CS3110 Testing Plan

Camel Capital

May 2024

Note: In order to run “dune test”, make sure that you are in the “CC” directory.

1 What Was Tested

Manual Testing:

Development of the GUI was a very testing-intensive endeavor. The library (Bogue) which we used is limited and prone to buggy behavior at times. During development, once a widget was implemented, visual inspections immediately followed. Much tweaking was necessary to get the GUI to work properly. Development continued like this until it was in a visually complete state, at which point its functionality was tested. Edge cases were tried in the text-input boxes and buttons were spammed- bugs which occurred in this manner were fixed if possible. One bug which occurred and could not be fixed was fullscreening- the GUI will crash and kill the program if it is put into fullscreen. We believe this is an issue with the GUI library, but we decided that it was worth this small loss of functionality to keep what we had already built.

The main application (the programs in main.exe and realtime.exe) were both tested manually through hours of use-case scenarios. We ran the program off of CSV files, data downloaded straight from Yahoo Finance. Many bugs were found in this way, and in resolving those bugs sometimes we would find more. The application has gotten a lot of use over the course of its development, so we are confident that most glaring issues have been ironed out.

OUnit Testing:

Two OUnit suites were developed for this project. Our basic data structures, StockData and Garch, can be tested thoroughly by running the dune testing application. These suites were developed using both black-box and glass-box testing, and every test case was written with the goal of breaking the function it was testing. This technique allowed us to iron out many bugs which were not readily apparent from manual testing, assuring the

stability of our application in most scenarios. These suites are discussed more in the next section.

2 What Modules Were Tested

Garch:

Because the GARCH model is a highly numerical statistical model, the modules functions were very conducive to a black box OUnit testing suite. To ensure a holistic and thorough testing suite, this was done by creating different sets of stock data to cover various edge cases for different functions in the module. Some of the edge cases we considered were negative changes in price, zero price change, or for some functions operating with a single stock data point. We used these various stock data points for the more complex functions in the module, such as the GARCH objective function or the parameter estimation function. For more foundational functions such as the log return, testing was as simple as comparing actual versus expected values. As mentioned above, while the general approach would be best characterized as black box testing, because of the nature of the module and the more complex functions, many of our tests were more concerned with demonstrating that the behavior of the functions was as expected without so much concern for the precise values. A good example of this is our testing suite for the final estimation of variance. We know for instance that variance should be higher between two stock when the changes in price are more dramatic relative to other times, but what that precise value is isn't as important as knowing that our code replicates what we expect. To capture this idea, we employed a testing strategy that gave very conservative estimates of what a 'high variance' stock and 'low variance' stock would be and made sure that our mock data replicated high and low variances in those different scenarios. Once all of the individual tests were written, they were pooled together and ran under a set of relevant suites. This approach further contributed to the modular nature of our program.

StockData:

The StockData structure is reliant on the CSV structure of data fetched from Yahoo finance. The main type in this structure is a record, developed to make dealing with large sets of stock data easier and quicker. Due to this, tests were developed around a set of StockData records that represent most of the results we might get from YFinance. For each function, we developed a suite of tests that checks for proper functionality in both regular cases and edge cases. For example, for Buy Price calculations, we tested the function with a zero true range, a high buy price multiplier, a negative true range, and with varying close prices. This kind of coverage is provided for each function, covering the expected range of inputs for these functions. Most of these tests were written black box, based on the specifications. However, we also wrote several glass-box tests. Functions like shift_in were

written with the source code open right beside the testing suite. This was done to pick out edge cases that we thought had a high probability of breaking the function. This strategy proved useful in refining both the specification and functionality of these functions. In doing this, we found many faults that might have lied dormant until we got one unlucky YFinance result. This

3 Why The System Is Correct

Our testing suite provides coverage for all functions in the library modules. These functions are ones which have been factored out of the main code because they supported a specific data structure which could be made independent. Our entire program is built on these two data structures- StockData and Garch params. We tested these structures with edge cases and regular use-cases. In our testing, these functions perform how they are expected to and should not cause issues under a wide range of use conditions. By thoroughly testing these structures, we have ensured the correctness of our application, up until the point of the trading algorithm.

This trading algorithm was tested manually, and has been observed to produce profits most of the time on most tickers. We have also ran the algorithm for hours in total on different tickers and CSV inputs, and we have never observed a crash or an error thrown by the algorithm. This gives us confidence that our entire program is stable and correct.