AdvisorBot-Report

A brief documentation with regards to AdvisorBot

Name: Lee Chang Loong Benjamin

AdvisorBot is a sophisticated tool developed to help cryptocurrency investors analyze exchange data. AdvisorBot, with a variety of instructions at its disposal, can give important insights and predictions to assist investors in making educated decisions. One of AdvisorBot's essential features is the ability to present all accessible commands, which allows users to become acquainted with its capabilities quickly. Users may also request particular assistance for a given command, which will provide further information on how to use it efficiently. AdvisorBot may assist consumers in determining the minimum and maximum bid or ask for a specific product at the current time step, in addition to giving information on available items. AdvisorBot forecasts the maximum or minimum ask or bid for a product in the following time step for individuals wishing to make future predictions. This might be a valuable tool for investors who want to remain on top of market developments.AdvisorBot also records the current time in the dataset, allowing users to stay current on market circumstances. It also allows users to go to the next time step, delivering continuous insights as the market changes.

Overall, AdvisorBot is a helpful tool for any cryptocurrency investor seeking to make educated judgments in the fast-paced world of digital currencies.

|  |  |
| --- | --- |
| Applied AdvisorBot Command List | |
| C1:help | Status: Implemented and working |
| C2:help[cmd] | Status: Implemented and working |
| C3:prod | Status: Implemented and working |
| C4:min | Status: Implemented and working |
| C5:max | Status: Implemented and working |
| C6:avg | Status: Implemented and working |
| C7:predict | Status: Implemented and working |
| C8:time | Status: Implemented and working |
| C9:step | Status: Implemented and working |
| C10:current | Additional feature[Implemented however error thrown ] |
| C11:exit | Additional feature[Implemented and working ] |

**Command Explanation/Command Parsing Explanation**

C1[**help**]: shows a list of available commands in AdvisorBot

C2[**help[cmd**]]: throws help for a specific command in AdvisorBot, for example, "help prod". AdvisorBot will then tell the user the correct syntax for the specific command, along with an example of how to use it.

C3[**prod**]: displays a list of all the available currencies for trade from the CSV file.

C4[**min**]: finds a minimum bid/ask of a specific currency in the current timestep.

C5[**max**]: finds a maximum bid/ask of a specific currency in the current timestep.

C6[**avg**]: finds the average bid/ask of the specific currency over the entered number of timesteps

C7[**predict**]: predicts the min/max bid/ask of the selected product for the next timestep

C8[**time**]: prints the current time in the CSV file.

C9[**step**]: move to the next time step when the user triggers this command.

C10[**current**]: lists the currency with the most bids/asks in the timestep and its price.

C11[**exit**]: exit command which closes the AdvisorBot program.

My code has a method named 'tokenise' that takes in a string and a delimiter string as arguments and returns a vector of substrings of the input string, which are delimited by the delimiter string. For example, if the input string is "hello 2023" and the delimiter string is" ", the method will return a vector containing two strings: "hello" and "2023". Firstly, **the "promptUserInput"** method reads a line of input from the user and passes it to the 'tokenise' method, along with a delimiter string of" ", to parse the input into a vector of strings.

Secondly, **"validateUserInput"**checks if the user has inputted a valid command. This method takes in a string (the user input command) and a vector of strings (a list of known commands) and returns a boolean value indicating whether the input string is in the list of known commands. It does this by using the find method of the vector class to search for the input string in the list of known commands. If the input string is found in the list, the method returns true; otherwise, it returns false. The code also checks the number of elements in the input vector to ensure that the command is being used with the correct number of arguments (if any). For example, the "min" command requires three arguments: the command itself, a currency type, and an order book type. If the input vector has more or fewer than three elements, the command is not recognized, and an error message is printed.

Thirdly, **the "processUserInput"** method then processes the input vector by checking the first element (which is the command) against a list of known commands and executing the corresponding action if it is a known command. An error message is printed if the first element is not a known command. If the element is empty or consists only of white space characters, it will prompt the user to enter a command. If neither of these conditions is true, the "notACommandError" method is called within the input vector as the argument to print an error message stating that the input command is not recognized.

**Custom Command Description/Explanation**

**Custom Command[current]:** The **getCurrentTrends** method is a member function of the AdvisorBot class that takes in a vector of strings representing a user input command and its arguments. The purpose of this method is to display the current trends (i.e. the minimum or maximum price) for a given type of order (either “bid” or “ask”) across all available products.

The method first checks if the input vector has the correct number of arguments (3 elements) to implement this functionality. If the input vector does not have the correct number of arguments, it prints an error message and returns.

Next, the method extracts the second and third elements of the input vector as strings, which represent the “min” or “max” command and the “bid” or “ask” order type, respectively. It then initializes a double variable called **price** to 0 and two vectors of **OrderBookEntry** objects called **entries1** and **entriesTemp**, as well as a vector of vectors of **OrderBookEntry** objects called **comparisonList**.

The method then checks if the “min” or “max” command and the “bid” or “ask” order type are valid. If either of these values is invalid, the method returns. Otherwise, it converts the order type string to an **OrderBookType** enum value and sets it to the variable **orderBookType**.

The method then iterates over the list of available product types and retrieves a list of **OrderBookEntry** objects for each product type at the current time step and for the given order type (either “bid” or “ask”). It stores each list of **OrderBookEntry** objects in the **entriesTemp** vector.

**Custom Command[exit]:**The **exit** method is a member function of the AdvisorBot class that takes in a vector of strings representing a user input command and its arguments. The purpose of this method is to close the AdvisorBot program when the user enters the “exit” command.

The method first checks if the input vector has the correct number of arguments (1 element). If the input vector does not have the correct number of arguments, it returns without doing anything.

If the input vector has the correct number of arguments, the method checks if the first element of the input vector is the string “exit”. If it is, the method prints a message thanking the user for using AdvisorBot and then exits the program using the **std::exit** function.

The **exit** method is called in the **processUserInput** method when the user enters the “exit” command as the first element in the input vector. If the input vector satisfies the conditions in the **exit** method, the program will close.

**Improvements to be made**: Currently, when the “current” command is being inputted, my error message will be thrown to handle the exception. I have not been able to fully implement it to my desired output, as shown in my function, due to the “vector subscript out of range”, which I have not been able to debug, and I attribute it to my lack of experience in c++.

**Optimising Of Exchange Code/Description**

* the **parseInput** function takes in the input string and tokenises it using the **split** function, which breaks the input string down into a vector of individual words. This allows for easier processing of the input as it can be accessed by indexing the vector rather than searching through the entire string.
* the **getStats** function has been optimised by breaking down the input string into its individual components (product, time frame, and type) and then using these components to retrieve the relevant data from the orderBook object directly. This is more efficient than searching through the entire **orderBook** object for the relevant data.
* The **getStats** function also calculates the min and max prices for the given product and time frame, rather than retrieving all orders and then calculating the min and max prices from the entire list. This reduces the amount of data that needs to be processed and makes the function more efficient.

In conclusion, the critical difference between my advisorbot and the exchange code is that I have added the required commands' workflow and validation of user input to smoothen the user experience. My code is more efficient because it uses a **validateUserInput** function to check if the input currency is a valid type instead of the exchange code, which hardcodes the list of products that it accepts. This makes it more flexible and adaptable to handle a broader range of products.

Next, it processes each product one at a time and gets the required data for that product instead of getting all of the data for all products simultaneously, as in the exchange code. This reduces the amount of data that needs to be processed and stored in memory, making the code more modular.