Harsh Sikka
CS 7646
12/04/2017

Project 8: Strategy Learner

Brief Note:

The goal of Strategy Learner is to build either a Regression, Classification, Optimization, or Reinforcement based model and use it to narrow down on an effective trading strategy. My personal goal is to have results better than those that I received using the Manual Strategy based trader that was submitted as a part of Project 6.

The general format of this report is broken into two parts, and I include figures and statistics followed by rigorous explanation of the methods involved as well as an analysis of their efficacy with the trading challenge. This structure allows me to keep the total word count under the limit while also maintaining a thorough, informative and hopefully enjoyable experience for the reader.

Part 1 - Learner and Methods

For this particular project, I elected to use the Q Learning model that was designed in Project 7 to solve the navigation problem given. Q Learning is an excellent approach for a lot of general issues because it is model free, and can be applied to instances where the state of a problem is not completely fleshed out. Because of this, there isn't really any sort of need to use particular data structures to represent the state or reward, and I opted to use Professor Balch's experience tuple method in my implementation.

The key detail that is of the utmost importance when implementing and using a Q learner is the framework in which you will wrap it to feed it the state and reward data in order to properly train it. In this assignment, that wrapper can be found in *StrategyLearner.py*. While we could have implemented it in whichever way we wanted, I focused on a method similar to what was outlined during lecture, and I've broken down, along with particular decision choices like discretization, below.

The Strategy Learner has two primary methods addEvidence and testPolicy, focused on training and testing respectively.

Training

The first method, addEvidence, works by taking in a particular stock symbol, date range, and starting value. Using these pieces of information it then generates the same indicator that was used in my Manual Strategy project, BB Value. BB value is actually a composite indicator that follows a simple equation, making use of both the Simple Moving Average, as well as the Volatility, given by the standard deviation. BB Value’s equation is as follows:

```
bb_value[t] = (price[t] - SMA[t])/(2 * stdev[t])
```

It is important to discretize this value, allowing us to convert the continuous, real numbers that make up BB Value into integers that can be factored into the state being fed to the Learner. I followed the discretization equation set forth in lecture, looping through a 100 states rather than ten, since I’m only using BB Value as I did in my manual strategy.

Upon discretizing the values, I begin training my learner, checking convergence or running for a maximum of 50 epochs. This allows my Strategy Learner to understand that training is complete. Within each iteration, the program loops through the dates, feeding state made up of the discretized BB Values, as well as a simple reward function that allocates the daily return, multiplies it by current holdings, and then takes a away a portion based on the impact score of the market. This is particularly important for Experiment 2 in part 2 of this report. The actions outputted by the Learner can be 3 different types, and are used to generate an trading orders data frame when it has completed its 50 epochs and is queried for testing.

Testing

Testing my learner is a very straightforward process, making use of the testPolicy function, which takes in the same parameters as addEvidence, a stock symbol, date range, and starting value. It also discretizes the bb value just as addEvidence did. The only real difference between this method and the one before is twofold:

1. testPolicy doesn’t loop through 50 epochs, as it only needs to query the q learner once per date
2. It also doesn’t feed any reward data to the learner, only feeding the state through the querystate method.

These 2 differences make sure that testPolicy isn’t training the learner, and is simply allowing for it to generate a trades data frame based on 3 actions, corresponding to Buying, Selling, or doing nothing. This data frame is then returned and can be passed to marketsim to ascertain its performance.

So in effect, testPolicy acts just like addEvidence, but without the training and unnecessary epochs, only running through the date range one time.

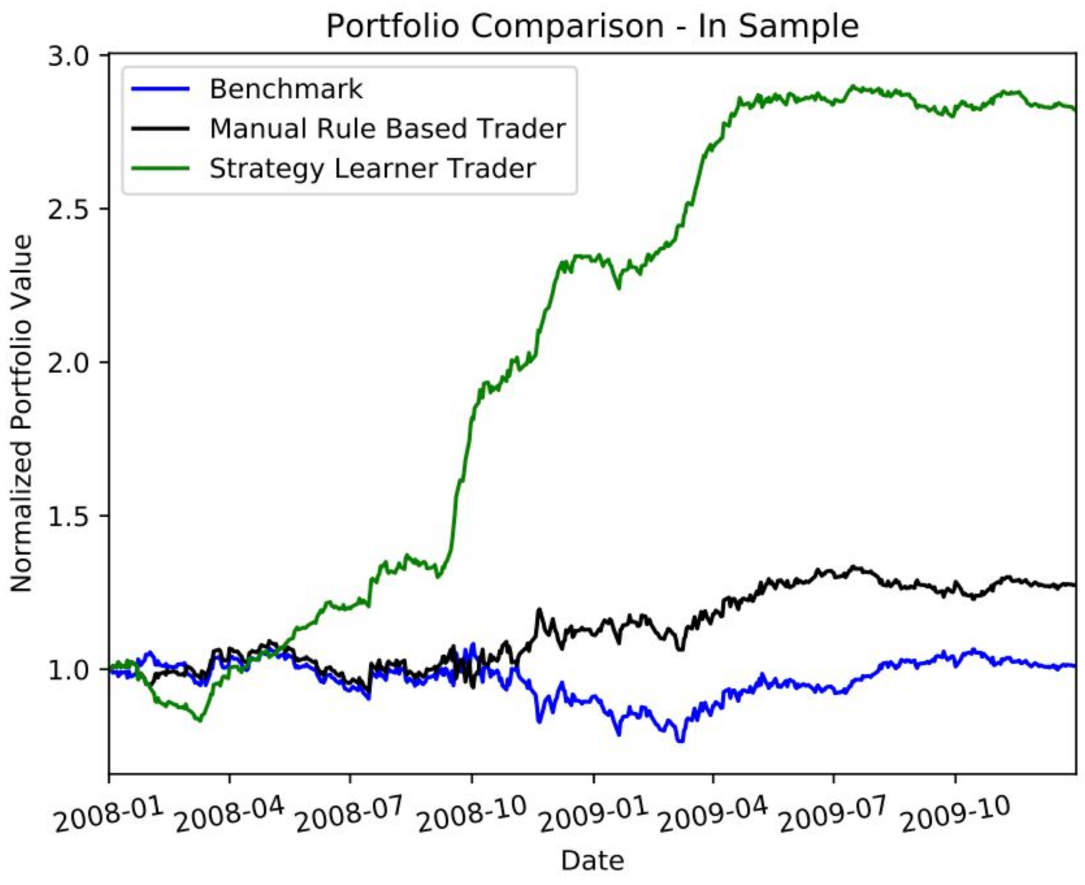
Part 2 - Experiments and Results

Part 2 consists of a two experiments, where the learner was tested against the Manual Strategy, as well as under different market impact conditions. Note that upon running my code, the values that will be returned will be very slightly different than the ones reported below. This is because of the random reward nature of the Q Learner, something I chose not to seed and control for to maintain realism and demonstrate that a Learner is truly a dynamic agent.

Experiment 1 - Manual Strategy vs Q Learning

In Experiment 1, we are tasked to rerun Manual Strategy in the exact same conditions as we were instructed to do previously, with the same indicators. There were also some key assumptions I made to make for an equal comparison.

For the sake of accuracy, I also used the same market conditions, including a 9.95 commission and 0.005 impact score sourced from Project 6. The results are below, followed by an in depth analysis. To generate the figures and tabular data yourself, simply run the experiment1.py file in the assignment directory. This comparison was generated by feeding both Strategy Learner and Manual Strategy with an in sample window for the stock JPM. The scores are normalized.



	Benchmark	Manual Strategy	Strategy Learner
Cumulative Return	0.0123249333401	0.275448732786	1.81879291807
Mean of Daily Returns	0.000168759162146	0.000593273911236	0.00210231741398
Standard Deviation of Daily Returns	0.0170412470682	0.0149012919643	0.00942401287156
Final Portfolio Value	101027.7	127286.85	281309.05

The results above are really astonishing, and demonstrate how a simple Q Learner can absolutely demolish the returns, even versus a fairly successful indicator based manual strategy. The values of the Benchmark, which is simply buying 1000 shares of JPM and holding them, and Manual Strategy have a return of 1.2 and 27.5 percent respectively, while the Strategy Learner has a cumulative return of nearly 182 percent.

Running Strategy Learner over several hundred iterations on this data consistently saw it achieving between 250,000 and 320,000 in its final portfolio value. The variability is due to the inherent random reward mechanism in the Q Learner, using the RAR and RADR parameters.

This success versus Manual Strategy was actually an expected outcome, as we are training the Q Learner on in sample data and then testing it on that very same data. I would imagine that it actually performs more poorly in out of sample data, because it hasn't been trained on that data. What is again interesting however, is that just like Manual Strategy, Q Learner is also only using BB Value to determine its trades, and against other versions of the learner where I implemented other indicators, it performs more successfully. I suspect this is because BB Value takes into account both the stability and overall value of the stock.

Note:
To run the experiment above, I simply used the MatplotLib libraries after running my Learner and Manual Strategy and passing them to Marketsim to gain my results. The stats were generated using helper functions in ManualStrategy.py, and everything is available upon running experiment1.py.

Experiment 2 - Manual Strategy vs Q Learning

In Experiment 2, we're tasked to hypothesize and test the corresponding results of applying varying degrees of impact to our learners. I've included my hypothesis and assumptions, followed by the experimental procedure, results, and analysis of my findings.

Hypothesis and Assumptions

I believe that Impact will affect the strategy learner by reducing its reward, making it less likely to trade in a greedy fashion. The more impact there is in a market, the more my particular strategy learner should become conservative as certain purchases will no longer be as tantalizing and will actually result in less value gained for the user.

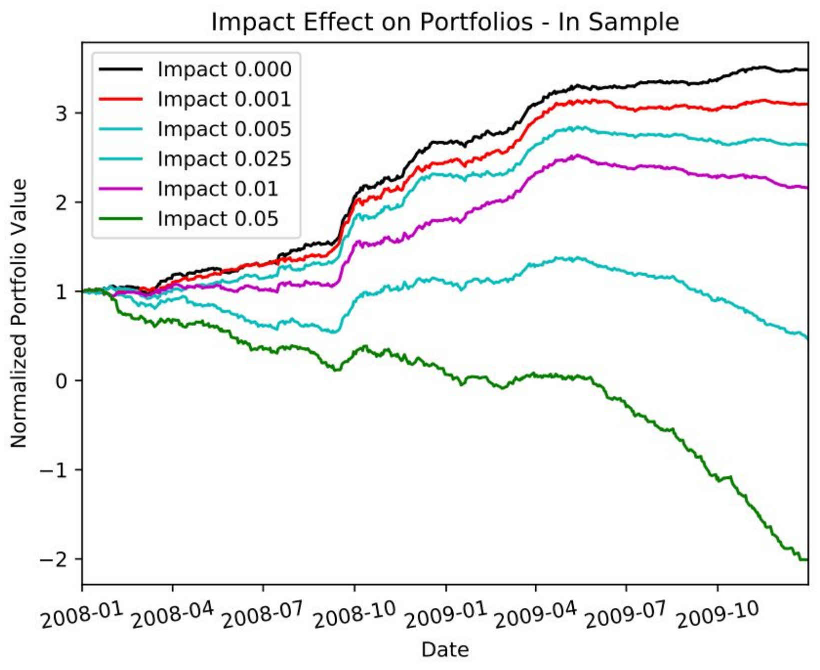
2 Metrics I would use to quantify this are the gross number of trades, which I suspect will be less as impact increases because the Learner should become more conservative due to reward lessening, and overall final portfolio value, which should be less since the learner is more conservative with its trades and may not shoot for riskier gambits. However, it should also be less volatile theoretically, and that is something we will observe as well. I've included a large number of measured analytics and metrics following the graph.

I'm making several assumptions in my statements above and my experimental procedure, which I will outline below. The first of these is that we are really talking about the Learner that I designed, or Learner's like it. Theoretically reward should be affected by impact because of the reasons I stated above, since the value is not complete until the impact score is applied and taken away from it. For the sake of the experiment I removed any commision from the operation, so we could observe the pure impacts of impact (hehe pun) on the various learners, each initialized with the corresponding impact that the market has and using the reward function I outlined in my addEvidence description.

Experimental Procedure and Results

The way I executed this experiment is actually very straightforward. I had created functions that calculate simple statistics like volatility and average of daily returns for the sake of my manual strategy and Q Learner comparison. These numerous metrics, along with the number of trades being observed are collected and analyzed below. MatPlotLib was used to plot each learner normalized and alongside each other. You can see the functions responsible and generate the charts and tabular data by running experiment2.py

The data was all gathered using the in sample window for the same stock, JPM, to demonstrate with the most clarity possible.
For the sake of demonstration, the table on focuses on 3 learner's metrics so that we can analyze each one in detail



	Impact of 0.005	Impact of 0.01	Impact of 0.05
Cumulative Return	1.63698073244	1.16213071687	-3.00908984313
Mean of Daily Returns	0.0019715447253	0.0015877215334	0.0147610718215
Standard Deviation of Daily Returns	0.00961738116625	0.0106837506488	1.26509106216
Final Portfolio Value	263190.85	215381.3	-197044.5
Total Trades	186	176	160

These are three very different impact scores, with the first one being the default parameter for many of our class projects, including Experiment 1 in this report. While I took a look at several different metrics, I think some are particularly important in the context of this report.

The first of these is the total number of trades by each learner, which with an increasing impact clearly decreases, going from 186 to 176, and finally 160. This is inline with the hypothesis put forward above, indicating that the learner becomes progressively more conservative over time and making less trades since its reward is now being impacted.

Another interesting supplement to this is the observation that the final portfolio values significantly decrease. Ignoring the largest impact values as those become negative due to what seems to be an undeveloped strategy or reward, even the ones that are scoring well are sorted by increasing impact size. This may have to do with the Learner's behavior, something I mentioned during my hypothesis. Since the Learner isn't making as many risky trades, its portfolio value may be more stable and not reach such heights.

One shocking thing that is observed is that the volatility actually increases, which isn't what I would have expected, I would have thought that the learner would decrease in volatility, making less risky decisions. But perhaps since it is now holding on to more trades, its portfolio is subject to more dips. These mistakes can all potentially be fixed by improving the rewards mechanism to be more robust, allowing the learner to properly understand that making trades and holding them may or may not be the best decision in the context of the current market based on what it knows previously. Of course, these results more than answer the research question and hypothesis set forth in Experiment 2, and improving it may only be necessary if one intends to continue to work on this, perhaps in a real world trading environment. After all, it was someone new to programming who wrote this ;)

Going forward, I think there are a few key takeaways from this project's experiments:

1. A well trained and designed Learner can be more successful than a given manual strategy, and that it should be more successful if trained and then tested on the same date. This is visible in our results during Experiment 1, and essentially reminds us that dynamic systems like the Q Learning model are a powerful tool, when they aren't overfitting. It is important to note it wouldn't have performed this well out of sample.
2. Market conditions and indicators can seriously impact or improve a Learner's performance, many of the Learner examples used with various degrees of impact above performed fairly well under their corresponding conditions, albeit conservatively.

This was an awesome project, and I learned a lot. I'm looking forward to applying these new found skills in my future journeys!