For my final project, I would like to build upon the research survey I conducted for the midterm into automated bug detection using Machine Learning. For the midterm, my work assessed the current status quo in the field of bug detection by examining existing research into individual and aggregator ML methods each adapted for different contexts, bugs and languages. In summarizing the aforementioned research, I was able to gain an understanding into the three distinct spheres in which the recent advancements made in bug detection largely fall under: data acquisition, code representation and learning modifications. My final midterm paper presented these advancements and the results thereof as well as the results of a user survey on bug detection and programming preferences among software developers. I believe that work established a comprehensive base on which I can now attempt to build a new bug detection tool with improvements in the right directions - stimulated by both research and user perspectives.

The bug detection tool/system that I would like to build for the final project will thus have the following attributes:

- The system will be written in Python to primarily analyze Python codebases. Python is currently the fastest growing programming language which points not only to vast amounts of training data (open source codebases on the Internet) but also to a large potential impact for the tool. Currently, static analysis tools that analyze Python code for structural, syntactical or other types of bugs and errors are based primarily on rulesets. While this ensures the robustness of such tools, it also limits what these tools can be used to detect. Machine Learning algorithms that can be extended to infer non-explicit rules about non-obvious features of Python code would go a long way in extending current CI/CD pipelines. In addition, the existing research in this field is across multiple languages - making it not only difficult to compare and combine their work but also difficult for future work to build on. This system being in Python will make any future improvements easier to implement than if it was in C, Java, etc.
- The trained system will take as an input a path to a directory of Python files and for each file, return the line numbers of any potential bugs.
- The system will use the 'ast', 'inspect', 'sample' and 'pprint' modules native to Python to extract the code data from provided Python files.
- The system will use modules such as 'flake8', 'pyflakes', 'pylint', etc. to extract metadata about the code. AST metadata such as node density per line and line number will also be included. The system will attempt to also use the 'pylint' module's dependency graphs as metadata.
- The system will combine the aforementioned metadata and AST representations (including the augmentations suggested in the research papers) for each line of code in two ways: a) as a single representation to be fed into a single ML method for classification and b) as multiple representations each passed into different ML methods later to be aggregated for classification. Python offers a variety of data structures to support these conversions/ representations. The single classifier and multi-classifier approaches will both be tested with their associated implementations and results presented to the reader
- Once each line has its equivalent representation, the system will strip any magic numbers or other constants irrelevant to the static analyses (if not already done) and convert the remaining information into an n-dimensional array per line using the 'bin' library. This n-dimensional array will be the final input into the classifier(s) similar to passing a 3D array of pixel values to an image classifier(s).
- The system will use a combination of attention layers, convolutional layers and dense layers among other learning modifications (suggested in the research papers) in both the single and multi-classifier approaches to determine a) the key distinguishing features (eg: edges in an image) between the representations of each line of code and b) the relative importance of these features when attempting to classify the line as buggy or not.
- The system will also examine various methods of negative data generation in the training stage to ensure that the buggy data used in training is not only representative of buggy

data in real-life but also that the generated bugs are of a wide variety of different types. The research papers summarized for the midterm suggested several ways in which positive data could be used as negative data as well by manually injecting bugs such as variable swapping, operator swapping, typos, etc. While the generation of these types of artificial bugs has been shown to be successful when training bug detectors, they require manual injection (via creation of rules) of each type of bug that the bug detector need catch. An alternative approach will also be investigated in this system:

1. Magic numbers and other constants will first be stripped from a ‘correct’ line of code.
2. The line of code will then be converted into one of a variety of representations such as binary, ascii, etc.
3. At random, one or more characters in the above representation will be displaced slightly from its true value such as changing a 0 to 1 in binary or ‘a’ to ‘A’ in ascii or ‘w’ to ‘q’ in qwerty to mimic false inputs from a user.
4. The line of code will then be converted back into its original form (with the displacements present) and the constants, etc. re-added. The original line will be checked against the newly generated false line to ensure that a bug has indeed been created but that the similarity of the two lines is high enough to simulate an actual real-world bug and not improbable ones.
5. The original line of code will be passed into training with the class ‘non-buggy’ while the new line of code will be passed into training with the class ‘buggy’.

Now having multiple methods including a proposed technique to generate bugs with a higher degree of randomness, the system will examine whether training a single classifier on all the types of bugs yields better results than training individual classifiers each with their own specific type of bug in the negative data. In addition, the proposed technique of bug generation will also be examined and compared to the bug generation techniques of prior research to determine whether the proposed method yields more flexibility in terms of type of bug generated/detected while still maintaining prior levels of accuracy or better.

- Given that the system will possess its own line similarity measure, lines of code that appear to be similar/have similar nodes to multiple other lines in the codebase can be determined. The continuous usage of such lines/nodes/function calls indicates that these dominate other lines in the code and hence must be assigned higher priority when checking for bugs. The system will attempt to inject more training examples of such lines to assign high priority to any potential bugs in such lines owing to their repeated occurrence. Other data augmentation techniques on such lines might also be attempted if research indicates any techniques that can be implemented within a reasonable timespan.