Algorithmic Trading using Reinforcement Learning augmented with Hidden Markov Model

Simerjot Kaur (sk3391)
Stanford University

Abstract

This work presents a novel algorithmic trading system based on reinforcement learning. We formulate the trading problem as a Markov Decision Process (MDP). The formulated MDP is solved using Q-Learning. To improve the performance of Q-Learning, we augment MDP states with an estimate of current market/asset trend information which is estimated using a Hidden Markov Model. Our proposed algorithm is able to achieve an average profit of $6,379 when we invest $10,000 as compared to a baseline profit of $1,008 and an estimated oracle profit of $10,000.

1. Introduction

Algorithmic trading also called as automated trading is the process of using computers programmed to follow a defined set of instructions for placing a trade in order to generate profits at a speed and frequency that is impossible for a human trader. These defined sets of rules are based on timing, price, quantity or other mathematical model. In addition to providing more opportunities for the trader to make profit, algorithmic trading also makes markets more liquid and makes trading more systematic by ruling out any human bias in trading activities. Further, algorithms and machines easily outperform humans in analyzing the correlation in large dimensions. Since trading is a really high dimensional problem it is likely that algorithms can outperform humans in trading activities.

While there is significant scope for automated trading, current algorithms have achieved limited success in real market conditions. A key disadvantage of the current algorithmic trading methodologies is that the system uses pre-defined sets of rules to execute any trade. These predefined set of rules are coded based on years of human experience. This process of creating rules for the computer is cumbersome and hard, and limits the potential for automated trading. Further, while traders are able to adapt their strategies according to sudden changes in environment like when markets are heavily influenced in non-systematic ways by changing fundamental events, rule based automated trading lacks this important flexibility.

In this project, we are exploring the possibility of using reinforcement learning to build an AI agent that performs automated trading. Essentially, the problem can be summarized as: ***"Train an AI agent to learn an optimal trading strategy based on historical data and maximize the generated profit with minimum human intervention."*** The training strategy is considered optimal if the average profits generated by using the strategy is significantly greater as compared to various other algorithmic strategies commonly in use these days. Formally, given a dataset D, the agent is expected to learn a policy i.e. set of rules/strategies, to maximize the generated profit without any inputs from the user.

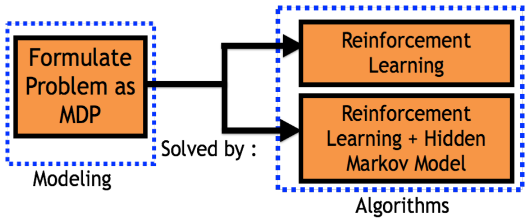


Figure 1: Proposed approach

Figure 1 above describes our overall strategy for this problem. We formulate the trading problem as a Markov Decision Process (MDP) which is then solved using Reinforcement Learning. We also propose a novel approach where we augment the states in the MDP with trend information which is extracted using Hidden Markov Model. Reinforcement learning is again used to solve this augmented MDP.

It may be noted that in the above form the state space of the problem is really large, especially given that the stock prices and profit generated are large floating point numbers. To simplify the problem to manageable level, the state space has been discretized and we have tried to solve the problem in increasing order of complexity. In our initial approaches, described in Section 3 and Section 4, we limit ourselves to single-stock portfolios only. In our first approach, described in Section 3, we discretize both the stock prices and the generated rewards to reduce the state space and simplify the problem. Subsequently, in Section 4, we modify our approach to allow it to work with discretized stock prices and real valued reward functions. In section 5, we further extend and modify our approach to allow it to work with Multi-Stock Portfolios. Our approach described in Sections 3-to-5 are based on reinforcement learning only. In section 6, we further extend our approach by augmenting the states in MDP with current market/industry trend information. Essentially, we use Hidden Markov Model (HMM) to estimate the current market/industry trend, and use this information to augment the states in MDP used in reinforcement learning. Section 7 provides a summary of our results. Section 8 further identifies the current limitations of our model. Finally, section 9 summarizes and concludes this paper and lays foundation for our future work.

2. Literature Review

Various different techniques have been implemented in literature to train AI agents to do automated trading. A large number of these implementations rely on supervised learning methods such as logistic regression, support vector machines, decision trees etc. to predict the future price of the stock, and then use a simple buy/sell strategy to implement automated trading.

In this project we are trying to follow a different approach where we are trying to learn the optimal policy/strategy directly from data, and not limiting ourselves to a simple buy/sell of stocks. We instead formulate trading as an MDP problem and use Q-learning to solve the MDP.

3. Problem Formulation with Discretized Stock Prices and Discretized Rewards – Single-Stock Portfolio

This section describes the proposed algorithm for building an AI agent to help with automated trading. As discussed above, since the state space with real valued stock prices and rewards is really huge, in the first step, the state space has been discretized for both the stock prices and rewards. The problem of choosing when to buy/sell/hold a stock is formulated as a Markov Decision Process (MDP) and is solved using Q-Learning. This section assumes that the portfolio consists of a single stock only.

MDP Formulation:

MDP can be completely defined by describing its States, Actions, Transition Probability, Rewards, and discount factor.

States: The state of the system at any point is represented by a tuple consisting of *[#of Stocks, Current Stock Price, Cash in Hand]*. The number of (#) stocks can only be represented as integers. The stock price information is taken as the daily closing price of the stock from Yahoo Finance. The stock price is discretized by rounding over to even integers, i.e. for example the stock price can only be \$20, \$22 etc. If the stock price for any day was say, \$21.523, it is rounded off to its nearest value, \$22 in this case. Cash in hand is evaluated at every time step based on the action performed.

Initial State: The system starts initially with the state *[0, initial stock price, \$10,000]*, i.e. the agent has 0 stocks and only \$10,000 as an initial investment.

Actions: At any point the AI agent can choose from three actions: *BUY, SELL, and HOLD*. The action BUY buys as many stocks as possible based on the current stock price and cash in hand. Action SELL, sells all the stocks in portfolio and adds the generated cash to cash in hand. Action HOLD, does nothing, i.e. neither sells nor buys any stock.

Transition Probability: The transition probability is chosen to be *1* always as whenever the action is SELL we are sure to SELL the stocks, and whenever the action is BUY we are sure to buy the stocks. It may be noted that the randomness in the system comes from the fact that the stock price changes just after a stock is bought or sold.

Rewards: The reward at any point is calculated as the *[current value of the portfolio - initial investment started with, at initial state]*. It may be observed that the rewards as such are floating point numbers which increase the state space significantly. To reduce the state space, in this section, the rewards are discretized in such a way so as to achieve *'+1'* if the Current value of the portfolio is greater than initial investment, i.e. the reward generated was positive and *'-1000'* otherwise. This reward function intends to force the algorithm to learn a policy that maximizes the rewards and significantly penalizes for losing money.

Discount Factor: In this project, the discount factor is assumed to be *1* always.

Solving the MDP:

The above MDP was solved using vanilla Q-Learning algorithm described in CS 221 class.

Q-learning is a form of model-free learning, meaning that an agent does not need to have any model of the environment. It only needs to know what states exist and what actions are possible in each state. The way this works is as follows: assign each state an estimated value, called a Q-value. When a state is visited and a reward is received, this is used to update the estimate of the value of that state.

The algorithm can be described as:

On each (s, a, r, s'):

$$\widehat{Q_{opt}}(s,a) \leftarrow (1-\eta)\widehat{Q_{opt}}(s,a) + \eta(r + \gamma\widehat{V_{opt}}(s'))$$

where:

$$\widehat{V_{opt}}(s') = \max_{a' \in Actions(s')} \widehat{Q_{opt}}(s',a')$$

s = current state, a = action being taken, s′ = next state, γ = discount factor, r = reward, η = exploration probability.

As Q-learning doesn't generalize to unseen states/actions, function approximation has been used which parametrizes $\widehat{Q_{opt}}$ by a weight vector and feature vector. The Q-learning algorithm with function approximation can be described as: *On each (s, a, r, s′):*

$$w \leftarrow w - \eta\left[\widehat{Q_{opt}}(s,a;w) - \left(r + \gamma \widehat{V_{opt}}(s')\right)\right]\phi(s,a)$$

where:

$\widehat{Q_{opt}}(s,a;w)$ *= prediction*

$r + \gamma \widehat{V_{opt}}(s')$ *= target*

The implied objective function is:

$$\min_{w} \sum_{(s,a,r,s')} \left(\widehat{Q_{opt}}(s,a;w) - \left(r + \gamma \widehat{V_{opt}}(s')\right)\right)^2$$

For our problem, we used the following feature vectors:

(i) Number of Stocks
(ii) Current Stock Price
(iii) Cash in Hand

The Q-learning algorithm does not specify what the agent should actually do. The agent learns a Q-function that can be used to determine an optimal action. There are two things that are useful for the agent to do:

a) exploit the knowledge that it has found for the current state 's' by doing one of the actions 'a' that maximizes Q[s,a].
b) explore in order to build a better estimate of the optimal Q-function. That is, it should select a different action from the one that it currently thinks is best.

To deploy the tradeoff between exploration and exploitation, epsilon-greedy algorithm has been used which explores with probability ε and exploits with probability 1 − ε. An exploration probability of 0.2 has been used in this project.

Dataset Used:

10-years of daily historical closing prices for various stocks were obtained from Yahoo Finance to form the dataset. For this section, the results are restricted to a portfolio consisting of Qualcomm (QCOM) stocks only.

Results:

This section discusses the performance of the above implemented Q-learning system. The Q-learning system was run on a dataset containing stock prices from last 10 years, i.e. over 2500 data points. Each run consists of the AI agent trying to maximize the reward at the end of 2500 data points. The number of trials, i.e. the number of times the AI agent was run on these 2500 data points, was set to 10,000.

The plot below (Figure 2) shows the generated reward (+1 or -1000) at the last data point of the dataset (i.e. at 2500th data-point) as a function of the number of trials. The above can be thought of as a game which ends after 2500 iterations and we are playing the game 10,000 times. The x-axis represents the i-th game being played, and the y-axis is the reward at the end of the i-th game. As can be observed, the AI agent initially gets a lot of negative rewards, but eventually it learns an optimal strategy that minimizes the loss and maximizes the profit. It may be noted that profit is +1 and loss is -1000 because of the discretization in rewards.

(Please note that the y-axis is limited to -10 in display to easily see the rewards of +1 graphically.)

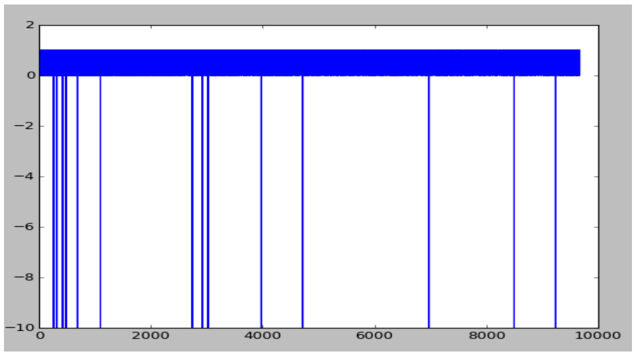


Figure 2: Discretized rewards vs number of trials using Q-Learning

To validate that the algorithm was effective in learning the optimal strategy, Monte-Carlo simulation has been performed. In this simulation, the agent is forced to choose an action at random. The plot below shows the discretized reward vs number of trials using Monte-Carlo simulation. As can be observed, the AI agent generates a lot of losses when choosing actions at random, hence validating the Q-learning algorithm for this problem.

(Please note that the y-axis is limited to -10 in display to easily see the rewards of +1 graphically.)

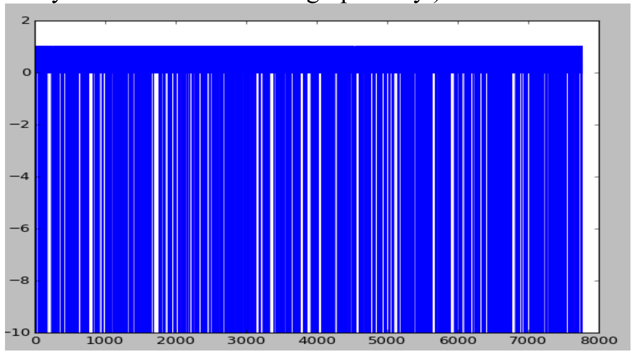


Figure 3: Discretized rewards vs number of trials using Monte-Carlo Simulation

4. Problem Formulation with Discretized Stock Prices only – Single-Stock Portfolio

To further extend the problem, in this section, the rewards are no-longer discretized. Instead the reward generated is the real value of profit or loss, i.e. it is defined as: *(current value of the portfolio - initial investment started with, at initial state)*

All the other parts of MDP definition and MDP solution are same as in Section 3 above.

Dataset Used:

Similar to section 3 above, 10-years of daily historical closing prices for various stocks were obtained from Yahoo Finance to form the dataset. For this section also, the results are restricted to a portfolio consisting of Qualcomm (QCOM) stocks only.

Results:

Similar to section 3 above, MDP was run over the data-set consisting of 2500 time steps and profit/loss at the last step, i.e. 2500th time step was reported as the final profit/loss. As before, we performed 10,000 trials, i.e. the game was run for a total of 10,000 iterations with each iteration lasting for 2500 time steps. The plot below shows the profit/loss generated at the last time step for all 10,000 iterations. The AI agent on-average obtained a profit of $739.

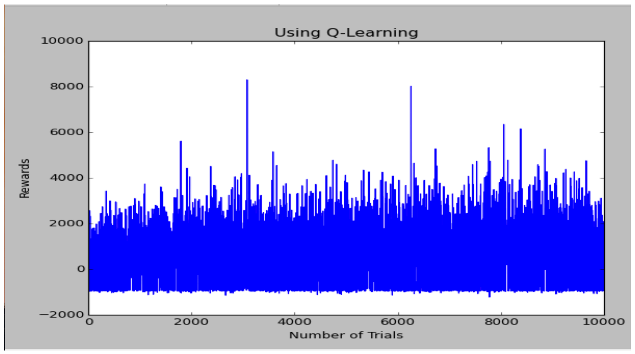


Figure 4 : Profit Generated vs Number of Trials using Q-Learning

The above graph shows that while the AI agent does generate significant profits at the end of the game, the exact value of the profit varies significantly from run-to-run. One possible reason for this could be the large number of states with continuous values of rewards and only a limited data set of 2500 time steps.

To compare the performance of the AI agent and ensure it was really learning, we compared the output to that obtained using random action at each time step. The plot below shows the performance from Monte-Carlo simulation. While there are few points with really large profit, on average, the Monte-Carlo simulation produces a profit of -$637, i.e. it produces a loss. The performance of the AI agent, while needs further refinement, is still quite good when compared to this as a baseline.

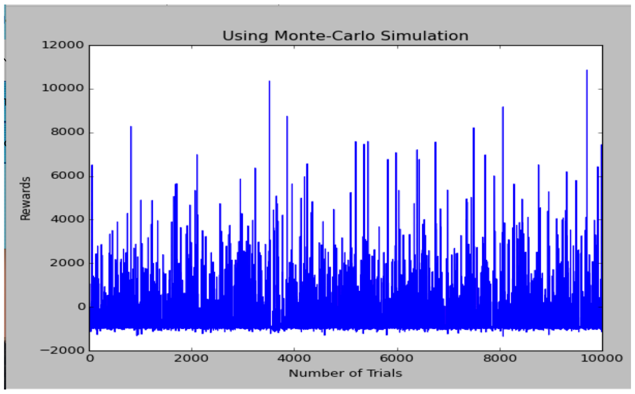


Figure 5: Profit Generated vs Number of Trials using Monte-Carlo simulation

5. Problem Formulation with Discretized Stock Prices only – Multi-Stock Portfolio

To further extend the problem, we extended the portfolio from single-stock to multi-stock. The MDP formulation for multi-stock portfolio is described below:

MDP Formulation:

States: The state of the system at any point is represented by a tuple consisting of *[(#of Stocks for each asset), (Current Stock Price for each asset), Cash in Hand]*.

The first part of the state consists of a tuple containing number of stocks for each asset. The second part of the state consists of a tuple containing the current stock price for each asset. Similar to single-stock portfolio the information is taken as the daily closing price of each asset from Yahoo Finance. The third part of the state consists of cash in hand which is evaluated at every time step based on the action performed. As for single stock portfolios, in this problem too, the stock prices for each asset have been discretized by rounding over to even integers.

Initial State: The system starts initially with the state

[(0, 0 ...), (S_1, S_2, . .), $10,000]

i.e. the agent has 0 stocks for each asset and only $10,000 as an initial investment.

Actions: At each time step, the AI agent chooses an action to either *BUY, SELL, or HOLD for each of its asset*. Therefore, the action is represented as a tuple where each entry of the tuple is either BUY, SELL or HOLD and represents an action to either BUY that asset, SELL that asset, or HOLD that asset. The length of the tuple equals the number of assets in portfolio. For example, if the portfolio consists of stocks of QCOM, YHOO, and APPL, a valid action may be [BUY, HOLD, SELL] which represents an action to BUY QCOM, HOLD YHOO and SELL APPL. It may be noted that if more than one asset has BUY as the action, we distribute the investment/cash in hand equally amongst all assets (with action BUY) and BUY as many stocks as possible based on current stock price for that asset. Action SELL, sells all the stocks of the asset and adds the generated cash to cash in hand. Action HOLD, does nothing, i.e. neither sells nor buys any stock of the asset. It may be noted that the number of possible actions grow exponentially as the number of assets in the portfolio increases. In-fact, the number of actions is exactly $3^{\#\text{ of assets in portfolio}}$. This increases the state-action pair space significantly.

Transition Probability: The transition probability is chosen to be *1* always as whenever the action is SELL we are sure to SELL the stocks, and whenever the action is BUY we are sure to buy the stocks irrespective of the asset being bought or sold.

Rewards: The reward at any point is calculated as:
Current Value of the Portfolio - Initial Investment started with, at initial state

where

$$Current\ Value\ of\ the\ Portfolio = \sum_{i \in assets\ in\ portfolio} (\#\ of\ stocks\ of\ asset\ i * Current\ Stock\ Price\ of\ asset\ i)$$

Discount Factor: In this project, the discount factor is assumed to be *1* always.

All the other parts of MDP solution are same as in Section 4 above.

Dataset Used:

Similar to section 3 and 4 above, 10-years of daily historical closing prices for various stocks were obtained from Yahoo Finance to form the dataset. For this section, the results are restricted to a portfolio consisting of 3 stocks *[Qualcomm (QCOM), Microsoft (MSFT), Yahoo (YHOO)].*

Results:

Similar to section 3 and 4 above, MDP was run over the data-set consisting of 2500 time steps and profit/loss at the last step, i.e. 2500th time step was reported as the final profit/loss. As the state space has increased, we increased the number of trials to obtain convergence. In this section, 30,000 trials were performed, i.e. the game was run for a total of 30,000 trials with each iteration lasting for 2500 time steps. The plot below shows the profit/loss generated at the last time step for all 30,000 trials.

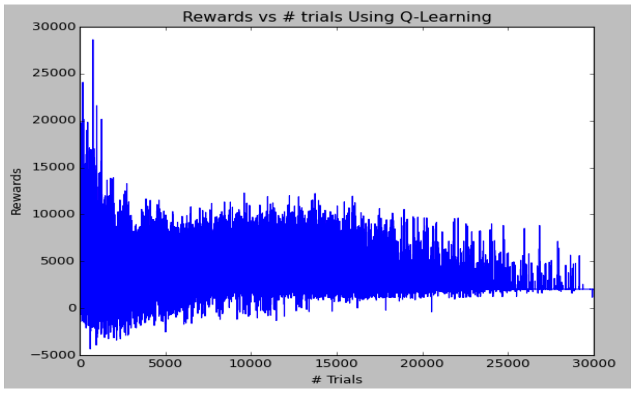


Figure 6: Profit Generated vs Number of Trials using Q-Learning

The above graph shows the training process of the AI agent which is trying to learn to maximize the reward. It may be noted from the graph above that the AI agent generates profits that are very volatile and have low average for the first few iterations. Then, as it continues to learn the optimal strategy, the variance decreases and the average profit increases. In the above graph, the AI agent made an average profit of $3,466 over 30,000 trials which although is significantly less than the oracle profit of $10,000 but performs much better than the baseline described below. It may be noted that we assume an oracle can make a profit of $10,000 based on a general observation that on average experienced human traders can almost double the investment within a span of 5 years.

Similar to sections 3 and 4, Monte-Carlo Simulation was considered as a baseline. The plot below shows the performance from Monte-Carlo simulation. While there are few points with really large profit, on average, the Monte-Carlo simulation produces a profit of $579 which is significantly less than that from Q-learning.

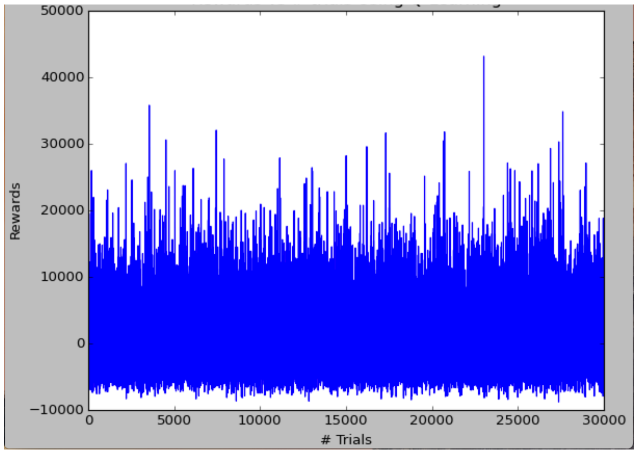


Figure 7: Profit Generated vs Number of Trials using Monte-Carlo simulation

6. Algorithmic Trading using Reinforcement Learning augmented with Hidden Markov Model (HMM) – Multi-Stock Portfolio

When a human trader makes a trading decision, in addition to the stock price, his decision is also based on his view of the market/industry's trend, i.e. he has a notion whether the market/industry is in a bearish or bullish state. Knowing the current trend has an obvious benefit as the trader, he formulates a trading strategy based on this information and is able to profit from the upward movement of the stock and avoid the downfall of a stock. In this section, we are trying to provide such trend information to our AI agent also, to help it perform similar to a human agent. Essentially, we are trying to capture the industry/market/asset's current trend and use this trend information as part of the state definition in our MDP formulation. We expect, the knowledge of the assets current trend will help our AI agent to take better decisions as to when to BUY, SELL or HOLD and in turn maximize the profit, just in the same way, as this information helps a human trader improve his performance.

The practical implementation of the above approach entails two steps: (A) Finding the current trend information (B) Augmenting the trend information into our MDP formulation. We use a Hidden Markov Model (HMM) to estimate the current trend information. The output from the Hidden Markov Model is then incorporated into our MDP. The details for finding the trend information using HMM is described in section 6A below. Section 6B then provides details of how we augmented our MDP with the output from our HMM Model.

6.A Using HMM to find Trend Information

To identify the trend information, we model the dynamics of the stock return as a Hidden Markov Model. We assume the returns of the stock are normally distributed with mean and variance depending on the hidden random variable which represents the stock trend. Our Hidden Markov Model is shown in the figure below:

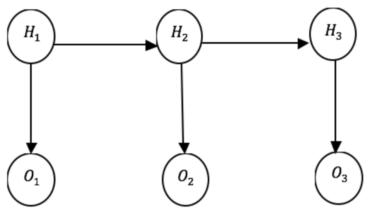


Figure 8: Hidden Markov Model where $\boldsymbol{H_1}, \boldsymbol{H_2}$, .. are hidden random variables representing stock trend and $\boldsymbol{O_1}, \boldsymbol{O_2}$, .. are observed variables representing returns of the asset

The states $\boldsymbol{H_1}, \boldsymbol{H_2}, \boldsymbol{H_3}$ are the hidden states that represent the current trend, i.e. whether the market is bearish, bullish or neutral. The observed values O_1, O_2, O_3 are the real observed returns. The model parameters are estimated using Expectation-Maximization algorithm and subsequently the hidden state i.e. the current stock trend is estimated using Viterbi algorithm. We describe our formulation more formally below as:

Let Z_t be a Markov chain that takes values from state space $S = \{s_1, s_2, \ldots, s_n\}$ where S is a set of all possible trends in an asset. For instance, S can be stated as {bullish, bearish, neutral}.
Let the initial distribution of returns of the stock be denoted by π and the transition probability of $Z_t$be P:
$\pi_i = \mathbb{P}[Z_0 = s_i]$, $P_{ij} = \mathbb{P}[Z_{t+1} = s_j | Z_t = s_i]$

Let's assume that returns of the stock follow a Gaussian distribution with mean μ_i and variance σ_i^2 when Z_t is in state s_i. Let S_t be the stock price at t and returns be $R_t = \left(\frac{S_{t+1}}{S_t}\right) - 1$.
The conditional density function of R_t is:

$$f_{R_t|Z_t=s_i}(r) = \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{(r-\mu_i)^2}{2\sigma_i^2}}$$

The main assumption of this model is that Z_t is unobserved. So the main goal is to find most likely sequence Z_t by computing the conditional distribution of Z_t given the observed sequence R_t.

Now *Expectation-Maximization Algorithm* is used to estimate the set of model parameters i.e. initial distribution π and transition matrix P of unobserved Z_t.

Expectation Maximization Algorithm [1]

The EM algorithm computes the Maximum Likelihood (ML) estimate in the presence of hidden data. In ML estimation, the model parameters need to be estimated for which the observed data are most likely. Each iteration of the EM algorithm consists of two steps:

- ➔ **The E-step**: In the expectation, or E-step, the hidden data is estimated given the observed data and current estimate of the model parameters. This is achieved using the conditional expectation.
- ➔ **The M-step**: In the maximization, or M-step, the likelihood function is maximized under the assumption that the hidden data is known. The estimate of the hidden data from the E-step is used in place of the actual hidden data.

Let **X** be random vector and θ be the parameter that needs to be estimated, then the Maximum Log-Likelihood function is defined as

$$L(\theta) = \ln \mathcal{P}(X|\theta)$$

Since *ln(x)* is a strictly increasing function, the value of θ which maximizes $\mathcal{P}(X|\theta)$ also maximizes $L(\theta)$. The EM algorithm is an iterative procedure for maximizing $L(\theta)$.

Assume that after the n^{th} iteration the current estimate for θ is given by θ_n. Since the objective is to maximize $L(\theta)$, an updated estimate of θ needs to be computed such that,

$$L(\theta) > L(\theta_n)$$

Equivalently the difference needs to be maximized,

$$L(\theta) - L(\theta_n) = \ln \mathcal{P}(X|\theta) - \ln \mathcal{P}(X|\theta_n)$$

Let Z be the hidden random vector and z be the realization. Then the total probability $\mathcal{P}(X|\theta)$ may be written in terms of the hidden variables z as,

$$\mathcal{P}(X|\theta) = \sum_z \mathcal{P}(X|z,\theta)\mathcal{P}(z|\theta)$$

Placing the above value in difference equation:

$$L(\theta) - L(\theta_n) = \ln \sum_z \mathcal{P}(X|z,\theta)\mathcal{P}(z|\theta) - \ln \mathcal{P}(X|\theta_n)$$

$$L(\theta) - L(\theta_n) \geq \sum_z \mathcal{P}(z|X,\theta_n) \ln \left(\frac{(X|z,\theta)\mathcal{P}(z|\theta)}{\mathcal{P}(z|X,\theta_n)} \right) - ln\mathcal{P}(X|\theta_n)$$

As $\sum_z \mathcal{P}(X|\theta_n)$=1,

$$L(\theta) - L(\theta_n) \geq \sum_z \mathcal{P}(z|X,\theta_n) \ln \left(\frac{(X|z,\theta)\mathcal{P}(z|\theta)}{\mathcal{P}(z|X,\theta_n)} \right) \triangleq \Delta(\theta|\theta_n)$$

Equivalently, $L(\theta) \geq L(\theta_n) + \Delta(\theta|\theta_n)$

Let, $l(\theta|\theta_n) \triangleq L(\theta_n) + \Delta(\theta|\theta_n)$ so $\Longrightarrow$ $L(\theta) \geq l(\theta|\theta_n)$

If $\theta = \theta_n$,

$$\begin{aligned} l(\theta|\theta_n) &= L(\theta_n) + \Delta(\theta_n|\theta_n) \\ &= L(\theta_n) + \textstyle\sum_z \mathcal{P}(z|X,\theta_n) ln1 \\ &= L(\theta_n) \end{aligned}$$

This means that $l(\theta|\theta_n) = L(\theta_n)$ when $\theta = \theta_n$. Therefore, any θ which increases $l(\theta|\theta_n)$ in turn increase the $L(\theta)$. In order to achieve the greatest possible increase in the value of $L(\theta)$, the EM algorithm calls for selecting θ such that $l(\theta|\theta_n)$ is maximized. Let this updated value be denoted as θ_{n+1}. This process is illustrated in figure below:

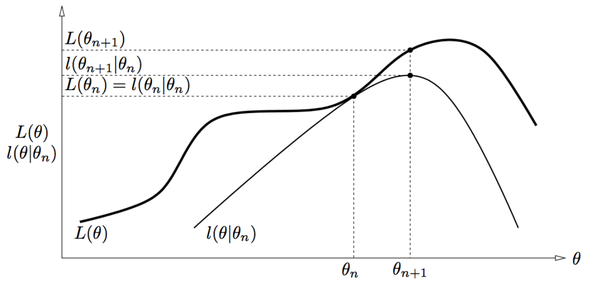


Figure 9: Graphical interpretation of a single iteration of the EM algorithm [1]

Formally,

$$\theta_{n+1} = \arg\max_\theta \{l(\theta|\theta_n)\}$$

Placing the value of $l(\theta|\theta_n)$ and dropping terms which are constant with respect to θ,

$$\begin{aligned} \theta_{n+1} &= \arg\max_\theta \{\sum_z \mathcal{P}(z|X,\theta_n) ln\, \mathcal{P}(X,z|\theta)\} \\ &= \arg\max_\theta \{E_{Z|X,\theta_n}\{ln\mathcal{P}(X,z|\theta)\}\} \end{aligned}$$

Here the expectation and maximization steps are apparent. The EM algorithm thus consists of iterating the:

➔ **E-step**: Determine the conditional expectation $E_{Z|X,\theta_n}\{ln\mathcal{P}(X,z|\theta)\}$

➔ **M-step**: Maximize this expression with respect to θ.

Now since the model parameters are estimated, the next step is to find the most likely sample path of unobservable Markov chain, which is done using *Viterbi Algorithm*.

Viterbi Algorithm [3]

The Viterbi algorithm is a dynamic programming algorithm for finding the most likely sequence of hidden states, called the Viterbi path, that results in a sequence of observed events.

Given,

- observation space $O = \{o_1, o_2, \dots, o_N\}$,
- the state space $S = \{s_1, s_2, \dots, s_K\}$,
- a sequence of observations $Y = \{y_1, y, \dots, y_T\}$,
- transition matrix A of size $K \cdot K$ such that A_{ij} stores the transition probability of transiting from state s_i to state s_j,
- emission matrix B of size $K \cdot N$ such that B_{ij} stores the probability of observing o_j from state s_i,
- an array of initial probabilities π of size K such π_i stores the probability that $x_1 == s_i$.

It can be said that path $X = \{x_1, x_2, \dots, x_T\}$ is a sequence of states that generate the observations $Y = \{y_1, y, \dots, y_T\}$. Hence $X = \{x_1, x_2, \dots, x_T\}$ is the Viterbi path.

In this dynamic programming problem, 2-dimensional table T_1, T_2 of size $K \cdot T$ is constructed. Each element $T_1[i,j]$ of T_1 stores the probability of the most likely path so far $\hat{X} = \{\widehat{x_1}, \widehat{x_2}, \dots, \widehat{x_j}\}$ with $\widehat{x_j} = s_i$ that generates $Y = \{y_1, y, \dots, y_j\}$. Each element $T_2[i,j]$ of T_2 stores $\widehat{x_{j-1}}$ of the most likely path so far $\hat{X} = \{\widehat{x_1}, \widehat{x_2}, \dots, \widehat{x_{j-1}}, \widehat{x_j}\}$ for $\forall j, 2 \leq j \leq T$.

The entries of two tables $T_1[i,j]$, $T_2[i,j]$ are filled by increasing order of $K \cdot j + i$.

$$T_1[i,j] = \max_k (T_1[k,j-1] \cdot A_{ki}) B_{iy_i}$$

$$T_2[i,j] = arg \max_k (T_1[k,j-1] \cdot A_{ki})$$

The Viterbi algorithm can be written as follows [3]:

```
function VITERBI( $O, S, \pi, Y, A, B$ ) : $X$
    for each state $s_i$ do
        $T_1[i,1] \leftarrow \pi_i \cdot B_{iy_1}$
        $T_2[i,1] \leftarrow 0$
    end for
    for $i \leftarrow 2,3,...,T$ do
        for each state $s_j$ do
            $T_1[j,i] \leftarrow \max_k (T_1[k,i-1] \cdot A_{kj})$
            $T_1[j,i] \leftarrow (T_1[j,i] \cdot B_{jy_i})$
            $T_2[j,i] \leftarrow \arg\max_k (T_1[k,i-1] \cdot A_{kj})$
        end for
    end for
    $z_T \leftarrow \arg\max_k (T_1[k,T])$
    $x_T \leftarrow s_{z_T}$
    for $i \leftarrow T,T-1,...,2$ do
        $z_{i-1} \leftarrow T_2[z_i,i]$
        $x_{i-1} \leftarrow s_{z_{i-1}}$
    end for
    return $X$
end function
```

After reviewing the algorithms, the data is fitted to data to HMM using EM Algorithm for a given number of states. The EM algorithm outputs the estimated parameters and then the Viterbi Algorithm is used to obtain the most likely hidden path.

Results:

To verify our implementation of the EM Algorithm, Viterbi Algorithm, and our pipeline to solve the HMM model, we initially test our pipeline on a small data-set. The data-set is generated by sampling 150 samples from two Gaussian distributions, one with mean 0.1 and standard deviation 0.15, and the other with mean of -0.1 and standard deviation 0.2. Essentially the first and last fifty samples of the data-set are sampled from one Gaussian distribution, and the middle fifty samples are sampled from the other Gaussian distribution. We use this test data-set as an input to our HMM pipeline and compare the state information returned from our pipeline with ground truth. The results from this test are shown in Figure 10 below. The bottom most plot in figure 10 shows the samples, which look rather random, although there is a hidden state information in the data. The top-most plot in Figure 10 shows the output of our HMM pipeline. As is evident from the plot our HMM pipeline predicts with a high degree of confidence that the first and last fifty samples come from one distribution and the middle fifty samples come from another distribution, which matches our expectation.

Figure 11 and 12 below show the performance of HMM algorithm in identifying the trend information on real stock data. We take 10-years of daily historical closing prices for S&P 500 index from Yahoo Finance to form the dataset. Our HMM pipeline is applied to the dataset, and Figures 11 and 12 show the result when we run our algorithm with number of hidden states as 2 (Figure 11) and 3 (Figure 12) respectively. As can be observed from the figures, the HMM algorithm does a good job in identifying the general trend. For example, in Figure 11, it can be observed that whenever there is a general trend of increasing stock prices the state of the system is returned as being ‘1’ and whenever there is general downward trend the state of the system is returned as ‘0’. Figure 12 tries to identify 3 states in the dataset.

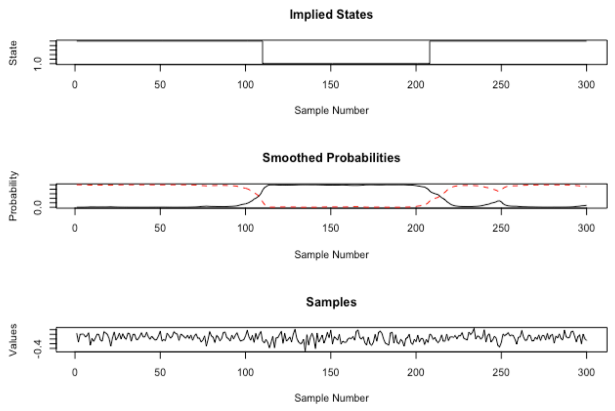


Figure 10: State Distribution of data generated by the concatenation of 3 random variables

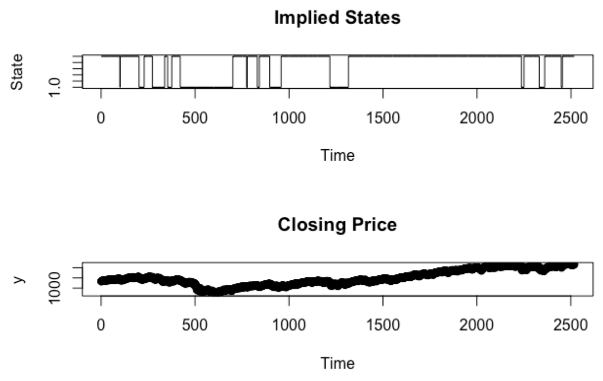


Figure 11: State Distribution of S&P 500 when # of possible states is 2

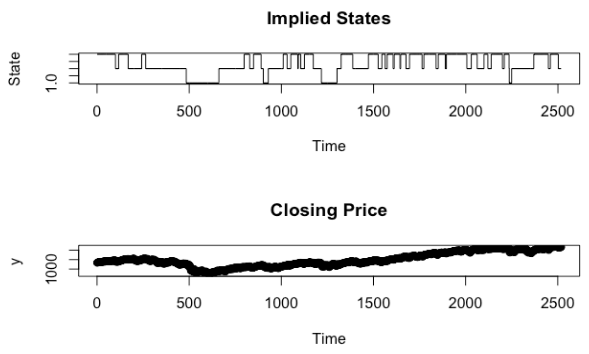
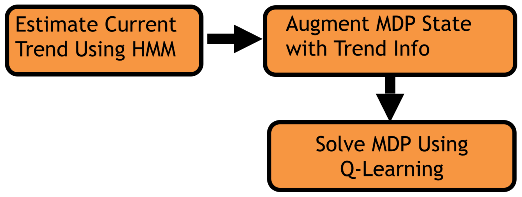


Figure 12: State Distribution of S&P 500 when # of possible states is 3

6.B Augmenting the trend information into our MDP formulation

The above section described how we can identify current trend in stock data using Hidden Markov Model. To provide this information to our AI agent, we augmented the state of the system with this information and the new state implemented is *[(#of Stocks for each asset), (Current Stock Price for each asset), Cash in Hand, (trend of each asset)]*. All other aspects of the MDP definition are same as before.

The MDP was still solved using Q-learning approach as in section 5, with the only difference being that the trend of asset is also being used as feature in our function approximation based Q-Learning. The overall pipeline of our solution can essentially be summarized below as:


Estimate Current Trend Using HMM
Augment MDP State with Trend Info
Solve MDP Using Q-Learning

DataSet:

Similar to section 5 above, 10-years of daily historical closing prices for various stocks were obtained from Yahoo Finance to form the dataset. For this section also, the results are restricted to a portfolio consisting of 3 stocks *[Qualcomm (QCOM), Microsoft (MSFT), Yahoo (YHOO)]*.

Results:

Similar to section 5 above, MDP was run over the data-set consisting of 2500 time steps and profit/loss at the last step, i.e. 2500th time step was reported as the final profit/loss. In this section also, 30,000 trials were performed, i.e. the game was run for a total of 30,000 trials with each iteration lasting for 2500 time steps. The plot below shows the profit/loss generated at the last time step for all 30,000 trials.

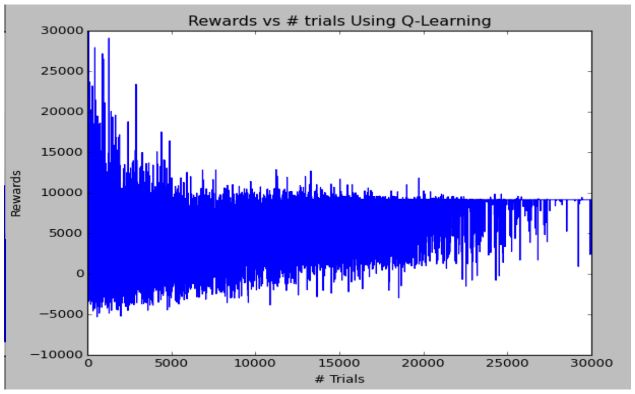


Figure 13: Profit Generated vs Number of Trials using Q-Learning

The plot above shows the training process of the AI agent which is trying to learn to maximize the reward. It may be noted from the plot above that the AI agent generates profits that are less volatile than the profits generated in Section 5. In the above graph, the AI agent made an average profit of $6,379 over 30,000 trials which is closer to the oracle profit of $10,000 and performs much better than the baseline described below.

Similar to sections 5, Monte-Carlo Simulation was considered as a baseline. The plot below shows the performance from Monte-Carlo simulation. While there are few points with really large profit, on average, the Monte-Carlo simulation produces a profit of $1008 which is significantly less than that from Q-learning.

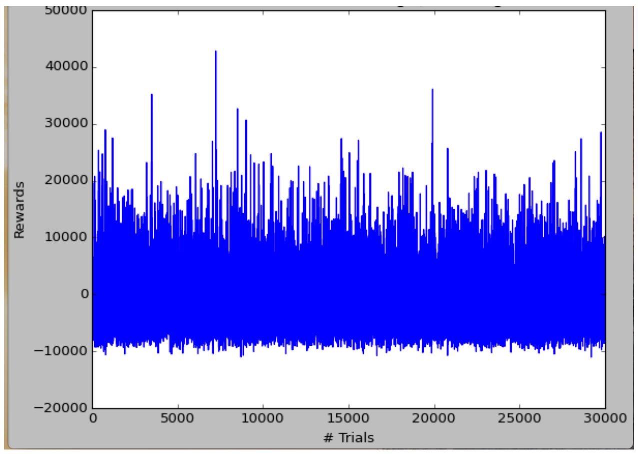


Figure 14: Profit Generated vs Number of Trials using Monte-Carlo simulation

7. Summary of Results

The table below summarizes the mean profit achieved from 30,000 trials using the various proposed algorithms. While all algorithms are better than our baseline Monte-Carlo simulations (MCS), as expected, the best performance is achieved when we use Q-Learning to solve MDP augmented with trend from HMM. The performance for Q-Learning on MDP augmented with trend information has a much higher mean value, and a much lower variance. This is expected because adding the trend information to our state helps the AI agent make much more informed decisions.

Furthermore, it may be noted that all algorithms give us a higher profit when the portfolio consists of multiple stocks rather than single stock. This is also expected because having multiple stocks allow us to diversify the portfolio hence making us risk-averse to variations in highly-volatile stock market, and also provide more opportunity to make optimal decisions.

		Mean Profit from 30,000 trials	Standard Deviation of Profit
Single-Stock Portfolio	***Q-Learning***	$789	$979
	MCS	-$637	N/A
Multi-Stock Portfolio	***Q-Learning***	3,466	$3,333
	MCS	$579	N/A
Multi-Stock Portfolio augmented with HMM	***Q-Learning***	6,379	$1,454
	MCS	$1,008	N/A
Oracle	-	10,000	-

Table 1: Summary of Results from various proposed algorithms

8. Challenges and Error Analysis

We observed that while our proposed approach works well in a lot of scenarios, there are still significant challenges that need to be solved to build a truly autonomous trading system. Some of the major challenges/limitations in the current approach are:

a. Large State Space: One of the biggest challenges is the exponential increase in the state-action pair space as the number of assets in our portfolio increases. Since our state definition consists of *(number of stocks for each asset, stock prices of each asset, cash in hand, trend path followed by each asset)*, as the number of assets increase, the action space increases exponentially. This increases the run-time of our system considerably, and limits the performance of the system. This is a major limitation of this project. In case of multi-stock portfolios, the state space increases exponentially with the increase in the number of assets taken in a portfolio.

b. Discretization of Stocks: The discretization of stocks i.e. rounding off the stock price in order to reduce state space, varies from asset to asset. For instance, in case of Qualcomm (QCOM) stocks, the stock price is rounded over to even integers while in case of S&P 500 index, as the value of index is high, the closing price needs to be rounded off to nearest 10th digit. Hence there is no one size fit all strategy that we could apply to all assets in the portfolio.

c. Limitation of HMM: Our proposed method of estimation of trend information using HMM provides good results for many stocks, but it fails to perform well under certain constrained situations such as in events where there is sudden change in stock price because of stock split event, for instance Apple (AAPL) stock.

9. Conclusion and Future Work

In this paper, we implemented reinforcement learning algorithm to train an AI agent to develop an optimal strategy to perform automated trading. In the first section, the AI agent was successful in learning the optimal policy that maximizes the profit when the reward space was discretized on single-stock portfolio. In the next section, the agent also achieved good results when the rewards were not discretized on single-stock portfolio.

We then further extend our implementation to multi-stock portfolio where the AI agent successfully learns a trading strategy to maximize the profits. Although, the agent made a decent average profit of $3466 but still, it was significantly lower than the oracle. In order to improve the learning efficiency of the agent, Hidden Markov Model was used to augment MDP with trend information. The agent performed significantly better and made an average profit of $6379 which was much closer to the oracle.

However, although the average profit generated in 30,000 trials is closer to Oracle, we still see a significant run-to-run variation. The run-to-run variations could be because of limited data set, i.e. we only have stock data for 2500 time steps.

Finally, our current work can be extended in various ways. For instance, while creating the HMM, we made an assumption that the returns are Gaussian. This is a simplistic assumption and we can extend our HMM approach to work with other common distributions that might explain the generated returns more closely. Further, we could improve the trend estimation by augmenting data from other sources such as news feed, twitter accounts etc. Also, given the large state space, it might be worthwhile to try to use policy gradient methods which directly try to estimate the best policy without really generating the Q-Value. Deep Q-Learning and Deep Reinforcement learning methods have recently been applied to many different problems and have been shown to achieve very good performance. Based on the initial results from this report, we feel it could be a good idea to try to implement Deep-Reinforcement learning methods for this problem too.

10. References

[1] Sean Borman, The Expectation Maximization Algorithm – A Short Tutorial, July 2004.
[2] B.H.Juang and L.R. Rabiner, An Introduction to Hidden Markov Models, IEEE, January1986.
[3] Viterbi Algorithm, Wikipedia.
[4] Markov Decision Process, Lecture Notes
[5] Q-Learning and Epsilon Greedy Algorithm, Lecture Notes