Predicting Developers' IDE Commands with Machine Learning

Tyson Bulmer, Lloyd Montgomery, Daniela Damian
University of Victoria
Victoria, Canada
{tysonbul,lloydrm,danielad}@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 Chrome's 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 analyze and cleanse the data to be parsed into event series, which can then be used to train a variety of machine learning models, including a neural network, to predict user induced commands. Our highest performing model is able to obtain a 5 cross-fold validation prediction accuracy of 64%.

CCS CONCEPTS

• **Computing methodologies → Neural networks**; *Feature selection*; • **Software and its engineering → Integrated and visual development environments**;

KEYWORDS

IDE Monitoring, Machine Learning, Neural Network, Developer Commands

ACM Reference Format:
Tyson Bulmer, Lloyd Montgomery, Daniela Damian. 2018. Predicting Developers' IDE Commands with Machine Learning. In *MSR '18: MSR '18: 15th International Conference on Mining Software Repositories , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3196398.3196459

1 INTRODUCTION AND BACKGROUND

The task of developing software is a thought-intensive process and interruptions can derail a train of thought easily. These interruptions can come in a variety of ways, one of which is slow loading time within an integrated development environment (IDE). Past research in bug prediction has touched on the topic of developer "focus" as a contributor to bug proneness. Di Nucci et al. propose that the higher a developer's focus on their activities, the less likely they are to introduce bugs to the code [3]. Zhang et al. proposed that certain file editing patterns by developers writing code could contribute to bug proneness as well. Two of their patterns which highlight the "focus" of a developer are the "Interrupted" and "Extended" editing patterns [21], which they found introduced 2.1 and 2.28 (respectively) times more future bugs than without.

The bigger picture of *developer activity within an IDE*, as characterized by our own investigation of this research area, can be grouped into four categories: tracking, characterizing, enhancing, and predicting. Tracking developer activity is the most published area, due in large part to the necessity of tracking information before it can be characterized, enhanced, or predicted. One of the first published works in this area is by Teitelman and Masinter [19] who sought to expand the functionality of IDEs with automatic analyzing and cross-referencing of user programs. Takada et al. built a tool that automatically tracks and categorizes developer activity through the recording of keystrokes [18]. Their work introduced the concept of live-tracking developers, making way for the refined IDE activity tracking tools to follow.

Mylyn is a task and application lifecycle management framework for Eclipse [1]. It began as Mylar, a tool for tracking developer activities in IDEs, focusing on building degree of interest (DOI) models to offer suggestions to developers on which files they should be viewing [7]. Mylyn has always had a focus on tracking, characterizing, and enhancing developer activity, but not predicting [4, 8, 13].

DFlow is a tool developed by Minelli and Lanza to track and characterize developer activities within IDEs [11, 12]. Their research focused on the tracking and characterization of developer activities, and not enhancing or predicting their activities.

Other notable IDE integrations includes Blaze—a developer activity enhancer by Singh et al. [16], WatchDog—an IDE plugin that listens to UI events related to developer behavior designed to track and characterize developer activity by Beller et al. [1], and FeedBaG—the developer activity tracker for Visual Studio [2] coupled with research designed to track and characterize developer activity.

In this paper we use a corpus of over 10 million developer activity events generated in Visual Studio and recorded by FeedBaG [15], to train a model to predict developer commands. We formally address our intent with the following research question:

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

Similar work has been done on trying to predict developer behaviour in the IDE by Damevski et al., where they abstracted

MSR '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196459

[1] https://www.eclipse.org/mylyn/
[2] https://www.visualstudio.com/

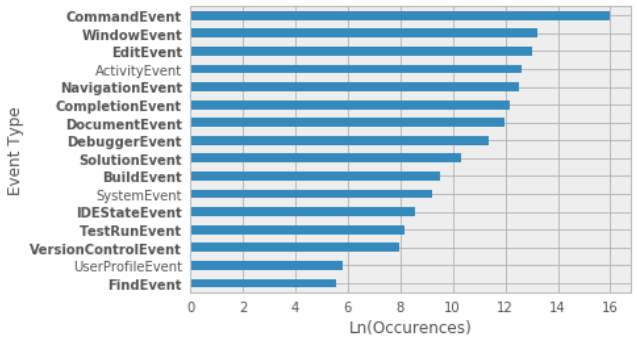

Figure 1: Frequency (ln()) of event-type occurrences in sessions, with utilized event-types in bold

recorded IDE event sequences using topic models to predict developer future tasks, such as debugging or testing [2]. Our research differs from Damevski's work by harnessing a neural network and the granularity in which our model predicts command-level actions.

We believe that if we can predict user commands, then we can reduce slow command process times of actions by preprocessing the predicted user commands, similar to applications such as Google Chrome which has predictive page loading to speed up browsing [3].

This paper is structured as follows: Section 2 describes the methodology, Section 3 covers the results, Section 4 discusses the threats to validity, and Section 5 concludes the paper.

2 METHODOLOGY

In this section we describe our process to analyze, clean, and format the data into a form which can represent features and labels for our machine learning models, as well as our selection of models, and their performance evaluation.

2.1 Data Description

The dataset we use is supplied by Proksch et al. [15] for the 2018 MSR Challenge[4]. The dataset was collected by FeedBaG++[5], an IDE integration that logs events occurring within the IDE including developer activities, IDE environment changes, navigation, and more. Released in March 2017, this dataset contains over 10 million events, originating from 81 developers and covering a total of 1,527 aggregated days of developer work.

The dataset contains 18 unique IDE event types, but we only utilized 13 event types initiated by the developer. The most common developer-induced events are "Command" events which are an "invocation of an action" by the developer, e.g. "clicking a menu button or invoking a command via shortcut" [15]. A complete listing and descriptions of these events can be found on the data supplier's website[6]. Figure 1 shows the natural logarithm of the distribution of 16 of the event types across all sessions, with the 13 developer-induced events in bold (2 of the 18 types were lost in the original parsing of the data). From this we can see that the Command event type is the most commonly occurring event type.

[3]https://support.google.com/chrome/answer/1385029
[4]https://2018.msrconf.org/track/msr-2018-Mining-Challenge
[5]http://www.kave.cc/feedbag
[6]http://www.kave.cc/feedbag/event-generation

2.2 Data Processing

In this section we describe the steps taken to select, cleanse, and format the dataset. The dataset began with ~10.7 million events.

2.2.1 Event-Type Selection. As previously mentioned, we only utilize 13 of the original 18 event types available in the dataset, which are developer-induced. These 13 event types are desired because they are invoked by the developer, and therefore part of the developer's working pattern. Although other dynamic system responses are occurring, we wanted to focus specifically on the workflow of developers. The non-developer-induced event types, "SystemEvent", "InfoEvent", and "ErrorEvent", as well as the "ActivityEvent" (recorded when the mouse moves) and "UserProfileEvent" (recorded when user profile for the FeedBaG++ tool is modified), were removed from the dataset. Roughly 300,000 events were removed due to this selection.

2.2.2 Data Cleansing. Just over 400,000 events had been recorded twice with the same triggered-at date, event type and session ID. These events are considered duplicate records of the same triggered event and therefore were removed from the dataset.

From our manual investigation of the data, we found that many events, including Command events, were repeated hundreds of times in quick succession, milliseconds apart. This behaviour is not easily explained, nor is it insignificant, so we did not want to remove them completely. However, we believed that this repetition of events would negatively influence our models as they would learn to predict these heavily repeated events and still achieve high performance in the test environment—while failing to perform well in the real environment. To avoid this situation, while preserving the existence of the repetitions, we applied the regular expression notation of "+" (meaning one or more) to events which repeated one or more times. For example, the sequence "Command, Command, Command" would be replaced with "Command+." This conversion process removed just over 6 million events.

2.2.3 Event Representation. Most of the event types have additional descriptors about the recorded event which distinguish them from other similar event types. For example, the Window Event also records an "action" value which represents whether the action of the Window event 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 models. For each event type, we look through the supplied descriptors and map the associated values which are generated by the event.

We append onto the event type the respective descriptor. For example, a Command event instance would be represented as follows: *"CommandEvent-Refresh"*, where *"CommandEvent"* is the event type and *"Refresh"* is the additional descriptor.

2.3 Feature and Target Class Extraction

Due to the *event stream* structure of the data, it is important to frame how we are extracting the features and target classes from the data.

Although our data contains 13 developer-induced event types, we are only trying to predict Command events with their additional descriptors. There are a total of 651 unique Command descriptors, but since the 61 most frequently used Command descriptors account

for 90% of the total Commands invoked, we narrowed down the target classes to just those 61. In trying to predict developer-induced events (specifically Commands) for the purpose of preprocessing them, focusing on the most frequent Commands descriptors allows us to target the most likely actions that will occur.

Secondly, the data is recorded as multiple sessions of user event streams, and we need to frame what the features of the target classes are. For illustrative purposes, let's say one session's event stream is *G1, G2, C1, G3, C2*, where *C* events are one of the 61 Command descriptor types described above, and *G* events are any other generic event, including any of the other command descriptor types, or any of the other event types along with their various additional descriptors. As described above, we are only interested in predicting the top 61 Command descriptors, so from the example we would extract two rows in our X and y machine learning matrices (where X contains the features and y contains the target classes). The first row of features would be *{G1, G2}* with the target class of *C1*, and the second row of features would be *{G1, G2, C1, G3}* with the target class of *C2*.

2.4 Models

We used four functionally different models to explore which would perform the best for the given task of predicting the next event. We trained and tested on a randomly sampled 100,000 event series from our 3.9 million events (remaining after processing described in Section 2.2). This reduced sample is due to the Logistic Regression model's computationally expensive algorithm performed with over 12 thousand features which are a result of the largest N-Gram range. For all the models, we conduct 5-fold cross-validation, this was chosen in favour of 10-fold cross-validation, as it required training only half as many models which reduces the computational load. The models we tested are as follows:

2.4.1 Naive Bayes. We harnessed Sci-Kit-learn's implementations of the classification models Bernoulli and Multinomial Naive Bayes which are based on conditional probability and used commonly for text classification tasks [14]. To represent these series of events we use a method called N-Grams [10] which takes series of tokens, in our case *events*, and breaks them up into tuples of N length. Take the following event series for example: *E1, E2, E3*, A bigram breakdown (2-Gram) of it would look like (*E1,E2,*) (*E2,E3.*) These N-Grams can be a variety of length which can affect the performance of our models so we experiment with runs on different N-Grams lengths ranging from 1 to 3. We limit the max N-Gram range to 3 due to the limited computational resources available. 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.

2.4.2 Logistic Regression. To widen our variety of models, we also tried Scikit-learn's Logistic Regression model [9]. Similarly to the Naive Bayes models, vectorized N-Grams are used to represent the data.

2.4.3 Neural Network. We also implemented a Neural Network (NN), using the high-level neural networks API Keras, which consisted of rectified linear unit hidden layers of 500, and 100 nodes, including a dropout layer with a dropout rate of 0.5 between the two hidden layers to help reduce overfitting [17], and then a final layer with softmax activation for the step of classification. The way in which we represent the data for this model requires that we encode the events to categorical numbers, and that we one-hot encode the target classes, which is a method for representing multiple classes as binary features [5].

2.5 Reporting Results

To compare the performances of our models, which are performing multi-class classification, we use two measures: accuracy and micro-averaged receiver operator characteristic area under the curve (ROC AUC). The accuracy measure is the total accuracy across the 5 folds of the 5-fold cross validation. Meaning, the predictions from each fold across the entire dataset are aggregated and compared to the actual classes. Since our target classes are imbalanced, we use the micro-averaged ROC AUC, which is the weighted average ROC AUC for each class [20]. Meaning, the ROC AUC was calculated for each class as if it was a binary classification task, and then we take the weighted average of these ROC AUCs to account for the disproportion of class data points. The micro-average is an aggregate of the contributions of all classes to compute the average metric. For both the accuracy and micro-averaged ROC AUC, higher values are better.

3 RESULTS

In this section we discuss the results of each of our predictive models by reporting the accuracy and micro-averaged AUC ROC of the 5-fold cross-validation results.

3.1 Naive Bayes

The Bernoulli Naive Bayes models achieved its highest accuracy of 20.52% with N-Grams of ranges 1, 2, and 3. Its micro-average ROC AUC was 0.60, as shown in Table 1. AUC's for each target class, or target command, ranged from 0.5 to 0.96, with a mean of 0.55 and standard deviation of 0.10.

The Multinomial Naive Bayes models achieved its highest accuracy of 20.14% with N-Grams of ranges 1, 2, and 3. Its micro-average ROC AUC was 0.59, as shown in Table 1. AUC's for each target class, or target command, ranged from 0.5 to 0.97, with a mean of 0.61 and standard deviation of 0.14.

3.2 Logistic Regression

The Logistic Regression model achieved its highest accuracy of 35.87% with N-Grams of ranges 1, 2. Its micro-average ROC AUC was 0.67, as shown in Table 1. AUC's for each target class, or target command, ranged from 0.50 to 0.95, with a mean of 0.63 and standard deviation +/- 0.12.

3.3 Neural Network

The NN performed achieved the highest accuracy among the classifiers with an accuracy of 64.39%. Its micro-average ROC AUC was 0.82, as shown in Table 1. AUC's for each target class, or target command, ranged from 0.5 to 0.98, with a mean of 0.72 and standard deviation +/- 0.17.

Table 1: All models results

Model	N-Gram Range	Accuracy (5-fold)	Micro AUC
Bernoulli Naive Bayes	[1,1]	15.74%	0.57
	[1,2]	18.77%	0.57
	[1,3]	20.53%	0.60
Multinomial Naive Bayes	[1,1]	16.31%	0.57
	[1,2]	17.73%	0.58
	[1,3]	20.14%	0.59
Logistic Regression	[1,1]	25.37%	0.62
	[1,2]	35.88%	0.67
	[1,3]	35.22%	0.67
Neural Network	—	**64.39%**	**0.82**

4 THREATS TO VALIDITY

This section discusses the construct, internal, and external threats to the validity of this study and their respective mitigations.

Construct Validity. The main concern regarding construct validity in this research is the assumption placed on the data received. We only have a small description of what the developers experience and projects were, meaning they could have been working on anything. Some of their IDE activities might have not even been code related, or their IDE could have been open while using other applications and it still would have recorded various activities like those that pertain to "Window" events. However, we mitigated this by only predicting developer-induced events, which are more likely to be done during active development.

Internal Validity. A threat to internal validity in this study comes with 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 able to be used for this study. However, since there is always the opportunity for more data to be used in a machine learning implementation to improve results, we see the impact of excluding that data as minimal to the study design.

External Validity. A threat to external validity in this study pertains to the events and commands recorded by the FeedBaG++ tool. However, the recorded event types we utilized are mostly generic, such as TestRunEvent and BuildEvent, and therefore this threat is mitigated by the general nature these attributes.

5 CONCLUSION AND FUTURE WORK

To help keep developers focused, and therefore less likely to introduce bugs, one could ensure that their IDEs 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 Neural Network can be used to accomplish this task with an accuracy of 64%.

Answer to RQ: Based on our results, we see that it is possible to achieve an accuracy of 64% across a target set of 61 possibilities while maintaining a micro-averaged ROC AUC of 0.82; therefore, we conclude that the answer to our research question is yes.

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. For improved prediction performance, given access to more computational resources, one could try using a Recurrent Neural Network (RNN) with the entire (unfiltered) event series. As RNN's are able to achieve high performance when used for time series prediction tasks [6].

REFERENCES

[1] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, how, and why developers (do not) test in their IDEs. In *Proc. 2015 10th Jt. Meet. Found. Softw. Eng. - ESEC/FSE 2015*. ACM Press, New York, New York, USA, 179–190. https://doi.org/10.1145/2786805.2786843

[2] Kostadin Damevski, Hui Chen, David Shepherd, Nicholas A. Kraft, and Lori Pollock. 2017. Predicting Future Developer Behavior in the IDE Using Topic Models. *IEEE Trans. Softw. Eng.* 1, 1 (2017), 1–12.

[3] Dario Di Nucci, Fabio Palomba, Sandro Siravo, Gabriele Bavota, Rocco Oliveto, and Andrea De Lucia. 2015. On the role of developer's scattered changes in bug prediction. In *2015 IEEE 31st Int. Conf. Softw. Maint. Evol. ICSME 2015 - Proc.* IEEE, Bremen, Germany, 241–250. https://doi.org/10.1109/ICSM.2015.7332470

[4] Thomas Fritz, Gail C. Murphy, and Emily Hill. 2007. Does a programmer's activity indicate knowledge of code?. In *Proc. 6th Jt. Meet. Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng. - ESEC-FSE '07*. ACM Press, New York, New York, USA, 341. https://doi.org/10.1145/1287624.1287673

[5] Hynek Hermansky, Daniel PW Ellis, and Sangita Sharma. 2000. Tandem connectionist feature extraction for conventional HMM systems. In *Acoustics, Speech, and Signal Processing, 2000. ICASSP'00. Proceedings. 2000 IEEE International Conference on*, Vol. 3. IEEE, Istanbul, Turkey, Turkey, 1635–1638.

[6] Michael Hüsken and Peter Stagge. 2003. Recurrent neural networks for time series classification. *Neurocomputing* 50 (2003), 223–235.

[7] Mik Kersten and Gail C. Murphy. 2005. Mylar: a degree-of-interest model for IDEs. In *Proc. 4th Int. Conf. Asp. Softw. Dev. - AOSD '05*. ACM Press, New York, New York, USA, 159–168. https://doi.org/10.1145/1052898.1052912

[8] Mik Kersten and Gail C. Murphy. 2006. Using task context to improve programmer productivity. In *Proc. 14th ACM SIGSOFT Int. Symp. Found. Softw. Eng. - SIGSOFT '06/FSE-14*. ACM Press, New York, New York, USA, 1.

[9] Su-In Lee, Honglak Lee, Pieter Abbeel, and Andrew Y Ng. 2006. Efficient l˜ 1 regularized logistic regression. *AAAI* 6 (2006), 401–408.

[10] Huma Lodhi, Craig Saunders, John Shawe-Taylor, Nello Cristianini, and Chris Watkins. 2002. Text classification using string kernels. *Journal of Machine Learning Research* 2, Feb (2002), 419–444.

[11] Roberto Minelli and Michele Lanza. 2013. Visualizing the workflow of developers. In *2013 1st IEEE Work. Conf. Softw. Vis. - Proc. Viss. 2013*. IEEE, Eindhoven, Netherlands, 1–4. https://doi.org/10.1109/VISSOFT.2013.6650531

[12] Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time. (2015), 25–35 pages. https://doi.org/10.1109/ICPC.2015.12

[13] Gail C. Murphy, Mik Kersten, and Leah Findlater. 2006. How are java software developers using the eclipse IDE? *IEEE Softw.* 23, 4 (jul 2006), 76–83.

[14] Kevin P Murphy. 2006. *Naive bayes classifiers*. Technical Report. University of British Columbia, Department of Computer Science.

[15] Sebastian Proksch, Sarah Nadi, Sven Amann, and Mira Mezini. 2018. Enriching in-IDE process information with fine-grained source code history. In *Proc. 15th Int. Work. Min. Softw. Repos. - MSR '18*. IEEE, Gothenburg, Sweden.

[16] Vallary Singh, Will Snipes, and Nicholas A. Kraft. 2014. A framework for estimating interest on technical debt by monitoring developer activity related to code comprehension. In *Proc. - 2014 6th IEEE Int. Work. Manag. Tech. Debt, MTD 2014*. IEEE, Victoria, BC, Canada, 27–30. https://doi.org/10.1109/MTD.2014.16

[17] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research* 15, 1 (2014), 1929–1958.

[18] Y Takada, K Matsumoto, and K Torii. 1994. Programmer Programmer State Performance Measure and Based Debugging on Process Transitions in Testing. In *Proc. 16th Int. Conf. Softw. Eng.* IEEE Computer Society Press, Sorrento, Italy, 123–132. https://dl.acm.org/citation.cfm?id=257752

[19] Warren Teitelman and Larry Masinter. 1981. The Interlisp Programming Environment. *IEEE Computer* 14, 4 (1981), 25–33.

[20] Yiming Yang. 1999. An Evaluation of Statistical Approaches to Text Categorization. *Information Retrieval* 1, 1 (01 Apr 1999), 69–90.

[21] Feng Zhang, Foutse Khomh, Ying Zou, and Ahmed E. Hassan. 2014. An empirical study of the effect of file editing patterns on software quality. *J. Softw. Evol. Process* 26, 11 (nov 2014), 996–1029. https://doi.org/10.1002/smr.1659