Categorizing Compiler Error Messages with Principal Component Analysis

Brett A. Becker* and Catherine Mooney

Abstract: Being a competent programmer is critical for students in all computing disciplines and software engineering in particular. Novice programming students face a number of challenges and these have been shown to contribute to worrying dropout rates for students majoring in computing, and the growing number of non-majors who are learning to program. Methods of identifying and helping at-risk programming students have been researched for decades. Much of this research focuses on categorizing the errors that novice programmers make, in order to help understand why these errors are made, with the goal of helping them overcome these errors quickly, or avoid them altogether. This paper presents the first known work on categorizing compiler errors using Principal Component Analysis. In this, we find a new way of discovering categories of related errors from data produced by the students in the course of their programming activity. This method may be used to identify where these students are struggling and provide direction in efforts to help them.

Keywords: Compiler errors; Compiler error messages; Novice programmers; CS1; Error message categorization; Java programming; Principal component analysis

1 Introduction

Novice programming students face a number of challenges, and difficulties with computer programming in particular have been cited as a contributing factor in worrying dropout rates [1]. Globally one third of students fail their introductory programming course [2, 3], often referred to as CS1 [4]. One source of difficulty is struggling with basic programming concepts themselves – some students may struggle with repetition statements, others with relational operators, etc. Another challenge comes in the form of notoriously cryptic compiler error messages (CEMs), which result from the coding errors made by students in source code [5]. One line of effort to understand these challenges is categorizing novice errors. Generally these studies fall into two camps, those that categorize errors committed by students in source code (referred to in this paper as simply *errors*), and those that categorize the compiler error messages (CEMs) resulting from these errors. Often, both camps focus on the most frequent errors as for many languages the majority of students spend the majority of their time on a minority of error types [6, 7].

This paper presents the first known work on categorizing compiler errors using Principal Component Analysis (PCA), a non-parametric method of reducing a complex data set to reveal hidden, simplified dynamics within it. This is accomplished by converting a set of observations of variables (which may be correlated) into a set of values of linearly uncorrelated principal components. These principal components may then reveal relationships between the variables. Using this technique, we attempt to 'relate' two or more CEMs. In this way we identify several CEMs which if generated by a student, indicate that the same student has a high likelihood of generating other identifiable errors. In effect this forms categories of (related) errors. We focus on Java errors as Java is the most popular programming language for teaching novices to program [8, 9] and is

• Brett A. Becker is with the Beijing Dublin International College, 100 Ping Le Yuan, Chaoyang District, Beijing 100124, China, and School of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland. brett.becker@ucd.ie

• Catherine Mooney is with the Physiology and Medical Physics Department, Royal College of Surgeons, 123 St. Stephen's Green, Dublin 2, Ireland. catherinemooney@rcsi.ie

∗ To whom correspondence should be addressed. Manuscript also the most popular language in general [10]. We feel that approaches such as that presented in this paper are made more necessary as more data-driven metrics appear [11].

This paper is organized as follows. Section 2 presents related work, Section 3 presents our aims, methodology and principal component analysis. Section 4 presents our results and Section 5 presents our conclusions.

2 Related Work

Both student-committed errors and CEMs lend themselves to many types of categorization, for instance: frequency; type (syntax, semantic, logic, runtime, etc.); defined categories (bracketing, repetition, variable declaration / initialization, scope, etc.); non-literal errors (discussed later); and more.

Hristova et al. [12] compiled a list of common Java errors committed by students, combining data from sources including teaching assistants, students and professors from 58 institutions, and members of SIGCSE. This list of 62 errors was reduced to 20 as some were deemed too advanced for their students to encounter, or because the editor they used was capable of identifying the error in a useful enough way (their goal was to enhance CEMs). Of these 20, 13 were syntax errors (such as '= versus ==', '&& versus &', etc.), 6 were logic errors (such as 'improper casting'), and 1 was a semantic error ('invoking class method on object'). Nine years later Chan Mow [13] gathered the most common errors generated by her students and categorized them in the same way as Hristova et al. 94.1% of these errors were syntax, 4.7% semantic, and 1.2% logic.

One problem that these studies reveal is the lack of a one-to-one mapping between errors and CEMs – that is, a single error may, in different contexts, produce difference CEMs, and the same CEM may be produced by entirely different and distinct errors. As the ultimate goal of these studies is to help students learning to program, not being able to identify an error based on the CEM it generates (or vice versa) complicates the approach, and draws a distinction between research on errors and research on CEMs, despite both approaches normally having a common end goal.

Dy and Rodrigo [14] explored 'non-literal errors' where the CEM does not match the actual error committed by the student. In doing so they compiled a list of the most common CEMs encountered in their data. 'Similar' errors were grouped together, for instance all *cannot find symbol* errors formed one category, regardless of if the symbol was a variable, method, or class. Unfortunately, such categorization can introduce difficulty when comparing results to those of others. However due to the sheer number of possible CEMs, and the fact that they are not guaranteed to be stable across Java versions [15], such categorization is hard to escape, particularly when a study is done without considering generalization.

McCall and Kölling [16] sought to improve this situation by developing and validating error categories. Their results showed that error causes can be manually analyzed by independent researchers with good reliability, and that work (using CEMs only) tended to group some distinct errors together in single categories, which can be listed more accurately. They identified a total of 80 categories which were divided into syntax, semantic, and logic errors. They found that in general, semantic errors showed a greater tendency to correspond to CEMs when compared to syntax errors, showing that categorization scheme can achieve a higher precision in describing these errors.

Many studies either categorize errors by frequency [17], or categorize errors (into types, etc.) after being arranged by frequency. Again, this is because educators are most concerned with the most frequent errors, which normally only feature a small subset of possible errors, particularly in the case of novices who generally do not utilize many advanced features of a language and therefore do not encounter errors arising from their use. Categorizing errors by frequency alone can provide insight and direct efforts to aid students, but also opens questions. In [18] Jadud noted that the distributions of the most frequent errors across six languages including COBOL, Fortran, Haskell, Java, LOGO and SOLO are very common given the diversity of these languages. Of these, Haskell stood out most, with 50% of all errors encountered being one type. Jadud felt that a refinement of *type error* (the most frequent) would bring the distribution more into line with the other languages. In [6], we showed that the distributions of the ten most frequent CEMs from six different studies all focusing on Java had very similar distributions. This similarity is despite the fact that some of these studies categorized some errors together before ranking them by frequency, while others did not. In fact, four of the top five errors were common to all six studies, subject to minor differences due to using different Java versions, or study-specific groupings such as some grouping all *cannot find symbol* CEMs together, and others reporting *cannot find symbol* (*variable*/*method*/*class*) separately.

Recent work by Pritchard [19] examined the link between errors (in this case CEMs) and statistical distributions. A good fit was found between Python and Java error frequency distributions and Zipf-Mandelbrot distributions. This work gives one possible way to contrast languages or compilers quantitatively and holds promise for future research.

Also recently, the Blackbox dataset [15] has begun storing large-scale naturally accumulating process data (NAPD). This data is available to researchers and holds significant potential to explore the behavior of programmers including categorization of errors. Altadmri and Brown [20] used a year of this data involving 250,000 users to investigate 18 errors from their previous work [21], ultimately derived from [12]. The results are intended to inform tools to target the most frequent (or hardest to fix) errors. Jadud and Dorn [22] used this data to explore repeated errors at a large-scale applying Jadud's error quotient [18]. Providing novel insight, but complication in using such large-scale data for such purposes, their results show quantifiable and statistically significant country-specific differences. They conclude that external, objective measures of student performance, either integrated at the time of learning or applied post-hoc, hold promise to radically transform our understanding of novice programming behavior at both small and large scales.

Rounding out recent developments, Ahadi et al. [23] explored analyzing NAPD with machine learning methods to identify high- and low-performing students. Although not focused on categorizing errors, this approach holds promise for potential use in categorization, which would provide another avenue to detecting at-risk students. Further, such advanced techniques are most likely required for the analysis of the large-scale NAPD data that is now available.

The current state of play is summarized by McCall and Kölling's conclusion: 'There still is no real agreement about the most common problems students encounter' [16] and by Brown and Altadmri [21] who found that educators only have a weak consensus about the frequency of student errors. Nonetheless the availability of NAPD and tools to analyze such large scale data hold promise, but currently the complications of such large data sets make smaller more controlled studies still very relevant.

3 Categorizing Errors Using Principal Component Analysis

3.1 Aims

In this work we seek to categorize CEMs, by relating them to each other by analyzing how users encounter them. In other words, we would consider two CEMs to be related if it can be shown that if a student commits an error generating a particular CEM, that student has a high likelihood of also committing an error generating a different CEM. For instance, if a particular student frequently forgot closing braces, we would reasonably expect that student to generate some number of *reached end of file while parsing* CEMs as omitting a closing } at the end of the final class in a file will generate this CEM. However we might also expect a relatively large number of *'else' without 'if'* CEMs, as this can result from omitting a closing brace at the end of an `if` statement which is followed by an `else`. In this hypothetical case it is fairly easy to predict that this student will most likely have higher than expected frequencies of both of these CEMs, thus relating them. However, would it be possible to analyze the behavior of a large group of students in a way that may reveal hidden (or at least not obvious) relations between two or more errors? In other words is it possible to say that finding a high frequency of error *x* is an indication of a high frequency of error *y*, or conversely a low frequency of error *z*? The ultimate goal of establishing these error categories is to help students overcome these errors quickly, or avoid them altogether.

3.2 Methodology

In this study we analyze the compilation behavior of over 100 students who undertook a CS1 course learning Java as part of a BSc in Information Technology as described in [5]. Students used a custom editor that had the capability of enhancing CEMs, although this feature was disabled for this group. Student data was logged during normal activity (working on assignments, labs, etc. as well as any other programming, such as individual practice, following along in lecture, etc.) and no skeleton code or other structures were provided. The data was filtered to remove students with very low levels of activity, as there was not necessarily a one-to-one mapping between anonymous compiler IDs and actual students, as described in [5]. Data was logged for six weeks, resulting in 39,127 errors, across 76 unique CEMs. This group was typical in that the most frequent errors accounted for the vast majority of errors: the 15 most frequent errors accounted for 85.3% of all errors, with the 15th accounting for only 1.7% of all errors in isolation. Figure 1 shows the number of errors per CEM for the 15 most frequent CEMs.

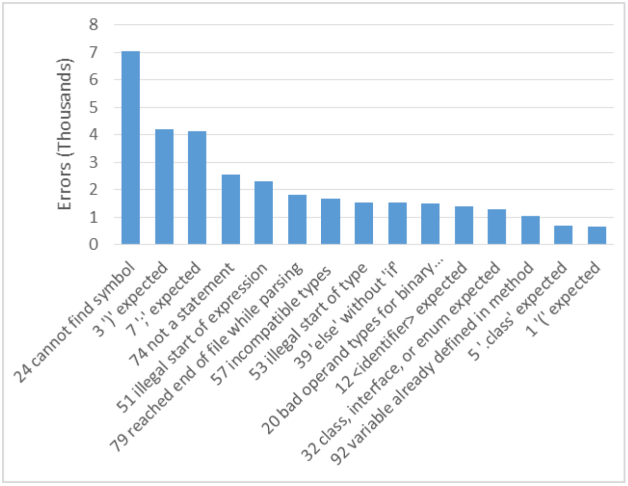

Figure 1. Frequency of top 15 CEMs

3.3 Principal Component Analysis

Principal Component Analysis (PCA) is a non-parametric method of reducing a complex data set to reveal hidden, simplified dynamics within it [24]. PCA takes as input a set of variables (in this case CEMs) which may be correlated and converts them into a set of linearly uncorrelated principal components (PCs), less than or equal to the number of original variables. PCA is useful for retaining data that accounts for a high degree of variance, and removing data which does not. We used the ggbiplot[1] function for the R statistical/graphical programming language to perform PCA.

4 Results

Figure 2 shows a plot displaying PCs 1 and 2, where each point represents a student. It shows a 68% confidence ellipse, arrows representing the correlation of CEMs with the PCs, and a circle representing the theoretical maximum extent of the arrows. The angle between any two arrows represents the correlation between those CEMs (90% is linearly uncorrelated). Thus, for example we could say that the correlation between CEMs 20 and 12 for PCs 1 and 2 is minimal.

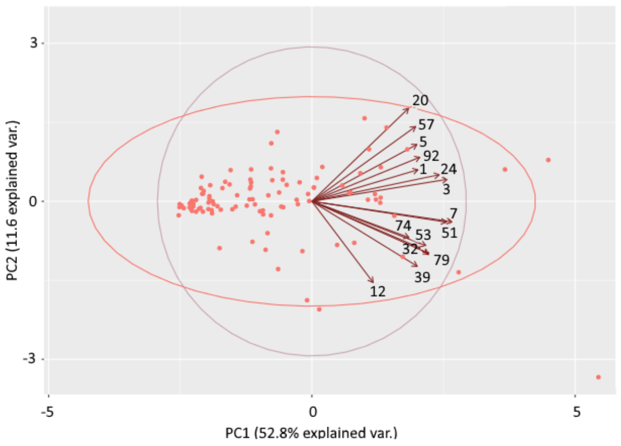

Figure 2. PC2 vs PC1 (<10% of outliers are beyond axis limits)

Two relative groupings of CEMs are seen (1, 3, 5, 20, 24, 57, 92) and (7, 12, 32, 39, 51, 53, 74, 79). Table 1 shows that all 15 CEMs are positively correlated with PC1, but for PC2, the first group is positively correlated while the second is negatively correlated.

Table 1. Pearson Correlation Coefficients of the first 4 principal components with the top 15 CEMs.

CEM	PC1	PC2	PC3	PC4
1	0.69	0.20	0.03	-0.26
3	0.88	0.14	-0.05	-0.16
5	0.68	0.37	0.13	-0.23
7	0.87	-0.13	-0.03	-0.26
12	0.40	-0.53	0.15	-0.57
20	0.63	0.60	0	-0.05
24	0.82	0.17	0.09	0.03
32	0.70	-0.30	0.41	0.38
39	0.68	-0.42	0.42	0.06
51	0.91	-0.13	-0.08	-0.01
53	0.74	-0.29	-0.51	0.11
57	0.67	0.48	0.01	0.07
74	0.63	-0.23	-0.68	0.09
79	0.76	-0.34	0.08	0.27
92	0.70	0.29	0.06	0.36

Figure 3 shows PCs 2 and 3 and reveals three groups (12, 32, 39, 79), (1, 3, 5, 20, 24, 57, 92) and (7, 51, 53, 74). Similarly, Figure 4 also shows three groups (51, 53, 74), (24, 32, 39, 57, 79, 92) and (1, 3, 5, 7, 12, 20).

[1] github.com/vqv/ggbiplot

CEMs 1, 3, 5, 20 appear in common groups in all three figures, as do CEMs 32, 39, 79 and CEMs 51, 53, 74. CEMs 1 *'(' expected* and 3 *')' expected* are the only 'bracketing' errors in the top 15 and based on that observation it is not surprising to find them in the same group. This could be taken as evidence that students who often generate CEM 1 also often generate CEM 3. CEMs 32 *class, interface, or enum expected*, 39 *'else' without 'if'* and 79 *reached end of file while parsing* all can result from forgetting a beginning or ending brace '{' or '}' in different contexts, and again it is not entirely unsurprising to see them grouped together (as discussed in Section 3.1). Finally, CEMs 51 *illegal start of expression* and 53 *illegal start of type*, both occur when a construct is expected but not found, while CEM 74 *not a statement* typically occurs when an otherwise syntactically correct statement occurs where it is not expected. In other words, these three CEMs all occur when a construct is expected but not found or when a construct is found where it is not expected. Students found struggling with this group of errors could struggling with a significant programming misconception involving these errors. We find these groupings (or categories), originating from principal component analysis, worthy of further investigation as a potential avenue towards helping students struggling with particular groups of errors.

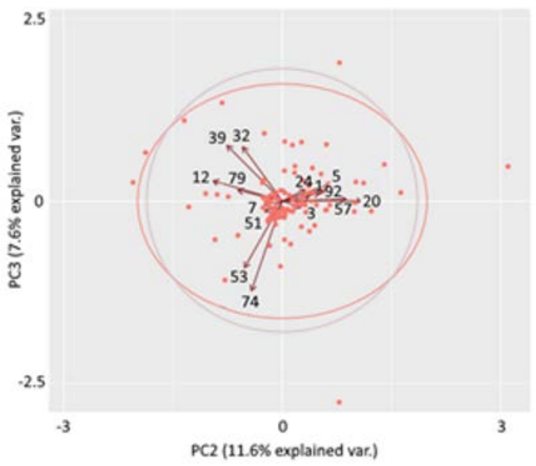

Figure 3. PC3 vs PC2 (<10% of outliers are beyond axis limits)

5 Conclusion

This paper presented an investigation into relating compiler error messages using principal component analysis. We found that PCA identified several groups of related errors that upon inspection provide grounds for further research. Many of the errors that we find to be related share common features – for instance, the only two bracketing errors were found in the same groups for all PCs investigated. These results are taken as the beginning of a larger investigation into relating compiler error messages (and the errors that generate them). Future work will involve establishing if these groups of errors can be validated and if so, developing specific interventions to help students who are found to make frequent errors in a specific group or groups.

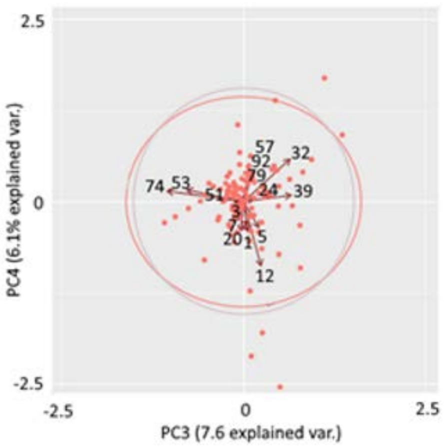

Figure 4. PC4 vs PC3 (<10% of outliers are beyond axis limits)

References

[1] A. Yadin, "Reducing the dropout rate in an introductory programming course," *ACM Inroads,* vol. 2, no. 4, pp. 71-76, 2011.

[2] J. Bennedsen and M. Casperson, "Failure rates in introductory programming," *ACM SIGCSE Bulletin,* vol. 39, no. 2, pp. 32-36, 2007.

[3] C. Watson and F. W. Li, "Failure rates in introductory programming revisited," in *Proceedings of the 2014 conference on Innovation & technology in computer science education*, 2014.

[4] M. Hertz, "What do CS1 and CS2 mean? Investigating differences in the early courses," in *Proceedings of the 41st ACM technical symposium on Computer science education*, 2010.

[5] B. A. Becker, *An Exploration of the Effects of Enhanced Compiler Error Messages for Computer Programming Novices,* Dublin Institute of Technology, 2015.

[6] B. A. Becker, "An Effective Approach to Enhancing Compiler Error Messages," in *Proceedings of the 47th ACM Technical Symposium on Computer Science Education*, Memphis, Tennessee, 2016.

[7] P. Denny, A. Luxton-Reilly and E. Tempero, "All syntax errors are not equal," in *ITiCSE 12*, 2012.

[8] S. Davies, J. A. Polack-Wahl and K. Anewalt, "A snapshot of current practices in teaching the introductory programming sequence," in *Proceedings of the 42nd ACM technical symposium on Computer science education*, 2011.

[9] R. M. Siegfried, D. Greco, N. Miceli and J. Siegfried, "Whatever happened to Richard Reid's list of first programming languages?," *Information Systems Education Journal*, vol. 10, no. 4, pp. 24-30, 2012.

[10] TIOBE Software, "TIOBE Index for February 2016," [Online]. Available: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[11] B. A. Becker, "A new metric to quantify repeated compiler errors for novice programmers," in *Proceedings of the 21st annual conference on Innovation and technology in computer science education*, Arequipa, Peru, 2016a.

[12] M. Hristova, A. Misra, M. Rutter and R. Mercuri, "Identifying and correcting Java programming errors for introductory computer science students," in *Proceedings of the 34th SIGCSE technical symposium on computer science education*, 2003.

[13] I. Chan-Mow, "Analyses of student programming errors in Java programming courses," *Journal of Emerging Trends in Computing and Information Sciences*, pp. 739-745, 2012.

[14] T. Dy and M. M. Rodrigo, "A detector for non-literal Java errors," in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, 2010.

[15] N. C. C. Brown, M. Kölling, D. McCall and I. Utting, "Blackbox: A large scale repository of novice programmers' activity," in *The 45th SIGCSE technical symposium on computer science education (SIGCSE)*, 2014.

[16] D. McCall and M. Kölling, "Meaningful categorisation of novice programmer errors," in *Proceedings of the 2014 International Conference on Frontiers in Education: Computer Science and Computer Engineering*, 2014.

[17] J. Jackson, M. Cobb and C. Carver, "Identifying top Java errors for novice programmers," in *Frontiers in Education, 2005. FIE'05. Proceedings 35th Annual Conference*, 2005.

[18] M. C. Jadud, *An exploration of novice compilation behaviour in BlueJ,* University of Kent, 2006.

[19] D. Pritchard, "Frequency Distribution of Error Messages," in *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2015)*, 2015.

[20] A. Altadmri and N. C. Brown, "37 million compilations: Investigating novice programming mistakes in large-scale student data," in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, Kansas City, 2015.

[21] N. C. Brown and A. Altadmri, "Investigating novice programming mistakes: educator beliefs vs student data," in *Proceedings of the tenth annual conference on International computing education research - ICER '14*, 2014.

[22] M. C. Jadud and B. Dorn, "Aggregate Compilation Behavior: Findings and Implications from 27,698 Users," in *Proceedings of the eleventh annual International Conference on International Computing Education Research*, 2015.

[23] A. Ahadi, R. Lister, H. Haapala and A. Vihavainen, "Exploring machine learning methods to automatically identify students in need of assistance," in *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, 2015.

[24] J. Shlens, "A tutorial on principal component analysis: derivation, discussion, and singular value decomposition," 25 March 2003. [Online]. Available: https://www.cs.princeton.edu/picasso/mats/PCA-Tutorial-Intuition_jp.pdf. [Accessed 14 August 2015].

[25] P. Denny, A. Luxton-Reilly, E. Tempero and J. Hendrickx, "CodeWrite: supporting student-driven practice of java," in *Proceedings of the 42nd ACM technical symposium on Computer science education*, 2011.

Brett A. Becker received the M.A. degree in higher education from the Dublin Institute of Technology, Dublin, Ireland in 2015, the Ph.D. degree in computer science and M.Sc. degree in computational science from University College Dublin, Dublin, Ireland in 2011 and 2003 respectively, and the BA degrees in computer science and physics from Drew University, Madison, New Jersey, USA in 2003. In 2006 he joined the Department of Computing at Griffith College Dublin as a Lecturer, and in 2009 became a Senior Lecturer. In 2012 he joined the College of Computing Technology as Head of Faculty and later served as Head of Academics. Since 2015 he has been a Lecturer in the School of Computer Science at University College Dublin. His research interests are computer science education and high performance computing. He is on the technical program committee (system software) for Supercomputing 2016, is the maintainer of the Irish Supercomputer List, the technical editor of the All-Ireland Society of Higher Education Journal, and has served with the International Conference on Engaging Pedagogy since its inception in 2008 having served twice as general chair. He has authored over 20 papers in computer science and pedagogy and is a member of the ACM Special Interest Group on Computer Science Education.

Catherine Mooney received the PhD degree in Computer Science, from University College Dublin, Dublin, Ireland in 2008, followed by a four year post-doctoral position in clinical bioinformatics with Prof Denis Shields at UCD. She is currently a Research Fellow in the Physiology and Medical Physics Department, Royal College of Surgeons in Ireland. She has published more than 20 peer reviewed research articles and is a recipient of a prestigious Science Foundation Ireland Advance Award. Her research focuses on identifying biomarkers for babies at risk of seizures with the goal of applying these discoveries towards the development of point-of-care diagnostics.