TABLE OF CONTENTS

Page

CHAPTER 1 Introduction 1

CHAPTER 2 Related Work 7

2.1. Taxonomy 7

2.2. Detecting Code in Unstructured Text 8

2.3. Comment Quality 10

CHAPTER 3 Background 12

3.1. Decision Trees 12

srcML 16

CHAPTER 4 Taxonomy of Comments 23

CHAPTER 5 Data Collection 36

5.1. Corpus Selection 36

5.2. Comment Extraction 37

5.3. Manual Classification 38

5.4. Analysis of the Gold Set 48

5.5. External Validity 57

5.6. Internal Validity 59

CHAPTER 6 Approaches 60

6.1. Syntax-based Approach 60

6.2. Bag of Words Approach 61

6.3. Frequency-based Approach 62

6.4. Internal Validity 65

CHAPTER 7 Experiment and Results 66

CHAPTER 8 Commented-out code in Open-source Software 72

CHAPTER 9 Future Works 80

CHAPTER 10 Conclusion 82

LIST OF FIGURES

Figure .....Page

Figure 1. 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. 16

Figure 2. 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. 50

Figure 3. 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. 62

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. 67

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. 71

LIST OF TABLES

Table .....Page

TABLE 1. The table shows the frequencies of each symbol in a random comment line. The majority of characters are blank (0). 15

TABLE 2. 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. The sample contains a comment as marked up in srcML. 19

TABLE 3. This is a sample of an XML archive file generated using srcML on a directory. The primary difference between this and a standard srcML file is that a new archive unit tag is used to wrap each of the file unit tags. 20

TABLE 4. Results of applying an XPath command to a srcML archive. Each file unit is an element that matched the query (e.g., a comment). The filename attribute for each matched element is the file from which the element is found. The item attribute numbers the entry found in each file. 22

TABLE 5. Provides a detailed description of each of the six types of comments, line, block, #IF 0, IF(0), Doxygen, Javadoc. 24

TABLE 6. 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 remaining contain snippets of commented-out code. 27

TABLE 7. 5 examples of block comments. The first three contain standard English prose, the remaining contain commented-out code. 29

TABLE 8. 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. 33

TABLE 9. The top example is one that contains code which has been commented out using the preprocessor method #if. The bottom example contains the same commented-out code which has been commented out using a standard if(0) block. 35

TABLE 10. 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 is the total number of authors in each project as listed on GitHub. The fourth column gives the total number of lines of code of each project capable of being parsed by srcML. The fifth column provides a total count of the number of comments present in the source code that srcML is capable of parsing. The sixth column provides the total number of lines of comments present in the gold set from each project. The seventh column provides the total number of lines of commented-out code present in the gold set from each project 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..... 39

TABLE 11. Gold set sample. 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 gold set 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. 46

TABLE 12. Breakdown of the gold set by comment type and 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 that 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. 49

TABLE 13. Give information on lines of commented-out code 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. 51

TABLE 14. Below is a concise form of TABLE 10 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 is the total number of authors in each project as listed on GitHub. The fourth column gives the total number of lines of code of each project capable of being

parsed by srcML. The fifth column provides a total count of the number of comments present in the source code that srcML is capable of parsing. The sixth column provides the total number of lines of comments present in the gold set from each project. The seventh column provides the total number of lines of commented-out code present in the gold set from each project. 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. 53

TABLE 15. Breakdown of the mathematical frequencies of the above comment line. .. 64

TABLE 16. This table shows each equation used as a to evaluate our decision tree models. 68

TABLE 17. The following values are the results of each fold from the stratified k-fold cross validation. 69

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..... 73

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 number of lines of commented-out code amongst each of the 50 projects. The

ratio of comments to statement LOC is calculated by dividing the lines of comments by the LOC. The percentage of Commented-out code is calculated on a per project basis and represents the number of lines of commented-out code out of total lines of comments. The final column contains the aver number of lines per comment. 76

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, etc. change. One of the largest and most important parts of this is keeping the 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 for a successful product [Lehman 1996], as a program continues to change, the design and maintainability of the code will degrade. This loss of design leads to a lack of comprehension of the software which in turn leads to slowed development and increased cost [Parnas 1994]. In order to continue to maintain the system successfully, effort must be put in place by developers to increase their comprehension and to improve the comprehensibility and design of the system. Related to this, it is known that during evolution and maintenance of software, developers spend more than half their time comprehending code [Corbi 1989]. As such, continued research is required to provide support for the comprehension of code, and more specifically increasing the comprehensibility of code.

From previous studies, we know 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]. During this process of comprehension (either top-down or bottom-up), comments play an important role in understanding the code. In general, 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]. One problem with comments that violates this 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 it contradicts logic present in and around where it has been commented out. Alongside problems with comprehension, their comes the question of why a piece of code has been commented out in the first place. The commented-out code might be a security vulnerability, a feature that needs to be implemented later, reference code that was used to build another section of code, or code that does not work correctly? As evolution of the software product continues and if the commented-out code continues to exist, it is likely that the original

rationale for why the code was commented out in the first place has been lost. Without a known purpose, the commented-out code inhibits comprehension and should be removed.

In addition to comprehension, 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 system documentation generating approaches, feature location approaches, and other approaches that utilize comments, commented-out code can degrade these techniques performance. As an explicit example, Alhindawi et al. [Alhindawi et al. 2013] and many others apply LSI for feature location. The approaches use comments in building the LSI model. In this case, having commented-out code that is not reflective of actual program features may degrade approach performance. If the commented-out code is removed, it is likely that we can improve these approaches. Lastly, there is a legal precedent to the problem of commented-out code. Companies are being sued for leaving commented-out code that does not belong to them in their source code. [Flexra 2017] [United States District Court Northern District of California 2017] [Vaughan-Nichols 2015].

The goal of this thesis is to develop a method for automatically detecting commented-out code within a software project, so that it can be removed automatically. In order to develop the approach, we manually derive a gold set of approximately 3000 lines of comments which were pulled from a total of 80 projects via a random selection process. These 80 projects are distributed evenly amongst the four languages that are being studied in our research (C, C#, C++, Java). We classify each comment as commented-out code based on a taxonomy we introduce in CHAPTER 4. Next, for each line of comments, we compute the frequency of each individual character and store them in a dictionary. We then form a classification model using decision trees. We chose to use decision trees as it uses a supervised learning style, which makes it easy

to learn how the tree is differentiating between our two classes (i.e., if commented-out code or not). This allows us to do verification on the model. During the training process, stratified K-fold cross validation is used to split our data into training and testing. We further validate the model on the remaining gold set that was not included in the stratified K-fold, and on comments taken from an additional 50 systems. This whole process will be described in detail in CHAPTER 7 and CHAPTER 8. Finally, we apply the resulting model to all comments in the additional 50 open-source systems to determine the prevalence of commented-out code.

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. 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?**

  Here we investigate and taxonomize the ways of providing comments, and how code is commented-out.

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

  The difficulty 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. Here we investigate how to accurately 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 to the availability of open-source software, we take 50 open-source systems and apply our model to each line of comments to determine the extent of commented-out code.

This thesis makes the following contributions:

- A detailed taxonomy of comments and types of comments.
- 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 80 open-source projects.
- Differences between English prose (non-commented-out code) and commented-out code
- The development of an approach to automatically classify comments into 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.

The remainder of this thesis is laid out as follows. In CHAPTER 2, we go over related work. CHAPTER 3 provides an in-depth discuss on decision trees and the parsing tool srcML which is used as part of our approach. In CHAPTER 4, we cover our taxonomy on comments. In CHAPTER 5, we detail the process of data collection to form the gold set and describe the gold set. CHAPTER 6 gives the approach to how we process a comment for use with decision trees. Additional approaches we considered are also discussed in this chapter. CHAPTER 7 presents and discusses forming the model based on decision trees and the results of applying it to

the gold set. CHAPTER 8 presents and discusses the results of the open source study. CHAPTER 9 outlines our plans for future work based off of the results from this study. Finally, CHAPTER 10 gives the conclusion.

CHAPTER 2

Related Work

In this chapter, we present the related work. Despite the 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. Finally, in Section 2.3. , we focus on various works in comment quality, and the importance it has in code comprehension.

2.1. Taxonomy

In, Chen et al. introduce a comment taxonomy with four different types. The first type is the non-prose and low purpose Code Comments, which we call commented-out code. The second type is Task Comments, which are notes such as TODO or FIXME. The third type are IDE comments, which are special comments designed to communicate to the IDE directly, and the last type is 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 the objective of a comment. An example of this is whether or not a comment is related to a snippet of code or an entire function [Haouari et al. 2011]. Second, they consider the comment type. Here they define four types of comments: a code explanation, TODO comments, commented-out code, and licensing agreements [Haouari et al. 2011]. Next, they consider the style, which is only used in explanatory comments and defines whether or not the explanation is explicit or implicit (i.e., if you need to read the code to understand the comment) [Haouari et al. 2011]. Following style is 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, only the comment types is related. Haouari et al. define commented-out code as old code which remains in a comment and 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. In contrast, Haouari et al focus more 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 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 show that no individual method was enough to be consistently accurate for detecting code in emails. Furthermore, most of the 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 detection rate of 85-95% by using end of line in combination with regular expressions. However, this varies on the language it is trying to detect [Bacchelli et al. 2010a]. When considering their approach with special characters, they concentrate only on certain special characters. As such, they may miss characters that can be very important. For example, 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 nearly all ASCII characters. Additionally, unlike Bacchelli et al., we utilize decision trees.

Another example of the detection of code is provided by Abdalkareem et al. In their study, they pull 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 code in the code tags have to be five lines or greater. The reason they choose to exclude anything less than five lines is because less than five lines would end up generating false positives in their analysis [Abdalkareem et al. 2017]. While their code detection is related to the work we are doing on detecting commented-out code, their work relies on the preexisting code tags found in the StackOverflow posts. 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 of our work is applicable to detecting text in code-tags (i.e., it can be used to further validate if they are code or not).

2.3. Comment Quality

One method of analyzing readability and comprehensibility of software is to directly analyze the comments left by the authors of source code. 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. Examples are: amount of programming experience, programmer's domain knowledge on the subject software, and when the native language of the speaker differs from the language comments are written in [Zhou et al. 2019] [Borstler and Paech 2016] [Flisar and Podgorelec 2019]. A method of analyzing comments and source code that takes some of this into account is through utilization of `word2vec` [Zhou et al. 2019]. The vectors produced when using `word2vec` can be grouped and analyzed for similarity. The similarity measure resulting from the approach helps to deal with complex semantic relations and can provide a solution to the problem of native language among others [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 provides further motivation on the need to automatically detect and remove commented-out code.

Another example of comment quality is given by Fluri et al. and is based off of the evolution of comments as source code evolves. First, Fluri et al. detect 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, they are able to visualize the changes to see if comments are being kept cohesive and how they relate to the complexity of the source code. One of the findings of Fluri et al. is that the closer a comment is to the code within the source file, the higher its quality 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. As part of Riehle and Arafat study on the commenting practices of open source projects, they include a deep analysis of the relative density of comments in projects. The results show that the standard deviation of comment coverage in open source projects is over 10.88% [Arafat and Riehle 2009]. The largest finding of their research shows that the larger the project gets, the more comment density falls. For example, 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**.

CHAPTER 3

Background

In this chapter, we provide background information on both decision trees and on srcML. More specifically, we go over decision trees in Section 3.1. and srcML in Section 0

3.1. Decision Trees

In this section, we briefly talk about machine learning and why we choose decision trees. We then discuss decision trees, how they are built, and what their output looks like. In 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 consider: first, each character frequency varies widely between comment lines meaning that any machine learning algorithms that rely on the data being more uniform 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 used for classification is equally as important. Thirdly, decision trees are able to handle blank data very well. This is extremely

important when considering that our data is character frequencies of comment lines. This is because comment lines (especially short comments) won't have very many different characters. An example of the character frequencies for a comment is shown in TABLE 1. As can be seen in the table, the vast majority of characters are 0. As decision trees work well with these three factors, we choose to use them in our approach. More specifically we use the decision tree implementation provided by scikit-learn.

In scikit-learn, their current 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, which is necessary as our frequencies are numerical [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], otherwise it will be biased toward the dominant class. The ability to handle various types of data, non-linear data, and working well for classification 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 1, 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 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 the majority class. White indicates 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 node, since 5.13 is greater than 2.45, we travel down the right side of the tree and arrive at the right child node. With a petal width of 1.46 being less than 1.75, we travel down to the left child of our current node. With the petal length of 5.13 being greater than 4.95, we again travel down to the left child of our current node. With a petal width of 1.46 being less than 1.65, we travel down to the right child from our current node. As we reached a leaf node, we classify the sample as being of the virgincia class.

TABLE 1. THE TABLE SHOWS THE FREQUENCIES OF EACH SYMBOL IN A RANDOM COMMENT LINE. THE MAJORITY OF CHARACTERS ARE BLANK (0).

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			

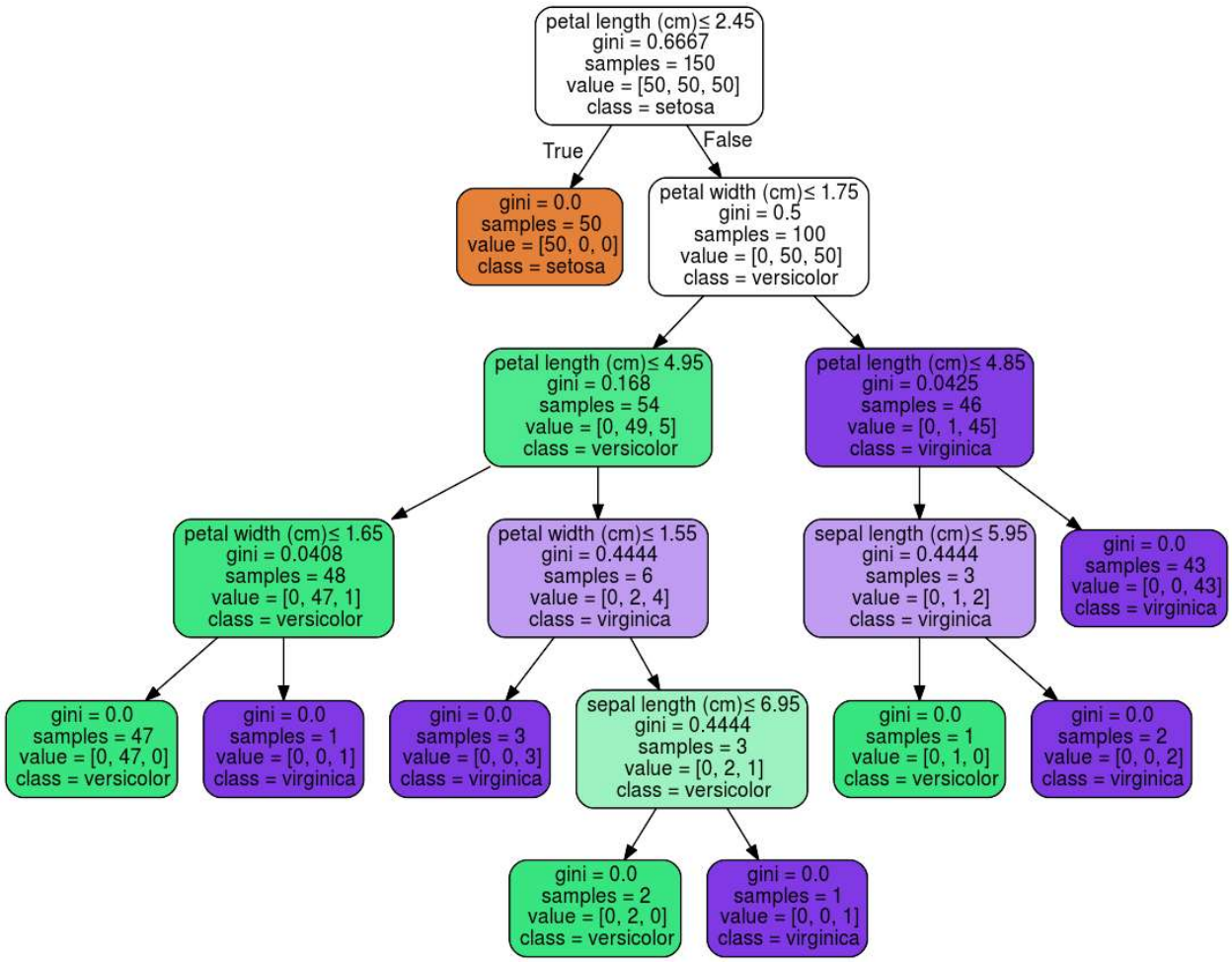

Figure 1. This is a decision tree sample from the IRIS dataset. The colors separate the tree based on which class is the most present in a node. The class shows which of the three class a node primarily consists of. The sample shows how many values are in the node. The gini shows how major of a deciding factor a particular variable is. The top of each node (on non-leafs) is the decision itself.

srcML

srcML [Collard and Maletic] is used in this thesis in order to ensure that all of the comments found in the source code are properly pulled for analysis and to calculate statement lines of code and lines of comments. At its core, srcML is a tool designed to take source code and automatically convert it into an XML representation (also known as srcML). srcML

processes source code without the use of a preprocessor/compiler, and so it does not require such things as 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 leaves the original structure of the source code entirely intact, meaning that whitespace, comments, and all preprocessing statements are left untouched. In TABLE 2, 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 code provided in the top is the main function of the sample C++ file `srcMLsample.cpp`. The bottom contains the resulting srcML from the `srcMLsample.xml`. The first line of the srcML provides the encoding information for XML. The second line contains the root tag `unit` (short for compilation unit). This provides the details on the current version of srcML that was used for the command. Additionally, it contains metadata about the file such as the language the file is written in (i.e., C++) and the file name `srcMLsample.cpp` as provided to the command. Each part of the source code is wrapped in XML tags. First, the outer most tag is the `function` tag, which wraps around and identifies the entire function. Contained within it are additional tags such as type and name tags, which identify those parts of the function. Markup of program elements continues down to the expression level. It is important to note that certain characters cannot be used in XML. For example, the less than symbol in the control block of the for loop is escaped as `<`. Included in the srcML example is a comment. The comment is a sample of a doxygen comment. Attributes exist to denote that this comment is both a line comment (`type="line"`) and a doxygen style comment (`format="doxygen"`).

Another option for running srcML is to run it on a full directory of files rather than on a single file. When we perform this sort of batch operation using srcML, a single XML file is still created. However, rather than being a standard type of srcML file, it generates a unique type of XML file known as a srcML archive. There is one major characteristic of archive files that differ from the standard format. While, a *unit* tag still represents a file within the archive, a new archive *unit* tag is added that encompasses all the file *unit* tags. TABLE 3 is a sample of an XML archive file which has been generated using srcML with the command:

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

The second line provides a sample of an archive *unit* tag. This contains information such as the srcML namespace and the version used to create the archive. For the inner file *unit* tags, a hash attribute has been added. The full path for each file in the directory is provided as well in the *filename* attribute (rather than the file argument provided to the command). Other than these changes, the remainder of this XML archive file maintains the same convention as a standard srcML file.

TABLE 2. 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. THE SAMPLE CONTAINS A COMMENT AS MARKED UP IN 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 3. THIS IS A SAMPLE OF AN XML ARCHIVE FILE GENERATED USING SRCML ON A DIRECTORY. THE PRIMARY DIFFERENCE BETWEEN THIS AND A STANDARD SRCML FILE IS THAT A NEW ARCHIVE UNIT TAG IS USED TO WRAP EACH OF THE FILE UNIT TAGS.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<unit xmlns="http://www.srcML.org/srcML/src" revision="0.9.5">

<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 srcML tool can be used to analyze, transform, and/or extract information. One method for fact extraction (and one we use in this thesis) is XPath. XPath is a query language for selecting nodes from an XML document. XPath queries allow for specific information needed from the original source code to be pulled (i.e., extracted) 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 in the output file (i.e., `Comments.xml`) containing only the comments found. The XML document is a srcML archive. TABLE 4 contains sample output. There are a few notable differences from a standard srcML archive. First, the XPath command results have a `unit` tag for every entry (i.e., comment found) instead of each file. Second, each filename attribute contains the path to the original file the query result is pulled from. Each entry also contains a new unique attribute called item, which maintains a count of each instance the query found in a file and restarts at each new file.

TABLE 4. RESULTS OF APPLYING AN XPATH COMMAND TO A SRCML ARCHIVE. EACH FILE UNIT IS AN ELEMENT THAT MATCHED THE QUERY (E.G., A COMMENT). THE FILENAME ATTRIBUTE FOR EACH MATCHED ELEMENT IS THE FILE FROM WHICH THE ELEMENT IS FOUND. THE ITEM ATTRIBUTE NUMBERS THE ENTRY FOUND IN EACH 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 4

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 thesis, and it defines what we consider commented-out code for the purpose of this thesis. This taxonomy is needed because none of the current taxonomies are specific enough on what commented-out code is to meet all of the needs for our research

In this thesis, we will refer to commented-out code as commented-out code and other comments as English prose. First, in TABLE 5, 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, where 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 and can only be 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 can contain commented-out code and English prose. Most notably, these may also contain code references. The sixth is Javadoc comments which function similarly to Doxygen comments. The main difference being Doxygen is used for many programming languages, and 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 5. PROVIDES A DETAILED DESCRIPTION OF EACH OF THE SIX TYPES OF COMMENTS, LINE, BLOCK, #IF 0, IF(0), DOXYGEN, JAVADOC.

Category	Description
Line Comment	A single comment line. Uses // to denote the start of a comment. Used for both commented-out code and English prose.
Block Comment	A multi-line comment beginning with /* and ending with */. Used for both commented-out code and English prose.
#if 0	Can be used to form a multi-line comment. All text/code starting from the preprocessor directive up until a matching #endif is removed automatically by the preprocessor during compilation. Sometimes used to comment out code.
if(0)	Can be used to form a multi-line comment. The difference between this and #if 0 is this is compiled.. Thus, this can only be used to comment-out code.
Doxygen	A special type of block comment. These comments can have source code references. Used for both commented-out code and English prose.
Javadoc	Same as Doxygen, but specific to Java. Used for both commented-out code and English prose.

In this thesis, though 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 `#if 0` is used for English prose.

At this point we formally define English prose and commented-out code. As comments such as Block Comments can contain multiple lines with individual lines being commented-out code or English prose within the same comment, we define each on a line basis. We define an English prose comment line as *any comment line which is not syntactically correct code for the language that it is present in*. While typically a comment line will be primarily composed of English explanations, you may also see references to variables, mathematical equations, or full algorithms (pseudo-code). These types of comment lines 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 line of source code that has been disabled by one of the methods from TABLE 5.*

TABLE 5 gives examples of the different types of comments and how they are used with English prose and commented-out code. Line comments have a prefix operator of `//` in the C family. This 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, such as 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 a series of line comments can be used to create a pseudo block comment. TABLE 6 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 a math formula. 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. As these have the appearance of commented-out code, these will be difficult to classify correctly. 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 for loop, which is also commented-out code. The final comment is a pseudo 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 7 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 6. 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 REMAINING 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 is a math equation.
`// 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<<" ";`
`//  }`
`//`
`//  Print(res, i-1, j-1, capacity-i);`
	This comment is a sample of a block comment made of single line comments.

```
//      }
//      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 7. 5 EXAMPLES OF BLOCK COMMENTS. THE FIRST THREE CONTAIN STANDARD ENGLISH PROSE, THE REMAINING 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)`
`return;`
`var items = cour.Inventory.Items;`
`Console.WriteLine("----------------------");`
`foreach (var item in items)`
	This comment is a sample of commented-out code.

`{`
`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.

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 8 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 code references, 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. Some examples of why this may be done are: for testing purposes or to lock out certain features that are not yet ready to be implemented. TABLE 9 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. Now that we have gone over the ways to comment-out

code and the definition of commented-out code, we have now answered **RQ 1 (What is commented-out code?).**

TABLE 8. 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 code references.
/// 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 9. THE TOP EXAMPLE IS ONE THAT CONTAINS CODE WHICH HAS BEEN COMMENTED OUT USING THE PREPROCESSOR METHOD #IF. THE BOTTOM EXAMPLE CONTAINS THE SAME COMMENTED-OUT CODE WHICH HAS BEEN COMMENTED OUT USING A STANDARD IF(0) BLOCK.

Comment Examples

`#if 0`
`    for(int i = 0; i < 10; ++i){`
`        a += i;`
`    }`
`#endif`

`if(0){`
`    for(int i = 0; i < 10; ++i){`
`        a += i;`
`    }`
`}`

CHAPTER 5

Data Collection

In this chapter we present our method for data collection and provide a sample of the resulting gold set. In Section 5.1. we discuss the process of selecting the 80 projects we used to collect comment-out code. Section 5.2. explains the methodology used to extract the comments from our corpus using srcML. In Section 5.3. we explain how the manual verification of our two classifications is completed. In section 5.4. we analyze various aspects of the data set. Finally, in sections 5.5. and 5.6. we present external and internal threats to validity, respectively.

5.1. Corpus Selection

To ensure 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 a higher number of stars are likely to be better maintained due to there being greater scrutiny on the projects. Second, these projects are more likely to be under active development/maintenance due to their popularity. Based on this, the 20 topmost starred C, C++, C# and Java projects (80 total) have been selected and pulled for use in building

the data artifact used in this project. Two of these projects are discarded as they contained a significant amount of non-English comments. We choose C, C++, C#, and Java as these are the 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 remaining 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 srcML file, comments were selected at random from all projects. This resulted in 2,935 lines of comments. The reason that we specify lines of comments rather than 2,935 comments is because each line is analyzed independently, even for multi-line comments. The reason for this is because it is very possible to have a mutli-line comment which contains lines of both English prose and lines of commented-out code. These 2,935 comment lines are chosen on a two-point randomization method. First, a selection of 4,660 random source code files are pulled from the main corpus. After this, 1,242 comments were selected at random from the new archive created out of these source code files. The random selection process resulted in comments from 26 of

the original 78 projects. These comments were then classified into either English prose or commented-out code using the definition from CHAPTER 4. This process resulted in 2,935 lines of comments. TABLE 10 is a detailed analysis of the 78 projects from which the 2,935 comments were pulled and it will be described in detail later in this chapter.

5.3. Manual Classification

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 11. The manual verification process was done by a single reviewer and took a total of 185 hours for both the initial classification and a second pass verifying the classification. The whole process was performed over the course of three months. We now discuss what we collected as part of the classification process using TABLE 11 as an example. We collect seven data points for each comment line. The first of these columns contains the comments themselves. In the case of block-style comments, each line is stored independently and each of the seven columns are filled out for each line. 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. All comment characters are kept. This includes '//', '/*', '*', and '///'. We use this instead of a separate column to maintain what type of comment the line is. The purpose of this is information to allow for the determination if certain types of comments are more likely to generate false positives in the machine learning algorithm. These characters are stripped before computing character frequencies. 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.

TABLE 10. 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 IS THE TOTAL NUMBER OF AUTHORS IN EACH PROJECT AS LISTED ON GITHUB. THE FOURTH COLUMN GIVES THE TOTAL NUMBER OF LINES OF CODE OF EACH PROJECT CAPABLE OF BEING PARSED BY SRCML. THE FIFTH COLUMN PROVIDES A TOTAL COUNT OF THE NUMBER OF COMMENTS PRESENT IN THE SOURCE CODE THAT SRCML IS CAPABLE OF PARSING. THE SIXTH COLUMN PROVIDES THE TOTAL NUMBER OF LINES OF COMMENTS PRESENT IN THE GOLD SET FROM EACH PROJECT. THE SEVENTH COLUMN PROVIDES THE TOTAL NUMBER OF LINES OF COMMENTED-OUT CODE PRESENT IN THE GOLD SET FROM EACH PROJECT 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	Ratio of Comments to Code	Percentage of Corpus
rui314/8cc	C	9	9791	712	1	0	0.07	0.0341
Xeoneux/30 Days of Code	C#/Java/C++	40	2408	66	0	0	0.03	0.0000
abseil/Abseil cpp	C++	62	100544	33736	0	0	0.34	0.0000
ethereum/aleth	C++	148	90154	9979	134	0	0.11	4.5656
keon/algorithms	C++	134	9390	2809	1	0	0.30	0.0341
chriskohlhoff/asio	C++	28	105180	38013	47	0	0.36	1.6014
kriscross07/atom gpp compiler	Java	5	343	38	0	0	0.11	0.0000
aws/aws sdk cpp	C++	57	2503490	2509104	217	0	1.00	7.3935
daseyb/BansheeEngine	C#/C++	1	70046	19781	0	0	0.28	0.0000

nativelibs4java/BridJ	Java/C++/C	10	29002	14317	0	0	0.49	0.0000
rswier/c4	C	5	495	12	0	0	0.02	0.0000
mewebstudio/captcha	Java	28	510	26	0	0	0.05	0.0000
aamine/cbc	Java/C	1	9638	769	0	0	0.08	0.0000
eclipse-cdt/cdt	Java/C++/C	154	1122144	450027	85	0	0.40	2.8961
cgeo/cgeo	Java	101	79981	14019	0	0	0.18	0.0000
civetweb/civetweb	C/C++	160	581605	191390	0	0	0.33	0.0000
MyDearGreatTeacher/C	C/C++	1	4684	829	0	0	0.18	0.0000
eranif/codelite	C++/C	64	892859	228394	0	0	0.26	0.0000
TheAlgorithms/C Plus Plus	C++/C	100	5767	576	31	26	0.10	1.0562
Rapptz/cpp sublime snippet	C++	3	502	0	0	0	0.00	0.0000
yusugomori/DeepLearning	C/C++/Java	1	4083	239	0	0	0.06	0.0000
distcc/distcc	C/C++	45	27906	8514	0	0	0.31	0.0000
Dukweeno/Duckuino	Java/C++/C	5	1208	173	0	0	0.14	0.0000
davideicardi/DynamicExpresso	Java/C#	15	15245	4521	9	0	0.30	0.3066
EasyHttp/EasyHttp	C#	12	3518	1382	0	0	0.39	0.0000
JumpAttacker/EnsageSharp	C#	8	36262	3345	13	13	0.09	0.4429
enyim/EnyimMemcached	C#	9	14195	5020	0	0	0.35	0.0000
microsoft/FASTER	C#/C/C++	32	32710	5607	0	0	0.17	0.0000

felixge/faster than c	Java	3	446	18	1	0	0.04	0.0341
NaturalIntelligence/fast xml parser	Java	33	3724	504	0	0	0.14	0.0000
statianzo/Fleck	C#	24	3135	58	0	0	0.02	0.0000
facebook/folly	C++/C	501	286037	84631	1249	78	0.30	42.5554
cs01/gdbgui	Java/C/C++	27	1999	568	0	0	0.28	0.0000
google/glog	C++/C	76	8077	3869	0	0	0.48	0.0000
RyanFehr/HackerRank	Java/C#/C++	9	9230	4487	0	0	0.49	0.0000
redis/hiredis	C/C++	99	6283	1611	0	0	0.26	0.0000
nodejs/http parser	C/C++	77	6349	704	0	0	0.11	0.0000
Redth/HttpTwo	Java/C#	2	9152	2499	0	0	0.27	0.0000
arnetheduck/j2c	Java/C/C++	1	10948	350	27	0	0.03	0.9199
google/j2objc	Breaks CLOC	66	881870	636880	63	0	0.72	2.1465
bytedeco/javacpp	Java/C/C++	33	20291	6279	0	0	0.31	0.0000
Bytedeco/javacpp presets	Java	57	500002	318650	0	0	0.64	0.0000
json c/json c	C/C++	94	6536	2469	0	0	0.38	0.0000
open source parsers/jsoncpp	C++/C	146	8526	1386	0	0	0.16	0.0000
fishercoder1534/LeetCode	Java/C++/C	38	3782	65	0	0	0.02	0.0000

haoel/leetcode	C	28	12636	487	0	0	0.04	0.0000
Tencent/libco	C++/C	12	3099	448	0	0	0.14	0.0000
hyperrealm/libconfig	C/C++	27	43670	2108	0	0	0.05	0.0000
libigl/libigl	C++/C	82	92359	28712	385	94	0.31	13.1175
edenhill/librdkafka	C/C++	163	77178	31555	0	0	0.41	0.0000
ElementsProject/lightning	C/C++	129	104892	22790	0	0	0.22	0.0000
toptensoftware/markdowndeep	Java/C#	6	51833	10893	24	0	0.21	0.8177
neuecc/MessagePack Csharp	C#	45	45564	4585	32	6	0.10	1.0903
Fedjmike/mini c	C	1	3454	6	11	0	0.00	0.3748
ArduPilot/MissionPlanner	C#	71	798373	240724	188	42	0.30	6.4055
mono/mono	C#/C/C++	747	6049345	1295150	11	11	0.21	0.3748
msgpack/msgpack c	C/C++	113	93200	11583	0	0	0.12	0.0000
miloyip/nativejson benchmark	C++/C	32	16083	1337	42	7	0.08	1.4310
nilproject/NiL.JS	Java/C#	8	505660	139548	0	0	0.28	0.0000
brianc/node pg native	Java	10	1234	66	0	0	0.05	0.0000
mapbox/node pre gyp	Java/C++/C	47	2533	271	0	0	0.11	0.0000
TooTallNate/NodObjC	Java/C/C++	11	9062	2495	0	0	0.28	0.0000
vurtun/nuklear	C/C++	100	60370	12260	0	0	0.20	0.0000
oclint/oclint	C++/C	29	21478	478	59	13	0.02	2.0102

miao1007/Openwrt NetKeeper	C/C++	15	1519	616	0	0	0.41	0.0000
Project OSRM/osrm backend	C/C++	109	195834	30768	42	0	0.16	1.4310
nayuki/QR Code generator	C/Java/C++	2	4088	1206	53	0	0.30	1.8058
Iballabio/QuantLib	C++/C	99	376347	72562	163	11	0.19	5.5537
Andersbakken/rtags	C++/C	110	22545	2178	0	0	0.10	0.0000
ServiceStack/ServiceStack.Redis	C#	61	29413	3529	0	0	0.12	0.0000
GavinYellow/SharpSCADA	C#	4	57238	3846	0	0	0.07	0.0000
octo technology/sonar objective c	Java/C/C++	8	1931	895	0	0	0.46	0.0000
nothings/stb	C/C++	158	69880	11952	14	0	0.17	0.4770
stratisproject/stratisBitcoinFullNode	C#	62	208433	38753	0	0	0.19	0.0000
Unity Technologies/UnityCsReference	C#	150	467413	35634	0	0	0.08	0.0000
cesanta/v7	C/C++/Java	17	30268	8982	0	0	0.30	0.0000
zaphoyd/websocketpp	C/C++	41	19577	12500	0	0	0.64	0.0000
xamarin/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 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 srcML 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-style comments 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 represents the language that the source code is 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 columns 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 11. GOLD SET SAMPLE. 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 GOLD SET 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 things 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 for such things as 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 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 we 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 validate by looking at the source code. The following comment is commented-out code as it is a 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 also mark things such as `if`, `else`, `else if`, etc. These words, however, are less likely to be helpful due to the fact that they are common English words. All data collected is stored in a csv.

5.4. Analysis of the Gold Set

TABLE 12 contains a breakdown of the gold set by comment type and 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 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 12 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 prevalence of C/C++ across all projects. This is exemplified by looking at the percentage of coverage each language has across all projects. C/C++ appears in over 59% and 67% of projects respectively, meanwhile C# and Java only appear in 27% and 31% of projects. Figure 2 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 approximately 35% of the comment types and Block comments make up 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. Looking more at individual language, another interesting detail that we found in

TABLE 12. BREAKDOWN OF THE GOLD SET BY COMMENT TYPE AND 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 THAT 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++	2409	168 (17.20%)	506 (51.79%)	304 (31.12%)	978	4.03
Java	239	5 (5.75%)	22 (25.29%)	60 (68.97%)	87	3.34
Total	2935	177 (14.25%)	625 (50.40%)	440 (35.43%)	1242	3.74

TABLE 12 is that Java and C# use regular block comments very rarely, preferring to use Doxygen and Javadoc for block commenting. Equally interesting, based off of the data we gathered, it is less likely to use Doxygen and Javadoc when commenting in C/C++. Additionally, in our data set, C/C++ have a higher average length of block type comments than both Java and C#. 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 most likely contributed to why C/C++ has more comment lines.

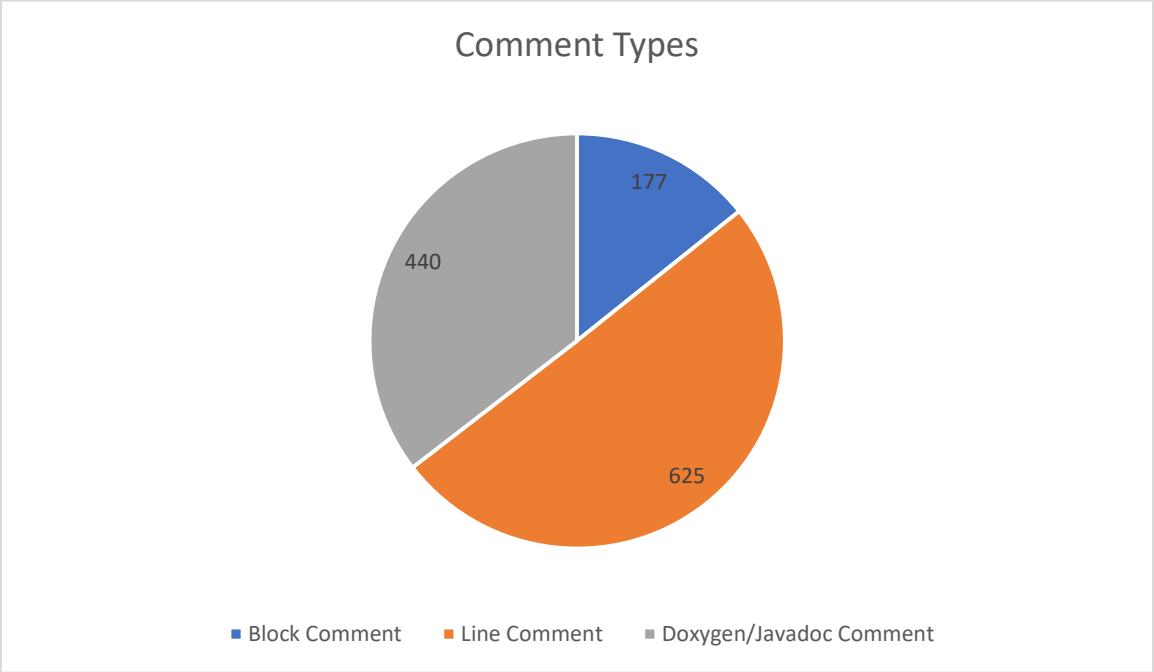

Figure 2. This figure provides a breakdown of the comments in the gold set by type. Line comments make up almost exactly 50% of the comment lines. Doxygen/Javadoc comments make up another approximately 35% of the comment lines with Block comments making up the remainder.

TABLE 13 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 that particular language. These are largely C/C++. The lines of commented-out

TABLE 13. GIVE INFORMATION ON LINES OF COMMENTED-OUT CODE 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.

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++	24089	231 (9.59%)	51 (2.12%)	85 (3.53%)
Java	239	0	4 (1.67%)	0
Total	2935	289 (9.85%)	55 (1.87%)	97 (3.30%)

5

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. In our gold set, C# and C/C++ have a much higher rate of containing standard terms than Java. This is due to the fact that Java has no samples of commented-out code 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 12 and 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 then ran using srcML. The XSLT scripts pulled out code or comments (each on a separate line), 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 on the srcML archive. 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 that project (those that srcML is capable of parsing). The third column gives the total number of authors who committed to the project. The fourth column gives the number of lines of statement code. The fifth column provides a total count of the number of comments present in the source code that srcML is capable of parsing. The sixth column provides the total number of

TABLE 14. BELOW IS A CONCISE FORM OF TABLE 10 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 IS THE TOTAL NUMBER OF AUTHORS IN EACH PROJECT AS LISTED ON GITHUB. THE FOURTH COLUMN GIVES THE TOTAL NUMBER OF LINES OF CODE OF EACH PROJECT CAPABLE OF BEING PARSED BY SRCML. THE FIFTH COLUMN PROVIDES A TOTAL COUNT OF THE NUMBER OF COMMENTS PRESENT IN THE SOURCE CODE THAT SRCML IS CAPABLE OF PARSING. THE SIXTH COLUMN PROVIDES THE TOTAL NUMBER OF LINES OF COMMENTS PRESENT IN THE GOLD SET FROM EACH PROJECT. THE SEVENTH COLUMN PROVIDES THE TOTAL NUMBER OF LINES OF COMMENTED-OUT CODE PRESENT IN THE GOLD SET FROM EACH PROJECT. 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
rui314/8cc	C	9.00	9615.00	712.00	1.00	0.00	0.07	0.03
ethereum/aleth	C++	148.00	89971.00	9979.00	134.00	0.00	0.11	4.57
keon/algorithms	C++	134.00	9243.00	2809.00	1.00	0.00	0.30	0.03
chriskohlhoff/asio	C++	28.00	120472.00	38013.00	47.00	0.00	0.32	1.60
aws/aws sdk cpp	C++	57.00	2498435.00	2509104.00	217.00	0.00	1.00	7.39
eclipse-cdt/cdt	Java/C++/C	154.00	1119730.00	450027.00	85.00	0.00	0.40	2.90
TheAlgorithms/C Plus Plus	C++/C	100.00	5768.00	576.00	31.00	26.00	0.10	1.06
davideicardi/DynamicExpresso	Java/C#	15.00	15245.00	4521.00	9.00	0.00	0.30	0.31

JumpAttacker/EnsageSharp	C#	8.00	36262.00	3345.00	13.00	13.00	0.09	0.44
felixge/faster than c	Java	3.00	446.00	18.00	1.00	0.00	0.04	0.03
facebook/folly	C++/C	501.00	285918.00	84631.00	1249.00	80.00	0.30	42.56
arnetheduck/j2c	Java/C/C++	1.00	10948.00	350.00	27.00	0.00	0.03	0.92
google/j2objc	Breaks CLOC	66.00	857752.00	636880.00	63.00	0.00	0.74	2.15
libigl/libigl	C++/C	82.00	92036.00	28712.00	385.00	94.00	0.31	13.12
toptensoftware/markdowndeep	Java/C#	6.00	51833.00	10893.00	24.00	0.00	0.21	0.82
neuecc/MessagePack Csharp	C#	45.00	45564.00	4585.00	32.00	6.00	0.10	1.09
Fedjmike/mini c	C	1.00	3453.00	6.00	11.00	0.00	0.00	0.37
ArduPilot/MissionPlanner	C#	71.00	798229.00	240724.00	188.00	42.00	0.30	6.41
mono/mono	C#/C/C++	747.00	6049345.00	1295150.00	11.00	11.00	0.21	0.37
miloyip/nativejson benchmark	C++/C	32.00	16006.00	1337.00	42.00	7.00	0.08	1.43
oclint/oclint	C++/C	29.00	21478.00	478.00	59.00	13.00	0.02	2.01
Project OSRM/osrm backend	C/C++	109.00	187682.00	30768.00	42.00	0.00	0.16	1.43
nayuki/QR Code generator	C/Java/C++	2.00	3910.00	1206.00	53.00	0.00	0.31	1.81
Iballabio/QuantLib	C++/C	99.00	376343.00	72562.00	163.00	11.00	0.19	5.55
nothings/stb	C/C++	158.00	69880.00	11952.00	14.00	0.00	0.17	0.48

xamarin/XobotOS	Java/C++/C#/C	1.00	2002448.00	1125133.00	33.00	0.00	0.56	1.12
Total		2606.00	14778012.00	6564471.00	2935.00	303.00		100.00
Average		100.23	568385.08	252479.65	112.88	11.65	0.25	3.85
Median		51.00	60856.50	10436.00	37.50	0.00	0.20	1.28

lines of comments present in the gold set from each project. The seventh column provides the total number of lines of commented-out code present in the gold set from each project. The eight column provides the ratio of comments when compared to lines of code (i.e., comments LOC / statement LOC). The ninth column gives what percentage of the gold set that project contributes. We now discuss details form these tables. First, even though only 26 of the 78 projects made it through random selection, as many as 2,606 authors contributed to the creation of these comments (minimum of ~26). Next, the files selected from folly, which makes up 43% of the gold set, are not extremely large files. 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. Each of these authors could have written or edited comments and thus, 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 outlier, the average contribution of a project to the gold set is 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 widespread between the ratios of comments to code. For example, the lowest is nearly 0 while the highest is over 1 comment line per line of code. 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 lines of comments it will generally have. However, there is a .05 difference between the average ratio of comments to code and the median, with the average ratio being .25 and the median being .20. Furthermore, the correlation between lines of code to the ratio of comment to

code is 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 100 authors (median 51). 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.

5.5. 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 comment lines, it is less than the majority. That is, the majority of comment lines come from separate projects. Additionally, we ran our tests with and without the parts of this project and saw almost no impact on the results. Lastly, in CHAPTER 8, we perform a secondary study containing over 3,000 unseen comment lines and verify that the approach achieves extremely high recall and a precision still almost 80%.

Another threat lies with us only having 2,935 lines coming from a little over 1,200 comments. The reason for this is manual classification is a slow and tedious effort with our process of creating this gold set taking three 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 gold set can be expanded. However, when executing our tests of our decision tree and on the comment lines from CHAPTER 8, 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 CHAPTER 7, 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.

Another threat comes from the project selection process. We only gathered the highest rated projects in GitHub. As mentioned earlier, this was to get projects that are under higher scrutiny and thus, are more likely to be of higher quality. As such, the gold-set may not contain comment-lines from a variety of lower quality projects. This treat is mitigated as the ratio of comment-lines to code varies greatly within our data set which would likely indicate a diversity in quality.

Lastly, our method relies on the way that the comments are written. This means that the more programmers that had the potential to write these lines, the more diverse the comments we will have and the more generalizable the results. 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 significantly less than that as 2606 is the maximum number. A possible minimum number is 54 (29 for folly and 1 for each of the remaining projects). Additionally, the successful results of CHAPTER 7 and CHAPTER 8 seem to indicate reasonable variation.

5.6. Internal Validity

A threat to internal validity is any threat to validity that is caused by something that we did directly. One 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, then 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.

Another threat is that only a single reviewer reviewed the comments. The threat of mistaken classification is mitigated through an additional pass by the reviewer. However, there is still a threat based on reviewer bias. This threat is mitigated as the taxonomy is created to make classification clear.

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. This does, however, make comment counts larger.

CHAPTER 6

Approaches

In this chapter, we present our method for classifying commented-out code, as well as, other approaches that we considered. The investigated approaches are a syntax-based approach (Section 6.1.), a bag of words approach (Section 6.2.), and a frequency-based approach (Section 6.3.).

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. All three checks are done in the following way. First, the line is parsed 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 matching closing brace or closing parenthesis is not found, then the comment will not be considered commented-out code. There are several reasons why this approach is insufficient. First, there are varies times that a line of commented-out code may not contain a semicolon, such as the beginning of an if-statement. The following is an example:

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

Here the beginning of the if-statement is commented out. Although there is a matching open and closing parenthesis, there is no closing curly brace nor is there a semicolon, so the approach fails to detect this as commented out code. The second approach, which was considered, but never implemented is a bag of words approach.

6.2. Bag of Words Approach

The concept of this approach is to break down an entire piece of source code and create a bag of words from it. This could then be used to cross check comment lines 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 such as rarely used variables and variables created in a piece of commented-out code, they are all highly likely to be miss-classified. This is due to the fact that the identifiers may be infrequent or not appear at all. In contrast, other terms in the bag of words like `int`, `void`, or `count` will appear much more frequently. 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 who is well-known for designing alternate versions of the key board layout used on English typewriters and computers. As detailed in [Nakic-Alfirevic and Durek 2004], Dvorak examined which letters are most frequently used in the English language, and relocated their positions on the keyboard 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

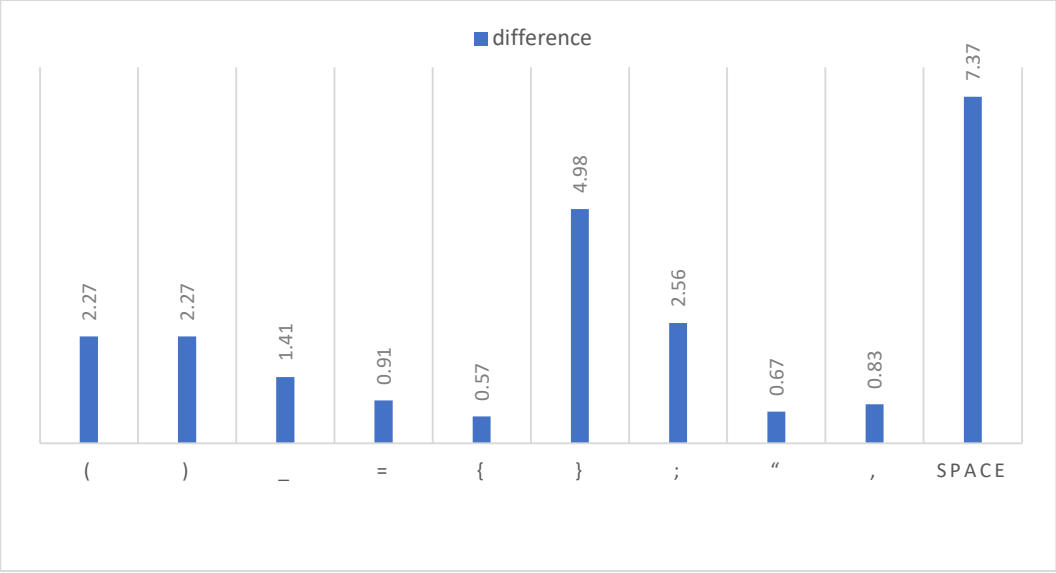

Figure 3. From left to right 10 symbols which have the greatest difference between prose and commented-out code are shown. Parenthesis both open and close, underscore, equals, curly brace open and close, semi colon, quotation, comma, and space. Each value represented in the graph is the difference between commented out code and English Prose, all of which favor commented-out code.

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. Shown in Figure 3 are ten characters from our gold set which have a 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 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 of 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 about the same space of total characters, fourteen in the above example, about 3 words with 2 spaces would fit (total of 16.1). Importantly, what this means is that spaces would be making up about 12% of the total number of characters in the line which is considerably less than in the commented-out code example. This observation continues 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 processing a variety of source code, you are able to create a wide variety of frequency distributions. 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 we get a good representation of what the frequency distributions are for commented-out code and English prose. This helps with generalizability and avoiding overfitting.

We now explain and illustrate via an example how we compute frequencies for a comment line. The way this is done is by taking each character one at a time, converting it to lowercase for normalization, and then storing the current count of that character in a dictionary. Once the entire line has been read and all characters have been counted , the frequency of each character out of the total number of characters for that line is calculated and stored in a list. This is performed by a simple python program that we wrote to automate the process. TABLE 15 below is an example of how the calculation is performed by dividing the count of each symbol by the total count of all symbols. TABLE 1 contains a listing of all the characters we make counts for with UNK being any other character not listed. For classification, these character frequencies are used to train a decision tree model which we discuss in the following chapter

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

TABLE 15. BREAKDOWN OF THE MATHEMATICAL FREQUENCIES OF THE ABOVE COMMENT LINE.

Symbol	Count	Frequency
A	1	.048
Space	4	.190
=	1	.048

S	1	.048
Q	1	.048
R	1	.048
T	1	.048
(	1	.048
B	1	.048
*	4	.190
2	2	.095
+	1	.048
C	1	.048
)	1	.048

6.4. Internal Validity

A threat to internal validity is namely our frequency calculation program. In this case, the program was thoroughly tested and additionally, we performed mathematic verification of the results.

CHAPTER 7

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. All blank line comments are removed. Following this creation process, we use stratified K-fold cross validation to split our data (in a separate Python program). An illustration of this is provided in Figure 4. Stratified K-fold cross validation is the process of randomly splitting your data in a balanced manner based off of the number K, where K is the number of folds. The commonly accepted value for K is 5 to start, however, it is important to ensure that your folds never become too small for the algorithm. With our gold set, 5 is a perfectly acceptable number of folds.

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 an accurate image of the quality of the model. 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 16 contains the calculations used for accuracy, F1 score, precision, and recall which are all commonly used heuristics in both Natural Language Processing and Software Engineering.

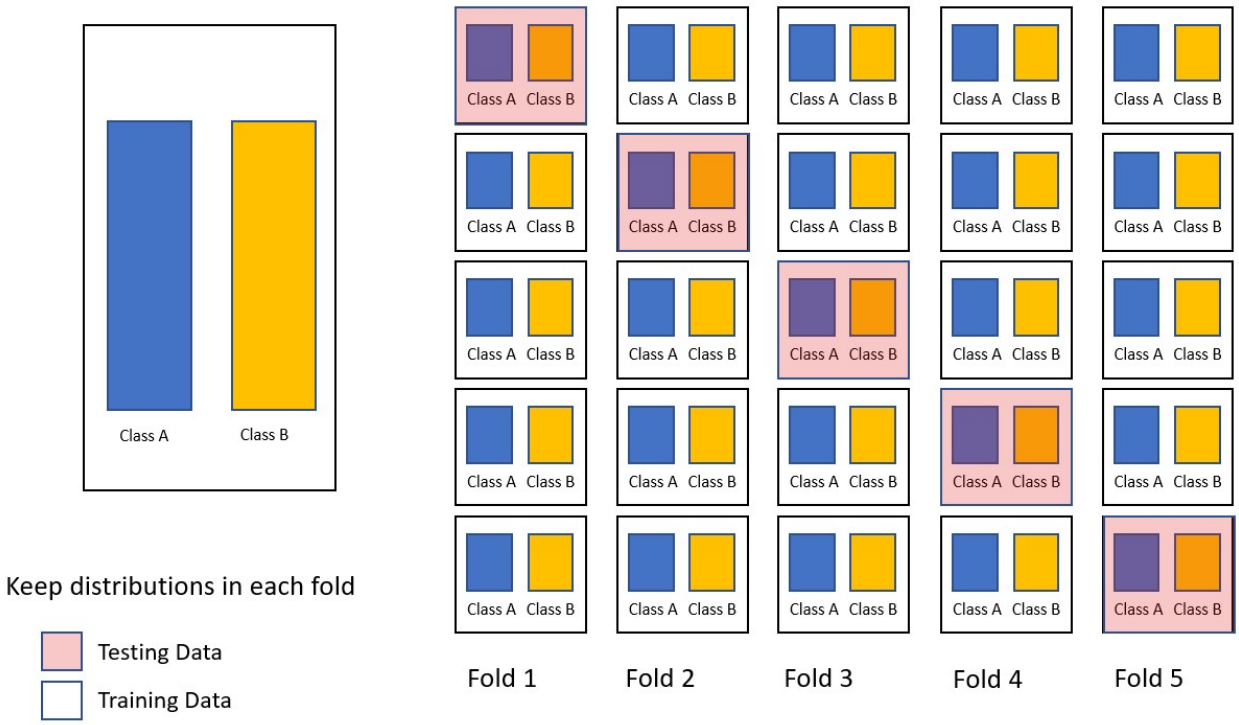

Figure 4. This is a breakdown of how stratified K-fold works. The main data is split into five parts with one of the five being used for testing and the remainder being used for training. With stratified K-fold it is important to ensure that the distribution is always equal amongst all of the groups.

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. Then, we mathematically verified to ensure that the results were holding true. We chose 250, 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 from the stratified K fold cross-validation), we want to have enough unseen examples of commented-out code to detect.

TABLE 16. THIS TABLE SHOWS EACH EQUATION USED AS A TO EVALUATE OUR DECISION TREE MODELS.

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	$\text{F1} = 2 * (\frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}})$

Once again, 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 fold were checked against the initial findings of the research (i.e., experience gained when classifying the golds set and CHAPTER 5) 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 comment-line. This is a highly reliable character as initial research we performed 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 are promising. 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 show. Our best results are represented in fold 1.

Following the completion of this stratified cross validation process, we use the full 250 x 250 samples without folding to generate our final decision tree model. This decision tree is shown in Figure 5. The two colors represent the classes with 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 decision. 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 thesis. Another example is the presence of semicolons, if the semicolon 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.

After creating our final decision tree, we ran a final test against the remainder of the gold set. Our resulting F1 score is 97.89%, the precision is 98.32% and recall is 98.23%. The resulting accuracy is 98%. These results are consistent with the results we obtained from the stratified cross validation. As such, the application of the frequency-based approach and a decision tree works extremely well, and we are now able to answer RQ 3 affirmatively. That is, we can automatically detect commented-out code using the character frequency in combination with decision trees within an acceptable margin of error.

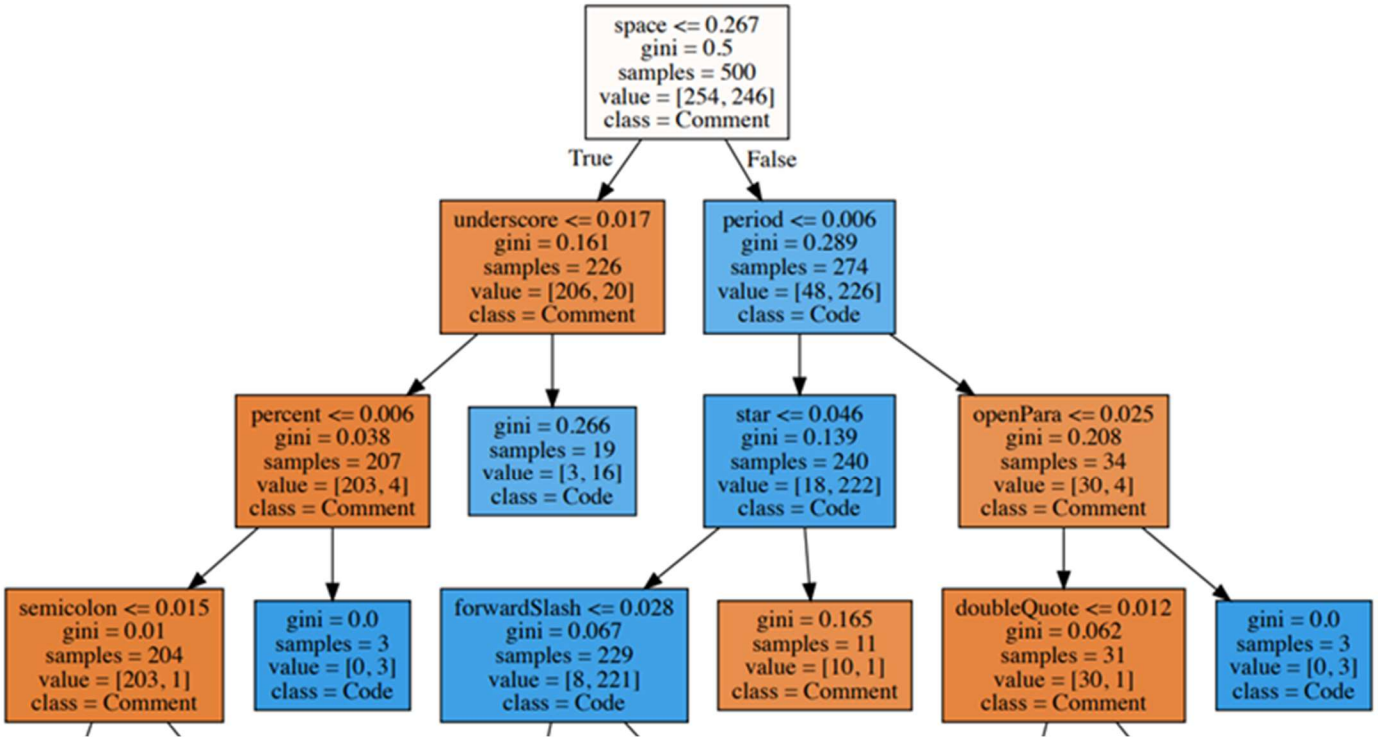

Figure 5. This figure shows our decision tree model. The two colors represent the classes, orange being a normal comment and blue being commented-out code. the samples show the number of samples which are in a node. The gini is the numerical representation of the importance of the gini, the lower the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

CHAPTER 8

Commented-out code in Open-source Software

After verifying that our tool works, the next step is 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, we select 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 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 commented-out code, which we succeed in as clearly shown in our recall. 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

Having verified the model is good enough for a full study of the 50 open source systems, we apply the model to all lines of comments in the 50 systems. 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 approach used earlier. The comment count is attained using the grep command on the srcML archive. 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 allows us to have an overview 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 ratio of comments to statement LOC is lines of comments divided by lines of code. Percentage

of commented-out code is the percent of lines of comments that are commented-out code. The final column is the number of lines of commented-out code in each of the 50 projects. These numbers provide 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 for our gold set. In fact, the ratio of comments to lines of statement code in this study are .24 vs the .23 in the previous 78 projects. This indicates a possible consistency in open-source software as these are entirely different sets of projects and 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 4.16% with a median value of 3.02%. The lowest percentage of commented-out code coverage amongst all of the projects is 0% while the highest is 25.9%. 31 of the projects fall below the average value with 20 of the projects consisting of less than 2% commented-out code. However, only 5 projects contain no commented-out code at all, and these are all small projects. 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. It may be the case that the amount of commented-out

code is project and perhaps author(s) 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 1.51 lines. What this means is that almost all of these projects consist of mostly line comments. However, the average is 2.17 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 90% of projects studied 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 NUMBER OF LINES OF COMMENTED-OUT CODE AMONGST EACH OF THE 50 PROJECTS. THE RATIO OF COMMENTS TO STATEMENT LOC IS CALCULATED BY DIVIDING THE LINES OF COMMENTS BY THE LOC. THE PERCENTAGE OF COMMENTED-OUT CODE IS CALCULATED ON A PER PROJECT BASIS AND REPRESENTS THE NUMBER OF LINES OF COMMENTED-OUT CODE OUT OF TOTAL LINES OF COMMENTS. THE FINAL COLUMN CONTAINS THE AVER NUMBER OF LINES PER COMMENT.

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
Trinea/Android Common	6614	5728	921	160	0.87	2.79	6.22
Blankj/AndroidUtilCode	37120	17191	3674	739	0.46	4.30	4.68
chrisjenx/Calligraphy	1031	569	151	1	0.55	0.18	3.77
CameraKit/camerakit android	7603	3001	1234	58	0.39	1.93	2.43
alibaba/canal	82006	18797	6575	137	0.23	0.73	2.86
apereo/cas	216752	65348	11896	451	0.30	0.69	5.49
ceph/ceph	993305	96032	63741	7682	0.10	8.00	1.51

hdodenhof/CircleImageView	397	22	6	0	0.06	0.00	3.67
cmderdev/cmder	610	28	36	0	0.05	0.00	0.78
cmus/cmus	38252	5292	2413	141	0.14	2.66	2.19
microsoft/CNTK	195054	32780	45492	1081	0.17	3.30	0.72
CodeHubApp/CodeHub	25642	1023	1000	46	0.04	4.50	1.02
collectd/collectd	113747	16648	12183	195	0.15	1.17	1.37
quarnstar/CompleteSharp	802	32	16	0	0.04	0.00	2.00
facebookarchive/conceal	38475	13684	4433	109	0.36	0.80	3.09
yck1509/ConfuserEx	28084	3104	3009	87	0.11	2.80	1.03
dotnet/csharplang	64	0	0	0	0.00	0.00	-
joafarias/csl traffic	9588	671	628	6	0.07	0.89	1.07
careercup/ctci	12659	719	1632	3	0.06	0.42	0.44
foretagsplatsen/Divan	4558	1089	1103	18	0.24	1.65	0.99
davisking/dlib	382186	126239	54776	7751	0.33	6.14	2.30
entt/entt	13191	12200	2167	526	0.92	4.31	5.63
fluent/fluent bit	757518	220988	93546	17953	0.29	8.12	2.36
Ramotion/folding cell android	888	193	80	7	0.22	3.63	2.41

oracle/graal	865667	342664	54025	1465	0.40	0.43	6.34
ferventdesert/Hawk	28092	4595	2678	55	0.16	1.20	1.72
hexchat/hexchat	55789	4591	3865	38	0.08	0.83	1.19
lionsoul2014/ip2region	2037	758	653	9	0.37	1.19	1.16
lzyzsd/JsBridge	2397	391	386	0	0.16	0.00	1.01
juce framerwork/JUCE	421952	117252	55509	3859	0.28	3.29	2.11
ctubio/Krypto trading bot	7353	163	132	5	0.02	3.07	1.23
google/libphonenumber	58447	8989	8451	706	0.15	7.85	1.06
tj/luna	4908	1978	935	57	0.40	2.88	2.12
google/material compnents android	69461	23054	6191	683	0.33	2.96	3.72
MvvmCross/MvvmCross	100281	9058	8698	32	0.09	0.35	1.04
openFrameworks/openFrameworks	162954	34943	33628	9049	0.21	25.90	1.04
OpenRCT2/OpenRCT2	496084	28175	27517	986	0.06	3.50	1.02
php/phppredis	14373	3051	2684	39	0.21	1.28	1.14
cmusphinx/pocketsphinx	22691	7480	3057	738	0.33	9.87	2.45
AeonLucid/POGOLib	12110	716	648	78	0.06	10.89	1.10
riot/RIOT	1754804	1402961	1023087	101873	0.80	7.26	1.37

dotnet/roslyn	2720159	327652	48679	14155	0.12	4.32	6.73
pbatard/rufus	69747	17482	9094	572	0.25	3.27	1.92
chakrit/sider	6572	484	415	87	0.07	17.98	1.17
CoatiSoftware/Sourcetrail	342175	49061	44938	3233	0.14	6.59	1.09
domaindrivendev/ Swashbuckle.AspNetCore	12609	1027	941	71	0.08	6.91	1.09
Kokke/tiny AES c	673	150	130	11	0.22	7.33	1.15
Wind4/vlmcsd	16540	8691	7152	686	0.53	7.89	1.22
wren-lang - wren	42873	7985	5145	542	0.19	6.79	1.55
dmlcx/xgboost	30398	6170	3975	323	0.20	5.24	1.55
Total	10287292	3050899	1663325	176503			
Average	205745.84	61017.98	33266.5	3530.06	0.24	4.16	2.17
Median	28088	5949	3033	123	0.19	3.02	1.51

CHAPTER 9

Future Works

We envision several primary enhancements that we believe need to be handled in the future to extend the power and validity of this research, as well as, applications of the said research.

In CHAPTER 5, it is discussed that the scope of this study is limited, because it has only worked with 26 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. 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 immature or out of date coding styles. A third group of programmers, 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 a potential to cause shifts in the frequencies. Another, similar investigation is into highly specific coding styles.

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 and when. This allows us to ask the programmer exactly why they commented out the code in the first place. Additionally, we gain the ability to track commented-out code as it enters and exits source code. We can develop a method for automatic removal or develop a tool for correcting developer behavior over time.

Lastly, the approach can be used to improve information retrieval and other approaches for feature location, etc. that rely on comments. It is our believe that removing the commented-out code will improve such approaches. As such, we plan to apply the technique to several such techniques and measure the amount of improvement.

CHAPTER 10

Conclusion

The results of our analysis are definitive and show that we can use machine learning in combination with character frequency in order to detect commented-out code. We were able to accomplish this within the original 26 projects with an F1 score of 97.89%, a precision of 98.32% a recall of 98.23%, and an accuracy of 98%. Additional results of this thesis are a gold set which is derived from these original 26 projects and a comment taxonomy. The gold set can be used to support future research, and the comment taxonomy details what is and what is not commented-out code.

In the application of the model to 50 new open-source projects, we saw a reduction in precision (down to 78.03%) due to the detection of lines that had code references but were not fully commented-out code themselves. In contrast, the model achieved recall of 99.62% and a F1 score of 87.51%. This shows that we are able to detect commented-out code with an acceptable margin of error. Furthermore, we applied the model to all comment lines in the 50 open-source systems to determine the prevalence of commented-out code. Results show that most projects contain commented-out code, but the amount varies greatly.

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.

ALHINDAWI, N., DRAGAN, N., COLLARD, M.L., AND MALETIC, J.I. 2013. Improving feature location by enhancing source code with stereotypes. *IEEE International Conference on Software Maintenance* ieee 2013, 300–309.

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. 2010. Extracting Source Code from E-Mails. *2010 IEEE 18th International Conference on Program Comprehension*, IEEE, 24–33.

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.

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

CORBI. 1989. Program understanding: Challenge for the 1990s. 294–306.

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.

FLEXRA. 2017. 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.

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.

LEHMAN, M.M. 1996. Laws of software evolution revisited. In: C. Montangero, ed., *Software Process Technology*. Springer Berlin Heidelberg, Berlin, Heidelberg, 108–124.

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.

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. .

PARNAS, D. 1994. Software Aging. .

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.

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.