Capstone Project

Machine Learning Engineer Nanodegree

Jeroen Sweerts
December 13th, 2016

I. Definition

Project Overview

In this project, we apply Q-Learning to develop an automated trading system. The model uses technical analysis indicators as input for training. All the code was written in Python. We used the Sharpe ratio as metric to compare the performance of the different models and to measure the success of the stepwise improvements of the model.

Problem Statement

In this project, we assumed a trading agent investing in WTI front month futures contracts. For this futures contract (ticker CLc1) we built a Python program with the following functionalities:

1. Load a csv file with historic daily [Open,High,Low,Close] quotes for ticker CLc1
2. Calculate for this data set a technical indicator (RSI) and add it to the data set
3. Split the data set in a training and test set (75-25%).
4. Apply Q-learning on the training data set to develop an optimal trade policy for 3 possible trading actions [buy,sell,hold]
5. Test performance of the optimal policy on the test data set.

Technical analysts (as opposed to fundamental analysts) believe that the market is always right. They don't spend time in analyzing all the fundamental factors that could influence demand or supply of a certain investment asset because they are convinced that all this information is already factored into the price of the asset. Most technical indicators can be calculated with the open, high, low, close prices and traded volumes of the asset. Generally, it is easier to automate a trading system based on technical analysis as compared to fundamental analysis.

Technical analysts are also convinced that humans will behave similarly to how they have responded in the past in similar circumstances. They believe that because of this human nature (fear and greed), prices contain predictive patterns.

With the Python tool that we developed for this project we investigate if technical analysis is a good basis for making investment decisions and how to automate this decision process.

Metrics

We use the Sharpe Ratio to evaluate the performance of an optimal trading policy.
The Sharpe ratio characterizes how well the return of an asset compensates the investor for the risk taken. When comparing two trading strategies against this common benchmark, the one with

a higher Sharpe ratio provides better returns for the same risk (or, equivalently, the same return for lower risk[1]).
To calculate the Sharpe Ratio, you need to calculate the expected value of the return on a portfolio, minus the risk-free rate and divide it by the standard deviation of the same difference.

$\boldsymbol{Sharpe\ Ratio} = \frac{\boldsymbol{E[Rp-Rf]}}{\boldsymbol{std[Rp-Rf]}}$, where Rp is in our case the daily return that we get by applying our optimal policy on the test data set and Rf is the risk-free rate (in the current low interest environment we decided to set Rf to zero).

We have 252 trading days in a year, so to annualize the Sharpe Ratio, we need to multiply the above equation by $\sqrt{252}$.
A negative Sharpe ratio means that the return of the investment is smaller than the risk-free rate.

Usually, any Sharpe ratio greater than 1 is considered acceptable to good by investors. A ratio higher than 2 is rated as very good, and a ratio of 3 or higher is considered excellent[2]. We will apply this same classification to decide if our optimal Q-learning policy is acceptable, very good, or excellent.

II. Analysis

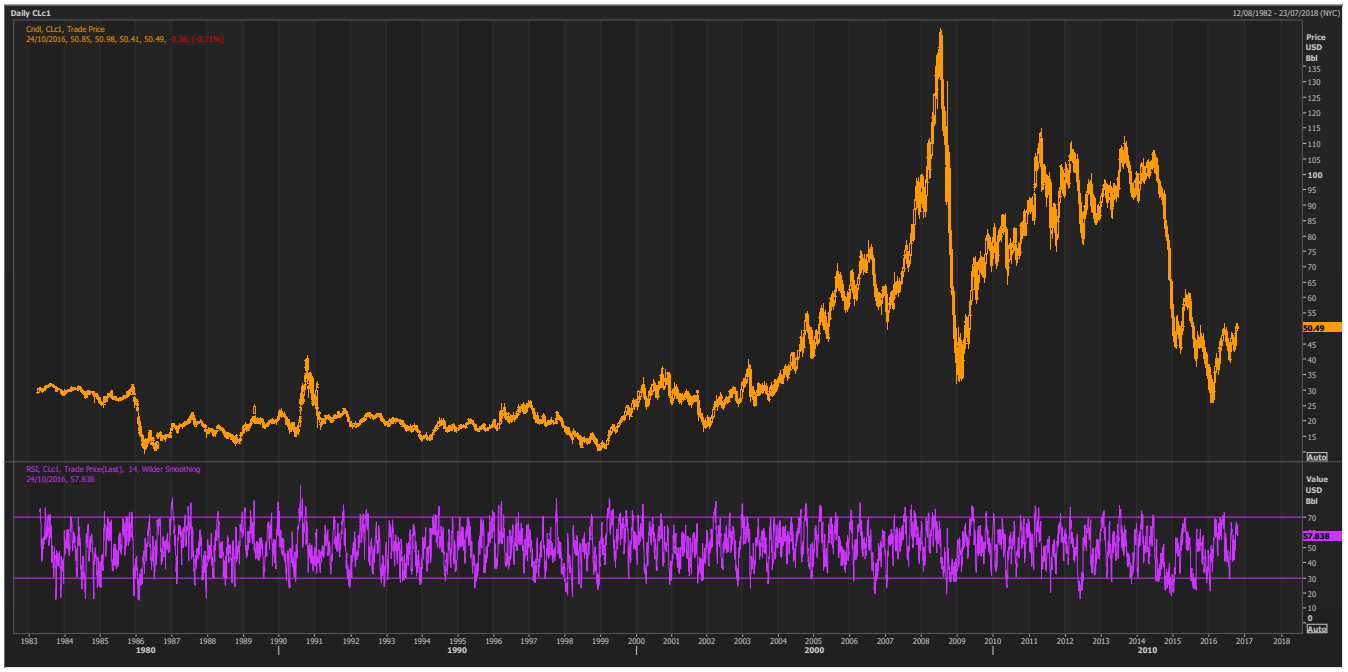
Data Exploration

The historical WTI price data set was taken from Reuters. It contains daily [Open,High,Low,Close] prices for the period starting 30 March 1983 and ending 14 September 2016 (8399 data points). This was the maximum amount of data available in Reuters for this ticker (CLc1). For the same period, we calculated the Relative Strength Index (RSI) which "technicians" believe to have a predictive power for the expected future price movement. For the calculation of the RSI, we use the TA-lib package which has an API available for Python. The calculated RSI values are added to the dataset.

The quality of the data is good. Reuters has already cleaned all data before making it available to their customers. Also, we compared the calculated values of the technical indicators with those made available by Reuters. It showed that the TA-lib library can reproduce the Reuters numbers.

On the chart below the yellow curve represents the historical WTI rolling front month future prices of our data set. We can see that the data contains all possible market conditions (stable, volatile and strong trends), which makes it an ideal training and test set. The purple curve graphs the Relative Strength Indicator (RSI). The RSI has values in the range [0,100].

[1] https://en.wikipedia.org/wiki/Sharpe_ratio
[2] http://www.investopedia.com/ask/answers/010815/what-good-sharpe-ratio.asp



Exploratory Visualization

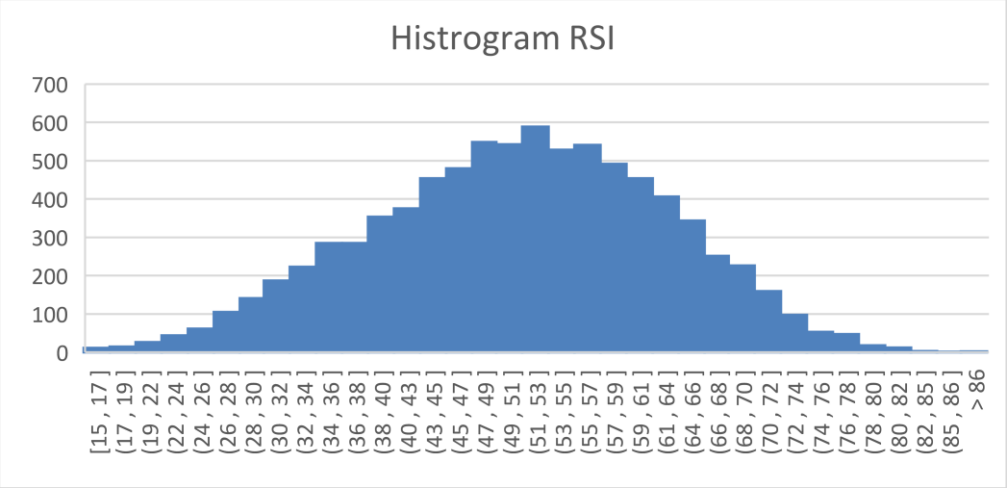
The **Relative Strength Indicator** (RSI) is one of the most popular and widely used technical indicators and was developed by J. Welles Wilder in 1978. The RSI measures the strength of an asset by comparing the up days to down days. Wilder based his index on the assumption that overbought levels generally occur after the market has advanced for a disproportionate number of days, and that oversold levels generally follow a significant number of declining days.[3]

The **algorithm** to calculate the RSI is as follows:

- UPS= (Sum of gains over N periods)/N
- DOWNS = (Sum of losses over N periods)/N
- RS=UPS/DOWNS
- RSI=100-[100/(1+RS)]

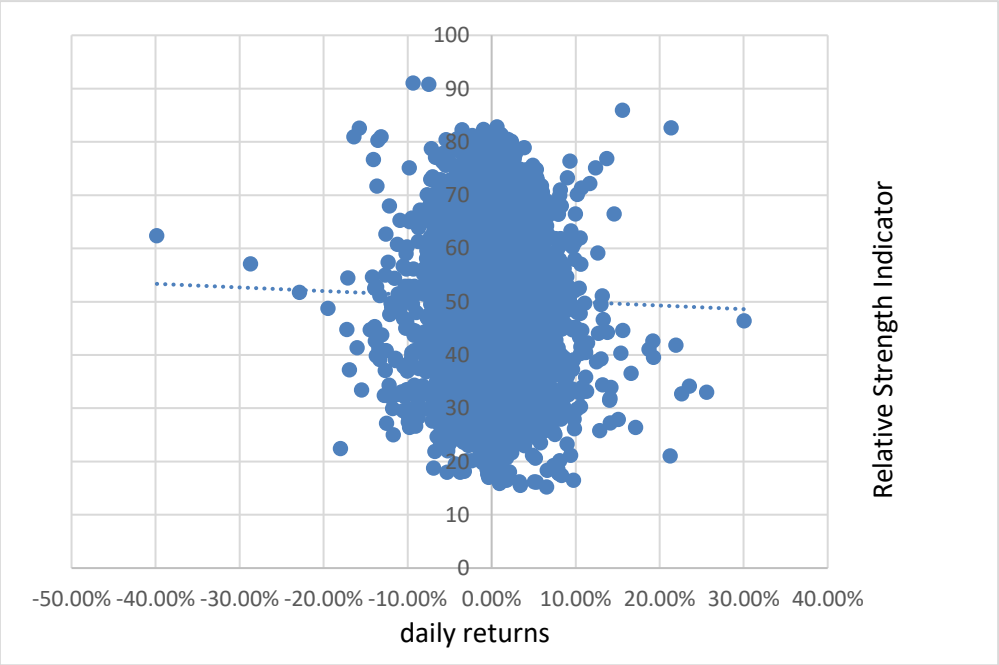
As mentioned earlier, the RSI can have values between 0 and 100. Typically, a value above 70 is considered as overbought and below 30 is considered as oversold. A popular value for N is 14 days.

[3] Technical Analysis by C. Kirkpatrick & J. Dahlquist



percentile	RSI
95%	69.5
5%	30.2

For our data set we plotted the histogram. 90% of the RSI values are between 70 and 30.



We also plotted daily returns of the WTI front month futures contract against its RSI(14) values to see if there is any correlation. This scatter plot doesn't show any clear correlation between both values, which seems to indicate that the RSI value is not a predictor for daily returns. Similar plots for 5, 10 and 30 day future returns give the same conclusion. So, from a first visual inspection, the RSI is not a good predictor for future price movements.

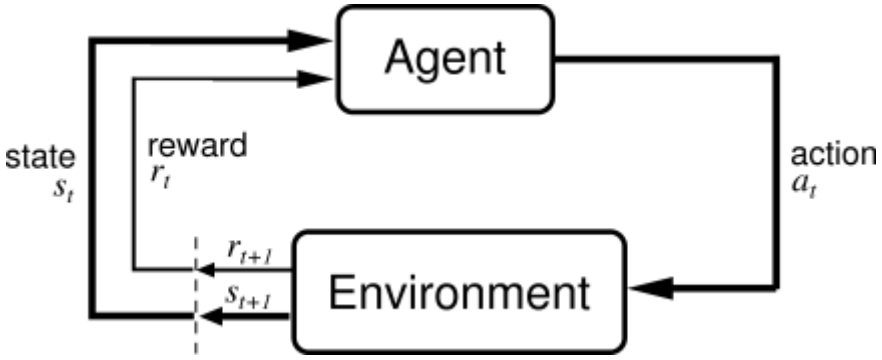
Algorithms and Techniques

We use **reinforcement learning** to create a trading policy that maximizes reward. The 2 main features of reinforcement learning are search through trial-and-error and delayed reward[4]. For a state the learner must discover which actions (buy, sell, hold) give him/her the most profit by trying them out. The great thing about reinforcement learning is that learning can start without any prior knowledge of the system. Only the rules of the game (the immediate reward for each action) must be known. This does not mean that applying this technique as a black box is a guarantee for success. Careful selection of state variables proves to be crucial for the quality of the optimal policy.

At first glance, this learning environment seems to be very well suited for developing optimal trading strategies:

- In a real trading environment, a certain trade will not only result in an immediate profit, but has also an impact on the future rewards. A trader will have limited resources (money) to invest. Investing some of these resources today will reduce the amount of resources that the investor will have available in the future.
- Also, the lack of prior knowledge is inherent to a trading environment. There might be hidden patterns in certain data that make this data a good trading indicator at certain moments (states) and a bad trading indicator at other moments. If these patterns would exist, then reinforcement learning seems to be a good tool to learn how to optimally behave in such an unstructured environment.

The reinforcement learning environment can be represented in the following scheme[5]:



The **agent**: is the learner and decision maker. In our case this is the investor who makes the buy, sell or hold decisions based on a policy.

The **environment** is everything outside the agent where he/she interacts with. In our case this is the market where Nymex WTI futures are traded.

---

[4] Reinforcement Learning: An Introduction. By Richard S.Sutton and Andrew G. Barto
[5] [5] Reinforcement Learning: An Introduction. By Richard S.Sutton and Andrew G. Barto

The environment can have different **states** after each time step. A state is the information available to the agent at a certain time. In our trading environment, we use a daily time step. The state at the end of each day is a tuple containing today's RSI value and the position (long, short, none) of the trader at the end of the day.

The agent can take **actions** at each time step based on the state that he/she is observing. In our training environment, the agent can take 3 possible actions (buy, sell, hold). The actions that an agent can take depend on his/her position at the time of action. Because of limited financial resources, the agent is only allowed to hold -1, 0 or +1 future contracts at a certain time. For example, if the agent holds 1 future contract, then he/she can only choose from 2 actions (hold or sell). This position constraint is also built in our Q-learning model.

The environment gives rise to **rewards** for a specific state-action combination. And it is these rewards that the agent tries to maximize over time. The reward in our case is the additional profit that a specific position generates over a 1 day period. For example, if the agent holds 1 WTI future contract in his portfolio, then the reward of holding that position for 1 more day will be equal to the daily return of WTI over that additional day.

The agent's **policy** $\pi(s)$ is the mapping from states to probabilities of selecting each possible action. It is a specific strategy to take high value actions that maximize our rewards over time. The function $\pi(s)$ evaluates all possible actions given the state *s* and returns the action with the highest value. In our trading environment, $\pi(s)$ will give for the current RSI(14) value and trading position the best action that will result in the highest expected future profit.

An **action-value function** *Q(s,a)* is a function (or a table) that accepts a state *s* and action *a* and returns the value of taking that action given that state. The value is the sum of the following 2 components:

- The immediate reward that you get for taking action *a* in state *s*
- The discounted reward, which is the expected reward you get for future actions

Reinforcement learning methods specify how the agent changes its policy over time because of its trial-and-error experience. The agent's goal is to find an optimal policy that optimizes the cumulative reward in the long run (not just immediate rewards). And reinforcement learning provides an algorithm to find that optimal policy.

Q-learning is a reinforcement learning algorithm used to calculate state-action values. In the simplest form, the action-value function *Q(s,a)* is a table containing the Q-values for every possible state-action pair. The table is iteratively updated as we play the game. The policy $\pi(s)$ is based on choosing the action a with the highest Q value for that given state ($\boldsymbol{\pi(s)} =$ **argmax$_a$(Q[s,a]**). If we run Q-learning long enough, we will eventually converge to the optimal policy $\boldsymbol{\pi} * (\boldsymbol{s})$ and optimal Q-table Q^*[s,a].

In our trading agent case this tabular **Q-learning algorithm** would involve the following steps:

1. Select training data: In our trading agent case, we use 75% of the data set for training the model. The other 25% is used to test the performance of the optimal policy.
2. Initialize the *Q(s,a)* table: for our trading agent case we would have to convert the values of the technical indicator to integers because our Q-table needs to store all possible state-action pairs. Continuous values would simply "blow up" the size of the Q-table. For a

state tuple containing the RSI value and position of the trader, we have 900 possible state-action combinations. We initialized the table by putting all values of the table to zero.

3. Repeat for several episodes:
    a. Repeat for number of days in training set:
        i. **For the first day**;
            1. we choose random state s_0 out of set of all possible states S and a random action a_0 out of set of all possible actions A. In the trading agent case we took s_0={RSI=0,position=0} and a_0={hold}.
            2. Out of s_0 and a_0 we can calculate the associated immediate reward r_0 and new state s_1.
            3. We update $Q(s_0,a_0)$ with value r_0.
        ii. **For the other days**:
            1. On every day t, we have an experience tuple $\{s_{t-1},a_{t-1},s_t,r_{t-1}\}$
            2. We will use this experience tuple to update the Q-table with the following **update rule**:

$Q'[s,a]=(1-\alpha) * Q[s,a] + \alpha * \text{improved estimate}$ where α is the learning rate. Alpha determines to what extend new information overrides old information in the learning process. The higher the value of α, the more you will use what you learnt in the last learning step. A typical value for alpha is 0.2. Instead of fixing the value of alpha over the entire training phase, one can also chose to decrease the value of alpha over time, such that there is a lot of learning in the beginning of the training phase and less when the model starts to converge to the optimal policy. We tried out the time decay on alpha, but didn't find evidence for improved learning of the model.

In the above equation, **$\text{improved estimate} = (r + \lambda \text{ later rewards})$,** where λ is the discount rate by which later rewards are discounted. Lambda determines the importance of future rewards. A low value of λ means that we value later rewards less. A typical value for λ is 0.9. Similar as for alpha, one can gradually increase the value of lambda during the learning process. This means that initially the model will mainly focus on immediate rewards. When the policy is converging at the end of the training process, the model will be more focusing on optimizing the long-term reward rather than immediate rewards. We tried out this technique for alpha on our model but didn't find any evidence of improved learning.

In the above equation:

later rewards = $Q[s',\text{argmax}_{a'}(Q[s',a'])]$, which is the value of the best action *a'* we can take given that we are in state *s'*. Here we need to mention the importance of the right trade-off between exploration and exploitation. If we exploit too much what we have learned,

then we will not try any new actions and stop learning. That's why we introduce a parameter ε which is the probability of choosing a random action instead of the best action a' in a training step. For high epsilon values the model will explore a lot of alternative actions, but exploit little of what it has learned so far. A typical value for ε is 0.2. Alternatively, we could also apply the same time decay for epsilon as explained for alpha. This means that in the initial steps of the training, the model will explore a lot of random actions (high epsilon) and learn a lot form those random actions (high alpha). When the model consolidates at the end of the training period, the model will explore less and exploit what it has been learning so far. We tried out this technique for epsilon on our model but didn't find any evidence of improved learning.

If we bring all the above elements together, then the update rule becomes:

$$Q'[s,a] =$$

$$(1-\alpha) * Q[s,a]$$

$$+ \alpha * (r + \lambda * Q[s', \text{argmax}_{a'}(Q[s',a'])])$$

In words this equation means:

The new Q-value for state s and action a is the old Q-value for state s and action a, multiplied by (1-α) plus α times our new best estimate, where our new best estimate is the sum of our immediate reward and the discounted reward for all our future actions.

Benchmark

In section 1 we explained how to calculate and interpret the Sharpe ratio. We use this metric to benchmark the performance of our Q-learning policies. The higher the Sharpe ratio, the better the model.

The Sharpe ratio is negative when the return of the investment is smaller than the risk-free rate. Usually, any Sharpe ratio greater than 1 is considered acceptable to good by investors. A ratio higher than 2 is rated as very good, and a ratio of 3 or higher is considered excellent[6]. After each training episode, we apply the Q-learning policy on the test data set to see how well it performs on "unknown" data. The resulting Sharpe ratio is plotted on a learning curve which indicates if there is convergence and how good the optimal Q-learning policy performs as a trading instrument.

[6] http://www.investopedia.com/ask/answers/010815/what-good-sharpe-ratio.asp

A positive Sharpe ratio would indicate that our optimal policy would result in an investment with a higher return than the risk-free rate. A Sharpe ratio above 1 would make us confident enough to use the policy for daily trading.

III. Methodology

Data Preprocessing

We used Reuters data which didn't require any preprocessing. We split this data set in a training set (75% of the data) and a testing set (25% of the data set). In this way, we can test the performance of the model on data that the model never saw during training.

In our trading learning environment, the state is defined by the tuple {RSI index, trading position}:

- RSI index is the Relative Strength Index. How to calculate the RSI was explained in section 2 above. We used 14 as value for N, as it is most commonly used. The RSI values can range from 0 to 100. We used the TA-lib library in Python to calculate the historical RSI values and compared these with the RSI values that Reuters publishes to make sure that our trading agent sees the same RSI values as the real traders. TA-lib's results are matching the Reuters values.
- The state variable trading position has a value of -1, 0 or +1, indicating if the trader is short 1 future, not holding any futures or long 1 future contract. The reason we limited the allowed trading positions to 3 possible states is to enter the notion of limited investment resources into the model.

There are 3 possible actions the agent can take: {buy,hold,sell}. For our first attempt, we applied the Q-learning algorithm using **Q-tables**. The algorithm is explained in section 2 above. 1 of the drawbacks of using Q-tables is that states can't have continuous values. Therefore, we needed to take the integer values of the RSI. The size of our Q-table is 900. If we would add a second technical indicator to our state, then the size of our Q-table should have to grow to above 200,000 data points.

Before we can start running the Q-learning algorithm, an empty Q-table needs to be built with a size of 900 data points. We also initialize the Q-table by entering zeros for each field in the table.

In a refinement step, we needed to add more state variables to improve the performance of our model. Increasing the number of state variables exponentially increases the size of our Q-table as explained above. Therefore, we needed to switch from a Q-table model to a **Q-function** model. This means that all state-action pairs and their corresponding q-value are saved as inputs for a machine learning algorithm that forecasts the q-value for a new state-action pair. The implementation of this refined model will be explained later in this paper.

Implementation

In this section, we will go through the main Python code that implements the Q-learning algorithm based on **Q-tables**. In the next section, we will go over a few improvements to the code.

- **Initialization step:**

As a first step, we need to prepare the necessary data and Q-table before we can start the Q-learning algorithm.

The role of the **class LearningAgent** is to initialize a Q-table and to create a state-action pair for the first training step. The Q-table contains 1 field for each possible state-action pair combination. In our case RSI can take an integer value between 0 and 100, there are 3 possible positions (-1,0-1) and 3 possible actions ('buy','sell','hold'). So, in total the Q-table needs space for 900 possible state-action pair combinations. For the first training step, we use an RSI value of 50, a position value of 0 and 'hold' as the first action. I chose these values because they are the most neutral, but in practice it doesn't really matter which values you pick because this is only 1 training step out of many thousands to follow.

```
class LearningAgent:
    def __init__(self):
        self.OldState={'RSI':50,'position':0}
        self.OldAction="hold"

    #Create a list of all the possible states
    def initStateSpace(self):
        states = []
        for RSIval in range(100):
            for position in range(-1,2):
                states.append((RSIval,position))
        return states

    #Create a dictionary (key-value pairs) of all possible state-actions and their values
    #This creates our Q-value look up table
    def initStateActions(self,states):
        av = {}
        for state in states:
            av[str((state,'buy'))] = 0.0
            av[str((state,'sell'))] = 0.0
            av[str((state,'hold'))] = 0.0
        return av
```

In the initialization step we also prepare the training data set and the test data set. In the code below we make the following steps to prepare this data:

1. Open the csv file 'WTI.csv', which contains the Reuters data with historical daily {open,high,low,close} prices for WTI front month futures contract and read the data into a data frame called 'data'.
2. We calculate the RSI(14) on this data set and add it to 'data'.
3. We slice off the first 14 elements of 'data', because they contain NaN values. This is because RSI(14) needs data up to 14 days back to calculate today's value.

4. We take the first 75% of 'data' as the training data set and the other 25% as the test data set.
5. The function reset() will create a LearningAgent object 'a' with an initialized Q-table and initialized state-action pair.

```
import pandas as pd
import TA
import RL
import random
import csv
import numpy as np
import math

LengthOfPeriod=365 #Lenght in number of days of 1 trial

data=pd.read_csv('WTI.csv')
data['RSI']=TA.RSI(data['Last'].as_matrix(),14,'ASC')
data=data[14:]
data['RSI_int']=data['RSI'].astype(int)
trainData=data.loc[:len(data)*0.75,:]
trainData= trainData.reset_index(drop=True)
trainData.to_csv('trainData.csv')
testData=data.loc[len(data)*0.75:,:]
testData= testData.reset_index(drop=True)
testData.to_csv('testData.csv')


def reset():
    a=RL.LearningAgent()
    return a
```

- **Learning step:**

The model is now ready for training. The **train2() function** is the core of the program and executes the following steps:

1. The learning rate, discount rate and epsilon parameter are made time variant. The learning rate and epsilon are decreasing from value 1 at the start of the training to value 0 at the end of the training. The discount rate is increasing from value 0 to value 1. We also made a train1() variant to apply fixed values for these parameters. As explained earlier, we didn't find any evidence of improved learning applying time decay. Another disadvantage of time decay is that the learning curve is more difficult to read because initially there will be a lot of exploration, making the learning curve very volatile at the start. To compare sensitivities to the model (impact of different state variables for example), you have to wait until the end of the training session before knowing the best sensitivity.
2. Next the Q-learning algorithm is applied in 3 distinct steps:
    a. The function **newState()** makes a new tuple {s,a,r,s'}, with s,a,r the old state, action and immediate reward and s' the new state for the next day.
    b. The function **BestActionReward()** choses the best action a' for this new state s'. First a random number between 0 and 1 is generated. If this random

number is less than epsilon, then the best action a' is randomly picked from the list of possible actions. Otherwise a' will be chosen as the action that results in the highest value in the Q-table for (s',a').

c. The function **update()** updates the Q-table with this new state-action pair (s'a'). In this function we apply the update rule **$Q'[s,a]=(1-\alpha) * Q[s,a] + \alpha * (r + \lambda * Q[s',\text{argmax}_{a'}(Q[s',a'])])$.**

3. Once we have trained the model for 365 days, we test its performance on the entire test data set and calculate the generated total profit and Sharpe ratio for this test run. The total profit and Sharpe ratio for each of the test runs are stored in the arrays profits[] and sharpe[] and give at the end of the training session a learning curve.
4. Steps 1-3 are repeated for several trials (for the training of our model 100,000 times). The learning curve will show if the numbers of trials were sufficient for convergence.

```
def train2(alpha,gamma,epsilon,trials,trainData,LengthOfPeriod):
    a=reset()
    profits=[]
    sharpe=[]
    counter=1
    for trial in range(trials):
        print(trial)
        for day in range(len(trainData)-1):
            #we make the learning rate, discount rate and exploite/explore parameters
            #variable over time
            maxepsilon=max((1-(float(counter/float(trials))))*epsilon,0)
            maxalpha=max((1-(float(counter/float(trials))))*alpha,0)
            maxgamma=max(((float(counter/float(trials))))*gamma,0)
            #the following 3 steps are the core of the Q-learning algorithm
            #in step 1, we make 1 time step and get into a new state (RSI, position)
            #in step 2, we chose the best action for this new state
            #in step 3, we update the Q-value table with this new state-action pair
            newState(a,day,trainData) #step1
            BestAction,MaxQ=BestActionReward(a,day,trainData,maxepsilon) #step2
            update(trainData,day,a,MaxQ,BestAction,maxalpha,maxgamma) #step3
        #In the next 3 lines of code, we apply the Q-learning model to the entire test data set.
        #We calculate the resulting profits and sharpe ration and append it to the arrays profits[] and sharpe[]
        #The array sharpe[] contains the sharpe ratio value for each trial step, and as such is our learning curve.
        PL,SR=predictFullPeriod(a,testData)
        profits.append(PL)
        sharpe.append(SR)
        counter=counter+1
    Results=pd.DataFrame(profits,columns=['profit'])
    Results['Sharpe']=sharpe
    pd.DataFrame(Results).to_csv('profits7(NN)WITH position limits during training_ADXR.csv')
    #We write the Q-table to a csv file for later use.
    with open('Qtable.csv','w') as csv_file:
        writer=csv.writer(csv_file)
        for key,value in a.Qtable.items():
            writer.writerow([key,value])
    return a
```

```
def newState(a,day,trainData):
    #calculates the new state for the agent based on the OldState
    oldRSI=a.OldState['RSI']
    oldPos=a.OldState['position']
    oldAction=a.OldAction
    if oldAction=='buy':
        newPos=oldPos+1
    if oldAction=='sell':
        newPos=oldPos-1
    if oldAction=='hold':
        newPos=oldPos+0
    newRSI=trainData['RSI_int'].iloc[day]
    a.NewState={'RSI':newRSI,'position':newPos}
    a.immediateReward=((float(trainData['Last'].iloc[day])/float(trainData['Last'].iloc[day-1]))-1)*newPos
```

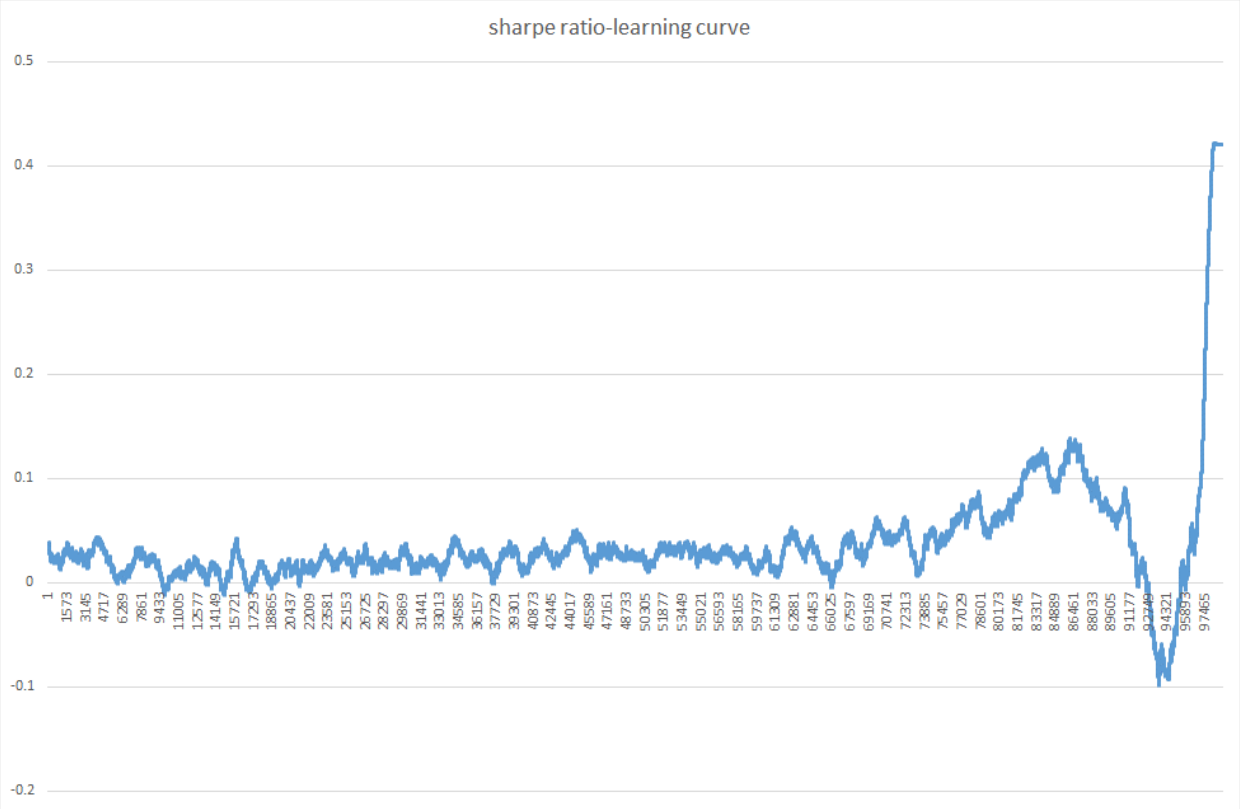
```
def BestActionReward(a,day,trainData,epsilon):
    #this function calculates the best action and reward for a give state
    if (random.random()<epsilon):
        if (a.NewState['position']==-1):
            BestAction=random.choice(['buy','hold'])
        if (a.NewState['position']==0):
            BestAction=random.choice(['buy','hold','sell'])
        if (a.NewState['position']==1):
            BestAction=random.choice(['hold','sell'])
        MaxQ=float(a.Qtable[str(((a.NewState['RSI'],a.NewState['position']),BestAction))])
    else:
        rewardArray=[]
        actions=[]
        if (a.NewState['position']==-1):
            actions=['buy','hold']
        if (a.NewState['position']==0):
            actions=['buy','hold','sell']
        if (a.NewState['position']==1):
            actions=['hold','sell']
        for action in actions:
            rewardArray.append(float(a.Qtable[str(((a.NewState['RSI'],a.NewState['position']),action))]))
        BestAction=actions[rewardArray.index(max(rewardArray))]
        MaxQ=max(rewardArray)
    a.NewAction=BestAction
    return BestAction,MaxQ
```

```
def update(trainData,day,a,MaxQ,BestAction,alpha,gamma):
    V=float(a.Qtable[str(((a.OldState['RSI'],a.OldState['position']),a.OldAction))])
    X=a.immediateReward+(gamma*MaxQ)
    a.Qtable[str(((a.OldState['RSI'],a.OldState['position']),a.OldAction))]=((1-alpha)*V)+(alpha*X)
    a.OldState=a.NewState
    a.OldAction=a.NewAction
```

- **Results:**

We trained the model for 100,000 steps with the training data set. After each training step, we tested the model with the entire test set. The series of daily profits for each test step generated a Sharpe ratio, which we could plot to visualize the progress of learning. At the end of the 100,000 training steps, we could plot a learning curve with 100,000 Sharpe ratio values. To make the graph more readable, we plotted the moving averages of the Sharpe ratios.

The next graph shows the learning curve over the entire training period of 100,000 steps. At the end of the training period the model able to achieve a Sharpe ratio of 0.42.



Refinement

The Q-learning algorithm explained in the previous section has been adapted on several points to make the following improvements:

- **Use of a Q-function instead of Q-table:** Instead of storing the values of the state-action pairs in a table, we now store all the state-action pairs in a data frame X and the corresponding Q-values in a data frame Y. Instead of looking up the value of a state-action pair in a Q-table, we now use a supervised learning algorithm that is trained with the X (inputs) and Y (target values) data frames. For this purpose we had to rework the **update()** function. In code **line 239** we use a trained supervised learning algorithm (stored in the variable clf) to predict the Q(s,a) value instead of a Q-table to look up Q(s,a). We used the available models from the sklearn library.
  In code **lines 242** we apply the same update rule as in the original update() function, but instead of looking up the value of Q(s,a) in the Q-table, we use the predictions made by the supervised learning algorithm in **lines 239-240**.
  Using ensemble learners as a Q-function seemed to give the best results.
- **Speed up learning, by only refitting the Q-function after a certain number of learning steps:** We can improve speed by reducing the number of times the supervised learning model **clf** is updated. In our code, we only re-fit the model every 300 time steps, which saves computation time and is not significantly slowing down the speed of learning.

```
229 def update(teller,trainData,day,a,MaxQ,alpha,gamma):
230     predictArr=[]
231     for key in a.StateActive:
232         if(a.StateActive[key]==1 and key != 'position'):
233             predictArr.append(a.OldState[key])
234     predictArr.append(a.OldState['position'])
235     predictArr.append(['buy','hold','sell'].index(a.OldAction))
236     #We apply the Q-function on the current state-action pair and predict
237     #its value and store it in the data frame Y. The corresponding state-action
238     #pair is stored in the data frame X.
239     V=clf.predict([predictArr])
240     X=a.immediateReward+(gamma*MaxQ)
241     a.X.append(predictArr)
242     a.Y.append((((1-alpha)*V)+(alpha*X))[0])
243     #In the next block of code we allow the dataframes X and Y to grow until they have
244     #reached the size of 1,000,000 rows. Once this size has been reached, we will take the last 1,000,000
245     #data points of X and Y
246     #We also speed up the code by only refitting the Q-function every 300 steps.
247     size=1000000
248     if len(a.Y)<size:
249         if math.fmod(teller,300)==0:
250             clf.fit(a.X,a.Y)
251             saveX_Y()
252     else:
253         a.X=a.X[-size:]
254         a.Y=a.Y[-size:]
255         if math.fmod(teller,300)==0:
256             a.X=a.X[-size:]
257             a.Y=a.Y[-size:]
258 ##            clf.fit(a.X[-size:],a.Y[-size:])
259             clf.fit(a.X,a.Y)
260             saveX_Y()
261     a.OldState=a.NewState
262     a.OldAction=a.NewAction
```

- **Use additional input variables to define the state:** In the initial model we used RSI(14) and the trader's position to define the state. In the refined model we used 2 additional features of the RSI(14) indicator:
    - **a.** The correlation between RSI(14) and the WTI price has been added as a state variable. At times of high correlation, the RSI indicator is more relevant to traders than in case of no correlation.
    - **b.** The slope of the RSI(14) curve: is the RSI(14) currently declining or increasing? This might give an indication whether a high RSI(14) value has reached its peak or a low RS(14) value has bottomed out.
- **Discretize the state variables to make the model converge faster:** It was clear that even a relative simple model with 4 state variables, as defined above, took a considerable amount of time to converge to the optimal policy. To speed up the learning process, we decided to discretize the state variables.
    - **a. RSI(14):** was divided by ten an rounded to the nearest integer, reducing the value range to [0,10] instead of continuous values from 0 to 100.
    - **b. Correlation between RSI(14) and WTI price:** This state variable could only take value zero for negative correlation and 1 for positive correlation.
    - **c. Slope of the RSI curve:** This state variable could also only take value zero for negative slope and value 1 for a positive slope.

  By discretizing the state variables, we noticed a much faster convergence to the optimal policy. On the other hand, we probably lose information by discretizing the state

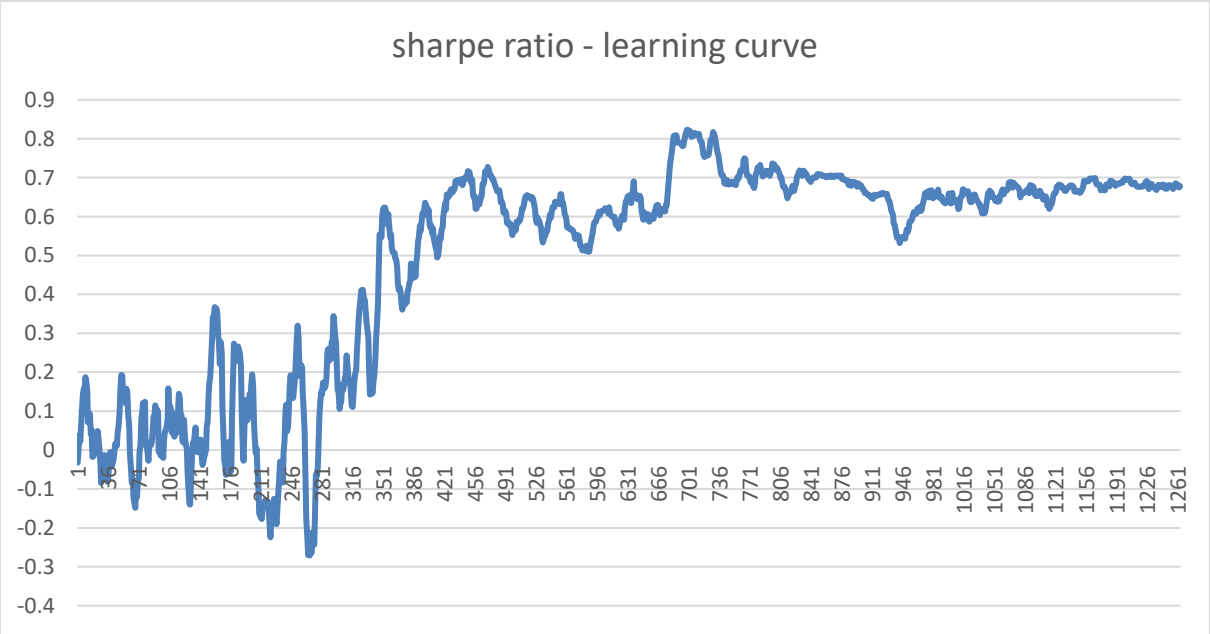
variables. So, a model with continuous state variables probably would have resulted in an optimal policy with higher Sharpe ratios, but would have taken much longer to converge.

- **Use several ensemble-learners as Q-functions and implement a voting process:** In our final model, we used 5 ensemble learners as Q-functions:
    1. **RandomForestRegressor**
    2. **ExtraTreesRegressor**
    3. **GradientBoostingRegressor**
    4. **AdaBoostRegressor**
    5. **BaggingRegressor**

  Instead of using 1 Q-function to find the optimal action in a learning step, we used 5 ensemble learners. The action with the most votes would be the best action a' for the current learning step.

- **<u>Results:</u>**

  The next graph shows the learning curve for the improved model.
  Thanks to the improvements, the optimal policy is now capable to generate a Sharpe ratio of 0.68 versus 0.42 for the initial model.



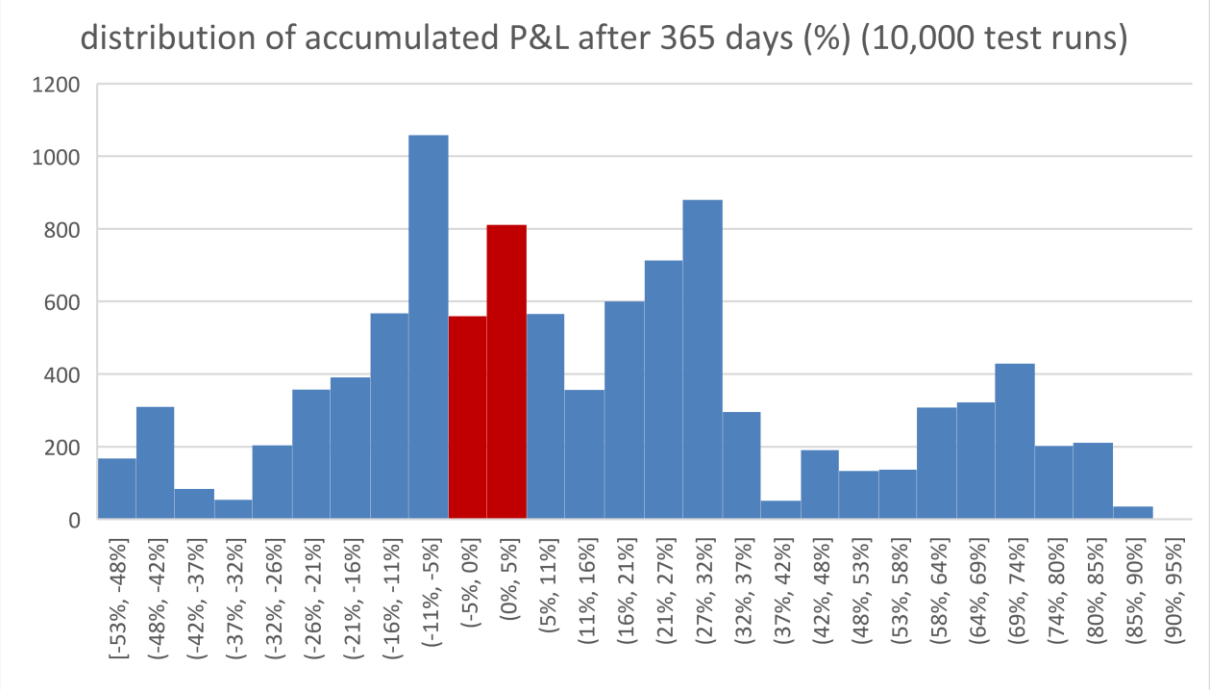
V. Conclusion

Free-Form Visualization

The **initial model** was further tested on the test set in the following way:

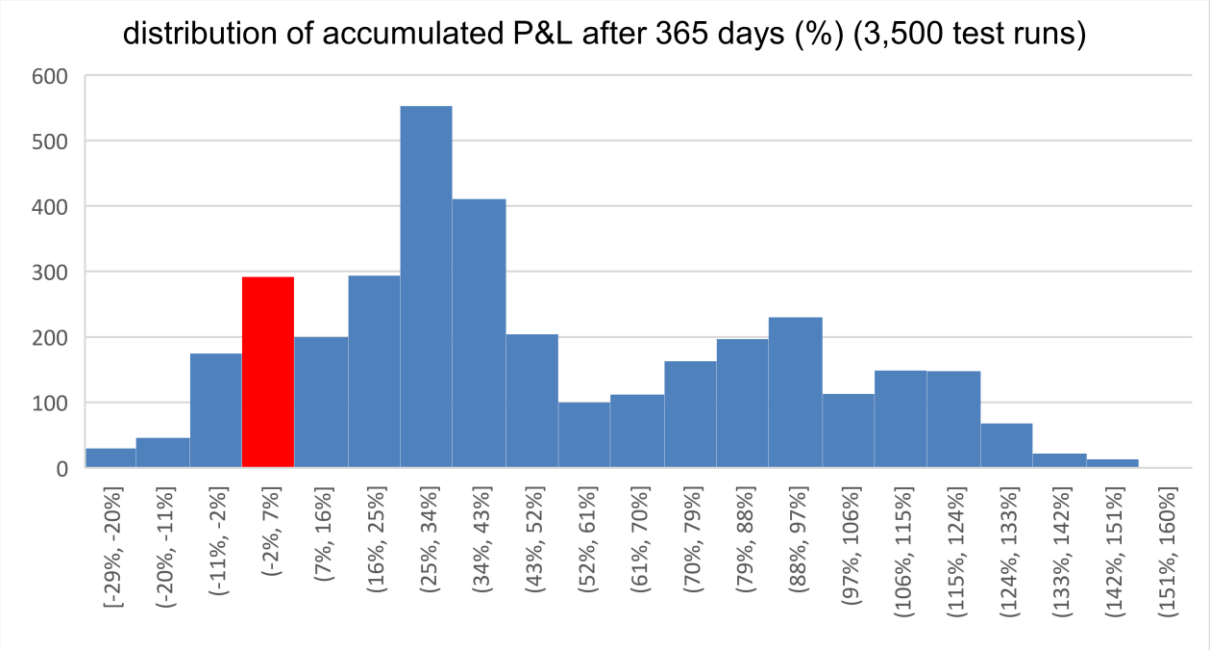
- We took a random subset of 365 consecutive test data points out of the test data set.
- We applied the trained model on this subset and calculated the cumulative profit/loss at the end of this 365 days trading session.
- We repeated these steps 10,000 times and plotted the histogram of the P&L's.
- It showed that 62.5% of these 10,000 training sessions were profitable and that the average 365 days' return was 14%.

The histogram below shows the distribution of the 365 day returns (P&L's) for each of the 10,000 test sessions. The histogram shows that the blue surface to the right of the red (break-even) bars is larger than to the left. So, using the model would give you a higher chance (62.5%) of making a profit than making a loss. On the other hand, a rather low Sharpe ratio of 0.42 indicates that the investor needs to take a considerable risk for these profits.



We applied the same test to the **refined model**. In this case we simulated 3,500 trading sessions of 365 days each. 91% of these resulted in a positive cumulated profit at the end of the 365-day trading period (versus 62% of the initial model). The average return was 49% (versus 14% for the initial model).
When we look at the histogram, then we can clearly notice that the histogram of the improved model is much more skewed to the right: the maximum losses for the improved model are much

lower than for the initial model (-29% versus -53%) and the maximum profits of the improved model are much higher than for the initial model (151% versus 90%). In other words, the improved model reduces risk and increases expected profit which is reflected in a higher Sharpe ratio.



Reflection

In this project our goal was to test if Q-learning can be used to support investment decisions. Our case assumed a WTI trader using technical analysis. We selected 1 of the most popular technical indicators (RSI).

We first plotted the daily WTI returns against the RSI values in a scatter plot to see if high RSI values go together with low returns and vice versa. This was not obvious from this chart. This could mean either:

- RSI has no predictive power and is not a suitable instrument for trading.
- Using RSI is much more complex than just buying or selling above or below certain threshold RSI values. To use the RSI indicator successfully, a complex underlying process (policy) should be learned through experience.

To see which of the above 2 conclusions apply, we used Q-learning. The way a trader learns how to use the RSI indicator is very similar to the way Q-learning learns the optimal policy: trial and error and remembering what action worked best in a particular situation. The Q-learning algorithm also assumes no prior knowledge about the underlying process it tries to learn. Only a list of allowed actions and their associated rewards are needed.

We used the Sharpe ratio as our benchmark metric to compare the performance of different models. The Sharpe ratio is a widely used metric measuring both profitability and risk of an investment strategy. In general, a Sharpe ratio above 1 is considered as a good investment strategy.

First the Q-learning algorithm was applied using Q-tables where the expected value for each state/action pair is stored in a lookup table. Our model could learn and produce an optimal policy with positive Sharpe ratios. But some limitations of the Q-table approach became also clear. To produce higher Sharpe ratios, we needed to add more state variables to define our environment better. By increasing the number of state variables, our Q-table and computation time would exponentially increase.

Using Q-functions instead of Q-tables would solve this problem. It would also make the use of continuous state variables or integer state variables with a large range possible. We expanded the set of state variables by adding 2 additional variables. The choice of the 2 additional variables was based on the way a trader could use the RSI indicator. Probably he would only use RSI during times where it was obvious that other market participants were also using this indicator as a trading tool. The correlation between RSI and future price returns would give such evidence. Also, a high RSI value could be interpreted different when RSI is trending upward versus trending downward. So, the slope of the RSI curve would also help the trader to put the RSI value in the right context.
Finally, instead of using 1 Q-function to learn the optimal policy, we applied 5 Q-functions. Each of the 5 Q-functions would predict the best action the trader should take for each learning step. The action with the most votes would be picked by the model.
The final model showed improved learning behavior versus the initial model. On the other hand, we didn't succeed in producing a policy with Sharpe ratio above 1.

Personally, I am very pleased and surprised by the results of the model. A Sharpe ratio of 0.68 is clearly not meeting the standards of the industry. But there are still a lot of tweaking options left to further improve the model, which makes me think that a Sharpe ratio above 1 is within reach.

A lot of improvements to the model were based on trial and error, and this is the major challenge. It takes a lot of computing time before any learning behavior becomes apparent. Tweaking parameters through brute force is simply impossible and that's where domain knowledge becomes important to pick only the most promising trials.

Improvement

There are several interesting options we can explore to further improve our model and by doing this I am confident that the resulting model can produce a tool that can support the day to day trading decisions of an investor.
This is a list of improvement ideas:

- Expand the state tuple with additional technical indicators. For each individual indicator, we could first train a model with only this indicator defining the state tuple. The most promising indicators would then be joined in an improved model. The idea is the more these indicators agree, the stronger the buy or sell recommendation of the model.
- When we selected the Q-function, we didn't tweak the parameters of the supervised learning algorithms. We just used the default parameter settings from sklearn. Tweaking the Q-functions will probably improve the results.

- We could also include other types of state variables instead of only technical indicators. These could be fundamental market data, or outputs from other machine learning algorithms predicting the current state of the market.
- We could also try to train a model with intra-day prices instead of daily prices. Most high frequency funds are using automated trading engines driven by mainly technical inputs. So chances are high that the same approach applied on minute by minute data will give better results as using daily time steps.

All the above improvement ideas would most probably result in a reliable trading system, but will also require a lot of computing power. A run of 2000 training steps takes several hours or even days to complete. Running all the possible permutations of the above improvement ideas is simply not possible on a CPU.

2 possible solutions could deal with this problem:

1. Apache Spark: allows to spread the workload over different nodes in a cluster. Spark has a limited library of machine learning algorithms and an interface to Python (PySpark). This approach would limit the amount of recoding needed.
2. CUDA: this will be probably the best solution to our problem, but would require that most of the code of the original model would need to be reprogrammed to make it suitable for a parallel world. Python has also a CUDA interface (Pycuda) which allows to integrate CUDA kernels in Python code. In this way, we only need to run the time critical parts of the model on the GPU. The less time critical steps can still be done on the CPU.