**Chapter 5**

**Data Analysis**

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 multiple approaches are considered with the final method being the one that is currently in use at this time. 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.

In the syntax-based approach the method for analysis of lines is very simple and is broken down into a series of different checks. 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 in their standard comments. The second and third checks relied both on checking for the opening and closing of parenthesis and curly braces respectively. 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, 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 was a bag of words approach.

This bag of words approach is not to be confused with the bag of words 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. 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. The other issue with this method comes down to explanations of how code functions, in the case of thorough documentation where a programmer may be referencing function names and variable names, to many of such references is likely to cause false positives. This brings us to our third and most current approach, what we call the frequency approach.

The original basis of the frequency approach owes itself to a discussion on the works of Dvorak and Blickensderfer, both of whom are famous for designing alternate versions of the key board layout used on English typewriters and computers today. To simplify the concepts explained in the related works chapter, Dvorak and Blickensderfer both 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 Fig ## large frequency differences, 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:

Fig ## large frequency differences

*// 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 in 1965 and Googles 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 scripts you are able to create frequency distributions that are consistent across the board. 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, ensuring 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.