Machine Learning Engineer Nanodegree

Capstone Project

Siting Huang
May 2, 2017

I. Definition

Project Overview

Hedge fund companies are increasingly turning to machine learning to help them predict market movements and optimize tradings. In stock trading domain, two kinds of approaches are promising:

1. apply machine learning to forecast price movements and then buy or sell with the predicted price change
2. apply machine learning to directly learn a trading strategy which tells when to trade and how to trade

The first approach ignores some important issues such as the certainty of the price change. It also doesn't tell when to enter or exit a position. To overcome this, we look at the second approach in this project to learn a more integrated trading strategy.

The second approach describes stock trading as a sequential decision making task in which the trader needs to perform a sequence of actions to reach some goal. These sequential decision making tasks are effectively dealt with in the area of reinforcement learning.

Problem Statement

Q-learning is one reinforcement learning technique. In this project I will implement a Q-Learner and develop a Strategy Learner that can learn a trading policy using my Q-Learner.

My Strategy Learner will implement the following API:

```
import StrategyLearner as sl
learner = sl.StrategyLearner(verbose = False) #contruct a learner
learner.addEvidence(symbol = "IBM", sd =dt.datetime(2007,12,31),\
        ed =dt.datetime(2009,12,31), sv = 10000) # training step
trades = learner.testPolicy(symbol = "IBM",\
        sd =dt.datetime(2009,12,31), ed =dt.datetime(2011,12,31),\
        sv = 10000) # testing step
```

The input parameters are:

- symbol: the stock symbol to train on
- sd: A datetime object that represents the start date
- ed: A datetime object that represents the end date
- sv: Start value of the portfolio

The output result is:

- trades: A data frame whose values represent trades for each day. This project only allows holdings of -100, 0 and 100 shares of stock. Therefore trades have legal values of +200, +100, 0, -100 and -200 as long as net holdings are constrained to -100, 0, and 100. Values of +100 (+200) indicate a BUY of 100 (200) shares, values of -100 (-200) indicate a SELL of 100 (200) shares, and 0 indicates NOTHING.

This project originated from the below sources:

1. The project assignment by Gerogia Tech University:

   http://quantsoftware.gatech.edu/Summer_2016_Project_5

2. Udacity's course Machine Learning for Trading:

   https://de.udacity.com/course/machine-learning-for-trading–ud501/

Metrics

The benchmark in this project is the buy-and-hold strategy, where we buy 100 shares on the first day and sell 100 shares on the last day.

The metric used in the project is the cumulative return of the portfolio over the chosen period, computed as following:

```
cumulative_return = Portfolio/StartValue - 1
```

StartValue is initial cash holding. Portfolio is the total portfolio value which is the initial cash plus gain/loss from trading the stock over the period. The cumulative return of my Strategy Learner and the buy-and-hold strategy will be compared. This is equivalent to compare the absolute cash gain/loss from the two strategies. The result will show whether the strategy learner can outperform the buy-and-hold strategy or not. This is an important result to know since it matters the most whether any active trading returned by the Srategy Learner will give more investment income than passively holding the stock.

However this metric is very sensitive to the StartValue. In this project, we are limited to hold or short 100 shares of stock. For simplicity we will choose the stocks and the StartValue where the StartValue will be enough to hold 100 shares of the chosen stocks in the long run. so we don't need to care about whether we will run out of cash to buy 100 shares. But the StartValue will only be a little bit more than enough to buy 100 shares; we will allocate most of our StartValue to trade the stock instead of using only for example 1% of it. So the metric in percentages will still make sense in terms of investment return in real world.

II. Analysis

Data Exploration

The data consists of daily historical prices of two stocks (IBM and GOOG) and one S&P 500 ETF (SPY). The data was downloaded from Yahoo Finance in fotmats of CSV files. We have:

- SPY: Jan 29, 1993 - Apr 21, 2017 (6102 days with trading data)
- GOOG: Aug 19, 2004 - Apr 21, 2017 (3191 days with trading data)
- IBM: Jan 2, 1962 - Apr 26, 2017 (13925 days with trading data)

The starting time varies for stocks depending on when the stock started to be traded in the exchange.

Below is a slice of sample data of SPY for three days.

Date	Open	High	Low	Close	Volumn	Adj Close
2017-04-21	235.25	235.30	234.13	234.58	100595400	234.58
2017-04-20	234.14	235.85	233.77	235.33	88657000	235.33
2017-04-19	234.52	234.94	233.17	233.44	66861500	233.44

The historical financial data usually has the same form consisting of the following information:

- Date: the date when the stock information was recorded.
- Open: the price that the stock opened at.
- High/Low: the highest/lowest price of the stock throughout the day.
- Close: the actual price that was reported at the exchange when the stock closed for that day.
- Adj Close: the stock price adjusted for stock splits, dividend payments and etc.
- Volumn: the amount of money traded for the stock during the day

Close and Adj Close are the same on the most current date and might differ for past dates. While Close shows the closing price, Adj Close shows how the stock value truly developed over the past.

There are dates when the stock market was closed. SPY is used as a reference to select the dates with open stock market. For other stocks, we further check whether there were any missing data on the dates when the market was open. This could happen for certain stocks in reality. After checking, we know that IBM and GOOG have no missing values in our data.

Exploratory Visualization

In this section, we discuss and visualize relevant characteristics about the data.

1. Daily returns of stock price

As stocks are priced at different level, it makes more sense to compare their performance through daily return. The daily return for date t is

```
daily_return(t) = stock_price(t)/stock_price(t-1) - 1 .
```

From the below summary statistics of the daily return of the three stocks between Aug 19, 2004 and Apr 21, 2017, the average daily return of GOOG is the highest but it also has the highest standard deviation. SPY has higher daily return than IBM and lower standard deviation.

	SPY	GOOG	IBM
count	3191	3191	3191
mean	0.000392	0.001075	0.000367
std	0.011999	0.019594	0.013364
min	-0.098448	-0.116091	-0.082975
25%	-0.003982	-0.007886	-0.006158
50%	0.000677	0.000480	0.000283
75%	0.005483	0.010033	0.007226
max	0.145198	0.199915	0.115150

2. Cumulative returns of stock prices

Another fair value to compare stock performance is the cumulative return. We use "0" to refer the first date. Cumulative return on date t is:

```
cumulative_return(t) = stock_price(t)/stock_price(0) - 1.
```

Figure 1 plots the cumulative returns of the three stocks from Aug 19, 2004 to Apr 21, 2017. It shows that the cumulative daily return of GOOG is the highest. If we follow a buy-and-hold strategy from Aug 19, 2004 to Apr 21, 2017, our stock holding value would have increased by 1.77 times for SPY, 15.82 times for GOOG and 1.42 times for IBM.

3. Bollinger Bands (R)

Bollinger Bands (R) is a technical indicator based on moving average and volatility:

```
    upper_band(t) = rm(t) + rstd(t) * 2
    lower_band(t) = rm(t) - rstd(t) * 2
```

where rm(t) is the N-period moving average and rstd(t) is the N-period moving standard deviation, both with N being a window size.

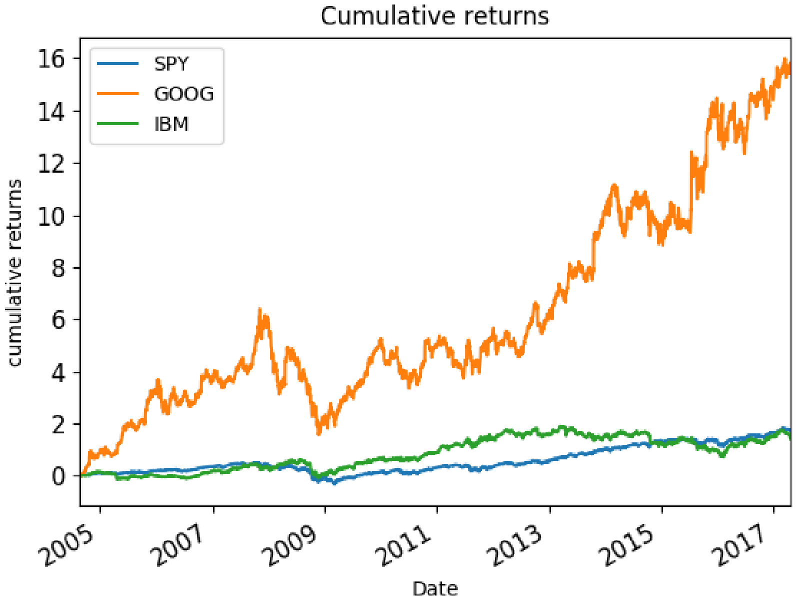

Figure 1: Cumulative returns

Some traders believe that when the prices move closer to the upper band, the market is overbought, meaning that the stock is overpriced; when the prices move closer to the lower band, the market is underbought. Some trading rules are developed based on the idea. For example, when the price falls below the lower Bollinger Band and then goes up across it again, one can see this as a BUY signal; when the stock arises above the upper Bollinger Band and then goes down across it, one can see this as a SELL signal. If we apply this to Figure 2 which is GOOG price with Bollinger Bands $^{\text{(R)}}$, there were a BUY signal in mid of June 2016 and a SELL signal end of August 2016. In this particular example, we would have a good investment return if we follow the signals.

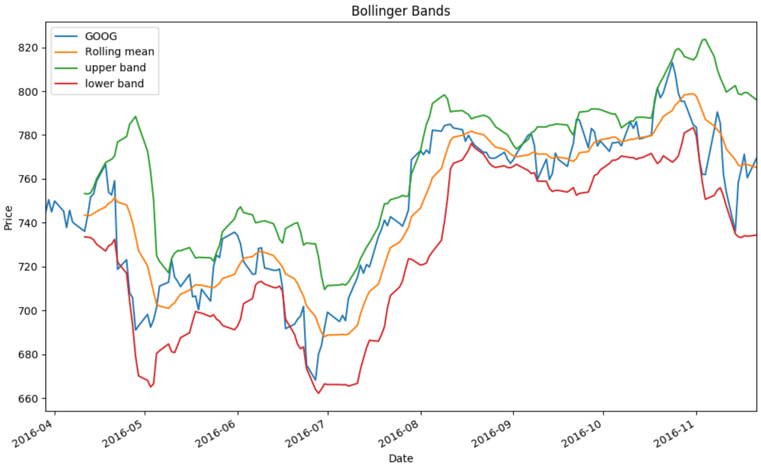

Figure 2: GOOG with Bollinger Bands $^{\text{(R)}}$

To more effectively use Bollinger Bands $^{\text{(R)}}$ as a scalar indicator, we include Bollinger percent b (%b):

`percent_b(t) = (price(t) - lower_band(t))/(upper_band(t) - lower_band(t)).`

Algorithms and Techniques

We will use a decaying, ϵ-greedy Q-learning algorithm to build a Strategy Learner that will learn to trade stock. The algorithm and techniques are heavily based on the project assignment by Gerogia Tech University and Udacity's course Machine Learning for Trading. We will first identify states, actions and rewards and then discuss how build a Strategy Learner

The advantages of using O-Learning include:

- It is a model-free approach and can be applied to the domains where the states and transitions are not fully defined.
- The Q-value for any state-action pair takes into account future rewards. Thus, it encodes both the best possible value of a state ($\max_a Q(s, a)$) as well as the best policy in terms of the action that should be taken ($\operatorname{argmax}_a Q(s, a)$).

1. Identify states, actions and rewards

States includes two technical features of the stocks, Adj Close/SMA and Bollinger percent b, and two portfolio features, holding stock and return since entry:

- Adj Close/SMA where SMA is the simple moving average with a window size to be specified
- Bollinger percent b (%b) with window size to be specified
- holding stock: the amount of stock the portfolio is holding. If not holding a stock, we might not necessarily want to sell it
- return since entry: helpful to set exit point

Surely the state are multidimensional and continuous based on the nature of the first two features. And there are some reinforcement learning methods that are able to work with continuous state. But here we will discretize these features to work more easily with Q-table.

Rewards for an action is the daily return of the stock holding. We use this instead of cumulative return to ensure fast convergence.

Actions has a different meaning than the activities of SELL, BUY and NOTHING in the code I intend to implement. In this projects we can only be long or short 100 shares. As long as the net holding is -100, 0 or 100 shares, we might perform five types of trading activities to change shares of -200, -100, 0, 100 and 200. To simplify this and to make actions have only three values, I refer three actions as “holding -100 shares”, “holding 0 shares” and “holding 100 shares”.

2. Build a Strategy Learner

We will first implement a basic Q-Learner: For every state the Q-Learner visits, create an entry in the Q-table for all state-action pairs available. Then, when the agent encounters a state and performs an action, update the Q-value associated with that state-action pair based on the reward received. The agent choose the best action for each state based on the Q-values of each state-action pair possible.

Then we build the Strategy Learner using the basic Q-Learner. The Strategy Learner will have one method for policy learning and one method for testing. For learning:

- Instantiate a Q-learner
- For each day in the training data:
    - Compute the current state (including holding)

    - Compute the reward for the last action
    - Query the learner with the current state and reward to get an action
    - Implement the action the learner returned (BUY, SELL, NOTHING), and update portfolio value
- Repeat the above loop multiple times until cumulative return stops improving.

For testing:

- For each day in the testing data:
    - Compute the current state
    - Query the learner with the current state to get an action
    - Implement the action the learner returned (BUY, SELL, NOTHING), and update portfolio value
- Return the resulting trades in a data frame (details below).

Benchmark

Cumulative return is the metric used in the project. The benchmark trading policy will be the buy-and-hold strategy, where we buy 100 shares on the first day and sell 100 shares on the last day and the portfolio value will increase by 100 times the stock price changes between the last day and the first day.

III. Methodology

Data Preprocessing

We have three CSV files for SPY, GOOG and IBM. Each CSV file consist of columns of Date, Open/Close, High/Low, Volumn and Adj Close. The dates in the CSV files are in the reversed order with the most current date in the first row.

The CSV files are read in as Pandas dataframes indexed by dates. Each dataframe consists of one feature, for example Adj Close. Column names are stock symbols such as "GOOG". SPY are used as a reference whenever we read in any stock data to select the dates when the market was open. This can be done by first setting up a empty dataframe for the time period we are interested and setting index to be the dates. All the natural dates (including weekends) will be included in chronological order. We then join the empty dataframe with SPY and remove the dates that had no SPY values.

It could happen for stocks that they are not traded in some dates even when the market was open. To check this, we examine in the stock dataframe whether there are any NA values. Luckily, in our data for GOOG and IBM, the data are complete. If there are missing data for some dates, we can use Backfill and Forwardfill techniques to fill the missing data by the price before or after.

Implementation

In this section, the algorithms and techniques outlined in section **Algorithms and Techniques** will be documented in details.

1. Create States through discretization

The chosen technical features are first computed and then discretized. For discretization, we follow the simple approach described in Machine Learning for Trading. First we find out the thresholds for the bins:

```python
def compute_discretizing_thresholds(data, steps):
    """ compute thresholds """
    thresholds = np.zeros(steps)
    stepsize = size(data)/steps
    data.sort()
    for i in range(0, steps):
        thresholds[i] = data[(i+1)*stepsize]
    return thresholds
```

Steps is the number of groups we want to discretize the data. For example if we want convert real number into an integer from 0 o 9, we set the steps to be 10. Then we will have thresholds of length 10, which are the right limits. With the thresholds, we can further convert a real number to an integer from 0 to 9. For example, if a real number is between thresholds[7] and thresholds[8], the corresponding discretized value is 8. When applying the thresholds calculated from training set to testing test, the range of testing set values might exceed the range of the training set. The smaller values in testing below thresholds[0] will have discretized values 0, the larger values in the testing set above thresholds[9] will have discretized values 9. Since training data and testing data might have different distribution, the discretized values from testing set might not distribute evenly among 0-9.

The state includes four features to be discretized:

- the ratio of Adj Close/SMA

The ratio of Adj Close/SMA can be immediately calculated through column-wise calculation after the stock price are read in. The results will have NA in the first rows since SMA is only available after a window size. Therefore we use function dropna(inplace=True) to remove the first few dates before discretization. After removing NA, we discretize it from 0-9.

- Bollinger percent b (%b)

Bollinger percent b can be immediately calculated through column-wise calculation after the stock price are read in. The results will have NA in the first rows since the Bollinger Bands $^{(R)}$ includes moving average and moving standard deviation with a window size to be specified. Again we remove the first few dates with NA and then discretize the values from 0 to 9.

- holding stock indicator

This indicates the amount of stock the portfolio is holding. In this project, the stock holding is limited to -100, 0 and 100 shares. Corresponding, the holding indicator will take values of -1, 0 and 1. Since we don't want to include negative values in the state, 1 will be added to the holding whenever we use it to create a state. This feature can only be calculated and incorporated into the state while we loop through each day in training and testing.

- return since entry indicator

I set the threshold to be 10%. If the portfolio cumulative return is above 10%, the indicator will be 1 otherwise 0. This feature can only be calculated and incorporated into the state while we loop through each day in training and testing.

After we have the above four features discretized to one digit, we combine them into one number by stacking them together:

```
state[t]= (holding+1) * 1000 + state_since_entry * 100  + \
          discrete_close_SMA[t] * 10 + discrete_percent_b[t]
```

Therefore the size of the state is 3*3*10*10 = 600 and the state has valid values of 0-199, 1000-1199 and 2000-2199.

2. Training and update Q-table

One note for action before describing the algorithms: Action indicates holding states and action = holding +1. Action has three values 0,1,2 while holding has values -1,0,1. If on a certain day the action is 0, it means that the holding is -1 meaning to hold -100 shares at the end of the day. If the next day the action is 2, it means that we will increase the stock holding to 100 shares. Therefore the underlying trading activity for that day is BUY 200 shares.

The algorithm for training step:

- Instantiate a Q-learner with Q(s,a)
- Perform n trials (until cumulative return stops improving):
    - Initialize holding and portfolio value: holding = 0, portfolio = sv
    - Kick off the first day:
        - state s ← current state
        - action a ← argmax_aQ(s,a)
        - update holding ← action - 1
    - For each day in the rest training data:
        - New state s' ← current state
        - Compute the reward of last action:
          r = stock_daily_returns * holding
        - Query the Q-Learner with s' and r to get an action:
            - update Q table with <s, a, s',r>:
              Q(s,a) ← Q(s,a) + α * [r + γ * $\text{max}_{\text{a'}}$Q(s',a') - Q(s,a)],
              where α is the learning rate and γ is the discount rate

      - with probability *rar*, a is a random action. With probability $1 - rar$, a ← argmax_aQ(s',a), where *rar* is random action rate
      - s ← s'
      - decay *rar* using random action decay rate *radr*: $rar = rar * radr$
    - Update portfolio value:
      portfolio += stock_daily_changes * holding * 100
    - Implement action: holding = action a - 1
  - Compute cumulative return: cumulative_return = portfolio/sv - 1

3. Testing a model

For testing we will backtest on a later data. The algorithm is as follows:

- Initialize holding and portfolio value: holding = 0, portfolio = sv
- For each day in the testing data:
  - Compute the current state s
  - Query the Q-Learner with the current state to get an action:
    action a ← argmax_aQ(s,a)
  - Update portfolio value:
    portfolio += stock_daily_changes * holding * 100
  - Implement action and return the resulting trades:
    - old_holding = holding
    - holding = action a - 1
    - trades[t] = (holding - old_holding) * 100
- Compute cumulative return: cumulative_return = portfolio/sv - 1cumulative_return = portfolio/sv - 1

Refinement

We call it one simulation for the process of instantiating a Strategy Learner and subsequently training and testing with this Strategy Learner.

Within each simulation, we will perform 300 trials where each trail the Strategy Learner will go through each day in the training data and calculate the cumulative return at the end. Thus for each simulation, we can plot the cumulative returns over the trial number. In the initial run, I did 10 simulations with

- Input data: IBM, training 2007-12-31 to 2009-12-31, testing 2009-12-31 to 2011-12-31
- Initial parameters: $\alpha = 0.2$, $\gamma = 0.9$, $rar = 0.98$, $radr = 0.9999$, window = 15

As shown in Figure 3, in around 200 - 300 trials the cumulative return will not increase.

However it's noticeable that the final cumulative returns are different in each

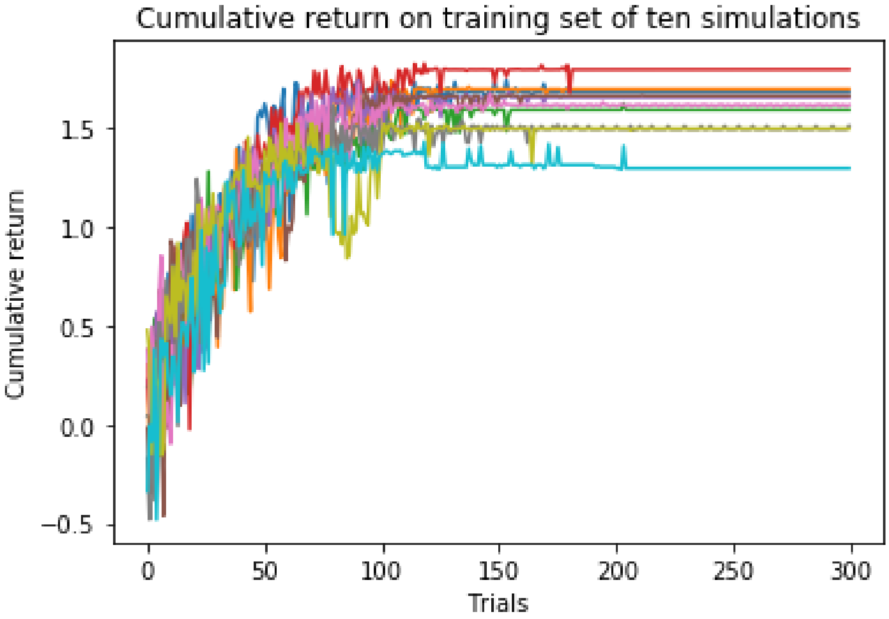

Figure 3: Cumulative return on training of ten runs

simulation though overall consistent. We use the average of 10 simulations to estimate the cumulative return on training and testing.

The model has a few hyperparameters: learning factor α, exploration factor rar, decaying factor for exploration $radr$, discount factor γ, window size. These parameters can be optimized using grid search and cross-validation. Due to time constraint and the limited amount of data, I did not fully implement cross-validation. I explored a bit the parameter combinations on a few some training and testing data. I ran ten simulations for each combination of the below parameters with two to three options. In total there are 24 combinations.

- Input data:
  - IBM, training 2007-12-31 to 2009-12-31, testing 2009-12-31 to 2011-12-31
  - IBM, training 2009-12-31 to 2011-12-31, testing 2011-12-31 to 2013-12-31
- learning factor α: 0.1 and 0.2
- exploration factor rate of random action rar: 0.9 and 0.98
- decaying factor for exploraion $radr$: 0.9999 and 0.999
- window: 10,15 and 20

The best parameter combination with the highest cumulative return on average on the testing samples is: $\alpha = 0.1$, $rar = 0.98$, $radr = 0.9999$, window = 15.

We will use this set of parameter for futher testing.

Implementation and results of this section can be found in:

- 101_intitial_learner.ipython
- 102_find_optimal_parameters.ipython

IV. Results

Model Evaluation and Validation

As illustrated in Figure 3, the model returns different cumulative returns in each simulation.
The model is not stable in this sense.

I check one simulation to evaluation the model. Figure 4 shows the trades returned by the Strategy Learner. The trades are very frequent.

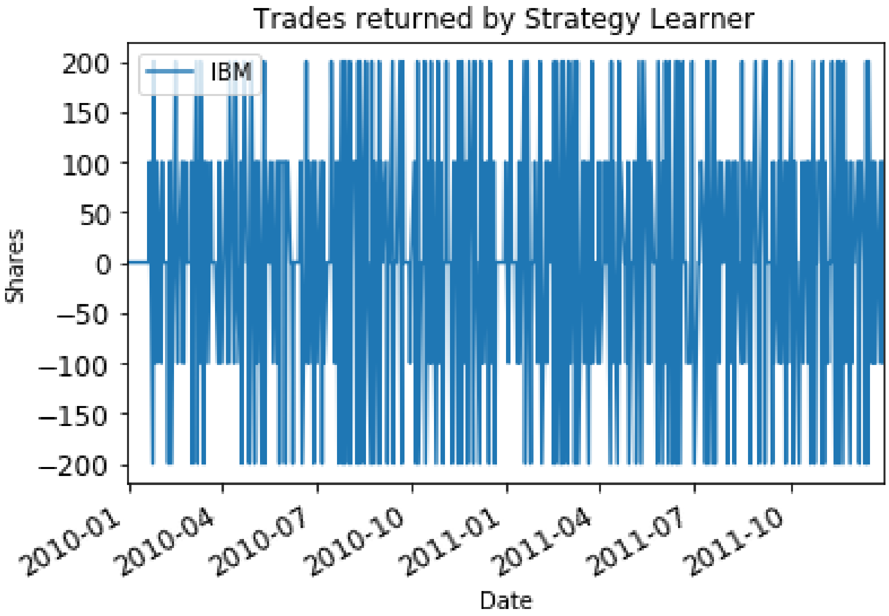

Figure 4: Trades returned by Strategy Learner

In general, I don't think the model is robust enough.

Justification

I test the model on three stocks with in total 13 different input. For each input, I use the average of 10 simulations to estimate the cumulative return and compare the results to the benchmark. Below is the result of the 13 runs:

Stock	testing dateframe	learning dataframe	Strategy Learner	benchmark	Wining Strategy
IBM	2003,12,31 ~ 2005,12,31	2005,12,31 ~ 2007,12,31	0.1276	0.2289	BM
IBM	2005,12,31 ~ 2007,12,31	2007,12,31 ~ 2009,12,31	0.4427	0.2249	SL
IBM	2007,12,31 ~ 2009,12,31	2009,12,31 ~ 2011,12,31	0.3954	0.5018	BM
IBM	2009,12,31 ~ 2011,12,31	2011,12,31 ~ 2013,12,31	0.1724	0.0708	SL
IBM	2011,12,31 ~ 2013,12,31	2013,12,31 ~ 2015,12,31	-0.3669	-0.3795	SL
SPY	2003,12,31 ~ 2005,12,31	2005,12,31 ~ 2007,12,31	0.0187	0.1981	BM
SPY	2005,12,31 ~ 2007,12,31	2007,12,31 ~ 2009,12,31	0.6430	-0.2424	SL
SPY	2007,12,31 ~ 2009,12,31	2009,12,31 ~ 2011,12,31	0.0572	0.1657	BM
SPY	2009,12,31 ~ 2011,12,31	2011,12,31 ~ 2013,12,31	0.1021	0.5847	BM
SPY	2011,12,31 ~ 2013,12,31	2013,12,31 ~ 2015,12,31	0.6687	0.2571	SL
GOOG	2005,12,31 ~ 2007,12,31	2007,12,31 ~ 2009,12,31	0.8733	-0.3571	SL
GOOG	2007,12,31 ~ 2009,12,31	2009,12,31 ~ 2011,12,31	-0.3104	0.1294	BM
GOOG	2009,12,31 ~ 2011,12,31	2011,12,31 ~ 2016,12,31	-0.8763	2.2742	BM

The Strategy Learner outperforms the buy-and-hold strategy in 6 out of 13 runs. The Strategy Learner returns negative returns in 3 out 13 runs.

Overall I don't think the final solution has solved the problem perfectly. But given the fact that it does outperform the buy-and-hold strategy for quite a few times, I think it's worthwhile to further improve this approach and it's promising that it will can solve the problem better, at least have higher chance to outperform the buy-and-hold strategy.

Implementation and results of this section can be found in:

- 201_model_validation.ipython
- 202_evaluate_result_one_simulation.ipython

V. Conclusion

Free-Form Visualization

For model validation, my Strategy Learner was run on three stocks in total 13 different input periods. Since it does not make sense to compare the return by different stocks and different periods, the figure below shows results of the 13 runs separately. It should give an indication on the performance of the Strategy Learner compared to the benchmark buy-and-hold strategy. The figure presents the cumulative returns of the Strategy Learner vs buy-and-hold strategy for the 13 runs.

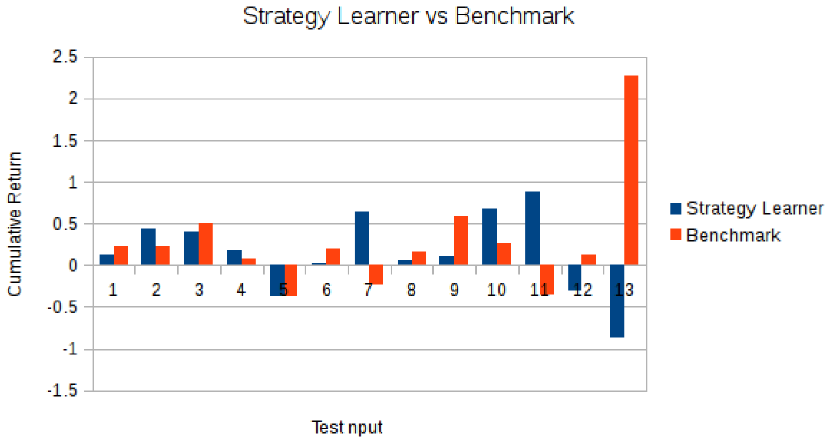

Figure 5: Strategy Learner vs Benchmark

The cumulative return of the benchmark buy-and-hold strategy shows that prices development vary significantly among differently stocks and different periods. Buy-and-hold strategy is certainly not a good strategy to apply to all stocks but it needs careful picking. However my Strategy Leaner fails to overcome this uncertainty either. From the Figure we can see that

(1) The Strategy Leaner sometimes give negative investment return (in the 5th, 12th and 13th runs) which is definitely not ideal.

(2) The Strategy Leaner outperforms the buy-and-hold strategy only around half of the runs
(3) The investment results from the two strategies in the same period of time can vary significantly (e.g. in the 11th and 13th runs)

Overall the performance of the Strategy Learner is not stable and there is still great room to improve.

Reflection

In this project I have applied Q-Learning to build a Strategy Learner to trade stocks. The API of the Strategy Learner allows it to learn a trading policy given a period of historical stock prices and return a series of trading actions over other period given their historical stock prices. We can easily backtest the learned policy on a later timeframe. The cumulative return of the portfolio is used as a metric to evaluate the performance of the Strategy Learner. We refine the model through a grid search over relevant parameters. We applied this approach to SPY, GOOG and IBM over different period and compare the cumulative return of it to buy-and-hold strategy.

I found most interesting and most challenging part about the project is how to define the state space with technical features and portfolio features. I think this project set up a good basis to further include other technical features and see how it will improve.

One concern I have in the project is the size of the state space. In our construction, the size of the state space is 600. So we will have at least two years of daily stock prices to visit each state. However I doubt the relevance of the prices two years ago on today's stock price.

I was impressed by the performance of the approach given the limited information we include in the state. Although it's in general not as good as buy-and-hold strategy, it still gives a nice investment return which fits my expectation.

Improvement

Some aspects of the model can be improved:

1. Improve the state

There are other technical features that also include information of stock movements. In general we have four types of indicators to be considered:

- Trend indicators, such as Moving Average Convergence Divergence (MACD)
- Momentum indicators, such as Relative Strength Index (RSI)
- Volumn indicators such as Money Flow Index (MFI)
- Volatility indicators other than Bollinger Bands $^{(R)}$ such as Volatility Index (VIX)

Include more indicators definitely requires larger state space and we will face curse of dimensionality. It will also require more effort on feature selection.

2. Use Functional Approximation to allow continuous state space

To scale the model up to continuous state space, we can use Q-learning with functional approximation where we approximate Q as a function of continuous state s and actions a.

3. Refine reward function

The trades returned by the model suggest frequent trading which in reality will triger more trading expenses and lower the total return. Therefore we can refine the reward function to take into account the trading frequency.