

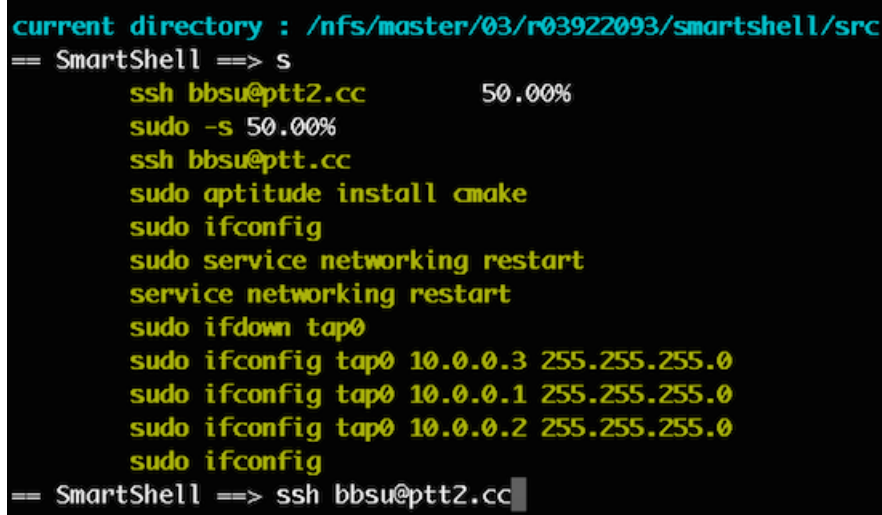
Artificial Intelligence Final Project Report

Group 29 SmartShell

b99902054 資工四 薛祐婷, b99902108 資工四 李昀儒

r03922093 資工碩一 王友伶, r03944012 網媒碩一 陳彥儒

Introduction



```
current directory : /nfs/master/03/r03922093/smartshell/src
== SmartShell ==> s
    ssh bbsu@ptt2.cc          50.00%
    sudo -s 50.00%
    ssh bbsu@ptt2.cc
    sudo aptitude install cmake
    sudo ifconfig
    sudo service networking restart
    service networking restart
    sudo ifdown tap0
    sudo ifconfig tap0 10.0.0.3 255.255.255.0
    sudo ifconfig tap0 10.0.0.1 255.255.255.0
    sudo ifconfig tap0 10.0.0.2 255.255.255.0
    sudo ifconfig
== SmartShell ==> ssh bbsu@ptt2.cc
```

While working with Bash, it is tedious for most of the users to type repeated commands. Even with the Bash build-in auto-completion feature, sufficient characters must be provided to solve ambiguity of prefix. If the commands can be auto-completed with just a few number of characters typed, it saves a lot of time. Although around 72%, according to the past research [1], of the commands can be predicted from history file, it still work especially when users are coding or doing another work which will type repeated commands in a period. The common solution for this condition is reverse search, which only find the latest commands of whose prefix corresponds to what the user just type. However, it still left something to be desired. Therefore, we introduce a better shell which can preform customized auto-completion with information extracted from user history.

Problem definition

What we want to do is to introduce a "smarter" auto-completion shell based on command history. Our goal is to predict the most probable command that the user want to type next. We compare our result with that of Bash built-in tab completion and reverse search. The following is the PEAS of SmartShell custom auto-completion.

- **Performance Measure:** accuracy of the prediciton list, the speed of prediction
- **Environment:** the user and the shell
- **Actuators:** screen (stdout)
- **Sensors:** keyboard (stdin)

Besides, since our implementation is a improved shell, we regard usability as an important metric. Our ultimate goal is to work with shell easily.

Proposed methods

We proposed a probabilistic model to perform command prediction. The flow of our method is as follow.

The training data is from user history file. We maintain a window of N commands. In the window we compute probabilities for each commands right after another commands. For examples, if the training data is "ABCBCD", it implies that C probably appears right after B. So if the previous command is B, we can predict that the next one is C. We set the window size because the older history has less effect on the current work. We compute probabilities with how many times it appears. In the example "ABCBCD", the probability "C right after B" is $2/2 = 100\%$, "B right after A" and "B right after C" are both $1/2 = 50\%$. We maintain separated set of counts for "command only" and "command with arguments". Not only right previous one, we take previous 3 commands with different coefficient representing different weight to improve our result. In the boundary case, the coefficient set to 0 can avoid another commands' effects. The system generates a command candidate list by prefix-match of the user input and all the commands ever entered by user. The candidate list is sorted by the conditional probability described above. One point we do better than reverse search is that we can predict without any input base on previous commands. Once the users want to type another different with our prediction, then we change our result from the input prefix. The window then moves as new commands add to the history and compute another probability table for another auto-completion.

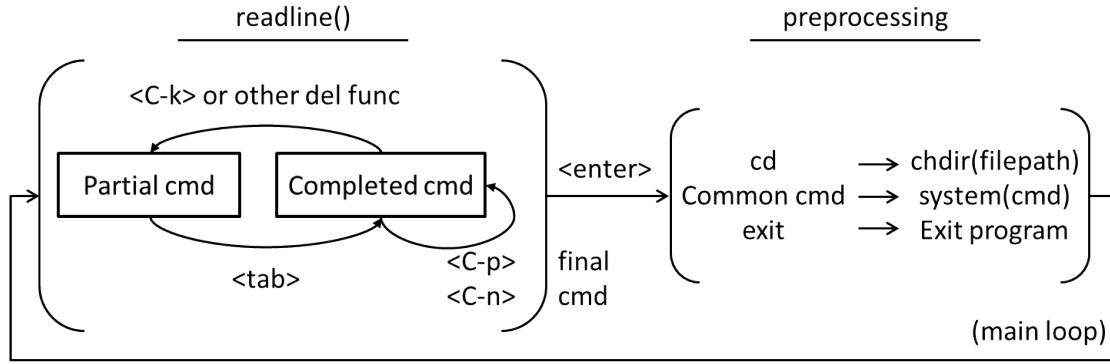
According to research before[1], we adopt this probability model due to the high accuracy. It says that more previous commands leads less accuracy, which are 47.4% and 36.9%. However, we think another previous commands may help to predict. Therefore, we take it's opinion and fix it. We take some previous commands and set different coefficient to them. In the worst case, the accuracy will be 47.4%, while previous one owns all weight. It guarantees at least accuracy and can be further improved. That paper is mentioned that it takes some factors such as times in one day or one week, we think user may work in different time, but follow his/her own habits. That's why we compute probabilities from users' history file. We hope our auto-completion can maintain high accuracy for every different user.

Implementation

The sliding window stated above is implemented in a C++ class HistoryWindow. The class provide methods to update window content, query command counts and generate candidate list. When a new command is entered, the window moves. The oldest command leaves with its count decremented, and the newest command comes with its count incremented. The dynamic update of command counts prevented re-calculation of all probabilities every time the window moves and make the prediction real-time.

To let bash apply our custom completion predict list, we implement our completion function for bash, by api of the GNU Readline Library. GNU Readline Library provides api of reading a line from input (just like shell prompt). We can bind keys for different functions like deleting a word, moving cursor to the head of the line or getting some possible command based on the partial command text in current line prompt. Furthermore, this library provided some mechanism to let user implement custom functions and bind the key to specific function that the user wants. By the library, we implement our completion function, where the list of the possible command completion is from our HistoryWindow class, which learns from the bash command history and uses the partial command text to predict and then generate the possible list.

The figure below illustrate the flow of our program. In the left side of this figure, there are arrows with keys, which are the binded with the functions we implement. When tab is pressed, our program as HistoryWindow to provide sorted command candidate list, and complete the input line of the first candidate. Ctrl-n and ctrl-p not are to change the current completion command to the next/previous candidate. Then, the completion our implemented are all with arguments, so we implement the new functionality of delete all the line except the "command", which is Ctrl-k. Since we simply use system() to execute the command, we have to process the path of working directory changes on our own. That's what the right side of this figure presents.

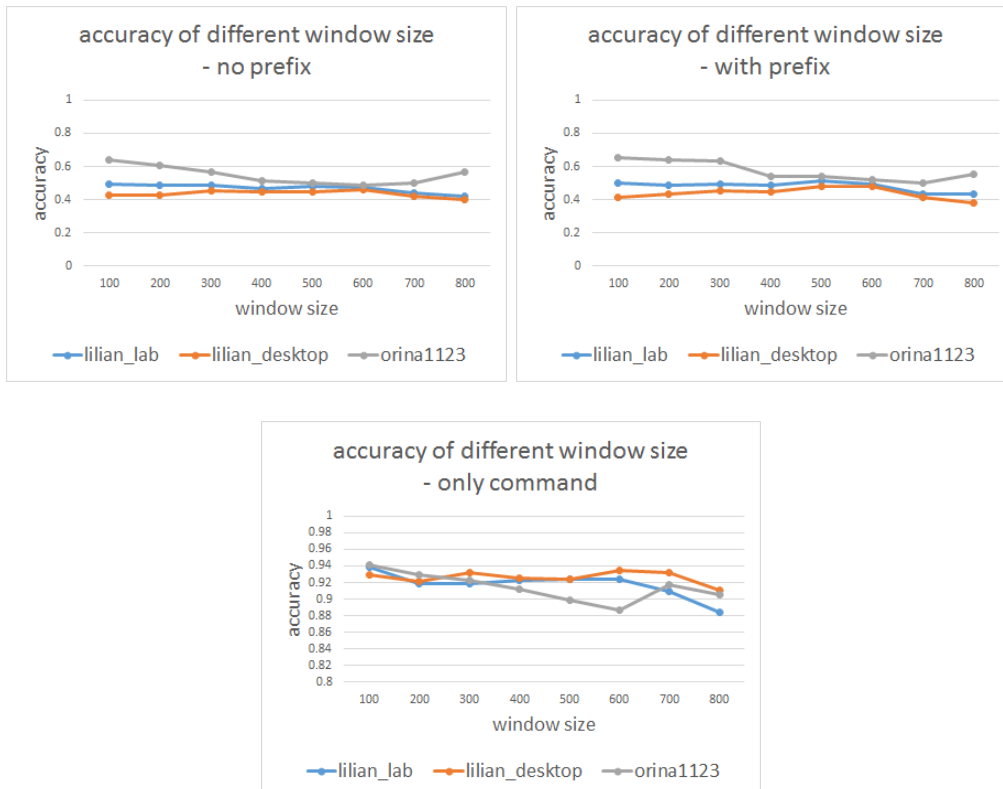


Result

To evaluate the prediction list from our model, we use a history file with about 1500 2000 commands, fill the partial text of the next command, and then perform our probabilistic model and reverse search to get 2 prediction commands. We compare 3 type of results. One is the accuracy on no input partial command text, another is the accuracy on half partial command text, and the other is the accuracy of reverse search on half partial command text. The following table show the result of window size $N = 500$.

	Prob. model (with no input)	Prob. model (with half cmd input)	Reverse search (with half cmd input)
orina_nlp_history	0.503055	0.543061	0.529543
lilian_desktop_history	0.444706	0.483721	0.442353
lilian_lab_history	0.483529	0.517015	0.497647

The below line graphs illustrate the effect of different window size N . The first two graph is for prediction with arguments, and the third one is for prediction of the command only (without argument).



Results references

After experiments, we found that our method gets almost the same or worse performance than reverse search. That is because when our method cannot find any possible prediction, it returns nothing. On the other hand, reverse search can always find a prediction. So we combine our method and reverse search only when our method cannot find any command with probability greater than 0, we adopt reverse search as our method. To our surprise, it gets better performance than both methods, especially only take command into account. In the research before[1], it also suggests combination leads to better result. Due to different user habits for everyone, combination of methods can find the best prediction which is corresponding to user's habit with less bias.

Future work

We adopt a sliding window to lower the effect of older commands, but the point of time of each command should also serves as a good clue of user work period and should be take into consideration. In addition, it will be more attractive that we can predict command arguments that have never appeared. For example, we would like to predict the next command of "vim test.c" to be "gcc test.c" even if it is the first time the user compile the source.

Job responsibility of members

- 薛祐婷 - model design, model implementation, main program implementation
- 李昀儒 - model design, model implementation, main program implementation
- 王友伶 - model design, GNU readline library survey, main program implementaion
- 陳彥儒 - model design, model implementation, papers about shell completion survey

Collaboration/communication mechanism

Each member has complete freedom of deciding the time of finishing the task assigned. We basically discuss about our project in Facebook message, and the one who has time will take the tasks of that time to do. We had some meetings in CSIE Building before the due date of milestones. Whenever someone finish one functionality, the others will run the program and check whether there are bugs in short time. Besides, we devided the task clearly in the first, so that we had less conflict when merging codes.

References

1. Korvemaker, Benjamin, and Russell Greiner. "Predicting Unix command lines: adjusting to user patterns." AAAI/IAAI. 2000.
2. Durant, Kathleen T., and Michael D. Smith. "Predicting unix commands using decision tables and decision trees." Proceedings of the Third International Conference on Data Mining. 2004.

Source Code

<http://github.com/orina1123/smartshell>