Optimized LinUCB for Simplified Day Trading

Matthew Fong
The University of Texas at Austin
jmfong@cs.utexas.edu

Hassaan Markhiani
The University of Texas at Austin
hassaan@cs.utexas.edu

ABSTRACT

In this paper, we present an approach to managing a day trader's stock portfolio. By viewing the investment decision making as an armed bandit problem, we can choose the stocks to invest in based on the expected reward. More specifically, we view the problem as a contextual bandit because the expected reward varies based on the context of the stock for any given day. An interesting attribute of investing is that the reward for all arms, or stocks, is presented after a choice is made, so we apply a modified version of the LinUCB algorithm to take advantage of this property. To limit the problem for experimentation, we only consider the top 100 NASDAQ companies and always invest (long) in 5 stocks per day. Using an optimized set of stock features discovered from a policy search algorithm, we are able to obtain a compound annual growth return of 38.7% over a 3 year training and 2 year testing time frame.

Keywords

LinUCB, UCB, n-armed bandit, least squares methods, stock trading, reinforcement learning

1. INTRODUCTION

Day traders are investors that buy and sell financial instruments all within the same trading day, such that all held positions are closed before the market closes for the trading day. Their goal is to maximize performance metrics, such as profit or risk-adjusted return, from many short term gains. In this paper, we present an approach to managing a day trader's stock portfolio, i.e. decide which stocks to invest in every day. The process of choosing the stocks to invest in every day can be pictured as an armed bandit problem. With this approach, stocks can be chosen based on their expected reward. However, the expected reward for a stock is not consistent and largely varies based on the given context, so this led us to consider the problem as a contextual armed bandit [2], as opposed to the standard armed bandit.

[2] presents the LinUCB algorithm for personalized news recommendation as a contextual armed bandit, which we felt was ideal for our problem. In the armed bandit problem, an arm is chosen, and reward is obtained for the selection of that arm. The reward is then used to decide how to select arms in the following iterations. However, an interesting attribute of investing is that the reward for all arms, the daily performance of the stocks, is presented after a choice is made, so we modified the LinUCB algorithm to take advantage of this property. One downside with this approach is that it is stateless, i.e. the algorithm does not keep track of the sequence of actions chosen. [1] presents a direct reinforcement (policy search) approach to optimizing portfolios, which keeps track of state.

The scope of this project can easily become large, so to guarantee the progress of the project, we simplified the day trading problem. First of all, we only consider the top 100 NASDAQ companies. The idea here was to limit our choices to the companies that are more likely to be stable. Secondly, day traders perform anywhere from just a few trades to hundreds of trades per day, but for our problem, we only consider a trader that buys 5 stocks at the beginning of the day and sells those 5 stocks at the end of the day. More specifically, we only consider long selling and exclude the option to short sell a stock. Finally, we decided to limit our data to the years 2008-2012 to keep it fairly recent.

The remainder of the paper is structured as follows. Section 2 discusses the dataset used for our experimentation. Section 3 gives an overview of the LinUCB and the policy search algorithms. Section 4 describes our experiments. Section 5 presents our results. Finally, section 6 concludes. Section 7 details how to obtain our code.

2. DATA

All of the data we used was downloaded from Yahoo! Finance, and we were therefore limited to using only the historical data that they provide via their APIs. This is a small data set compared to what is available for stock quotes in real time. The only data points that are available are the open, close, high, low, and volume for a company on a day in the past. Because of this, only limited simulation could be done to evaluate the performance of our algorithm,

and we were forced to make a number of assumptions when testing. These assumptions are listed in Section 4.

For the majority of our experiments and tests, we used data from the NASDAQ 100 most traded companies for the last five years (2008 - 2012). We chose this data set because we wanted to use companies that are slightly more stable and are less likely to have extreme price fluctuations from day to day. If we had traded with companies that were more volatile, our trading strategy would make us extremely vulnerable to losing large amounts of money if a company's stock price loses a large percentage of its value. Using the NASDAQ top 100 companies insulated us in two ways. First, they tend to have higher stock prices, so when they lose value, it tends to be a smaller percentage of their value. These stocks also have a generally upward trend, meaning we are more likely to succeed if we make the right selections from day to day.

We also experimented with using a dataset of random stocks chosen from the NYSE, with less than optimal results. This suggests that using only the top 100 companies in the NASDAQ may have been a crucial to our success. The results of our experiment with the NYSE random stocks can be seen in Section 5.

3. ALGORITHM

Before discussing the LinUCB algorithm, we introduce the original multi-armed bandit problem. Next, we give a small overview of UCB, the algorithm LinUCB stems from. Then, we provide the details of our modified LinUCB algorithm. Following the LinUCB subsection, we describe the feature set used for our stocks and provide an overview of our policy search algorithm, which was used to optimize our feature set.

3.1 Multi-Armed Bandit

The traditional multi-armed bandit is the problem of being faced with k independent gambling machines with independent reward functions. Every time frame, an agent pulls one of the k independent arms and receives a reward. The goal of the agent is to maximize the total reward received over time (Equation 1). One simple approach to the problem is to pull each arm once and to then keep pulling the arm with the maximum observed average reward.

$$Equation\ 1 : \sum_{t \in T}^{T_f} r_t$$

3.2 UCB

Another approach to the problem is the UCB algorithm [3]. The idea behind UCB is similar to the approach described above with one modification. UCB selects the arm with the largest upper confidence bound, or in other words, the arm with the possibility of having the largest reward. The idea behind this is to achieve logarithmic regret uniformly over time [3], where regret is the expected loss due to the fact that the optimal arm is not always selected. Formally, UCB selects the arm that maximizes Equation 2, where x_a is the average reward for arm a, n_a is the number of times arm a has been pulled, and n is the total number of times all of the arms have been pulled. There are multiple variations of the UCB algorithms, each with a slightly different calculation of the upper confidence bound.

$$Equation\ 2 : x_a + \sqrt{\frac{2ln(n)}{n_a}}$$

3.3 LinUCB

The LinUCB algorithm is an extension of the UCB algorithm for the contextual bandit problem, where the confidence interval is computed in the closed form for a linear payoff model. LinUCB assumes that the expected payoff, $r_{t,a}$, of an arm is linear in its d-dimensional features, $x_{t,a}$, with some coefficient vector, Θ_a, as shown in Equation 3 [2]. Please note that the version of the algorithm presented here is for disjoint linear models because parameters of the coefficient vectors are not shared between different arms. The algorithm estimates Θ_a by Equation 4, where A_a is a matrix of the feature model of arm a and b_a is a vector of the reward-feature multipliers of arm a. Finally, the algorithm selects the arm that maximizes Equation 5, similarly to equation 2 for UCB, at each time frame, where α is the tunable parameter.

$$Equation\ 3 : E[r_{t,a}|x_{t,a}] = x_{t,a}^T \bullet \theta_a$$

$$Equation\ 4 : \theta_a = A_a^{-1} \bullet b_a$$

$$Equation\ 5 : x_{t,a}^T \bullet \theta_a + \alpha\sqrt{x_{t,a}^T \bullet A_a^{-1} \bullet x_{t,a}}$$

The version of LinUCB presented by [2] only selects one arm at each time frame and only receives the reward for that particular arm. Our modified version of the algorithm selects the top five arms that maximize equation 5. Unlike the application of LinUCB in [2], our application allows us to observe and utilize the reward for all of the arms, which our modified version of the algorithm takes advantage of. Algorithm 1 presents the details of our modified version LinUCB. The algorithm loops through each arm at every time frame and obtains $p_{t,a}$ (Equation 5) for every arm. Then, the algorithm selects five arms with the largest p values. Finally, the algorithm updates the A_a and b_a for each arm a based on the

features and rewards for each time frame. We informally experimented with varying types of rewards (profit, return, and Sharpe ratio) and noticed that return provided the best results.

Algorithm 1 :

Inputs : $a \in \mathbb{R}_+$

for $t = 1, 2, 3, \ldots T$ *do*

Observe features of all arms $a \in A_t : x_{t,a} \in \mathbb{R}^d$

for all $a \in A_t$ *do*

if a is new then

$A_a \leftarrow I_d$ ($d-$*dimensional identity matrix*)

$b_a \leftarrow 0_{d \times 1}$ ($d-$*dimensional zero vector*)

end if

$\theta_a \leftarrow A_a^{-1} \bullet b_a$

$p_{t,a} \leftarrow x_{t,a}^T \bullet \theta_a + \alpha \sqrt{x_{t,a}^T \bullet A_a^{-1} \bullet x_{t,a}}$

end for

Choose top 5 *arms* a_1 *through* $a_5 = argmax_{a \in A_t} p_{t,a}$

where each time the selected arm is removed from A_t

Observe rewards of all arms $a \in A_t : r_t$

for all $a \in A_t$ *do*

$A_a \leftarrow A_a + x_{t,a} \bullet x_{t,a}^T$

$b_a \leftarrow b_a + r_t \bullet x_{t,a}$

end for

end for

3.4 Features

The performance of LinUCB depends heavily on the set of features used to represent a stock at each time frame. We informally experimented with varying sets of features to end up at our final set of features. All of our features are binary features. The first five of our features indicate the day of week, i.e. 00100 for features 1-5 indicates the current day is a Wednesday. [4] were able to achieve an increase in performance when using the day of the week feature, which led us to include those features as well. The next four features give information about the returns over the last two days. Features 6 and 7 indicate if the return yesterday was >-0.04 and <-0.04, i.e. 01 means the return yesterday was <-0.04. Features 8 and 9 indicate if the return two days ago was >-0.04 and <-0.04, i.e. 10 means the return yesterday was >-0.04. Essentially, feature 6 is the opposite of feature 7, and feature 8 is the opposite of feature 9. Even though it seems redundant to put two put features that are opposites of each other (features 6-7) in the feature set, we noticed that the performance of the system was better with the opposing features. The next two features indicate if the open price yesterday was > and < the average open price over the last five days, i.e. 01 for features 10 and 11 indicates that yesterday's open price was less than the average open price of the last five days. The next two features indicate if the slope of the open prices over the last two days was >1.2 and <-1.8, i.e. 00 for features 12 and 13 means the slope of the open prices over the last two days was <1.2 and >-1.8. The next eight features are the same as the last four features (10-13) but for close prices and the volume. Finally, the last feature, 22, indicates if yesterday's volume and open price were higher than the average of the volume and open prices over the last five days respectively.

3.5 Policy Search

The original performance of our system was noticeably less than our current system. To improve the system's performance, we had to optimize the parameters into the feature selection (number of previous days' returns to include, number of days to use for the running average and slope, the threshold values, etc.). To discover the optimal parameters, we employed a policy search algorithm. The algorithm begins with our original set of parameters and its compound annual growth return. Then, the algorithm loops until one of two conditions is met: 1) the current set of parameters remains unchanged for ten consecutive iterations or 2) max number of iterations has occurred. At each iteration, the algorithm constructs five perturbations of our current optimal parameter. Each perturbation is simply the current parameter plus a random whole number between -5 and 5 times epsilon for that parameter. The compound annual growth return is obtained for each perturbation. The current parameters are updated to the perturbation with the largest compound annual growth return, if it is larger than the current parameters' compound annual growth return. Algorithm 2 provides the exact details. Our algorithm was derived from the policy gradient algorithm presented in [5].

4. EXPERIMENTS

During our experiments we compared our algorithm to two baselines. The first baseline was an agent that selects companies to trade at random. The second was an agent that selects the companies that had the best returns on the previous day. We also compared the effects of using different proportions of our data for testing and training, and the effect of different values of alpha on the performance of our optimized LinUCB.

Algorithm 2 :

```
Inputs : initial_parameters, perturbation_num. max_same, max_iteration
current_parameters ← initial_parameters
current_cagr ← runLinUCB(stocks, Featurizer(current_parameters))
i ← 0
same_current ← 0
while same_current < max_same and i < max_iteration do
    P ← []
    CAGR ← []
    for j = 1, 2, 3...perturbation_num do
        P.add(perturbation(current_parameters))
        CAGR.add(runLinUCB(stocks, Featurizer(P[j])))
    end for

    i ← i + 1
    same_current ← same_current + 1

    for j = 1, 2, 3...perturbation_num do
        if CAGR[j] > current_cagr
            current_parameters ← P[j]
            current_cagr ← CAGR[j]
            same_current ← 0
        end if
    end for
end while
```

4.1 Trading Strategy

During all of our experiments, we start our agents with $10,000 and measure their performance with compound annual growth rate (CAGR) and total stock return. At the beginning of the day, our agents choose 5 stocks to buy and split their money evenly between them. At the end of the day, all stocks are sold and we calculate the return and update the amount of money that the agent has. While it is possible that using or learning a more complicated strategy (such as buying variable amounts of each stock or short selling) could lead to better results, the strategy we used simplifies the task of the agents considerably. The only decision that an agent must make is what stocks to buy for the day. If the agents choose poorly and the stocks decrease in value over the day then the agents will lose money. If the agents choose mediocre stocks that only hold their value, the agents may still lose money because of trading costs. The only winning strategy is to choose companies that will have returns greater than the cost of trading shares of the stock.

4.2 Assumptions

Because of our limited data set, we were forced to make a number of assumptions during training and testing of our agents. The first assumption we made was that when we purchase and sell a stock we are able to buy and sell at exactly the opening and closing prices of the stock for that day, respectively. This assumption is necessary because while we also have data for high and low prices, this is the limit of our knowledge. Without more granular information about the movement of stock prices during the day, we cannot say what price we could have actually gotten. At the same time, assuming we can buy and sell at the opening and closing is not completely unreasonable because they are often in between the high and low prices, meaning the opening and close prices were passed at least twice during the day. This is also a reasonable assumption because we do all of our testing starting with only $10,000 so selling and buying all of stocks at the same time is very unlikely to have flooded the market. For example, if the agent had chosen to buy Microsoft stock, with the price of Microsoft stock currently at $33, we are only trading about 60 shares at a time when the average volume for the day is currently over 60,000; we are less than 0.1% of the market. Even more extreme, if we had purchased Apple stock at $463 we would have traded about 5 shares, about 0.03% of the market.

Another key assumption we made is that we are able to buy and sell shares at a cost of $0.01 per share traded. Retail stock brokers tend to charge $7 or more for trades, but paying that price is not practical for day trading and it is nearly impossible to absorb that cost no matter how well the agent performs. Our strategy assumes we can get direct day trading prices. We selected a cost at the high end of the range for what a direct day trader pays, which is typically between $0.01 and $0.002. This assumption was necessary for success, but moves our algorithm out of the range of being useful for anyone but a direct access stock broker.

4.3 Methodology

Evaluating exactly how well our algorithm is performing is difficult when using anything as volatile as the stock market. Because of this, both the amount of money our agent was able to make and the compound annual growth rate (CAGR) varied quite a bit for slightly different feature sets and for different sized testing sets. It is also worth noting that the only difference between testing and training was that the amount of money our agent has is updated during testing. This simulates making actual trades on the stock market. Learning is done during both training and testing, so the agent continuously improves. During testing we also have our other baseline agents make trades, so that all agents will have traded on the same days for the same amount of time.

During the day, all of the agent's money is invested in the five stocks it has chosen. At the end of the day, it sells all of the stocks it has, and we calculate the return it has made, shown in Equation 6 where r is

the return, P_c is the closing price of the stock and Po is the opening price of the stock. We calculate the cost of all transactions we've made for the day and update the amount of money the agent has using Equation 7, where *inv* is our initial investment in the stock (1/5 of our total money). The values are multiplied by two because we incur a trade cost both when we purchase and when we sell shares. The amount of money the agents have is updated using Equation 8, where m is the amount of money the agent has, r_{avg} is the average return of all stocks the agent traded and c_{total} is the total cost of all trades. We also use CAGR as a measure of how well the agents are performing over time (Equation 9).

$$Equation\ 6\ :\ r = \frac{P_c - P_o}{P_o}$$

$$Equation\ 7\ :\ c = \frac{inv}{P_o} \times 0.01 \times 2$$

$$Equation\ 8\ :\ m = m + (m \times r_{avg}) - c_{total}$$

$$Equation\ 9\ :\ CAGR = (m / 10000)^{1/years} - 1$$

5. RESULTS AND DISCUSSION

Our results were promising and our agent was able to significantly outperform both of our baselines in almost every case. In Figure 1, we used a testing percentage of 0.4, and in the end our algorithm manages to achieve a CAGR of 38.7%. If we extend our testing period, as seen in Figure 2, the algorithm does not perform as well in the beginning because it has received less training, but its total money grows significantly more and it manages to achieve a CAGR of 52.1%. The complete results of testing LinUCB compared to Random and PrevBest can be seen in Table 1. LinUCB also outperformed Random and PrevBest when we measure total return, which is also the total reward that LinUCB has achieved over the time period. We also compared performance of LinUCB for different values of alpha (Figure 4) but found that varying alpha largely did not make a difference on the performance of LinUCB, so we used an alpha of 0.1 for all of our experiments. A crucial factor in evaluating the performance of an agent is if it is able to overcome the transaction cost of buying stocks and actually make money. In Figure 3 we can see that both Random and PrevBest were able to achieve positive returns, but they were not able to make a profit.

The fact that our random agent was able to achieve positive returns is telling of the fact that selecting almost any company in the NASDAQ top 100 companies is not likely to hurt by a large amount. However, we also ran an experiment using companies from the NYSE that were much more volatile. The results are summarized in Figures 5 and 6. While our agent was able to achieve a positive return, it was not able to overcome the trading costs in order to make a profit and ended up losing money. However, the agent still managed to perform better than the baselines even in this more volatile market.

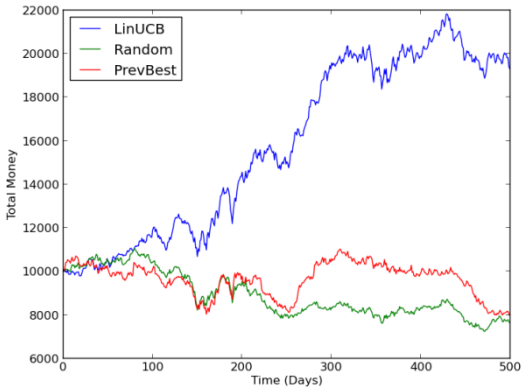

Figure 1. The amount of money held by the agents after training for three years and testing for two years.

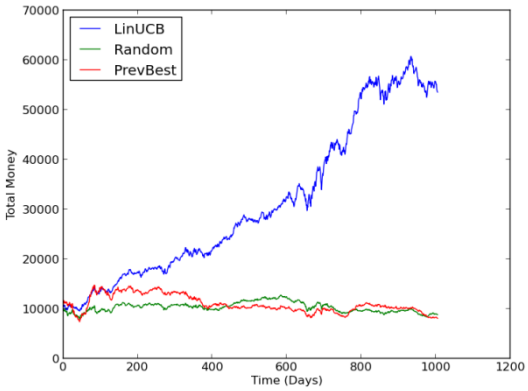

Figure 2. The amount of money held by the agents after training for training for one year and testing for four years.

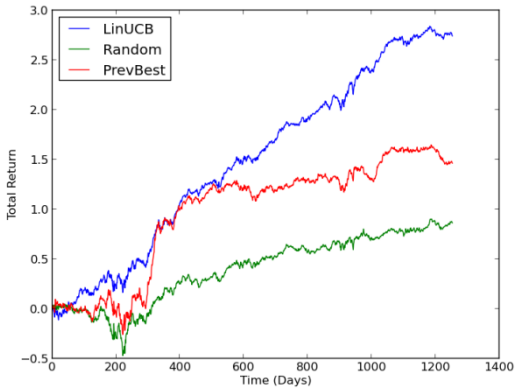

Figure 3. The total return for both testing and training periods.

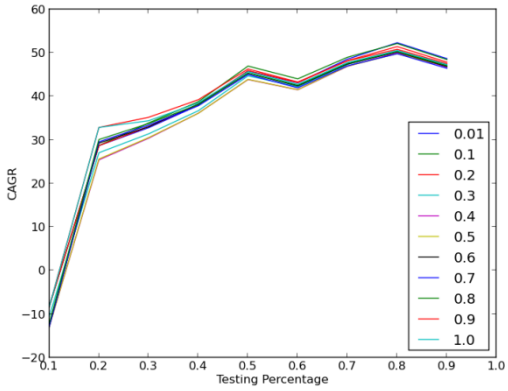

Figure 4. The performance of LinUCB with different values for alpha.

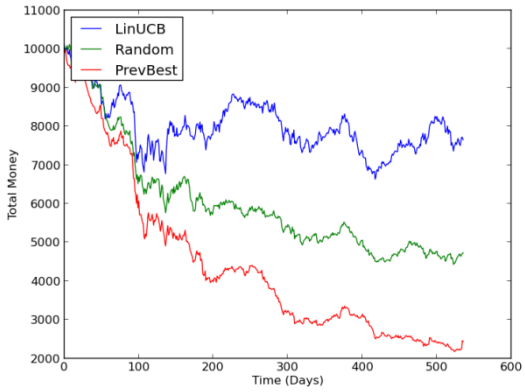

Figure 5. The amount of money held by the agents after training for three years and testing for two years on companies in the NYSE.

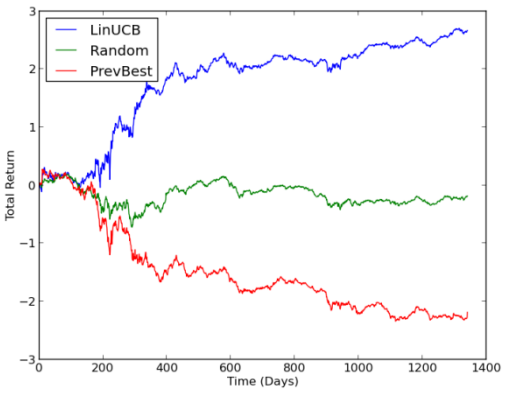

Figure 6. The total return for both testing and training periods on companies in the NYSE.

There are multiple possible areas of improvement for our LinUCB agent. While we optimized our feature parameters for trading the top 100 NASDAQ companies, it is possible we could have gotten better results on the NYSE by tailoring our features to that stock exchange. We also tried to incorporate a few features that were the output of a neural network trained to predict whether the stock would go up or down the next day. These features were not successful at improving our results, but we believe it is because the neural network was not accurate enough to be useful and that a sufficiently advanced neural network could still be useful. The addition of more historical stock trading information could also be useful for both training our agent and for evaluating.

Table 1. CAGR for all agents over different testing percentages of our data.

	0.2	**0.4**	**0.6**	**0.8**
LinUCB	28.7	38.7	43.9	52.1
Random	-16.1	-16.2	-10.1	-2.7
PrevBest	-3.2	-10.9	-16.5	-5.2

6. CONCLUSION

We have used reinforcement learning techniques to train an agent to manage a small portfolio of stocks and perform day trading on the NASDAQ stock exchange. LinUCB is a variant on UCB for solving the contextual n-armed bandit problem and uses features of the arms as well as least squares methods to learn when that arm will be performing optimally. We also optimized LinUCB for our problem by updating the reward for every arm despite the action chosen by the agent by using historical stock data. Despite the seemingly random nature of the stock market and limited data, we have managed to design a learning algorithm that profits in our trading simulation. Our LinUCB algorithm also outperforms several baseline trading agents, namely a random agent and an agent that chooses the best performing stocks from the previous day.

7. CODE

All of the code is available on github.com. The entire project was written in python and requires the numpy, scipy, and pyplot packages. The codebase includes code from pybrain, but this is not necessary to run our main LinUCB algorithm.

Link: *https://github.com/hassaanm/stock-trading*

8. REFERENCES

[1] J. Moody and M. Saffell. Learning to Trade via Direction Reinforcement. *IEEE Transactions on Neural Networks*, Vol. 12, No. 4, July 2001.

[2] L. Li, W. Chu, J. Langford, and R. Schapire. A Contextual-Bandit Approach to Personal News Article Recommendation. *Nineteenth International Conference on World Wide Web*, April 2010.

[3] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, (47): 235-256, 2002.

[4] B. Egeli, M. Ozturan, and B. Badur. Stock Market Prediction Using Artificial Neural Networks. *Hawaii International Conference on Business*, June 2003.

[5] N. Kohl and P. Stone. Policy Gradient Reinforcement Learning for Fast Quadrupedal Locomotion. *In Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2619-2624, May 2004.

[6] L. Kocsis and C. Szepesvari. Bandit Based Monte-Carlo Planning. *In Proceedings of European Conference on Machine Learning*, pages 282-293, September 2006.

[7] Y. Nevmyvaka, Y. Feng, and M. Kearns. Reinforcement Learning for Optimized Trade Execution. *In Proceedings of the 23rd International Conference on Machine Learning*, 2006.