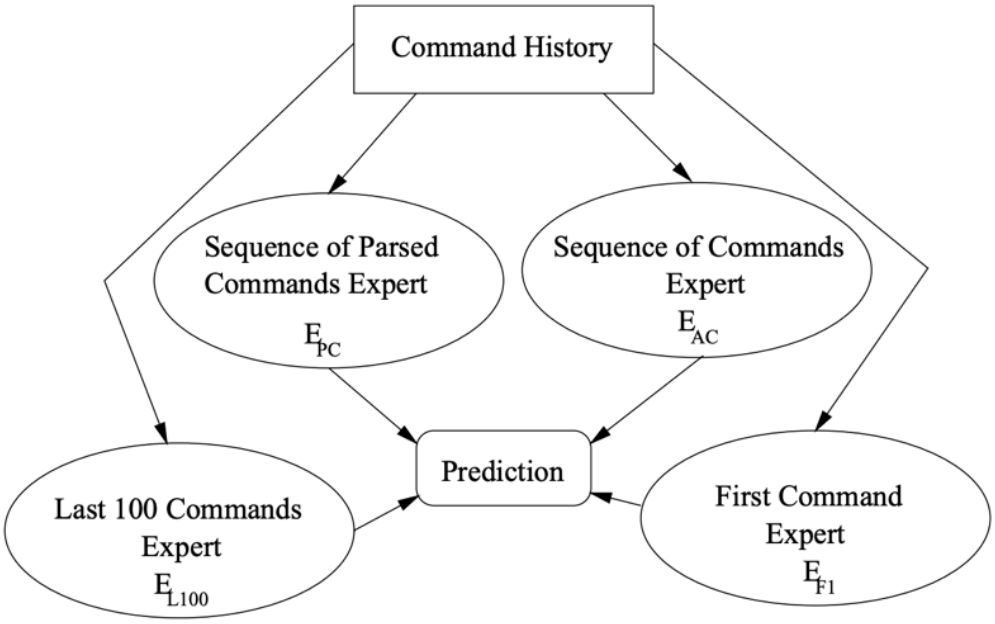


Figure 22: Diagram of the Mixture of Experts Model.

The researchers aimed to use any possible combination of 2 commands in a command pair to predict a third similar to how D&H used the previous 2 commands to predict a third. However, this would quickly result in an unlearnable number of features. With a history of 500 commands, this would result in 500^2 command pairs with the number of features only increasing when considering the researchers' intent to include the features of error code, time of day, and the day of the week [2].

As only a relative few command pairs occur frequently, maintaining all command pairs would lead to overfitting the model. Furthermore, with so many features, came the need for dimensionality reduction. The Mixture of Experts model came in an attempt to solve this problem while still incorporating the additional features. This model contained 4 experts: m Previous Actual Commands, m Previous Abstracted Commands, Short-Term Frequency Predictions, and Predicting the Sessions First Command. A diagram of this model is shown in figure 22 [2].

The m Previous Actual Commands Expert consists of a data structure that conditions on the previous command pair when that pair is significant, i.e., it has occurred frequently before, otherwise it just conditions on the previous command. This data structure consists of nodes that contain the command and a distribution of potential next commands and these nodes can have children to represent significant command chains that have occurred frequently. Figure 23 shows an illustrated version of this data structure. In practice, the researchers found the levels of children rarely exceeded 3 indicating that significant command chains were relatively short in length [2].

The m Previous Abstracted Commands Experts works the same way as the m Previous Actual Commands Expert with the exception that file arguments are parsed and replaced with generic, matchable terms, similar to how the research in this paper does, allowing it to reuse patterns in matching new terms. The Short-Term Frequency Predictions Expert maintains a frequency table of the last 100 commands in the shell's history and predicts the distribution of the next command from the frequency table (see figure 21 for an example of a frequency table). The Predicting the Sessions First Command Expert is relatively simple as it uses a probability table composed of the first command from the last 100 sessions to predict the first command of the session. As a result, this expert is only active for the first command of each session [2].

The accuracy of the Mixture of Experts model is a significant improvement over the decision tree and decision table used by the previously mentioned works. The authors also implemented this model into `zsh` in an interface where the user is presented with the top 5 predictions and can use the F-keys to select a prediction that will be automatically entered in the prompt of the shell. This results in a more interactive shell experience than `ilash` as now the user can choose from a list of full commands instead of just one stub. Of all the predictive approaches in the shell covered, the Mixture of Experts model is by far the best. However, it is very complex and does not provide a significant improvement

over the simple AUR method that only uses a conditional probability table (figure 21) described earlier nor does it reach accuracy levels as high as our predictive approach when looking at the accuracies of the top 5 predictions returned.

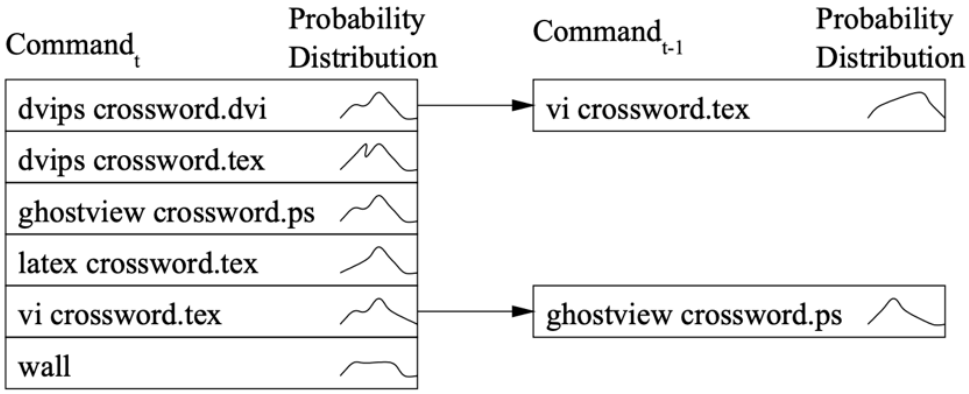


Figure 23: Data Structure used in Mixture of Experts Model

VI: Contribution

This paper has described a previously unexplored method of predicting a user's next shell command. We do this without standard Machine Learning techniques and in a novel approach compared to previous related works on command prediction with top three predictions yielding an accuracy as high as 70%. This novel approach uses a variable size sliding window parser and a graph data structure for traversing command workflows and predicting based on frequencies of previously seen commands. This approach differs from previous works in both how a workflow is defined as well as the data structure used to store commands. Specifically, we differentiate and define implicit and explicit workflows, where the latter consists of multiple commands chained together through pipes, and we store workflows in a graph data structure. These implicit workflows are created from the variable size sliding window parser which takes 2 to 5 sequential commands in a shell history to define as a workflow with the aim to extract all possible implicit workflows from a user's history. The sliding window parser is unique to our method as previous methods used a constant number of previous commands in a user's shell history to predict a command.

We also examine the results of applying our algorithm in a co-learning context, where we attempt to use the histories of all but one type of user to predict the held-out user. While the accuracies of this application were much lower than our standard applications described in section 4 subsections B through D, the results are still significant as it indicates that a user's shell workflows have an element of uniqueness to them that cannot be predicted well with shell workflows from a collection of other users.

In the next section, we describe how this interface can be implemented as a plug-in for the shell.

VII: Future work

While this paper describes a method to predict the user's next shell command, there lies a further opportunity in integrating this approach into the shell. Integration into the shell can benefit both the novice user, who may lack sufficient knowledge of the shell to efficiently use it, and the advanced user, who may find a system that can predict their next command accurately would increase their productivity.

Since the command prediction algorithm is user history-based, users in the same setting, such as the same programming class, can consolidate their histories to form more accurate predictions for each other. In that same vein, a shell integration could also bias predictions towards the current user's idiosyncrasies or potentially even predefined workflows set by the instructor of a course. This method would, of course, tie in a scripted approach into a predictive approach.

As discussed before, the third stage of parsing replaces command line arguments with `${arg num}` values. This means that the predicted command will also have command line arguments replaced. Since the third stage of parsing does not arbitrarily replace arguments but instead keeps track of repeated arguments to replace them with the same argument number, we can potentially recover the original argument of the current command workflow and present it back to the user as the prediction. For example, if the user enters `vim file.c` and then wishes to compile the file, the command `vim file.c` will be replaced with `vim $0.c` internally in the graph data structure, `gcc $0.c -Wall -o $0.o` will be predicted (provided the path in the graph exists and the frequencies of the command sequence in the history are prominent enough), and then the `$0` will be replaced with "`file`" to finally output `gcc file.c -Wall -o file.o` to the user.

Furthermore, we see that our accuracy metrics varied depending on the number of previous commands used to make a prediction, specifically we see that using the past 3 or 4 commands yields prediction accuracies of around 70% compared to around 35% and around 50% for using the past 1 or 2 commands, respectively. With that noted, one could potentially incorporate a system similar to the Mixture of Experts model described in Section V that looks at the previous 1, 2, 3, and 4 commands in the history to perform the predictive algorithm and take the best predictions based on some determined heuristic [2]. This determined heuristic could be a conditional probability equation, like the one used by K&G, or something simpler that would check and return predictions that were common between the predictions of using the past 1, 2, 3, and 4 commands.

Finally, we can integrate the predictive algorithm into the shell as a plugin. While the Bash shell is the default shell on many Linux distributions, Zsh is a newer variant that can support plugins [13]. K&G implemented their predictive method as a Zsh plugin and we propose a plugin with the same semantics [2]. These semantics are to simply present the user with several top predictions from the predictive algorithm which the user may choose or not choose with the F-keys to enter automatically into the shell prompt.

References

[1] S. Ray, "Towards a Helpful Shell: Scenarios and Commands," 2020.

[2] B. Korvemaker and R. Greiner, "Predicting UNIX Command Lines: Adjusting to User Patterns," *AAAI,* 2000.

[3] S. Greenberg and I. H. Witten, "How Users Repeat Their Actions on Computers: Principles for Design of History Mechanisms," *ACM,* 1988.

[4] S. Greenberg, "Using Unix: Collected traces of 168 Users," 1988.

[5] P. Dewan, B. Joyce and N. Merchant, "Human-Centric Programming in the Large - Command Languages to Scalable Cyber Training," *IEEE,* 2018.

[6] B. Schneiderman, "Direct Manipulation: A Step Beyond Programming Languages," *IEEE,* 1983.

[7] S. Kesh, Medium, 27 April 2020. [Online]. Available: https://medium.com/analytics-vidhya/fuzzy-matching-in-python-2def168dee4a. [Accessed 14 April 2022].

[8] D. Koop, C. E. Scheidegger, P. S. Callahan, J. Freire and C. T. Silva, "VisComplete: Automating Suggestions for Visualization Pipelines," *IEEE,* 2008.

[9] K. T. Durant and M. D. Smith, "Predicting UNIX commands using decision tables and decision trees," *WIT Press,* 2022.

[10] H. Hirsh and B. D. Davison, "Experiments in UNIX Command Prediction," Rutgers University, 1997.

[11] H. Hirsh and B. D. Davison, "Toward An Adaptive Command Line Interface," Rutgers University, 1997.

[12] Seatgeek, "seatgeek/thefuzz: Fuzzy String Matching in Python," GitHub, 22 April 2021. [Online]. Available: https://github.com/seatgeek/thefuzz.

[13] "Comparison of command shells," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_command_shells.

[14] P. Dewan and M. Solomon, "An approach to support automatic generation of user interfaces," *ACM,* 1990.

Appendix

A. Predicting Commands with All Data

1. Predicting the 2nd command from the 1st

	Computer Scientists	Experienced Programmers	Non-programmers	Novice Programmers	All Programmers
Correct in 3	35.74%	37.64%	39.30%	60.57%	42.63%
Correct in 5	42.81%	46.63%	46.44%	68.89%	50.50%
Correct in 15	60.13%	64.04%	65.05%	80.18%	66.32%
Incorrect in 3	64.26%	62.36%	60.70%	39.43%	57.37%
None Proportion	0.00%	0.00%	0.00%	0.00%	0.00%
First Prediction	20.47%	20.66%	22.07%	40.90%	25.77%
Avg Success Freq.	0.19	0.33	1.04	0.37	0.05
Avg Fail Freq.	0.21	0.37	1.19	0.39	0.05

2. Predicting the 3rd command from the 1st through 2nd

	Computer Scientists	Experienced Programmers	Non-programmers	Novice Programmers	All Programmers
Correct in 3	51.78%	56.31%	55.15%	71.38%	56.81%
Correct in 5	60.29%	64.66%	64.25%	77.87%	64.96%
Correct in 15	76.80%	80.20%	79.24%	88.37%	79.43%
Incorrect in 3	48.22%	43.69%	44.85%	28.62%	43.19%
None Proportion	0.00%	0.00%	0.00%	0.00%	0.00%
First Prediction	34.43%	35.00%	32.87%	49.40%	36.53%
Avg Success Freq.	0.23	0.39	1.23	0.40	0.05
Avg Fail Freq.	0.25	0.42	1.34	0.43	0.05

3. Predicting the 4th command from the 1st through 3rd

	Computer Scientists	Experienced Programmers	Non-programmers	Novice Programmers	All Programmers
Correct in 3	68.03%	70.60%	65.27%	78.04%	68.78%
Correct in 5	75.52%	78.44%	74.24%	84.18%	76.36%
Correct in 15	87.03%	89.01%	87.08%	92.69%	87.73%
Incorrect in 3	31.95%	29.40%	34.73%	21.95%	31.21%
None Proportion	0.04%	0.01%	0.00%	0.02%	0.03%
First Prediction	47.98%	50.40%	42.10%	41.89%	44.13%
Avg Success Freq.	0.25	0.41	1.30	0.41	0.05
Avg Fail Freq.	0.26	0.44	1.39	0.44	0.05

4. Predicting the 5th command from the 1st through 4th

	Computer Scientists	Experienced Programmers	Non-programmers	Novice Programmers	All Programmers
Correct in 3	70.55%	73.89%	68.87%	79.67%	70.00%
Correct in 5	74.53%	78.08%	75.08%	83.60%	74.27%
Correct in 15	80.12%	82.62%	82.73%	88.01%	79.57%
Incorrect in 3	28.85%	25.57%	30.76%	20.20%	29.45%
None Proportion	2.03%	2.09%	1.17%	0.64%	1.81%
First Prediction	54.76%	57.73%	48.29%	61.12%	53.59%
Avg Success Freq.	0.25	0.42	1.32	0.42	0.06
Avg Fail Freq.	0.27	0.47	1.40	0.44	0.06

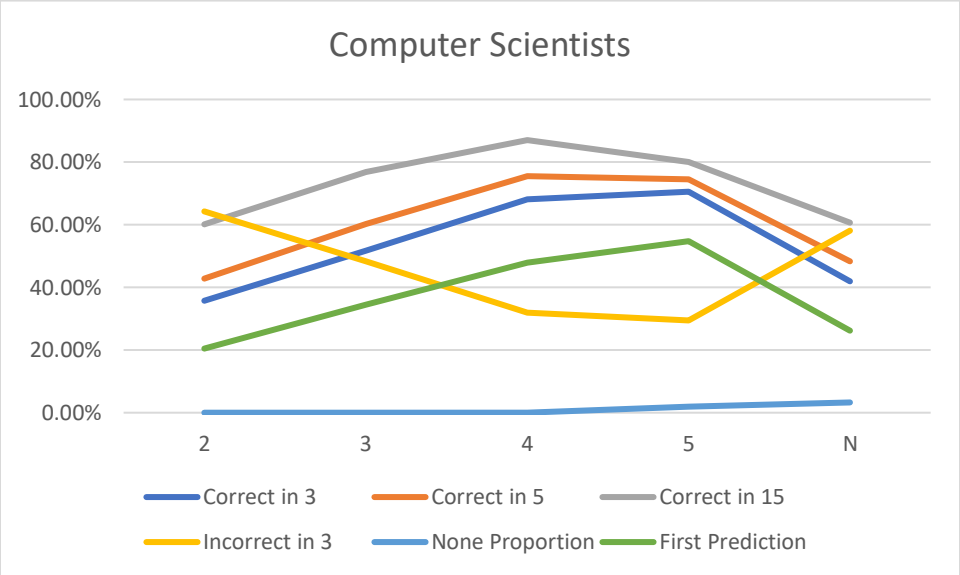
5. Predicting the Nth command from the 1st through N-1th

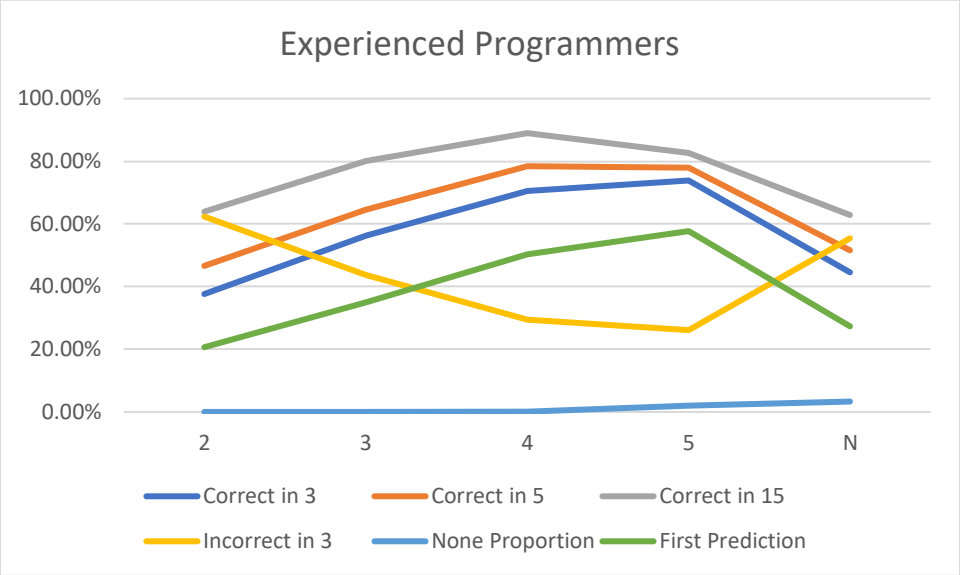
These results are from predicting the last command using all but the last command from all the command subsets formed through the variable sized sliding window parser.

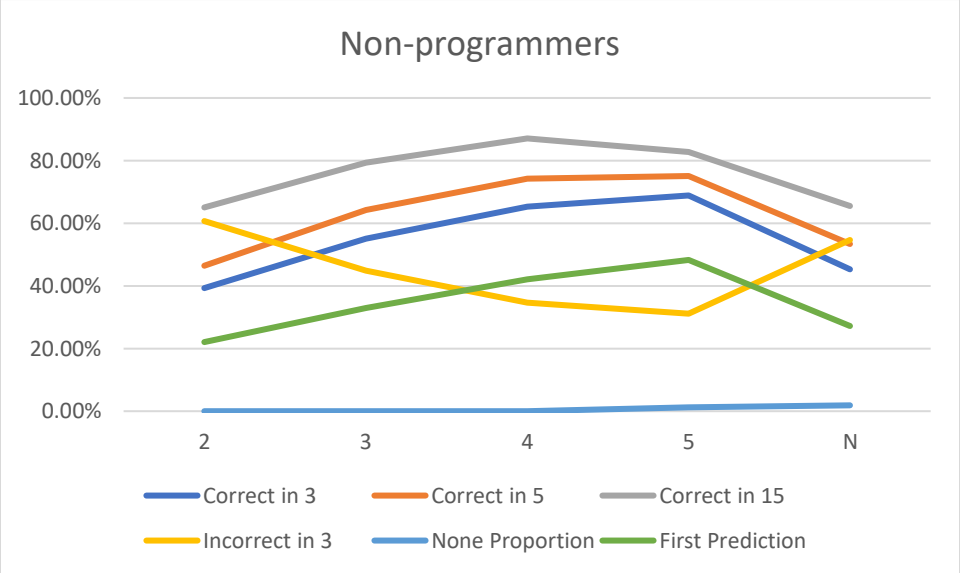
	Computer Scientists	Experienced Programmers	Non-programmers	Novice Programmers	All Programmers
Correct in 3	41.87%	44.55%	45.25%	64.12%	75.43%
Correct in 5	48.35%	51.53%	53.45%	70.11%	70.11%
Correct in 15	60.57%	62.86%	65.53%	78.49%	78.49%
Incorrect in 3	56.23%	53.61%	53.73%	35.33%	35.33%
None Proportion	3.27%	3.31%	1.87%	1.51%	1.51%
First Prediction	26.27%	27.23%	27.29%	40.91%	40.91%
Avg Success Freq.	0.02	0.04	0.14	0.06	0.06
Avg Fail Freq.	0.02	0.04	0.14	0.05	0.05

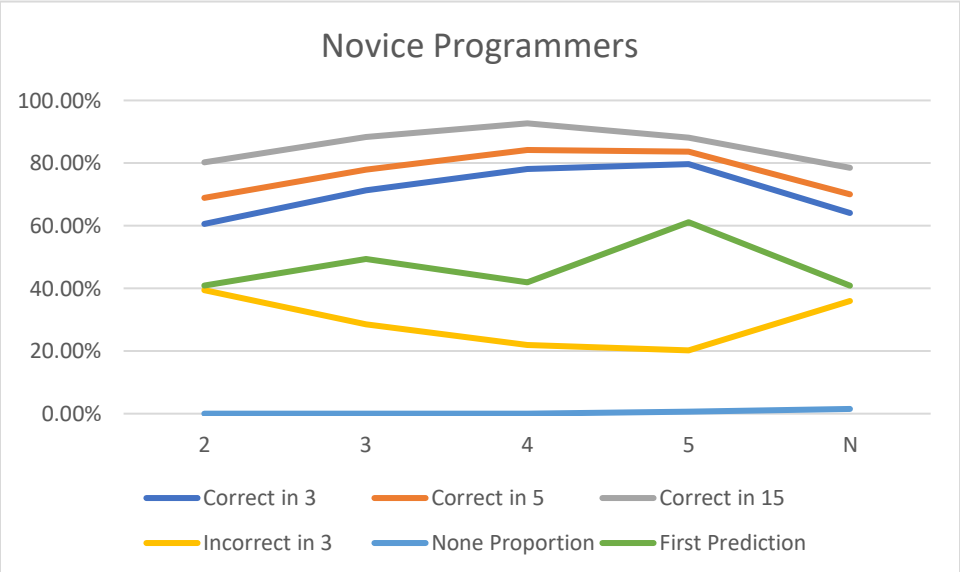
6. Data Graphed by User

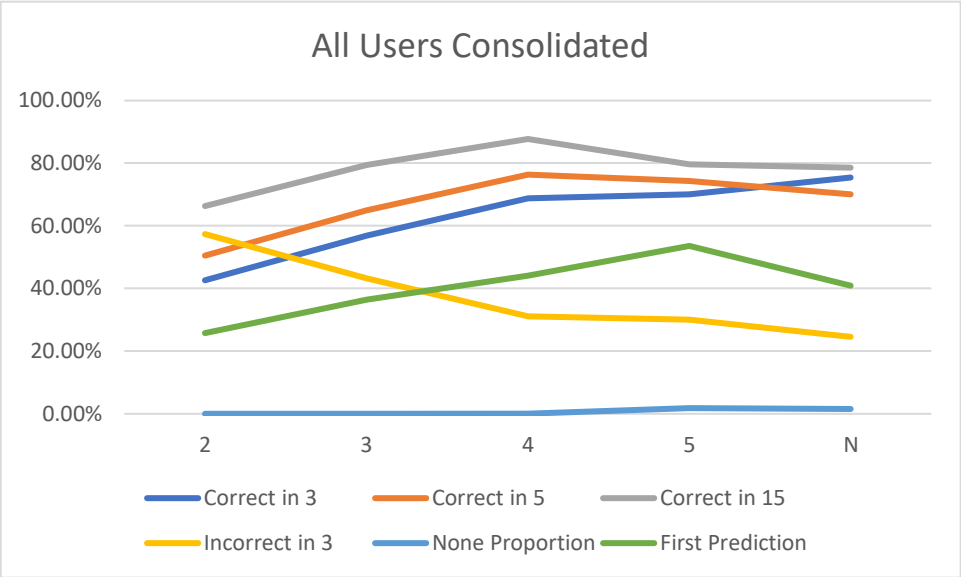
This section shows the data in Appendix A subsections 1 through 5 graphed per user. The X-axis indicates which command we are predicting, so 3 means we are predicting command 3 with commands 1 and 2, or predicting a command by looking at two previous commands.




Experienced Programmers
100.00%
80.00%
60.00%
40.00%
20.00%
0.00%
2
3
4
5
N
Correct in 3
Correct in 5
Correct in 15
Incorrect in 3
None Proportion
First Prediction


Non-programmers
100.00%
80.00%
60.00%
40.00%
20.00%
0.00%
2
3
4
5
N
Correct in 3
Correct in 5
Correct in 15
Incorrect in 3
None Proportion
First Prediction


Novice Programmers
100.00%
80.00%
60.00%
40.00%
20.00%
0.00%
2
3
4
5
N
Correct in 3
Correct in 5
Correct in 15
Incorrect in 3
None Proportion
First Prediction



B. Predicting Commands with 10% Sample of Parsed Data

1. Predicting the 2nd command from the 1st

	Computer Scientists	Experienced Programmers	Non-programmers	Novice Programmers	All Programmers
Correct in 3	33.64%	37.54%	41.00%	59.63%	42.76%
Correct in 5	41.20%	46.13%	47.60%	67.78%	50.79%
Correct in 15	58.49%	62.18%	63.93%	79.82%	66.19%
Incorrect in 3	66.14%	62.26%	58.85%	40.23%	57.15%
None Proportion	0.33%	0.33%	0.26%	0.36%	0.15%
First Prediction	19.48%	20.34%	22.43%	39.91%	25.33%
Avg Success Freq.	0.20	0.35	1.12	0.40	1.05
Avg Fail Freq.	0.18	0.32	1.04	0.35	0.90

2. Predicting the 3rd command from the 1st through 2nd

	Computer Scientists	Experienced Programmers	Non-programmers	Novice Programmers	All Programmers
Correct in 3	46.93%	51.46%	50.49%	69.00%	58.81%
Correct in 5	54.08%	59.03%	59.10%	75.55%	67.27%
Correct in 15	68.66%	71.65%	72.74%	85.70%	81.19%
Incorrect in 3	53.07%	48.54%	59.51%	31.00%	41.19%
None Proportion	1.94%	2.46%	1.06%	1.37%	0.63%
First Prediction	29.30%	31.18%	30.70%	47.04%	38.92%
Avg Success Freq.	0.26	0.43	1.35	0.45	1.14
Avg Fail Freq.	0.23	0.38	1.20	0.38	0.98

3. Predicting the 4th command from the 1st through 3rd

	Computer Scientists	Experienced Programmers	Non-programmers	Novice Programmers	All Programmers
Correct in 3	52.17%	54.22%	54.69%	69.75%	72.28%
Correct in 5	57.08%	59.90%	61.14%	75.10%	79.54%
Correct in 15	64.73%	66.99%	71.99%	83.04%	87.46%
Incorrect in 3	42.30%	40.77%	42.16%	28.61%	27.01%
None Proportion	11.55%	10.94%	6.96%	5.44%	2.57%
First Prediction	34.00%	33.95%	31.51%	33.84%	50.80%
Avg Success Freq.	0.28	0.46	1.44	0.46	1.17
Avg Fail Freq.	0.24	0.40	1.26	0.40	1.01

4. Predicting the 5th command from the 1st through 4th

	Computer Scientists	Experienced Programmers	Non-programmers	Novice Programmers	All Programmers
Correct in 3	47.11%	51.53%	48.53%	66.21%	84.82%
Correct in 5	50.44%	54.34%	54.04%	68.53%	86.99%
Correct in 15	52.82%	56.38%	58.46%	71.07%	89.16%
Incorrect in 3	35.73%	33.57%	40.00%	28.47%	14.06%
None Proportion	35.73%	30.74%	22.29%	15.75%	7.37%
First Prediction	32.46%	29.77%	28.29%	48.36%	66.52%
Avg Success Freq.	0.28	0.47	1.47	0.47	1.18
Avg Fail Freq.	0.25	0.41	1.28	0.40	1.02

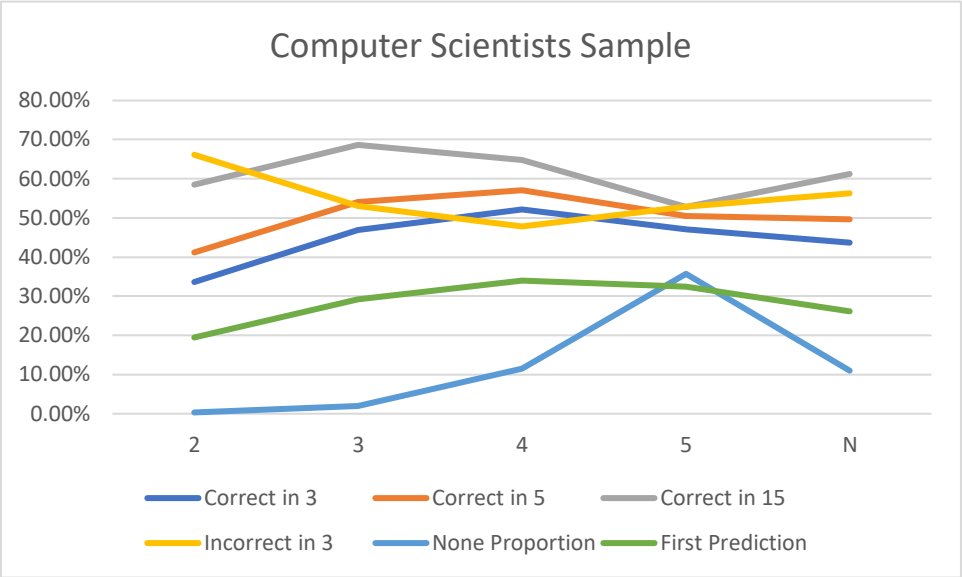
5. Predicting the Nth command from the 1st through N-1th

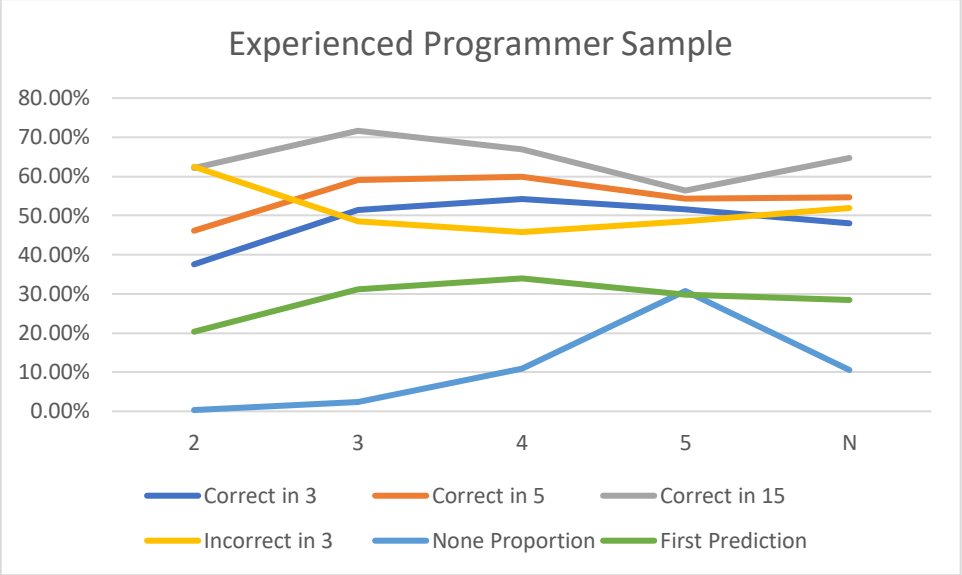
These results are from predicting the last command using all but the last command from all the command subsets formed through the variable sized sliding window parser.

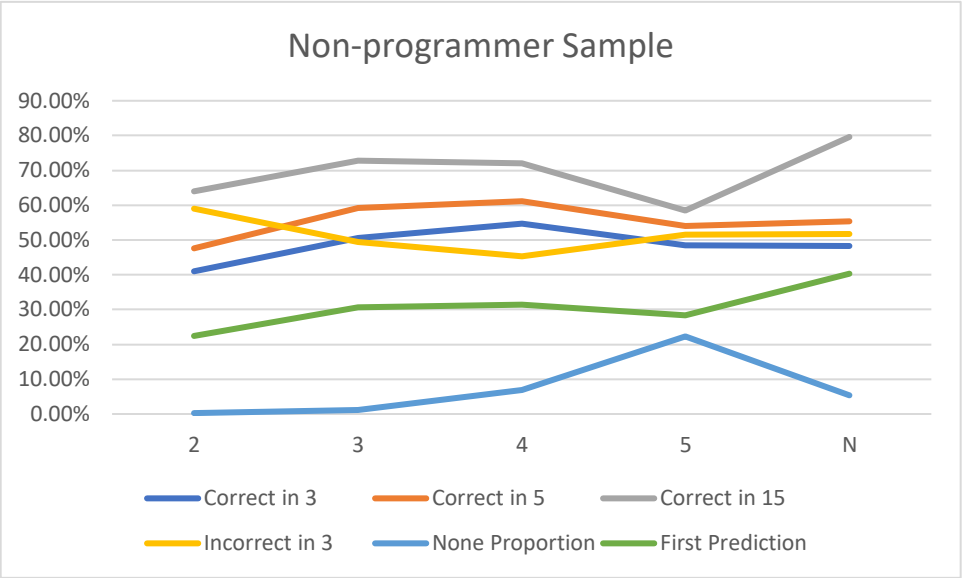
	Computer Scientists	Experienced Programmers	Non-programmers	Novice Programmers	All Programmers
Correct in 3	43.73%	47.99%	48.32%	65.33%	64.18%
Correct in 5	49.59%	54.70%	55.38%	70.84%	70.69%
Correct in 15	61.20%	64.64%	79.60%	79.60%	81.72%
Incorrect in 3	50.10%	46.53%	32.80%	32.80%	34.83%
None Proportion	10.95%	10.53%	5.38%	5.38%	2.76%
First Prediction	26.18%	28.36%	40.31%	40.31%	45.61%
Avg Success Freq.	0.29	0.50	0.52	0.52	1.34
Avg Fail Freq.	0.26	0.45	0.46	0.46	1.18

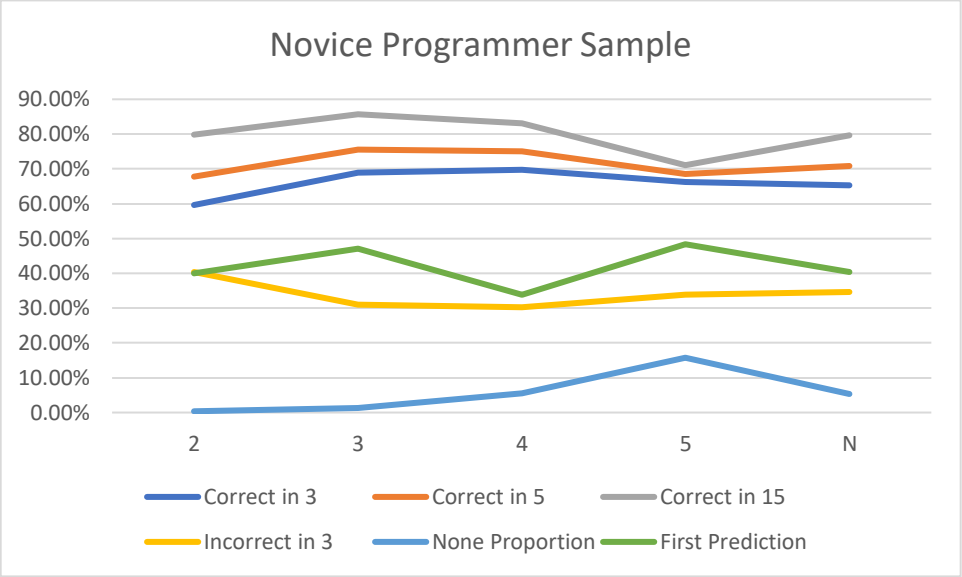
6. Data Graphed by User

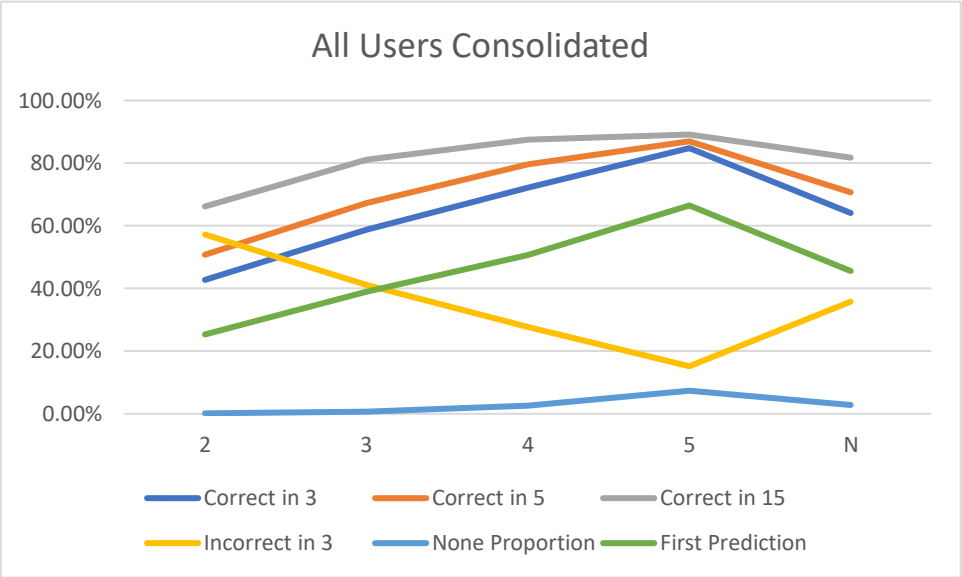
This section shows the data in Appendix B subsections 1 through 5 graphed per user. The X-axis indicates which command we are predicting, so 3 means we are predicting command 3 with commands 1 and 2, or predicting a command by looking at two previous commands.






Non-programmer Sample
90.00%
80.00%
70.00%
60.00%
50.00%
40.00%
30.00%
20.00%
10.00%
0.00%
2
3
4
5
N
Correct in 3
Correct in 5
Correct in 15
Incorrect in 3
None Proportion
First Prediction


Novice Programmer Sample
90.00%
80.00%
70.00%
60.00%
50.00%
40.00%
30.00%
20.00%
10.00%
0.00%
2
3
4
5
N
Correct in 3
Correct in 5
Correct in 15
Incorrect in 3
None Proportion
First Prediction

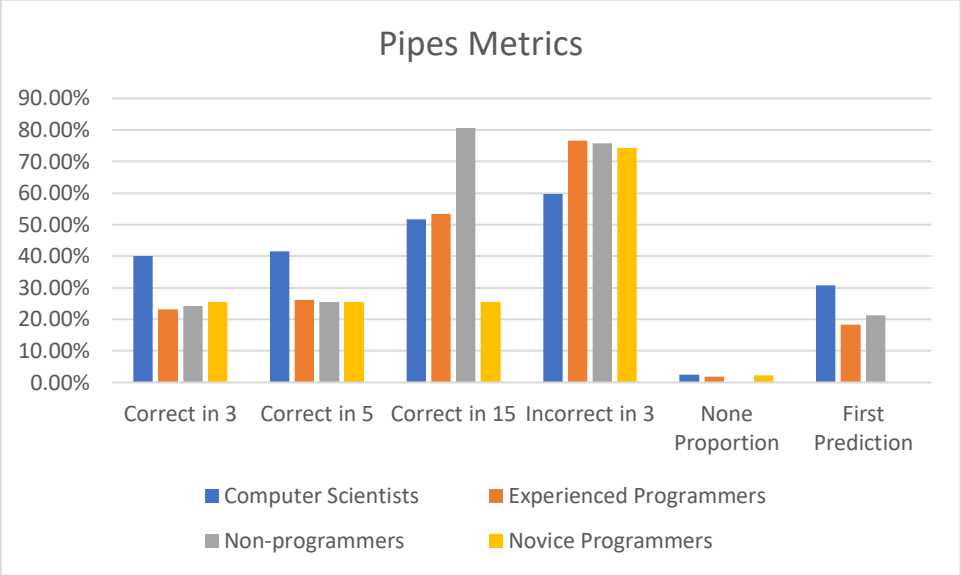

All Users Consolidated
100.00%
80.00%
60.00%
40.00%
20.00%
0.00%
2
3
4
5
N
Correct in 3
Correct in 5
Correct in 15
Incorrect in 3
None Proportion
First Prediction

C. Predicting Decomposed Pipes

1. Data in Table Form

	Computer Scientists	Experienced Programmers	Non-programmers	Novice Programmers
Correct in 3	40.17%	23.26%	24.17%	25.58%
Correct in 5	41.47%	26.05%	25.54%	25.58%
Correct in 15	51.62%	53.49%	80.70%	25.58%
Incorrect in 3	59.83%	76.74%	75.83%	74.42%
None Proportion	2.53%	1.83%	0.00%	2.27%
First Prediction	30.74%	18.26%	21.25%	0.00%
Avg Success Freq.	8.17	17.81%	7.54	89.91
Avg Fail Freq.	8.79	19.16%	8.10	96.66

2. Data Graphed by Metric and User



D. Co-learning

1. Predicting the 2nd command from the 1st

	Computer Scientists	Experienced Programmers	Non-programmers	Novice Programmers
Correct in 3	35.14%	28.65%	22.79%	40.66%
Correct in 5	40.94%	32.78%	33.61%	42.68%
Correct in 15	53.86%	43.98%	51.57%	44.14%
Incorrect in 3	64.45%	71.35%	77.21%	59.34%
None Proportion	0.63%	12.00%	0.76%	2.74%
First Prediction	20.50%	14.16%	15.62%	6.33%
Avg Success Freq.	0.28	0.54	1.87	0.54
Avg Fail Freq.	0.19	0.36	1.25	0.38

2. Predicting the 3rd command from the 1st through 2nd

	Computer Scientists	Experienced Programmers	Non-programmers	Novice Programmers
Correct in 3	33.89%	29.04%	25.75%	32.58%
Correct in 5	40.11%	34.75%	31.31%	35.07%
Correct in 15	49.05%	42.06%	44.46%	41.51%
Incorrect in 3	66.11%	70.96%	74.25%	67.42%
None Proportion	4.84%	31.58%	6.52%	19.90%
First Prediction	21.11%	13.17%	12.22%	13.73%
Avg Success Freq.	0.40	0.69	2.23	0.66
Avg Fail Freq.	0.28	0.48	1.58	0.47

3. Predicting the 4th command from the 1st through 3rd

	Computer Scientists	Experienced Programmers	Non-programmers	Novice Programmers
Correct in 3	29.74%	31.31%	25.66%	25.27%
Correct in 5	36.14%	35.25%	30.39%	44.75%
Correct in 15	43.64%	40.29%	40.18%	47.16%
Incorrect in 3	70.26%	68.69%	74.34%	74.73%
None Proportion	20.12%	56.52%	22.47%	55.61%
First Prediction	15.88%	8.23%	9.62%	8.69%
Avg Success Freq.	0.42	0.72	2.31	0.69
Avg Fail Freq.	0.31	0.52	1.69	0.51

4. Predicting the 5th command from the 1st through 4th

	Computer Scientists	Experienced Programmers	Non-programmers	Novice Programmers
Correct in 3	25.65%	30.93%	26.52%	16.73%
Correct in 5	32.30%	32.20%	30.82%	16.73%
Correct in 15	37.77%	36.02%	34.23%	19.77%
Incorrect in 3	74.35%	69.07%	73.48%	83.47%
None Proportion	41.42%	74.64%	42.06%	73.86%
First Prediction	9.51%	5.96%	7.48%	3.88%
Avg Success Freq.	0.43	0.73	2.34	0.71
Avg Fail Freq.	0.32	0.54	1.75	0.53

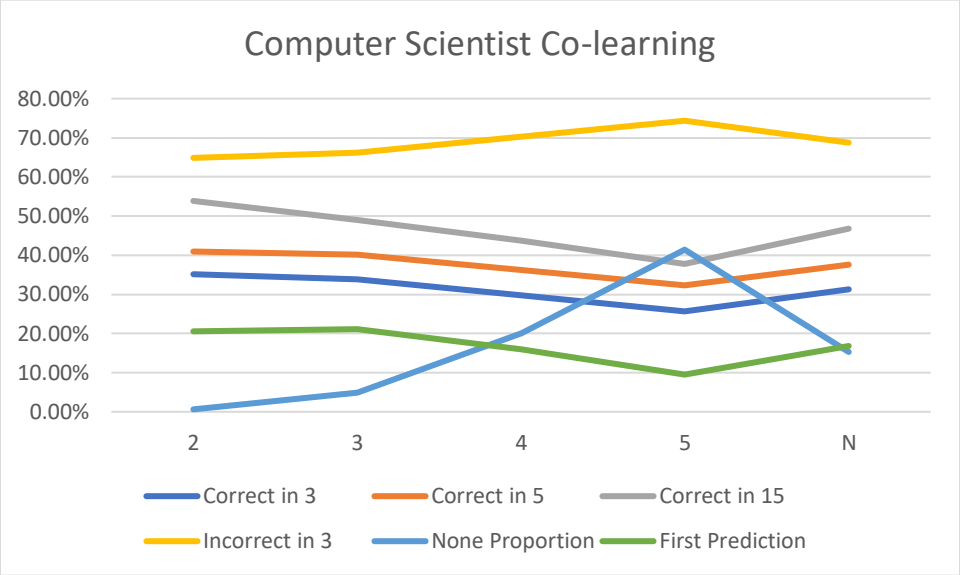
5. Predicting the Nth command from the 1st through N-1th

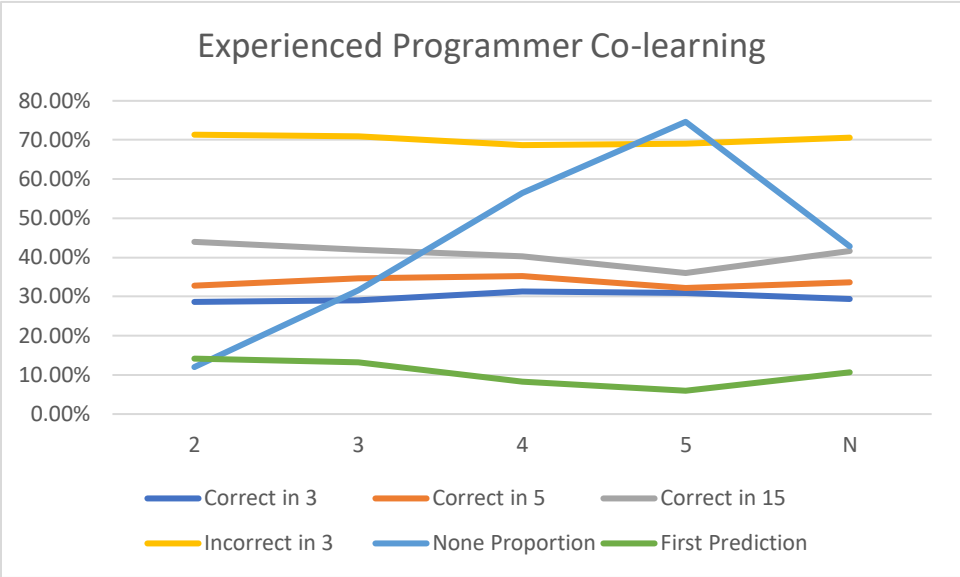
These results are from predicting the last command using all but the last command from all the command subsets formed through the variable sized sliding window parser.

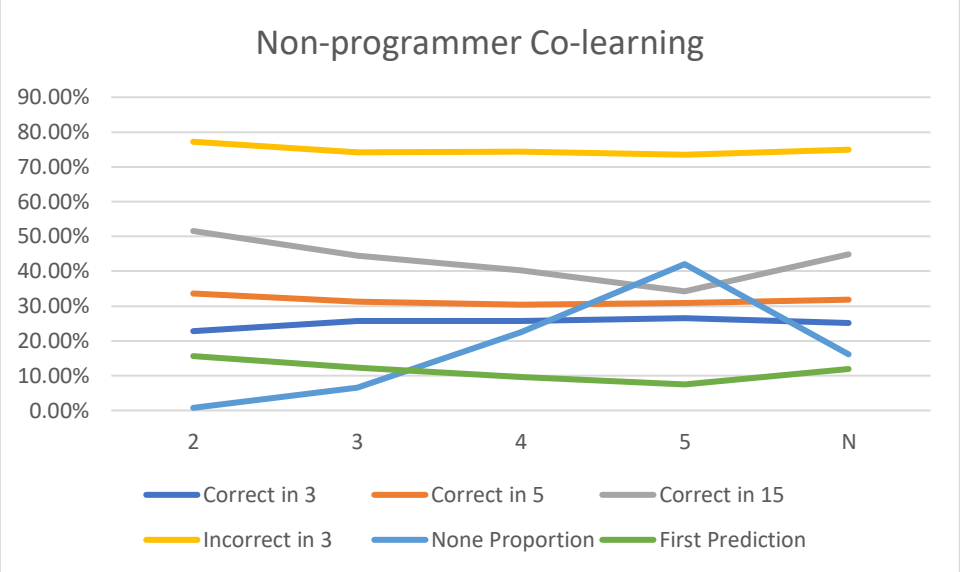
	Computer Scientists	Experienced Programmers	Non-programmers	Novice Programmers
Correct in 3	31.32%	29.35%	25.09%	32.98%
Correct in 5	37.54%	33.58%	31.85%	37.85%
Correct in 15	46.82%	41.66%	44.85%	46.96%
Incorrect in 3	68.68%	70.65%	74.91%	67.02%
None Proportion	15.36%	42.81%	16.16%	36.67%
First Prediction	16.79%	10.70%	11.82%	8.16%
Avg Success Freq.	0.45	0.79	2.62	0.78
Avg Fail Freq.	0.33	0.59	1.95	0.59

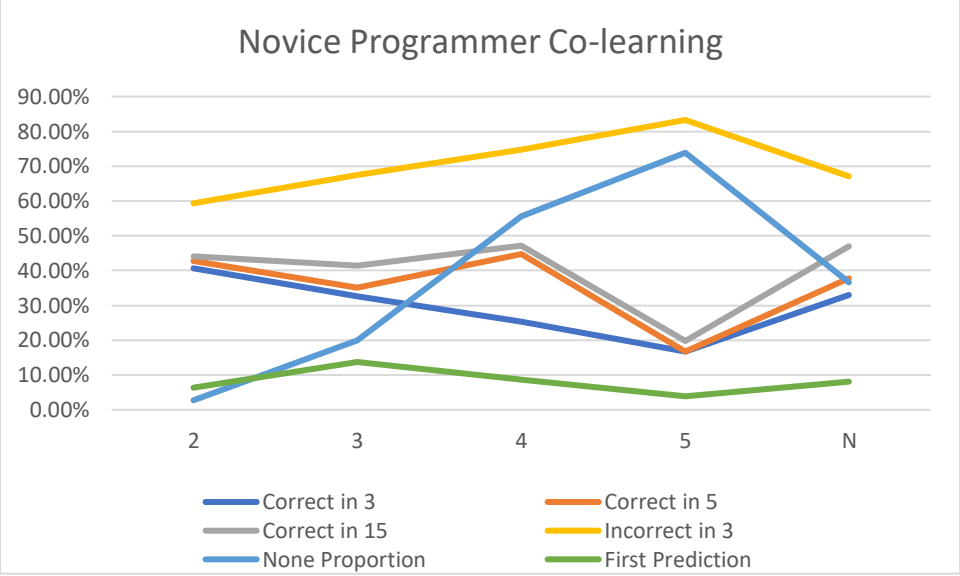
6. Data Graphed by User

This section shows the data in Appendix D subsections 1 through 5 graphed per user. The X-axis indicates which command we are predicting, so 3 means we are predicting command 3 with commands 1 and 2, or predicting a command by looking at two previous commands. We do not consolidate all users in this section as the goal is to see how well a single user's commands can be predicted without their own history.




Experienced Programmer Co-learning
80.00%
70.00%
60.00%
50.00%
40.00%
30.00%
20.00%
10.00%
0.00%
2
3
4
5
N
Correct in 3
Correct in 5
Correct in 15
Incorrect in 3
None Proportion
First Prediction


Non-programmer Co-learning
90.00%
80.00%
70.00%
60.00%
50.00%
40.00%
30.00%
20.00%
10.00%
0.00%
2
3
4
5
N
Correct in 3
Correct in 5
Correct in 15
Incorrect in 3
None Proportion
First Prediction


Novice Programmer Co-learning
90.00%
80.00%
70.00%
60.00%
50.00%
40.00%
30.00%
20.00%
10.00%
0.00%
2
3
4
5
N
Correct in 3
Correct in 5
Correct in 15
Incorrect in 3
None Proportion
First Prediction