A simple Automated Trading Strategy

Elías Ron

Abstract
A basic framework was implemented in order to perform backtesting of automated trading strategies. The framework is able to run different algorithms simultaneously (i.e. running only once over historical data) , monitor their performance and draw comparisons amongst them.

Contents

Introduction

The code has been developed in python 2.7, and it can be found at:

https://github.com/elelias/idlabAssign

It makes use of standard python libraries with the exception of `matplotlib` which is used for plotting. It can be run ”out of the box” by providing the .csv file for the stock historical data. Additionally, if a .csv with the dividends is provided, such dividends will be taken into account to properly evaluate the daily value of the portfolio. To run it out of the box, one can run:

python tradingPlatform.py orderedTable.csv dividends.csv

Both tables can be downloaded from the repository on github.

1. Structure of the platform

To best describe the structure of the platform, the building blocks are described:

- the algorithm: represents the trading strategy
- the trader position: represents the position and operations performed by the trader.
- the stock history: it records the historical data of the stocks being traded.

These are all implemented as python classes and they interact with each other to decide what trades to perform, execute them, and keep track of the performance. Let's describe them with some detail:

1.1 AlgorithmRSI

This class, coded in `algorithmRSI.py`, represents an algorithm that performs trades using the *Relative Strength Index* (*RSI*), which is defined by:

$$RSI = 100 - \frac{100}{1+RS} \tag{1}$$

where RS corresponds to the average gains over average losses of the stock value in the last N_P periods:

$$RS = \frac{1}{N_P}\sum_{i=1}^{i=N_P} G(i) / \frac{1}{N_P}\sum_{i=1}^{i=N_P} L(i) = \frac{avgG}{avgL} \tag{2}$$

Given a new gain G and a new loss L, one can relate the new averages with their predecessors:

$$avgG_i = \frac{avgG_{i-1} * (N_P - 1) + G}{N_P} \tag{3}$$

$$avgL_i = \frac{avgL_{i-1} * (N_P - 1) + L}{N_P} \tag{4}$$

The *RSI* index is a momentum oscillator that measures the speed at which the stock price changes. It is generally considered that a stock is overbought if the *RSI* index is above 70, and oversold if it is below 30.
The strategy that this algorithm implements is to sell the stock whenever the *RSI* index is above a certain threshold, and to

buy it whenever it is below another threshold. The specific values for these thresholds are varied and the sensibility to these changes is studied.
Each possibility corresponds to a different instance of the same algorithms. The thresholds, together with the number of periods considered, are parameters that can be given externally and are processed in the class constructor.
The main method of this class is **`make_decision()`**, which makes a decision of whether to buy, sell or close the position in which the trader currently is.
For the sake of simplicity, if the algorithm decides to go long, the trader is only allowed to buy as many shares as it can with the cash available (taking into account trading costs). Conversely, if it decides to short the stock, it is only allowed to short as many shares as it would have been able to buy, to limit the amount of potential shorted stock. Also, if the algorithm decides to sell while being long, or to buy while being short, the position is closed.
The algorithm accepts a placing of a stop-loss order, coded in `applyStopLoss.py`, via a parameter that can be set externally and processed in the constructor. The improvements in performance due to this order are also studied.

1.2 TraderPosition

This class, coded in `traderPosition.py`, that represents the position of a trader who is trading under a certain algorithm, for example the previous one. This means that there are as many instances of this object as algorithms in use. The class accounts for the number of stocks of every symbol that the trader has (at the moment, only 'SPY' is used) and the daily value of the portfolio amongst other things. Most importantly, this class is responsible of executing the trade orders that the algorithm decides, which it does by calling the method **`execute_decision`**`(decision)`, where `decision` is the output of the algorithm to which this position is associated. In this way, trader and algorithm are only loosely coupled and one of the paradigms of object-oriented programming is respected.

1.3 StockHistory

This class, coded in `stockHistory.py`, represents the historical data of a set of a stock. One can retrieve statistical information such as the the mean or standard deviation, and also retrieve past data. An instance of this class is used by the algorithm as part of the decision-making process.

1.4 Putting it all together

In `tradingPlatform.py` an example of how to use the previous classes to perform trades using the *RSI* algorithm is found. For the stockHistory, only one is needed since the history is unique. If one wishes to test several algorithms simultaneously, one needs to create one instance of the algorithm as well as one instance of `TraderPosition` that goes together with the algorithm. These two instances are bundled together into a python tuple.
The function **`make_money`**`()`, coded in `makeMoney.py` is then called. This function loops over the historical data and it is the context in which the stock, the trader, and the algorithm finally meet and talk to each other. For every trading day and for every algorithm, the instance of the algorithm is fed with the stock historical data, a decision is performed and this decision is communicated to the trader, which then executes it. After executing the decision, possible dividends are processed and added/subtracted to/from the cash account if the trader is long/short.

1.5 Trading rules

Apart from the rules which are specific to the algorithms, it is assumed that one can only trade at *Open* prices. It is not possible to know in real time what the values at *High*,*Low* and *Close* are. However, it is possible to know at *Open* what the *Close* price was the day before, so even though this price is not considered for actions, it is a valid "period" in the sense that it enters the computation of *RSI*.

2. Results and Discussion

2.1 *RSI* (30,70,14)

The first case that is studied is that for the *RSI* algorithm with the standard parameters. This means that the stock is bought if $RSI < 30$ and it is sold if $RSI > 70$. The expectation is that after a large drop in the value, it will go back up, and ideally one would like to posses the shares to profit from this increase in value, or, if one is short on the stock, close the position to avoid losing more value. Also, the number of periods which are considered to compute *RSI* is 14.

Figure 1 shows the result of this algorithm. The black line represents the value of the portfolio as a function of the number of trading days. The green line represents the value of the underlying stock, multiplied by a factor of 1000 for the sake of visibility.

On top of the portfolio value, one can see when the decisions of buying, selling and closing were made. Sharp decreases are usually triggers of buys and sharp increases are triggers for sales, as can be seen in the figure. Also, straight lines follow the execution of a close order, due to the fact that money not invested generates no interest and the value of the portfolio stays constant. Interestingly, this strategy seems to lose money in the period considered. The first thing that strikes the eye is that there are periods of considerable loss, especially in the vicinity of the trading days 1000 and 2000. The most obvious remedy for this situation would be to add a stop-loss order that limits the loss of the portfolio. This can be achieved by setting:

```
'stop_loss':True
```

when creating the instance of the algorithm. The stop-loss order applied here closes the position if the portfolio losses a 10% of its value since the last trader order was executed. Figure 2 shows the new performance when the stop-loss order is used, and indeed, the performance is significantly better.

2.2 Sensitivity to the parameters

One can play around with the parameters to get a feel of how sensitive the algorithm is to them. In figure 3 a comparison is drawn among several choices of the parameters. These results can be achieved by running:

```
python compareRSIParameters.py orderedTable.csv
dividends.csv
```

As can be seen on the figure, small variations in the parameters yield significant differences in the performance. Lowering the buying threshold from 30 to 22 accentuates the drop needed to trigger a buy. This facts accounts for the algorithm taking a short position on the stock during the large drop in stock value at the beginning of the trading period (d̃ay1 to day 800), which yields large gains in the value of the portfolio, as shown by the blue line.

Conversely, if one increases the value of the selling threshold, it takes a larger increase in the stock value to trigger a sale. This is the reason the red line, with a threshold of 75, performs so well at the last part of the trading process. The trader holds the stock during the large increase in value, and the occasional sharp increases in value are not enough to trigger their sale, which benefits the trader.

However, this does not happen for the green line, even tough it also has a sale trigger at 75. The reason is that the buy threshold is very low, so when a buy trigger was issued for the red-line trader, which led to a large gain, this did not happen for the green-line trader due to its low buy threshold, missing the chance of profiting from the large increase in stock value. It is thus clear that the strategy is extremely sensitive to the specific value of the parameters, which I believe makes this method hard to fit and very prone to over-fitting.

On table 1 the values of the Sharpe ratio for the different algorithms are shown. The Sharpe ratio was obtained with:

$$s = \sqrt{252}\,\frac{r_x - r_i}{\sigma_x} \tag{5}$$

where r_x was the average daily return of the portfolio and σ_x the standard deviation. r_i is the risk-free interest rate which for the purpose of this exercise was assumed to be zero. The values obtained for these strategies are quite low, even negative, signaling than even when the return was positive, the risk taken to achieve it was excessive.

Table 1. Sharpe ratios for the different algorithms

Algorithm	Sharpe Ratio
RSI(22,70,14)	0.16
RSI(22,75,14)	-0.31
RSI(30,75,14)	0.27
RSI(30,70,14)	-0.05

2.3 Improvements and refinements

Ideally, I would liked to code up another strategy to compare against the one reported here. Now that the framework is ready, one would only have to code up the algorithm, and substitute the instance of the *RSI* for the instance of the new one. As long as it has the same API, it should be fairly easy. Also, I would have liked to extend the possibility of building a portfolio to more than just one more symbol. Most of the code is already prepared for that situation, the only unknown is how the information would be read and brought into the code.

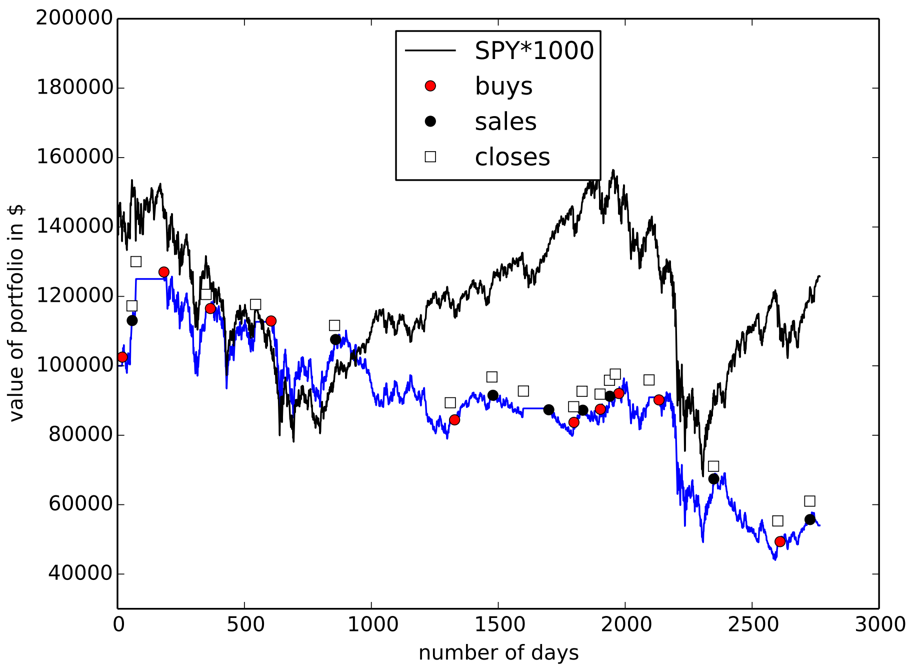

Figure 1. value of a portfolio trading under the *RSI*(30,70,14). The blue line represents the value of the portfolio.

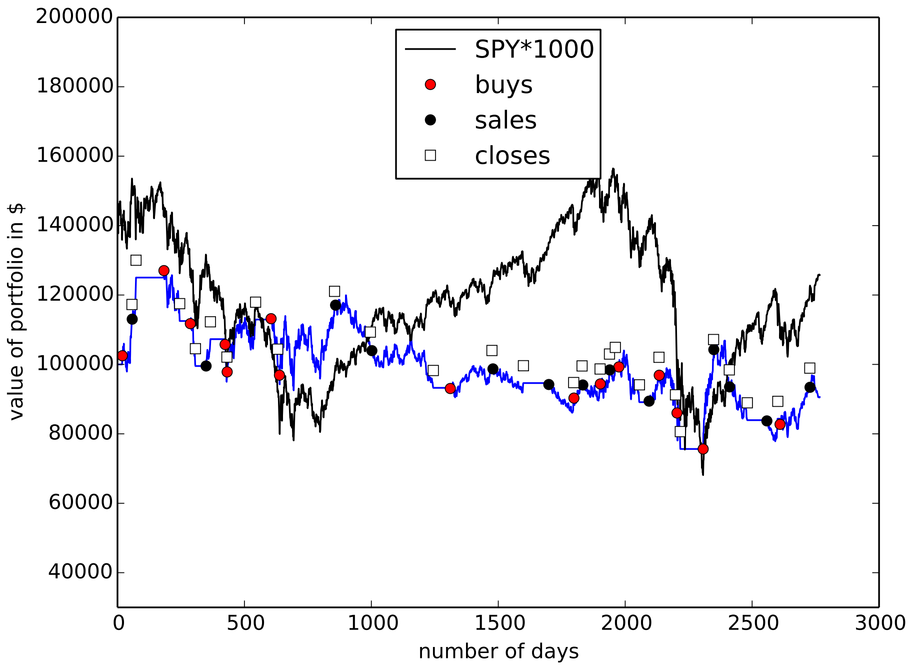

Figure 2. value of a portfolio trading under the *RSI*(30,70,14) with a stop-loss order that closes the position if the loss is bigger than 10% of the portfolio value in the last action. The blue line represents the value of the portfolio

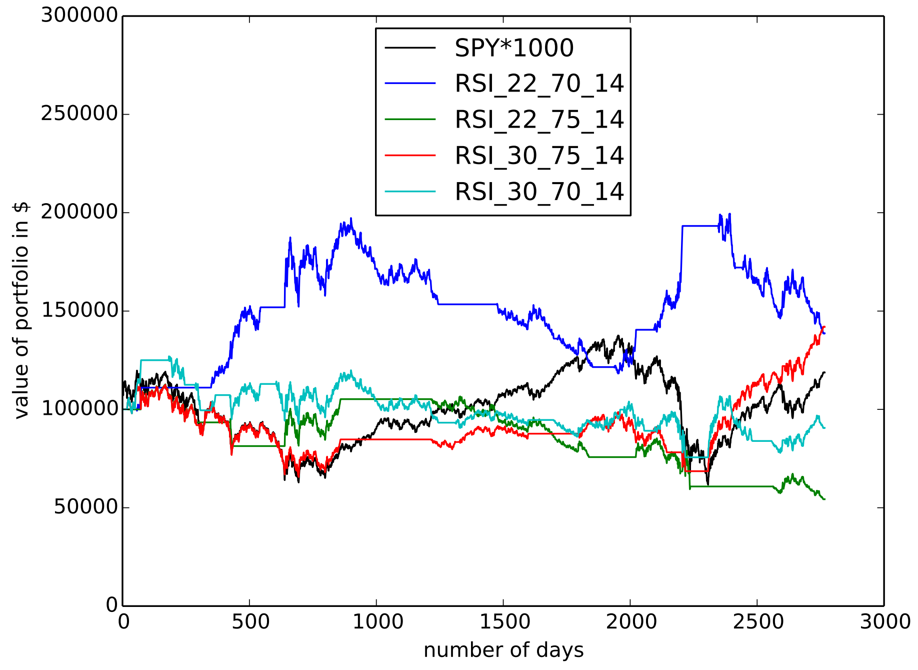

Figure 3. Comparison between the performance of several RSI algorithms which differ in their value of the parameters. A stop-loss order was used in all of them