Identifying Measures that Represent the Variance in Novice Programmers' Code

by

Ian Probyn-Smith

A Thesis
presented to
The University of Guelph

In partial fulfilment of requirements
for the degree of
Master of Science
in
Computer Science

Guelph, Ontario, Canada

ABSTRACT

IDENTIFYING MEASURES THAT REPRESENT THE VARIANCE IN NOVICE PROGRAMMERS' CODE

Ian Probyn-Smith
University of Guelph, 2023

Advisor:
Judi McCuaig

Motivation for this work is to improve automated feedback, producing individualized feedback for novice programmers. Metrics (industrially) are used to discern software attributes for multiple objectives, including quality assessment, characterizing source code, or runtime software attributes. Using software metrics to analyse software developed by students, this research aims to facilitate self-reflection and self-assessment. The research presented examines relationships between statically calculated characteristics of student C code collected over 3 semesters. Effort is taken to include a wide variety of measures. Measures showing high variation are selected for use in metrics. Statistical dimension reduction is conducted to simplify observing consistent patterns across and within assignments. Software metrics can be intended to open a multitude of perspectives evaluating programming associated academic parties. Advantages include assessing student code in greater volume and presenting results for third party analysis. This opens further possibilities of evaluation and comparison of students as well as educators.

Acknowledgements

I would like to thank Dr. Judi McCuaig, my advisor for this thesis. Judi inspired me to pursue my interest in the field of computer science and welcomed me to study the subject. She made the completion of this thesis possible and guided me toward learning far beyond the scope of this research. I would like to thank Dr. Denis Nikitenko for his work as the second member of my advisory committee, and also Dr. Dan Gillis who was present at many stages of this research including the defense committee of the thesis. Additional current and retired members of the School of Computer Science faculty I would also like to thank include (not exclusively): Dr. Blair Nonnecke, Dr. Mark Wineberg, Dr. Gary Grewal, Dr. Deborah Stacey, Dr. William Gardner, and Dr. Joseph Sawada. Other University of Guelph Staff members that I would like to thank include Wendy Walsh, Jennifer Hughes, and Janice Ilic. John Harmer's research, conducted while he was a student of Judi McCuaig, is appreciated as the catalyst for this research. Finally I would like to thank for motivation, advice, and matters of personal assistance: my brother, Merrick Probyn-Smith; my girlfriend, Elizabeth Infante; my parents, Elizabeth and Noel Probyn-Smith; and Dr. Sherief Marzouk and Dr. Peter Tai from the Toronto Western Hospital.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

This research is situated in the field of automated feedback. The motivation for this work is to improve the ability of automated feedback systems to produce individualized feedback for novice programmers. The goal of this thesis is to explore the use of software metrics as an enabling technology for creating automated feedback for higher education. In higher education software metrics can be used in multiple ways. Some applications which have been suggested include assessment of students, teachers, and schools. The focus of the research presented in this thesis is to facilitate self-reflection and self-assessment, by using software metrics to analyse software developed by students.

Barker et al. (2019) list multiple examples of previous studies from multiple fields, which demonstrate that students perform better when receiving immediate feedback. With the current post secondary class sizes, it is nearly

impossible to individualize feedback, especially when provided in a timely manner. For this reason, a system to provide automated feedback could be helpful to students. Most existing systems for automatically generated feedback use traditional software metrics.

Traditional metrics are most often based on measures related to lines of code, and are the most common software metrics (Hall and Fenton, 1997; Molnar. et al., 2019; Fenton and Neil, 1999). The typical metrics are highly correlated with the size of the software which can lead to size being unfairly important when assessing the quality of code (Fenton and Neil, 1999). Relationships between different software metrics have been investigated with varying degrees of success.

Historically, as implemented in American Telephone and Telegraph Company (AT&T) Bell Laboratories, software metrics have been seen to provide insight and clarity about the quality of software development (Inglis, 1986). Some common uses have been: assessment of the productivity of workers, estimating the time for development, and estimating software quality. If software metrics are to be used, it is important to have standard validity criteria. Researchers largely disagree on the validity parameters of software metrics (Kitchenham, 2010). Although empirical validation research was widely collected, efficacy was not properly tested (Kitchenham, 2010).

Humans grading student code can often notice obvious attributes of poorly written code. When flaws are obvious it is easy to provide feedback about the noticeable issues, but human graders are not always consistent and cannot

provide adequate feedback when class sizes are very large. This research explores the possibility of using software metrics for examining characteristics of code with differing quality to identify the quality of student code.

1.2 Research Questions

This thesis presents exploratory research examining the relationships between statically calculated characteristics of student C code.

This research will explore the following research questions:

1. What are the main characteristics of novice code that contribute to the differences between submissions?
2. Are there characteristics of student code which can be used to create an assessment profile to assist in providing formative feedback to students?

1.3 Outline

This thesis is organized into five chapters. Chapter 2 begins with a history of the use of software metrics. This chapter continues with examples of typical uses of software metrics. Following the examples is a discussion about quality of feedback. This chapter concludes with a discussion about the relevance of feedback to software quality in higher education. The method of analysis is presented in chapter 3. This is followed by chapter 4 which presents data, and overviews the results of the principal component analysis (PCA) test outlined in chapter 3. The implications of the analysis and potential future directions of research are presented in chapter 5.

Chapter 2

Background

The work in this thesis builds on previous work in software metrics, and in automated feedback for students learning to program. The data sample was collected from university students. Previous work on samples from the same source was conducted by Harmer (2019). This chapter explores the history of software metrics, and the development of software metrics for use in education.

2.1 Software Metrics

Harmer (2019) defines metrics "as measures of quantitative elements of a given subject". With this definition applied to software as a "software metric", the beginning of software metrics can be considered as whenever a quantifiable attribute of software was first noticed or planned. Software metrics are used in an effort to discern attributes of software for a wide range of objectives. Some examples include quality assessment, and characterizing source code or

runtime attributes of the software. The broad scope of the definition can make the term misleading (Fenton and Neil, 1999).

Fenton and Neil (2000, 1999) estimate the beginning of active software metrics as the mid 1960s. The first book they identify dedicated to the subject is "Software Metrics" by Gilb, which they note is published in 1976, much later than their estimated beginning date.

Fenton and Neil (1999) propose the most significant generic benefit of information collected using software metrics is support for "managerial decision-making during software development and testing". They claim that usage in the late 1960s was largely based on size measures such as counts of code lines in effort to measure programmer productivity, and errors per thousand lines of code in assessment of program quality.

Early goals common in the software metrics of the mid 1970s sometimes target a model of measurements applicable to all programming languages (Fenton and Neil, 2000). Instead, findings in the 1970s have been presented showing that such a model can not be consistently constructed to be effective between different languages used for programming. With a lines of code based metric there is a problem comparing effort, functionality, or complexity between high level languages and assembly (Fenton and Neil, 2000). Molnar. et al. notice in 2019 a general model of software quality based on metrics has not yet been created.

Between the 1970s and 1990s Nuñez-Varela et al. (2017) find the most often studied software metrics to be lines of code, McCabe Cyclomatic Complexity, and Halstead Software Science Metrics. In the 1980s and 1990s Voas

and Kuhn (2017) observe numerous selections of software testing tools, however they describe quality and usability of the tools is often low. In 1993, Shelley observed low attention to accuracy. During the 1990s much of the attention of research began to target object oriented metrics (Nuñez-Varela et al., 2017), and the target shifted from process improvement to project management (Shelley, 1993).

Although the presence of software metrics had increased both industrially and academically by 1999, the metrics used industrially were not related to academic research activity at the time (Fenton and Neil, 1999). Fenton and Neil (1999) believe a reason for the lack of collaboration is low relevance of the research to industrial needs, with academic research targeting projects of insufficient size. Fenton and Neil (1999) describe response to negative influences such as software problems, or satisfaction of a third party assessment as primary reasons for industrial use of software metrics.

Fenton and Neil (2000, 1999) promoted use of causal modelling (particularly Bayesian nets) with simple software metrics, because they believed the regression models being used with software metrics did not provide information suitable to supplement managerial decisions. They noted poor implementation and little development of the type of metrics used industrially by 2000. The same size based metrics observed from the late 1960s were still the basis of industrial activity with software metrics, and techniques sometimes used had been considered invalid for 20 years (Fenton and Neil, 2000, 1999).

Shelley (1993) states simple metrics should be used until properly understanding the intended measures. They suggest simplicity to prevent later

difficulty, and identify goal setting, as well as analysis and definition of measures before beginning a project as the most important component of software management. Hall and Fenton (1997) claim that without clear goals, failure is near inherent for metrics programs. Basili and Rombach (1988) propose the use of a scheme to ensure goal driven metrics. Fenton and Neil (1999) however, observe some criticism of Basili and Rombach's "top-down approach".

A mapping study by Kitchenham (2010) selected papers for their study from between 2000 and 2005. Fault based evaluation and object oriented metrics were common in the majority of papers selected as software metrics evaluation studies (Kitchenham, 2010). Of the papers the study deems suitable for aggregation, fault based quality, development or maintenance effort, or size are seen as a property of the paper's target evaluation (Kitchenham, 2010).

Hall and Fenton conduct a mapping study in 1997 in which they conclude the majority of cases where industrial application of software metrics led to disadvantage, information drawn from the measures was misinterpreted or misused. Information collected from the metrics was often used as a scale for rating other factors, and sometimes, used for rating factors without valid relation (Hall and Fenton, 1997).

Meneely et al. (2012) declare "researchers should choose criteria that can confirm that the metric is appropriate for its intended use. [They] conclude that metrics validation criteria provide answers to questions that researchers have about the merits and limitations of a metric". The Kitchenham (2010) study, however, finds problems in most empirical validation studies, claiming

invalid empirical validations are often used, and previous research is overlooked (Kitchenham, 2010).

It is important to consider scaling results of static size based metrics. Gil and Lalouche (2017) conclude in their study that there are many metrics with stable correlations applicable to multiple projects. They find the metrics that are considered valid when applied to multiple projects have a correlation with size. Once the information collected by a metric has been controlled for the confounding factor of size, however, the metric is no longer valid (Gil and Lalouche, 2017).

2.2 Uses of Software Metrics

Fenton and Neil (2000) emphasise strong significance of cases using software metrics to assist management decisions. Estimating the time and effort required to develop software can also assist estimations of development costs. Knowing the productivity of developers has potential usefulness in assigning tasks of known scale to different people. If quality is measured, further inference can be made about the usefulness of a developer. Certain detected attributes can also be presented as indicators of quality to assist in marketing. Due to the potential observed, multiple projects have been conducted previously making these sort of efforts. Some early examples of industrial software metrics can be seen in AT&T and Motorola (Musson, 1994), as well as Hewlett-Packard (HP) (Shelley, 1993).

2.2.1 Effort Estimation

One main use case is estimating time and effort to develop software. Sheela (2017) declares the total cost of software to be 60 - 80 % maintenance costs. In Sheela's 2017 research on application of cognitive complexity metrics to aspect oriented programming models, they find the metrics supplementary to accuracy in prediction of maintenance effort.

Lind and Vairavan (1989) investigate relation of software metrics to effort. They found size correlates with effort, but complexity is more closely related to effort than size. Also, characters in comments were related to effort, likely because comments are more commonly included with complex code (Lind and Vairavan, 1989).

Mishra and Mishra (2008) suggest using software metrics for construction of estimation models to predict staffing required for object oriented programming projects. They estimate impact to relative effort of certain factors on categories of object class, then estimate the total staffing per category of classes. Yadav (2017) proposes a model using fuzzy logic to improve correlation between effort estimations and actual effort.

2.2.2 Employee Productivity

Effort is often made to estimate productivity in software development. Weinberg experiments show better metrics are needed for productivity than the commonly used delivered source instructions per man hour (Boehm and Papaccio, 1988). They note that use of better programmers, that make less

errors, increases productivity. Measuring productivity is conducted by repeatedly measuring software produced during times of development. The software is most often measured by scales such as "Halstead Effort", lines of code, or other measures that are usually size based. If implemented correctly, intended benefits include assistance in management decisions (Chidamber et al., 1998), such as task distribution. HP implemented a company wide initiative to improve productivity, applying software metrics to their development (Grady, 1992). Hall and Fenton (1997) report that software metrics have often been used inappropriately in industrial initiatives, due to misunderstandings of the meaning of measurements, or distrust of the information from results (Hall and Fenton, 1997; Fenton and Neil, 2000). Collection initiatives can sometimes be expensive, and if results are not used correctly, the intended benefit will probably not be achieved (Hall and Fenton, 1997).

2.2.3 Software Quality

Quality assessment is often a target of industrial software metrics. Customers typically take interest primarily in software quality, and the metrics are used in an effort to convey details of the quality of the software (Fenton, 2015).

Fenton and Neil (2000) describe reasons for invalidity of software metrics based on size, or complexity based regression. In 2018, Omri et al. conduct this type of research, with an objective to use complexity as an indicator of software defects. Omri et al. however, find their method to be an effective predictor of defects.

Daskalantonakis follows a usage implemented by Motorola in 1992 for a company wide software evaluation. From a series of goals, questions with answers allowing achievement of the goals were determined, then answered using a metric. Daskalantonakis (1992) concluded benefits from the evaluation particularly when targeting use of feedback for software improvement, and believed a wider range of metrics may have led to greater benefits.

Usage of software metrics in quality assessment by AT&T is reported in a 1986 study by Inglis. The evaluation was targeted at measuring quality using "process based rather than product based" (Musson, 1994) assessment. Completion of multiple repairs was seen as both an example of either reliable, or badly developed software. This leads to conclusion that inference from metrics may not be clear.

Detection of "code smells" is attempted by Ahmed et al. (2017), using meta-model and machine learning based models in addition to software metrics. They note that more connections increases likelihood of problems when changes are made.

2.3 Software Metrics in Higher Education

Software metrics can be intended to open a multitude of perspectives evaluating programming associated academic parties. The code written by students can be assessed in greater volume. Results can be offered for analysis by third parties, including those intending to evaluate and compare the students and the educators.

Williamson (2019) describes effects with the commercialization of schools, where political and financial interests show presence in software used for evaluation of the metrics. Such factors can introduce a bias towards an intended result upon evaluation or presentation. It is important to be aware of what a metric truly represents.

In a 1996 article, McConnell addresses the importance of active learning in education, and observes that while active learning has been implemented when teaching many subjects, it has not yet been integrated effectively enough into computer science education. In a 50 minute lecture they note a trend of concentration decline in passive students after periods of 10 - 15 minutes. As a method to help overcome this issue, they suggest operations including pair discussion at intervals to resolve misunderstandings, further sharing results of the discussion between some student pairs. Alternatively they suggest tasks completed by small groups. A possible shortcoming of McConnell's suggestion however, arises when the students are divided. Feedback about misunderstandings of other students will only be available when a minimum of one student within each group properly understands the target material sufficiently to identify and assist with the misunderstandings of others.

Automated feedback offers a greater assurance of consistency when constructing feedback for tasks assigned to students (McConnell, 1996). Jurado et al. (2012) believe their research assists in providing improved automated feedback in assessment of students. They propose a method in which an ideal algorithm authored by a teacher is subjected to measure by software metrics. The metrics from the algorithm they deem as ideal are then compared using

fuzzy logic, with the same metrics applied to the work of students. They describe their results as "successful enough" to conclude a potential effectiveness of their technique.

In an effort to avoid subjectivity when evaluating programming competence, Hung et al. (1993) propose using automated feedback to evaluate skill, complexity, style, and efficiency in programming of students. They selected lines of code to measure skill, McCabe's cyclomatic complexity metrics for complexity, execution times for efficiency, and a subset of a style metric proposed by Rees (1982) to evaluate style. They believe their assessment method is effective at classifying the quality of a programmer.

Abegaz and Spence (2019) address previous research, claiming three fundamentals in feedback messages. The receiver of the message must be alerted, informed of the condition's nature, and directed toward correction. Ott et al. (2016) and Huisman et al. (2019) both address a formative model proposed by Sadler (1989) claiming benefit to students when feedback provides information about an expected standard or goal, comparison of their work to the expectation, and information about how to bring their work closer to the expectation. In both perspectives, corrective information is seen as the lacking component (Ott et al., 2016; Abegaz and Spence, 2019).

Possible feedback weaknesses identified by McCarthy (2017) include insufficiency in quality, detail, constructive criticism, consistent availability, and primarily association to assessment criteria. Ott et al. (2016) describe error messages as often cryptic, misleading, or requiring further tutor assistance.

McCarthy (2017) state quality work is less likely from students with poor understanding of criteria.

Abegaz and Spence (2019) observe the importance of feedback in providing a path of communication about task success or failure. McCarthy (2017) regard assessment as crucial to learning in higher education, and feedback as crucial to assessment. Feedback is identified by Ott et al. (2016) as a primary influence to student achievement and learning. Ott et al. (2016) specifically remark about a 2009 analysis by Hattie which placed feedback within the top 10 of 138 considered influences on student achievement.

Studying peer feedback in 2019, Huisman et al. found qualitative and quantitative feedback more effective than quantitative feedback alone. Weaker students may view grades as discouraging, as a measurement of their failure (Ott et al., 2016). Ott et al. (2016) argue that with a score, meaning is important to provide information about available resources and needed revisions.

Ott et al. (2016) and McCarthy (2017) mention a similar subset from conditions proposed by Gibbs and Simpson (2005) addressing quality feedback due to distaste towards assessment, its associated expenses, and ineffectiveness. The conditions selected by both Ott et al. (2016) and McCarthy (2017) for feedback with a positive effect indicate feedback should be frequent, detailed, performance focused, appropriate to level of understanding, and timely to within the scope of relevance.

According to Shute (2008)'s definition, feedback is formative when "intended to modify (...) thinking or behavior to improve learning". They

list preferred attributes of formative feedback as "nonevaluative, supportive, timely, and specific". The important attributes of formative feedback identified by McCarthy (2017) are feedback both with qualitative focus, and non-graded.

When a computer system displays feedback, there is an impact on the emotional state of the receiver of the feedback (Abegaz and Spence, 2019). Current common feedback methods often produce feedback with negative emotional effects such as discouragement and frustration (Abegaz and Spence, 2019). Concerned with previous research showing more than 40% of programming time wasted was due to frustrating compiler feedback, Abegaz and Spence (2019) predicted less execution attempts would be needed to fix problems when students were in a state of positive emotion. They concluded that the intended emotional nature of feedback can have an effect on programming performance, however their results showed negative emotional feedback led to less execution attempts instead (Abegaz and Spence, 2019).

Ahadi et al. (2018) try to introduce a general approach for developing a metric. They conclude a compiler error metric better fills the role usually filled by measures such as "Jadud error quotient", based on tendency to add and fix errors (Ahadi et al., 2018). Their view of previous error quotient based approaches led them to believe that while the methods did show strong correlation to midterm evaluation, data samples limited to only the first week "showed poor performance"(Ahadi et al., 2018). More simply, course scores have strong correlation to time spent resolving programming errors(Ahadi et al., 2018).

Ahadi et al. (2018)'s perspective on the attributes of a strong metric entails replicability, ease of implementation, applicability, context independence, no free parameters, minimal bias, and population independence. They believe population independence should be achieved by comparing a student's progress with their own previous learning, as they believe other comparison is more related to difficulty or teaching strategy.

There are some concerns in context independence noticed by Ahadi et al. (2018). Compiled languages are seen by Ahadi et al. (2018) as having strength compared to interpreted languages, because in comparison to interpreted languages, some types of errors can be detected more easily prior to run-time. Also, they address the irregularity of what constitutes an error compared to a warning in different languages.

Usually seen as a representation of a quality programmer, a low number of compiles was also seen to represent memorization rather than understanding of a task (Ahadi et al., 2018). This demonstrates the possibility of multifaceted associations with some metrics, emphasizing the necessity of understanding the metrics being used.

Ahadi et al. (2018) noticed an association between error to edit ratio and the difficulty of a task to a programmer. They claim automated progress measurements are closely concept related to intelligent tutoring systems.

Although Ahadi et al. (2018) view their compiler error metric to not share the same context and language dependency issues of many other metrics, there should be additional research confirming this, due to their acknowledgement of variation in the nature of errors and warnings between languages.

In summary of the articles reviewed, issues have often been found with the effectiveness of previous software metrics in identifying software code quality. One common problem is that the metrics studied are often influenced by the size of the written programs. This research considers a broad set of potential indicators of identifiable characteristics of student code.

Chapter 3

Methodology

Some sub-goals of this research are:

1. To search for characteristics detectable by software metrics that indicate differences between submissions of student code.
2. To detect characteristics of student code usable in assessment to assist in providing automated formative feedback to students.

To explore these sub-goals we first collect as many measures as possible using static analysis of the code, then reduce the number of measures using statistical approaches to dimension reduction. Finally, we observe whether the reduced measures form consistent patterns across, and within different assignments.

3.1 Data

3.1.1 General Information About the Input Data

The data set consisted of student's assignments collected over 3 different semesters. During this time 392 assignments were collected in total. The assignments were provided in C.

The main files included in the analysis are source files (.c), and header files (.h). The makefile of each submission is used only to associate results of testing for successful GNU Compiler Collection (GCC) compilation with the source and header files. Other files were included with the submissions, but due to the nature of the research studying potential feedback from software metrics, particularly in C, other files (e.g. .txt) were omitted from the research. From all submissions, a total of 1871 source and header files were included, with 1094 as source and 777 as header files.

The first dataset (DS1) consisted of 76 submissions and 439 files. The included files from DS1 were 221 source files, and 218 header files. The second dataset (DS2) consisted of 187 submissions. The analysis of DS2 included 678 files, with 492 source and 186 header files. The third dataset (DS3) consisted of 129 submissions including 381 source and 373 header files.

3.2 Method

3.2.1 Measures Collected

Student codes files are parsed using a tool built specifically for this research. This tool was created using python code and the python regular expressions library. The purpose of this tool is to detect patterns of a variety of static characteristics in the student code.

A complete list of the targets of searches and tests initially used is provided in table 3.2.1. For completeness, this includes measures that are expected to return little or no results. The first column of the table contains the name of the measure. The second column contains a definition describing the measure in this research. Each of these measures help to quantify the code in some way.

Matching the compiler used by the students while writing, the GCC compiler is used for this research. Successful compilation is measured as a distinction of the quality of code.

When examining physical lines, blank lines, and splices, these can all be considered physical attributes of the code. None of these affect the functionality of the code.

Physical lines are included as a measure since this is one of the most common software metrics, often called “lines of code” or abbreviated to “LOC” in other research. This measure is also used as an indicator of the general scale of the code files.

Readability is often considered important. Blank lines are included since this could be an indication of an effort to maintain order in the code, although in excess can also be an indication of sloppiness. Another metric which may help determine the quality of code is splices which are sometimes seen as an indication of poor quality code.

Comments may indicate quality or effort in documentation. Alternatively, comments may be of excessive length, or may just contain code that the programmer did not use or clean.

Measures such as words, keywords, operators, spaces, and various forms of numbers may provide information about the content of the code. Information can also be extracted using combinations of the measures. An example would be words per physical line, which may help measure density of the code.

Blocks and preprocessor operations may be indicators of organisation, with includes also helping measure organisation of code between files.

Table 3.1
Code attributes collected

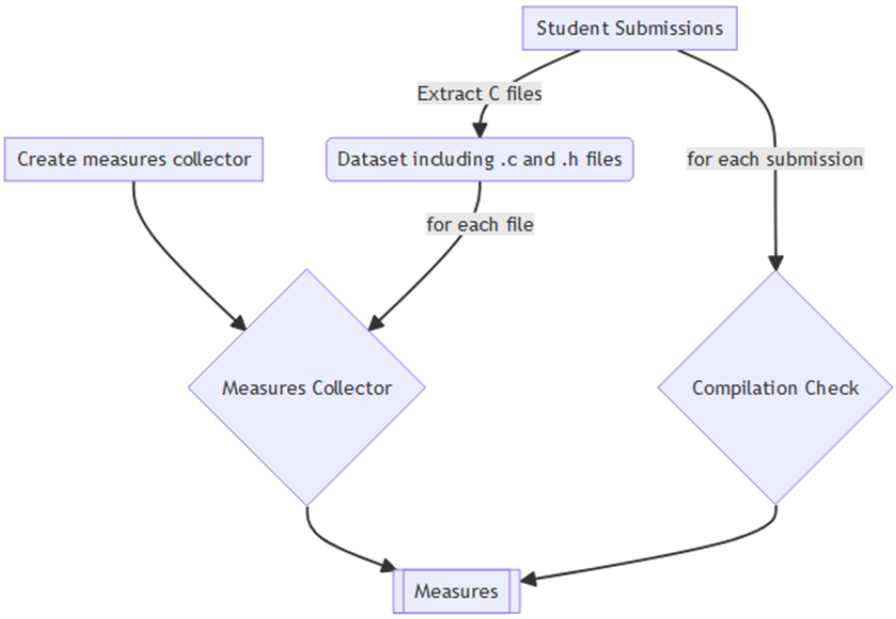
Name	Definition
successful GCC compilation	Complete compilation using the GCC compiler. Any errors occurring which terminate compilation before completion are considered as unsuccessful compilation. Warnings that do not stop compilation are permitted.
physical lines	Sections of characters within each file separated by a newline delimiter.
blank lines	Physical lines that are empty or consist only of spaces.
splices	Newlines directly preceded by a backslash.

Name	Definition
words	Character sequences consisting of only alphanumeric characters and underscores, beginning with a letter or underscore.
comments	Comments are intended to convey information to a reader of the code without impacting the operation of the code. Comments can be categorised as either single or multi line. A comment can't be started within a string or another comment.
single line comments	A series of characters beginning with two forward slash characters, and ending with the next occurring newline.
multi line comments	A series of characters that starts with "/*" and continues until "*/".
strings	A series of characters beginning with either a single or double quotation mark character, and ending with the matching quotation mark character. A string can't be started within a comment or another string.
keywords	All 32 of the standard keywords in the C programming language.
operators spacing characters	The standard operators of the C programming language. Unicode whitespace characters.
logical lines	Sections of characters within each file separated by a semicolon delimiter.
numbers	Sequences consisting of only numeric characters.
negative numbers	Numbers preceded by a "-".
decimal numbers	Numbers containing a ".".
scientific notation numbers	Two number sequences separated by an "e" or "E".
hexadecimal	To comply with C syntax, hexadecimal is expected to begin with a "0x" or "0X". In addition to numeric characters, letters between "a" and "f" are permitted, insensitive to case.
blocks	A character sequence enclosed in braces.
preprocessor conditionals	Sequences starting with "#if", "#ifdef", or "#ifndef" at the beginning of a line, and ending with "#endif".
preprocessor conditions	Lines starting with "#elif", or "#else" inside a preprocessor conditional, or starting with "#if", "#ifdef", or "#ifndef" at the beginning of a preprocessor conditional.

Name	Definition
includes	Lines that begin with “#include”.
defines	Lines that begin with “#define”.
undefs	Lines that begin with “#undef”.

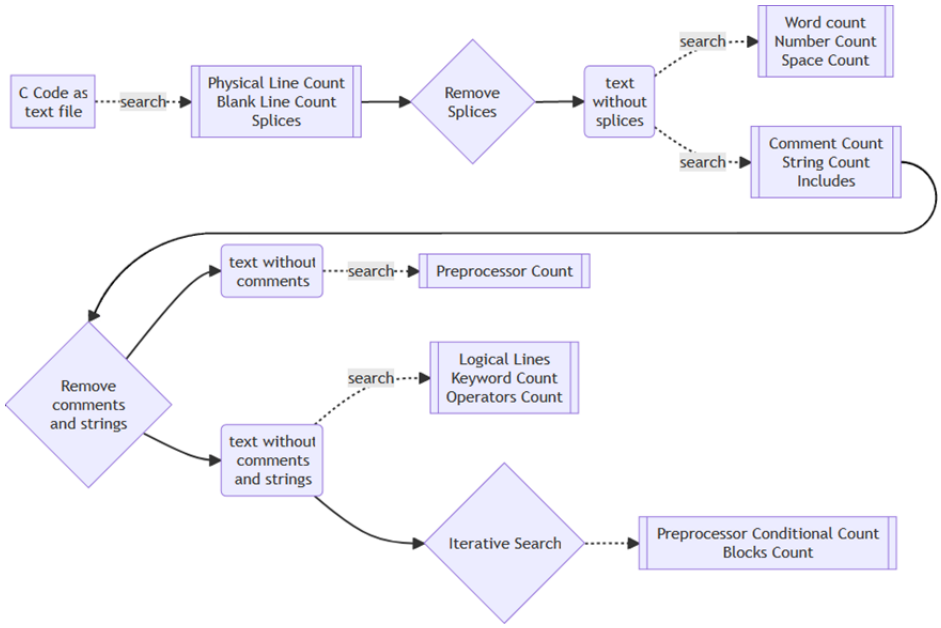
3.2.2 Collection Process

Figure 3.1
Process diagram: overall



A python program created for this research begins measurement by finding every makefile within projects from student code, and testing for successful GCC compilation. Source (“.c”) and header (“.h”) files are found next, and each is individually searched using “regular expressions” for the remaining measures. After parsing using regular expressions, the character locations of

Figure 3.2
Process diagram: measures



the beginning and ending of all pattern matches is recorded. The location records are used to develop metrics for further analysis.

3.2.3 Regular Expressions for Parsing

To generate the data sets used in this research individual files had to be parsed. This task counted the number of occurrences of multiple elements in each code file to create measures used later in the analysis. Since this task was not concerned with the speed of parsing, this research used regular expressions in python to get these counts.

Regular expressions is a common method of searching text and extracting information about attributes of the text. Available in many programming and

scripting languages, regular expressions was selected for the searches used in measures for the metrics in this research. Regular expressions can be modified easily. This allows adaptability for future uses, such as assessment of other programming and scripting languages instead of C, and modification or addition to the currently included set of software metrics being searched.

The python "re" (regular expression) module was used to search for matches with regular expressions. This library can be found at `https://github.com/python/cpython/tree/3.7/Lib/re.py` with documentation at `https://docs.python.org/library/re.html`

The primary reasons for using python are cross-platform availability, and python's named groups extension upon regular expressions. Named groups simplifies some parts of the collection process such as categorising embedded attributes of the matches. An example using named groups with python regular expressions can be seen in figure 3.3. The example in figure 3.3 detects simple physical lines, blank lines, and splices. Naming groups allows labels to be applied to groups representing certain attributes.

Figure 3.3
One line of code showing how python regular expressions named groups were used

```
 re.compile(r"(?P<physical_line>^((?P<blank_line>^\s*?)|(.*?))
((?P<splice>\\\n)|\n))", re.MULTILINE)
```

The labelled attributes allow additional metrics to be collected from sections of matches. This simplifies processes such as ensuring matching single or double quotes around strings. It is also safe to search for some attributes as

sections of matches. For example, a blank line is always going to be a physical line; it needs to qualify as blank by having no visible text.

The search of each file was conducted by finding all matches of the regular expression patterns. Measurement is taken by mapping the beginning and ending character numbers of each instance detected in the search. Within searches, some pieces of files may be excluded (for example: splicing). Adjustments are made to the search results to compensate for excluded sections, to ensure characters are mapped to the correct location in the original file. Some uncommon syntax (for example: scientific notation numbers) is not expected to be found, but will be included in some searches. Measures are further extracted by determining the distance between the starting and ending characters to determine the length of matches, or by counting the qualifying instances.

3.2.4 Collection Order

Software developed using a compiler depends on the compiler to translate the code to an executable format. In a search for patterns in code, it is important that attributes are identified correctly. Results from searches must find details in the code that match identification given by the compiler. The collection order is important because some attributes of files may interfere with matches detected by regular expression patterns. For example, splicing and comments may interrupt other code sequences. It is important to know where such potential interference exists within the files to avoid incorrectly categorizing a pattern.

3.2.4.1 File and Line Characteristics

It is important that splices are detected as early as possible. Splices ("\\n") are omitted from searches early because they can cause modification to character patterns of other syntax. Physical lines in C code ending with a backslash are considered to be spliced with the following physical line. Splices in C code are usable within all sequences, including comments and strings. Splices cause two physical lines to act in conjunction, working as a single line that ignores the backslash and following newline characters from the splice. It is important that splices do not cause sections of code to be misidentified due to the interrupting backslash and newline. Since splicing interrupts other syntax patterns but does not impact the functionality of the code, removing them simplifies the collection process in later searches.

Physical lines and blank lines are included in code files primarily to simplify readability to a person reading the code. Since physical lines and blank lines are not significant to the compilation, they can be detected simultaneously while searching for splices. Detecting splices while searching for physical lines is simple. The newline character is essential in detecting the points of separation between physical lines. If the physical line ends with a backslash character, a splice is detected. Blank lines can be detected by determining if physical lines contain only spacing characters.

To avoid the impact of splicing on searches, after splices are found, an additional string is constructed for later searches. The new string excludes all backslashes that indicate a splice, and the affected newlines. When this splice-free string gets searched, the position and length of splices in the original code

file will be used to calculate the equivalent character positions in the original code.

3.2.4.2 Strings and Words

Words, numbers, and spaces are searched next. The numbers detected are then categorised if they are negative, decimal, hexadecimal, or scientific notation.

The next search is simultaneously for comments, strings, and includes. Comments and strings are searched simultaneously because both comments and strings can impact the meaning of text in a code file. It is important that the content of comments and strings is not misinterpreted as code. Characters normally indicating the start of a comment or string do not have such functionality when contained within either comments or strings. Includes are searched at the same time because they are often associated directly with double-quoted strings. Also, for strings, it is important to check for matching quotes at both ends. All of this leads to some difficulties when searching for both comments and strings.

The code files were originally tested with GCC as the C compiler. GCC searches a code file from the beginning towards the end for the comments and strings. To indicate the beginning of a comment or string, the start indicator must not be preceded by the start indicator of another comment or string that has not yet ended. In this case the characters that normally indicate the beginning of a comment or string are considered part of the text of the earlier string or comment that has not yet ended. The direction searched is important as shown in table 3.2. If the character patterns are searched from

the beginning to the end, the colour indicates how the characters would be classified. It can be seen that reversal of the character patterns changes the classification of some characters within the pattern.

Table 3.2
Strings and comments; order and interference

```
/*/**/          Comment
/**/*/          String
//""             Other
""//
"'"'
/*"*/"
"/*"*/
```

The search conducted in this research is constructed to use similar behaviour. Comments were categorized as single or multi-line comments. As explained above, later searches omit comments, splices, and strings. These were omitted by constructing a temporary string by concatenating all sections of the string containing the code file that are not identified as one of the omitted items. The temporary string is searched using regular expressions, and after the search, locations of the omissions are used to calculate the relative position of the pattern in the original code.

After comments, splices, and strings are removed, the order of the remaining searches is not as important. Each of the remaining attributes searched for the measures will not be confused as another attribute, and will not interrupt the detection of other patterns. Preprocessor details are searched next, followed by logical lines, keywords, and then operators.

In the search for keywords, all 32 of the standard C keywords listed in table 3.3 are searched simultaneously, but the identity of each mapped keyword is separately preserved in case extra details need to be extracted later.

Table 3.3
C keywords

unsigned	signed	short	long	int	float	double	char
void	while	for	do	goto	continue	break	volatile
union	typedef	switch	struct	static	sizeof	return	register
if	extern	enum	else	default	const	case	auto

The operators are also detected in one search. Similar to the search for keywords, the identity of each instance of mapped operators is recorded. During the search, the operators are considered with an ordered priority. This is done because many operators contain multiple characters, and some sub-strings of the operator may also be operators. These operators need to be properly detected. For example, ">>" and "<<" need to be detected as "bitwise right" and "bitwise left" operators, and not two "greater than" or "less than" operators. In the order used in this research, the character patterns of all operators containing multiple characters are given higher priority than sub-strings that are also operators. The order used is listed in table 3.4.

Table 3.4
This list of C operators were searched in order left to right, top to bottom.

sizeof	++	--	->	<<=	>>=	<<
>>	<=	>=	==	!=	*=	/=
%=	+=	-=	&=	|=	^=	?:
&&	||	(	)	[	]	*
/	%	+ -	!	~	&	
|	^	<	>	.	,	=

Preprocessor details are searched next, and block details will be searched last. For metrics about code that can be nested, (for example: blocks) the search is conducted by finding only the narrowest matching components between the beginning and ending indicators of qualifying sections that are not separated by the beginning indicator. For any bracket type it is important that the pairings are correctly identified as shown in table 3.5.

Table 3.5
Matching brackets

```
{{}}          Incorrect Pair
{{}}
{{}}            Correct Pair
{  }
```

Code blocks, for example, were searched as the nearest "{" followed by a "}" without any "{" in between. Once these are found, the matches are

removed from the code, and the search is repeated in a loop until matches are no longer found. The depth of nesting is determined by the number of loop cycles which returned matches in these tests.

3.2.5 Data Collection

Once the measure collection is complete, the measures are analysed to produce a set of counts for each file. The variables representing the counts are shown in table 3.6.

Information will be calculated about the maximum, minimum, mean, and standard deviation of character quantity for physical lines, comments, and logical lines within each file. The maximum, minimum, and mean character quantity in each file will also be calculated for strings and blocks. The mean quantity of words per comment will be calculated for each file.

It is expected that many of the planned measures will rarely be observed in the code, or not detected at all. Some syntax has greater suitability dependant on context. For example, the research sample is unlikely to include any use of scientific notation numbers. For completeness, the code submissions are examined for all measures, even ones that were not expected to be present.

Metrics constructed for later analysis will mostly target measures relative to 100 physical lines of code to avoid size as a confounding factor from counted measures.

Table 3.6
Initial measures

Counted instances in single search
physical lines
blank lines
splices
words
comments
single line comments
multi line comments
comment words
comment characters
strings
words in code
keywords
operators
spacing characters
logical lines
numbers
negative numbers
decimal numbers
scientific notation numbers
hexadecimal numbers
blocks
preprocessor conditionals
preprocessor conditions
includes
defines
undefs
Counted cycles in repeated search
longest block nest depth
longest preprocessor nest depth

3.2.6 Expected Analysis

After collecting the data, analysis will be performed using R and R studio. Due to the quantity of variables, PCA will be conducted using the FactoMineR package to reduce the number of dimensions. The nature of the research, targeting enhancement of feedback, justifies preference of PCA compared to

factor analysis as the method of dimension reduction. Unlike PCA both main types of factor analysis, exploratory and confirmatory, target creation of a latent construct. The additional dimension created by the latent construct, as an unobserved factor, would have less value to this research than the component created by PCA. The component from PCA is instead impacted by other variables representing the outcome of the student's work.

When observing the results of PCA the two most common methods of determining relevant components are observing a scree plot, and observing eigenvalues ≥ 1. Observing the scree plot involves visually finding a strong point of inflection where the rate of change in the data has a sudden falloff. Since both methods are widely adopted, both will be included in this research. With these techniques in mind, the aim will be finding explanation of 70% of the cumulative variance from the PCA. This will account for the majority of the data while likely not having too many components overcomplicating the findings and leading to less useable results.

Pearson correlations will be used to conduct simple correlation analysis. This will allow basic relations between variables to be exposed. Correlation coefficients will be used to determine if similarities between components are consistent across the different data sets. Correlations sufficiently high (≥ 0.5) between two components will be used to determine if components move in a similar way between different data sets, or if the components of each are unique.

Chapter 4

Results and Analysis

4.1 Summary Results

This chapter contains results and analysis of a set of student code files consisting of assignments collected over 3 different semesters including 392 assignments provided in C. The files included source files (.c) and header files (.h).

The results from the measures initially collected are summarised in tables 4.1, 4.2, and 4.3. There are differences between the data sets in terms of means, ranges, and standard deviations. Table 4.1 shows the ranges of all the collected measures. DS3 typically has a much higher range for measures that quantify length of code. This includes measures for the number of physical lines, numbers, words, comments, total comment characters, strings, and operators. Additionally, DS3 has a larger range of the maximum length of strings, and a larger range of the amount of negative numbers. Even though

DS3 has a far larger range of physical lines, the difference in ranges of blank lines is not proportional. DS3 is the only data set where the minimum range is not zero for every measure. There are some unique ranges for DS1 and DS2 such as the number of characters in blocks, but overall, these are not nearly as pronounced as the differences that exist between the other data sets with DS3.

Table 4.2 shows the mean values of the measurement results. Similar to the ranges, DS3 has higher mean values for most measures relating to total size of the code. In terms of means and ranges, DS1 often appears to have the shortest files. DS2 and DS3 appear to have not only more blocks, but also longer blocks. The means and ranges show an approximately similar picture of the data sets.

Table 4.3 contains the standard deviation of the measures. Since standard deviation is highly correlated with range, the results of standard deviation appear to have similar findings to the range. The standard deviations are usually higher in DS3 than the other data sets. This can be seen particularly well in physical lines, words, operators, spacing characters, numbers, and blocks.

4.2 Analysis

The measures used to create metrics were ones that showed variety. One example is percent of blank physical lines, where the total number of blank lines in one file ranged between 0 and 686. Variety was not consistent in all the samples. For example, there was a larger range of operators in DS3 (0-11463) than in the other data sets (DS1: 0-3475, DS2: 0-4922). The amount

Table 4.1
Range of values (minimum - maximum) measured in DS1, DS2, and DS3

Name	DS1 Range	DS2 Range	DS3 Range
successful gcc compilation	True / False	True / False	True / False
physical lines	0 - 1942	0 - 2989	6 - 4800
physical line length max	0 - 451	0 - 514	27 - 965
physical line length min	0 - 19	0 - 17	1 - 4
physical line length mean	0 - 58.39	0 - 48.05	12 - 72.10
physical line length standard deviation	0 - 48.61	0 - 80.50	11.84 - 49.89
blank lines	0 - 503	0 - 683	0 - 686
splices	0 - 6	0 - 4	0 - 1
words	0 - 4829	0 - 6879	0 - 10567
comments	0 - 339	0 - 395	0 - 573
single line comments	0 - 318	0 - 388	0 - 556
multi line comments	0 - 61	0 - 183	0 - 224
comment characters total	0 - 17013	0 - 15535	0 - 22766
comment characters max	0 - 6185	0 - 3402	0 - 5568
comment characters min	0 - 546	0 - 960	0 - 842
comment characters mean	0 - 547.50	0 - 1291.33	0 - 842.00
comment characters standard deviation	0 - 1070.15	0 - 1500.94	0 - 732.99
comment words total	0 - 2364	0 - 2511	0 - 3068
comment words mean	0 - 68.43	0 - 111.00	0 - 104.00
strings	0 - 172	0 - 233	0 - 551
string length max	0 - 420	0 - 463	0 - 939
string length min	0 - 24	0 - 23	0 - 27
string length mean	0 - 42.19	0 - 45.20	0 - 44.35
words in code	0 - 3644	0 - 4933	8 - 9671
keywords	0 - 586	0 - 924	0 - 1665
operators	0 - 3475	0 - 4922	0 - 11463
spacing characters	0 - 17345	0 - 44192	9 - 88444
logical lines	0 - 966	0 - 1486	0 - 2136
logical line length max	0 - 4023	0 - 5380	0 - 6049
logical line length min	0 - 392	0 - 541	0 - 501
logical line length mean	0 - 589.09	0 - 670.50	0 - 650.86
logical line length standard deviation	0 - 684.38	0 - 505.40	0 - 587.47
numbers	0 - 432	0 - 651	0 - 1157
negative numbers	0 - 31	0 - 23	0 - 76
decimal numbers	0 - 10	0 - 11	0 - 14
scientific notation numbers	0	0	0
hexadecimal	0 - 20	0	0 - 15
longest block nest	0 - 11	0 - 16	0 - 24
blocks	0 - 240	0 - 341	0 - 677
block characters max	0 - 42948	0 - 63900	0 - 100583
block characters min	0 - 433	0 - 9913	0 - 611
block characters mean	0 - 1096.91	0 - 9913.00	0 - 5875.08
longest preprocessor nest	0 - 1	0 - 1	0 - 2
preprocessor conditionals	0 - 2	0 - 71	0 - 2
preprocessor conditions	0 - 30	0 - 71	0 - 75
includes	0 - 14	0 - 10	0 - 11
defines	0 - 19	0 - 33	0 - 19
undefs	0	0	0

Table 4.2
Means of measures collected

Name	DS1 Mean	DS2 Mean	DS3 Mean
successful gcc compilation	NA	NA	NA
physical lines	306.16	381.25	438.27
physical line length max	139.08	136.91	152.45
physical line length min	1.04	1.03	1.00
physical line length mean	29.38	26.75	31.63
physical line length standard deviation	26.89	24.80	29.10
blank lines	55.91	64.48	73.30
splices	0.03	0.01	0.00
words	1045.97	1130.16	1502.43
comments	25.22	26.99	36.15
single line comments	16.60	19.85	26.08
multi line comments	8.62	7.14	10.07
comment characters total	3334.51	2724.65	4209.86
comment characters max	463.42	395.30	571.08
comment characters min	45.46	29.78	45.08
comment characters mean	171.97	149.33	169.27
comment characters standard deviation	106.52	97.41	121.93
comment words total	535.17	435.36	661.65
comment words mean	27.68	23.68	27.39
strings	19.35	28.20	32.06
string length max	32.98	41.22	42.38
string length min	5.84	4.42	6.37
string length mean	10.72	11.54	12.34
words in code	519.49	707.20	855.93
keywords	111.65	139.92	163.40
operators	481.91	674.65	843.60
spacing characters	2044.36	2984.11	3655.94
logical lines	109.32	150.24	172.34
logical line length max	632.38	601.22	781.39
logical line length min	16.65	19.35	14.04
logical line length mean	129.65	90.57	134.60
logical line length standard deviation	128.91	100.78	141.41
numbers	35.48	52.82	66.28
negative numbers	0.64	0.68	0.84
decimal numbers	0.31	0.19	0.13
scientific notation numbers	0	0	0
hexadecimal	0.05	0	0.03
longest block nest	2.71	3.79	3.17
blocks	33.68	51.54	54.77
block characters max	2069.27	3633.49	3378.91
block characters min	33.35	32.72	31.76
block characters mean	232.04	275.74	279.28
longest preprocessor nest	0.48	0.25	0.46
preprocessor conditionals	0.48	0.45	0.47
preprocessor conditions	1.28	1.29	1.28
includes	4.050	3.19	4.02
defines	0.76	0.49	0.63
undefs	0	0	0

Table 4.3
Standard deviation (SD) of measures collected

Name	DS1 SD	DS2 SD	DS3 SD
successful gcc compilation	NA	NA	NA
physical lines	283.68	365.64	527.89
physical line length max	49.63	58.75	70.10
physical line length min	0.86	0.65	0.11
physical line length mean	8.06	6.41	7.81
physical line length standard deviation	7.36	6.32	6.87
blank lines	53.65	64.54	77.00
splices	0.38	0.18	0.04
words	686.70	917.79	1384.24
comments	31.51	42.88	50.43
single line comments	30.06	40.90	48.36
multi line comments	8.19	12.63	13.70
comment characters total	2731.78	2547.21	3195.20
comment characters max	432.49	364.45	465.75
comment characters min	66.11	69.53	67.55
comment characters mean	149.36	150.32	142.24
comment characters standard deviation	93.79	104.23	87.86
comment words total	454.08	412.69	492.81
comment words mean	25.92	23.99	24.45
strings	29.90	38.48	58.02
string length max	49.24	51.13	75.49
string length min	3.76	4.09	4.50
string length mean	3.95	5.57	4.75
words in code	594.52	725.17	1278.50
keywords	107.09	135.16	215.15
operators	615.71	740.81	1410.95
spacing characters	2391.20	4418.43	6613.02
logical lines	134.79	173.68	266.56
logical line length max	363.96	420.17	502.40
logical line length min	36.36	38.88	32.09
logical line length mean	103.76	66.80	95.16
logical line length standard deviation	88.98	66.39	85.22
numbers	64.82	83.00	140.32
negative numbers	2.63	2.42	4.16
decimal numbers	0.80	0.79	0.74
scientific notation numbers	0	0	0
hexadecimal	0.96	0	0.57
longest block nest	2.41	2.95	2.93
blocks	44.14	55.55	89.47
block characters max	4334.51	7005.42	8654.95
block characters min	41.56	380.86	55.19
block characters mean	184.45	471.01	312.99
longest preprocessor nest	0.50	0.43	0.50
preprocessor conditionals	0.51	3.17	0.50
preprocessor conditions	3.07	4.59	4.33
includes	2.12	2.07	2.08
defines	1.73	2.13	1.12
undefs	0	0	0

of variation in these measures led them to be of interest for creating metrics. There was no specific threshold set due to the differences in nature of the measures collected, and each having different criteria for what may constitute interesting variation.

The metrics constructed, and a definition of each, can be seen in table 4.4. Metrics were calculated separately for each file in each data set. From the collected measures described in the plan, multiple measures returned no useful results. Some examples of this were:

- Scientific notation was never used for numbers in any of the samples
- Defines (#define) were used but undefines (#undef) were never used
- Splicing was rare
- Multiple variables showed almost no variation

For this research, count based measures are usually scaled. The scaling is done to measure density relative to one hundred physical lines. Size is a common factor in other investigations, and a potentially confounding factor for counted measures. Scaling these measures allows for the ones that might conflict, such as number of spacing characters and number of operators to both be included without the risk of them confounding each other. Used to scale other measures in this research, the size of the file as determined by the count of "physical lines" was not included in the PCA.

The comparative ratio was selected as one hundred rather than the more traditional one thousand physical lines, so measures counting instances that still qualify as complete physical lines can be considered as percentages. A blank line is always a physical line, so this can easily be interpreted as a percentage. To get a frequency instead of a count, the number of instances

Table 4.4
Metrics used in the analysis

Name	Definition
Percent Blank Physical Lines	A count of all the physical lines containing nothing or entirely blank space characters, per hundred physical lines.
Physical Lines Max Length	The highest amount of characters between a newline and the following newline.
Logical Lines Per Hundred Physical Lines	The total counted sections of code separated by a semicolon character per hundred physical lines.
Logical Line Max Length	The highest amount of characters between a ";" and the following semicolon.
Logical Line Min Length	The lowest amount of characters between a ";" and the next ";".
Code Words Per Hundred Physical Lines	Total counted sections mapped as words that were not contained within strings and comments, per hundred physical lines.
Comments Per Hundred Physical Lines	Counted sections mapped as comments per hundred physical lines.
Words Per Comment	Sections mapped as words also mapped within comments, divided by the total number of comments.
Comment Characters Max Length	The longest distance between the starting character in a comment and the ending character in the same comment.
Comment Characters Min Length	The shortest distance between the starting character in a comment and the ending character of the comment.
Strings Per Hundred Physical Lines	Counted sections mapped as strings per hundred physical lines.
String Characters Max Length	The highest amount of characters between a quotation mark that depicts the beginning of a string, and the matching quotation mark depicting the end of the string.
String Characters Min Length	The lowest amount of characters between a quotation mark that depicts the beginning of a string, and the matching quotation mark that depicts the end of the string.
Spacing Characters Per Hundred Physical Lines	Unicode spacing characters such as spaces, tabs, and carriage returns; counted, and then scaled per hundred physical lines.
Blocks Per Hundred Physical Lines	The total amount of blocks per hundred physical lines. Blocks nested any level within other blocks are included.
Longest Block Nest	The highest amount of unended blocks at any point.
Block Characters Max Length	The longest distance between the braces mapped as starting and closing the same block.
Block Characters Min Length	The shortest distance between "{" and the following "}".
Keywords Per Hundred Physical Lines	The quantity of keywords used per hundred physical lines.
Operators Per Hundred Physical Lines	The quantity of operators used per hundred physical lines.
Numbers Per Hundred Physical Lines	The quantity of numbers used per hundred physical lines.
Includes	The counted number of lines that begin with "#include".
Preprocessor Conditions	A count of the total preprocessor conditions within a file. This includes "#if", "#ifdef", "#ifndef", "#elif", or "#else".
Longest Preprocessor Nest	The highest amount of unended preprocessor conditionals at any point.

of some measures is divided by the number of total physical lines and then multiplied by one hundred.

The list of counted measures scaled per hundred physical lines in the PCA analysis include:

- Blank lines
- Logical lines
- Code words
- Comments
- Strings
- Spacing characters
- Blocks
- Keywords
- Operators
- Numbers

There are multiple types of measures which are created. In table 4.4 these are separated by which category they fall into by colour. Red measures are counted measures which have been scaled to the number of occurrences per 100 physical lines of code. One example of a red measure is "strings per hundred physical lines". Orange measures are counted measures. These have not been scaled to the length of the code. "Includes" and "preprocessor conditions" are orange measures, and considered directly as counted measures. These orange measures are not scaled due to the low counts that these measures usually contain.

Blue measures represent a count of characters, and teal measures represent a count of words. Only "words per comment" is teal. Measures of maximum and minimum length are coloured blue, and usually showed more interesting variation than the mean or standard deviation values. One example of a blue measure is "physical line max length". These are also not scaled, they are

the counted number for the minimum or maximum character length. The final colour category of measures is violet which are nesting depth measures. There are two examples of violet measures, "longest block nest" and "longest preprocessor nest". Both of these show the farthest depth of nested structures of that classification.

Probably due to the usage of empty blank lines, "physical line min length" was usually 0 as expected, and was not included in the PCA. When calculating lengths, characters considered to indicate the beginning or end of the section are not included in the determined lengths. Characters excluded from calculated lengths:

- "/*", "*/", and "//" are not included in the length of comments.
- Quotation marks are not included in the length determined for strings.
- Braces are not counted in the length of blocks.
- The length of physical lines does not include the newline.
- Logical line length does not include the semicolon.

4.2.1 Principal Component Analysis

A PCA was conducted to find patterns in code of the files from the assignments in the samples. This analysis was conducted at the aggregate level. Individual differences between students were not analysed. Patterns between files within each total data set are the focus of the results of the PCA.

The analysis was conducted in Rstudio using R version 3.5.2 and the "FactoMineR" and "factoextra" packages. When selecting dimensions to keep, the first things examined were the scree plots shown in 4.1, 4.2, and 4.3. Figure 4.1 represents DS1, figure 4.2 represents DS2, and figure 4.3 represents DS3.

Figure 4.1
DS1 scree plot. This scree plot shows the eigenvalues for up to 10 components of DS1.

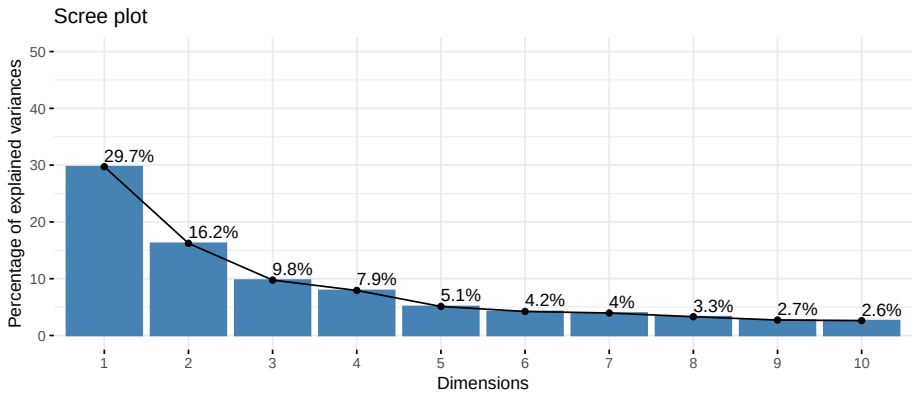


Figure 4.2
DS2 scree plot. This scree plot shows the eigenvalues for up to 10 components of DS2.

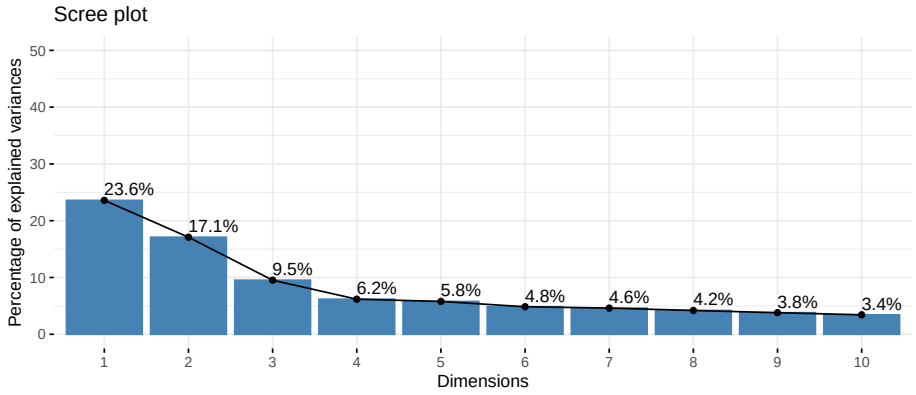
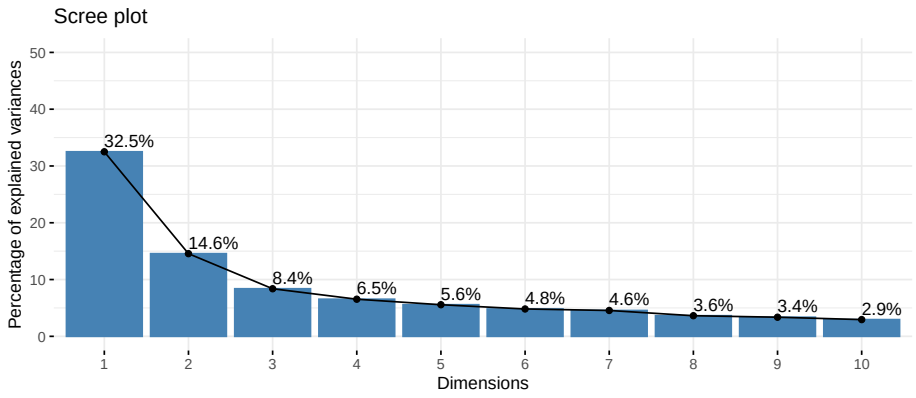


Figure 4.3
DS3 scree plot. This scree plot shows the eigenvalues for up to 10 components of DS3.



In all of these graphs, the vertical axis shows the sum of squared residuals related to the included number of dimensions depicted by the values on the horizontal axis. The downward slope in all the figures is representative of the percentage of variance explained by the included components, because as components are added, more variation is explained. A clear inflection or falloff would demonstrate a strong change in the quantity of information exhibited by subsequent included components. In these graphs there is no clear "elbow", and selecting dimensions this way without such an indication can be considered subjective. A different common method, selecting dimensions with eigenvalues >1 was used instead.

Table 4.5 is constructed of information from DS1, table 4.6 has information from DS2, and table 4.7 has the information from DS3. Each of these tables is designed to display the amount of variance explained by the components created from the PCA. The components are sorted from the highest to lowest

significance of impact to the variance, essentially ordered from most to least important in the analysis of the PCA results for this study. The examination is simplified by including the cumulative variance of components in the same order, so the smallest number of components can be selected that account for the greatest degree of the variance. Eigenvalues of the components are listed as well, since these values are required for the chosen method of selecting components in this research.

For DS1, 6 dimensions with an eigenvalue $\geq$1 qualify, which account for 72.960% of the cumulative variance. This can be seen in table 4.5. Component 6 of DS1 has an eigenvalue of 1.014, which is above the qualifying threshold of $\geq$1, and explains 4.225% of the variance in DS1. The first component alone accounts for 29.718% of the variance, but the target threshold of $\geq$70 is met by including the subsequent 5 as well, which is sufficient to explain the majority of the data of DS1.

In table 4.6 (DS2), it can be seen that there are instead 8 components with eigenvalues $\geq$1. All 8 components with eigenvalues $\geq$1 account for 75.799% of the cumulative variance. To pass the $\geq$70 threshold for cumulative variance, only 7 components are required, accounting for 71.611% of the variance. If only 6 components are selected, 67.003% of the cumulative variance is explained.

Table 4.7 shows a difference in the amount of qualifying eigenvalues of vectors compared to both other tables. There are 7 components with eigenvalues $\geq$1 in DS3, and these components account for 76.909% of the cumulative variance. Similar to DS1, only 6 components are required to pass the threshold

Table 4.5
DS1 eigenvalues. This table shows the complete list of eigenvalues, variance, and cumulative variance for all components of DS1.

Dimension	Eigenvalue	Variance Percent	Cumulative Variance Percent
1	7.132	29.718	29.718
2	3.894	16.227	45.944
3	2.341	9.753	55.697
4	1.903	7.930	63.627
5	1.226	5.108	68.734
6	1.014	4.225	72.960
7	0.949	3.956	76.916
8	0.792	3.302	80.218
9	0.652	2.718	82.936
10	0.627	2.614	85.551
11	0.574	2.393	87.943
12	0.534	2.224	90.168
13	0.483	2.013	92.180
14	0.403	1.677	93.858
15	0.302	1.259	95.117
16	0.235	0.979	96.097
17	0.223	0.929	97.026
18	0.167	0.698	97.724
19	0.155	0.645	98.369
20	0.115	0.480	98.849
21	0.106	0.441	99.290
22	0.081	0.339	99.628
23	0.065	0.272	99.900
24	0.024	0.100	100.000

of $\geq$70 of the cumulative variance of DS3, with the 6 components accounting for 72.356% of the cumulative variance.

Since we are looking for similarities in contributions to all 3 data sets, the same number of dimensions were selected from all 3 sets. Since all sets have $\geq$6 qualifying dimensions, 6 dimensions were examined for this research. At 6

Table 4.6
DS2 eigenvalues. This table shows the complete list of eigenvalues, variance, and cumulative variance for all components of DS2.

Dimension	Eigenvalue	Variance Percent	Cumulative Variance Percent
1	5.660	23.581	23.581
2	4.102	17.093	40.674
3	2.286	9.524	50.198
4	1.482	6.175	56.373
5	1.388	5.783	62.156
6	1.163	4.847	67.003
7	1.106	4.607	71.611
8	1.005	4.188	75.799
9	0.912	3.800	79.599
10	0.822	3.426	83.025
11	0.749	3.120	86.145
12	0.571	2.381	88.526
13	0.526	2.192	90.718
14	0.391	1.629	92.347
15	0.372	1.552	93.899
16	0.309	1.289	95.188
17	0.254	1.060	96.249
18	0.204	0.852	97.101
19	0.176	0.735	97.836
20	0.130	0.542	98.378
21	0.127	0.529	98.907
22	0.106	0.441	99.348
23	0.105	0.436	99.784
24	0.052	0.216	100.000

dimensions, 67.003% of the cumulative variance is accounted for in DS2, and 72.356% in DS3.

Contribution values $\geq$10% were selected for a first analysis. There were not too many contribution values $\geq$10% to complicate interpretation, but the first 6 dimensions of every data set had at least one factor contributing more

Table 4.7
DS3 eigenvalues. This table shows the complete list of eigenvalues, variance, and cumulative variance for all components of DS3.

Dimension	Eigenvalue	Variance Percent	Cumulative Variance Percent
1	7.800	32.500	32.500
2	3.494	14.559	47.059
3	2.011	8.379	55.438
4	1.569	6.537	61.975
5	1.335	5.564	67.539
6	1.156	4.817	72.356
7	1.093	4.553	76.909
8	0.871	3.629	80.538
9	0.808	3.366	83.904
10	0.707	2.946	86.850
11	0.588	2.449	89.299
12	0.518	2.159	91.457
13	0.360	1.499	92.956
14	0.352	1.465	94.421
15	0.294	1.224	95.645
16	0.251	1.044	96.689
17	0.150	0.626	97.315
18	0.149	0.620	97.935
19	0.136	0.567	98.502
20	0.111	0.463	98.965
21	0.090	0.375	99.340
22	0.082	0.341	99.681
23	0.055	0.229	99.909
24	0.022	0.091	100.000

than 10%. All of the first 6 dimensions of all 3 data sets were compared with every dimension of the other data sets.

The selected 6 components of each data set are displayed across the top of tables 4.8, 4.9, and 4.10. The variables used to create the components in the PCA are listed down the side of the table, and the amount each variable

contributes to the created components is shown. These listed values allow for examination of the potential similarity or uniqueness of the components.

Table 4.8
DS1 contribution values. This table shows the contribution values of the first 6 components from the PCA for each of the variables constructing DS1.

	1	2	3	4	5	6
Percent Blank Physical Lines	2.026	3.375	0.047	1.495	11.121	0.343
Physical Lines Max Length	0.060	15.904	0.562	0.041	0.667	4.223
Logical Lines Per Hundred Physical Lines	7.848	0.337	0.511	14.091	0.528	4.689
Logical Line Max Length	3.379	8.934	0.420	0.355	0.306	2.499
Logical Line Min Length	0.347	2.010	0.938	19.934	5.093	22.269
Code Words Per Hundred Physical Lines	8.536	4.492	0.088	0.074	6.566	0.083
Comments Per Hundred Physical Lines	2.226	0.050	19.585	4.595	0.284	9.262
Words Per Comment	4.547	8.093	10.223	0.011	1.428	1.481
Comment Characters Max Length	3.407	7.354	0.072	1.408	2.404	0.792
Comment Characters Min Length	3.825	1.251	2.867	10.699	12.387	2.522
Strings Per Hundred Physical Lines	1.795	5.520	0.130	12.273	0.296	1.611
String Characters Max Length	3.716	2.142	3.732	2.181	0.007	6.002
String Characters Min Length	7.042	4.746	3.298	0.451	4.484	0.024
Spacing Characters Per Hundred Physical Lines	0.074	11.739	4.310	0.507	8.660	0.007
Blocks Per Hundred Physical Lines	8.544	3.258	1.449	1.344	0.255	0.663
Longest Block Nest	7.922	6.961	0.200	0.028	1.750	0.040
Block Characters Max Length	4.003	3.469	7.619	3.014	1.615	1.915
Block Characters Min Length	1.058	0.949	13.328	10.443	0.003	1.971
Keywords Per Hundred Physical Lines	2.516	5.936	0.283	1.197	26.855	0.0002
Operators Per Hundred Physical Lines	11.199	0.076	0.440	0.086	5.339	0.109
Numbers Per Hundred Physical Lines	5.440	0.419	5.038	2.959	0.882	4.663
Includes	0.840	0.031	12.242	6.810	0.436	7.717
Preprocessor Conditions	0.548	1.031	8.861	5.640	5.330	26.770
Longest Preprocessor Nest	9.101	1.923	3.755	0.362	3.306	0.342

This analysis shows similarities mostly between DS1 and DS3. "Operators per hundred physical lines" is the only contributing value $\geq$10% in the first dimension of both DS1 (11.199%) and DS3 (11.117%). Dimension 1 of DS2 does not share this characteristic, with the high values for "operators per

Table 4.9
DS2 contribution values. This table shows the contribution values of the first 6 components from the PCA for each of the variables constructing DS2.

	1	2	3	4	5	6
Percent Blank Physical Lines	5.401	0.014	1.582	0.404	0.095	16.661
Physical Lines Max Length	3.974	4.700	0.026	1.312	4.304	23.174
Logical Lines Per Hundred Physical Lines	6.914	2.442	6.773	5.558	3.768	0.047
Logical Line Max Length	0.937	10.304	4.167	17.244	0.0003	1.084
Logical Line Min Length	4.044	0.043	4.240	9.614	3.090	2.061
Code Words Per Hundred Physical Lines	1.202	15.228	0.806	6.340	6.001	0.179
Comments Per Hundred Physical Lines	0.493	0.027	12.057	4.941	10.623	0.932
Words Per Comment	0.145	15.911	5.076	0.566	7.329	1.054
Comment Characters Max Length	0.466	14.677	0.234	4.067	4.217	0.129
Comment Characters Min Length	0.677	0.977	0.289	0.409	26.024	11.454
Strings Per Hundred Physical Lines	0.003	9.520	4.716	1.554	0.026	0.380
String Characters Max Length	5.886	0.408	4.128	0.370	1.289	20.057
String Characters Min Length	7.921	1.943	2.837	1.412	1.945	1.057
Spacing Characters Per Hundred Physical Lines	7.224	1.274	7.292	0.417	2.203	1.550
Blocks Per Hundred Physical Lines	12.055	0.492	2.926	1.954	0.584	0.226
Longest Block Nest	13.223	0.505	0.589	1.944	0.015	0.009
Block Characters Max Length	7.100	0.002	8.981	0.762	0.024	0.025
Block Characters Min Length	0.001	0.183	3.476	19.143	9.908	5.593
Keywords Per Hundred Physical Lines	0.056	9.964	1.176	14.598	10.681	0.324
Operators Per Hundred Physical Lines	7.226	7.600	1.489	2.329	4.455	1.316
Numbers Per Hundred Physical Lines	5.047	2.988	4.629	2.195	0.042	3.511
Includes	0.131	0.351	10.848	1.717	1.243	8.303
Preprocessor Conditions	0.504	0.019	5.197	0.084	0.180	0.412
Longest Preprocessor Nest	9.370	0.428	6.465	1.067	1.952	0.461

hundred physical lines" separated between 2 dimensions in DS2. The highest contribution value of "operators per hundred physical lines" in DS2 is in dimension 2 as 7.6%, and only 7.226% in dimension 1.

Another similarity between DS1 and DS3, but not DS2, is in dimension 4. There are 3 contributing values $\geq$10% for the 4th dimension of DS3 (logical line min length: 22.753%, strings per hundred physical lines: 14.666%, comment characters min length: 10.849%). All 3 of these factors contribute $\geq$10% for dimension 4 of DS1 as well (logical line min length: 19.934%,

Table 4.10
DS3 contribution values. This table shows the contribution values of the first 6 components from the PCA for each of the variables constructing DS3.

	1	2	3	4	5	6
Percent Blank Physical Lines	2.746	3.451	2.440	5.509	5.679	1.688
Physical Lines Max Length	0.663	12.528	0.004	0.722	21.235	4.502
Logical Lines Per Hundred Physical Lines	8.108	1.532	2.660	9.150	0.008	0.207
Logical Line Max Length	0.586	16.958	3.043	1.814	0.960	1.978
Logical Line Min Length	0.290	0.209	0.148	22.753	0.403	0.109
Code Words Per Hundred Physical Lines	9.866	2.692	0.079	0.116	0.498	0.040
Comments Per Hundred Physical Lines	2.748	0.337	23.634	0.100	5.371	0.633
Words Per Comment	4.233	3.763	20.696	0.0001	0.028	0.265
Comment Characters Max Length	2.184	14.176	1.078	4.224	0.562	1.254
Comment Characters Min Length	2.919	0.805	13.298	10.849	1.111	0.604
Strings Per Hundred Physical Lines	3.767	1.386	1.570	14.666	0.759	3.860
String Characters Max Length	3.662	6.195	1.353	0.093	23.907	2.878
String Characters Min Length	5.314	1.872	3.958	2.427	0.352	10.003
Spacing Characters Per Hundred Physical Lines	0.564	10.859	1.056	3.761	14.030	3.093
Blocks Per Hundred Physical Lines	9.127	0.160	2.241	3.548	1.329	0.425
Longest Block Nest	9.153	3.206	0.404	0.004	2.221	0.307
Block Characters Max Length	4.265	5.207	1.115	2.863	4.491	2.852
Block Characters Min Length	0.419	2.335	5.449	0.278	0.111	33.025
Keywords Per Hundred Physical Lines	2.347	5.199	1.987	0.010	1.085	0.070
Operators Per Hundred Physical Lines	11.117	0.226	0.407	0.406	0.013	0.545
Numbers Per Hundred Physical Lines	7.713	0.615	0.202	5.273	0.718	0.160
Includes	0.006	1.504	8.408	9.329	1.183	31.054
Preprocessor Conditions	0.205	4.676	1.737	0.00003	13.818	0.363
Longest Preprocessor Nest	7.999	0.109	3.031	2.104	0.126	0.083

strings per hundred physical lines: 12.273%, comment characters min length: 10.699%). The 4th dimension of DS1 however, also has contribution values $\geq$10% from two additional factors (logical lines per hundred physical lines: 14.091%, block characters min length: 10.443%).

4.2.2 Correlations of PCA Results

The appearance of a variable as a component member for more than one data set could help identify which measures are helpful in distinguishing amongst

the code produced by novice programmers. Table 4.11 shows the variables with more than a 10% contribution to one of the first six components in each data set.

Table 4.11
Variables that contributed more than 10% to a dimension. Columns represent data sets. Cells represent the dimension that the variable was a member of.

	DS1	DS2	DS3
Block Characters Min Length	3, 4	4	6
Blocks Per Hundred Physical Lines		1	
Code Words Per Hundred Physical Lines		2	
Comment Characters Max Length		2	2
Comment Characters Min Length	4, 5	5, 6	3, 4
Comments Per Hundred Physical Lines	3	3, 5	3
Includes	3	3	6
Keywords Per Hundred Physical Lines	5	4, 5	
Logical Line Max Length		2, 4	2
Logical Line Min Length	4, 6		4
Logical Lines Per Hundred Physical Lines	4		
Longest Block Nest		1	
Operators Per Hundred Physical Lines	1		1
Percent Blank Physical Lines	5	6	
Physical Lines Max Length	2	6	2, 5
Preprocessor Conditions	6		5
Spacing Characters Per Hundred Physical Lines	2		2, 5
String Characters Max Length		6	5
String Characters Min Lengt			6
Strings Per Hundred Physical Lines	4		4
Words Per Comment	0	2	3

To further extract information about similarities between dimensions of the data sets, Pearson correlation analysis was conducted on the contribution values of variables in the first 6 dimensions of all 3 data sets. In table 4.12,

the correlation coefficients between all 6 of the selected components from all 3 of the data sets are included.

Table 4.12

Correlation of contribution values. This table depicts the Pearson correlation coefficients of the first 6 components from all 3 assignments (DS1, DS2, and DS3)

	DS1-1	DS1-2	DS1-3	DS1-4	DS1-5	DS1-6	DS2-1	DS2-2	DS2-3	DS2-4	DS2-5	DS2-6	DS3-1	DS3-2	DS3-3	DS3-4	DS3-5	DS3-6
DS1-1	1	−0.272	−0.328	−0.334	−0.105	−0.446	0.577	0.158	−0.226	−0.173	−0.076	−0.364	0.947	−0.371	−0.134	−0.272	−0.423	−0.327
DS1-2	−0.272	1	−0.353	−0.428	0.061	−0.306	0.018	0.397	−0.343	−0.007	−0.134	0.199	−0.263	0.743	−0.177	−0.211	0.367	−0.182
DS1-3	−0.328	−0.353	1	0.088	−0.290	0.287	−0.346	−0.297	0.714	0.033	0.189	−0.065	−0.332	−0.244	0.708	−0.221	0.043	0.476
DS1-4	−0.334	−0.428	0.088	1	−0.091	0.519	−0.272	−0.292	0.207	0.226	0.227	−0.051	−0.287	−0.380	0.051	0.816	−0.216	0.208
DS1-5	−0.105	0.061	−0.290	−0.091	1	−0.105	−0.174	0.130	−0.353	0.173	0.421	−0.016	−0.112	−0.017	−0.084	−0.022	−0.082	−0.231
DS1-6	−0.446	−0.306	0.287	0.519	−0.105	1	−0.261	−0.332	0.282	−0.005	−0.099	−0.001	−0.427	−0.124	0.030	0.345	0.257	−0.025
DS2-1	0.577	0.018	−0.346	−0.272	−0.174	−0.261	1	−0.448	−0.087	−0.376	−0.423	−0.067	0.600	−0.202	−0.399	−0.108	0.061	−0.273
DS2-2	0.158	0.397	−0.297	−0.292	0.130	−0.332	−0.448	1	−0.367	0.205	0.082	−0.259	0.128	0.372	0.045	−0.174	−0.277	−0.240
DS2-3	−0.226	−0.343	0.714	0.207	−0.353	0.282	−0.087	−0.367	1	−0.106	−0.202	−0.238	−0.215	−0.208	0.424	0.103	0.011	0.251
DS2-4	−0.173	−0.007	0.033	0.226	0.173	−0.005	−0.376	0.205	−0.106	1	0.175	−0.212	−0.270	0.217	−0.051	−0.023	−0.327	0.312
DS2-5	−0.076	−0.134	0.189	0.227	0.421	−0.099	−0.423	0.082	−0.202	0.175	1	0.129	−0.129	−0.173	0.541	0.040	−0.165	0.028
DS2-6	−0.364	0.199	−0.065	−0.051	−0.016	−0.001	−0.067	−0.259	−0.238	−0.212	0.129	1	−0.325	0.190	−0.038	−0.021	0.663	0.177
DS3-1	0.947	−0.263	−0.332	−0.287	−0.112	−0.427	0.600	0.128	−0.215	−0.270	−0.129	−0.325	1	−0.466	−0.181	−0.217	−0.354	−0.384
DS3-2	−0.371	0.743	−0.244	−0.380	−0.017	−0.124	−0.202	0.372	−0.208	0.217	−0.173	0.190	−0.466	1	−0.230	−0.276	0.402	−0.086
DS3-3	−0.134	−0.177	0.708	0.051	−0.084	0.030	−0.399	0.045	0.424	−0.051	0.541	−0.038	−0.181	−0.230	1	−0.102	−0.161	0.096
DS3-4	−0.272	−0.211	−0.221	0.816	−0.022	0.345	−0.108	−0.174	0.103	−0.023	0.040	−0.021	−0.217	−0.276	−0.102	1	−0.274	0.016
DS3-5	−0.423	0.367	0.043	−0.216	−0.082	0.257	0.061	−0.277	0.011	−0.327	−0.165	0.663	−0.354	0.402	−0.161	−0.274	1	−0.090
DS3-6	−0.327	−0.182	0.476	0.208	−0.231	−0.025	−0.273	−0.240	0.251	0.312	0.028	0.177	−0.384	−0.086	0.096	0.016	−0.090	1

This allows for examination of differences in the selected 6 components both between, and within all the data sets. Correlations of the first 6 dimensions of all data sets show similar results to the comparison of contributions values $\geq$10%.

Looking at only the highest correlation coefficients, there are only 5 $\geq$0.7, and 4 of these are between DS1 and DS3. The 4 correlation values $\geq$0.7 between DS1 and DS3 are in their 1st (0.947), 2nd (0.743), 3rd (0.708), and 4th (0.816) dimensions. The only correlation $\geq$0.7 with DS2 was between the 3rd dimensions of both DS2 and DS1, 0.714. Even though dimension 3 had correlation coefficients $\geq$0.7 between DS1 and DS2 as well as DS1 and DS3, the correlation coefficient between dimension 3 of DS2 and DS3 was only 0.424.

There were an additional 5 correlation coefficients $\geq$0.5 and $\leq$0.7. The correlation coefficient for dimension 1 of both DS2 and DS1 was 0.577, and for the 1st dimension of both DS2 and DS3, 0.600. Considering the correlation coefficient of 0.947 in the 1st dimension between DS1 and DS3, dimension 1 seems to be the most consistently strongly related for all 3 data sets. The remaining correlation coefficients $\geq$0.5 between dimensions of different data sets exist between DS2 and DS3, with a correlation coefficient of 0.633 for dimension 6 of DS2 and dimension 5 of DS3, as well as 0.541 for dimension 5 of DS2 and dimension 3 of DS3. DS1 had a correlation coefficient of 0.519 between its own 4th and 6th dimensions. All correlation coefficients $\geq$0.5 had positive values. 6 correlation coefficients had values $\geq$0.4 and 5 had values $\leq$-0.4.

Chapter 5

Discussion and Future Work

5.1 Discussion

The results presented in this thesis work towards answering the research questions outlined at the start of this thesis. The research questions were:

1. What are the main characteristics of novice code that contribute to the differences between submissions?
2. Are there characteristics of student code which can be used to create an assessment profile to assist in providing formative feedback to students?

The results from this research do not have sufficient information to fully answer either of these questions. Main characteristics of differences between submissions of novice code were not revealed in the PCA. Without finding significant characteristics of student code more information is needed to create an assessment profile. Although the two questions remain unanswered, observations of details in the results of the research warrant further examination.

While there are no consistent patterns that exist across all three samples in the results of the PCA, there are some patterns present when considering less samples than the total collected. One such example of this is preprocessor conditions. Preprocessor conditions appears important to one of the first six components for DS1 and DS3, but it does not for DS2. This low contribution to DS2 is unique to only that sample. Across the samples, DS1 and DS3 have the most commonalities.

From patterns that are present in the samples, operators per hundred physical lines offers high contribution to the most important component for DS1 and DS3 in the PCA. In DS2 operators per hundred physical lines has less, but moderate contribution to the top two components in a different pattern from DS1 and DS3. In DS2 longest block nest and blocks per hundred physical lines are top contributors to component 1. This has an implication that the necessity of blocks is different between DS2 and the other samples. It might be that DS2 has a different need for functions, conditional statements, or loops. The contribution to the component can not make a statement about whether this is good or bad, just that DS2 differs from DS1 and DS3. The results of the PCA suggest the importance of context upon tested code.

The existence of components overall implies that there are correlations and patterns that exist between different measures used in this study. This means that if one metric was well selected, other metrics would likely correlate with this. The work of this thesis shows interesting patterns between variables in a data set. Difficulty arises when attempting to generalise across multiple data

sets. We must ask what is possible when using this information to try to generate automated feedback. The results of this study suggest improvement of automated feedback should be possible, but the information used to generate feedback is highly context dependent.

To generate feedback, information must first be evaluated. The results of the evaluation must then be structured in a way that can be presented as the feedback. Simplification is useful in both the evaluation and the construction of the presented feedback. The purpose of PCA is to simplify data. Due to the lack of a success criterion in the available data, attention was directed at finding patterns in the data. The simplification by PCA did not show clear trends existing across all three samples. PCA works to aggregate measures, but this may not be useful since even the non-aggregate, unsimplified data was not consistent across all 3 samples. Inconsistency limits the usefulness to generate automated feedback.

5.1.1 Limitations

The results of the research presented in this thesis, although potentially informative for future work, faced some limitations.

5.1.1.1 Sample Quantity and Size

The first of these limitations was in the volume of data and the sample sizes available. This research is limited by having only three data sets to compare, which inhibits the ability to generalize to a larger scope. More samples could lead to a stronger ability to make inferences. Although DS1 and DS3 have

similarities they do not appear to be strong. Increasing the sample sizes (e.g. more students working on each assignment) would also be a strength in attaining more reliable results.

5.1.1.2 No Evaluative Criteria

The results presented in this research do not include a measure of the quality of code. The information that could have been extracted from these results would have been enhanced if there were qualitative criteria, such as grade. The purpose of an assignment impacts judgement of quality. If two assignments are for different purposes, what constitutes "good" code may be different. The purpose of this research is to find ways to increase the quality of feedback. Using a general purpose metric to then assess code with different purposes may impact the validity of assessment. Patterns that exist in the components of the samples, as well as differences between the components of the samples, are not representative of code quality.

A shortcoming of the results of this PCA is the lack of influence from quantification of student success. The only metric eluding to this is compilation success, but not success of the code for its intended purpose, or any scale of quality. The binary nature of compilation success makes it a weak variable for quality assessment. Attributes that constitute quality can not be assessed if quality is not measured. Code that compiles successfully may not be effective for its intended purpose. For these reasons, compilation success was not considered to scale success in this research.

5.1.1.3 Context Dependence of Assignments

If a metric that contributes highly to a component is considered valid for the assessment of any of the samples, there is no confirmation that use of this metric would be suitable in a different assessment. Traditional software metrics are heavily based on physical lines of code. Even though physical lines of code can be applied to code files in general, comparisons between use of this metric on different samples should not be considered accurately comparable. This sort of context dependence necessitates knowledge of the purpose of an evaluated sample to develop metrics useful in its assessment. General purpose automated feedback is restricted by this limitation.

Each of the samples is representative of a different assignment. In each assignment a student is expected to perform a different task. The differences between assignments impose a difference of context on the samples. The differentiation of necessity to complete the goals of an assignment introduce a difference to the underlying construct of quality in the code as well. Assignments may differ beyond the task to use of different programming languages. Ideal patterns differ between programming languages. The limitation of context could be reduced by examining more data sets and larger data sets.

5.1.2 Future Work

Some of the themes explored in the limitations future work. A first direction of research would be to evaluate the assignments at different levels. The research in this thesis looked at student code by file for each sample. Student code can also be evaluated by project and by student. Evaluation by file may

be more useful in error detection or changes to the context of the file. When the evaluation is by project this may find additional details such as weaknesses in the organization of the code. Analysis by student could reveal persistent trends in the code of a student that could be used for feedback of lasting details that need attention.

A strength to add to the research would be the inclusion of a dependent variable (such as grade) to scale quality so associative factors can be determined. To generate useful automated feedback for students, criteria for quality is necessary to direct feedback toward improvement.

It is important to ensure that validity is maintained as coding practices and technology change. Although the underlying idea of software metrics remains similar, software has changed. For example, in the C programming language, the syntax of functions has greatly changed. Another example is change from primarily using 32 bit vs 64 bit processors. This creates differences both before, and after compiling software, so collection tools must also change. With each change, we must not only check if collection methods will still consistently perform the intended operations, but also if the intended operations correctly represent what they are meant to. It is simpler to confirm the integrity of software metrics, than to confirm if they truly support our inferences.

Although context introduces confounding implications, if studied, the confounding factors may be identified and further understood. The understanding attained from study of context may be used to find ways for controlling or avoiding confounding factors of context.

Adding to software simply introduces the possibility of more instances of any possible attribute that can be introduced within the addition. Size can be considered a confounding factor for many software metrics. As software size increases, the need to treat the collected data as an increased sample size is introduced.

The research in this thesis may expose new pathways in examination of the potential uses of software metrics in providing formative feedback to students. The research questions have not been fully answered, but the examination of the results shows representation that there may be patterns in the code. Further analysis of similar research including a scale of quality would enhance the conclusions and their use in improving formative feedback in education.

Bibliography

Abegaz, T. T. and Spence, D. J. (2019). Impact of compiler's feedback on coding performance. In Human Interface and the Management of Information. Visual Information and Knowledge Management, Lecture Notes in Computer Science, pages 265–279. Springer International Publishing, Cham.

Ahadi, A., Lister, R., and Mathieson, L. (2018). Syntax error based quantification of the learning progress of the novice programmer. In Proceedings of the 23rd Annual ACM Conference on innovation and technology in computer science education, ITiCSE 2018, pages 10–14. ACM.

Ahmed, I., Brindescu, C., Mannan, U., Jensen, C., and Sarma, A. (2017). An empirical examination of the relationship between code smells and merge conflicts. In Proceedings of the 11th ACM/IEEE International Symposium on empirical software engineering and measurement, ESEM '17, pages 58–67. IEEE Press.

Barker, L.-K., Moore, J. W., Olmi, D. J., and Rowsey, K. (2019). A comparison of immediate and post-session feedback with behavioral skills training to improve interview skills in college students. Journal of organizational behavior management, 39(3-4):145–163.

Basili, V. and Rombach, H. (1988). The tame project: towards improvement-oriented software environments. IEEE transactions on software engineering, 14(6):758–773.

Boehm, B. and Papaccio, P. (1988). Understanding and controlling software costs. IEEE transactions on software engineering, 14(10):1462–1477.

Chidamber, S., Darcy, D., and Kemerer, C. (1998). Managerial use of metrics for object-oriented software: an exploratory analysis. IEEE transactions on software engineering, 24(8):629–639.

Daskalantonakis, M. (1992). A practical view of software measurement and implementation experiences within motorola. IEEE Transactions on Software Engineering, 18(11):998–1010.

Fenton, N. and Neil, M. (2000). Software metrics: Roadmap. Proceedings of the Conference on the Future of Software Engineering.

Fenton, N. E. (2015). Software metrics : a rigorous and practical approach. Chapman & Hall/CRC innovations in software engineering and software development. CRC Press, Boca Raton, FL, third edition. edition.

Fenton, N. E. and Neil, M. (1999). Software metrics: successes, failures and new directions. The Journal of systems and software, 47(2):149–157.

Gibbs, G. and Simpson, C. (2005). Conditions under which assessment supports students' learning. Learning and teaching in higher education, (1):3–31.

Gil, Y. and Lalouche, G. (2017). On the correlation between size and metric validity. Empirical software engineering : an international journal, 22(5):2585–2611.

Gilb, T. (1976). Software metrics. Studentlitteratur, Lund.

Grady, R. B. (1992). Practical software metrics for project management and process improvement. Prentice-Hall, Inc.

Hall, T. and Fenton, N. (1997). Implementing effective software metrics programs. IEEE software, 14(2):55–65.

Harmer, J. (2019). Identifying indicators of maturity in software developed by novice programmers. Master's thesis, University of Guelph.

Hattie, J. (2009 - 2009). Visible learning : a synthesis of over 800 meta-analyses relating to achievement. Routledge/Taylor & Francis Group, London ;.

Huisman, B., Saab, N., van den Broek, P., and van Driel, J. (2019). The impact of formative peer feedback on higher education students' academic writing: a meta-analysis. Assessment and evaluation in higher education, 44(6):863–880.

Hung, S.-L., Kwok, I.-F., and Chan, R. (1993). Automatic programming assessment. Computers and education, 20(2):183–190.

Inglis, J. (1986). Standard software quality metrics. AT&T Technical Journal, 65(2):113–118.

Jurado, F., Redondo, M. A., and Ortega, M. (2012). Using fuzzy logic applied to software metrics and test cases to assess programming assignments and give advice. Journal of network and computer applications, 35(2):695–712.

Kitchenham, B. (2010). What's up with software metrics? – a preliminary mapping study. The Journal of systems and software, 83(1):37–51.

Lind, R. and Vairavan, K. (1989). An experimental investigation of software metrics and their relationship to software development effort. IEEE transactions on software engineering, 15(5):649–653.

McCarthy, J. (2017). Enhancing feedback in higher education: Students' attitudes towards online and in-class formative assessment feedback models. Active learning in higher education, 18(2):127–141.

McConnell, J. J. (1996). Active learning and its use in computer science. In SIGCSE bulletin, volume 28, pages 52–54.

Meneely, A., Smith, B., and Williams, L. (2012). Validating software metrics: A spectrum of philosophies. ACM transactions on software engineering and methodology, 21(4):1–28.

Mishra, D. and Mishra, A. (2008). Some observations on staff estimation metrics for object: oriented software engineering. Software engineering notes, 33(5):1–4.

Molnar., A., Neamţu., A., and Motogna., S. (2019). Longitudinal evaluation of software quality metrics in open-source applications. In Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE,, pages 80–91. INSTICC, SciTePress.

Musson, T. (1994). The evaluation of instructional software quality. WIT Transactions on Information and Communication Technologies, 8.

Nuñez-Varela, A. S., Pérez-Gonzalez, H. G., Martínez-Perez, F. E., and Soubervielle-Montalvo, C. (2017). Source code metrics: A systematic mapping study. The Journal of systems and software, 128:164–197.

Omri, S., Montag, P., and Sinz, C. (2018). Static analysis and code complexity metrics as early indicators of software defects. Journal of software engineering and applications, 11(4):153–166.

Ott, C., Robins, A., and Shephard, K. (2016). Translating principles of effective feedback for students into the cs1 context. ACM transactions on computing education, 16(1):1–27.

Rees, M. (1982). Automatic assessment aids for pascal programs. SIGPLAN notices, 17(10):33–42.

Sadler, D. R. (1989). Formative assessment and the design of instructional systems. Instructional science, 18(2):119–144.

Sheela, G. S. (2017). Maintenance effort prediction model using aspect-oriented cognitive complexity metrics. International journal of advanced research in computer science, 8(8):278–281.

Shelley, C. (1993). Experience of implementing software measurement programmes in industry. In Proceedings 1993 Software Engineering Standards Symposium, pages 72–78. IEEE Comput. Soc. Press.

Shute, V. J. (2008). Focus on formative feedback. Review of educational research, 78(1):153–189.

Voas, J. and Kuhn, R. (2017). What happened to software metrics? Computer (Long Beach, Calif.), 50(5):88–98.

Williamson, B. (2019). Policy networks, performance metrics and platform markets: Charting the expanding data infrastructure of higher education. British journal of educational technology, 50(6):2794–2809.

Yadav, R. K. (2017). Optimized model for software effort estimation using cocomo-2 metrics with fuzzy logic. International journal of advanced research in computer science, 8(7):121–125.