# 

Introduction

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

To support the study of comprehension we have to not look at just the pieces of code and comments that lead to improved understanding, in fact just as important as these pieces is understanding how people read and comprehend code. There are two prominent ways people read and comprehend code, which 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, choosing to guess what a called function is doing and waiting until they read the function definition later on. 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]. Comprehension is directly relational to maintenance effort, which is one of the main aspects of software evolution that needs improving.

The most expensive, time consuming, and longest part of the software development life cycle is widely known to be maintenance, and work is always being done to try and simplify this lengthy portion of the life cycle. 90 percent of the total cost of software comes from maintenance, and the amount of time that we must spend in maintenance is directly proportional to the amount of time programmers spend comprehending the code they are reading [Borstler and Paech 2016]. It is this concept of comprehension, alongside maintenance, that is the basis for why more research is needed to support better comprehension. Comments are one means of providing support for comprehension. Comments should always provide cognitive support, assist in comprehension, and reduce the difficulty of maintenance, reinforcing ideas and concepts that are present in the source code [Storey 2005] [Détienne 1990]. The concept of cognitive support can be reinforced in comment structure by including beacons, which are familiar pieces that are pulled directly from the code [Von Mayrhauser and Vans 1995] [Storey 2005].

One problem with the comprehension of software comes from that of commented-out code. Commentd-out code bloats the software and is a distractor from the meaningful parts of the program (source-code and meaningful comments). Additionally, commented-out code can lead to confusion, especially if something that is commented out seems to be unrelated to the section that it is in, or if a piece of commented out code directly contradicts logic present in and around where it has been commented out. Of course, alongside comprehension, when you consider the long-term maintenance of code, the simple question of why a piece of code has been commented out is likely to come up. In this case it is likely unknown as to whether or not the commented out code is a security vulnerability, is a feature that needs to be implemented later, is it reference code that was used to build another section earlier on, or does it cause a total crash if it is run? There is a legal precedent to the problem of commented out code too. Companies are being sued for leaving commented out code that does not belong to them in their source code. Companies are also encountering problems with commented out code being moved back into an active status. The next paragraph summarizes one such case.

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

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

Our goal with this current research is to offer a method for automatically detecting commented out code within a software project, with the hopes that we can improve maintenance time and mitigate confusion later on when commented out code is found. Our thought process is that if we can detect commented out code rapidly throughout the development phase of the software when there is a direct route to question why it has either been added in or commented out at the time of origin rather than trying to decipher the meaning later on. Of course the benefits are not limited merely to maintenance, by detecting commented out code early we have the ability to protect companies from disclosing security vulnerabilities that may be outlined in sections of commented out code or to avoid sections of commented out code that have the potential to be accidently made active.

Research in this area is exceptionally limited and the debate on text detection is an ever changing and evolving field. Our hope is that our research can help drive code detection in a good, steady direction. Our focus is in the automated identification of commented out code by using machine learning. Our chosen machine learning approach is the C4.5 Decision Learning Tree algorithm. One of the reasons that we chose to use the Decision Learning Tree algorithm is that it uses a supervised learning style. This makes it easy to learn how the tree is differentiating between our two classes, and do mathematical and statistical verification on the data. In order to detect commented out code we break down every line into individual characters and store them in a dictionary which is then further broken down into frequencies of each character in comparison to the total number of characters in the line. In order to automate the process, we develop a data artifact that is read and evaluated by hand in order to create an environment that is optimized for training our automation process. This curated data is the culmination 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) and our random selection process ensures that the comments chosen are distributed evenly across these languages. Once this process is completed we can then use stratified K-fold cross validation to split our data into training and testing sections to optimize the learning schema. We then feed our data through multiple times in order to ensure overfitting has not occurred. These processes will be described in more detail in Chapter 8. All of the decisions that were made in regard to the classification of these comments is based off of a taxonomy of comments that we have developed.

In this thesis we answer the following research questions:

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

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

* **RQ 2: What are the different types of comments and commented-out code?**

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

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

Detection of commented out code is vital to solving the problems that they cause, as frequently the people commenting out the code do not see it as an issue or may just forget to delete it. While others have attempted to detect code in general, it is a new effort to detect commented out code in comments. As such, in this **RQ**, we will determine how to detect commented out code.

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

Open-source software is a constantly growing environment in the coding community. However, open-source software is a very different environment, with none of the normal positions found in a large company. As such, in this **RQ**, we will determine how often commented out code makes it into open-source software.

This thesis makes the following contributions:

* A detailed taxonomy of comments and what various types of comments there are.
* A gold set created from manual investigation and verification of nearlhy 3,000 comments (classified as English prose and commented out code)from a corpus of 80 open-source projects.
* Investigation on differences between English prose and commented out code
* The development of an approach to automatically classify comments into the fields of English prose and commented out code with low margin of error.
* A in-depth study of the prevalence of commented-out code in 50 additional open-source projects outside of our original training and testing set.

The remainder of this thesis is laid out as follows. In Chapter 2, we go over related works focused primarily in natural language processing. In Chapter 3, we cover our taxonomy on comments. In Chapter 4, we cover the source code parsing tool srcML. In Chapter 5, we detail the process of data collection. Chapter 6 focuses on the analysis of the data that we have obtained and processed in the data artifact. Chapter 7 provides an in-depth discuss on decision trees, our machine learning method. In Chapter 8, we discuss our methodology and how we obtain our results. Chapter 9 presents and discusses the results of the automation process on test data. Chapter 10 focuses on our threats to validity. Chapter 11 outlines our plans for future work based off of the results from this study. Finally, Chapter 12 is our conclusion, giving a brief overlay on all our results.

# 

Related Work

In this chapter we present the related works and break them down section by section. Despite the growing need to detect commented out code, there has not been much research. As such this we primarily focus the related work discussion on a few related areas. In Section 2.1, we provide related work on identifying code in unstructured text. In Section 2.2, we provide related work related to major works in automatic code generation and studies on nature of comments themselves. In Section 2.3, we focus on various works in vectorization that does not fit in either comment generation or automated summarization. The final Section, 2.4, focuses on research and studies that work on automatization of both detection and summarization of various types of source code and text. Separate to all of these areas is the study of a taxonomy of a comment, which is detailed by Chen et al. The results of the research provided by [Chen et al. 2019] give us four different types of comments which they filtered out of their study as non-prose and low purpose, Code Comments, which we call commented out code, Task Comments which are notes such as TODO or FIXME, IDE comments, which are special comments designed to communicate to the IDE directly, and non-text comments which are links to websites or other comments that are not directly related to the source code. This taxonomy which they have developed is in particularly highly related to our research, however, they choose to ignore these comments and provide little focus on them whereas it is these types of comments that we want to focus on and expand the taxonomy of.

## Detecting Code in Unstructured Text

In Bacchelli et al.[Bacchelli et al. 2010a], the authors developed an approach for automatically detecting code in emails. In the process of developing their method for automated code detection Bacchelli et al. tested a variety of different methods, They test frequency of special characters, occurrence of keywords, end of line symbols, beginning of line symbols, regular expression, and a series of combinations between all of them [Bacchelli et al. 2010a]. The results of these approaches are interesting, what they show is that no individual method was enough to be consistently accurate for detecting code in emails. Furthermore, most of their combinations involve adding in regular expression to increase precision and recall, sometimes by a very significant amount. The final results of testing these methods both with and without regular expression shows an optimal case of 85-95% detection rate by using end of line in combination with regular expression, varying on the language it is trying to detect [Bacchelli et al. 2010a]. There is however a shortcoming to their research, when they detect special characters their focus is too tight and fails to note characters that can be syntactically very important, for example a common method of declaring string variables requires a quotation mark, which is a symbol rarely present in common speech. This is where our research and the work of Bacchelli et al. differs, we cast a much broader net on special characters and utilize decision trees to help decide which characters are the most important.

## Comment Generation and Comment Studies

Related to our work is research on comment generation and comment studies. Comment generation for source code has a large variety of different benefits, from increasing the quality of preexisting comments, adding documentation where there is none at all, and aiding in the understanding of the source code for users outside the original writing base [Song et al. 2019]. Generating comments using machine learning approaches is not without its problems however, because of the complexities of both varying structures of languages and coding styles in addition to different naming conventions often times comments will require some amount of human verification in order to ensure they make sense. Additionally, because there is no one way to form a sentence in a language, auto generating text can sometimes sound unnatural or just be gibberish [Song et al. 2019]. Not only is automatic generation difficult but the actual process of generating the comments is extremely computationally expensive. The reason being is that you first need to process all of the text and analyze it using a variety of different machine learning models such as a VSM or LSI model and then take those results and run them through deep neural networks in order to produce viable output [Song et al. 2019]. The end result of the research of Song et al. is a method of applying machine learning to automatically generate comments with little human intervention. While this method of comment generation may be effective, it does however have its pitfalls and relies on developing an effective method to interpret code, which is something that we hope our research may prove as an aide in.

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

## Vectorization

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

A large part of natural language processing when considering comment and comprehension analysis includes not just the quality of coverage but also external factors that are considered such as native language, subject experience and subject background [Zhou et al. 2019] [Borstler and Paech 2016] [Flisar and Podgorelec 2019]. A method of analyzing comments and source code that takes these methods into account which is very popular in natural language processing is using vector decentralization to normalize semantic cognition through utilization of word2vec, a popular tool for turning a document into a vector of words [Zhou et al. 2019]. These vectors can be grouped and analyzed for similarity that focuses on the raw value of the grouped terms, which provides a solution to some of the problems, such as native language, natural language processing engineers are facing with code today [Zhou et al. 2019]. Similar to this method of vector grouping is the use of word embedding which is becoming more popular in recent studies [Flisar and Podgorelec 2019]. Word embedding focuses on low dimensional real value vectors rather than looking at groupings of words for semantic value [Flisar and Podgorelec 2019]. These varying methods all come to the same conclusion, comments should be meaningful and related to the source code that they are in, which is something that commented out code does not do. This supports the importance of our research in today’s modern coding age.

## Automated Summarization and Text Detection

As an alternative to the generation of comments within source code, the possibility of generating full summaries of source code provides an option that helps to mitigate some of the shortcomings of generating text [Haiduc et al. 2010]. This method of summary has two different routes that it can take, one of which is the idea of extracting valuable information and placing it directly in the summary while the other method focuses on abstraction, which chooses to provide a general overview of the source code. Both methods rely on the same base method however, the first step is to take all of the terms out of the source code and then convert the terms into a corpus sorted by frequency to determine which terms are the most relevant [Haiduc et al. 2010]. These types of summarizations can be evaluated in much the same way as the generated comments, because they are built on natural language premises an LSI or VSM model can be built to compare them at a mathematic level, however the best method is human analysis and questionnaires [Haiduc et al. 2010] [Song et al. 2019]. Ultimately, this means that these types of summarization methods are very similar to comment generation, but they work on a much larger scale.

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

The concept of text detection, particularly in relation to email is something that is becoming more and more common and is becoming needed in today’s society. For example, in the field of detecting opinion spam new research has been published utilizing neural networks to identify spam that is deliberately misleading in its review [Ren and Ji 2019]. There is a major difference between detecting the spam itself and detecting spammers, in our case what we care about is the spam detection. Detecting spam occurs in a series of three main phases, no unlike the method that we are using in our own research. First, they use human beings to read and identify spam that they consider to be malicious or misdirecting. Second, the results are filtered using filtering algorithms, most of which are proprietary. Finally the data is fed through performance evaluation to verify filter quality such as F1 score, Roc and AUC [Ren and Ji 2019]. While this method of detection does not focus on detecting or understanding code, what it does do is provide valuable insight into understanding natural language at a machine learning level.

# 

**Taxonomy of Comments**

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

How we define the structure of a comment is extremely important to our research as well as defining exactly what commented out code is. We define a standard comment as any comment which does not contain syntactically correct code for the language that it is present in. While typically a comment will be primarily composed of English Prose, you may also see mathematical equations or full algorithms.

We define commented out code as any piece of source code that has been disabled by means of commenting the line it is on. We consider anything to be a comment if it is ignored at compile time with two special exemptions. The first of these exemptions are if(0) statements, which so long as you do not change the 0 to a 1 all text within that if block will be ignored by the compiler. However, this method of commenting can be enabled by changing the 0 to a 1 allowing the text inside, commonly commented out code, to be interpreted by the compiler. Similar to the if(0) statement is if(false) which is instead enabled by changing false to true. Aside from these two exceptions all comments will follow one of two different structures, either a line comment or a block comment.

TABLE 1. PROVIDES A DETAILED DECRIPTION OF EACH OF THE FIVE TYPES OF COOMENTS, LINE, BLOCK, IF(0), DOXYGEN, JAVADOC.

|  |  |
| --- | --- |
| Line Comment | A single line within source code that has been commented out by means of a line comment marker such as //. |
| Block Comment | Multiple lines within source code that have been commented out by means of a block comment marker such as /\* ending with \*/. |
| If(0) | Multiple lines of code within source code that have been commented out by means of wrapping the code inside an if block set to 0. This sets the if block to false and does not run any of the code inside. |
| Doxygen | Used to write software reference documentation, these comments can have hyperlinks and other document wide references. |
| Javadoc | Used to write software reference documentation, these comments have hyperlinks and other document wide references. While similar to Doxygen it is limited to java languages. |

Line comments have a prefix operator such as // in the c family that tells the compiler to ignore anything after the operator. This can be placed anywhere on a line, even after code. Typically, line comments are used to make small notes in a specific section of code, either saying what a variable is used for or marking areas that need fixed.

TABLE 2. 5 examples of line comments. The first three contain regular English prose, the second two contain snippets of commented out code.

|  |
| --- |
| //returns the final cost after calculating tax |
| This comment is in reference to a return of a variable and is an example of an inline comment meant to explain what piece of code is doing. |
| //Variable instantiation section |
| This comment references a section of code giving a simple description. |
| //accuracy = (TP + TN)/(TP + TN + FP + FN) |
| This comment references the equation used to build a snippet of code. |
| // m\_depth(0) |
| This comment is a simple function call 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. |

A block comment differs from a line comment in that it uses both a prefix and suffix operator to block off an entire section for writing or for commenting out code. There are a few different methods for accomplishing this, a common method is the /\* prefix and the \*/ suffix within the C family of languages.

TABLE 3. 5 examples of block comments. The first three contain standard English prose, the last two contain commented out code.

|  |
| --- |
| /\*  \* 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)  {  Console.WriteLine($"{item.Name} {item.OldOwner?.Name} | {item.Owner?.Name}");  }  }, 1000);\*/ |
| This comment is a sample of commented out code. |
| /\*  \_messagesService.AddMessageAsync("Loading weather info...")  .ContinueWith(async i =>  {  try  {  await UpdateWeatherData(\_missionPlanner.FlightDataMap.GetCenter());  await \_messagesService.AddMessageAsync("Weather loaded");  }  catch  {  }  });\*/ |
| This comment is a sample of commented out code. |

These are not your only options however, there are many programmers who use Doxygen and Javadoc comments. Doxygen and Javadoc comments function the same way that a standard block comment does but offer a variety of supplemental features such as cross-referencing and source code linking. Doxygen and Javadoc have a wide variety of prefixes and suffixes that are used to demarcate a span of comments such as /// and /\*\*.

TABLE 4. 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.

|  |
| --- |
| /\*\*  \* 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 &#x2264; len &#x2264; 31 and 0 &#x2264; val >2<sup>len</sup>.  \* @param val the value to append  \* @param len the number of low-order bits in the value to take  \* @throws IllegalArgumentException if the value or number of bits is out of range  \* @throws IllegalStateException if appending the data  \* would make bitLength exceed Integer.MAX\_VALUE  \*/ |
| This comment is a sample of a Doxygen/Javadoc comment that includes hyperlinks to both parameters and exceptions. |
| /// Construct an alive Account, with given endowment, for either a normal (non-contract)  /// account or for a contract account in the conception phase, where the code is not yet known. |
| This comment is a sample of Doxygen/Javadoc that is using the /// rather than /\*\* and \*/ to block off a comment. |
| /\*!  Copyright (C) 2002, 2003 Sadruddin Rejeb  Copyright (C) 2004 Ferdinando Ametrano  Copyright (C) 2005, 2006, 2007 StatPro Italia srl  This file is part of QuantLib, a free-software/open-source library  for financial quantitative analysts and developers - http://quantlib.org/  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.  \*/ |
| This comment is a sample of Doxygen/Javadoc that is using the /\*! And \*/ rather than /\*\* and \*/ to block off a comment. |
| /\*\*  \*  \* 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. |

# 

**srcML**

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

Once source code has been converted to XML using srcML, the user is able to write XPath, a query language for selecting nodes from an XML document, queries to pull any specific information needed from the original source code quickly and easily. This allows us to ignore the actual code in the source and just the comments can be extracted. A sample XPath command to extract all functions from a srcML archive is:

srcml --xpath “//src:function” project.xml -o functions.xml

This results in a new xml document containing only the functions found in the project.xml document. Currently, the greatest limitation of srcML is that it can only parse C, C++, C#, and Java though for the purposes of this research this is not an issue.

# 

Data Collection

In this chapter we present our method for data collection and provide a sample of the output. The process of our data collection was vital in order to create a corpus that provided a diverse variety of different coding styles, comment styles, and frequencies of commented out code. in section 5.1. we discuss the process of selecting the 80 projects which make up the data set we use in training our decision tree. Section 5.2. explains the methodology used to extract the comments from our corpus using srcML. Finally, in section 5.3. we explain how the manual verification of our two classifications was completed.

## Corpus Selection

To ensure that the quality of the base source code that is being used in this project we pull highly starred projects from GitHub using the filter preferences on GitHub. The reason for this is two-fold, first, projects that have higher rates of traffic are likely to be better maintained as there is greater scrutiny on the projects, and second, these projects are more likely to be written by programmers with greater experience and better represent the general population of programmers. Based on this, the 20 topmost starred C, C++, C# and Java projects have been selected and pulled for the use in building the data artifact used in this project. srcML currently only supports C, C++, C# and Java, 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.

## Comment Extraction

The first step in the comment extraction process was 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 an initial srcML query to place all of the source code from the 80 projects into one archive file. After this is done an XPath query is used to extract all the comments from the archive. The extracted comments are placed in a new archive by srcML. The query we use is:

*srcml --xpath “//src:comment” project.xml -o comments.xml*

In the case of this research, this is the appropriate step to take as the rest of the source code is not needed. Once all of the comments have been pulled and placed into their own XML file 2,935 lines of comments were selected at random evenly distributed amongst all of the languages. These selected lines were then manually classified into either English prose or commented out code.

## Manual Classification

The entire process of manual verification covered a spread of 2,935 lines of comments from amongst the 80 different projects and covers a mix of all four languages selected for this project. We verify all comments on a line-by-line basis. In the case of block comments, each line was reviewed and classified separately as shown in Table 2. The reason for reviewing even block comments in this manner is that it is very possible to have a block comment that is a mix of both commented out code and standard English prose. The manual verification process took a total of 185 hours both of initial review and second pass verifying the classification. The whole process was performed over the course of two months.

TABLE 6. The first column of the table is the comment pulled directly from the source code. The second column is the file the comment comes from. The third column shows whether or not the comment is part of a block comment, if it is then it shows the lines of the file the block comment is comprised of. The fourth column lists the language the file is written in. The fifth column shows whether or not the line contains code. The sixth column shows whether or not the line is entirely code. Column seven contains any terms which are standard to the language the line is written in.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **comment** | **file** | **block comment** | **language** | **contains code** | **is code** | **contains standard terms** |
| // 0-1 Knapsack problem - Dynamic programming | 0-1 Knapsack.cpp | n | C++ | n | n | 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 |

Each of the 7 columns represent what are the most important notes on each comment though only two of the columns will actually be used for the machine learning process, namely the comment itself and the column that specifies whether or not a line is code while the remainder is present for the purpose of continued research. The first of these columns contains the comments themselves, in the case of block comments, each line is stored independently and each of the 7 columns are filled out for each line, as described in the previous paragraph. In the interest of maintaining the integrity of the data, all of the blank lines within block comments have been kept as well and are stored on their own lines. To maintain comments of all different types the markers for the comments are also maintained in these lines. Some examples of this include ‘//’, ‘/\*’, ‘\*’, ‘///’ and in the case of C++ and C style block comments potentially no marker at all. The purpose of this was to determine if certain types of comments were more likely to generate false positives in the machine learning algorithm and, if this was the case, to ensure that we manipulate the comments by removing these markers before feeding them into the machine learning algorithm. The second through fourth columns are used primarily for bookkeeping purposes but do provide important information especially towards future research.

The second column is the name of the source-code file from which the comment has been pulled from. This file name is extracted from the path information provided by srcML in the XML archive used in the production of this data artifact. The third column of the csv file is labeled block comment, and there are two different ways that this is marked down. 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 that are a block comment that the line is a part of, note here this number applies only to the range of entries and not to the source code itself. The fourth column is labeled as language and represent the coding language that the source code was written in. We decided to add this column for the purpose of both future research and to ensure that anyone viewing the data artifact will know what language the comment was written in regardless of whether or not they are familiar with all of the different file endings attributed to a language. The language column is followed by two different column’s that are related to one another, the first is the contains code column and the second is the code column.

The first of these two columns, the contains code column, is the fifth column of the csv file and was determined to be extremely important when verifying false positives when catching commented out code with the machine learning algorithm. The primary thing that we check for when determining whether or not to mark this comment with a yes are function names and equations. While equations seem to be less common function names may be included in order to aid in the description of what a section of source code does or to mark what functions need to be called within an area of the source code. The sixth column, which is the column labeled is code, is the second column directly important to the machine learning algorithm. This column is very straight forward and is marked with either a y or n depending on whether or not it is determined that a comment line is commented out code. However, it is important to note that this has nothing to do with the actual source code itself, rather, we decided to mark anything that is syntactically valid if uncommented. for example, in the C family any line that appears like the line below would cause a crash if any of the variables had not been declared beforehand.

// totalCost = price + salesTax - discount;

The seventh column, standard terms, is only ever filled when a comment line is commented out code. The primary purpose of this column is to provide a list of terms that could be used as a bag of words when identifying lines of commented out code. for example, in C++ *#include, return, void, int, string, virtual, float, and double* are all fairly common within code and are terms that could be used to identify commented out code. We are also marking things such as if, else, else if etc. though these are less likely to be helpful due to the fact that they are common English words. In Figure 2 a sample section of the csv is shown to help visualize everything that has been discussed in this chapter.

# 

Data Analysis

In this chapter we present our method for data analysis as well as the data analysis efforts that have failed. When attempting to determine whether or not a comment, or a line of a block comment, is a piece of commented out code things become much more complicated then when a trained programmer is simply able to review it. Over the progress of this research we investigated several approaches: a syntax-based approach (Section 4.1), a bag of words approach (Section 4.2), and a frequency-based approach (Section 4.3). The first of these methods which had proven to be fairly ineffective on larger test cases is what we would call the syntax-based approach.

## 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 [Bacchelli et al. 2010a]. The first check, run on every line, is whether or not the line contains a semicolon, which has the direct ability to generate a number of false positives depending on the writing style of the programmer (i.e., if they tend to use semicolons in standard comments). The second and third checks rely both on checking for the opening and closing of parenthesis and curly braces respectively. What we do when we are checking for these syntax markers is parse the line one character at a time checking for semicolons, parenthesis and curly braces. If a semicolon, open and closing parenthesis, or open and closing curly brace is found we mark the line as containing code. This was not something that we had at first expected to be a problem, and in fact it was, as in cases where optional snippets of code had been commented out such as

//if(x > 10){

if(x == 10){

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

## Bag of Words Approach

This bag of words approach is not to be confused with the keyword approach mentioned earlier in the data collection chapter, which proposes the use of common terms as an additional method of verification. Rather, the concept of this approach is to break down an entire piece of source code and create a bag of words from it, which could then be used to cross check comments for terms that are present in the line which are found to be frequent in the bag of words. For example, if we scan an entire script broken down by the spaces and then scan the comments for matching terms then we could potentially identify variables and function names held within the comments. While this could be helpful in finding commented out code that is modifying common variables or using common variables as part of a greater equation, it has a number of strong failing points. First, when considering variable names, one time use variables, variables created in a piece of commented out code, and commented out functions are all highly likely to be ignored due to the fact that in comparison to other terms in the bag of words they may only have an appearance rate of 1-3 times in the entire source code where as a term like int, void, or count will appear much more frequently. This is where frequency comes into play and why bag of words is bound to fail in this case, if a piece of commented out code contains a variable that occurs nowhere else it is not going to be caught. The other issue with this method comes down to explanations of how code functions. In this case, in thorough documentation a programmer may reference function names and variable names with too many of such references it will cause a false positive. This brings us to our third and most current approach, what we call the frequency-based approach.

## Frequency-based Approach

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

*// i = a + b;*

In this example the average size of a term is roughly 1 character and a total of five non-space based characters being present, now when you consider the fact that there is also eight spaces in the line, that means that the spaces are making up over 50% of the lines total number of characters. Further, taking into consideration Mayzner’s work and Googles [Norvig] follow-on research using modern computational methods, it has been determined that the average length of an English word is 4.7 characters. This means that in the same space of total characters, fifteen, on average 3 words would fit, assuming that it ends in a period and contains 2 spaces. Importantly, what this means is that spaces would be making up about 13% of the total number of characters in the line which is roughly 80% less spaces than the commented-out code example. These methods continue to hold true at different frequencies for a wide variety of different characters besides the ones mentioned previously, though in smaller amounts. One of the benefits of using a method like this is by scanning a variety of source code you are able to create frequency distributions that are consistent. In the case of the final frequency distributions used in this research the values are pulled from code and comments from amongst different projects, ensuring that it gets a good general representation of what a frequency distribution should look like and helps with generalizability and avoiding overfitting. Of course, an added benefit to this is if you are examining code and comments that are required to follow a very specific structure then the process is equally as beneficially once the scanning process is complete. The way this is done is by taking each line one at a time and verifying each character converted to lowercase for normalization against a dictionary of characters and then consequently stored in the dictionary. Once the entire line has been read and all characters have been stored and a final count of characters is obtained, the frequency of each character is calculated and stored in a list. This ensures that they remain in order by using key based verification. These frequencies can then be used individually, as a group, or averaged into a single working list as shown in Table 1.

# 

Decision Trees

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

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

In scikit-learn’s current state their decision tree algorithm is based off an optimized version of the Classification and Regression Trees (CART) algorithm. This is a variation of the popular C4.5 algorithm which proceeded ID3 style decision trees [scikit-learn developers]. One of the major changes that came with the C4.5 algorithm is the ability to handle non-categorical data, as well as a new method for pruning that focused on pruning if a rules precondition improved without the pruned node [scikit-learn developers]. Decision trees require the data used to train the tree to be as balanced as possible. This is because at its root, a decision tree is a series of binary decisions and the optimization of such a sorting method requires this sort of distribution [scikit-learn developers]. The idea is that as shown in Figure 3 each time a rule is created it is a simple yes or no question, is a value less than or greater than a certain number being a very common method.

The ability to handle various types of data, non-linear data, and work well for both classification and regression are not the only reasons why we chose decision trees however. Decision trees can be fully visualized as shown in Figure 2[SKLearn 2019], which makes them both very easy to understand and equally easy to explain. The colors, along with the class, in the tree represent each class within data set for ease of comprehension. 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 in that part of the tree. Finally the GINI is the value that shows how effective a decision tree, the farther down a decision tree you go the lower the GINI will generally become with leaf nodes always having a GINI of 0.



FIGURE 2. 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.

This is aided by scikit-learn further using their export method which can allow you to color code and label the tree to aid in interpretation [scikit-learn developers]. To add to this decision trees use a white box method, and all the rules created are made clearly visible; this allows all rules to be statistically verified unlike with other methods such as neural networks [scikit-learn developers].

# 

**Methodology & Experimental Setup**

In this chapter, we present the methodology we used for applying decision trees and obtaining our results, as well as, how we setup the experiment. Once we finish the process of manual verification and have a distinct set of comments divided into our two classes, English Prose and commented out code, and our comments have been converted into frequency data frames then we can move on to our final steps of automating our process. The first step in the process of developing our results is the process of developing our data frame. Our data frame is created by concatenating lists of the frequencies of every line of comments within an XML file. We have a variety of options here, either we can pull this information from individual scripts, whole projects or entire directories of projects should we so choose. For the purpose of developing our decision tree we also include a second array within our data frame containing the exact class of each line of comments from our manual classification. Once this process is complete, we begin the process of creating our decision tree.



FIGURE 3. Starting at the process of downloading all the projects from GitHub. After completing the process of downloading all the test projects files are fed through a srcML command immediately followed by an XPath command. The results from the XPath are then randomized, a portion of the randomized data is additionally stored for manual analysis. All of the data is put through a frequency calculation algorithm before finally being fed into the Decision Tree Algorithm for final analysis.

The first step of creating our decision tree is to split the data that we are going to use for training and testing it. This process is accomplished by means of stratified K-fold cross validation. Stratified K-fold cross validation is the process of randomly splitting your data in a balanced manner based off of the number K, being the number of folds, that you want to make.

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.

The commonly accepted value for K is 5 to start, but it is important to ensure that your folds never become to small for the algorithm that you are working with. This is not an exact science, and many data scientists and other professionals will all say different things, but generally speaking this is where having good evaluation criteria to validate the fact that you are not over or under fitting comes into play.

For the purposes of validation, we will be using accuracy, precision, recall, and the F1 score. The combination of these four scores gives a good image of what our results look like when running the prediction model of the decision tree with our test data. If accuracy and the F1 score are to 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 explain why either the F1 score or accuracy may be low, as both accuracy and F1 score use precision and recall in their calculations as shown in Table 3.

TABLE 7. This table shows each equation used as a hueristic in the analysis of our results.

|  |  |
| --- | --- |
| **Accuracy**  **Equation** |  |
| **Precision**  **Equation** |  |
| **Recall**  **Equation** |  |
| **F1**  **Equation** |  |

# 

Results

The results of this research are attained through the use of a CART decision tree style model present in the Sci-kit Learn module for Python. The reason for choosing to use a decision tree is based off of the benefits that are naturally present when using them. First, decision trees are easy to understand and the model itself can be fully graphically visualized such as in the sample section of our decision tree in Figure 2 Decision Tree Sample.



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.

Second, decision trees are able to handle blank data very well, this is extremely important when considering our data, this is because even if a line does not contain a symbol from the portion of selected ASCII characters that we are analyzing such as in this sample:

Table 8. The table shows the frequencies of each symbol in a sample line. It is important to note how sparse the matrix is as this is the case with many if not all lines.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** |
| *0.0208* | *0.0208* | 0 | 0 | *0.0416* | *0.0208* | 0 | 0 | *0.0208* | 0 |
| **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** |
| *0.0208* | *0.0208* | 0 | 0 | *0.0625* | *0.0416* | 0 | *0.0208* | *0.0416* | *0.0625* |
| **u** | **v** | **w** | **x** | **y** | **z** | **!** | **@** | **#** | **$** |
| *0.0208* | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **%** | **^** | **&** | **\*** | **(** | **)** | **1** | **2** | **3** | **4** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **5** | **6** | **7** | **8** | **9** | **0** | **-** | **\_** | **+** | **=** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **{** | **[** | **}** | **]** | **|** | **\** | **:** | **;** | **‘** | **“** |
| 0 | 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 |  |  |  |

It is still very relevant to our decision-making process. Third, because decision trees are a white box model, we can verify all of the decisions through either Boolean or mathematical approaches. Finally, when considering the immense size of software projects today and the rapid rate at software is growing the fact that the prediction process is logarithmic is extremely important.

In order to ensure that we are receiving high quality and statistically sound results the decision tree was trained using stratified K-fold cross validation utilizing five folds. The reason behind choosing to utilize five folds is because there was a minimal change in using 5x2 but by cutting down to a single 5-fold run we are able to cut the runtime in half. We chose to use K-fold over naïve-bayes style verification because studies show the benefits of using K-fold are very clear and help to ensure that we are not having any issues with overfitting.[Kohavi]

Initial tests were then preformed on smaller samples of 250 lines of true comments and 250 lines of commented out code to prevent a training bias based on a dominate class and then mathematically verified to ensure that the results were holding true. Afterwards the questions generated by the tree were checked against the initial findings of the research 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 a comment. This is a highly reliable question as initial research indicated that on average comments are constructed of approximately 24% spaces while commented out code is constructed of approximately 33% spaces.

Once the integrity of both the initial results and the decision tree model have been verified it is time to move on to larger data set to evaluate the overall quality of this identification approach, and the results are very promising. Over a series of 10 tests randomly selecting 1000 lines of comments from the 20 different projects we show an accuracy of 96.5%, a precision of 97.6%, a recall of 94.3% and a F1 score of 96.6%. All of these results were calculated automatically using metrics from Sci-kit Learn, the calculations for each metric are shown in TABLE 7. Below is a breakdown sample of all 5 folds built into TABLE 9.

TABLE 9. 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 |

# 

Threats to Validity

Throughout the duration of our research there have been a few different points that point to threats to validity that we hope to both mitigate and eliminate in our future research. In 6.1. External Validity we will present our concerns about the validity brought by our choices in projects that our data artifact is based on. In 6.2. Internal Validity we consider the second threat to validity that we face, which has to do with the way that we deal with unique or less common coding styles.

## External Validity

The problem here is twofold, our sample size is limited, and the quality of writing is very high. The first issue is relatively simple to resolve and mainly only requires that we increase our sample sizes, though this comes with a number of computational challenges it is overall something that we can fix. The second part of this issue however is more complex, due to the fact that the code and comments are written in a very clean and consistent manner it has the direct potential to skew our results. An example of this is when programmers use very poor or no spacing methods, when this happens it throws off the root of our decision tree which first checks whether or not there is a high frequency of spaces on the given line. This is the same issue that we encounter if programmers don’t use proper indentation as it throws off spacing counts again, which are as previously stated, the root of our tree. The majority of the code that we are working with for this project also tend to have concise and standard naming conventions. Now of course traditionally this allows for increased readability of code, but not everyone uses this coding practice, if a programmer prefers to use long variable names that fully describe what a variable is, this will cause the ratio of alphabetic variables to increase, which in rare cases can lead to a piece of commented out code failing to be identified.

## Internal Validity

A second threat to validity that we intend to fix with our future research comes from well-established styles of coding, two prominent examples of this is Hungarian Notation and Whitesmiths style. Both of these styles cause new symbols to have increased prominence in our research though the cause is different between the two of them. Whitesmith style causes distribution changes because of its necessity to have minimalistic and concise lines. An example of this is that in Whitesmith style all curly braces should be on their own lines, this means that any line of code containing a curly brace is made of only a curly brace and an end-line marker.

|  |
| --- |
| *If(total > 0)*  *{*  *salesTax = total \* taxRate;*  *}* |

Figure 6 WHITESMITH SAMPLE

Of course, this causes a massive increase in the frequency distribution of curly braces and end-lines even though these symbols do not generally have a lot of importance when reviewing individual lines to determine if it is commented out code or not. Hungarian Notation provides a unique addition to this issue that has been the topic of many thoughts on the future development of this research. Hungarian Notation uses a standard method of labeling variables so that by looking at only a variable name you can gain a basic understanding of both what it is and what purpose it may have. An example of this is the use of the lowercase letter *m* to indicate that a variable is a class member.

|  |
| --- |
| *mDiceRoll;* |

Figure 7 HUNGARIAN SAMPLE

Hungarian notation provides a unique problem, namely, variables are very repetitious and depending on what the script does different letters become extremely important in identifying commented out code, but only when it is written in this style. Because of this there is some debate on whether Hungarian Notation should either be excluded from the tool as an outlier or if a special tree should be used to handle cases like Hungarian Notation.

# 

Future Works

We envision four primary enhancements that we believe need to be handled in the future to extend the power and validity of this research. The first subject that we would like to see extended upon is handling specific coding styles such as Hungarian Notation, vertical alignment, Whitesmiths style, and indentation styles. The second focus of our future research is to extend the number of languages that our method works with. The third focus of our future research is to handle the various levels of coding skill as well as bad coding practices that are in use today. The final focus of our future research is to be able to search merge history within version control to identify exactly when and by whom code has been commented out.

When dealing with unique and specific coding styles such as Hungarian notation, vertical alignment, Whitesmiths style, and various indentation styles there are many different problems that must be considered. As discussed in further depth in the threats to validity section, Hungarian notation and Whitesmiths style cause different values that would not normally indicate commented out code to indicate commented out code. In these cases, it needs to be decided if unique trees need to be created for these problems or if there is a way to incorporate these styles into the current tree without causing trouble for the more common coding styles. Vertical alignment and various indentation styles that involve large amounts of spacing also provide a unique problem as the incorporation of excessive spaces can skew our values, again this is a problem that will either warrant the creation of unique trees for lines with excessive spacing or a way to modify the values without creating bias when integrating them into the tree. There is some argument that if they are not included in the tree creation however, that they will likely be properly identified in either case, though this is something that will require further analysis.

In the Chapter 6, it is discussed that the scope of this study is limited, because it has only worked with twenty projects from GitHub. This choice was made with the idea in mind that we wanted to have a very well written sample of code to work with for the first iteration of this project and it did give us access to almost 100,000 lines of comments. However, the code in these projects tend to be very well written and fairly uniform, and while this does give us a good example of what code and comments should look like it does not account for junior and veteran programmers who use out of date coding styles. Of course, a third group of coders, those who are self-taught, and who lack common and good practices and standards within our field also provide an additional layer of content that we wish to explore. When looking at these groups of programmers and their coding styles they have to potential to cause shifts in the data similar to the highly specific coding styles discussed earlier in this chapter. However, the difficulty here is that unlike with those coding styles which have established rules within their designs, the coding styles that we are talking about here are much harder to identify and will require a lot of research to automate their identification.

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

# 

Conclusion

We are able to definitively answer our first research question, what makes a comment commented out code. We find that there are ten specific symbols that point to a comment containing code, the most likely of which to give the direct effect is a space. Our second question is whether or not we can identify a common taxonomy amongst the different types of comments. We found a number of commonalities among each of the types of comments which is laid out in more detail in the taxonomy chapter. Our third research question is whether or not we are able to automatically detect commented out code.

Our finding is that the analysis of the individual characters rather than the analysis of whole words is something that, as our research shows, produces tangible data that can be used with powerful machine learning methods at a low, order log(n), to produce effective results. These results as shown in TABLE 7 demonstrate that we meet our 5% confidence level in all heuristics as we had hoped to achieve. The importance here is that the overall computational cost of dealing with these numbers within data frames is much lower than say a bag of words approach which requires the storage of and analysis of a constantly varying list of words that must also account for lemmatization and stemming if it is going to work properly. With our approach stemming and lemmatization has a very limited effect, as in the end we are looking at frequency of the occurrence of certain characters, chosen by an optimized machine learning algorithm.

Our final research question is on the frequency of commented out code in open source software. (TBC)

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.

Bacchelli, A., D’Ambros, M., and Lanza, M. 2010a. Extracting Source Code from E-Mails. *2010 IEEE 18th International Conference on Program Comprehension*, IEEE, 24–33.

Bacchelli, A., Lanza, M., and Robbes, R. 2010b. Linking e-mails and source code artifacts. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE ’10*, ACM Press, 375.

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

Chen, H., Huang, Y., Liu, Z., Chen, X., Zhou, F., and Luo, X. 2019. Automatically detecting the scopes of source code comments. *Journal of Systems and Software* *153*, 45–63.

Collard, M.L. and Maletic, J.I. srcML. *srcML*.

Cortes-Coy, L.F., Linares-Vasquez, M., Aponte, J., and Poshyvanyk, D. 2014. On Automatically Generating Commit Messages via Summarization of Source Code Changes. *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, IEEE, 275–284.

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

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

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.

Haiduc, S., Aponte, J., Moreno, L., and Marcus, A. 2010. On the Use of Automated Text Summarization Techniques for Summarizing Source Code. *2010 17th Working Conference on Reverse Engineering*, IEEE, 35–44.

Kohavi, R. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. *IJACAI ’95*, 7.

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.

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.

Ren, Y. and Ji, D. 2019. Learning to Detect Deceptive Opinion Spam: A Survey. *IEEE Access* *7*, 42934–42945.

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

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

Song, X., Sun, H., Wang, X., and Yan, J. 2019. A Survey of Automatic Generation of Source Code Comments: Algorithms and Techniques. *IEEE Access* *7*, 111411–111428.

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.

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.