Predicting Command Line Inputs using Divide-and-Conquer Synthesis

Allen Wang
Cornell University / 312 Highland Road
aw578@cornell.edu

Abstract

This paper presents an algorithm for predicting command line inputs by synthesizing programs from previous inputs and outputs. While previous attempts to do this are either unable to fully make use of context and/or require long-term observation to predict outputs, our algorithm can accurately make predictions from a single shell session and use information from outputs through careful feature selection. We detail the design and implementation of our algorithm, then profile its performance on a set of sample shell sessions.

1 Introduction

1.1 Motivation

Command-line interfaces have been a core part of computing for decades, but autocorrect functionality for shell commands is still underdeveloped. Currently, the Linux shell allows users to access their command history with the arrow keys and fill in file names with tab. This has the advantage of being simple and fast, but is very limited in its predictive power. Although shell usage is highly patterned, it is still slightly more complex than the arrow-key approach assumes. However, large language models are able to use context and understand command semantics to predict the next command. However, LLMs are either slow (if locally run) or expensive (if cloud-based). We propose a new approach to next-command prediction that uses system-guided synthesis to take advantage of patterns in shell usage. Our approach has the advantage of being more efficient than LLMs and more accurate than the arrow-key approach.

2 Related Work

Davison and Hirsh (1997) uses a decision tree predictor to predict command stubs by generating a decision tree from a log of command usage to predict the next command given the previous command. Korvemaker and Greiner (2000) uses a similar approach, but uses additional features like the error code, parsed templates, and the second-most-recent command. Both of these papers use long-term profiles of command usage, which are not present in our dataset. They also do not use the output of the command, which we hope will provide more information to the synthesizer. Lau et al. (2004) uses Programming by Example (PBE) to generate a model that can perform a technical support task. They record a set of examples of expert interactions with the system, then generate a Hidden Markov Model that can be used to predict the next action. However, their approach requires a large amount of examples of a single workflow, which we do not have. It also only learns probabilities over actions, as opposed to our method which learns a complete program.

3 Methodology

Our overall solution is to encode the next-command prediction problem as a system-guided synthesis (SyGuS) problem. We do this by writing a DSL for building shell commands from previous commands and their outputs, then encoding commands as input points (previous commands, previous outputs, next command) to restrain the semantics. After specifying the constraints, we enumerate all potential templates until we find one that can fit all entries seen so far in a shell session, then run it to predict the next shell session.

The first and largest problem with this naive approach is that it is too slow. As the number of entries in the shell session grows, the program size grows linearly, which results in an exponential increase in the time it takes to find a correct program through enumeration. To solve this, we take inspiration from the divide-and-conquer approach in Alur et al. (2017). This paper notes that although the full solution might be too large to feasibly enumerate, many problems can be decomposed into

two smaller problems: learning a set of terms that collectively cover every example, then learning a decision tree over the terms. We apply this idea to our problem by first generating a template for each command using enumeration, then generating predicates to select between templates. We then combine the templates and predicates into a single decision tree that can generate the next command using previous commands and outputs. By doing this, we can reduce the largest program size from 100 to 10, which is a significant improvement.

Our algorithm also leverages the specific structure of the problem for further optimizations. Firstly, we can easily enumerate all possible templates for a command by further breaking down the problem and generating a template for each individual word in the command. This lets us further reduce the program size, as we only need to enumerate templates for each word in the command instead of the entire command. After generating the templates, we can then use a greedy set cover algorithm to find a minimal covering set of expressions that can generate all the commands in the log when combined. This step improves robustness by rewarding general-purpose templates over templates that are specific to one command. When done correctly, it also biases the predicate search to learn more useful and generalizable predicates. When generating the predicates and decision tree, we also use the problem structure to our advantage. As most programs take the form of switch statements over the templates, we can independently generate predicates for each template instead of generating a single decision tree for the entire program.

In addition to algorithmic improvements, feature engineering was the most important part of this project. The space of possible literals / programs is very large, which makes it difficult to find a good program to predict the next command. We address this by manually identifying the most relevant literals, which we then pass into the synthesizer. For example, we only use the last command and output for prediction, as it is the most relevant part of the shell session. Doing this lets us reduce the program space significantly. In addition, we only use space-separated words as constants and word indexes as variables for term generation, as this is the most common pattern in shell usage. Finally, we also add special number handling by identifying numbers in the shell log, using them as constants, and building expressions out of them.

Unfortunately, reducing the program space by this much means that some programs are no longer learnable. For example, only using the last command + output for prediction means that some entries might be mutually contradictory. If "explorer.exe <filename>" is sometimes followed by "cat <filename>" and sometimes by "echo <filename>", then the synthesizer will not be able to learn this pattern. To address this, we currently discard all mutually contradictory entries except the most recent one, which results in a loss of information. We also tried using the second-most-recent command and output to distinguish between entries where the most recent command and output are the same. Due to performance limitations, we did not pursue this in the final implementation, but it should theoretically result in better accuracy. The programs our algorithm generates are often fragile, as they often fail to generalize or depend on matching specific artifacts of the input and output. We attempt to address this by selecting literals that are more likely to be useful for generalization, but our solution is far from perfect. Especially in cases where the current options are insufficient or there are few examples, the synthesizer will often generate hardcoded programs. For example, if a command is not space-separated, then the synthesizer will not be able to learn it. We tried to address this by adding a "word" type, but this was difficult to implement and resulted in a more complex algorithm. Finally, some entries are still inherently unpredictable, even with a perfect synthesizer. If the next command cannot be built from previous commands and outputs, then it is inherently unpredictable by program synthesis tools. We currently ignore these cases, but this is a promising area for using LLMs for prediction.

4 Algorithm Overview

We present our implementation of a system-guided synthesizer for predicting the next shell command based on previous commands and their outputs. Our algorithm encodes the next-command prediction problem as a program synthesis task using a domain-specific language (DSL) of shell command transformations.

4.1 Problem Formulation

Given a sequence of shell command executions represented as triples (c_i, o_i, c_{i+1}) where:

- c_i is the i-th command

- o_i is the output of command c_i
- c_{i+1} is the subsequent command

We seek to synthesize a program P such that $P(c_i, o_i) = c_{i+1}$ for all observed triples.

5 Domain-Specific Language

5.1 Types and Base Expressions

Our DSL supports three primary types:

- `str`: String expressions
- `int`: Integer expressions
- `bool`: Boolean expressions

5.1.1 String Operators

$$\text{Concatenate}(s_1, s_2) : \text{str} \times \text{str} \to \text{str}$$
$$\text{ConstantString}(\text{literal}) : \text{str}$$
$$\text{Substring}(s, i, j) : \text{str} \times \text{int} \times \text{int} \to \text{str}$$
$$\text{Word}(s, n) : \text{str} \times \text{int} \to \text{str}$$
$$\text{StringVariable}(\text{name}) : \text{str}$$
$$\text{ToString}(n) : \text{int} \to \text{str}$$

5.1.2 Integer Operators

$$\text{Plus}(x, y) : \text{int} \times \text{int} \to \text{int}$$
$$\text{Minus}(x, y) : \text{int} \times \text{int} \to \text{int}$$
$$\text{Number}(n) : \text{int}$$
$$\text{NumberVariable}(\text{name}) : \text{int}$$
$$\text{ToInt}(s) : \text{str} \to \text{int}$$

5.1.3 Boolean Operators

$$\text{And}(b_1, b_2) : \text{bool} \times \text{bool} \to \text{bool}$$
$$\text{Or}(b_1, b_2) : \text{bool} \times \text{bool} \to \text{bool}$$
$$\text{Not}(b) : \text{bool} \to \text{bool}$$
$$\text{StringEquals}(s_1, s_2) : \text{str} \times \text{str} \to \text{bool}$$
$$\text{Contains}(s_1, s_2) : \text{str} \times \text{str} \to \text{bool}$$
$$\text{LessThan}(x, y) : \text{int} \times \text{int} \to \text{bool}$$
$$\text{If}(b, s_1, s_2) : \text{bool} \times \text{str} \times \text{str} \to \text{str}$$

6 Term Generation

6.1 Word-Level Expression Generation

For each word w in the target command c_{i+1}, we generate alternative expressions based on the context (c_i, o_i):

Algorithm 1 Generate Alternate String Expressions

function GENERATEALTERNATEEXPRESSIONS(w, c_i, o_i)
  $E \leftarrow \{\text{ConstantString}(w)\}$
  input_words $\leftarrow$ split(c_i)
  output_words $\leftarrow$ split(o_i)
  for $j \leftarrow 0$ to $|\text{input\_words}| - 1$ **do**
    if $w \in \text{input\_words}[j]$ **then**
      $E \leftarrow E \cup \{\text{Word}(\text{cmd}, j)\}$
      $E \leftarrow E \cup \{\text{Word}(\text{cmd}, j - |\text{input\_words}|)\}$
    end if
  end for
  for $j \leftarrow 0$ to $|\text{output\_words}| - 1$ **do**
    if $w \in \text{output\_words}[j]$ **then**
      $E \leftarrow E \cup \{\text{Word}(\text{output}, j)\}$
      $E \leftarrow E \cup \{\text{Word}(\text{output}, j - |\text{output\_words}|)\}$
    end if
  end for
  if w is numeric **then**
    $E \leftarrow E \cup \text{SynthesizeNumericExpression}(w, c_i, o_i)$
  end if
  return E
end function

Algorithm 2 Synthesize Numeric Expression

function SYNTHESIZENUMERICEXPRESSION(w, c_i, o_i)
  target $\leftarrow$ int(w)
  literals $\leftarrow \{\text{Number}(1)\}$
  for each numeric word n in c_i and o_i **do**
    literals $\leftarrow$ literals $\cup \{\text{ToInt}(\text{Word}(\cdot, \text{index}(n)))\}$
  end for
  operators $\leftarrow \{\text{Plus}, \text{Minus}\}$
  expr $\leftarrow \text{BottomUp}(5, \text{operators}, \text{literals}, [(\{c_i, o_i\}, \text{target})])$
  if expr $\neq$ null **then**
    return $\{\text{ToString}(\text{expr})\}$
  else
    return $\{\text{ConstantString}(w)\}$
  end if
end function

6.2 Numeric Expression Synthesis

For numeric words, we employ bottom-up synthesis to generate arithmetic expressions:

7 Optimized Term Merging

To reduce the exponential complexity of enumerating all possible term combinations, we employ an optimized merging strategy:

Algorithm 3 Optimized Term Merging

```
function OPTIMIZEDMERGETERMS(triples)
    max_words ← max_i |split(target_i)|
    word_coverage ← []
    for p ← 0 to max_words − 1 do
        coverage ← {}
        for each triple t_i do
            for each expression e for position p in t_i do
                if e produces correct word at position p for t_i then
                    coverage[e] ← coverage[e] ∪ {i}
                end if
            end for
        end for
        word_coverage.append(coverage)
    end for
    expressions ← GenerateExpressionsCombinations(word_coverage)
    return GreedySetCover(expressions)
end function
```

8 Predicate Synthesis

8.1 Literal Generation

For predicate synthesis, we generate Boolean literals based on the command and output context:

$$\begin{aligned}\text{Literals} = &\{\text{Contains}(\text{command}, w) \mid w \in \text{words}(c_i)\} \\ &\cup \{\text{Contains}(\text{output}, w) \mid w \in \text{words}(o_i)\} \\ &\cup \{\text{Word}(\text{command}, j) \mid j \in [-|c_i|, |c_i|]\} \\ &\cup \{\text{Word}(\text{output}, j) \mid j \in [-|o_i|, |o_i|]\}\end{aligned}$$

8.2 Bottom-Up Predicate Enumeration

We use bottom-up synthesis to find predicates that distinguish when each term should be applied:

Algorithm 4 Enumerate Predicate for Term

```
function ENUMERATEPREDICATEFORTERM(term, triples, covered_indices, literals)
    environments ← [{command: c_i,output: o_i} | (c_i, o_i, c_{i+1}) ∈ triples]
    target_vector ← [i ∈ covered_indices | i ∈ [0, |triples|)]
    input_outputs ← [(env_i, target_vector_i) | i ∈ [0, |triples|)]
    operators ← {If, Or, StringEquals, And, Not}
    return BottomUp(max_depth, operators, literals, input_outputs)
end function
```

9 Decision Tree Construction

Finally, we combine predicate-term pairs into a nested conditional expression:

Algorithm 5 Combine Predicate-Term Pairs

```
function COMBINEPREDICATETERMPAIRS(pairs)
    if |pairs| = 0 then
        return null
    end if
    result ← pairs[−1].term
    for i ← |pairs| − 2 down to 0 do
        result ← If(pairs[i].predicate, pairs[i].term, result)
    end for
    return result
end function
```

10 Complexity Optimizations

10.1 Cost Reduction for Numeric Expressions

We apply cost reduction to numeric expressions that satisfy certain criteria:

- Multiple different substring sources, OR
- At least one substring source AND at least one number literal

This encourages the synthesis of more general arithmetic expressions over hardcoded constants.

10.2 Priority-Based Enumeration

Expressions are enumerated in order of:

1. Expression size (smaller first)

2. Priority score (higher priority first)

3. Lexicographic ordering (for deterministic tiebreaking)

11 Algorithm Complexity

The overall time complexity is dominated by:

- Term generation: $O(|W|^k \cdot |E|^k)$ where $|W|$ is the vocabulary size, k is the maximum command length, and $|E|$ is the number of alternative expressions per word
- Bottom-up synthesis: $O(|O|^d \cdot |L|^d)$ where $|O|$ is the number of operators, d is the maximum depth, and $|L|$ is the number of literals
- Set cover: $O(|T|^2 \cdot |S|)$ where $|T|$ is the number of terms and $|S|$ is the number of examples

The optimized merging strategy reduces the effective search space by calculating coverage at the word level and using set operations instead of enumerating all combinations.

12 Benchmarking

12.1 Dataset

Following the approach in previous work (Davidson et al., 1997), we captured command histories from 5 friends to serve as our test dataset. However, beyond just recording the commands, we also recorded the output of the commands, which we hoped would provide more information to the synthesizer. Unfortunately, we ran into significant problems when trying to use the recorded outputs. Due to coding errors, the logfiles could not be played back, although some information was retained. Beyond this, the logfiles were not suited for the synthesis task, as they did not have any discernible logic to them. In many cases, decisions were based on information that was not recorded, such as visual output or the current open project. Instead, we decided to use a combination of manually written and LLM-generated shell logs. We wrote 3 sample shell logs manually that reflected some common patterns and used a large language model to generate 15 more logs based on the logfiles. After generating the logs, we manually verified that they were representative before using them for evaluation.

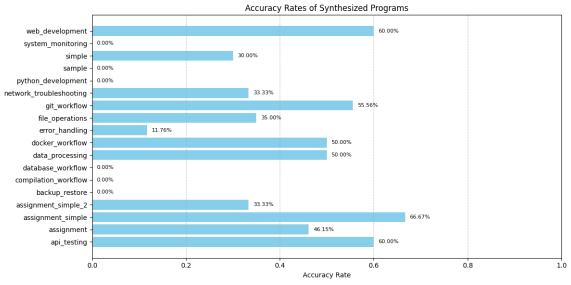
12.2 Evaluation

To evaluate our algorithm, we used the shell logs to simulate the execution of commands. After executing each command, we generated a program and used it to predict the output of the next command. To evaluate the performance of our algorithm, we compare the predicted output with the actual output of the next command. We have recorded the percentage of all predicted commands as well as the percentage of commands that were possible to predict (which we estimate by using a heuristic). In our implementation, we assume that predictable commands occur after the first 2 and are made up of numbers, previous inputs, and previous outputs. In other words, predictable commands are the commands that could be found using the current algorithm if it were given all substrings of all previous commands and outputs.

12.3 Baseline Performance

We take inspiration from Korvemaker and Greiner (2000) to estimate the baseline performance of our algorithm. In their work, they predict full commands from a combination of base frequencies, frequencies conditioned on the previous command, and parsed command templates. They found that their algorithm was able to predict around 48% of the commands and 48% / 72% = 66.6% of the predictable commands. However, they used longer-term profiles of more than 1700 commands per user. In addition, their definition of predictable commands is less strict than ours, as in their case, a command is unpredictable if it has never been seen before, while we include commands that can be generated from previous commands and outputs as well. As a result, we expect our performance to be lower than theirs.

13 Results



Our algorithm was able to predict 39/128 = 30.47% of predictable commands and 39/362 = 10.77% of all commands on average. Although this is somewhat low, we think that this is understand-

able, given that the algorithm has access to only 1 to 3% of the information of the baseline implementation and is still poorly optimized. In addition, our dataset clearly has fewer predictable commands as a percentage of total commands, which artificially lowers the performance across all commands. Regardless, we think that this is a promising approach, especially as a single expert in a mixture-of-experts model.

14 Next Steps

One of the largest improvements we could make would be better feature engineering. Right now, we use a pretty unoptimized subset of the possible program space, and finding better ways to spend our compute budget would result in much better performance. One promising improvement would be to find more relevant literals. Right now, we assume that a substring is relevant iff it is a number or a space-separated word in the previous command or output. We could improve this by identifying relevant non-space-separated words or removing irrelevant words from the predicate and term generation. Performance is also very poor right now, so making the algorithm faster would also be helpful, both by reducing the amount of time spent generating programs and by allowing more literals and operators to be used. Finally, one idea we considered but did not pursue was using embedding models to cluster similar input-output-input tuples. After clustering, we could generate one program per cluster and match the current command to the closest cluster, which would make it easier to generate predicates within clusters. As predicate generation is the most time-consuming part of the algorithm, this would result in a significant speedup.

References

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling enumerative program synthesis via divide and conquer. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 319–336, Berlin, Heidelberg. Springer Berlin Heidelberg.

Brian D. Davison and Haym Hirsh. 1997. Toward an adaptive command line interface. In *Interacción*.

Benjamin Korvemaker and Russell Greiner. 2000. Predicting unix command lines: Adjusting to user patterns. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, page 230–235. AAAI Press.

Tessa Lau, Lawrence Bergman, Vittorio Castelli, and Daniel Oblinger. 2004. Sheepdog: learning procedures for technical support. In *Proceedings of the 9th International Conference on Intelligent User Interfaces*, IUI '04, page 109–116, New York, NY, USA. Association for Computing Machinery.