Machine Learning Engineer Nanodegree

Capstone Project

Erik Flogvall
December 10, 2017

I. Definition

Project Overview

In this project have I created a trading agent that learns when to sell or buy a stock. The trader uses historical data derived from daily closing prices of stocks for learning how to trade a given stock to make the highest possible return.

The trading agent uses reinforcement learning to determine optimal trading policies. The iterative process of reinforcement learning is useful when the input data is added with every day. An automatic trading robot can be very useful if it can learn optimal policies for trading stocks. This can therefore be a very positive gain for any funds, stock brokers and individual currently not using reinforcement learning for this problem. The reinforcement learning has been in several scientific studies to make optimized and automatized traders [1][2].

The dataset used for this is the Kaggle S&P 500 stock data collected by Cam Nugent under a public license [3].

Problem Statement

The main problem to be solved is to set up and train a reinforcement learner to output optimal actions for buying and selling stocks. The output of the problem will be what action of buying, selling or doing nothing is most beneficial for increasing the return from trading any given stock.

The input data to this will the closing price for every trading day. A state for the stock will be created from the latest closing price and other values derived from historical closing prices (relative to the latest trading day). The state also needs to include if the stock is currently owned. Discretization will be necessary to create a discrete state as the closing price is a continuous input. The size of the state should be kept so small that it is possible to examine all states while training. The state must also carry enough information so that an optimal action can be selected. This will be one of the more challenging parts of this problem.

The proposed solution for the problem is to use Q-learning for learning an optimal trading strategy. The daily return for a stock should be the reward in the Q-learning algorithm. The possible actions in the Q-table should be buying, selling or doing nothing. When the Q-learning algorithm is implemented should it be able to output what action is the best for the given state of a given stock.

Metrics

To evaluate how well the model is working will the return over the trading period be computed. The total return over a trading period is a good metric to evalute a model as this would be the profit (or loss) when trading. This makes it good as it is directly linked to the purpose of the trading and it is very easy to understand.

The total return over the trading period is defined as:

$$R_{TOTAL} = \sum R_i$$

Where R_i is the return for the i-th trade session, i.e. when stocks is bough hold for a given period and then sold. R_i is defined as:

$$R_i = S_i - B_i$$

Where S_i is income from selling the stocks, B_i is the cost of buying the stocks. The cost of trading the stocks is neglected to simply the problem.

To implement this metric can a ledger be kept on the balance for trading each stock. The ledger is initialized with a balance of zero. When a stock is purchased the current price of the stock will be removed from the balance. When the stock later is sold will the current price of the stock be added to the balance. This means that the stock ledger will contain the profit after the trading period has been done and all stocks have been sold.

II. Analysis

Data Exploration

The dataset to be used in this capstone project is the Kaggle S&P 500 stock data collected by Cam Nugent under a public license [3]. The dataset contains stock prices in US dollar for all S&P 500 stocks for each trading day over five years. Each stock has the data stored in a separate text file.

The included data in each entry is the price at the beginning of the trading day (“Open”), the highest price for each day (“High”), the lowest price for each day (“Low”), the closing price at the end of trading (“Close”), the number of traded shares (“Volume”) and the stocks ticker name (“Name”). For this problem will only the closing price be used to build the state for the reinforcement learning. An example with some data entries for the Apple stock (AAPL):

```
Date,Open,High,Low,Close,Volume,Name
2012-08-13,89.06,90.0,89.04,90.0,69707463,AAPL
2012-08-14,90.27,91.23,90.03,90.24,85041824,AAPL
2012-08-15,90.19,90.57,89.68,90.12,64377278,AAPL
2012-08-16,90.17,90.97,90.07,90.91,63694204,AAPL
```

The data included in the dataset should be enough for making a discrete state for a given stock on a given date. A reward function can also be constructed by computing price changes. The size of the data set should be enough for both training and testing as the time series is relatively long (5 years) and there is data for 500 stocks.

Two main issues with the data has been observed. The first is that the first and last date for which the stock traded are different depending on the stock. The cause of this is that the S&P 500 index have contained different stocks over the five year period. Some stocks can also have their ticker name changed because of mergers. To mitigate this problem will stocks only be traded by the agent for the time period when the stock was tradeable. It would be very unrealistic to trade a non tradeable stock.

Another issue is that some stocks have missing data entries even though the stock was listed for trading. The reason for this is probably that trading was suspended for some reason. This will cause problems when using rolling stats or computing derivatives of the data.

Exploratory Visualization

An example of the data is shown in the figure below, where the closing price of the Apple stock is plotted over time. Both day to day variations and long run changes in price can be seen in the figure. The day to day variations seems somewhat random but an overall trend can be seen over a longer period. The problem to be solved is if any future changes in price can be predicted.

Algorithms and Techniques

Q-learning

The reinforcement learning algorithm chosen for this problem is Q-learning. The reason for selecting Q-learning is that it is model-free and relatively easy to understand when reading the Q-table. The algorithm is quite simple and any modifications are therefore easier made. The main problem with implementing Q-learning is to select how create its states. Multiple method for making states is therefore evaluated.

The data set contains all prices for S&P 500 stocks over a period of five years. The data has been split in to two smaller data sets. The first data set is the first 70% of all trading days with the second data set being the last 30% of the trading days. This allows for simulating training a Q-learner on historical data by using the first data set. The second data set can then be used for simulating a implementation of a pre-trained Q-learner.

Q-learning works by having a agent that executes predefined actions for each possible state. All possible states and actions are stored in Q-table. For each state s_t of the Q-table are all of the states actions a_t listed. The agent selects the highest valued action or chooses a random action if the there are multiple actions with the same highest value. When the selected action is executed will the agent receive a reward r_t . This reward is then used for updating the Q-table for the current action and selected state. The update rule is:

$$Q(s_t, a_t) \leftarrow (1-\alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max_a Q(s_t, a))$$

Where a is the learning rate and γ is the discount rate for future rewards. The Q-table is initialized with zero values for each action.

The process for Q-learning is:

1. Get the current state of the stock
2. Read the Q-table to select the best action
3. Get the reward from executing the action
4. Update the Q-table with the update rule

Random exploration is used for training on the simulated historical data. This allows the algorithm to explore the Q-table faster by randomly selecting actions. The randomness is controlled by the random exploration factor ε , which is decreased slightly after every Q-learner decision. The decrease is controlled by the exponential decay function $\varepsilon = \varepsilon_0 \cdot e^{-\theta t}$, where ε_0 is the initial random exploration factor, θ is the decay factor and t is the number of times an action (buy, sell or do nothing) has been decided for a stock. When simulating implementation on the second data set is the random exploration disabled by setting a constant value of zero .

Q-learning variables	**Used value**
Learning rate	0.05
Future reward discount rate	0.95
initial random exploration factor	1.0
decay factor for the random exploration factor	1.5e-5

Dyna

Dyna is used for making the Q-learning process more effective. Dyna works by randomly selecting a state and action and running an update rule iteration without any future rewards. Dyna requires that all

transition between states and expected rewards of all states actions are logged. Therefore has a T-table which counts transitions between states been implemented. A table with expected rewards for each state and action is also used. This allows the Dyna algorithm to run many iterations to effectively converge the Q-table for every trading day. In this project is Dyna run 1000 times between every trading day.

Discretization

The closing price and other input features derived from it are continuous. The states used for describing the stocks for each day in the Q-learning algorithm are discrete. The continuous features must therefore be discretized. The discretization method used in this project is sorting he values into bin that contain values within a given range. The discretized value will then be the bin number that identifies the bin. The bins are constructed by adding all data for a given feature of the first data set (used for simulating historical data) into one list. This list is then sorted from smallest to largest value and split into N (number of bins) equally sized groups. The limits for these groups will then be used to decide what bin a value will be sorted into.

Features

Multiple features are derived from the closing price. These are the daily return, the rolling mean of the price, the rolling standard deviation, the yesterday's daily return and the daily return for two days ago. The means that there six features available for making states with.

Benchmark

The benchmark model is buying the stock at the start of the evaluation period and selling it at the end. This is a non-active trading strategy commonly used for long term investment. It does not rely on daily stock information and should therefore be good for evaluating if the Q-learning trader can gain extra value by using more information.

III. Methodology

Data Preprocessing

Solution for missing data entries

Missing data entries in the stocks needs to be fixed so that features that requires continuing data for every trading day can be used. The solution for this is forward filling from the last known entry. This will allow the algorithm to get a return from the next day and allow rolling statistics to be used for making features for the state. Days that the stock exchange was open and trading will be identified from the data of all stocks. If any stock was traded during a day it will mean that the stock exchange was open during that day. For example will a given data series be forward filled like this:

Original series

Date,Open,High,Low,Close,Volume
2012-08-13,89.00,90.0,89.00,90.0,680000

Forward filled series

Date,Open,High,Low,Close,Volume
2012-08-13,89.00,90.0,89.00,90.0,680000

2012-08-15 90.00,91.0,90.00,91.0,690000

2012-08-14,89.00,90.0,89.00,90.0,680000
2012-08-15 90.00,91.0,90.00,91.0,690000

New features derived from the closing price

Multiple new features are derived from the closing price. These features are made from rollings stats and price differences. A mean and standard deviation is computed by rolling statistics with a rolling window of 20 day, This means that a average and standard deviation is taken for the past 20 days for every day in the data that is equal or later than the 20^{th} day in the data set. The daily return is computed as the change of the closing price in a day compared to the previous day. The closing price itself is not used as a feature. The reason for this is that stocks with high or low price levels will be separated in different states if the price is used for making states.

The five features that are available after preprocessing are:

- The 20-day rolling mean of the closing price
- The 20-day rolling standard deviation of the closing price
- The daily return
- Yesterday's daily return
- The daily return for the day before yesterday

All features are normalized by the closing price.

Implementation

Data
The data is stored in a sub folder to the project folder that contains the scripts. The data is formatted as comma separated values with one text file for each individual stock. These files contains dates and stock prices for every date.

Files structure
The Python code is divided into three separate files:

- A Jupyter notebook - "MLND Capstone Erik Flogvall.ipynb"
  - Runs the code
  - Output results
- A code file to preprocesses the data- "prepdata.py"
  - Loads the data
  - Computes new features from the closing price
  - Divides the data into:
    - A training only set for simulating training on historical data
    - A training & evaluation set for simulation implementing the trained Q-learner on new data
- A code file to implement a Python class Q-learning - "q_trader.py"
  - Contains a class named QTrader that is a Q-learning agent
  - Discretizes the prepossessed data
  - Initializes and performs Q-learning
  - Performs Dyna simulations to improve the Q-learning
  - Keeps a record (the stock ledger) of profits from trading each stock.

MLND Capstone Erik Flogvall.ipynb
The project is run from this jupyter notebook. It uses the os, Numpy, Pandas and Matplotlib Python libraries. The prepare_data function is imported from prepdata.py and the QTrader class from q_trader.py. First it loads and preprocesses the data with the prepare_data function.

```
split_ratio = 0.7

window_size = 20

csv_dir = 'individual_stocks_5yr'

data, trading_days, dates, stocks = prepare_data(csv_dir, split_ratio, window_size)
```

split_ratio controls what ratio of the trading days should be used for simulating training on historical data. In this project it is set the 0.7 meaning that the first 70% of all trading days in the data set is used for the training only set. The final 30% is used for simulating the implementation with training & evaluation set.

window_size controls the rolling window used for computing the rolling mean and standard deviation from the closing price. In this project it is set to 20 meaning that the rolling window is 20 trading days.
The states are set from a dictionary containing the features selected for the state as keys. The values are the number of discrete steps for the given feature. The used state settings are described under the Refinement section

An example of a state setting:

```
state_setting = {'Price': 10, 'Daily_return': 10}
```

The Q-learner class is initialized using a dictionary called qt_input containing the preprocessed data and the state settings.

Some constants for the Q-learning are also set when the Q-learner is initialized. These are:

- **epsilon**, the random exploration factor initialized to 1.0
- **alpha**, the learning rate is set to a constant 0.5
- **epsilon_decay** used for exponentially decay epsilon is set to 1.5e-5
- **gamma**, the discount ratio for future rewards is set to 0.9

```
qt_input = {'data': data, 'dates':dates, 'stocks':stocks, 'trading_days':
trading_days, 'state_settings': state_setting}

qt(QTrader(qt_input, epsilon = 1.0, alpha = 0.5,epsilon_decay = 1.5e-5, gamma
= 0.9))
```

The Q_learner is trained on the historical data using the training only set 'train_only'.

```
qt.run('train_only')
```

The return for training on historical data is checked by computing the total balance of the stock ledger.

```
total_balance_historical = qt.total_balance()
```

The implementation is simulated by running the Q-learner of the training & evaluation set 'train_eval'. When running this set is the stock_ledger reinitialized so that the profits can be compared to balance.

```
qt.run('train_eval')
```

The return of the simulated implementation is computed.

```
total_balance = qt.total_balance()
```

Finally is a comparison with the benchmark computed. The output is dictionary with the average profit margins for the benchmark, the Q-learner and the average ratio between these two.

```
benchmark = qt.check_benchmark()
```

prepdata.py contains three functions for loading and preprocessing the data. **get_data** loads the data from the CSV files and stores the closing for each stock and date in a dictionary. **add_financial_stats** creates the additional features from the closing price. **prepare_date** is used to load the data with **get_data** and create additional features with **add_financial_stats**. It also splits the data in a training only set and training & evaluation set.

q_trader.py contains **QTrader** class with the algorithm used for Q-learning. The main functionality of this class is to discretize the continuous data into states, perform Q-learning, run Dyna simulations and keep a ledger of what stocks are owned and what the profit (or loss) for each them are.

State making

The states are made as tuples made from the discretization of the features that are selected to make the states. The first the to make the state is to make bins from the train_only set (simulated historical data). The bins are made sorting all historical data for a feature in size order and splitting it into N sections. N is the number discrete values. The values at the splits of the sections will be used to define limits between bins that a new value can be sorted into. **make_bins** is used the create the bins.The values are then discretized with the **discretize** function

```
def discretize(self, value, feature):
  ''' Descritizes a value depending on the feature using the computed bins'''
  output = None
  # Loops through the bin limits for the feature to select what bin
  # the value should be placed in
  for n, b in enumerate(self.bins[feature]):
    # If the value is less or equal to bin limit it will output the bin
    # number and break the loop
    if value <= b:
      output = n
      break
  # If the value is larger than highest bin limit it outputs the final bin
  if value > self.bins[feature][-1]:
    output = len(self.bins[feature])
  return output
```

The states are made with the **make_states_from_data** function.

```
def make_states_from_data(self):
  ''' Makes states for all data using the bins created for every
    feature selected to be used'''

  # Initializes an empty dictionary for keeping the states made
  states = {}

    # Loops through the two splitted parts of the data: 'train_only' and 'train_eval'
    for train in self.data:

    # Initializes a subdictionary for keeping the states for each stock
    subdict = {}

    # Loops through each stock kept in the current train-set
    for stock in self.data[train]:

      # Gets the continuous data
      stock_data = self.data[train][stock]

      # Gets the dates for the current stock
      days = stock_data.index

      # Gets the features used for making states
      features = self.bins.keys()

      # Initializes a list with a tuple for every day
      state_array = [()]*len(days)

      # Loops through the features
      for feature in features:

        # Gets the values for all dates of the current feature and stock
        values = stock_data[feature].values

        # Loops through the values to discretize them and add the to the tuple of its day
        for n, val in enumerate(values):

          # Discretizes the value and adds it to the tuple
          state_array[n] = state_array[n] + (self.discretize(val,feature),)

      # Creates a subdictionary for the stock with the dates and states.
      subdict[stock] = dict(zip(days,state_array))
    states[train] = subdict
  return states
```

When the state is used in Q-learning is the ownership status of the stock added to the state tuple.

Q-learning

The most important functions for are **learn**, **select_action** and **get_reward. Learn** updates the Q-table using the update rule.

```
def learn(self, state, action, reward, stock):
    ''' Function for Q-learning with the update rule. Future rewards is discounted'''

    # Gets the next state for the stock
    next_state = self.get_state(stock,self.next_day)
    # Selects the best action for the next state without any randomness with
    # the use of the Q-table for the current day.
    best_action = self.select_action(next_state, True, False)
    # Gets the maximum Q value for the next state
    max_Q = self.Q[next_state][best_action]
    # Updates the Q values for the current state and action with the
    # update rule with discounted future rewards.
    self.Q[state][action] = (1-self.alpha)*self.Q[state][action] + self.alpha*(reward +
self.gamma*max_Q)
```

select_action select the best action. It is adapted for different use cases.

```
def select_action(self, state, searching_for_argmax,run_dyna):
      ''' Function for selecting the best action for the current state. The main input is
       the current state. The two other inputs are options used for altering the usage of
       the function.

      Use cases:
      1: During Q-learning to find the best action for the current state
         Settings: searching_for_argmax = False, run_dyna = False
      2: During Q-learning to find the best action for the next state when
         discounting future rewards. No random selection.
          Settings: searching_for_argmax = True,
         run_dyna = False
      3: When running Dyna simulations. No random selection and the Q-table is
         updated for every Dyna run and does not use the Daily Q-table.
         Settings: searching_for_argmax = True, run_dyna = True
      '''
      # If run_dyna is set to False will the Q-table for the current day be
      # loaded. This means that the updates from learning will first be
      # availible the next day.
      if not run_dyna:
          # The Q-values for the current state
          q_state = self.Q_today[state]
      # If rund_dyna is set to True will the Q-table with the lastest updates
      # be used.
      else:
          # The Q-values for the current state
          q_state = self.Q[state]

      # If searching_for_argmax is set to False will the random exploration be
      # used for learning
      if not searching_for_argmax:
          # Gets a random number p
          p = random.random()
          # If the random number p is smaller than the random exploration factor
          # and the training mode is train_only will a random action be selected
          if self.learning == 'train_only' and p < self.epsilon:
              for action in q_state:
                  # The Q-values for the current state is temporarily overwritten
                  # with random values.
                  q_state[action] = random.random()

      # Initializes the best action for the state (best_action) and the value (best_value) as
None
      best_action = None
     best_value = None

      # Loops through the actions and evaluates what action is best.
     for action in q_state:
          # Gets the value of the current action
      value = q_state[action]

          # If no action has been evaluted is the current action the best.
      if best_value == None:
        best_action = action
        best_value = value

          # If the current action has a higher value than the best_value will
          # the current action be set as the best one.
      elif best_value < value:
        best_action = action
        best_value = value

          # If the best values equals the value for the current action will
          # the current action be used for the best action with a 50% probabilty.
      elif best_value == value:
        p3 = random.random()
        if p3 >= 0.5:
          best_action = action
          best_value = value

     return best_action
```

get_reward uses the daily return for the following trading day to make the reward.

```
def get_reward(self, stock, action):
    ''' Gets the reward for the executed action for the given stock'''

    # Get the daily return for the following day.
    daily_return = self.data[self.learning][stock]['Daily_return'].loc[self.next_day]

    # Gets the ownership status for the stock.
    ownership = self.stock_ledger[stock]['ownership']

    # Initializes the reward to zero
    reward = 0

    if ownership == 'Owned':
        # If the stock is owned and the selected action is sell.
        if action == 'Sell':
            # The reward is the negativ of the return. I.e if the stock goes
            # up after selling will the reward be negative.
            reward = -daily_return
        # If the stock is owned and the selected action is nothing.
        elif action == 'Nothing':
            # The reward is the return. I.e if the stock goes
            # down when keeping (not selling) the stock will it be negative.
            reward = daily_return


    elif ownership == 'Not owned':
        # If the stock is not owned and the selected action is buy.
        if action == 'Buy':
            # The reward is the return. I.e if the stock goes
            # up after buying will the reward be positive.
            reward = daily_return
        # If the stock is not owned and the selected action is nothing.
        elif action == 'Nothing':
            # The reward is the negativ of the return. I.e if the stock goes
            # up after not buying will the reward be negative.
            reward = -daily_return

    return reward
```

Dyna

Dyna simulations are used to improve the model between the trading days.

```
def run_dyna(self, number_of_runs):
    ''' Function to run a set number of Dyna simulations'''

    # Loops through 0 to the number of set runs
    for n in range(0,number_of_runs,1):

        # Gets a random state
        random_state = self.get_random_state()

        # Gets a random action from the random state
        random_action = self.get_random_action(random_state)

        # Get the most likely next state from the T-table
        likley_next_state = self.most_probable_next_state(random_state, random_action)

        # Gets the expected reward from the R-table
        likley_reward = self.R[random_state][random_action]

        # Updates the Q-table with Dyna-learning
        self.learn_dyna(random_state, random_action, likley_reward, likley_next_state)
```

Stock ledger
The stock ledger is updated with the function **update_ledger**.

```
def update_ledger(self, stock, action):
  '''Updates the stock ledger to execute the action.'''

  # Gets the price for the current day (the day after the trade was decided)
  price = self.data[self.learning][stock]['Price'][self.today]

  # If the action is to sell a owned stock:
  if action == 'Sell' and self.stock_ledger[stock]['ownership'] == 'Owned':
    # The ownership status is changed to 'Not owned'
    self.stock_ledger[stock]['ownership'] = 'Not owned'
    # The balance of the stock is increased with the current price
    self.stock_ledger[stock]['balance'] += price

  # If the action is to buy the stock
  elif action == 'Buy' and self.stock_ledger[stock]['ownership'] == 'Not owned':
    # The ownership status is changed to 'Owned'
    self.stock_ledger[stock]['ownership'] = 'Owned'
     # The balance of the stock is decreased with the current price
    self.stock_ledger[stock]['balance'] -= price
```

Complications during the coding process

One of the main difficulties during the coding process was to implement the function to select actions so that it can be used for random exploration, Dyna and when there should be no random selection of action. This over-complicated the function and the code was not as clean as desired. Another big issue was implementing the Q-learner as a object so that multiple Q-learners could be trained. This meant that a lot of code needed to be used for a single object and it made more difficult to implement the Q-learner.

Refinement

To refine the Q-learner are different methods for making the states investigated. The methods varies in the features used for the states and the the number of discrete steps for the selected feature. The refinement is done on the training period simulating building the model on historical data. The initial method was to use all available features with three different steps for each (no. 1 in the table). The average return for the benchmark during the training period is 12.8%.

No	Daily return	Daily return 1 day ago	Daily return 2 days ago	Rolling mean	Rolling standard deviation	Average Return %
1	3	3	3	3	3	76.9
2	2	2	2	2	2	111.3
3	2	-	-	-	-	147.1
4	3	-	-	-	-	150.9
5	5	-	-	-	-	140.1
6	10	-	-	-	-	131.4
7	20	-	-	-	-	110.5
8	50	-	-	-	-	79.3
9	2	2	2	-	-	127.0
10	3	3	3	-	-	110.8
11	5	5	5	-	-	83.0
12	-	-	-	2	2	163.1
13	-	-	-	3	3	151.1
14	-	-	-	5	5	124.2
15	-	-	-	10	10	80.6
16	2	-	-	2	2	142.8
17	3	-	-	3	3	116.5
18	5	-	-	5	5	74.7

The five best methods are run three times more to get a mean return for them. This allows to account for variations in the return due to randomness of the exploration and from running Dyna. The results are presented in the table below,

No	Run 1 -%	Run 2 -%	Run 3 - %	Mean return - %
3	165.7	154.1	148.2	156.0
4	149.1	139.7	151.7	146.8
12	162.8	173.9	167.8	168.2
13	151.1	139.9	143.7	144.9
15	140.3	139.9	132.7	137.6

All of the 18 tested state methods performs better than the benchmark. Method no. 12 performs best

when running the five methods with the highest returns again. No. 12 has a mean return of 168.2% over the training period. This is significantly better than the benchmark.

No. 12 is therefore selected for the Q-learner. Combining the state method (no. 12) of using binary values for the rolling mean and the rolling standard deviation with the ownership status (owned or not owned) gives a total number of states to be 2^3 = 8.

IV. Results

Model Evaluation and Validation

The reliability of the final model is tested by running it five times on the evaluation period simulating a implementation. The benchmarks return on this period is 15.4%.. The results from these are shown in the table below:

Run	**Return of trading [%]**
1st	65.3
2nd	78.8
3rd	74.7
4th	74.9
5th	74.4
Mean	73.6

All the results from all five runs show that the the final model perform significantly better than the benchmark of 15.4%. The mean return of 73.6% over the period is much higher. The return of trading spans between 65.3% and 78.8%. The span is not very large but shows that there is some randomness affecting the Q-learning agent. The most likely causes is the random exploration or the randomness of the Dyna process.

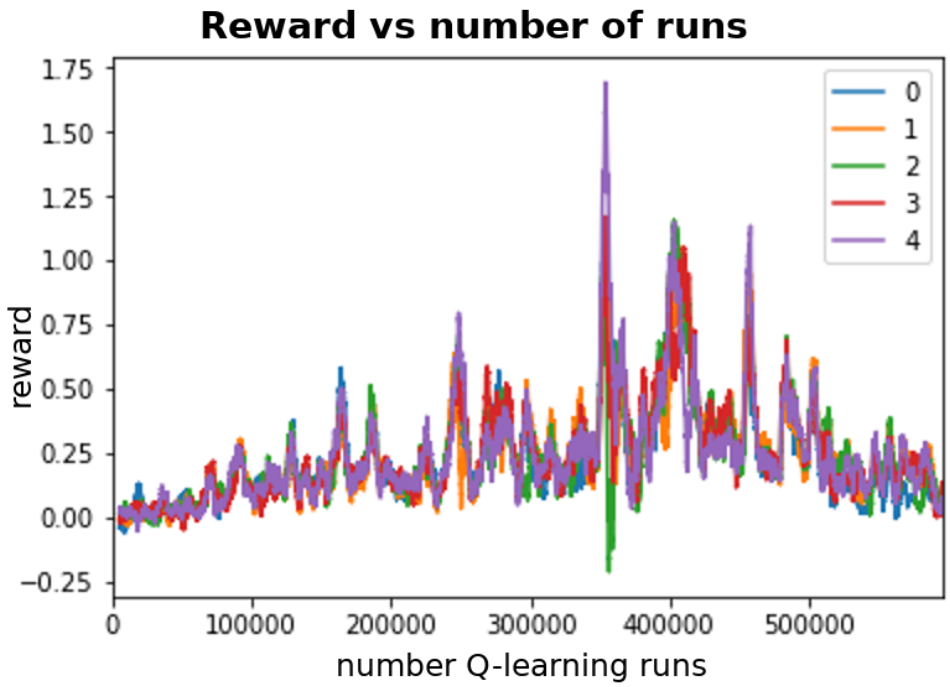
The model should be considered robust despite the the variation of the return. The reason for this is that the all returns with the final models are very high compared the benchmark. The results are reasonable as they are better than the benchmark but not so high that it would be unreasonable. The model can therefore be trusted.

Justification
The benchmark of buying stocks at the start of the evaluation period and selling them at the end gives a return of 15.4%. The return for the final model over the implementation period is 73.6%. This is a significant improvement compared to the benchmark. The problem can therefore be seen as solved.

V. Conclusion

Free-Form Visualization

The plot shows the rewards received by the agent when running the Q-learner five times with the selected refinements. The horizontal axis is the number of runs of the Q-learning (one for each stock and trading day). The vertical axis shows the rewards received when executing actions. The time line included both the training period and the evaluation period.



The plot shows that the reward are fluctuating around zero when the Q-learner are starting to learn. The size of the reward increases until around 400 000 Q-learning runs and then decreases somewhat. All five instances show relatively similar results.

The reason for the decrease of the rewards after around 400 000 runs is that it coincides with a peak of the stock prices that the Q-learner had to adjust to when the prices started to decline. Even though the rewards declined after 400 000 runs they are still overwhelmingly positive. This means that the Q-learner is still likely earn money on its trades.

Reflection

The process used for this project was not exactly like planned in the capstone proposal. The first step in the project was to determine how to make the rewards for the Q-learner to actually learn something useful. This was more difficult than expected. After that the attention was turned into how to make the continuous data into discrete for making states. This was also challenging as the the discrete variable has to be able to represent a lot of varying data without losing too much vital information. After the discretization was implementing was the work on making features from the price started. This was followed by writing the code for the Q-learner and Dyna process. When the Q-learner was functional was the refinement process started by running multiple Q-learners with different methods for making states on the the data used to simulate historical data. The returns from the Q-learners was evaluated

after this and the five best method were run three times more to check consistency and select the best performing one. The best performing method was then run five times on the later data used to simulate a implementation. Finally was the results then evaluated and analyzed and the report was written.

The most difficult aspects of this project was to figure out how the Q-learner should be rewarded and how to make the necessary states from the continuous data. The more interesting aspect of this project is the experience gained from starting with blank page to build a working machine learning solution. This was a cool and fun project.

The final model and solution fits my expectations for the problem, which were to have a Q-learner that can take good decisions on buying and selling stocks. The solution can be used in a general setting but it should probably be improved before that .

Improvement

There are several improvements that should be able to improve the trading agent. A possible improvement is to improve the making of the states for the Q-learner by using neural networks. This should be able to generate features and states automatically and thereby find better methods for making states. Another thing that could make the Q-learner agent more realistic is to add the cost of trading to rewards to Q-learner. This could ensure that the agent is only rewarded when a trade is profitable after any fees from stock brokers. Finally could possible the algorithm be improved by using deep learning instead of Q-learning. Deep learning should be able to be used for both making states and or to select good moments to buy and sell stocks.

References

[1] - https://www.cis.upenn.edu/~mkearns/papers/rlexec.pdf

[2] - http://cslt.riit.tsinghua.edu.cn/mediawiki/images/d/dd/%E6%B1%AA%E6%B4%8B-DQNStock.pdf

[3] - https://www.kaggle.com/camnugent/sandp500