Predicting Developer Commands

Tyson Bulmer
Computer Science
University of Victoria
Student Number: V00799439
Email: tysonbul@uvic.ca

***Abstract*—When a developer is writing code they are usually focused and in a state-of-mind which some refer to as flow. Breaking out of this flow can cause the developer to lose their train of thought and have to start their thought process from the beginning. This loss of thought can be caused by interruptions and sometimes slow IDE interactions. Predictive functionality has been harnessed in user applications to speed up load times, such as in Google Chromes browser which has a feature called "Predicting Network Actions". This will pre-load web-pages that the user is most likely to click through. This mitigates the interruption that load times can introduce. In this paper we seek to make the first step towards predicting user commands in the IDE. Using the MSR 2018 Challenge Data of over 3000 developer session and over 10 million recorded events, we cleanse and analyze the data to be parsed into N-Grams, which we then use to train a Multinomial Naive Bayes model to predict user induced commands. Our model is able to obtain a prediction accuracy of 70%.**

***Keywords*—IDE, command prediction, machine learning**

I. Background and Introduction

The task of developing software is a highly thought intensive process and interruptions can derail a train of thought easily. These interruptions can come in a variety of ways, one being a slow load time. Many bug prediction paper have touched on the notion of developer "focus" as an contributor to code bug proneness. Di Nuci et al tried to build on this notion that the higher a developer's focus on their activities, the less likely they are to introduce bugs to the code [1]. Zhang et al also insisted that certain file editing patterns a developer follows when writing code could attribute to bug proneness as well. Two of their patterns which highlight the "focus" of a developer is the "Interrupted" and "Extended" editing patterns [3], which they find introduces 2.28 and 2.1 times more future bugs than without.

Even an interruption such as a slow execution of a command can cause a disruption to a developer's focus and we believe that if we can predict user commands, then we can prevent slow load times of actions, just as found in applications such as Google Chrome the web browser which has predictive page loading to speed up browsing. In this paper we use a corpus of over 10 million IDE events and commands to see if we can train a model to accurately predict developer commands. We formally address our intent with the following research question:

- ***RQ1**: Can we use machine learning models to predict developer induced IDE commands?*

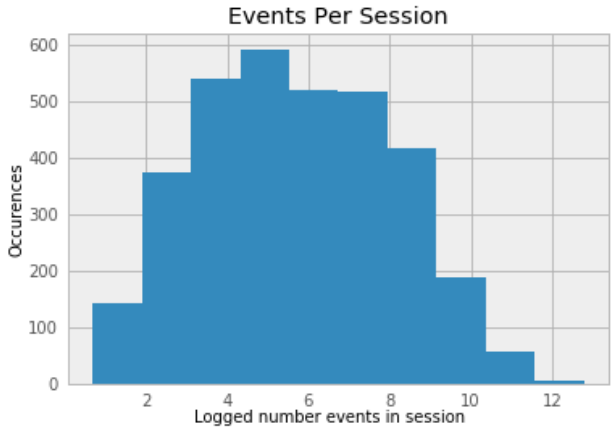

Figure 1: Logged frequency of events in sessions

II. Data

The dataset we use is supplied by [2], for the 2018 MSR Challenge paper. Released in March 2017, this data contains over 10 million events, originating from a diverse group of 81 developers and covers a total of 1,527 aggregated days of developer work. In this section we analyze the events recorded by the IDE tool, and then explain the process we took to prepare the data to be used for our predictive model.

A. Data Analysis

Since we seek to harness the pattern of commands in N-Gram models we needed to know the distribution of events per session and, similarly, the distribution of events types. The data contains 16 unique event types, we are mostly interested in the "Command" type events, which are user activated programmatic actions in the IDE, for example when a user clicks a button in a menu. Some other event types, for example, are "Window" and "Find", where the former refers to actions involving the IDE window, and the latter referring to when the user performs a file search. A complete listing and descriptions of these events can be found on the data suppliers website: http://www.kave.cc/feedbag/event-generation.

To get a better idea of the distribution of events per session that we are working with, we plot the logged distribution as shown in Figure 1. From it we can see that there is an almost normal, when logged, distribution of events per session.

Figure 2 shows the logged distribution of event types across all sessions. From this we can see that the "Command" event

occurs the most. Note that the original dataset contained more than 16 event types, but for our purpose we only needed the user induced event types, as described in the Data Preparation section of this report.

B. Data Preparation

In this section we describe the steps taken to filter, cleanse, and format the data in a usable way. We start out with 10,732,641 recorded events.

1) Filtering: While the online description of the dataset documents 20 unique event types, with 2 being deprecated, our data only contained 16 unique events. Of these, 13 were user induced events. The non user-induced events which were filtered out were the "System", "Info", and "Error" events. We filter these out because we only want events in which the user can induce, as these are part of the developers working pattern. After filtering these events we still had over 10 million recorded events to work with.

2) Cleansing: When looking at the data we also noticed that some events had been recorded twice with the same triggered-at date, event type and session ID. So we filter out all the duplicate entries and remain with 10,043,171 entries.

3) Preparation: Most of the event types have additional information about the action which can help identify it. For example, the Window Event also records an "action" value which represents whether the action was to move, open, or close the window. This is important data for helping understand the more precise actions of the developer, and this should be captured by our model. To do so, for each event type, by looking through the supplied descriptions, we map the associated values which are generated by the event. Table I shows event types with their respective fields and what they represent. Not all event types had them, the ones without additional descriptor fields are not included in the table. We append onto the event type string the respective descriptor. For example, the resulting command string looks like the following: *"CommandEvent-Refresh"*, where *"CommandEvent"* is the event type and *"Refresh"* is the additional descriptor. We then concatenate all the event strings per session. Resulting in 3310 session event strings.

III. Methodology

In this section we describe the process to obtain our predictions given the data we have prepared as described in the previous Data preparation section.

A. Target Class

With our event series prepared as strings we need to break these up into our features and corresponding target classes, or simply stated we need to break these strings up into the events before a command and then the command. To do so, we find all the command event type tokens per session string and then grab up to ten previous tokens for each of them. For example if we have the following session string: *"VisualStudio.IDEStateEvent VisualStudio.WindowEvent-1 VersionControlEvents.VersionControlEvent CommandEvent-FeedbackGenerator.UploadWizard VisualStudio.WindowEvent-1"* We would extract the *"CommandEvent-FeedbackGenerator.UploadWizard"* token, which is the target class, and then extract all the tokens before, e.g. *"VisualStudio.IDEStateEvent VisualStudio.WindowEvent-1 VersionControlEvents.VersionControlEvent"*, which are the corresponding features. We repeat this for each command token in the string. After the target class extraction, there was a large number of custom and infrequent target classes. We decided to only take the command events which made up most of the target classes. Figure 3 shows the cumulative sum of the command event types. From which we can see that a small proportion of command events make almost the entirety of target classes. We took only the most frequent commands which made 90% percent of the cumulative sum, resulting in 67 target classes.

B. N-Grams

To represent these series of events we use a method called N-Grams which takes strings and breaks them up into tuples of N length where a gram is a token. Take our previous feature string for example: *"VisualStudio.IDEStateEvent VisualStudio.WindowEvent-1 VersionControlEvents.VersionControlEvent"*. A bigram break down (2-Gram) of it would look like (*"VisualStudio.IDEStateEvent"*,*"VisualStudio.WindowEvent-1"*), (*"VisualStudio.WindowEvent-1"*,*"VersionControlEvents.VersionControlEvent"*). These N-Grams can be a variety of length and for our purposes we experiment with runs on different N-Grams lengths ranging from 1 to 4. Once we have these N-Gram representations of our features, we put them through a count vectorizer, which produces a matrix that represents the counts of N-Grams.

C. Model

We chose to use the classification model Multinomial Naive Bayes which is based off conditional probability and works well with text classification tasks. This works well considering we would like to harness the natural conditional probability occuring in event series. We report on the accuracy of 10-cross-fold validation.

IV. Results

In this section we discuss the results of our predictive models performance.

Table II shows the results of the model with different ranges of N-Grams. It also includes the number of N-Grams generated per range, as it grows quickly as the range increases. Due to the massive volume of data generated by the possible N-Gram combinations of events, we had to randomly sample 100,000 examples to do the classification on. To emphasize the reason for random sampling, take the last value in the Number of N-Grams column in Table II, 27,485, this means we had a 100,000x27,485 matrix which we had to store in memory. This almost maxed out our 16GB of RAM. To do grams of range [1,5] would produce 45,728 N-Grams. With the 10-fold-cross validation our model achieves an accuracy of about 70% when predicting across the 67 possible classes.

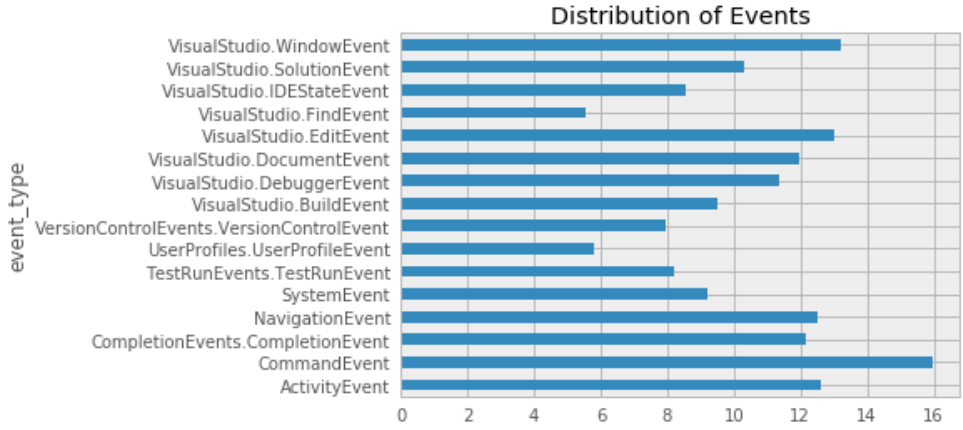

Figure 2: Logged frequency of event type occurrences in sessions

Table I: Events with Additional Descriptors

Event type	Additional descriptor field	Explanation of descriptor
CommandEvent	Command ID	Identifies the action/command that was executed
CompletionEvents.CompletionEvent	Terminated State	Result of the completion (e.g., APPLY, CANCEL, ...)
VisualStudio.DocumentEvent	Action	Action that took place (i.e., OPEN, SAVE, CLOSE)
VisualStudio.FindEvent	Cancelled	Was the find cancelled?
VisualStudio.SolutionEvent	Action	Action that was performed (e.g., OpenSolution, RenameSolution, ...)
VisualStudio.WindowEvent	Action	Action that was performed (e.g., MOVE, CLOSE, ...)
NavigationEvent	Type of Navigation	What kind of navigation was performed (e.g., ctrl-click)
TestRunEvents.TestRunEvent	Was Aborted	Was the execution aborted?

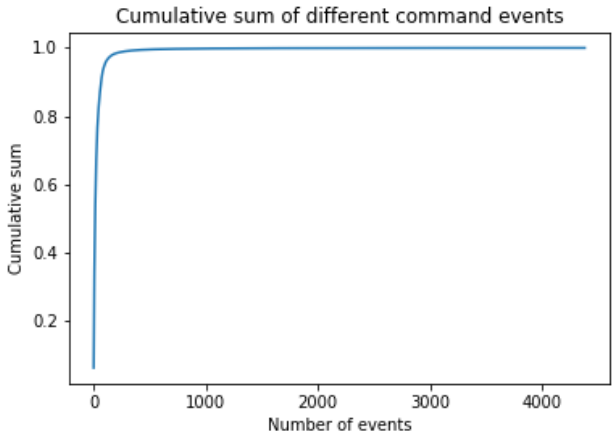

Figure 3: Cumulative sum of command type event classes

Table II: NGram Range Results

N-Gram range	Number of N-Grams	Accuracy
[1,1]	1452	70.42%
[1,2]	4927	70.26%
[1,3]	12642	70.48%
[1,4]	27485	70.67%

V. Threats to Validity

The following sections discuss the construct, internal, and external threats to validity of this study.

A. Construct Validity

The main concern regarding the construct validity of the study is with respect to the data source and its cleaning. As the data started in a JSON format and was converted to a SQL schema. The data has many other attributes which were available, but not used for this study and could have been overlooked when manually going over the supplied documentation.

B. Internal Validity

We only have a small description of what the developers experience and projects were, meaning they could have been working on anything, some of which could be not even code related, or their IDE could have been open while using other applications and it still record various activities like those that pertain to “Window” events.

C. External Validity

While the events and commands recorded by the tool which collected the data are mostly platform specific, we believe the methodology is generic enough to be applied to other event series collecting tools.

VI. Conclusion and Future Work

To help keep developers focused, and therefore less likely to introduce bugs, one could ensure that their IDE’s perform with little to no delay time. A step towards ensuring this can be to harness predictive models combined with the recently released data of recorded developer activity events. Our findings suggest that a Multinomial Naive Bayes model can be used to

accomplish this task to an accuracy of 70% on a small sample of the data.

RQ1: Can we use machine learning models to predict developer induced IDE commands?

Based on our results, we see that it is possible to achieve an accuracy of 70% across a target set of 67 possibilities, therefore we can infer the answer to RQ1 as yes, we can.

Future work which can harness these results would be to integrate command prediction into an IDE to test its feasibility and actual performance in a development environment.

ACKNOWLEDGMENT

The authors would like to thank Omar Elazhary for going through the tedious process of parsing the data, which was originally in many separate JSON files, into a easily accessible and usable Postgres SQL dump.

APPENDIX A
REPLICATION PACKAGE

To replicate this study one can access and download the same data dump which was used for this study at:

http://turingmachine.org/~dmg/temp/events-omar.sql.gz

A Jupyter Notebook which is broken into data analysis, data processing, formatting and predicting can be found at:

https://github.com/tysonbul/Predicting-Developer-Commands

REFERENCES

[1] Dario Di Nucci et al. “On the role of developer’s scattered changes in bug prediction”. In: *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE. 2015, pp. 241–250.

[2] Sebastian Proksch, Sven Amann, and Sarah Nadi. “Enriched Event Streams: A General Dataset for Empirical Studies on In-IDE Activities of Software Developers”. In: *Proceedings of the 15th Working Conference on Mining Software Repositories*. 2018.

[3] Feng Zhang et al. “An empirical study of the effect of file editing patterns on software quality”. In: *Journal of Software: Evolution and Process* 26.11 (2014), pp. 996–1029.