AUTOMATIC IDENTIFICATION AND ANALYSIS OF COMMENTED OUT CODE

Blake Grills

A Thesis

Submitted to the Graduate College of Bowling Green
State University in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE

May 2020

Committee:

Michael Decker, Advisor

Ray Kresman

Robert Dyer

ABSTRACT

Michael Decker, Advisor

In this thesis, we study commented-out code in software and its automatic identification. Commented-out code can inhibit program comprehension which leads to increased maintenance time, commented-out code can have legal ramifications, and degrade the performance of text analysis techniques (among others) that rely on comments. In order to address these issues, we develop a taxonomy on comments and more specifically on commented-out code, we manually verify nearly 3,000 lines of comments, develop an automatic approach to identifying commented-out code based on character frequencies and decision trees, and analyze the prevalence of commented-out code in open-source software. The results of our initial study to develop a model to automatically identify commented-out code boasts scores all greater than 95% (F1 score is 97.89%, precision is 98.32% and recall is 98.23%) on our test and training data. Results of applying our model to additional completely unseen projects gives a respectable precision of 78.03, recall of 99.62 and a F1 score of 87.51. From our study of open-source projects, on average a project lines of commented-code is 4.46% of the total lines of comments (median 2.76) and only four of the 50 projects contained no commented-out code.

This thesis is dedicated to Arthur Grills and the Brothers and Sisters

who have fallen in service to their country.

ACKNOWLEDGMENTS

I would be remiss to not mention a few people who have changed both my academic career and my life as whole. To the Bowling Green State University computer science department faculty and staff, you have all changed my life in ways beyond counting, we have shared many laughs and tears through the trials and tribulations that was my time here. To Dr. Green and Dr. Decker, both of you, whether or not you realize it, saved my life during some of the hardest and darkest times coming out of the service. But, neither of you were willing to stop at just that, you both pushed me to continue in my academic career, you showed me just what I could become, and I am proud to say that I am here now because of the two of you. I think of both of you as more than family or friends, you are both heroes in my eyes and will always hold a special place in my heart.

TABLE OF CONTENTS

Page

Contents

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1.

Introduction

Software evolution involves the process of continuously improving or changing code to ensure that it maintains a working status as software requirements, operating systems change, etc. One of the largest and most important parts of this is the concept of keeping overall cost down in the lifecycle of a software product and avoiding having to entirely replace expensive software. While the concept of continuous change is important the problem is that maintenance is expensive and as a program continues to change, the risk of reduced comprehension comes into play (which leads to slowed development and increased cost). To support this continuous change and software evolution we have to consider all facets of comprehension.

The most expensive, time consuming, and longest part of the software development life cycle is widely known to be maintenance, and work is always being done to try and simplify this lengthy portion of the life cycle. 90 percent of the total cost of software comes from maintenance [Détienne 1990], and the amount of time that we must spend in maintenance is directly proportional to the amount of time programmers spend comprehending the code they are

reading [Borstler and Paech 2016]. It is this concept of comprehension, alongside maintenance, that is the basis for why more research is needed to support better comprehension.

There are two prominent ways people read and comprehend code, with a typical person using a combination of both. The first is the top-down approach and the second is bottom-up. Top-down comprehension relies largely on inference and hypothesizing what exactly is going on at any given time in the source code and drilling down to find details later. For example, choosing to guess what a called function is doing and waiting to read the function definition later on [Détienne 1990] [Maletic and Kagdi 2008]. In contrast, the bottom-up approach focuses on understanding the smaller pieces of the source code first and working up to the large functions until eventually you understand the program as a whole [Storey 2005] [Maletic and Kagdi 2008] [Détienne 1990] [Von Mayrhauser and Vans 1995].

Comments are one means of providing support for comprehension (for both top-down and bottom-up). Comments should always provide cognitive support, assist in comprehension, and reinforce ideas and concepts that are present in the source code in order to reduce the difficulty of maintenance [Storey 2005] [Détienne 1990]. For example, the concept of cognitive support can be reinforced in comment structure by including beacons, which are familiar pieces that are pulled directly from the code [Von Mayrhauser and Vans 1995] [Storey 2005]. One problem with comments comes from that of commented-out code. Commented-out code bloats the software and is a distractor from the meaningful parts of the program (source-code and meaningful comments). Additionally, commented-out code can lead to confusion, especially if something that is commented out seems to be unrelated to the section that it is in, or if a piece of commented-out code directly contradicts logic present in and around where it has been commented out. Of course, alongside comprehension, when you consider the long-term

maintenance of code, the simple question of why a piece of code has been commented out is likely to come up. In this case it is likely unknown as to whether or not the commented-out code is a security vulnerability, is a feature that needs to be implemented later, is it reference code that was used to build another section earlier on, or does it cause a total crash if it is run? There is a legal precedent to the problem of commented-out code too. Companies are being sued for leaving commented-out code that does not belong to them in their source code. [Flexra] [United States District Court Northern District of California 2017] [Vaughan-Nichols 2015] Companies are also encountering problems with commented-out code being moved back into an active status. The next paragraph summarizes one such case.

A prominent example of the need for more research into commented-out code comes from the Knight Capital case which occurred on August 1, 2012. On that day, an updated copy of Knight Capital's stock purchasing software was deployed on seven of their eight servers with a fatal flaw. A flag was set to activate a portion of dead code meant purely for simulation purposes. The activation of this commented-out code led to the purchase of over seven billion dollars' worth of stock in the span of one hour. Even after all of the returns and buy backs, it still left the company at a net loss of 440 million dollars after just one hour of their software running [Dolfing 2019]. All of this loss could have easily been avoided if dead code was not permitted to be in the final launch version of the companies' software, but this is a task that is easier said than done with the sheer size of projects growing every day. This can be shown with a major change in terminology within the software community. For instance, Lines-of-Code (LOC) is a term that is falling out of practice as measurements are changing into Thousands-of-Lines-of-Code (KLOC) or Millions-of-Lines-of-Code (MLOC).

The problem that we are trying to alleviate is with the presence of commented-out code in software. Commented-out code is any piece of source code that has been disabled by means of commenting the line it is on, for example `//float alpha =.05;`. There are many different issues that this can cause, leaving security vulnerabilities easily visible to would be attackers. Often times by looking at commented-out code we can see the logic behind how a particular section of code was built. At times it can also identify a clear problem. For example, if `//if(val==-1){` has been commented out, and it is not utilized in the main source code, there is an implication that `val` equaling `-1` is something that causes a problem. In this case an attacker can now explore methods that may cause `val` to equal `-1` and assuming that they succeed, this could lead to potential security issues or a full out crash of their software.

Our goal with this current research is to offer a method for automatically detecting commented-out code within a software project, with the hopes that we can improve maintenance time and mitigate confusion later on when commented-out code is found. Our thought process is that if we can detect commented-out code rapidly throughout the development phase of the software, then there is a direct route to question why it has either been added in or commented out at the time of origin rather than trying to decipher the meaning later on. Additionally, we can automatically remove commented-out code at any time. Of course the benefits are not limited merely to maintenance, by detecting commented-out code early we have the ability to protect companies from disclosing security vulnerabilities that may be outlined in sections of commented-out code or to avoid sections of commented-out code that have the potential to be accidently made active. Yet another reason that commented-out code removal is important is that it can aide with other research and tools that utilize comments. For example, when considering automatic summarization techniques, feature location techniques, and techniques

that utilize natural language processing, commented-out code which is not related to the source code can degrade these techniques performance. As an explicit example, when considering the method *word2vec*, which relies on creating vector data based on the terms found in comments. In this case, it will create vectors which are not relevant to the source code. If the commented-out code is removed, it is likely that we can improve these techniques greatly.

In order to automate the process, we first develop a data artifact (i.e. a gold set) that is read and evaluated by hand. This manually derived gold-set is the culmination of approximately 3000 lines of comments which were pulled from a total of 78 projects via a random selection process. These 78 projects are distributed evenly amongst the four languages that are being studied in our research (C, C#, C++, Java) and our random selection process ensures that the comments chosen are distributed evenly across these languages. Next, in order to detect commented-out code we break down every line into individual characters and store them in a dictionary which is then further broken down into frequencies of each character in comparison to the total number of characters in the line. We then feed this data into a machine learning algorithm. Our chosen machine learning approach is the C4.5 Decision Learning Tree algorithm. One of the reasons that we chose to use the Decision Learning Tree algorithm is that it uses a supervised learning style. This makes it easy to learn how the tree is differentiating between our two classes (i.e., if commented-out code or not) and do mathematical and statistical verification on the data. We use stratified K-fold cross validation to split our data into training and testing sections to optimize the learning schema. We feed our data through multiple times in order to ensure overfitting has not occurred. These processes will be described in more detail in Chapter 8. All of the decisions that were made in regard to the classification of these comments (i.e., what is and what is not commented-out code) is based off of a taxonomy of comments that we

have developed. Research in this area is exceptionally limited and the debate on text detection is an ever changing and evolving field. Our hope is that our research can help drive code detection in a good, steady direction.

In this thesis we answer the following research questions:

- **RQ 1: What is commented-out code?**

  Comments can contain any amount of code. They can consist solely of code or just reference a single variable among normal prose text. As such, in this **RQ**, we investigate and determine what exactly it is that makes a comment commented-out code.

- **RQ 2: What are the different ways to provide comments and commented-out code?**

  The taxonomy of a comment is much more complex than can be assumed. We can split comments both between English prose, commented-out code, single line, multi-line (block), Doxygen, Javadoc, etc. As such, in this **RQ**, we investigate and taxonomize the ways of providing comments and commented-out code.

- **RQ 3: Can we automatically detect commented-out code within an acceptable margin of error?**

  Detection of commented-out code is vital to solving the problems that they cause, as frequently the people commenting out the code do not see it as an issue or may just forget to delete it. The trouble with detecting commented-out code is that high level programming languages are very similar in appearance to English, making it hard to immediately tell the difference. As such, in this **RQ**, we will determine how to detect commented-out code.

- **RQ 4: How prevalent is commented-out code in open-source software?**

  With this RQ, we want to quantify how much commented-out code typically exists in a software project. Due the availability of open-source software, we take 50 open-source systems and apply our classification approach to each comment to determine how often commented-out code makes it into open-source software.

This thesis makes the following contributions:

- A detailed taxonomy of comments and what various types of comments there are.
- A gold set created from manual investigation and verification of nearly 3,000 comments (classified as English prose and commented-out code) from a corpus of 78 open-source projects.
- Investigation on differences between English prose and commented-out code
- The development of an approach to automatically classify comments into the fields of English prose and commented-out code with low margin of error.
- An in-depth study of the prevalence of commented-out code in 50 additional open-source projects outside of our original training and testing set.

The remainder of this thesis is laid out as follows. In CHAPTER 2, we go over related work. In **Error! Reference source not found.**, we cover our taxonomy on comments. In **Error! Reference source not found.**, we cover the source code parsing tool srcML. In **Error! Reference source not found.**, we detail the process of data collection. **Error! Reference source not found.** focuses on the analysis of the data that we have obtained and processed in the data artifact. **Error! Reference source not found.** provides an in-depth discuss on decision trees, our machine learning method. In **Error! Reference source not found.**, presents and

discusses the results of the automation process on test data. **Error! Reference source not found.** presents and discusses the results of the open source study. **Error! Reference source not found.** focuses on our threats to validity. **Error! Reference source not found.** outlines our plans for future work based off of the results from this study. Finally, **Error! Reference source not found.** is our conclusion, which gives a brief overlay on all our results.

CHAPTER 2.

Related Work

In this chapter we present the related works and break them down section by section. Despite the growing need to detect commented-out code, there has not been much research. As such, we primarily focus the related work discussion on a few related areas. In Section 2.1. we provide related work in the study of comment taxonomies. In Section 2.2. we provide related work on identifying code in unstructured text. In Section 2.3. , we provide related work related to major works in automatic code generation and studies on nature of comments themselves. In Section 2.4. , we focus on various works in comment quality, and the importance it has in code comprehension. The final Section 2.5. , focuses on research and studies that work on automatization of both detection and summarization of various types of source code and text.

2.1. Taxonomy

In [Chen et al. 2019], Chen et al. introduce a comment taxonomy with four different types. These are the non-prose and low purpose, Code Comments, which we call commented-out code, Task Comments which are notes such as TODO or FIXME, IDE comments, which are special comments designed to communicate to the IDE directly, and non-text comments which

are links to websites or other comments that are not directly related to the source code. This taxonomy which they have developed is highly related to our research. However, the work of Chen et al. focuses only on that of prose comments and they do not study task comments and more importantly code comments. In our taxonomy we do a much more in-depth study of commented-out code and its many forms.

Haouari et al. have developed an in-depth taxonomy of comments which is designed to provide as much information as possible on the quality of a comment. First they consider what the object of a comment is, for example whether or not a comment is related to a snippet of code or an entire function [Haouari et al. 2011]. Second is the comment type, here they define 4 types of comments, a code explanation, TODO comments, commented-out code, and licensing agreements [Haouari et al. 2011]. The next field is the style, which is only used in explanatory comments and defines whether or not the explanation is explicit or implicit; essentially whether or not you need to read the code to understand the comment [Haouari et al. 2011]. Following style is the final field of their taxonomy, defined as the comment quality. They define three levels of quality which are fair+, fair, and poor. Each of these are based off the quality of the explanation that the comment gives [Haouari et al. 2011]. For the sake of our research the only part that is of real importance is the comment types, making the distinction between an explanatory comment that contains code vs commented-out code will be very important. Haouari et al. define commented-out code as old code which remains in a comment that provides no explanatory value [Haouari et al. 2011]. Our taxonomy differs from theirs because we focus more on the construction of the comment (i.e., how you form a comment) and what makes it commented-out code, whereas their team focuses on the location of the comment and explanatory value (i.e., quality).

2.2. Detecting Code in Unstructured Text

In Bacchelli et al.[Bacchelli et al. 2010a], the authors developed an approach for automatically detecting code in emails. In the process of developing their method for automated code detection Bacchelli et al. tested a variety of different methods, They test frequency of special characters, occurrence of keywords, end of line symbols, beginning of line symbols, regular expression, and a series of combinations between all of them [Bacchelli et al. 2010a]. The results of these approaches are interesting, what they show is that no individual method was enough to be consistently accurate for detecting code in emails. Furthermore, most of their combinations involve adding in regular expression to increase precision and recall, sometimes by a very significant amount. The final results of testing these methods both with and without regular expression shows an optimal case of 85-95% detection rate by using end of line in combination with regular expression. However, this varies on the language it is trying to detect [Bacchelli et al. 2010a]. There is a shortcoming to their research, when they detect special characters their focus is too tight and fails to note characters that can be syntactically very important. For example, a common method of declaring string variables requires a quotation mark, which is a symbol rarely present in common speech. This is where our research and the work of Bacchelli et al. differs, we include more special characters that they did not consider and utilize decision trees to help decide which characters are the most important.

Another example of the detection of code is provided by Abdalkareem et al. in their study where they pulled code from StackOverflow for part of a gold set. The heuristic Abdalkareem et al. employ is simple. Since StackOverflow posts are in html, they consider text within code tags as code. Additionally, the answers have to be five lines or greater. The reason they chose to exclude anything less than five lines was because they were concerned that code

tags with less than five lines would end up generating false positives in their analysis [Abdalkareem et al. 2017]. While code detection is related to the work that we are doing, the specific work of this team relies on the preexisting code tags found in the StackOverflow message board. Our approach does not rely on markers such as a code-tag and is capable at working on the granularity of a single line. Likewise, the results our work is applicable to the text in code-tags to further validate if they are code or not.

2.3. Comment Generation

Related to our work is research on comment generation and comment studies. Comment generation for source code has a large variety of different benefits, from increasing the quality of preexisting comments, adding documentation where there is none at all, and aiding in the understanding of the source code for users outside the original writing base [Song et al. 2019]. Generating comments is not without its problems, however, because of the complexities of both varying structures of languages and coding styles in addition to different naming conventions, often times comments will require some amount of human verification in order to ensure they make sense. Additionally, because there is no one way to form a sentence in a language, auto generating text can sometimes sound unnatural or just be gibberish [Song et al. 2019] [Binkley et al. 2013].

Not only is automatic generation difficult but the actual process of generating the comments can be extremely computationally expensive. An example is that of Song et al., where they first process all of the text and analyze it using a variety of different information retrieval methods such as a VSM or LSI model and then take those results and run them through deep neural networks in order to produce viable output [Song et al. 2019]. The end result of the research of Song et al. is a method of applying machine learning to automatically generate

comments with little human intervention. While this method of comment generation may be effective, it does however have its pitfalls and relies on developing an effective method to extrapolate the meaning of the code. One problem with their approach is that they are unable to differentiate between comment-out code and not, which the authors note as a problem. The results of this thesis (i.e., an automatic classifier) would directly benefit their approach.

Movshovitz-Attias and Cohen developed a method for predicting and generating programming comments using topic models and n-grams in order to reduce the amount of time spent writing comments [Movshovitz-Attias and Cohen]. The team considered two different types of values when detecting the code that they would be auto generating comments for. The first of these two types are text, which is primarily comments but also string literals. The second type, code, is the type that we are much more interested in. Movshovitz-Attias and Cohen focus on major syntax tokens such as public, private, for and so on as well as all variables and identifiers. In order to better identify these tokens they use a link-LDA model in order to vectorize these samples [Movshovitz-Attias and Cohen]. Once these vectors are created they can be used to generate comments based off of the source code. The way this works is by analyzing the highest vector values to focus down on topics that the comments will be modeled off of [Movshovitz-Attias and Cohen].

Abid et. al developed another method for generating summaries and comments from source code through the analysis of method stereotypes [Abid et al. 2015]. These method stereotypes are specified as the Structural Accessor (get, predict, property), Structural Mutator (set, command), Creational (factory) and Collaborational (collaborator, controller) [Abid et al. 2015], The approach utilizes a template for each stereotype. The contents of the templates are

then filled in with specific information taken from the method body to form a comment for the method [Abid et al. 2015].

2.4. Comment Quality

One method of analyzing readability and comprehension of software is to directly analyze the comments left by the authors of source code. Borstler and Paech note that one of the largest problems within the field of comment research is that much of the research is more than 20 years old, and higher degrees of decomposition has greatly changed the effect comments have on comprehension. When considering the quality of comments, each is analyzed individually to determine not only if it covers the strategic components of the code well but also if it provides additional information that is relevant to the overall comprehension of a code snippet [Borstler and Paech 2016].

One problem with comment quality is attributed to external factors such as when the native language of the speaker differs from the language comments are written in, subject programming experience, and domain knowledge on the subject software [Zhou et al. 2019] [Borstler and Paech 2016] [Flisar and Podgorelec 2019]. A method of analyzing comments and source code that takes these methods into account which is very popular in natural language processing is using vector decentralization to normalize semantic cognition through utilization of `word2vec`, a tool that vectorizes each word of a document [Zhou et al. 2019]. These vectors can be grouped and analyzed for similarity that focuses on the raw value of the grouped terms, which provides a solution to some of the problems, such as native language, natural language processing engineers are facing with code today [Zhou et al. 2019]. Similar to this method of vector grouping, is the use of word embedding [Flisar and Podgorelec 2019]. Word embedding focuses on low dimensional real value vectors rather than looking at groupings of words for

semantic value [Flisar and Podgorelec 2019]. These varying methods all come to the same conclusion, comments should be meaningful and related to the source code that they are in, which is something that commented-out code does not do. This supports the importance of our research in today's modern coding age.

Another example of comment quality is given by Fluri et al. and is based off of the evolution of comments as source code evolves. One of the first things that their team focuses on is the detection of changes to the source code in order to see where comments have been added, removed, extended, or changed [Fluri et al. 2007]. By tracking these changes their team is able to visualize the changes to the commenting on the source code over time to see if comments are being kept cohesive and relational to the complexity of the source code. One of the findings of their team is that the closer a comment is to code within the source file, the higher quality it tends to be [Fluri et al. 2007]. For example, a comment which is on the same line as code is going to be extremely related, whereas a comment before a function will give a less detailed overview of a section of code.

Another part of comment quality is the overall density of comments found within source code. Riehle and Arafat studied the commenting practices of open source programming, which included a deep analysis of the relative density of comments in a project. Interestingly, the standard deviation of comment coverage in open source projects is over 10.88% [Arafat and Riehle 2009]. The largest finding of their research showed that the longer the project gets, the more comment density falls going from 62.5% in small projects and descending to 18.67% in large projects [Arafat and Riehle 2009]. This is directly related to our **RQ 3**, which has to do with the relative density of comments in open source projects. However, Riehle and Arafat did not consider commented-out code in their study, which we investigate in **RQ3**.

Another area of study on comments is the study of comment coverage within source code. This research is directly applicable to work such as automatic comment generation as a way to verify that the comments that are being generated are accurately capturing the meaning of the source code in question. One such method of coverage analysis is to use *word2vec* which allows the user to create connections based on semantic similarities within the comments and the code [Chen et al. 2019]. To analyze and condense the massive amount of data that is produced by *word2vec*, Chen et al. recommend the application of random forest machine learning algorithms. This is because not only are they very powerful when it comes to the analysis of classifiers and have many well-established implementations, but random forests are also very good for bagging and bootstrapping data.

2.5. Automated Summarization and Text Detection

As an alternative to the generation of comments within source code, the possibility of generating full summaries of source code provides an option that helps to mitigate some of the shortcomings of generating text [Haiduc et al. 2010a]. This method of summary has two different routes that it can take, one of which is the idea of extracting valuable information and placing it directly in the summary while the other method focuses on abstraction, which chooses to provide a general overview of the source code. Both methods rely on the same base method however, the first step is to take all of the terms out of the source code and then convert the terms into a corpus sorted by frequency to determine which terms are the most relevant [Haiduc et al. 2010b] [Allamanis et al. 2016] . These types of summarizations can be evaluated in much the same way as the generated comments, because they are also built with information retrieval tools such as LSI or VSM, and they can be compared at a mathematic level. However, the best method is human analysis and questionnaires [Haiduc et al. 2010a] [Song et al. 2019]

[Allamanis et al. 2016] [Binkley et al. 2013]. Ultimately, this means that these types of summarization methods are very similar to comment generation, but they work on a much larger scale.

This type of large scale summarization serves other purposes however, for example if you are able to look at the bigger picture of code like this then it is very possible to create a method that can detect and track conversations about a piece of source code over email. Bacchelli et al. [Bacchelli et al. 2010b] recommend a method for actually accomplishing this using a series of different linking techniques that are not limited to the studies on natural language processing. Of course, when considering methods outside of natural language processing such as regular mailing lists that are bound to projects, you can get an idea of whether or not conversation about source code is occurring. However, in the end you will need to analyze the actual content of those emails to determine if they are directly related and to which part of the source code they are related to [Bacchelli et al. 2010b] [Allamanis et al. 2016]. This is an important distinction because while a project may end up with 30,000 emails, there may only be a few hundred actual links to the project itself [Bacchelli et al. 2010b]. Our research could prove to be a significant aid in further development of this sort of automatic generation, namely because you do not want commented-out code being summarized in a final product.

The concept of text detection, particularly in relation to email is something that is becoming more and more common and is becoming needed in today's society. For example, in the field of detecting opinion spam, new research has been published utilizing neural networks to identify spam that is deliberately misleading in its review [Ren and Ji 2019]. There is a major difference between detecting the spam itself and detecting spammers, in our case what we care about is the spam detection. Detecting spam occurs in a series of three main phases, not unlike

the method that we are using in our own research. First, they use human beings to read and identify spam that they consider to be malicious or misdirecting. Second, the results are filtered using filtering algorithms, most of which are proprietary. Finally the data is fed through performance evaluation to verify filter quality such as F1 score, Roc and AUC [Ren and Ji 2019]. While this method of detection does not focus on detecting or understanding code, what it does do is provide valuable insight into understanding natural language at a machine learning level. This work is highly related to what we are doing, as detection of spam is largely the same as detecting commented-out code. However, their method focuses on identifying differences in English speech where code, while it may have similarities to English, is essentially a different language.

CHAPTER 3.

Taxonomy of Comments

In this chapter, we present a taxonomy on comments and commented-out code. This provides us with the necessary background and terminology we will use throughout the paper, as well as, defines what we consider commented-out code for the purpose of this thesis. In the process we provide answers to **RQ1** and **RQ2**.

How we define the structure of a comment is extremely important to our research as well as defining exactly what commented-out code is. In this thesis, we will refer to commented-out code as commented-out code and other comments as English prose. First, in TABLE 1, we give a taxonomy (with examples) on various ways a programmer may provide comments. The first two are the traditional line-comment and block comments which are used to provide a one-line or multi-line comment and are used for both commented-out code and English Prose. The third is `#if 0` preprocessor directives, which so long as you do not change the 0 to a 1 (or another true value) all text/code up until a matching `#endif` will be stripped out by the compiler during the preprocessor step. This form of comment is largely used to comment out regions of code. The fourth type is an `if(0)` block. This method is very similar to the preprocessor method; however,

it is a language statement and is thusly compiled. This form of comment is largely used to comment out code. The fifth and sixth are Doxygen and Javadoc. These are special types of one-line and/ multi-line comments. Doxygen comments typically contain commented-out code and English prose but may also contain hyperlinks and code reference points unique to Doxygen comments. The sixth is Javadoc comments which function similarly to Doxygen comments where as Doxygen is for many programming languages, Javadoc is specifically designed for the Java language. With this, we have now answered **RQ 2 (What are the different ways to provide comments and commented-out code?)**.

TABLE 1. PROVIDES A DETAILED DESCRIPTION OF EACH OF THE SIX TYPES OF COMMENTS, LINE, BLOCK, #IF 0, IF(0), DOXYGEN, JAVADOC.

Line Comment	A single line within source code that has been commented out by means of a line comment marker such as //. Used for both commented-out code and English prose.
Block Comment	Multiple lines within source code that have been commented out by means of a block comment marker such as /* ending with */. Used for both commented-out code and English prose.
#if 0	Multiple lines of code within source code that have been commented out by means of wrapping the code inside a preprocessor if with a condition of 0. All text/code up until a matching #endif is removed automatically by the preprocessor during compilation. Typically, used to comment out code.
if(0)	While similar to the preprocessor, an if-statement is used, the standard if(0) is processed and compiled by the compiler. This type of block can be rapidly commented and uncommented by changing the 0 to 1 and vice-versa. Can be used to comment-out code.
Doxygen	Used to write software reference documentation, these comments can have hyperlinks and other document wide references. Typically, used to comment out code.
Javadoc	Used to write software reference documentation, these comments have hyperlinks and other document wide references. While similar to Doxygen it is limited to Java languages.

In this thesis, although a part of our taxonomy, we do not investigate `#if 0` and `if(0)` style comments for commented-out code. This is due both to the rarity of these types of comments, as well as, the fact that they are generally only used to comment out code and not used for English

prose. As future work, we can investigate the prevalence of `#if 0` and `if(0)` style comments in projects and investigate if there are any instance where they are used for English prose.

At this point we formally define English prose and commented-out code. We define an English prose comment as *any comment which does not contain syntactically correct code for the language that it is present in*. While typically a comment will be primarily composed of English explanations, you may also see references to variables, mathematical equations, or full algorithms. These types of comments make up the bulk of all comments in source code and are used as tools in order to aide in the understanding of the source code. We define commented-out code *as any piece of source code that has been disabled by means of commenting with one of the methods from TABLE 1.*

Here we explain and give examples of the different types of comments and how they how they are used with English prose and commented-out code. Line comments have a prefix operator such as `//` in the c family that tells the compiler to ignore anything after the operator until the end of a line. This can be placed anywhere on a line, even after code. Typically, line comments are used to make small notes in a specific section of code, either saying what a variable is used for or marking areas that need fixed. Many IDEs provide a feature to quickly comment a line or series of lines (generally of code). In this feature, the standard line comment is often the default commenting method used by IDEs. It is also important to note that these line comments can be used to create a pseudo block comment, this is very easy to do with IDE enabled commenting. TABLE 2 contains five examples of line comments as well as descriptions of what each comment contains. The first and second comments are simple English prose comments. The third comment is unique and may generate a false positive in our detection, however we do not consider it to be commented-out code because it is actually an algorithm reference that is used to explain the

calculation that follows. The fourth comment is a simple member initialization which has been commented out and is therefore commented-out code. The fifth comment is a commented-out head of a four loop, which is also commented-out code. The final comment is a block of commented-out code, commented out using individual line comments.

A block comment differs from a line comment in that it uses both a prefix and suffix operator to block off an entire section for writing or for commenting out code. There are a few different methods for accomplishing this, a common method is the `/*` prefix and the `*/` suffix within the C family of languages. TABLE 3 contains five samples of block comments as well as descriptions of what each comment contains. The first comment is a detailed licensing breakdown held inside a block comment. The second comment is a full description of a function broken down in detail. While there are references to code within it, we do not consider it to be commented-out code because as a majority it is English prose with the references behaving like nouns. The third comment is a sample of a single line which has still been commented out using a block comment, which is considered commented-out code. The fourth comment is a large block of commented-out code. The fifth comment is unique in that it is commented-out code, but only a single word and on the same line as additional code. In this case, the virtualization has been commented out making the function no longer virtual (additionally, the author may be highlighting that the method is an inherited virtual function).

TABLE 2. 5 EXAMPLES OF LINE COMMENTS. THE LEFT SIDE PROVIDES AN EXAMPLE WHILE THE RIGHT SIDE PROVIDES THE EXPLANATION. THE FIRST THREE CONTAIN REGULAR ENGLISH PROSE, THE SECOND TWO CONTAIN SNIPPETS OF COMMENTED-OUT CODE.

Comment Samples	Comment Description
`//returns the final cost after calculating tax`	This comment is in reference to a return of a variable and is an example of an inline comment meant to explain what piece of code is doing.
`//Variable instantiation section`	This comment references a section of code giving a simple description.
`//accuracy = (TP + TN)/(TP + TN + FP + FN)`	This comment references the equation used to build a snippet of code.
`// m_depth(0)`	This comment is a simple member initialization which has been commented out.
`//for(int p = 0;p<P.rows();p++)`	This comment is the start of a for loop which has been commented out.

`// void Print(int res[20][20], int i, int j,\`
`int capacity)`
`// {`
`//  if(i==0 || j==0)`
`//  {`
`//  return;`
`//  }`
`//  if(res[i-1][j]==res[i][j-1])`
`//  {`
`//  if(i<=capacity)`
`//  {`
`//  cout<<i<<" ";`
`//  }`
`//`
	This comment is a sample of a block comment made of single line comments.

```
//          Print(res, i-1, j-1, capacity-i);
//      }
//      else if(res[i-1][j]>res[i][j-1])
//     {
//          Print(res, i-1,j, capacity);
//      }
//      else if(res[i][j-1]>res[i-1][j])
//      {
//          Print(res, i,j-1, capacity);
//      }
// }
```

TABLE 3. 5 EXAMPLES OF BLOCK COMMENTS. THE FIRST THREE CONTAIN STANDARD ENGLISH PROSE, THE LAST TWO CONTAIN COMMENTED-OUT CODE.

Comment Samples	Comment Description

`/*`
` * Licensed under the Apache License, Version 2.0 (the "License");`
` * you may not use this file except in compliance with the License.`
` * You may obtain a copy of the License at`
` *`
` * http://www.apache.org/licenses/LICENSE-2.0`
` *`
` * Unless required by applicable law or agreed to in writing, software`
` * distributed under the License is distributed on an "AS IS" BASIS,`
` * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.`
` * See the License for the specific language governing permissions and`
` * limitations under the License.`
` */`
	This comment contains all of the details of the licensing related to the source code in the file.

`/*`
` * findInternal --`
` *`
` *  Sets ret.second to value found and ret.index to index`
` *  of key and returns true, or if key does not exist returns false and`
` *  ret.index is set to capacity_.`
` */`
	This comment contains a full description of what is occurring in a function.

`/* Tests are based on the examples in the pandoc documentation */`	This comment is a sample of a single line kept in a block comment marker.

`/*var cour = EntityManager<Courier>.Entities.FirstOrDefault();`
 	
`UpdateManager.Subscribe(() =>`
`{`
`if (cour == null)`
	This comment is a sample of commented-out code.

`return;`
`var items = cour.Inventory.Items;`
`Console.WriteLine("----------------------");`
`foreach (var item in items)`
`{`
`Console.WriteLine($"{item.Name} \`
`{item.OldOwner?.Name} | {item.Owner?.Name}");`
`}`
`}, 1000);*/`	

`/*virtual*/ void AbstractASTMatcherRule::setUp()`	This comment is a special sample of commented-out code were only the virtualization of the function has been commented out.

These are not your only options however, there are many programmers who use Doxygen and Javadoc comments. Doxygen and Javadoc comments function the same way that a standard block comment does but offer a variety of supplemental features such as cross-referencing and source code linking. These types of comments employ specific structure syntax and operators to allow for the automatic generation of API documentation. Doxygen and Javadoc have a wide variety of prefixes and suffixes that are used to demarcate a span of comments such as `///` and `/**`. TABLE 4 contains five examples of Doxygen/Javadoc comments as well as descriptions of what the comment contains. Additionally, all forms of commenting for Doxygen/Javadoc are covered here. The first comment is a standard Doxygen/Javadoc comment which contains no special markers or additional features available in the commenting style. The second comment is a standard Doxygen/Javadoc comment which contains hyperlinks and parameters, while these do directly reference code we do not consider this to be commented-out code as it is used as English words. The third comment is a sample of standard English prose being commented out using single line commenting from Doxygen/Javadoc. The fourth comment is a sample of an alternative commenting method of doing block comments when using Doxygen/Javadoc style comments. The final comment is all commented-out code, which has been commented out using the Doxygen/Javadoc style. The final methods for commenting, which is used exclusively (as far as we know) in order to comment out code, is with `#if0` or `if(0)` block comment. This method of commenting is used to quickly comment out portions of code. The difference between the two is the first is stripped by the preprocessor while the second is compiled and can be executed. Often this is done for testing purposes, though it may also be done in order to lock out certain features that are not yet ready to be implemented. TABLE 5 contains two examples of commented-out code, one using the preprocessing `#if 0` method and the other using the `if(0)` block. All of

these examples provide samples of commented-out code, this alongside the definition of commented-out code that we gave in this chapter, answers our **RQ 1 (**What is commented-out code?).

TABLE 4. 5 EXAMPLES OF JAVADOC/DOXYGEN COMMENTS. COMMENTS 3 AND 4 ARE IN THE ALTERNATIVE METHODS FOR COMMENTING WHEN USING DOXYGEN/JAVADOC. THE LAST COMMENT CONTAIN COMMENTED-OUT CODE.

Comment Samples	Comment Description
/** * Adds a benchmark. Usually not called directly but instead through * the macro BENCHMARK defined below. The lambda function involved * must take exactly one parameter of type unsigned, and the benchmark * uses it with counter semantics (iteration occurs inside the * function). */	This comment is a simple sample of a Doxygen/Javadoc comment with no special markers or extra features.
/** * Appends the specified number of low-order bits of the specified value to this * buffer. Requires 0 <= len <= 31 and 0 <= val >2^{len}. * @param val the value to append * @param len the number of low-order bits in the value to take * @throws IllegalArgumentException if the value or number of bits is out of range * @throws IllegalStateException if appending the data * would make bitLength exceed Integer.MAX_VALUE */	This comment is a sample of a Doxygen/Javadoc comment that includes hyperlinks to both parameters and exceptions.
/// Construct an alive Account, with given endowment, for either a normal (non-contract) /// account or for a contract account in the conception phase, where the code is not yet known.	This comment is a sample of Doxygen/Javadoc that is using the /// rather than /** and */ to block off a comment.
/*! Copyright (C) 2002, 2003 Sadruddin Rejeb Copyright (C) 2004 Ferdinando Ametrano Copyright (C) 2005, 2006, 2007 StatPro Italia srl This file is part of QuantLib, a free-software/open-source library for financial quantitative analysts and developers - http://quantlib.org/	This comment is a sample of Doxygen/Javadoc that is using the /*! And */ rather than /** and */ to block off a comment.

QuantLib is free software: you can redistribute it and/or modify it under the terms of the QuantLib license. You should have received a copy of the license along with this program; if not, please email <quantlib-dev@lists.sf.net>. The license is also available online at <http://quantlib.org/license.shtml>. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the license for more details. */	

/**
*
* BENCHMARK_START_GROUP(insertVectorBegin, n) {
* vector<int> v;
* BENCHMARK_SUSPEND {
* v.reserve(n);
* }
* FOR_EACH_RANGE (i, 0, n) {
* v.insert(v.begin(), 42);
* }
* }
*/
	This comment is a sample of commented-out code held in a Doxygen/Javadoc comment.

TABLE 5. THE FIRST CELL OF THIS TABLE CONTAINS CODE WHICH HAS BEEN COMMENTED OUT USING THE PREPROCESSOR METHOD #IF. THE SECOND CELL CONTAINS THE SAME COMMENTED-OUT CODE WHICH HAS BEEN COMMENTED OUT USING A STANDARD IF(0) BLOCK.

```
#if 0

    for(int i = 0; i < 10; ++i){

        a += i;

    }

#endif
```

```
if(0){

    for(int i = 0; i < 10; ++i){

        a += i;

    }

}
```

CHAPTER 4.

srcML

In order to ensure that all of the comments found in the source code are properly pulled for analysis the language parsing tool, srcML, is used[Collard and Maletic]. At its core srcML is a tool designed to take source code and automatically convert it into an XML representation. srcML processes source code independent of the preprocessor, and so you do not have to worry about things such as missing external libraries. Further, because srcML does not need to compile the code in order to analyze and extract information, it is able to run extremely quickly. srcML has the ability to leave the original structure of the source code entirely intact, meaning that whitespace, comments, and all preprocessing comments are left untouched. In TABLE 6, we provide a sample of the input and output of srcML. A simple C++ file was fed into srcML using the command:

```
srcml --verbose srcMLsample.cpp -o srcMLsample.xml
```

The function provided in the top is the main function of the sample C++ file *srcMLsample.cpp*. The second row contains the resulting srcML from the

srcMLsample.xml . The first line of the srcML provides the encoding information for XML. The second line of the output provides the details on the current version of srcML that was used for the command, additionally the language the file is written in, C++, and the file name *srcMLsample.cpp* is included on this line. Each part of the source code is wrapped in XML tags. First the outer most wrap is the function tag which includes the type and name tags relation to the term *int*. as we go deeper into the tags we see things like literals broken down by type, operators, controls and blocks. It is important to note that certain symbols cannot be visualized in XML, for example, the less than symbol in the control block of the for loop is visualized as <. Included is a sample of a doxygen comment, which differs from a standard comment in that the tag includes *format="doxygen"*. Another option for running srcML is to run on a full directory of files rather than a single file. When we perform this sort of batch operation using srcML we still create an XML file, however rather than being a standard type of XML file it generates a unique type of XML file known as a srcML archive. There are a few notable characteristics of archive files that differ from the standard format. First, a *unit* tag still represents a file within the archive and while the format remains largely the same we now include the hash of each file inside the unit tag attributes. This new archive *unit* tag encompasses all the file *unit* tags TABLE 7 is a sample of an XML archive file which has been generated using srcML with the command:

```
srcml -verbose path -o archivesample.xml
```

As for TABLE 7, The first three lines provide a sample of an archive unit tag with the special hash attribute, this is repeated later on in the sample for another file. The full path for each file is provided as well. Each archive contains a simplified root *unit* tag which contains the information on the version of srcML you are using. Other than these changes to the unit tag

the remainder of this XML archive file maintain the same convention as a standard xml file created using srcML.

TABLE 6. THE TOP BLOCK OF THE TABLE PROVIDES A SMALL SAMPLE FUNCTION WRITTEN IN C++. THE BOTTOM BLOCK OF THE TABLE PROVIDES THE SAME CODE AFTER IT HAS BEEN TRANSLATED TO XML USING SRCML. EACH LINE OF THE FUNCTION HAS BEEN BROKEN DOWN AND EACH PIECE IS THEN TAGGED. FOLLOWING THE BLOCK COMMENT SAMPLE, A SECOND ONE LINE SAMPLE IS PROVIDED AND THEN AGAIN BROKEN DOWN USING SRCML.

Original Source Code

```
int main(){
    /// sum of first 14 numbers
    int a = 0;
    for(int i = 0; i < 15; ++i){
        a += i;
    }

    return 0;
}
```

srcML

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<unit xmlns="http://www.srcML.org/srcML/src" xmlns:cpp="http://www.srcML.org/srcML/cpp"
revision="0.9.5" language="C++" filename="srcMLsample.cpp">
<function><type><name>int</name></type>
<name>main</name><parameter_list>()</parameter_list><block>{
    <comment type="line" format="doxygen">/// sum of first 14 numbers</comment>
    <decl_stmt><decl><type><name>int</name></type> <name>a</name> <init>= <expr><literal
type="number">0</literal></expr></init></decl>;</decl_stmt>
    <for>for<control>(<init><decl><type><name>int</name></type> <name>i</name> <init>=
<expr><literal type="number">0</literal></expr></init></decl>;</init>
<condition><expr><name>i</name> <operator>&lt;</operator> <literal
type="number">15</literal></expr>;</condition>
<incr><expr><operator>++</operator><name>i</name></expr></incr>)</control><block>{
        <expr_stmt><expr><name>a</name> <operator>+=</operator>
<name>i</name></expr>;</expr_stmt>
```

```
    }</block></for>

    <return>return <expr><literal type="number">0</literal></expr>;</return>
}</block></function></unit>
```

TABLE 7. THIS IS A SAMPLE OF AN XML ARCHIVE FILE GENERATED USING SRCML ON A DIRECTORY. THE PRIMARY DIFFERENCE BETWEEN THIS AND A STANDARD XML FILE THAT IS GENERATED BY SRCML IS THAT EACH UNIT TAG REPRESENTS AN ENTIRE FILE, AND EACH UNIT TAG GAINS AN ADDITIONAL HASH ATTRIBUTE WHICH CAN BE USED TO VERIFY FILE INTEGRITY.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<unit xmlns="http://www.srcML.org/srcML/src" revision="1.0.0">

<unit xmlns:cpp="http://www.srcML.org/srcML/cpp" revision="0.9.5" language="C#"
filename="Class1.cs" hash="693d899bc71f2dcd8335fac076940b2b8e1a933e">…</unit>

<unit xmlns:cpp="http://www.srcML.org/srcML/cpp" revision="0.9.5" language="C"
filename="ClassHierarchyJob.h" hash="51abc9d61f337d3cb3410bbbf4cb9cbf7db9a506">…</unit>

</unit>
```

Once source code has been converted to XML using srcML, whether it is an archive or single file, the user is able to write XPath, a query language for selecting nodes from an XML document. XPath queries allow someone to pull any specific information needed from the original source code quickly and easily. For the purposes of this thesis, this allows us to ignore the actual code in the source and extract just the comments. A sample XPath command to extract all comments from a srcML archive is:

```
srcml --xpath "//src:comment" project.xml -o Comments.xml
```

This results in a new xml document containing only the comments found in the Comments.xml document as a srcML archive. TABLE 8 contains a sample of the Comments.xml document. There are a few notable differences from the standard XML file format found in a standard srcML command. First the XPath command results, like an archive file, have a `unit` tag for every entry (i.e., comment found). Second each filename attribute contains the path to the original file the query result is pulled from, rather than the path of the XML file. Each entry also contains a new unique attribute called item, which maintains a count of each instance of the query in a file and restarts at each new file.

TABLE 8. THE SAMPLE PROVIDED IS OF THE RESULTS OF AN XPATH COMMAND RUN BY SRCML ON A SRCML XML ARCHIVE FILE. THE FIRST DIFFERENCE IN AN XPATH XML FILE AS SHOWN HERE IS THAT EACH ITEM IS GIVEN ITS OWN UNIT TAG. EACH OF THESE UNIT TAGS IS UNIQUE TO THE ONE FOLLOWING IT. FIRST THE FILENAME ATTRIBUTE USES THE PATH TO THE FILE CONTAINING THE RESULT OF THE XPATH QUERY RATHER THAN CONTAINING THE PATH TO THE FILE THAT THE SRCML COMMAND IS RUN AGAINST. THE NEXT ATTRIBUTE WHICH IS UNIQUE TO THE XPATH QUERY IS THE ITEM ATTRIBUTE THIS ATTRIBUTE COUNTS EACH OCCURRENCE OF THE QUERY WITHIN A GIVEN FILE AND RESETS ON EACH NEW FILE.

```
<unit xmlns:cpp="http://www.srcML.org/srcML/cpp" revision="0.9.5" language="C++"
filename="C:\Users\blake\OneDrive\Desktop\school\ThesisProject\ThesisCorpus\~ready\0-1
Knapsack.cpp" item="26"><comment type="line">//}</comment></unit>


<unit xmlns:cpp="http://www.srcML.org/srcML/cpp" revision="0.9.5" language="C++"
filename="C:\Users\blake\OneDrive\Desktop\school\ThesisProject\ThesisCorpus\~ready\8cc.h"
item="1"><comment type="line">// Copyright 2014 Rui Ueyama.  Released under the MIT
license.</comment></unit>

<unit xmlns:cpp="http://www.srcML.org/srcML/cpp" revision="0.9.5" language="C++"
filename="C:\Users\blake\OneDrive\Desktop\school\ThesisProject\ThesisCorpus\~ready\AABB.h"
item="1"><comment type="line">// This file is part of libigl, a simple c++ geometry processing
library.</comment></unit>


<unit xmlns:cpp="http://www.srcML.org/srcML/cpp" revision="0.9.5" language="C++"
filename="C:\Users\blake\OneDrive\Desktop\school\ThesisProject\ThesisCorpus\~ready\AABB.h"
item="5"><comment type="line">// This Source Code Form is subject to the terms of the Mozilla
Public License </comment></unit>
```

CHAPTER 5.

Data Collection

In this chapter we present our method for data collection and provide a sample of the output. The process of our data collection is vital in order to create a corpus that provided a diverse variety of different coding styles, comment styles, and frequencies of commented-out code. In Section 5.1. we discuss the process of selecting the 78 projects which make up the data set we use in training our decision tree. Section 5.2. explains the methodology used to extract the comments from our corpus using srcML. Finally, in Section 5.3. we explain how the manual verification of our two classifications is completed.

5.1. Corpus Selection

To ensure that the quality of the base source code that is being used in this project, we pull highly starred projects from GitHub using the filter preferences on GitHub. The reason for this is two-fold, first, projects that have higher numbers of stars are likely to be better maintained as there is greater scrutiny on the projects, and second, these projects are more likely to be under active development/maintenance. Based on this, the 20 topmost starred C, C++, C# and Java

projects (78 total) have been selected and pulled for use in building the data artifact used in this project. We choose C, C++, C#, and Java as these are the only languages srcML supports. However, as these are among the most popular languages used in industry and open source, we do not consider this a significant threat to validity in our current research.

5.2. Comment Extraction

The first step in the comment extraction process is making sure that all of the projects for our corpus were being held within the same directory for ease of use with srcML. Once all of the projects are in a centralized location, we run srcML to convert all of the source code from the 78 projects into one archive file. After this is done an XPath query is used to extract all the comments from the archive. The extracted comments are placed in a new archive by srcML. The query we use is:

srcml --xpath "//src:comment" project.xml -o comments.xml

In the case of this research, this is the appropriate step to take as the rest of the source code is not needed. Once all of the comments have been pulled and placed into their own XML file 2,935 lines of comments were selected at random from all projects. The reason that we specify lines of comments rather than 2935 comments is because each line is analyzed independent of whether it is in a block comment or not. These 2935 comments are chosen on a two-point randomization method. First, a selection of random source code files is pulled from the main corpus. After this, the 2935 comments were selected at random from the archive created out of these source code files. TABLE 9 is a detailed analysis of the 78 projects from which the 2935 comments were pulled and will be described in detail at the end of this chapter.. The reason for this is because it is very possible to have a block comment which contains lines of both English Prose and lines of pure commented-out code. These selected lines were then

manually classified into either English prose or commented-out code using the definition from **Error! Reference source not found.**.

5.3. Manual Classification

The entire process of manual verification covered a spread of 2,935 lines of comments from amongst the 26 different projects and covers a mix of all four languages selected for this project. We verify all comments on a line-by-line basis. In the case of block comments, each line was reviewed and classified separately as shown in TABLE 10. The reason for reviewing even block comments in this manner is that it is very possible to have a block comment that is a mix of both commented-out code and standard English prose. The manual verification process took a total of 185 hours both of initial review and second pass verifying the classification. The whole process was performed over the course of three months. The remainder is present for the purpose of continued research. The first of these columns contains the comments themselves, in the case of block comments, each line is stored independently and each of the 7 columns are filled out for each line, as described in the previous paragraph. In the interest of maintaining the integrity of the data, all of the blank lines within block comments have been kept as well and are stored on their own lines. To maintain comments of all different types the markers for the comments are also maintained in these lines. Some examples of this include '//', '/*', '*', '///' and in the case of C++ and C style block comments potentially no marker at all (when in middle of comment).

TABLE 9. BELOW IS A DETAILED TABLE OF ALL OF THE PROJECTS THAT WERE SELECTED IN ORDER TO BUILD THE GOLD SET. THE FIRST COLUMN GIVES THE NAME OF THE PROJECT. THE SECOND COLUMN GIVES ALL OF THE LANGUAGES THAT WERE USED IN THE PROJECT THAT SRCML IS CAPABLE OF PARSING. THE THIRD COLUMN GIVES THE TOTAL NUMBER OF LINES OF CODE OF EACH PROJECT CAPABLE OF BEING PARSED BY SRCML. THE FOURTH COLUMN PROVIDES A TOTAL COUNT OF THE NUMBER OF COMMENTS PRESENT IN THE SOURCE CODE THAT SRCML IS CAPABLE OF PARSING. THE FIFTH COLUMN PROVIDES THE TOTAL NUMBER OF LINES OF COMMENTS PRESENT IN THE GOLD SET FROM EACH PROJECT. THE SIXTH COLUMN PROVIDES THE TOTAL NUMBER OF LINES OF COMMENTED-OUT CODE PRESENT IN THE GOLD SET FROM EACH PROJECT. THE SEVENTH COLUMN IS THE TOTAL NUMBER OF AUTHORS IN EACH PROJECT AS LISTED ON GITHUB. THE EIGHT COLUMN PROVIDES THE COVERAGE OF COMMENTS WHEN COMPARED TO LINES OF CODE. THE NINTH COLUMN GIVES THE PERCENTAGE OF COMMENT COVERAGE ACROSS THE WHOLE CORPUS FOR EACH PROJECT WITHIN THE CORPUS.

Name	Language	Number of Authors	LOC	Comments	Comment Lines in Gold Set	Commented-Out Code in Gold Set	Comment Coverage	Percentage of Corpus
8cc	C	9	9791	712	1	0	0.07	0.0341
30 Days of Code	C#/Java/C++	40	2408	66	0	0	0.03	0.0000
Abseil cpp	C++	62	100544	33736	0	0	0.34	0.0000
aleth	C++	148	90154	9979	134	0	0.11	4.5656
algorithms	C++	134	9390	2809	1	0	0.30	0.0341
asio	C++	28	105180	38013	47	0	0.36	1.6014
atom gpp compiler	Java	5	343	38	0	0	0.11	0.0000
aws sdk cpp	C++	57	2503490	2509104	217	0	1.00	7.3935
BansheeEngine	C#/C++	1	70046	19781	0	0	0.28	0.0000
BridJ	Java/C++/C	10	29002	14317	0	0	0.49	0.0000
c4	C	5	495	12	0	0	0.02	0.0000

captcha	Java	28	510	26	0	0	0.05	0.0000
cbc	Java/C	1	9638	769	0	0	0.08	0.0000
cdt	Java/C++/C	154	1122144	450027	85	0	0.40	2.8961
cgeo	Java	101	79981	14019	0	0	0.18	0.0000
civetweb	C/C++	160	581605	191390	0	0	0.33	0.0000
C	C/C++	1	4684	829	0	0	0.18	0.0000
codelite	C++/C	64	892859	228394	0	0	0.26	0.0000
C Plus Plus	C++/C	100	5767	576	31	26	0.10	1.0562
cpp sublime snippet	C++	3	502	0	0	0	0.00	0.0000
DeepLearning	C/C++/Java	1	4083	239	0	0	0.06	0.0000
distcc	C/C++	45	27906	8514	0	0	0.31	0.0000
Duckuino	Java/C++/C	5	1208	173	0	0	0.14	0.0000
DynamicExpresso	Java/C#	15	15245	4521	9	0	0.30	0.3066
EasyHttp	C#	12	3518	1382	0	0	0.39	0.0000
EnsageSharp	C#	8	36262	3345	13	13	0.09	0.4429
EnyimMemcached	C#	9	14195	5020	0	0	0.35	0.0000
FASTER	C#/C/C++	32	32710	5607	0	0	0.17	0.0000
faster than c	Java	3	446	18	1	0	0.04	0.0341
fast xml parser	Java	33	3724	504	0	0	0.14	0.0000

Fleck	C#	24	3135	58	0	0	0.02	0.0000
folly	C++/C	501	286037	84631	1249	78	0.30	42.5554
gdbgui	Java/C/C++	27	1999	568	0	0	0.28	0.0000
glog	C++/C	76	8077	3869	0	0	0.48	0.0000
HackerRank	Java/C#/C++	9	9230	4487	0	0	0.49	0.0000
hiredis	C/C++	99	6283	1611	0	0	0.26	0.0000
http parser	C/C++	77	6349	704	0	0	0.11	0.0000
HttpTwo	Java/C#	2	9152	2499	0	0	0.27	0.0000
j2c	Java/C/C++	1	10948	350	27	0	0.03	0.9199
j2objc	Breaks CLOC	66	881870	636880	63	0	0.72	2.1465
javacpp	Java/C/C++	33	20291	6279	0	0	0.31	0.0000
javacpp presets	Java	57	500002	318650	0	0	0.64	0.0000
json c	C/C++	94	6536	2469	0	0	0.38	0.0000
jsoncpp	C++/C	146	8526	1386	0	0	0.16	0.0000
LeetCode	Java/C++/C	38	3782	65	0	0	0.02	0.0000
leetcode	C	28	12636	487	0	0	0.04	0.0000
libco	C++/C	12	3099	448	0	0	0.14	0.0000
libconfig	C/C++	27	43670	2108	0	0	0.05	0.0000
libigl	C++/C	82	92359	28712	385	94	0.31	13.1175

librdkafka	C/C++	163	77178	31555	0	0	0.41	0.0000
lightning	C/C++	129	104892	22790	0	0	0.22	0.0000
markdowndeep	Java/C#	6	51833	10893	24	0	0.21	0.8177
MessagePack Csharp	C#	45	45564	4585	32	6	0.10	1.0903
mini c	C	1	3454	6	11	0	0.00	0.3748
MissionPlanner	C#	71	798373	240724	188	42	0.30	6.4055
mono	C#/C/C++	747	6049345	1295150	11	11	0.21	0.3748
msgpack c	C/C++	113	93200	11583	0	0	0.12	0.0000
nativejson benchmark	C++/C	32	16083	1337	42	7	0.08	1.4310
NiL.JS	Java/C#	8	505660	139548	0	0	0.28	0.0000
node pg native	Java	10	1234	66	0	0	0.05	0.0000
node pre gyp	Java/C++/C	47	2533	271	0	0	0.11	0.0000
NodObjC	Java/C/C++	11	9062	2495	0	0	0.28	0.0000
nuklear	C/C++	100	60370	12260	0	0	0.20	0.0000
oclint	C++/C	29	21478	478	59	13	0.02	2.0102
Openwrt NetKeeper	C/C++	15	1519	616	0	0	0.41	0.0000
osrm backend	C/C++	109	195834	30768	42	0	0.16	1.4310
QR Code generator	C/Java/C++	2	4088	1206	53	0	0.30	1.8058
QuantLib	C++/C	99	376347	72562	163	11	0.19	5.5537

rtags	C++/C	110	22545	2178	0	0	0.10	0.0000
ServiceStack.Redis	C#	61	29413	3529	0	0	0.12	0.0000
SharpSCADA	C#	4	57238	3846	0	0	0.07	0.0000
sonar objective c	Java/C/C++	8	1931	895	0	0	0.46	0.0000
stb	C/C++	158	69880	11952	14	0	0.17	0.4770
stratisBitcoinFullNode	C#	62	208433	38753	0	0	0.19	0.0000
UnityCsReference	C#	150	467413	35634	0	0	0.08	0.0000
v7	C/C++/Java	17	30268	8982	0	0	0.30	0.0000
websocketpp	C/C++	41	19577	12500	0	0	0.64	0.0000
XobotOS	Java/C++/C#/C	1	2002448	1125133	33	0	0.56	1.1244
Total		5022.00	18999004.00	7762476.00	2935.00	303.00		100.0000
Average		64.38	243576.97	99518.92	37.63	3.88	0.23	1.2821
Median		32.50	19934.00	3687.50	0.00	0.00	0.19	0.0000

The purpose of this was to determine if certain types of comments were more likely to generate false positives in the machine learning algorithm and, if this was the case, to ensure that we manipulate the comments by removing these markers before feeding them into the machine learning algorithm. This includes removing the stars at the beginning of each line of a Doxygen/Javadoc comment. The second through fourth columns are used primarily for bookkeeping purposes but do provide important information especially towards future research.

The second column is the name of the source-code file from which the comment has been pulled from. This file name is extracted from the path information provided by srcML in the XML archive used in the production of this data artifact. The third column is labeled block comment, and there are two different ways that this is filled in. If this column is marked with a n then the line is not part of a block comment. If the line is given a range of numbers, then those numbers represent the range of lines (rows of data) that are a block comment that the line is a part of. Note, here this number applies only to the range of entries and not to the source code itself. The fourth column is labeled as language and represent the coding language that the source code was written in. We decided to add this column for the purpose of both future research and to ensure that anyone viewing the data artifact will know what language the comment was written in regardless of whether or not they are familiar with all of the different file endings attributed to a language. The language column is followed by two different column's that are related to one another. The first is the contains code column and the second is the is code column. Both contain either a *y* for yes or a *n* for no.

TABLE 10. THE FIRST COLUMN OF THE TABLE IS THE COMMENT PULLED DIRECTLY FROM THE SOURCE CODE. THE SECOND COLUMN IS THE FILE THE COMMENT COMES FROM. THE THIRD COLUMN SHOWS WHETHER OR NOT THE COMMENT IS PART OF A BLOCK COMMENT, IF IT IS THEN IT SHOWS THE LINES OF THE FILE THE BLOCK COMMENT IS COMPRISED OF. THE FOURTH COLUMN LISTS THE LANGUAGE THE FILE IS WRITTEN IN. THE FIFTH COLUMN SHOWS WHETHER OR NOT THE LINE CONTAINS CODE. THE SIXTH COLUMN SHOWS WHETHER OR NOT THE LINE IS ENTIRELY CODE. COLUMN SEVEN CONTAINS ANY TERMS WHICH ARE STANDARD TO THE LANGUAGE THE LINE IS WRITTEN IN.

comment	file	block comment	language	contains code	is code	contains standard terms
// 0-1 Knapsack problem - Dynamic programming	0-1 Knapsack.cpp	n	C++	n	n	
// #include <bits/stdc++.h>	0-1 Knapsack.cpp	n	C++	y	y	include
// void Print(int res[20][20], int i, int j, int capacity)	0-1 Knapsack.cpp	(4-27)	C++	y	y	void, int
// {	0-1 Knapsack.cpp	(4-27)	C++	y	y	
// if(i==0 || j==0)	0-1 Knapsack.cpp	(4-27)	C++	y	y	if
// {	0-1 Knapsack.cpp	(4-27)	C++	y	y	
// return;	0-1 Knapsack.cpp	(4-27)	C++	y	y	return

The first of these two columns, the contains code column, is the fifth column. This does not mean that the line is commented-out code, but contains a code-like entity (e.g., an equation or identifier). This was determined to be a likely source of false positives when categorizing commented-out code with the machine learning algorithm. The primary thing that we check for when determining whether or not to mark this comment with a *y* are identifiers and equations. While equations seem to be less common identifiers may be included in order to aid in the description of what a section of source code does or to mark what functions need to be called within an area of the source code. The sixth column, which is the column labeled is code, is the second column directly important to the machine learning algorithm. This column is very straight forward and is marked with either a *y* or *n* depending on whether or not it is determined that a comment line is commented-out code. However, it is important to note that this has nothing to do with the actual source code itself, rather, we decided to mark anything that is syntactically valid if uncommented. For example, in the C family any line that appears like the line below is syntactically valid and is therefore considered commented-out code. For certain cases where it may be unclear if it was commented-out code we made sure to validate by looking at the source code. The following comment is commented-out code as it is syntactically valid statement from the language it is written in:

```
// totalCost = price + salesTax - discount;
```

The seventh column, standard terms, is only ever filled when a comment line is commented-out code. The primary purpose of this column is to provide a list of terms that could be used as a bag of words when identifying lines of commented-out code. For example, in C++ *#include, return, void, int, string, virtual, float, and double* are all fairly common within code and are terms that could be used to identify commented-out code. We are also marking things

such as `if`, `else`, `else if`, etc. though these are less likely to be helpful due to the fact that they are common English words. All data collected is stored in a csv.

TABLE 11 contains a breakdown of the gold set by comment type and language. The first column contains the language, C and C++ have been combined. The number of sample lines is the total number of lines of comments for the particular language, these are largely C/C++. The number of Block comments is the number of non-Doxygen/Javadoc block comments. The number of line comments is the number of single line comments. The number of Doxygen/Javadoc comments is the count of block comments made in the Doxygen/Javadoc style only. The lines of commented-out code are the total number of lines which are commented-out code. The lines containing code references are a count of the lines which reference pieces of code but are not true lines of commented-out code. The lines containing standard terms are lines which contain standardized terms such as `virtual`, `void`, and `int`. While TABLE 11 does show us that a large amount of the gold set does contain a large amount of C/C++, there is a decent amount of non C/C++. A part of why there is so much more C/C++ is because of the exposure and prevalence of C/C++ across all projects. This is exemplified by looking at the percentage of coverage each language has across all 78 projects. C/C++ appears in over 59% and 67% of projects respectively, meanwhile C# and Java only appear in 29% and 31% of projects. Figure 1 shows the breakdown of comment type as a pie chart. Line comments make up almost exactly 50% of the comment types. Doxygen/Javadoc comments make up another approximately 30% of the comment types with Block comments making the remainder. This shows the data set contains a good variety of comment types. Interestingly amongst block style comments Doxygen/Javadoc style comments seem to dominate. This is likely because Doxygen/Javadoc comments offer much more utility than just a standard block comment has to

offer. Looking more at individual language, another interesting detail that we found in TABLE 11 is that Java and C# use regular block comments very rarely, preferring to use Doxygen and Javadoc for block commenting. Equally interesting is the fact that though C/C++ is less likely to use Doxygen and Javadoc in commenting, based off of the data we gathered. Additionally, C/C++ maintain a higher average length of block type comments then both Java and C# in our data set. C# has the smallest average block comment length in our data set being less than 60% the average length of a C/C++ comment. As we randomly drew full comments, this also one reason why C/C++ has more comment lines.

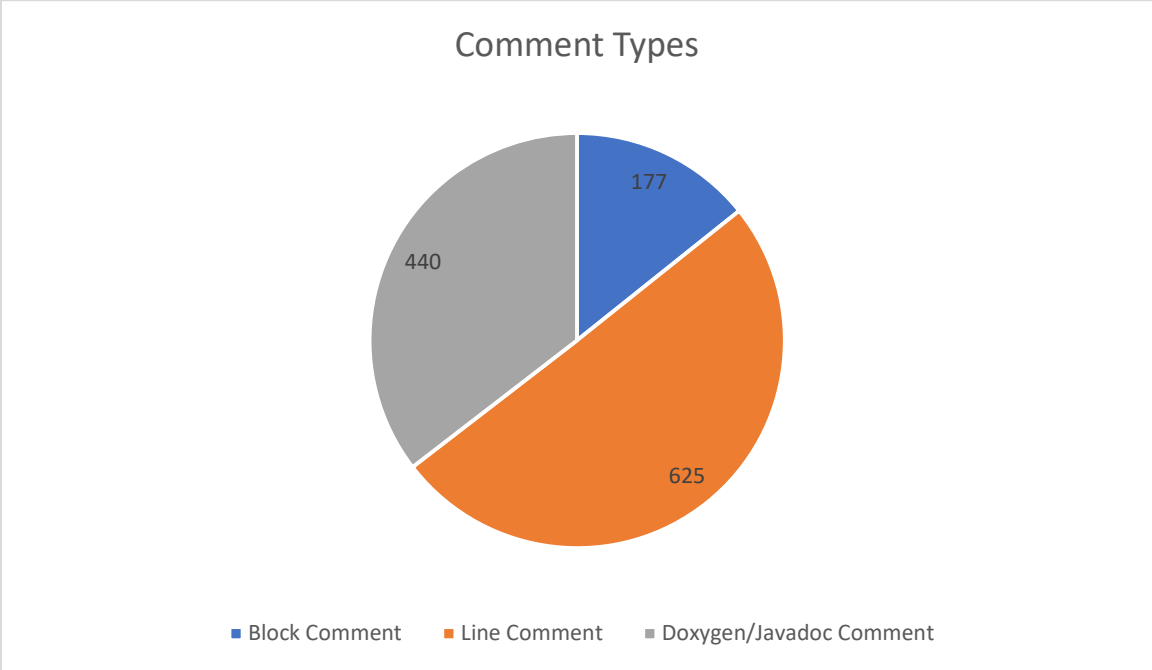

Figure 1. This figure provides a breakdown of the comments in the gold set by type. Line comments make up almost exactly 50% of the comment types. Doxygen/Javadoc comments make up another approximately 30% of the comment types with Block comments making the remainder.

TABLE 12 provides information on the commented-out code and other related information in our data set broken down by language. The first column contains the language with C and C++ having been combined. The number of sample lines is the total number of lines

of comments for the particular language, these are largely C/C++. The lines of commented-out code are the total number of lines which are commented-out code. The lines containing code references are a count of the lines which reference pieces of code but are not true lines of commented-out code. The lines containing standard terms are lines which contain standardized terms such as virtual, void, and int. C# and C/C++ have a much higher rate of containing standard terms than Java, this is due to the fact that these identifiers are typically not present unless there is commented-out code which Java had no samples of in our gold set. Interestingly, C# has over double the commented-out code as compared to C/C++ with over 20% of the comment-lines collected being commented-out code.. This could be an interesting point to investigate in the future. Do C# projects contain more commented-out code, or is this due to our sampling?

TABLE 13 are both described in detail here. The line count for statement LOC as well as the line count for comments is attained using XSLT scripts we wrote and ran on srcML and pulled out comments (each on a separate line) or code, respectively. For code we also removed blank lines. We then took the line count. For the comments, we wrote the XSLT parser such

that if more than one comment appeared on the same line it reported them as separate lines. The comment count is attained using the grep command of srcML. The number of authors is found using git log. The first of the two tables details all 78 projects while the second only details the projects that made it through random selection. The first column gives the name of the project. The second column gives all of the languages that were used in the project that srcML is capable of parsing. The third column gives the total number of lines of statement code of each project capable of being parsed by srcML. The fourth column provides a total count of the number of comments present in the source code that srcML is capable of parsing. The fifth column provides the total number of lines of comments present in the gold set from each project. The sixth column provides the total number of lines of commented-out code present in the gold set from each project. The seventh column is the total number of authors in each project as listed on GitHub. The eight column provides the coverage of comments when compared to lines of code (i.e., comments LOC / statement LOC). The ninth column gives what percentage of the data set that project contributes . We learned some interesting things from the creation of these tables. First, even though only 26 of the 78 projects made it through random selection, as many as 2606 authors contributed to the creation of these comments (minimum of ~26). Another interesting finding that we noticed during the creation of these tables is that C++ and C are much more prevalent even when downloading projects that were specifically listed as being Java or C#. C++ code was present in 52 of the 78 projects, and C code was present in 46 of the 78 projects. This is in comparison to Java which was present in 24 projects and C# which was present in 21. These tables provide us with a large amount of interesting information. First, folly, which makes up 42% of the gold set, is not an extremely large file. However, the size of the comments in its source files are extremely large with one particular block comment being 80 lines long. Another

reason is that four files were randomly selected from folly. Having these files be such a portion of the gold set is not seen as a large threat to validity due to the files having 29 separate authors who contributed to these files, all whom could have written and or edited comments and contributed to the diversity of the comments. Aside from this one project, only one other project makes it above 10% of the gold set, meaning that many other projects make up the remainder of the set. Even with the outliers there is an average contribution of a project to the gold set at 3.85 percent (median 1.28 percent). That is, besides the outlier there is quite a bit of variability in the data set. Another interesting point that we found in this analysis is that there is a wide spread of percentages of comment vs code coverage. For example, the lowest percentage of coverage is nearly 0 while the highest is a 1. Interestingly, we also find that the amount of comments in projects various even among projects of similar size, although there is a correlation of .75 (fairly strong) between project SLOC and lines of comments. So, the larger the project (amount of code) in our data set, the more comments it will generally have. However, there is a .04 deviation between the average ratio and the median of SLOC to lines of code, with the average ratio being .23 and the median being .19 and a correlation of only .39 . That is, the amount of code that a project has does indicate as well how well covered by comments that code is. Lastly, something else that we noticed in this analysis is that the projects on average have 64 authors. This is important for our study because each author will bring their own style to code and comments when writing, which is part of the reason that automated detection is so complex.

TABLE 11. BREAKDOWN OF THE GOLD SET BY COMMENT TYPE AND LANGUAGE. THE FIRST COLUMN CONTAINS THE LANGUAGE, DUE TO ISSUES WITH DETECTION C AND C++ HAVE BEEN COMBINED AS .H FILES DETECTED BY SRCML ARE LABELED AS C++ FILES. THE NUMBER OF SAMPLE LINES IS THE TOTAL NUMBER OF LINES OF COMMENTS FOR THE PARTICULAR LANGUAGE, THESE ARE LARGELY C/C++. THE NUMBER OF BLOCK COMMENTS IS THE NUMBER OF NON-DOXYGEN/JAVADOC BLOCK COMMENTS. THE NUMBER OF LINE COMMENTS IS THE NUMBER OF SINGLE LINE COMMENTS. THE NUMBER OF DOXYGEN/JAVADOC COMMENTS IS THE COUNT OF BLOCK COMMENTS MADE IN THE DOXYGEN/JAVADOC STYLE ONLY. TOTAL NUMBER OF COMMENTS IS THE TOTAL OF BLOCK, LINE, AND DOXYGEN/JAVADOC COMMENTS. THE AVERAGE LENGTHS OF DOXYGEN/JAVADOC/BLOCK COMMENTS CONTAINS THE AVERAGE LENGTH OF NON-LINE COMMENTS.

Language	**Number of Sample Lines**	**Number of Block Comments**	**Number of Line Comments**	**Number of Doxygen/Javadoc Comments**	**Total Number of Comments**	**Average Lines of Doxygen/Javadoc/Block Comments**
C#	287	4 (2.25%)	98 (55.06%)	76 (42.70%)	178	2.36
C/C++	2408	168 (17.20%)	505 (51.69%)	304 (31.12%)	977	4.03
Java	239	5 (5.75%)	22 (25.29%)	60 (68.97%)	87	3.34
Total	2934	177 (14.25%)	625 (50.32%)	440 (35.43%)	1242	3.74

TABLE 12. GIVE INFORMATION ON LINES OF COMMENTED-OUT CODE BROKEN DOWN BY LANGUAGE. THE FIRST COLUMN CONTAINS THE LANGUAGE, DUE TO ISSUES WITH DETECTION C AND C++ HAVE BEEN COMBINED AS. THE NUMBER OF SAMPLE LINES IS THE TOTAL NUMBER OF LINES OF COMMENTS FOR THE PARTICULAR LANGUAGE, THESE ARE LARGELY C/C++. THE LINES OF COMMENTED-OUT CODE ARE THE TOTAL NUMBER OF LINES WHICH ARE COMMENTED-OUT CODE. THE LINES CONTAINING CODE REFERENCES ARE A COUNT OF THE LINES WHICH REFERENCE PIECES OF CODE BUT ARE NOT TRUE LINES OF COMMENTED-OUT CODE. THE LINES CONTAINING STANDARD TERMS ARE LINES WHICH CONTAIN STANDARDIZED TERMS SUCH AS VIRTUAL, VOID, AND INT.

Language	**Number of Sample Lines**	**Lines of Commented-out Code**	**Lines Containing Code references**	**Lines Containing Standard Terms**
C#	287	58 (20.21%)	0	12 (4.18%)
C/C++	2408	231 (9.59%)	51 (2.12%)	85 (3.53%)
Java	239	0	4 (1.67%)	0
Total	2934	289 (9.85%)	55 (1.87%)	97 (3.31%)

TABLE 13. BELOW IS A CONCISE FORM OF TABLE 9 CONTAINING ONLY THE PROJECTS THAT MADE IT THROUGH THE RANDOM SELECTION PROCESS. THE FIRST COLUMN GIVES THE NAME OF THE PROJECT. THE SECOND COLUMN GIVES ALL OF THE LANGUAGES THAT WERE USED IN THE PROJECT THAT SRCML IS CAPABLE OF PARSING. THE THIRD COLUMN GIVES THE TOTAL NUMBER OF LINES OF CODE OF EACH PROJECT CAPABLE OF BEING PARSED BY SRCML. THE FOURTH COLUMN PROVIDES A TOTAL COUNT OF THE NUMBER OF COMMENTS PRESENT IN THE SOURCE CODE THAT SRCML IS CAPABLE OF PARSING. THE FIFTH COLUMN PROVIDES THE TOTAL NUMBER OF LINES OF COMMENTS PRESENT IN THE GOLD SET FROM EACH PROJECT. THE SIXTH COLUMN PROVIDES THE TOTAL NUMBER OF LINES OF COMMENTED-OUT CODE PRESENT IN THE GOLD SET FROM EACH PROJECT. THE SEVENTH COLUMN IS THE TOTAL NUMBER OF AUTHORS IN EACH PROJECT AS LISTED ON GITHUB. THE EIGHT COLUMN PROVIDES THE COVERAGE OF COMMENTS WHEN COMPARED TO LINES OF CODE. THE NINTH COLUMN GIVES THE PERCENTAGE OF COMMENT COVERAGE ACROSS THE WHOLE CORPUS FOR EACH PROJECT WITHIN THE CORPUS.

Name	Language	Number of Authors	Statement LOC	Lines of Comments	Comment Lines in Gold Set	Commented-out code in Gold Set	Ratio of Comments to Code	Percentage of Corpus
8cc	C	9	9791	712	1	0	0.07	0.0341
aleth	C++	148	90154	9979	134	0	0.11	4.5656
algorithms	C++	134	9390	2809	1	0	0.30	0.0341
asio	C++	28	105180	38013	47	0	0.36	1.6014
aws sdk cpp	C++	57	2503490	2509104	217	0	1.00	7.3935
cdt	Java/C++/C	154	1122144	450027	85	0	0.40	2.8961
C Plus Plus	C++/C	100	5767	576	31	26	0.10	1.0562
DynamicExpresso	Java/C#	15	15245	4521	9	0	0.30	0.3066
EnsageSharp	C#	8	36262	3345	13	13	0.09	0.4429
faster than c	Java	3	446	18	1	0	0.04	0.0341

folly	C++/C	501	286037	84631	1249	80	0.30	42.5554
j2c	Java/C/C++	1	10948	350	27	0	0.03	0.9199
j2objc	Breaks CLOC	66	881870	636880	63	0	0.72	2.1465
libigl	C++/C	82	92359	28712	385	94	0.31	13.1175
markdowndeep	Java/C#	6	51833	10893	24	0	0.21	0.8177
MessagePack Csharp	C#	45	45564	4585	32	6	0.10	1.0903
mini c	C	1	3454	6	11	0	0.00	0.3748
MissionPlanner	C#	71	798373	240724	188	42	0.30	6.4055
mono	C#/C/C++	747	6049345	1295150	11	11	0.21	0.3748
nativejson benchmark	C++/C	32	16083	1337	42	7	0.08	1.4310
oclint	C++/C	29	21478	478	59	13	0.02	2.0102
osrm backend	C/C++	109	195834	30768	42	0	0.16	1.4310
QR Code generator	C/Java/C++	2	4088	1206	53	0	0.30	1.8058
QuantLib	C++/C	99	376347	72562	163	11	0.19	5.5537
stb	C/C++	158	69880	11952	14	0	0.17	0.4770
XobotOS	Java/C++/C#/C	1	2002448	1125133	33	0	0.56	1.1244

Total		5022.00	18999004.00	7762476.00	2935.00	303.00		100.0000
Average		64.38	243576.97	99518.92	37.63	3.88	0.23	1.2821
Median		32.50	19934.00	3687.50	0.00	0.00	0.19	0.0000

CHAPTER 6.

Data Analysis

In this chapter we present our method for data analysis as well as the data analysis efforts that have failed. When attempting to determine whether or not a comment, or a line of a block comment, is a piece of commented-out code things become much more complicated then when a trained programmer is simply able to review it. Over the progress of this research we investigated several approaches: a syntax-based approach (Section 6.1.), a bag of words approach (Section 6.2.), and a frequency-based approach (Section 6.3.). The first of these methods which had proven to be fairly ineffective on larger test cases is what we would call the syntax-based approach.

6.1. Syntax-based Approach

In the syntax-based approach the method for analysis of lines is simplistic and is broken down into a series of 3 different checks. The method is similar to some of the work done by Bacchelli et al. [Bacchelli et al. 2010a]. The first check, run on every line, is whether or not the line contains a semicolon, which has the direct ability to generate a number of false positives depending on the writing style of the programmer (i.e., if they tend to use semicolons in standard

comments). The second and third checks rely both on checking for the opening and closing of parenthesis and curly braces respectively. What we do when we are checking for these syntax markers is parse the line one character at a time checking for semicolons, parenthesis and curly braces. If a semicolon, open and closing parenthesis, or open and closing curly brace is found we mark the line as containing code, however if the closing brace or closing parenthesis is not found then the comment would not be considered commented-out code. Further, there are varies times that commented-out code may not contain a semicolon, such as inside a simple if statement. This was not something that we had at first expected to be a problem, and in fact it was, as in cases where optional snippets of code had been commented out such as

```
//if(x > 10){
 if(x == 10){
```

(assuming the line following this comment is another if statement), the automation process would disregard these sections as it did not find the opening or closing piece that it was looking for. The second approach, which was considered but never implemented is a bag of words approach.

6.2. Bag of Words Approach

This bag of words approach is not to be confused with the keyword approach mentioned earlier in the data collection chapter, which proposes the use of common terms as an additional method of verification. Rather, the concept of this approach is to break down an entire piece of source code and create a bag of words from it, which could then be used to cross check comments for terms that are present in the line which are found to be frequent in the bag of words. For example, if we scan an entire source-code file broken down by the whitespace and then scan the comments for matching terms then we could potentially identify variables and

function names held within the comments. While this could be helpful in finding commented-out code that is modifying common variables or using common variables as part of a greater equation, it has a number of strong failing points. First, when considering variable names, one time use variables, variables created in a piece of commented-out code, and commented out functions, they are all highly likely to be ignored due to the fact that in comparison to other terms in the bag of words they may only have an appearance rate of 1-3 times in the entire source code where as a term like `int`, `void`, or `count` will appear much more frequently. This is where frequency comes into play and why bag of words is bound to fail in this case. If a piece of commented-out code contains a variable that occurs nowhere else, it is not going to be caught. The other issue with this method comes down to explanations of how code functions. In this case, in thorough documentation a programmer may reference function names and variable names within an English prose comment. Too many of such references will cause a false positive. This brings us to our third and most current approach, what we call the frequency-based approach.

6.3. Frequency-based Approach

The original basis of the frequency approach is derived from the works of Dvorak [Nakic-Alfirevic and Durek 2004] who is famous for designing alternate versions of the key board layout used on English typewriters and computers. Dvorak examined which letters are most frequently used in the English language and relocated their positions to allow for easier and less strenuous typing. This concept of common characters in English words brought forth a very powerful idea, what if we check the frequency of ASCII characters found in lines of both English prose style comments and commented-out code and compared them against each other? What the data shows us when analyzing the results of these frequencies is that there are key differences between English prose and commented-out code, and not only are these differences present, some of them are quite extreme. As shown in Figure 2, there are thirteen symbols which have a

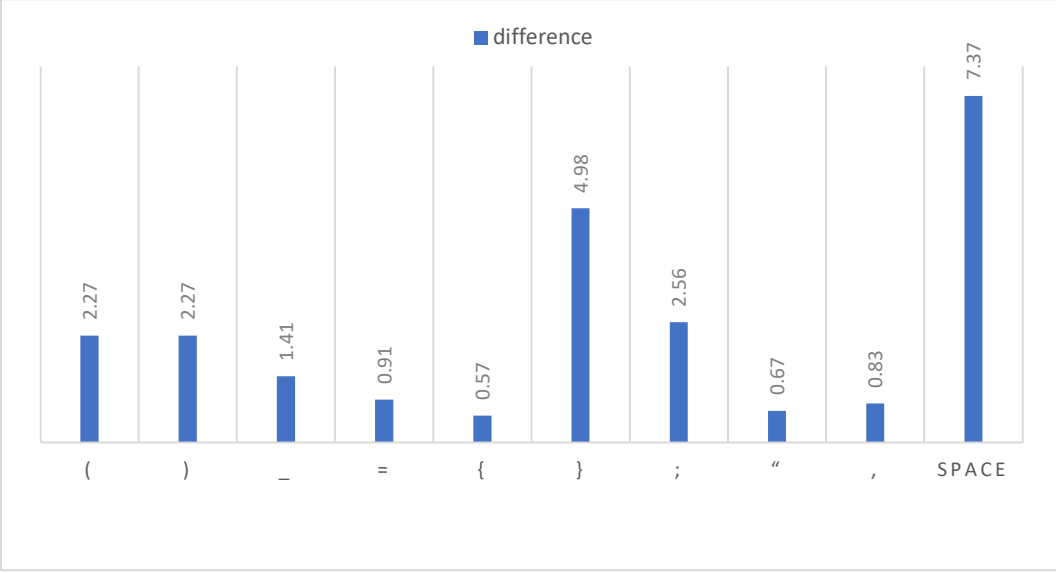

Figure 2. From left to right 10 symbols which have the greatest difference between prose and commented-out code are shown. Parenthesis both open and close, underscore, equals, curly brace open and close, semi colon, quotation, comma and space. Each value represented in the graph is the difference between commented-commented out code and English Prose, all of which favor commented-out code.

frequency near to or greater than one percent more common in commented-out code versus in a standard comment.

The most staggering of these numbers is actually the frequency of spaces found in commented-out code, for which a number of observations are made. Likely, one of the largest reasons for this is good indentation practices leaving large amounts of whitespace in commented-out code. However, upon closer analysis of some samples where spacing rates were particularly high, it was noted that the average character length of terms tended to be much shorter in commented-out code. A prime example is:

```
//    i = a + b;
```

In this example the average size of a term is roughly 1 character and a total of five non-space based characters being present, now when you consider the fact that there is also eight spaces in the line, that means that the spaces are making up over 50% of the lines total number of characters. Further, taking into consideration Mayzner's work and Googles [Norvig] follow-up research using modern computational methods, it has been determined that the average length of an English word is 4.7 characters. This means that in the same space of total characters, fifteen, on average 3 words would fit, assuming that it ends in a period and contains 2 spaces. The average length of a comment in our study is 40.5 characters. Importantly, what this means is that spaces would be making up about 13% of the total number of characters in the line which is roughly 78% less spaces than the commented-out code example. These methods continue to hold true at different frequencies for a wide variety of different characters besides the ones mentioned previously, though in smaller amounts. One of the benefits of using a method like this is by scanning a variety of source code you are able to create frequency distributions that are consistent. In the case of the final frequency distributions used in this research the values are

pulled from code and comments from amongst different projects, ensuring that it gets a good general representation of what a frequency distribution should look like and helps with generalizability and avoiding overfitting. Of course, an added benefit to this is if you are examining code and comments that are required to follow a very specific structure then the process is equally as beneficially once the scanning process is complete.

We now explain and illustrate via an example how we computer frequencies for a comment line. The way this is done is by taking each line one at a time and verifying each character converted to lowercase for normalization against a dictionary of characters and then consequently stored in the dictionary. Once the entire line has been read and all characters have been stored and a final count of characters is obtained, the frequency of each character is calculated and stored in a list. This ensures that they remain in order by using key based verification. This was performed by a simple python program that we wrote to automate the process. TABLE 14 below is an example of how the calculation is performed by dividing the count of each symbol by the total count of all symbols.

```
//a = sqrt(b**2 + c**2)
```

TABLE 14. BREAKDOWN OF THE MATHEMATICAL FREQUENCIES OF THE ABOVE COMMENT LINE.

Symbol	Count	Frequency
A	1	.048
Space	4	.195
=	1	.048
S	1	.048
Q	1	.048

R	1	.048
T	1	.048
(	1	.048
B	1	.048
*	4	.195
2	2	.095
C	1	.048
)	1	.048

CHAPTER 7.

Decision Trees

In this chapter we present decision trees, how they are built, and what their output looks like. Within the field of machine learning there are many different options, not just in algorithms, but also in preconstructed implementations. Of course, one can also always take the option of producing an implementation of an algorithm themselves, however for the sake of transparency, reproducibility, and validity we use verified implementations from within the scikit-learn Python library.

With machine learning, choosing what type of algorithm, you are going to use for data is extremely important. Sometimes an algorithm cannot function at all with the data you have available, and other times using the incorrect algorithm will cause poor fit or present results that contradict the output. For our data there are three major factors that we must consider: first, our character frequencies are completely non-linear meaning that any machine learning algorithms that rely on the data being linear immediately will not work. The second factor that is of particular importance is that we are working with classification of two distinct classes (i.e., English pose or commented-out code), so choosing a machine learning algorithm that is known

for classification is equally as important. Thirdly, decision trees are able to handle blank data very well, this is extremely important when considering our data as comments (especially short comments) won't have very many different characters as shown in the example frequency of a comment in TABLE 15. Considering these three factors the obvious choice of machine learning algorithm is the decision tree.

In scikit-learn's current state their decision tree algorithm is based off an optimized version of the Classification and Regression Trees (CART) algorithm. This is a variation of the popular C4.5 algorithm which proceeded ID3 style decision trees [scikit-learn developers]. One of the major changes that came with the C4.5 algorithm is the ability to handle non-categorical data, as well as, a new method for pruning that focused on pruning if a rules precondition improved without the pruned node [scikit-learn developers]. Decision trees require the data used to train the tree to be as balanced as possible. This is because at its root, a decision tree is a series of binary decisions and the optimization of such a method requires this sort of distribution [scikit-learn developers]. The ability to handle various types of data, non-linear data, and work well for both classification and regression are not the only reasons why we chose decision trees. Decision trees can be fully visualized [SKLearn 2019], which makes them both very easy to understand and equally easy to explain. These ideas are shown in Figure 3 which is an example of a decision tree made using the Iris data (a commonly used example). Each node is a rule (or decision) that asks a simple yes or no question with is a value less than or greater than a certain number being a very common method. Before we discuss how to classify an instance, we now go over the contents of each decision node as shown in the figure. Class is the majority class with the dataset for that point in the decision tree. Colors (green = versicolor, orange = setosa, purple = virgincia) are also used to show this with white indicating no majority. The values

section shows the exact distribution of each class within a zone of the tree, while the sample is the total number of samples (as part of training set) in that part of the tree. The Gini, a measure of statistical dispersion, is the value that shows how effective a decision node is. The farther down a decision tree you go the lower the Gini will generally become with leaf nodes always having a Gini of 0. At the top (except in leaves), is the Boolean decision for the decision node. As an example, take an iris with a petal-length of 5.13 and a petal-width of 1.46. Starting from the root, since 5.13 is greater than 2.45 so we immediately know that is not a setosa so we travel down the right side of the tree. With a petal width of 1.46 being less than 1.75 we travel down to the left of our current node. With the petal length of 5.13 being greater than 4.95 we again travel down to the left of our current node. With a petal width of 1.46 being less than 1.65 we travel down to the right from our current node identifying the iris as an outlier of the virgincia class.

TABLE 15. THE TABLE SHOWS THE FREQUENCIES OF EACH SYMBOL IN A SAMPLE LINE. IT IS IMPORTANT TO NOTE HOW SPARSE THE MATRIX IS AS THIS IS THE CASE WITH MANY IF NOT ALL LINES.

a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**
0.0208	*0.0208*	0	0	*0.0416*	*0.0208*	0	0	*0.0208*	0
k	**l**	**m**	**n**	**o**	**p**	**q**	**r**	**s**	**t**
0.0208	*0.0208*	0	0	*0.0625*	*0.0416*	0	*0.0208*	*0.0416*	*0.0625*
u	**v**	**w**	**x**	**y**	**z**	**!**	**@**	**#**	**$**
0.0208	0	0	0	0	0	0	0	0	0
<	**,**	**>**	**.**	**?**	**/**	**space**	**~**	**`**	**\t**
0	0	0	0	0	0	*0.5625*	0	0	0
\n	**\f**	**\r**	**\b**	**\v**	**\0**	**unk**			
0.0208	0	0	0	0	0	0			

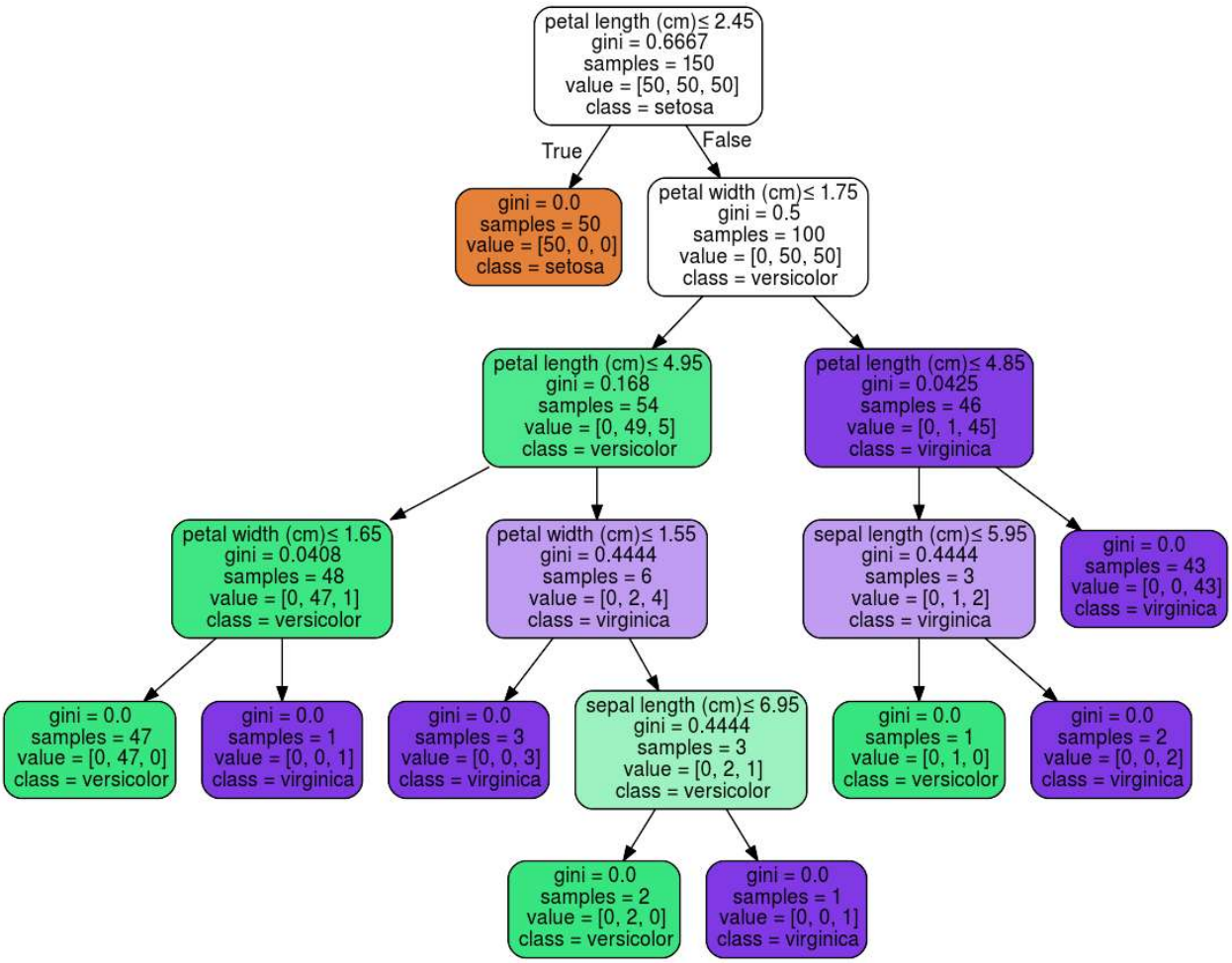

Figure 3. This is a decision tree sample from the well-known IRIS dataset. The colors separate the tree based on which class is the most present in a node. The class shows which of the three class a node primarily consists of. The sample shows how many values are in the node. The gini shows how major of a deciding factor a particular variable is, the lower the gini the more important it is. The top (on non-leafs) is the decision itself.

CHAPTER 8.

Experiment and Results

The first step of creating our decision tree is to form the data that we are going to use for training and testing into the input format required by scikit-learn's decision tree algorithm. This data needs to be turned into a PANDAS data frame. We opt to create our data frames as part of our frequency calculation program (written in Python). We output all of our data in the data frame format to a file in text format. Following this creation process, we use stratified K-fold cross validation to split our data. An illustration of Stratified K-fold cross validation is provided in **Error! Reference source not found.**. Stratified K-fold cross validation is the process of randomly splitting your data in a balanced manner based off of the number K, being the number of folds, that you want to make. The commonly accepted value for K is 5 to start, but it is important to ensure that your folds never become to0 small for the algorithm that you are working with. This is not an exact science, and many data scientists and other professionals will all say different things, but generally speaking this is where having good evaluation criteria to validate the fact that you are not over or under fitting comes into play.

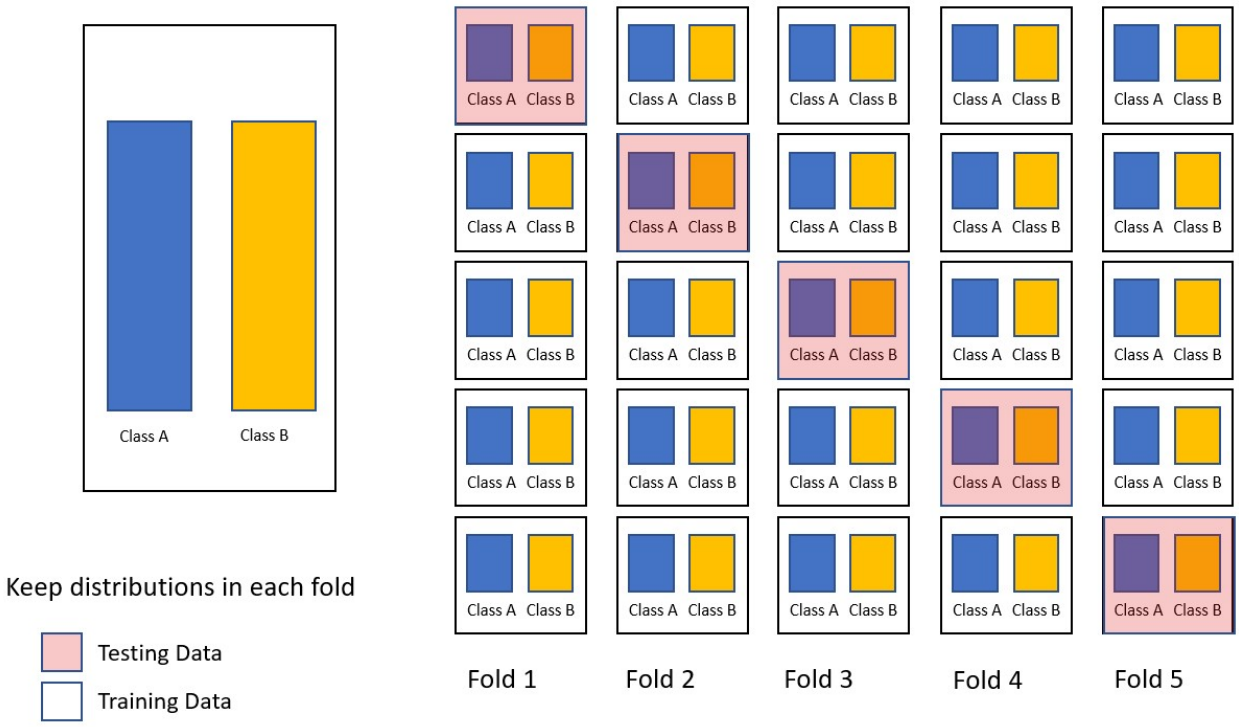

Figure 4. This is a breakdown of how stratified K-fold works. The main data is split into five parts with one of the five being used for testing and the remainder being used for training. With stratified K-fold it is important to ensure that the distribution is always equal amongst all of the groups.

For the purposes of validation, we will be using accuracy, precision, recall, and the F1 score (the harmonic mean of precision and recall). The combination of these four scores gives a good image of what our results look like when running the prediction model of the decision tree with our test data. If accuracy and the F1 score are too far apart then we know that our data is likely underfitting. If accuracy and F1 score are both consistently very close to 100% than we can also determine that our data is overfitting. We include precision and recall primarily as a means to further explain F1 score and as it is standardly reported. TABLE 16contains the calculations used for accuracy, F1 score, precision, and recall.

TABLE 16. THIS TABLE SHOWS EACH EQUATION USED AS A HUERISTIC IN THE ANALYSIS OF OUR RESULTS.

Accuracy Equation	$\text{accuracy} = \frac{\text{tp} + \text{tn}}{\text{tp} + \text{fp} + \text{tn} + \text{fn}}$
Precision Equation	$\text{precision} = \frac{\text{tp}}{\text{tp} + \text{fp}}$
Recall Equation	$\text{recall} = \frac{\text{tp}}{\text{tp} + \text{fn}}$
F1 Score Equation	$\text{F1} = 2 * (\frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}})$

First, we test the approach by applying stratified K-fold cross-validation on decision trees using 250 lines of true comments and 250 lines of commented-out code to prevent a training bias based on a dominate class and then mathematically verified to ensure that the results were holding true. We chose 250 in order to apply K-fold cross-validation because we had a total of 303 lines of commented-out code and when we test against the rest of the data (using the model we developed with results of our stratified K fold cross-validation) we wanted to have enough unseen examples of commented-out code to detect. For the number of folds, we pick 5. We choose 5 folds as it is the common choice when applying stratified K fold. Additionally, we wanted to make sure that there are enough examples to properly train the tree. Afterwards, the questions generated by the tree (decision nodes) for each tree fold were checked against the initial findings of the research (i.e., experience gained when classifying the golds set and **Error! Reference source not found.**) that they were all mathematically sound questions. For example, the first question at the root of the tree asks whether or not a line is composed of less than or equal to 26.7% spaces and if the statement is true then the sample is likely an English prose. This is a highly reliable character as initial research indicated that on average comments are

constructed of approximately 24% spaces while commented-out code is constructed of approximately 33% spaces. Below is a breakdown sample of all 5 folds built into TABLE 17.

TABLE 17. THE FOLLOWING VALUES ARE THE RESULTS OF EACH FOLD FROM THE STRATIFIED K-FOLD CROSS VALIDATION.

Fold Number	accuracy	precision	recall	F1
1	98.50	98.15	99.07	98.60
2	97.00	100.00	94.00	96.91
3	97.50	95.65	98.88	97.23
4	98.50	99.00	98.02	98.51
5	98.00	99.01	97.09	98.04

The results of this table show us a number of this, first our decision tree neither underfits nor overfits our data. Our F1 scores are consistently in the 98% range, showing that both our precision and recall lay close to each other. Our precision rests very high, never dipping below 95%. This consistency in precision is extremely important, as it shows our tree is very good at targeting the commented-out code while finding very little false positives. Our recall provides very similar feedback to our precision, however there is one sample that fell below 95% to 94%. This recall again shows that our true positive rate, or the rate of commented-out code detection, is high enough to outweigh any false negatives. Finally, while accuracy is regarded generally as a heuristic that is not good by itself, it does reinforce what our F1, precision and recall shows. Our best results are represented in fold 5 in all four fields.

Following the completion of these five runs we ran the full 250 x 250 samples without folding to generate our final decision tree. This decision tree is shown in Figure 5. This figure shows our decision tree model. The two colors represent the classes, orange being a normal comment and blue being commented-out code. the samples show the number of samples which are in a node. The gini is the numerical representation of the importance of the gini, the lower

the score, the more important the value is. After creating our final decision tree, we ran a final test against the remainder of the gold set. This tree asks questions of the parts of the data that we find to be the most prevalent. For example, the root of the tree asks about the frequency of spaces, which is something that we have discussed in depth in this paper. Another example is the presence of semicolons, if the semi colon frequency is extremely low then we know it is likely not commented-out code. This is similar to the existence of an asterisk, if the frequency of an asterisk is low then we also know that the comment is likely English prose. Our resulting F1 score is 97.89%, the precision is 98.32% and recall is 98.23%. The resulting accuracy is 98% which alongside our other results shows we follow the same trend of our original results. As such, the application of the combination of the frequency-based approach and a decision tree works extremely well, and we are now able to answer RQ 3 affirmatively. We can automatically detect commented-out code using the character frequency in combination with decision trees within an acceptable margin of error?

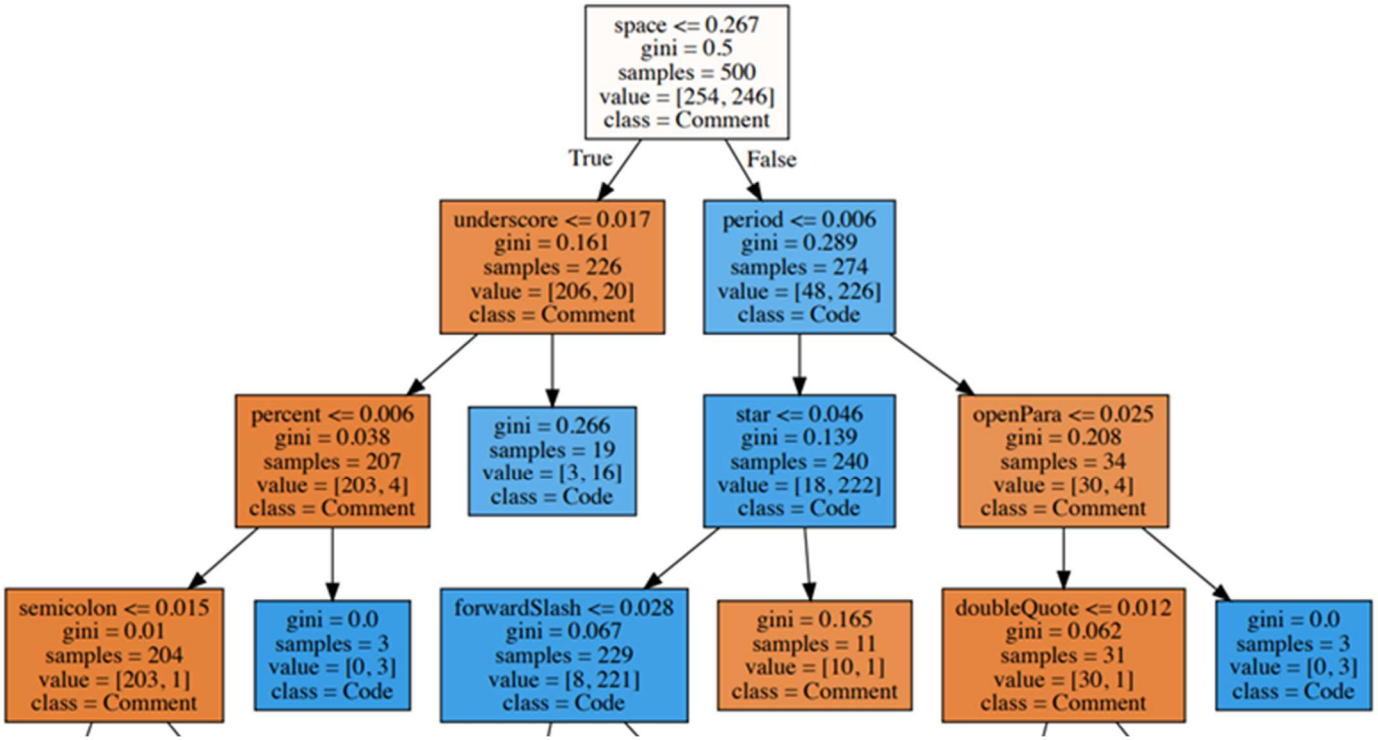

Figure 5. This figure shows our decision tree model. The two colors represent the classes, orange being a normal comment and blue being commented-out code. the samples show the number of samples which are in a node. The gini is the numerical representation of the importance of the gini, the lower the score, the more important the value is.

CHAPTER 9.

Commented-out code in Open-source Software

After verifying that our tool works, the next step was to do a full analysis on a full set of 50 new open-source projects. For this analysis, we use the tree constructed using all 250x250 comments/commented-out code from the previous chapter. First to check that the model works well enough for the data set. We selected the first 36,000 comment-lines from the 50 projects and classify them with the model. We then manually verified the classification in the following manner. We verified each comment-line that was identified as commented-out code. Then, we found the nearest comment-line above and below that appeared in the 36,000 that were classified as English prose and that we have not previously checked before. Although, this means we verified far less than 36,000, we validated 2 English prose comment line for every line of commented-out code. Unfortunately, the exact counts have been lost after computing the accuracy, precision, etc. but the number is believed to be well over 3,000 comment-lines, this took us approximately 40 hours to complete. As this was a simple check as opposed to the amount of processing needed for the gold set, this took far less time. TABLE 18 is the results for the over 3,000 comment-lines. The precision was lower than the original data set due to the decision tree finding numerous instances of code references, which appear very similar to commented-out code, inside English Prose comments. This of course had an effect on both accuracy and F1 score as well, lowering both of them respectively. Our results on these

comments-line show that we are in acceptable operating procedures for our decision tree algorithm. We can say this because while our precision is lower than our tests against the gold set, our goal is to identify the commented-out code which we succeed in as clearly shown in our recall. This also gives us a good representation of the distribution of commented-out code in open-source software. This very strong recall is what allows our F1 score to recover to nearly 88%.

TABLE 18. THIS TABLE IS THE HEURISTIC RESULTS FOR 36000 OF THE TOTAL SET OF COMMENTS WHICH WAS MANUALLY VERIFIED. THE PRECISION WAS LOWER THAN THE ORIGINAL DATA SET DUE TO THE DECISION TREE FINDING NUMEROUS INSTANCES OF CODE REFERENCES, WHICH APPEAR VERY SIMILAR TO COMMENTED-OUT CODE, INSIDE ENGLISH PROSE COMMENTS. THIS OF COURSE HAD AN EFFECT ON BOTH ACCURACY AND F1 SCORE AS WELL, LOWERING BOTH OF THEM RESPECTIVELY.

accuracy	precision	recall	F1
94.68	78.03	99.62	87.51

TABLE 19 details the 50 projects in this open-source study as well as the resulting number of lines of commented-out code. The line count for statement LOC as well as the line count for comments is attained using the same XSLT parser used in **Error! Reference source not found.**. The comment count is attained using the grep command on the srcML. The number of authors is found using git log. Due to the precision of about 78% we will be over reporting the amount of commented-out code, but the model is accurate enough to get us a rough estimate that we can use to view of the systems. The first column gives the official project name for each of the 50 projects. The second column provides the number of lines of code in each project. The lines of comments provide the total number of lines of comments in each project. The number of comments provides the combined count of line/block/Doxygen/Javadoc comments in each project. The final column is the number of lines of commented-out code amongst each of the 50

projects. These numbers provide some really interesting details about open source projects. First, we can note that open source projects have a wide variance in size, with one project having no comments at all and the largest having over 1.4 million lines of comments. This entire set contains 176,503 lines of commented-out code, with an average of 3,530 lines of commented-out code per project. However, this average is misleading as the median is only 123 lines of commented-out code. That is, the amount of commented-out code varies heavily from project to project with some having relatively large amounts of commented-out code. We will discuss this more when we talk about the percentage of comments that are commented-out code.

The ratio of comments to lines of statement code is remarkably similar to the original 78 projects that we selected our gold set from. In fact, the ratio of comments to lines of statement code in this study are .24 vs the .23 of the previous 78. This helps to show how similar the two sets are to one each other even though they are entirely different sets of projects, as they are less than 1% apart. Perhaps equally as interesting we see the same extreme outliers present in this data set as well. With the lowest ratio being incredibly near 0 and the highest being .92.

The percentage of comments that are commented-out code in each project on average is 4.43% with a median value of 2.76%. The lowest percentage of commented-out code coverage amongst all of the projects is 0% while the highest is 25.09%. 31 of the projects fall below the average value with 20 of the projects consisting of less than 2% commented-out code however only 4 projects contain no commented-out code at all. What this seems to imply is that there is a large deviation between projects containing either quite a bit of commented-out code or almost none at all and that the amount of commented-out code is project and perhaps authors specific. In future work, we will investigate this last point. Is the amount of commented-out code the work of a single small group of contributors or is it more pervasive among contributors.

When reviewing the average number of lines per comment we find that the median length of a comment is 1.64 lines. What this means is that almost all of these projects consist of mostly line comments. However, the average is 2.26 which shows there are some projects that make greater use of block-style comments. This chapter answers our **RQ 4**, How prevalent is commented-out code in open source software. In open-source projects, the amount of commented-out code various greatly from project to project, but over 90% of projects contain some amount of commented-out code.

TABLE 19. THIS TABLE DETAILS THE 50 PROJECTS IN THIS BLIND STUDY AS WELL AS THE RESULTING NUMBER OF LINES OF COMMENTED-OUT CODE. THE FIRST COLUMN GIVES THE OFFICIAL PROJECT NAME FOR EACH OF THE 50 PROJECTS. THE SECOND COLUMN PROVIDES THE NUMBER OF LINES OF CODE IN EACH PROJECT. THE LINES OF COMMENTS PROVIDE THE TOTAL NUMBER OF LINES OF COMMENTS IN EACH PROJECT. THE NUMBER OF COMMENTS PROVIDES THE COMBINED COUNT OF LINE/BLOCK/DOXYGEN/JAVADOC COMMENTS IN EACH PROJECT. THE FINAL COLUMN IS THE NUMBER OF LINES OF COMMENTED-OUT CODE AMONGST EACH OF THE 50 PROJECTS.

Project Name	LOC	Lines of Comments	Number of Comments	Lines of Commented-out Code	Ratio of comments to Statement LOC	Percentage of Commented-out Code	Average Lines per Comment
Android Common	6615	5730	921	160	0.87	2.79	6.22
AndroidUtilCode	37120	17327	3674	739	0.47	4.27	4.72
Calligraphy	1033	572	151	1	0.55	0.17	3.79
camerakit android	8402	3392	1234	58	0.40	1.71	2.75
canal	82043	19538	6575	137	0.24	0.70	2.97
cas	216998	65348	11896	451	0.30	0.69	5.49
ceph	1002980	110720	63741	7682	0.11	6.94	1.74
CircleImageView	399	24	6	0	0.06	0.00	4.00
cmder	610	48	36	0	0.08	0.00	1.33
cmus	38254	6288	2413	141	0.16	2.24	2.61

CNTK	205035	46405	45492	1081	0.23	2.33	1.02
CodeHub	25642	1023	1000	46	0.04	4.50	1.02
collectd	119059	28100	12183	195	0.24	0.69	2.31
CompleteSharp	804	34	16	-	0.04	-	2.13
conceal	38936	14656	4433	109	0.38	0.74	3.31
ConfuserEx	28084	3184	3009	87	0.11	2.73	1.06
csharplang	66	0	0	0	0.00	0.00	-
csl traffic	9590	701	628	6	0.07	0.86	1.12
ctci	24442	2059	1632	3	0.08	0.15	1.26
Divan	4558	1114	1103	18	0.24	1.62	1.01
dlib	382508	134719	54776	7751	0.35	5.75	2.46
entt	22345	12278	2167	526	0.55	4.28	5.67
fluent bit	764914	263795	93546	17953	0.34	6.81	2.82
folding cell android	888	193	80	7	0.22	3.63	2.41
graal	901893	402558	54025	1465	0.45	0.36	7.45
Hawk	27642	4595	2678	55	0.17	1.20	1.72
hexchat	60393	4591	3865	38	0.08	0.83	1.19

ip2region	3887	1557	653	9	0.40	0.58	2.38
JsBridge	2411	391	386	0	0.16	0.00	1.01
JUCE	502645	144319	55509	3859	0.29	2.67	2.60
Krypto trading bot	7384	163	132	5	0.02	3.07	1.23
libphonenumber	69101	12373	8451	706	0.18	5.71	1.46
luna	4908	2068	935	57	0.42	2.76	2.21
material compents android	69570	23435	6191	683	0.34	2.91	3.79
MvvmCross	100281	9148	8698	32	0.09	0.35	1.05
openFrameworks	162954	38384	33628	9049	0.24	23.57	1.14
OpenRCT2	496084	40975	27517	986	0.08	2.41	1.49
phppredis	16208	3387	2684	39	0.21	1.15	1.26
pocketsphinx	23121	8494	3057	738	0.37	8.69	2.78
POGOLib	12110	742	648	78	0.06	10.51	1.15
RIOT	1754804	1402961	1023087	101873	0.80	7.26	1.37
roslyn	2720159	48804	48679	14155	0.02	29.00	1.00
rufus	71458	19936	9094	572	0.28	2.87	2.19

sider	6574	484	415	87	0.07	17.98	1.17
Sourcetrail	408850	49061	44938	3233	0.12	6.59	1.09
Swashbuckle.AspNetCore	12609	1027	941	71	0.08	6.91	1.09
tiny AES c	676	172	130	11	0.25	6.40	1.32
vlmcsd	16542	8691	7152	686	0.53	7.89	1.22
wren	42873	8421	5145	542	0.20	6.44	1.64
xgboost	33488	6170	3975	323	0.18	5.24	1.55
Total	10549950	2980155	1663325	176503			
Average	210999	59603.1	33266.5	3602.102041	0.24	4.43	2.26
Median	27863	6229	3033	137	0.21	2.76	1.64

CHAPTER 10.

Threats to Validity

Throughout the duration of our research there have been a few different points that point to threats to validity that we both mitigate in current work and plan to eliminate in future studies. In section 10.1. , we will present our concerns about external validity brought by our choices in projects that our data artifact is based on. In section 10.2. , we consider the threats to internal validity.

10.1. External Validity

External validity covers any threats to validity that impact generalizability. The first threat has to do with how our random sampling was performed. Ideally, we would have sampled comments randomly from each project individually (and at an equal amount) instead of files from all projects and then from those files. This threat is mitigated in several ways. First, 26 projects are still represented in the gold set and although there is one project with a large percentage of the comments it is less than the majority, we ran our tests with and without the

parts of this project and saw almost no impact on the results. That is, the majority of comments come from separate projects. Secondly, in **Error! Reference source not found.** we perform a secondary study over 3,000 unseen comment lines and verify that the approach achieve extremely high recall and with a precision still almost 80%.

Another threat lies with us only having 2,935 lines coming from over 1,200 comments. Manual classification is a slow and tedious effort the process of creating this gold set taking two months. We feel that the number of comments is acceptable given the amount of time it takes to create the gold set and verify its correctness. In the future, this data set can be expanded. However, when executing our tests of our decision tree and on the comments from **Error! Reference source not found.** we are able to show that this gold set is highly effective, mitigating this external threat to validity.

A third threat to external validity has to do with working with multiple different languages. To try and mitigate this, we included C, C++, Java, and C#, however, in varying amounts. A related threat is that the gold-set contains far more C/C++ comment lines than it does Java and C#. Still, Java and C# account for nearly 18% of the comments. Coupled with the high accuracy, precision, recall, and F1 score in **Error! Reference source not found.**, this does not seem to be a significant threat. In the future, more Java and C# can be added to the gold set. Additionally, all the languages selected for study belong to the C family. A separate study is needed to show if our technique generalizes to other non-C family languages.

Lastly, our method relies on the way that the comments are written, meaning that the more programmers which had the potential to write these lines the better that our results will be. In this case we have identified that as many as 2606 authors may have had contact with these comments (reading/writing/modifying), though it is likely less than that as 2606 is the maximum

number. A possible minimum number is 26 (1 for each project). However, this seems to provide reasonable variation as shown by the study in **Error! Reference source not found.**.

10.2. Internal Validity

A threat to internal validity is any threat to validity that is caused by something that we did directly. A threat to internal validity in our project is the crux of our identification process, namely our frequency calculation program. In the case, the program was thoroughly tested and additionally we were performed mathematic verification of the results.

A second threat to internal validity comes from the way that we parse and obtain our comments from the source code, if the comments are changed in any way than we lose reliability of our frequencies thereby making our results no longer valid. However, we mitigate this threat by use of the parsing tool srcML which has been rigorously tested to prove that it preserves the integrity of the comments when pulling them from the source code.

XSLT scripts were used in order to generate our counts for our lines of comments and lines of code. We originally planned to use the cloc utility, but we found errors in how it counted, particularly in how it counted comments. If a line of code and a comment appeared on the same line, then it failed to count the comment. Additionally, it failed to count code under some encodings. Although, the XSLT script is chosen to be more accurate, there was at least one case that we noticed that a few comments were counted as code because of an encoding problem with one of the files. Additionally, we wrote the XSLT script to report comments that appeared on the same line as separate lines. We chose this specifically as we regard them as separate comments and to match more with the fact they are classified separately, but it does make comment count larger and so some may consider this number to be higher that it should be.

Lastly, the major threat to internal validity has to do with manual classification. Classification of comments into commented-out code and English prose was performed by a single person. This is mitigated by the careful collection and performing as second round of verification of the comments,

CHAPTER 11.

Future Works

We envision two primary enhancements that we believe need to be handled in the future to extend the power and validity of this research. The first focus of our future research is to handle the various levels of coding skill as well as bad coding practices that are in use today. The second focus of our future research is to be able to search merge history within version control to identify exactly when and by whom code has been commented out, as well as, the motivation for commenting-out code.

In **Error! Reference source not found.**, it is discussed that the scope of this study is limited, because it has only worked with eighty projects from GitHub. This choice was made with the idea in mind that we wanted to have a very well written sample of code to work with for the first iteration of this project and it did give us access to almost 100,000 lines of comments. However, the code in these projects tend to be very well written and fairly uniform, and while this does give us a good example of what code and comments should look like it does not account for junior and veteran programmers who use out of date coding styles. Of course, a third group of coders, those who are self-taught, and who lack common and good practices and

standards within our field also provide an additional layer of content that we wish to explore. When looking at these groups of programmers and their coding styles they have to potential to cause shifts in the data similar to the highly specific coding styles discussed earlier in this chapter. However, the difficulty here is that unlike with those coding styles which have established rules within their designs, the coding styles that we are talking about here are much harder to identify and will require a lot of research to automate their identification.

Finally, a big part of our future research, and one of the long-term purposes for this research is the ability to automate the process of locating exactly when commented-out code has been introduced into the code base. Once we can identify when commented-out code has been commented, then we can also figure out who actually commented out the code in the script. This allows us to ask the programmer exactly why they commented out the code in the first place hopefully find good solutions to the removal of this commented-out code so that when a project is finally shipped it will be much easier to maintain. Additionally, we gain the ability to track commented-out code as it enters and exits source code, develop a method for automatic removal and correcting developer behavior over time in order to prevent future issues and need for constant tracking.

CHAPTER 12.

Conclusion

The results of our analysis are definitive and show that we can use machine learning in order to detect commented-out code. We were able to accomplish this within a 5% confidence interval within the original 26 projects with an average precision of 98.36%, average recall of 97.42%, and an average F1 score of 97.86%. In addition, we create a gold set which was derived from these original 26 of the projects. Part of developing this gold set required the development of a comment taxonomy, which is covered in **Error! Reference source not found.**. Developing this taxonomy allowed us to answer both the question of what commented-out code is as well as the different ways to provide comments and commented-out code.

Following the testing of the projects used to form the gold set we moved on to 50 entirely new projects. These projects saw a reduction in precision to 78.03% due to the program we developed detecting lines that contained code but were not fully commented-out code themselves. However, our analysis showed that we succeeded in detecting commented-out code within an acceptable margin of error combined with the results from the original corpus when looking at both our recall of 99.62% and a F1 score of 87.51%. Furthermore, we were able to

use the results to determine the prevalence of commented-out code in open-source software projects.

The end result of all of our research is new information on the prevalence of commented-out code in open-source software. In addition to this we now have a fully functioning program which is capable of translating XML files to the datasets that we use for our decision tree. From there we are able to determine the prevalence of commented-out code which allows use to label each line and determine the exact location of the lines of commented-out code.

ABDALKAREEM, R., SHIHAB, E., AND RILLING, J. 2017. On code reuse from StackOverflow: An exploratory study on Android apps. *Information and Software Technology 88*, 148–158.

ABID, N. degree of Doctor of Philosophy. 195.

ABID, N.J., DRAGAN, N., COLLARD, M.L., AND MALETIC, J.I. 2015. Using stereotypes in the automatic generation of natural language summaries for C++ methods. *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 561–565.

ALLAMANIS, M., PENG, H., AND SUTTON, C. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. *arXiv:1602.03001 [cs]*.

ARAFAT, O. AND RIEHLE, D. 2009. The commenting practice of open source. *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications - OOPSLA '09*, ACM Press, 857.

BACCHELLI, A., D'AMBROS, M., AND LANZA, M. 2010a. Extracting Source Code from E-Mails. *2010 IEEE 18th International Conference on Program Comprehension*, IEEE, 24–33.

BACCHELLI, A., LANZA, M., AND ROBBES, R. 2010b. Linking e-mails and source code artifacts. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, ACM Press, 375.

BINKLEY, D., LAWRIE, D., HILL, E., ET AL. 2013. Task-Driven Software Summarization. *2013 IEEE International Conference on Software Maintenance*, IEEE, 432–435.

BORSTLER, J. AND PAECH, B. 2016. The Role of Method Chains and Comments in Software Readability and Comprehension—An Experiment. *IEEE Transactions on Software Engineering 42*, 9, 886–898.

CHEN, H., HUANG, Y., LIU, Z., CHEN, X., ZHOU, F., AND LUO, X. 2019. Automatically detecting the scopes of source code comments. *Journal of Systems and Software 153*, 45–63.

COLLARD, M.L. AND MALETIC, J.I. srcML. *srcML*.

CORTES-COY, L.F., LINARES-VASQUEZ, M., APONTE, J., AND POSHYVANYK, D. 2014. On Automatically Generating Commit Messages via Summarization of Source Code Changes. *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, IEEE, 275–284.

DÉTIENNE, F. 1990. Expert Programming Knowledge: A Schema-based Approach. In: *Psychology of Programming*. Elsevier, 205–222.

DOLFING, H. 2019. Case Study 4: The $440 Million Software Error at Knight Capital. In: *The Project Success Model*. Amazon.com Services LLC.

FLEXRA. Allegation of Open Source Non-Compliance Leads to Anti-Competitive Practice Lawsuit. .

FLISAR, J. AND PODGORELEC, V. 2019. Identification of Self-Admitted Technical Debt Using Enhanced Feature Selection Based on Word Embedding. *IEEE Access 7*, 106475–106494.

FLURI, B., WURSCH, M., AND GALL, H.C. 2007. Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes. *14th Working Conference on Reverse Engineering (WCRE 2007)*, IEEE, 70–79.

HAIDUC, S., APONTE, J., MORENO, L., AND MARCUS, A. 2010a. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. *2010 17th Working Conference on Reverse Engineering*, IEEE, 35–44.

HAIDUC, S., APONTE, J., MORENO, L., AND MARCUS, A. 2010b. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. *2010 17th Working Conference on Reverse Engineering*, IEEE, 35–44.

HAOUARI, D., SAHRAOUI, H., AND LANGLAIS, P. 2011. How Good is Your Comment? A Study of Comments in Java Programs. *2011 International Symposium on Empirical Software Engineering and Measurement*, IEEE, 137–146.

LINARES-VASQUEZ, M., CORTES-COY, L.F., APONTE, J., AND POSHYVANYK, D. 2015. ChangeScribe: A Tool for Automatically Generating Commit Messages. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, IEEE, 709–712.

LIU, M., LANG, B., AND GU, Z. Calculating Semantic Similarity between Academic Articles using Topic Event and Ontology. 21.

MALETIC, J.I. AND KAGDI, H. 2008. Expressiveness and effectiveness of program comprehension: Thoughts on future research directions. *2008 Frontiers of Software Maintenance*, IEEE, 31–37.

MCBURNEY, P.W. AND MCMILLAN, C. 2016. Automatic Source Code Summarization of Context for Java Methods. *IEEE Transactions on Software Engineering 42*, 2, 103–119.

MOVSHOVITZ-ATTIAS, D. AND COHEN, W.W. Natural Language Models for Predicting Programming Comments. 6.

NAKIC-ALFIREVIC, T. AND DUREK, M. 2004. The Dvorak keyboard layout and possibilities of its regional adaptation. 6.

NORVIG, P. English Letter Frequency Counts: Mayzner Revisited or ETAOIN SRHLDCU. https://norvig.com/mayzner.html.

PAPER-ESEM-2011.PDF. .

REN, Y. AND JI, D. 2019. Learning to Detect Deceptive Opinion Spam: A Survey. *IEEE Access 7*, 42934–42945.

SCIKIT-LEARN DEVELOPERS. 1.10. Decision Trees. *scikit-learn*. https://scikit-learn.org/stable/modules/tree.html.

SKLEARN. 2019. 1.10 Decision Tree 1.10.1 Classification. https://scikit-learn.org/stable/modules/tree.html.

SONG, X., SUN, H., WANG, X., AND YAN, J. 2019. A Survey of Automatic Generation of Source Code Comments: Algorithms and Techniques. *IEEE Access 7*, 111411–111428.

STEIDL, D., HUMMEL, B., AND JUERGENS, E. 2013. Quality analysis of source code comments. *2013 21st International Conference on Program Comprehension (ICPC)*, IEEE, 83–92.

STOREY, M.-A. 2005. Theories, methods and tools in program comprehension: past, present and future. *13th International Workshop on Program Comprehension (IWPC'05)*, IEEE, 181–191.

UNITED STATES DISTRICT COURT NORTHERN DISTRICT OF CALIFORNIA. 2017. Artifex Software, Inc. v. Hancom, Inc. https://casetext.com/case/artifex-software-inc-v-hancom-inc.

VAUGHAN-NICHOLS, S. 2015. VMware sued for failure to comply with Linux license. https://www.zdnet.com/article/vmware-sued-for-failure-to-comply-with-linuxs-license/.

VON MAYRHAUSER, A. AND VANS, A.M. 1995. Program comprehension during software maintenance and evolution. *28*, 8, 44–55.

ZAIDMAN, A., HAMOU-LHADJ, A., GREEVY, O., AND ROTHLISBERGER, D. 2008. Workshop on Program Comprehension through Dynamic Analysis (PCODA'08). 2.

ZHOU, S., XU, X., LIU, Y., CHANG, R., AND XIAO, Y. 2019. Text Similarity Measurement of Semantic Cognition Based on Word Vector Distance Decentralization With Clustering Analysis. *IEEE Access 7*, 107247–107258.